ADVANCED DATA-STRUCTURES
&
ALGORITHM ANALYSIS

Dr. Sukhamay Kundu

Computer Science Dept, Louisiana state University
Baton Rouge, LA 70803

kundu@csc.lsu.edu

Spring 2011

ROLE OF DATA-STRUCTURES IN COMPUTATION

Makes Computations Faster:

- Faster is better. (Another way to make computations faster is to use parallel or distributed computation.)

Three Basic Computation Steps:

Computation = Sequence of Computation Steps

External Input →
(1) Locate/Access data-values (inputs to a step)
(2) Compute a value (output of a step)
(3) Store the new value
→ External Output

Program: Algorithm + DataStructure + Implementation.

- *Algorithm*
  - The basic method; it determines the data-items computed.
  - Also, the order in which those data-items are computed (and hence the order of read/write data-access operations).
- *Data structures*
  - Supports efficient read/write of data-items used/computed.

Total Time = Time to access/store data + Time to compute data.

Efficient Algorithm = Good method + Good data-structures (+ Good Implementation)

Question:

•? What is an efficient program?

•? What determines the speed of an Algorithm?

•? A program must also solve a "problem". Which of the three parts algorithm, data-structure, and implementation embodies this?

ALGORITHM OR METHOD vs. DATA STRUCTURE

Problem: Compute the average of three numbers.

Two Methods: (1) aver = $(x + y + z)/3$.
(2) aver = $(x/3) + (y/3) + (z/3)$.

- Method (1) superior to Method (2); two less div-operations.
- They access data in the same order: $\langle x, y, z, \text{aver}\rangle$.
- Any improvement due to data-structure applies equally well to both methods.

Data structures:

(a) Three variables x, y, z.

(b) An array nums[0..2].

- This is inferior to (a) because accessing an array-item takes more time than accessing a simple variable. (To access nums[i], the executable code has to compute its address addr(nums[i]) = addr(nums[0]) + *i**sizeof(int), which involves 1 addition and 1 multiplication.)
- When there are large number of data-items, naming individual data-items is not practical.
- Use of individually named data-items is not suitable when a varying number of data-items are involved (in particular, if they are used as parameters to a function).

A Poor Implementation of (1): Using 3 additions and 1 division.

```
a = x + y; //uses 2 additional assignments
b = a + z;
aver = b/3;
```

LIMITS OF EFFICIENCY

Hardware limit:

- *Physical* limits of time (speed of electrons) and space (layout of circuits). This limit is computation problem *independent*.

  From 5 mips (millions of instructions per sec) to 10 mips is an improvement by the factor of 2.

  One nano-second = 10^{-9} (one billionth of a second); 10 mips = 100 ns/instruction.

Software limit:

- *Limitless* in a way, except for the inherent nature of the problem. That is, the limit is *problem dependent*.

Sorting Algorithm A1:	$O(n.\log n)$ time
Sorting Algorithm A2:	$O(n^2)$ time

(n = number of items sorted)

A1 is an improvement over A2 by the factor

$$\frac{n^2}{n.\log n} = \frac{n}{\log n} = \rightarrow \infty \text{ as } n \rightarrow \infty.$$

- $O(n.\log n)$ is the efficiency-limit for sorting Algorithms.

MEASURING PERFORMANCE

Analytic Method:

- Theoretical analysis of the Algorithm’s time complexity.

Empirical Methods:

- Count the number of times specific operations are performed by executing an *instrumented* version of the program.
- Measure directly the actual program-execution time in a run.

Example of Instrumentation:

Original code:

```
if (x < y) small = x;
else small = y;
```

Instrumentd code:

```
countComparisons++; //initialized elsewhere
if (x < y) small = x;
else small = y;
```

Question:

•? What is wrong with the following instrumentation:

```
if (x < y) { countComparisons++; small = x; }
else small = y;
```

•? Instrument the code below for readCount and writeCount of *x*:

```
if (x < 3) y = x + 5;
```

•? Show the new code when updates to loopCount is moved outside the loop:

```
for (i=j; i<max; i++) {
    loopCount++;
    if (x[i] < 0) break;
}
```

EXERCISE

1. Instrument the code below to count the number of Exchanges (numExchanges) and number of comparisons (numComparisons) of the array data-items. Show the values of numExchanges and numComparisons after each iteration of the outer for-loop for the input items[] = [3, 2, 4, 5, 2, 0].

```
void crazySort(int *items, int numItems)
{ int i, j, small,
  for (i=0; i<numItems; i++) //put ith smallest item in items[i]
      for (j=i+1; j<numItems; j++)
          if (items[i] > items[j]) { //exchange
             small = items[j]; items[j] = items[i];
             items[i] = small;
          }
}
```

   (a) If we use "i < numItems − 1" in place of "i < numItems" in the outer for-loop, do we still get the same final result? Will it affect the execution time?

   (b) Is the algorithm in the code more closely related to insertion-sort or to selection-sort? In what way does it differ from that?

2. For numItems = 6, find an input for which crazySort will give maximum numExchanges. When will numExchanges be minimum?

3. Give a pseudocode for deciding whether three given line segments of lengths x, y, and z can form a triangle, and if so whether it is a right-angled, obtuse-angled, or an acute-angled triangle. Make sure that you minimize the total number operations (arithmetic and comparisons of data-items)?

4. Given an array lengths[1..n] of the lengths of n line segments, find a method for testing if they can form a polygon (quadrilateral for n = 4, pentagon for n = 5, etc).

SOLUTION TO SELECTED EXERCISES:

1.
```
void crazySort(int *items, int numItems)
{ int i, j, small,
      numComparisons=0, //for two elements in items[]
      numExchanges=0;   //of elements in items[]
  for (i=0; i<numItems; i++) {//put ith smallest item in items[i]
      for (j=i+1; j<numItems; j++) {
          numComparisons++;     //keep it here
          if (items[i] > items[j]) { //exchange
             numExchanges++;
             small = items[j]; items[j] = items[i];
             items[i] = small;
          }
      }
      printf("numComparisons = %d, numExchanges = %d\n",
             numComparisons, numExchanges);
  }
}
```

After the comparison and exchanges (if any) for input items[] = [3, 2, 4, 5, 2, 0].

```
i=0, j=1, items[]: 2 3 4 5 2 0
i=0, j=2, items[]: 2 3 4 5 2 0
i=0, j=3, items[]: 2 3 4 5 2 0
i=0, j=4, items[]: 2 3 4 5 2 0
i=0, j=5, items[]: 0 3 4 5 2 2
numComparisons = 5, numExchanges = 2
i=1, j=2, items[]: 0 3 4 5 2 2
i=1, j=3, items[]: 0 3 4 5 2 2
i=1, j=4, items[]: 0 2 4 5 3 2
i=1, j=5, items[]: 0 2 4 5 3 2
numComparisons = 9, numExchanges = 3
i=2, j=3, items[]: 0 2 4 5 3 2
i=2, j=4, items[]: 0 2 3 5 4 2
i=2, j=5, items[]: 0 2 2 5 4 3
numComparisons = 12, numExchanges = 5
i=3, j=4, items[]: 0 2 2 4 5 3
i=3, j=5, items[]: 0 2 2 3 5 4
numComparisons = 14, numExchanges = 7
i=4, j=5, items[]: 0 2 2 3 4 5
numComparisons = 15, numExchanges = 8
i=5, j=6, items[]: 0 2 2 3 4 5
numComparisons = 15, numExchanges = 8
```

This is more closely related to selection-sort, which involves at most one exchange for each iteration of outer-loop. #(Comparisons) is still C_2^n.

2. Triangle classification pseudocode; assume that $0 < x \leq y \leq z$.

```
if (z < x + y) {
   zSquare = z*z; xySquareSum = x*x + y*y;
   if (zSquare == xySquareSum)
      right-angled triangle;
   else if (zSquare > xySquareSum)
           obtuse-angled triangle;
   else acute-angled triangle;
}
else not a triangle;
```

3. Condition for polygon:
   - The largest length is less than the sum of the other lengths.
   - The lengths [2, 4, 5, 20] will not make a quadrilateral because $20 \nless 2 + 4 + 5 = 11$, but the lengths [2, 4, 5, 10] will.

ANALYZING NUMBER OF EXCHANGES IN CRAZY-SORT

Pseudocode #1:

1. Create all possible permutations p of $\{0, 1, 2, \cdots, n-1\}$.
2. For each p, apply crazySort and determine numExchanges.
3. Collect these data to determine numPermutations[i] = #(permutations which has numExchanges = i) for $i = 0, 2, \cdots, C^n_2$.
4. Plot numPermutations[i] against i to visualize the behavior of numExchanges.

Pseudocode #2: //No need to store all n! permutations.

1. For (i=0; $i<C^n_2$; i++), initialize numPermutations[i] = 0.
2. While (there is a nextPermutation(n) = p) do the following:
   (a) Apply crazySort to p and determine numExchagnes.
   (b) Add 1 to numPermutation[numExchanges].
3. Plot numPermutations[i] against i.

Note: We can use this idea to analyze other sorting algorithms.

Question:

•? If p is a permutation of $S = \{0, 1, 2, \cdots, n-1\}$, then how to determine the nextPermutation(p) in the lexicographic order? Shown below are permutations for n = 4 in lexicographic order.

	0123		0312		1203		2013		2301		3102
↓	0132	↓	0321	↓	1230	↓	2031	↓	2310	↓	3120
	0213		1023		1302		2103		3012		3201
	0231		1032		1320		2130		3021		3210

PSEUDOCODE vs. CODE

Characteristics of Good Pseudocode:

+ Shows the key concepts and the key computation steps of the Algorithm, avoiding too much details.
+ Avoids dependency on any specific prog. language.
+ Allows determining the correctness of the Algorithm.
+ Allows choosing a suitable data-structures for an efficient implementation and complexity analysis.

Example. Compute the number of positive and negative items in $nums[0..n-1]$; assume each $nums[i] \neq 0$.

(A) Pseudocode:

1. Initialize positiveCount = negativeCount = 0.
2. Use each $nums[i]$ to increment one of the counts by one.

Code:

```
1.1 positiveCount = negativeCount = 0;
2.1 for (i=0; i<n; i++) //each nums[i] ≠ 0
2.2     if (0 < nums[i]) positiveCount++;
2.3     else negativeCount++;
```

(B) Pseudocode:

1. Initialize positiveCount = 0.
2. Use each $nums[i] > 0$ to increment positiveCount by one.
3. Let negativeCount = n − positiveCount.

Code:

```
1. positiveCount = 0;
2. for (i=0; i<n; i++) //each nums[i] ≠ 0
3.     if (0 < nums[i]) positiveCount++;
4. negativeCount = n - positiveCount;
```

Question:

•? Why is (B) slightly more efficient than (A)?

> Writing a pseudocode requires skills to express an Algorithm in a concise and yet clear fashion.

PSEUDOCODE FOR SELECTION-SORT

Idea: Successively find the ith smallest item, $i = 0, 1, \cdots$.

Algorithm Selection-Sort:

Input: Array items[] and its size numItems.
Output: Array items[] sorted in increasing order.

1. For each i in { 0, 1, ⋯, numItems-1}, in some order, do (a)-(b):
   (a) Find the ith smallest item in items[].
   (b) Place it at position i in items[].

Finding ith smallest item in items[]:

- Finding ith smallest item directly is difficult, but it is easy if we know all the kth smallest items for $k = 0, 1, 2, \cdots, (i - 1)$.
- It is the smallest item among the remaining items.
- If we assume that items[k], $0 \le k \le (i - 1)$, are the kth smallest items, then smallest item in items[$i..numItems - 1$] = ith smallest item. This gives the pseudocode:

```
(a.1)   smallestItemIndex = i;
(a.2)   for (j = i + 1; j<numItems; j++)
(a.3)       if (items[j] < items[smallestItemIndex])
(a.4)       then smallestItemIndex = j;
```

Question: In what way (a.1)-(a.4) is better than step (a)?

Placing ith smallest item at position i in items[].

```
(b.1)   if (smallestItemIndex > i) // why not smallestItemIndex ≠ i
(b.2)   then exchange items[i] and items[smallestItemIndex];
```

"What" comes before "how".

EXERCISE

1. Which of "put the items in right places" and "fill the places by right items" best describes the selection-sort Algorithm? Shown below are the steps in the two methods for input [3, 5, 0, 2, 4, 1].

	Put the items in right places	Fill the places with right items
1.	[2, 5, 0, **3**, 4, 1] 3 moved to right place	[**0**, 5, 3, 2, 4, 1] 1st place is filled by 0
2.	[0, 5, **2, 3**, 4, 1] 2 moved to right place	[**0, 1**, 3, 2, 4, 5] 2nd place is filled by 1
3.	[**0**, 5, **2, 3**, 4, 1] 0 already in right place	[**0, 1, 2**, 3, 4, 5] 3rd place is filled by 2
4.	[**0**, 1, **2, 3**, 4, **5**] 5 moved to right place	[**0, 1, 2, 3, 4, 5**] all places filled properly
5.	[**0, 1, 2, 3, 4, 5**] all items in right places	

   Note that once an item is put in right place, you must not change its position while putting other items in proper places. It is for this reason, we make an exchange (and not an insertion) when we move an item in the right place. The insertion after removing 3 from its current position in [3, 5, 0, 2, 4, 1] would have given [5, 0, 2, 3, 4, 1] but not [2, 5, 0, 3, 4, 1] as we showed above.

2. Which input array for the set numbers {0, 1, 2, 3, 4, 5} requires maximum number of exchanges in the first approach?

3. Give a pseudocode for the first approach.

ANOTHER EXAMPLE OF PSEUDOCODE

Problem: Find the position of rightmost "00" in binString[0..(n-1)].

1. Search for 0 right to left upto position 1 (initially, start at position n-1).
2. If (0 is found and the item to its left is 1), then go back to step (1) to start the search for 0 from the left of the current position.

Three Implementations: Only the first one fits the pseudocode.

```
(1) i = n; //= length of binString
    do { for (i=i-1 ; i>0; i--)
                  if (0 == binString[i]) break;
    } while (1 == binString[--i]); //has a bug; find it

(2) for (i=n-1; i>0; i--)
        if (0 == binString[i]) && (0 == binString[i-1])
           break; //inefficient but works

(3) for (i=n-1; i>0; i--) //bad for-loop; body updates i
        if (0 == binString[i]) && (0 == binString[--i])
           break; // works and efficient
```

Question:

- ? Show how these implementations work differently using the binString: $\cdots$000111010101. Extend each implementation to return the position of the left 0 of the rightmost "00".
- ? Instrument each code for readCount of the items in binString[ ].
- ? Which of (1)-(3) is the least efficient in terms readCount?
- ? Give a pseudocode to find rightmost "00" without checking all bits from right till "00" is found.

It is not necessary to sacrifice clarity for the sake of efficiency.

EXERCISE

1. BinStrings(n, m) = {x: x is a binary string of length n and m ones}, $0 \le m \le n$. The strings in BinStrings(4, 2) in lexicographic order are:

   0011, 0101, 0110, 1001, 1010, 1100.

   Which of the pseudocodes below for generating the strings in BinStrings(n, m) in lexicographic order is more efficient?

   (a) 1. Generate and save all binary strings of length n in lexicographic order.
   2. Throw away the strings which have numOnes $\neq m$.

   (b) 1. Generate the first binary string $0^{n-m}1^m \in$ BinStrings(n, m).
   2. Successively create the next string in BinStrings(n, m) until the last string 1^m0^{n-m}.

   Which of the three characteristics of a good pseudocode hold for each of these pseudocodes?

2. Give the pseudocode of a recursive Algorithm for generating the binary strings in BinStrings(n, m) in lexicographic order.

3. Give an efficient pseudocode for finding the position of rightmost "01" in an arbitrary string $x \in$ BinStrings(n, m). (The underlined portion in 10110<u>01</u>1100 shows the rightmost "01".) Give enough details so that one can determine the number of times various items $x[i]$ in the array x are looked at.

4. Given a string $x \in$ BinStrings(n, m), give a pseudocode for generating the next string in BinStrings(n, m), if any.

ALWAYS TEST YOUR METHOD AND YOUR ALGORITHM

- Create a few general examples of input and the corresponding outputs.
  - Select some input-output pairs based on your understanding of the problem and *before* you design the Algorithm.
  - Select some other input-output pairs *after* you design the Algorithm, including a few cases that involve special handling of the input or output.
- Use these input-output pairs for testing (but not proving) the correctness of your Algorithm.
- Illustrate the use of data-structures by showing the "state" of the data-structures (lists, trees, etc.) at various stages in the Algorithm's execution for some of the example inputs.

Always use one or more carefully selected example to illustrate the critical steps in your method/algorithm.

EFFICIENCY OF NESTED IF-THEN-ELSE

- Let E = average #(condition evaluations). We count 1 for evaluation of both x and its negation ($\neg x$).

Example 1. For the code below, E = 3·5.

```
if (x and y) z = 0;
else if ((not x) and y) z = 1;
else if (x and (not y)) z = 2;
else z = 3;
```

Value of z	#(condition evaluations)
0	2 ($x = T$ and $y = T$)
1	3 ($x = F$, $\neg x = T$, and $y = T$)
2	5 ($x = T$, $y = F$, $\neg x = F$, $x = T$, and $\neg y = T$)
3	4 ($x = F$, $\neg x = T$, $y = F$, $x = F$)

Question:

•? Show #(condition evaluations) for each z for the code and also the average E:

```
if (x)
   if (y) z = 0;
   else z = 2;
else if (y) z = 1;
     else z = 3;
```

•? Give a code to compute z without using the keyword "else" (or "case") and show #(condition evaluations) for each value of z.

•? Show the improved form of the two code-segments below.

(a). `if (nums[i] >= max) max = nums[i];`

(b).
```
if (x > 0) z = 1;
    if ((x > 0) && (y > 0)) z = 2;
```

BRIEF REVIEW OF SORTING

Questions:

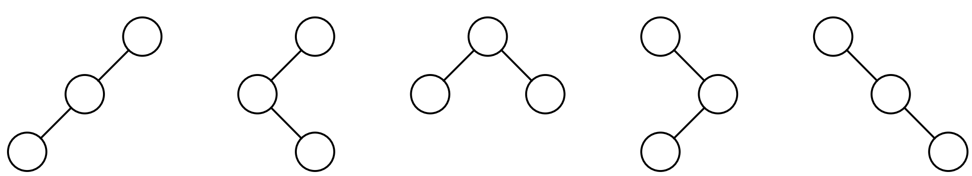
- What is Sorting? Explain with an example.
- Why do we want to sort data?
- What are some well-known sorting Algorithms?
- Which sorting Algorithm uses the following idea:

  > Successively, find the smallest item, the second smallest item, the third smallest items, etc.

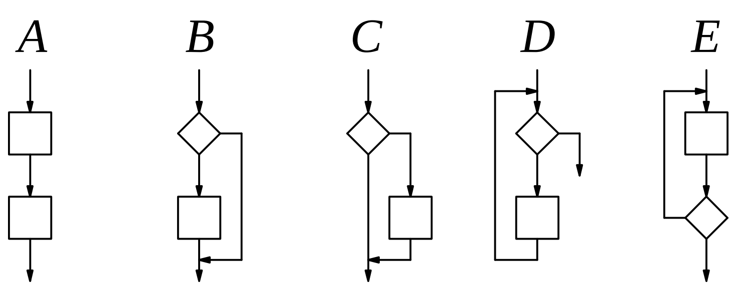
- Can we sort a set of pairs of numbers like {(1,7), (2,7), (5,4), (3,6)}? What is the result after sorting?
- Can we sort non-numerical objects like the ones shown below?

Strings: *abb*, *ba*, *baca*, *cab*.

Binary trees on 3 nodes (convert them to strings to sort):



Flowcharts with 2 nodes (convert them to trees or strings to sort):



EXERCISE

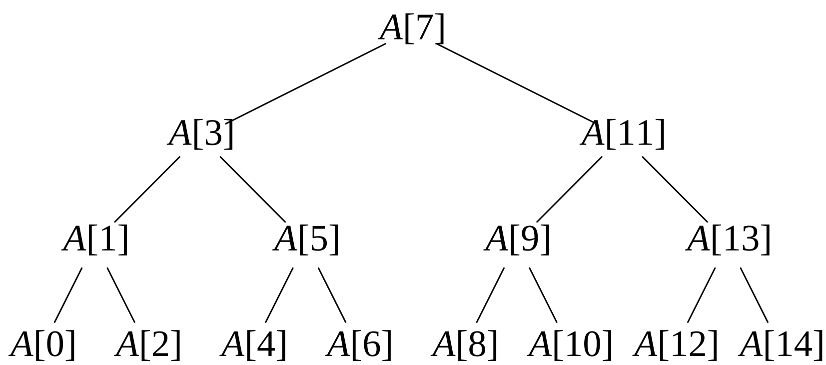
1. Give a more detailed pseudocode (not code) for sorting using the idea "put the items in the right places". Determine the number of comparisons of involving data from items[0..numItems-1] based on the pseudocode. Explain the Algorithm in detail for the input items[] = [3, 2, 4, 5, 1, 0].
2. Write a pseudocode for insertion-sort. Determine the number of comparisons of involving data from items[0..numItems-1] based on the pseudocode; also determine the number of data-movements (i.e., movements of items from the items-array) based on the pseudocode. Explain the Algorithm in detail for the input items[] = [3, 2, 4, 5, 1, 0].
3. For each of the sorting Algorithms insertion-sort, selection-sort, bubble-sort, and merge-sort, show the array after each successive exchange operation starting the initial array [3, 2, 4, 5, 1, 0].
4. Some critical thinking questions on selection-sort. Assume that the input is a permutation of $\{1, 2, \cdots, n\}$.
   (a) Give an example input for which the number of data-movements is maximum (resp., minimum).
   (b) In what sense, selection-sort minimizes data-movements?
   (c) Suppose we have exchanges of the form e_1: items[i1] and items[i2], e_2: items[i2] and items[i3], ... , e_{k-1}: items[i(k-1)] and items[ik]. Then argue that the indices {i1, i2, ..., ik} form a cycle in the permutation. Note that the exchange operations e_i may be interleaved with other exchanges.
5. Is it true that in bubble-sort if an item moves up, then it never moves down? Explain with the input items[] = [3, 2, 4, 5, 1, 0].

AVERAGE #(COMPARISONS) TO LOCATE A DATA-ITEM IN A SORTED-ARRAY

Binary Search: Assume N = numItems = 15 = $2^4 - 1$.

$A[0] < A[1] < A[2] < \cdots < A[14]$

#(Nodes at this level	#(Compar. per node)
1	1
2	2
4	3
8	4



- Number of comparisons for an item x:

  If x were $A[6]$, then we would make 4 comparisons:
  $x < A[7]$, $x > A[3]$, $x > A[5]$, and $x = A[6]$.

Total #(Comparisons) = 1×1 + 2×2 + 3×4 + 4×8 = 49;
Average = 49/15 = 3·3.

- General case ($N = 2^n - 1$): Total #(Comparisons) =

$$\sum_{i=0}^{n-1} \text{\#(compar. per node at level i)} \times \text{\#(nodes at level i)}$$
$$= 1\times1 + 2\times2 + 3\times4 + \cdots + n\times2^{n-1} = 1 + (n-1)2^n$$
$$= 1 + [log(N+1) - 1].\,(N+1) = O(N.\,log\ N)$$

Average #(Comp.) = $O(log\ N)$

A simpler argument:

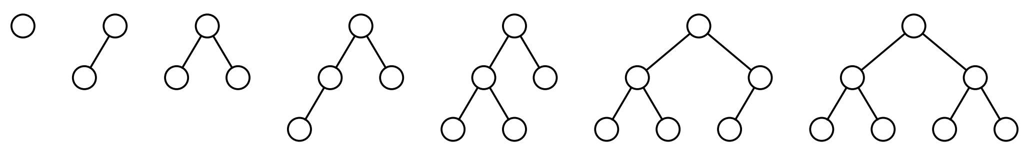
- Max(#Comp) = n and hence average ≤ $n = O(log\ N)$.

HEAP DATA-STRUCTURE

Heap: A special kind of binary-tree, which gives an efficient $O(N.\log N)$ implementation of selection-sort.

- *Shape constraints:* Nodes are added left to right, level by level.
  - A node has a rightchild only if it has a leftchild.
  - If there is a node at level m, then there are no missing nodes at level $m-1$.
- *Node-Value constraint:* For each node x and its children y, val(x) $\geq$ val(y), val(x) = the value associated with node x.

Example: The shape of heaps with upto 7 nodes.



Questions: Which of the following is true?

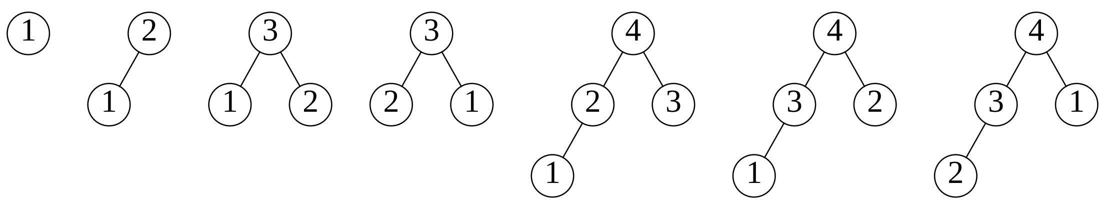
(1) Each node has exactly one parent, except the root.

(2) Each node has 0 or 2 children, except perhaps one.

(3) The leftchild node with no brother has the maximum height.

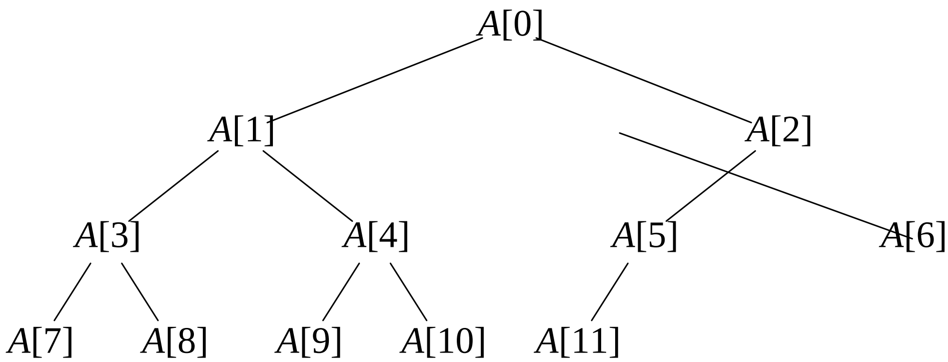
(4) The properties (1)-(3) define a heap.

Example. Heaps with upto 4 nodes and small node-values.



ARRAY-IMPLEMENTATION OF HEAP

Array-structure for Heap of 12 nodes:



- $A[3] \geq A[7]$, $A[8]$
- $A[4] \geq A[9]$, $A[10]$
- $A[5] \geq A[11]$
- $A[1] \geq A[3]$, $A[4]$
- $A[2] \geq A[5]$, $A[6]$
- $A[0] \geq A[1]$, $A[2]$

$$A[0] = \max\{A[0], A[1], \cdots, A[11]\}$$
$$A[1] = \max\{A[2], A[3], A[5], A[6], A[11]\}$$
$$\ldots$$

Parent-Child relations in the Array:

- Not dependent on values at the nodes and does not use pointers.

$$\begin{aligned} \text{leftchild of } A[i] &= A[2i+1] \\ \text{rightchild of } A[i] &= A[2i+2] \end{aligned}$$

EXERCISE

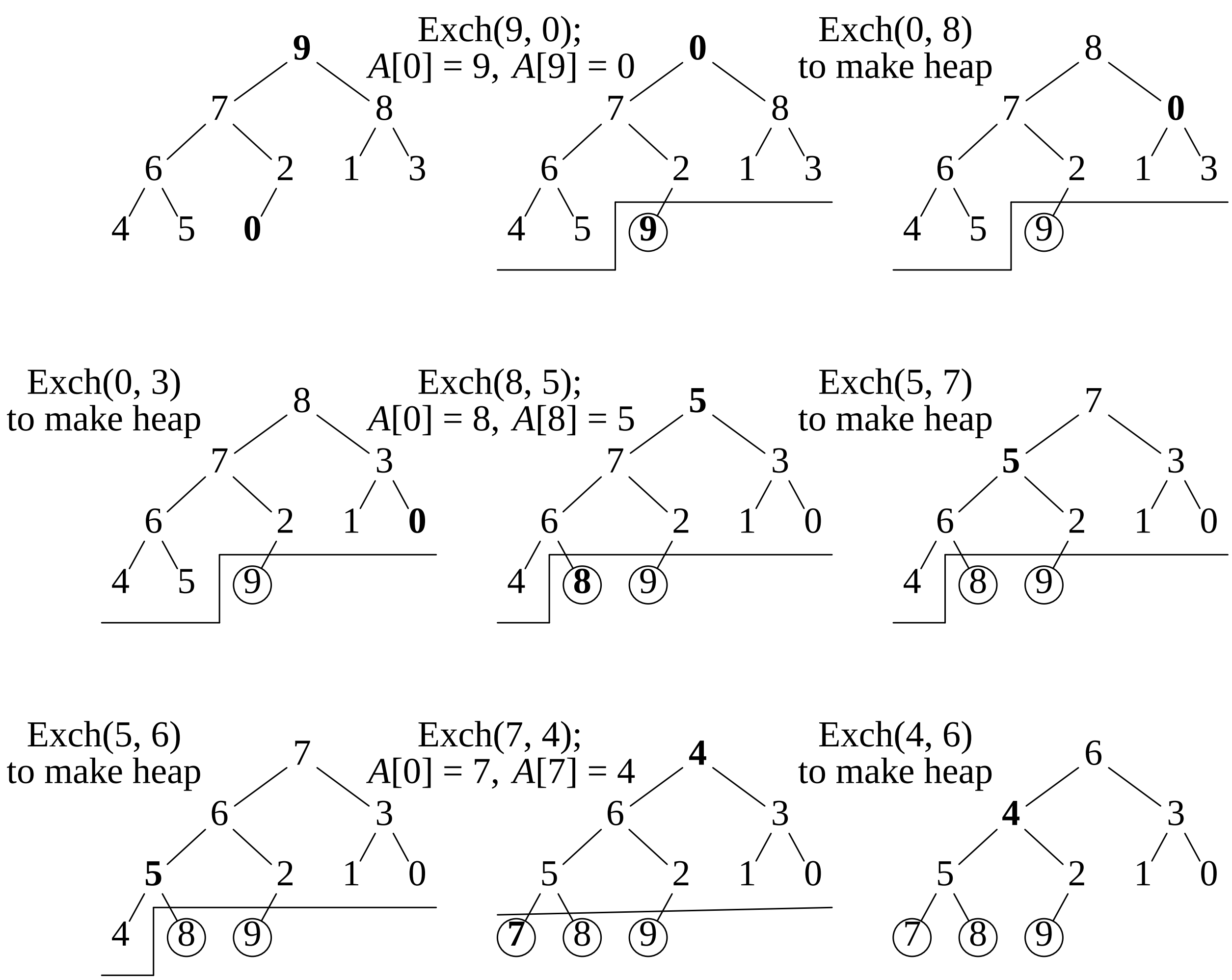
1. Show all possible heaps with 5 nodes and the node values {1, 2, 3, 4, 5}.

HEAP-SORTING METHOD

Two Parts in Heap-Sort: Let N = numItems.

- Make the input-array into a heap.
- Use the heap to sort as follows:
  - Exchange the max-item at root $A[0]$ with $A[N-1]$.
  - Make $A[0..N-2]$ into a max-heap: each child-value < parent-value.
  - Exchange the next max-item (again) at $A[0]$ with $A[N-2]$.
  - Make $A[0..N-3]$ into a heap and so on, each time working with a smaller initial part of the input-array.

Example. Part of the heap-sorting process.



HEAP-SORTING ALGORITHM

MakeHeap, using the recursive AddToHeap: n = numItems.

- nums[$(n-1)..(n-1)$] is an heap.
- For $i = n-2$, $n-3$, $\cdots$, 1, 0, make the tail part nums[$i..n-1$] into an heap by adding nums[i] to the heap nums[$i+1..n-1$].

AddToHeap(i, numItems): //call for i=numItems-1, numItems-2, ..., 0
1. If (nums[i] have no children) stop. //2i+1 > numItems-1
2. Otherwise, do the following:
   (a) Find index j of the largest child-items of nums[i].
   (b) If (nums[j] > nums[i]) then exchange(nums[i], nums[j])
   and call AddToHeap(j, numItems).

MakeHeap(numItems): //make nums[0..(numItems-1)] into a heap
1. If (numItems = 1) stop.
   //nums[i] has no children if i > numItems/2 - 1.
2. Else, for (i=numsItems/2 - 1; i≥0; i--) AddToHeap(i, numItems).

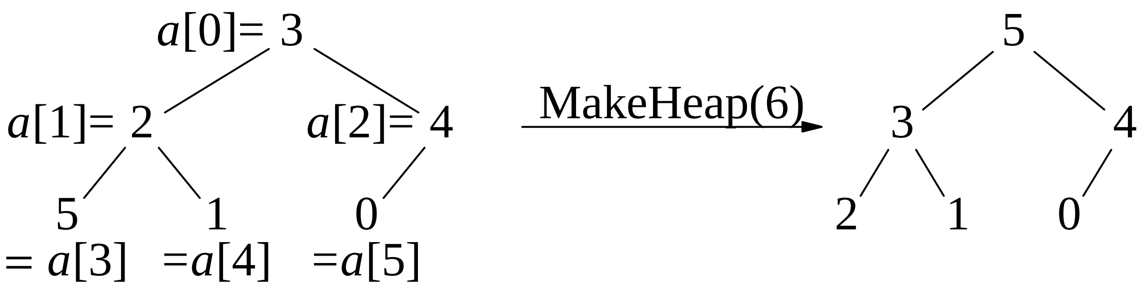
HeapSort, using recursion and AddToHeap:

- Implements Selection-Sort.
- Uses Heap-structture to successively find the max, the next max, the next next max and so on, filling the places nums[$n-1$], nums[$n-2$], $\cdots$, nums[0] in that order with the right item.

HeapSort(numItems): //sort nums[0..(numItems-1)] by heap-sort
1. If (numItems = 1) stop.
2. Otherwise, do the following:
   (a) If (this is the top-level call) then MakeHeap(numItems)
   (b) Exchange(nums[0], nums[numItems-1]),
   AddToHeap(0, numItems-1), and HeapSort(numItems-1).

UNDERSTANDING MakeHeap(numItems)

Input: nums[] = [3, 2, 4, 5, 1, 0] is not a heap; n = numItems = 6.

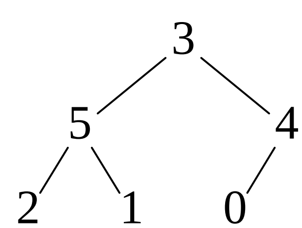


a[i] for nums[i], in short.

MakeHeap(6): Makes 3 calls to AddToHeap as shown below:

(1) AddToHeap(2,6): max-child index j = 5;
nums[5] = 0 $\not>$ 4 = nums[2], do nothing

(2) AddToHeap(1,6): max-child index j = 3;
nums[3] = 5 > 2 = nums[1], exchange(2, 5);
calls AddToHeap(3,6); //does nothing

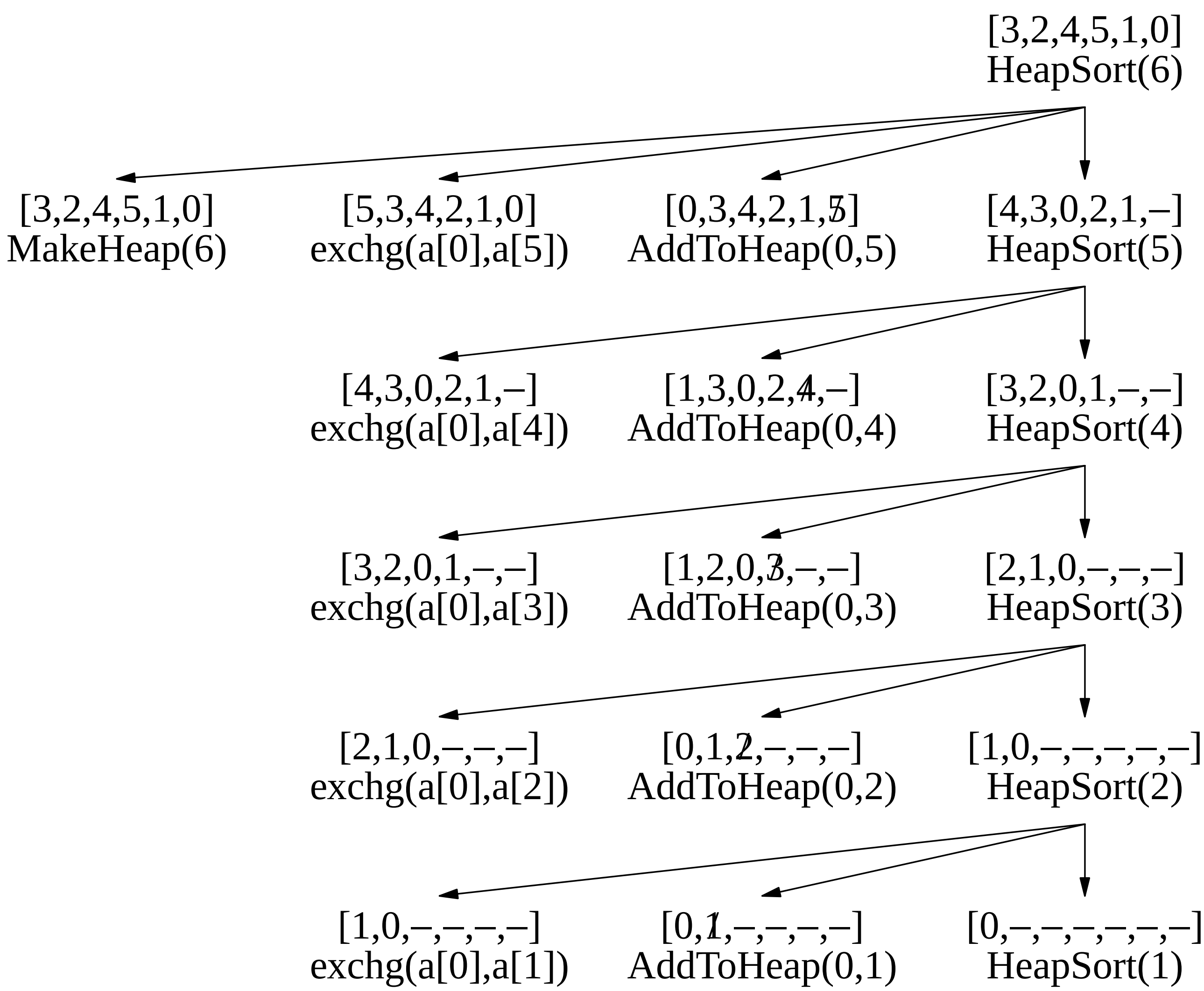


(3) AddToHeap(0,6): max-child index j = 1
nums[1] = 5 > 3 = nums[0], exchange(3, 5);
calls AddToHeap(3, 6); //does nothing
we get the final heap as shown on top.

Question: How can you modify AddToHeap(i, numItems) to eliminate some unnecesary calls to AddToHeap?

UNDERSTANDING HeapSort(numItems)

- Shown below are the recursive calls to HeapSort, calls to MakeHeap and AddToHeap, and the exchange-action, for sorting input [3, 2, 4, 5, 1, 0].
- Each node shows the input-array to its action, which is a function-call or the exchange operations.
- We only show the initial part of the array of interest at each point. An item is shown as marked by overstrike (such as ~~5~~ for 5 in 3rd child of root-node) before it is hidden away in remaining nodes.
- Calls to AddToHeap resulting from MakeHeap(6) are not shown.



PROGRAMMING EXERCISE

1. Implement the following functions; you can keep nums[0..(numItems-1)] as a global variable.

   void AddToHeap(int itemNum, int numItems)
   void MakeHeap(int numItems)
   void HeapSort(int numItems)

   Keep a constant NUM_ITEMS = 10.

   (a) First run MakeHeap-function for the input nums[0..9] = [0, 1, ..., 9], and show each pair of numbers (parent, child) exchanged, one pair per line (as shown below), during the initial heap-formation. These outputs will be generated by AddToHeap-function.

   (parent, child) exchanged: nums[4]=5, nums[9]=10
   ...

   (b) Then, after commenting out this detailed level output-statements, run HeapSort-function. This time you show successively the array after forming the heap and after exchange with the root-item (which puts the current max in the right place). The first few lines of the output may look like:

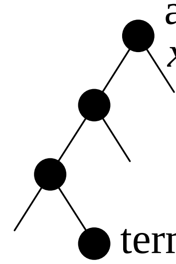
   Successive heap array and after exchange with root-item:
   [9, 8, 6, 7, 4, 5, 2, 0, 3, 1]
   [1, 8, 6, 7, 4, 5, 2, 0, 3, 9]
   [8, 7, 6, 3, 4, 5, 2, 0, 1]
   [1, 7, 6, 3, 4, 5, 2, 0, 8]
   ...

   (c) Repeat (b) also for the input [1, 0, 3, 2, ..., 9, 8].

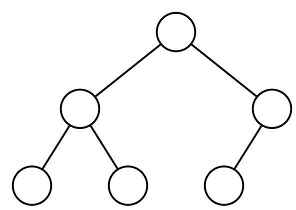
COMPLEXITY OF INITIAL HEAP FORMATION FOR *n* ITEMS

Cost of Adding a Node x:

- It may cause at most changes to the nodes along the path from x to a terminal node.



- The particular shape of an n-node heap means:



The shape of a heap on $n = 6$ nodes

  – At least $\lceil n/2 \rceil$ nodes are terminal nodes (no work for these).

  – The number of nodes on a path from root to a terminal node is at most $\lceil \log_2(n+1) \rceil$.

- Each change takes at most a constant time c (finding largest child and exchanging the node with that child).
- Total cost of adding a node $\leq c.[\lceil \log_2(n+1) \rceil - 1] = O(log\ n)$.
- Total for all nodes $\leq n.\, O(log\ n) = O(n.\, log\ n)$.

A better bound $O(n)$ for Total Cost: Assume $2^{m-1} \leq n < 2^m$.

- Total cost $\leq 1.(m-1) + 2.(m-2) + 4.(m-3) + \cdots + 2^{(m-2)}.1 = O(n)$.

COMPLEXITY OF HEAP-SORTING

Computing max, next max, next next max, $\cdots$:

- Each takes one exchange and one re-heap operation of adding nums[0] to the heap (of size less than the previous one).
  - This is $O(log\ n)$.
- Total of this phase for all nodes: $n. O(log\ n) = O(n. log\ n)$.

Total for Heap-Sort:

- Initial heap formation: $O(n)$.
- Rest of heap-sort: $O(n. log\ n)$.
- Total = $O(n) + O(n. log\ n) = O(n. log\ n)$.

APPLICATIONS OF SORTING

Car-Repair Scheduling:

You have a fleet of N cars waiting for repair, with the estimated repair times r_k for the car C_i, $1 \le k \le N$. What is the best repair-schedule (order of repairs) to minimize the total lost time for being out-of-service.

Example. Let $N = 3$, and $r_1 = 7$, $r_2 = 2$, and $r_3 = 6$. There are 3! = 6 possible repair-schedules.

Repair Schedule	Repair completion times			Total lost service-time
$\langle C_1, C_2, C_3\rangle$	7	7+2=9	7+2+6=15	31
$\langle C_1, C_3, C_2\rangle$	7	7+6=13	7+6+2=15	35
$\langle C_2, C_1, C_3\rangle$	2	2+7=9	2+7+6=15	26
$\langle C_2, C_3, C_1\rangle$	2	2+6=8	2+6+7=15	25
$\langle C_3, C_1, C_2\rangle$	6	6+7=13	6+7+2=15	34
$\langle C_3, C_2, C_1\rangle$	6	6+2=8	6+2+7=15	29

Best schedule: $\langle C_2, C_3, C_1\rangle$,
lost service-time = 2 + (2+6) + (2+6+7) = 25

Worst schedule: $\langle C_1, C_3, C_2\rangle$,
lost service-time = 7 + (7+6) + (7+6+2) = 35.

Question:

•? Show that the total service-time loss for the repair-order $\langle C_1, C_2, \cdots, C_N\rangle$ is $N.r_1 + (N-1).r_2 + (N-2).r_3 + \cdots + 1.r_N$.

•? What does this say about the optimal repair-order?

•? If $\langle C_1, C_2, \cdots, C_N\rangle$ is an optimal repair-order for all cars, is $\langle C_1, C_2, \cdots, C_m\rangle$ an optimal repair-order for C_i, $1 \le i \le m < N$?

PSEUDOCODE FOR OPTIMAL CAR REPAIR-SCHEDULE

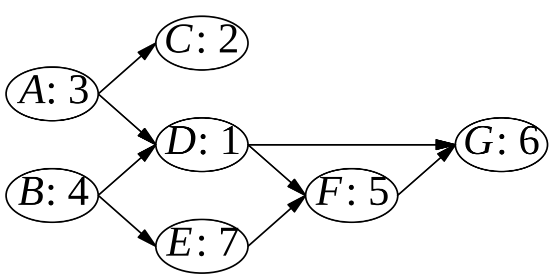
Algorithm OptimalSchedule:

Input: Repair times r_i for car C_i, $1 \le i \le N$.
Output: Optimal repair schedule $\langle C_{i_1}, C_{i_2}, \cdots, C_{i_N} \rangle$

1. Sort the cars in non-decreasing repair-times $r_{i_1} \le r_{i_2} \le \cdots \le r_{i_N}$.
2. Optimal repair schedule $\langle C_{i_1}, C_{i_2}, \cdots, C_{i_N} \rangle$, with total lost-time = $N.r_{i_1} + (N-1).r_{i_2} + (N-2).r_{i_3} + \cdots + 1.r_{i_N}$.

EXERCISE

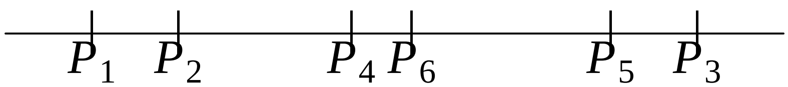
1. Give #(additions and multiplications) needed to compute $r_1 + (r_1 + r_2) + (r_1 + r_2 + r_3) + \cdots + (r_1 + r_2 + \cdots + r_N)$. (You may want to simplify the expressions first.)
2. How much computation is needed to find the lost service-times for all schedules?
3. What is the optimal car-repair order for the situation below, where a link (x, y) means car x must be repaired before car y?



The number next to each car is its repair time.

ANOTHER APPLICATION: FINDING A CLOSEST PAIR OF POINTS ON A LINE

Problem: Given a set of points P_i, $1 \le i \le N$ (≥ 2) on the x-axis, find P_i and P_j such that $|P_i - P_j|$ is minimum.



$\{P_4, P_6\}$ is the closest pair.

Application:

If P_i's represent national parks along a freeway, then a closest pair $\{P_i, P_j\}$ means it might be easier to find a camp-site in one of them.

Brute-force approach: Complexity $O(N^2)$.

1. For (each $1 \le i < j \le N$), compute d_{ij} = distance(P_i, P_j).
2. Find the pair (i, j) which gives the smallest d_{ij}.

Implementation (combines steps (1)-(2) to avoid storing d_{ij}'s):

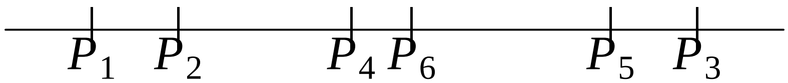
```
besti = 0; bestj = 1; minDist = Dist(points[0], points[1]);
for (i=0; i<numPoints; i++)  ////numPoints > 1
    for (j=i+1; j<numPoints; j++)
        if ((currDist = Dist(points[i], points[j])) < minDist)
           { besti = i; bestj = j; minDist = currDist; }
```

Question:

•? Give a slightly different algorithm (a variant of the above) and its implementation to avoid the repeated assignment "besti = i" in the nested for-loop; it should have fewer computations. Explain the new algorithm using a suitable test-data.

•? Restate the pseudocode to reflect the implementation.

A BETTER ALGORITHM FOR CLOSEST PAIR OF POINTS ON A LINE



$\{P_4, P_6\}$ is the closest pair.

The New Method:

- The point nearest to P_i is to its immediate left or right.
- Finding immediate neighbors of each P_i requires sorting the points P_i.

Algorithm NearestPairOfPoints (on a line):

Input: An array $nums[1: N]$ of N numbers.
Output: A pair of items $nums[i]$ and $nums[j]$ which are nearest to each other.

1. Sort $nums[1..N]$ in increasing order.
2. Find $1 \le j < N$ such that $nums[j+1] - nums[j]$ is minimum.
3. Output $nums[j]$ and $nums[j+1]$.

Complexity:

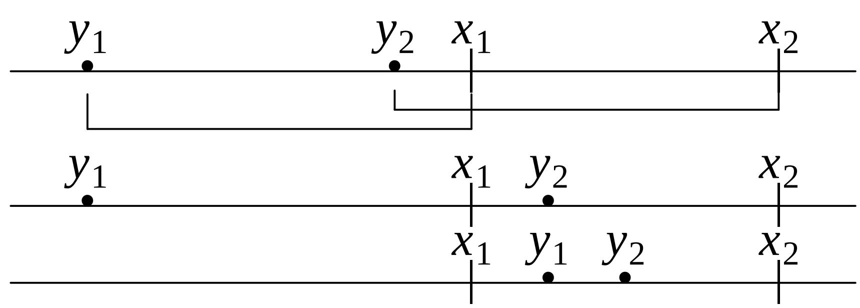
- Sorting takes $O(N\log N)$ time; other computations take $O(N)$ time.
- Total = $O(N\log N)$.

A geometric view sometimes leads to a better Algorithm.

A MATCHING PROBLEM

Problem:

- Scores $x_1 < x_2 < \cdots < x_N$ for N male students M_i in a test, and scores $y_1 < y_2 < \cdots < y_N$ for N female students F_i.
- Match male and female students $M_i \leftrightarrow F_{i'}$ in an 1-1 fashion that minimizes $E = \sum(x_i - y_{i'})^2$ $(1 \le i \le N)$, the squared sum of differences in scores for the matched-pairs.



The possible relative positions of x_i's and y_i's except for interchanging x_i's with y_i's.

Brute-force method:

1. For each permutation $(y_{1'}, y_{2'}, \cdots, y_{N'})$ of y_i's, compute E for the matching-pairs $x_i \leftrightarrow y_{i'}$.
2. Find the permutation that gives minimum E.

Question: How many ways the students can be matched?

Complexity: $O(N. N!)$.

- Computing $N!$ permutations takes at least $N(N!)$ time.
- Computing E for a permutation: $O(N)$; total = $O(N. N!)$.
- Finding minimum takes $O(N!)$.

A BETTER METHOD FOR THE MATCHING PROBLEM

Observation:

(1) The matching $\{x_1 \leftrightarrow y_1, x_2 \leftrightarrow y_2\}$ gives the smallest E for $N = 2$ in each of the three cases.

(2) The same holds for all $N > 2$: matching ith smallest x with ith smallest y gives the minimum E.

Question:

•? How can you prove (1)?

•? Consider $N = 3$, and $y_1 < y_2 < x_1 < y_3 < x_2 < x_3$. Argue that the matching $x_i \leftrightarrow y_i$ give minimum E. (Your argument should be in a form that generalizes to all N and to all distributions of x_i's and y_i's.)

Pseudocode (exploits output-properties):

1. Sort x_i's and y_i's (if they are not sorted).
2. Match M_i with $F_{i'}$ if x_i and $y_{i'}$ have the same rank.

Complexity: $O(Nlog\ N) + O(N) = O(Nlog\ N)$.

EXERCISE

1. Is it possible to solve the problem by recursion (reducing the problem to a smaller size) or by divide-and-conquer?

Every efficient Algorithm exploits some properties of input, output, or input-output relationship.

2-3 TREE: A GENERALIZATION OF SEARCH-TREE

2-3 Tree:

- An ordered rooted tree, whose nodes are labeled by items from a linear ordered set (like numbers) with the following shape constraints (S.1)-(S.2) and value constraints (V.1)-(V.3).

  (S.1) Each node has exactly one parent, except the root, and each non-terminal node has 2 or 3 children.

  (S.2) The tree is height-balanced (all terminal nodes are at the same level).

  (L.1) A node x with 2 children has one label, $label_1(x)$, with the following property, where $T_L(x)$ and $T_R(x)$ are the left and right subtree at x.

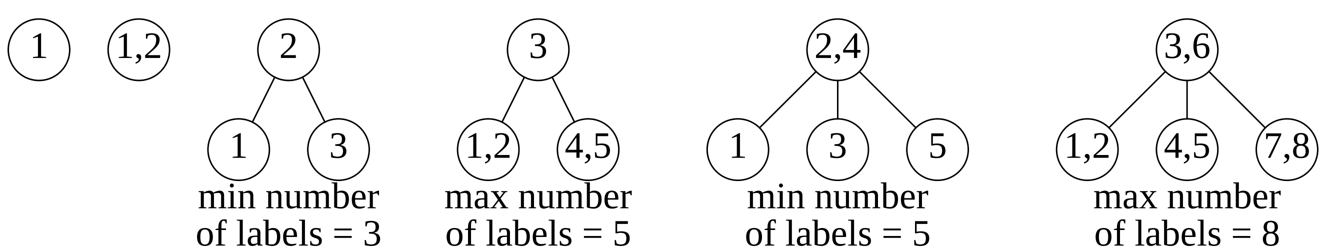
  $$\text{labels}(T_L(x)) < label_1(x) < \text{labels}(T_R(x))$$

  (L.2) A node x with 3 children has two labels, $label_1(x) < label_2(x)$, with the following property, where $T_M(x)$ is the middle subtree at x.

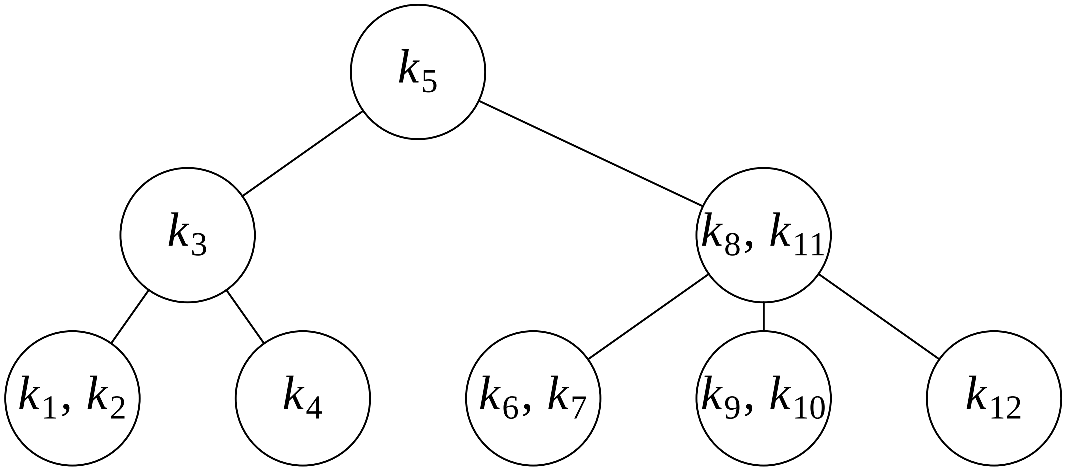
  $$\begin{aligned}\text{labels}(T_L(x)) < label_1(x) &< \text{labels}(T_M(x)) \\ &< label_2(x) < \text{labels}(T_R(x))\end{aligned}$$

  (L.3) A terminal node may have 1 or 2 labels.

Example. Some small 2-3 trees.



SEARCHING A 2-3 TREE



$k_1 < k_2 < k_3 < k_4 < k_5 < k_6 < k_7 < k_8 < k_9 < k_{10} < k_{11} < k_{12}$

Searching for a value $k_9 \leq x \leq k_{10}$:

- Compare x and the values at the root: $k_5 < x$; branch right
- Compare x and the values at the right child: $k_8 < x < k_{11}$; branch middle
- Compare x and the values at the middle child: $k_9 \leq x \leq k_{10}$; if $x = k_9$ or $x = k_{10}$, the value is found, else x is not there.

Role of Balancedness Property of 2-3 trees:

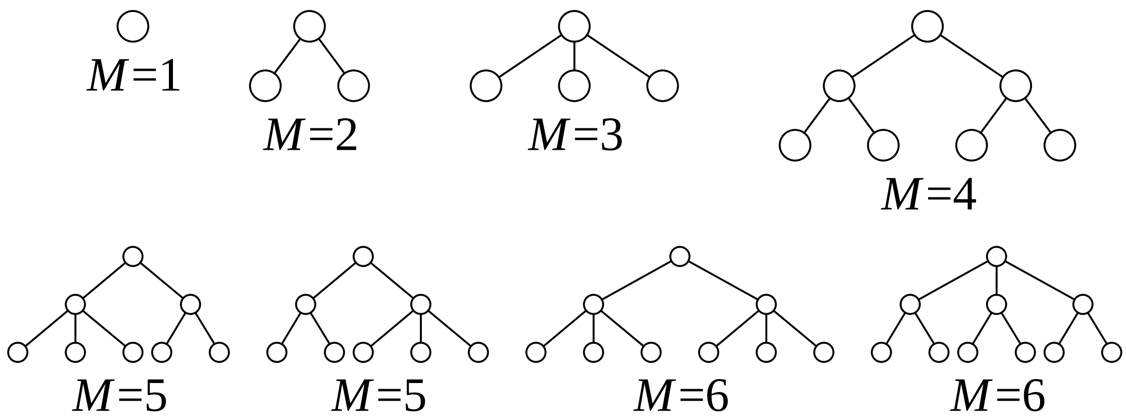
- Ensures optimum search efficiency.

B-tree and B^+-tree:

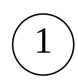
- These are more general form of 2-3 trees, which are the main data-structures used in databases to optimize search efficiency for very large data-sets. (We talk about them later.)

BUILDING 2-3 TREES

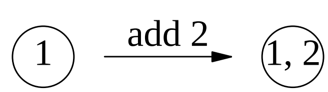
Shapes of 2-3 Trees (with different M = #(terminal nodes)):



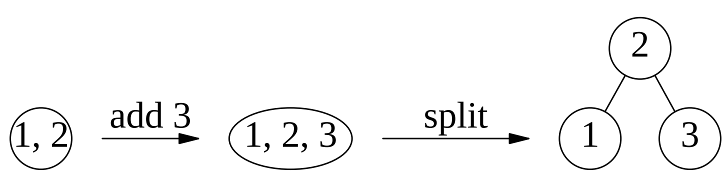
Adding 1 to an empty tree:



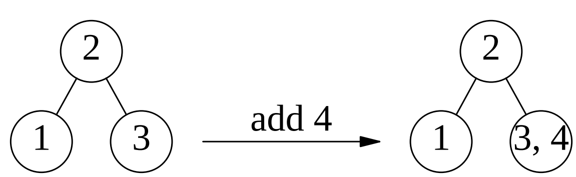
Adding 2: Find the place for 2, and add if there is space.



Adding 3: Find place for 3, split if no space adding a parent node.

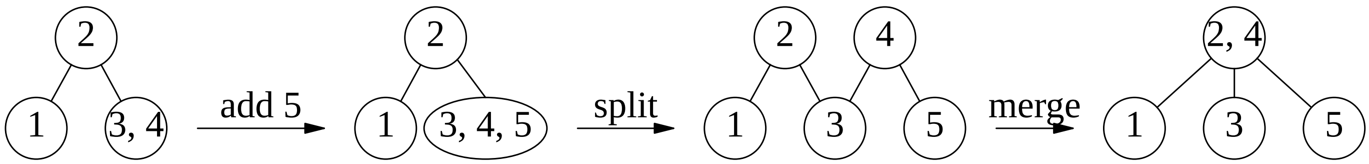


Adding 4: Find the place for 4 and add if there is space.

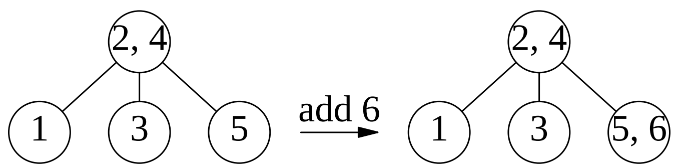


CONTD.

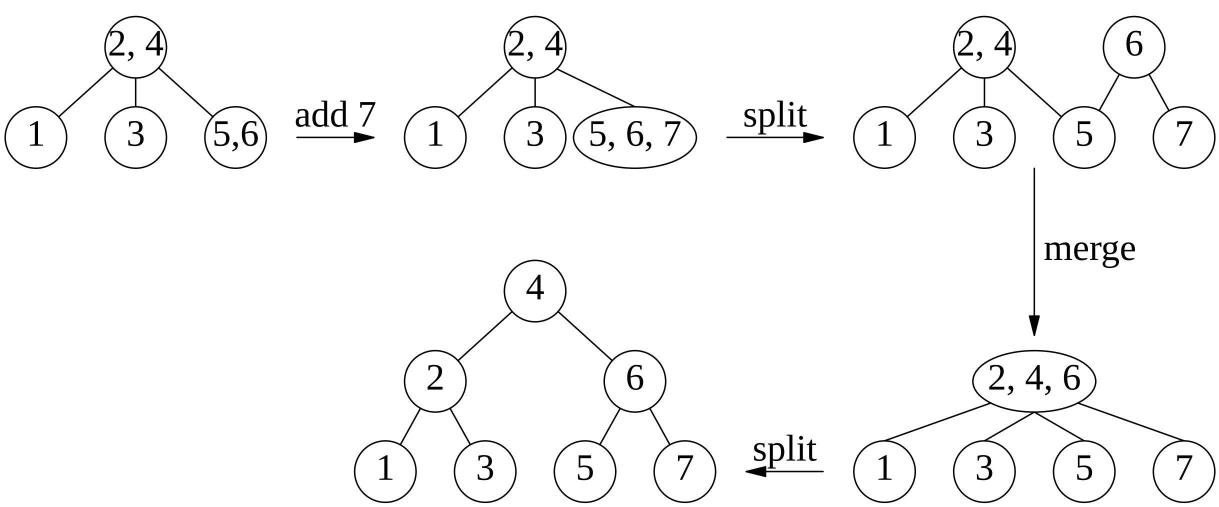
Adding 5: Find place for 5, split if no space adding a parent, and adjust by merging.



Adding 6: Find place for 6, and add it if there is space.



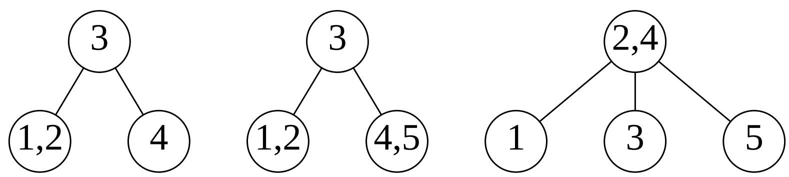
Adding 7: Find place for 7, split if no space adding a parent, adjust by merging, and if no space, then split by adding parent again.



Question: Show the results after adding 1.1, 2.3, and 1.2.

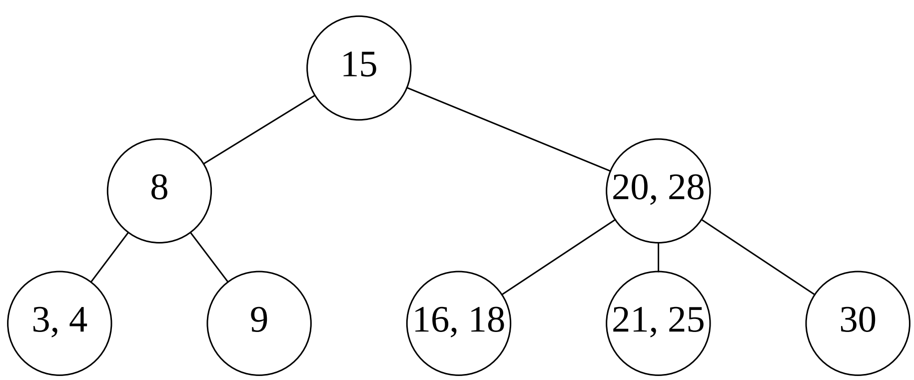
EXERCISE

1. How many ways the 2-3 tree on the left can arise as we build the 2-3 tree by inputting {1, 2, 3, 4} in different order. What were the 2-3 trees before the 4th item were added? Show that the two 2-3 trees on the right arise respectively from 48 and 72 (total = 120 = 5!) permutations of {1, 2, ⋯, 5}.



2. Show the minimum and the maximum number data-items that can be stored in 2-3 trees with 5 and 6 terminal nodes. Show the labels in the nodes (using the numbers 1, 2, 3, ⋯) for both cases.

3. What information we can store at the nodes of a 2-3 tree to quickly find the key-value of the *i*th smallest item? Explain the use of this information to find the 9th item in the 2-3 tree below.

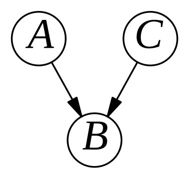


TOPOLOGICAL SORTING OR ORDERING NODES OF A DIGRAPH

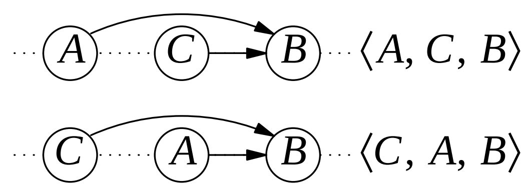
Topo. Sorting (ordering):

- List the digraph's nodes so that each link goes from left to right.
- This can be done if and only if there are no cycles in the digraph.

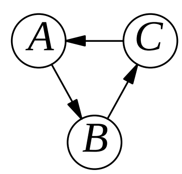
An *acyclic* digraph.



Two topo. orderings:



A digraph with a *cycle*.

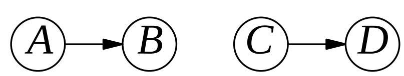


- Any linear arrangement of the nodes will have at least link going from right to left.
- No topological ordering.

- The topological orderings = The schedules for the tasks at nodes.

Questions:

•? Show all possible topological orderings of the digraph below with 4 nodes $\{A, B, C, D\}$ and two links $\{(A, B), (C, D)\}$. If we add the link (A, D), how many of these top. ordering are eliminated?



•? Is it true that each acyclic digraph has at least one source-node and at least one sink-node? Is the converse also true? For each "no" answer, give an examples to illustrate your answer.

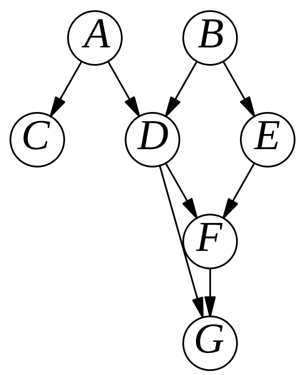
•? What is the maximum number of links in an acyclic digraph with N nodes? What is the number if we allow cycles?

•? Show all possible acyclic digraphs on 3 nodes (do not label nodes).

PSEUDOCODE FOR TOPOLOGICAL ORDERING

Pseudocode:

1. Choose a node x which is currently a source-node, i.e., all its preceding nodes (if any) have been output,
2. Repeat step (1) until all nodes are output.



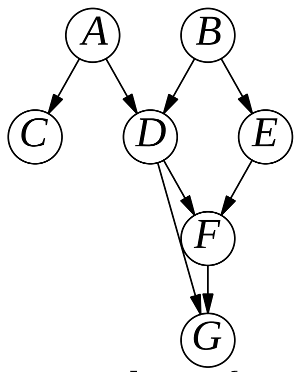
Example. Shown below are possible choice of nodes x and a particular choice of x at each iteration of step (1).

$\{A, B\}$	$\{B, C\}$	$\{C, D, E\}$	$\{D, E\}$	$\{E\}$	$\{F\}$	$\{G\}$
A	B	C	D	E	F	G

Relevant Data Structures:

- A stack to keep track of current source-nodes.
  - A node x enters the stack when it becomes a source-node.
  - When we remove x from the stack, we delete the links from it, add new source-nodes to the stack (if any), and output it.
- Keep track of inDegree(x) = #(links to x) to determine when it becomes a source-node.

USE OF STACK DATA-STRUCTURE FOR TOPOLOGICAL-SORTING



inDegree(y) = number of links (x, y) to y
outDegree(y) = number of links (y, z) from y
source-nodes = {x: inDegree(x) is 0}
sink-nodes = {z: outDegree(z) is 0}
adjList(x) = adjacency-list of node x

Source nodes = {A, B},
Sink nodes = {C, G}.

adjList(D) = ⟨F, G⟩
adjList(G) = empty-list

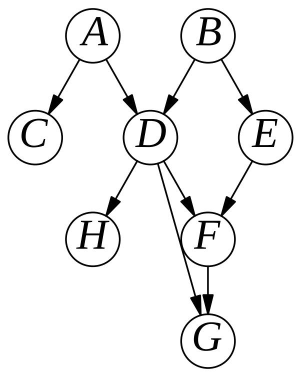
Stack = nodes with current inDegree(x) = 0 and not yet output.

Stack (top on right)	Node x Selected	Nodes and their initial or reduced inDegrees						
		A: 0	B: 0	C: 1	D: 2	E: 1	F: 2	G: 2
⟨A, B⟩	B		–	1	1	0	2	2
⟨A, E⟩	E		–	1	1	–	1	2
⟨A⟩	A	–	–	0	0	–	1	2
⟨C, D⟩	D	–	–		–	–	0	1
⟨C, F⟩	F	–	–		–	–	–	0
⟨C, G⟩	G	–	–		–	–	–	–
⟨C⟩	C	–	–	–	–	–	–	–

EXERCISE

1. Show the processing in the Topo-Sorting algorithm after adding the link (G, A), which creates one or more cycles in the digraph. (Remember the algorithm stops when the stack become empty.)
2. Show in a table form the processing of the digraph above using a queue instead of a stack in the topological-sorting Algorithm. Use the notation ⟨A, B, C⟩ for a queue with C as the head and A as the tail. If we add D, the queue becomes ⟨D, A, B, C⟩; if we now remove an item, the queue becomes ⟨D, A, B⟩.

ADJACENCY-LIST REPRESENTATION OF A DIGRAPH



Array Index	Node name & outDegree	Adjacency-list of node indices; array-size = outDegree(x)
0	A; 2	[2, 3]
1	B; 2	[3, 4]
2	C; 0	null
3	D; 3	[5, 6, 7]
4	E; 1	[5]
5	F; 1	[6]
6	G; 0	null
7	H; 0	null

```
typedef struct {
        char nodeName[MAX_LENGTH];
        int outDegree,
            *adjList; //array size = outDegree
            //*linkCosts; array size = outDegree
} st_graphNode;
```

Adjacency Matrix Representation:

- This is not suitable for some of our algorithms.

$$
\begin{array}{c} \\ A \\ B \\ C \\ D \\ E \\ F \\ G \\ H \end{array}
\begin{array}{c}
\begin{array}{cccccccc} A & B & C & D & E & F & G & H \end{array} \\
\begin{bmatrix}
0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0
\end{bmatrix}
\end{array}
$$

TOPOLOGICAL SORTING ALGORITHM

Computation of inDegrees:

1. For (each node i) initialize inDegree(i) = 0;
2. For (each node i and for each j in adjList(i)
   add 1 to inDegree(j);

Initialization of stack: (stack = array of size numNodes)

1. Initialize stack with nodes of indegree zero;

Selection of a node to process:

1. Select top(stack) and delete it from the stack;

Processing node ***i*****:**

l. Add node i to output;
2. For (each node j in adjList(i)) do the following:
   (a) reduce inDegree(j) by one;
   (b) if (inDegree(j) = 0) add j to stack;

Algorithm TopSort():

Input: An acyclic digraph, with adjLists representation.
Output: A topological ordering of its nodes.

1. Compute indegrees of all nodes.
2. Initialize the stack.
3. While (stack is not empty) do the following:
   (a) Let i = top(stack), delete it from stack, and add it to topOrder-array;
   (b) Process node i;

COMPLEXITY ANALYSIS OF TOPOLOGICAL-SORT ALGORITHM

Observations:

- Each link (x, y) of the digraph is processed exactly twice.
  - All links are looked at once in computing the indegrees.
  - All links are looked at the second time in course of the stack updates; specifically, when we remove x from the stack, we look at all links (x, y) from x the second time.
- We look at also each node x exactly 2*inDegree(x) + 2 times.
  - First time, in initializing inDegree(x) = 0.
  - Then, exactly inDegree(x) many times as it is successively updated by adding 1 till it reaches the value inDegree(x).
  - Then, another inDegree(x) many times as it is successively updated by subtracting 1 till it becomes 0.
  - Finally, when it is taken out of the stack.

Fact: $\sum_{all\ x}$ inDegree(x) = $\sum_{all\ x}$ outDegree(x) = #(links in the digraph).

Example. For the digraph on page 1.43, the two sums are

0 + 0 + 1 + 2 + 1 + 1 + 2 + 2 = 9 and 2 + 2 + 0 + 3 + 1 + 1 + 0 = 9.

Complexity:

- Since each of the operations listed above takes a constant time, total computation time is $O(\#(nodes) + \#(links))$.

PROGRAMMING EXERCISE

1. Implement a function topologicalSort() based on the algorithm TopSort. It should produce one line of output as shown below.

   stack=[0 1], node selected = 1, topOrder-array = [1]
   stack=[0 4], node selected = 4, topOrder-array = [1 4]

   - Use a function readDigraph() to read an input file digraph.dat and build the adjacency-list representation of the digraph. File digraph.dat for the digraph on page 1.43 is shown below.

     ```
     8 //numNodes; next lines give: node (outdegree) adjacent-nodes
     0 (2) 2 3
     1 (2) 3 4
     2 (0)
     3 (3) 5 6 7
     4 (1) 5
     5 (1) 6
     6 (0)
     7 (0)
     ```

   - In topologicalSort(), use a dynamically allocated local array inDegree[0..numNodes-1]. Compute inDegrees by

     ```
     for (i=0; i<numNodes; i++) {
         outDegree = nodes[i].outdegree;
         adjList = nodes[i].adjList;
         for (j=0; j<outdegree; j++)
             inDegrees[adjList[j]]++;
     }
     ```
     or
     ```
     for (i=0; i<numNodes; i++)
         for (j=0; j<nodes[i].outDegree; j++)
             inDegrees[nodes[i].adjList[j]]++;
     ```

EXERCISE

1. Given an ordering of the nodes of an acyclic digraph, how will you check if it is a topo. ordering? Give a pseudocode and explain your algorithm using the acyclic digraph on page 1.43.

2. How can you compute a topo. ordering without using inDegrees? (Hint: If outDegree(x) = 0, can we place x in a topo. ordering?)

3. Modify topological-sorting algorithm to compute for all nodes y, numPathsTo(y) = #(paths to y starting at some source-node). State clearly the key ideas. Shown below are numPathsTo(y) and also the paths for the digraph $\vec{G}$ on page 1.42.

x	num-PathsTo(x)	Paths
A	1	$\langle A\rangle$ //trivial path from A to A, with no links.
B	1	$\langle B\rangle$
C	1	$\langle A, C\rangle$
D	2	$\langle A, D\rangle, \langle B, D\rangle$
E	1	$\langle B, E\rangle$
F	3	$\langle A, D, F\rangle, \langle B, D, F\rangle, \langle B, E, F\rangle,$
G	5	$\langle A, D, G\rangle, \langle A, D, F, G\rangle, \cdots, \langle B, E, F, G\rangle,$

Hints:

(a) If (x, y) is a link, what is the relation between numPathsTo(x) and numPathsTo(y). What does it suggest about which of them should be computed first?

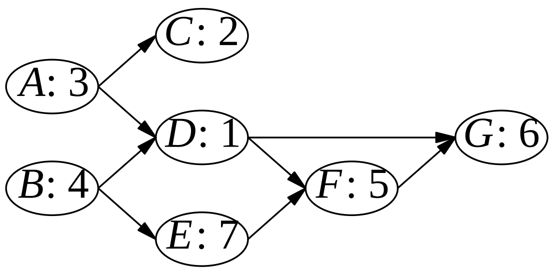
(b) How will you compute numPathsTo(y) in terms of all numPathsTo(x) for $\{x$: (x, y) is a link to $y\}$?

4. Modify your algorithm to compute numPathsFromTo(x, y) = #(paths to node y from node x) for all nodes y to which there is ≥ 1 path from x (which may not be a source-node). Explain the algorithm for x = A and y = F using the digraph shown earlier.

TOPOLOGICAL ORDERING AND TASK SCHEDULING

Precedence Constraint on Repairs:

- Each link (x, y) means car x must be repaired before car y.



The number next to each car is its repair time.

Possible Repair Schedules:

- These are exactly all the topological orderings.
- Two repair-schedules and their lost service-times:

$\langle A, B, C, D, E, F, G\rangle$: $3.7 + 4.6 + \cdots + 6.1 = 96$

$\langle B, A, C, D, E, F, G\rangle$: $4.7 + 3.6 + \cdots + 6.1 = 95$

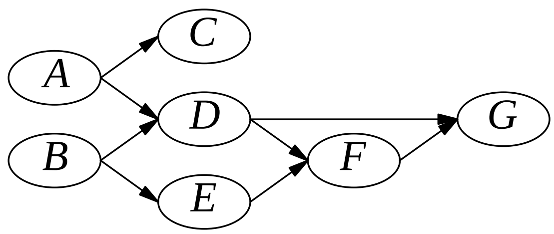
Question:

•? What is the optimal schedule?

•? What is the algorithm for creating optimal schedule?

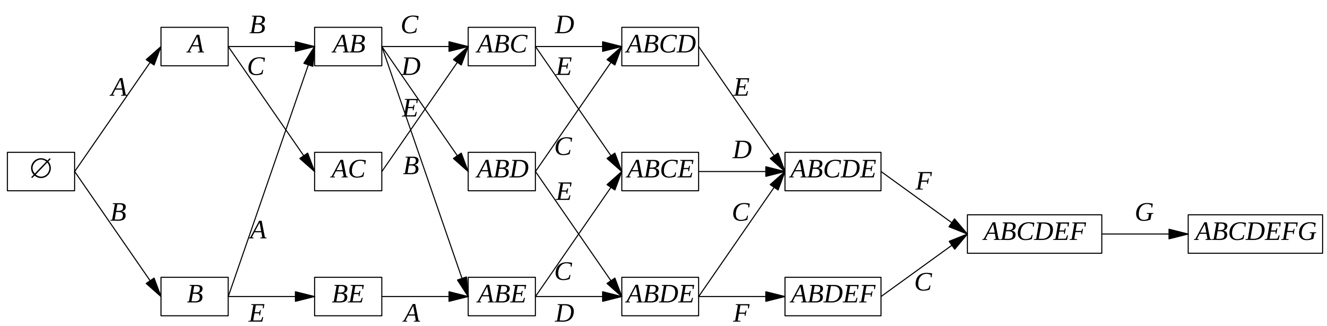
ALL POSSIBLE SCHEDULES

An Acyclic Digraph of Task Precedence Constraints:

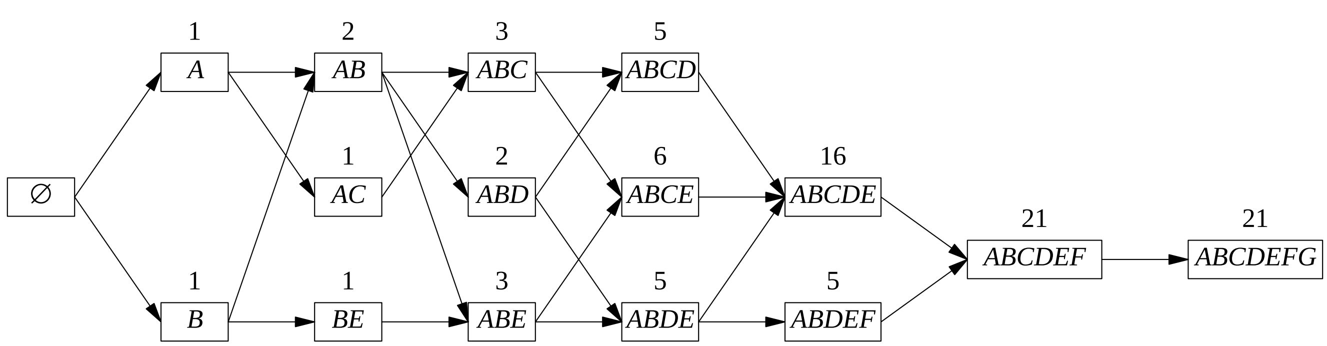


The Acyclic Digraph for Representing Schedules:

- Each node represents the tasks completed.
- Each path from the source-node ∅ to the sink-node *ABCDEFG* gives a schedule.



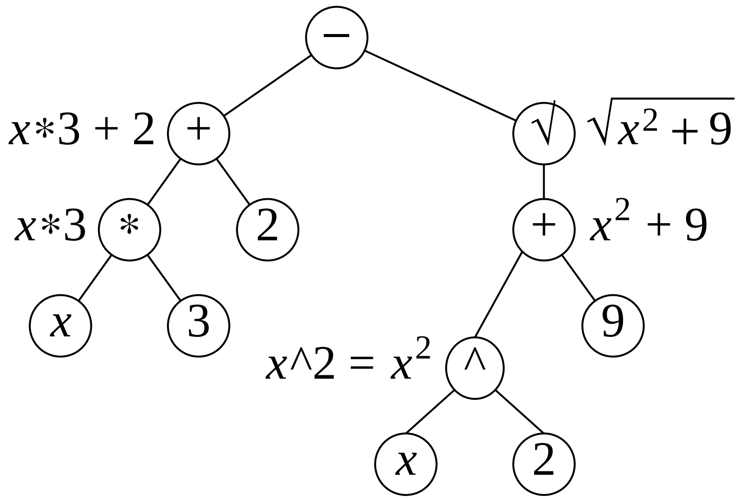
- The number of these paths gives #(schedules) = #(topological orderings).



SOME OTHER APPLICATIONS OF STACK DATA-STRUCTURE

Expression-Tree: It is an *ordered* tree (not a binary tree).

$$x*3 + 2 - \sqrt{x^2 + 9} = ((x*3) + 2) - \sqrt{x^2 + 9}$$



- Each non-terminal node gives an operator; also, associated with each node is the expression corresponding to the subtree at it.
- The children of a non-terminal node give the operands of the operator at the node.
- The terminal nodes are the basic operands.

Evaluation Method:

- The children of a non-terminal node are evaluated before evaluating the expression at a node.
- This requires the post-order traversal of the tree:

  Visit the children from left to right, and then the node.

Post-fix form (corresponds to post-order traversal):

$$\underline{\underline{\underline{x\ 3\ *}\ 2\ +}}\ \underline{\underline{\underline{x\ 2\ \wedge}\ 9\ +}\sqrt{}}\ -$$

POST-FIX EXPRESSION EVALUATION USING A STACK

Processing Method: Stack is initially empty.

- Processing an operand: add its value to stack.
- Processing an operator: remove the operands of the operator from the stack, apply the operator to those values, and add the new value to stack.
- The final value of the expression is the only item in the stack at the end of processing.

Example. If $x = 4$, then $x\ 3 * 2 + x\ 2$ ^ $9 + \surd -$ equals 9.

Top of stack is the right in the notation $\langle \cdots \rangle$.

Stack	After item processed	Stack	After item processed
$\langle 4 \rangle$	x	$\langle 14, 4, 2 \rangle$	2
$\langle 4, 3 \rangle$	3	$\langle 14, 16 \rangle$	^
$\langle 12 \rangle$	$*$	$\langle 14, 16, 9 \rangle$	9
$\langle 12, 2 \rangle$	2	$\langle 14, 25 \rangle$	+
$\langle 14 \rangle$	+	$\langle 14, 5 \rangle$	$\surd$
$\langle 14, 4 \rangle$	x	$\langle 9 \rangle$	–

EXERCISE

1. Show an infix expression that give rise to the post-fix expression "x 2 3 x $*$ $*$ + 2 / 15 +"; make sure that you use proper parentheses as needed, but no unnecessary ones. Show the stacks in evaluating this post-fix expression for $x = 5$.
2. Show the stacks in converting your infix expression in Problem #1 to the post-fix form (using the method on next page).

CONVERTING ARITHMETIC EXPRESSIONS TO POST-FIX FORM

Input: $x*3 + 2 - \mathrm{sqrt}(x \wedge 2 + 9)$ (’^’ = exponentiation)
Output: $x\ 3 * 2 + x\ 2 \wedge 9 + \mathrm{sqrt} -$

- Stack has only operators, including function-symbols and ’(’.
- Operator priority: $\{+, -\} < \{*, /\} < \wedge <$ function-names.

Conversion Method: Initially, stack is empty.

- Processing an operand: Output it.
- Processing ’(’ or a function-symbol: add it to stack.
- Processing ’)’: remove everything from stack upto the first ’(’ and a function-symbol below it, if any; ’(’ is not added to output.
- Processing an operator ’op’:
  - While ((stack ≠ ∅) and (top(stack) ≥ ’op’)), remove top(stack) and output it. (See next page.)
  - Then add ’op’ to stack.
- If end of input, output every thing in stack.

Stack	Item proc.	Output	Stack	Item proc.	Output
$\langle\rangle$	x	x	$\langle -, \mathrm{sqrt}, (\rangle$	(	
$\langle * \rangle$	$*$		$\langle -, \mathrm{sqrt}, (\rangle$	x	x
$\langle * \rangle$	3	3	$\langle -, \mathrm{sqrt}, (, \wedge \rangle$	$\wedge$	
$\langle + \rangle$	+	$*$	$\langle -, \mathrm{sqrt}, (, \wedge \rangle$	2	2
$\langle + \rangle$	2	2	$\langle -, \mathrm{sqrt}, (, + \rangle$	+	$\wedge$
$\langle - \rangle$	–	+	$\langle -, \mathrm{sqrt}, (, + \rangle$	9	9
$\langle -, \mathrm{sqrt} \rangle$	sqrt		$\langle - \rangle$	)	+, sqrt
			$\langle\rangle$		–

RIGHT-ASSOCIATIVE OPERATIONS AND ITS IMPACT ON POST-FIX CONVERSION

Left Association:

- $x - y - z$ means $(x - y) - z$ but not $x - (y - z)$.
- Post-fix form of $x - y - z$ is $x\ y - z\ -$.

  Post-fix form of $x - (y - z)$ is $x\ y\ z - -$.

Right Association:

- $x \wedge y \wedge z$ means $x \wedge (y \wedge z)$ and not $(x \wedge y) \wedge z$, where "$\wedge$" is the exponentiation operation.

  The post-fix form of $x \wedge y \wedge z$ is therefore $xyz\wedge\wedge$ instead of $xy\wedge z\wedge$.
- $x = y = 3$ means $x = (y = 3)$, i.e., $\{y = 3;\ x = y;\}$ instead of $\{x = y;\ y = 3;\}$.

  Likewise, x += y += 3 means x += (y += 3), i.e., {y += 3; x += y;} instead of {x += y; y += 3;}. Here, ’+=’ is the operator.
- Post-fix form of $x = y = 3$: $x\ y\ 3 = =$.

Processing Right Associative Operator ’op’:

- For conversion to post-fix form, we replace the test (top(stack) ≥ ’op’) by (top(stack) > ’op’).

Processing Assignment Operator "=" in Post-fix Form:

- In processing the post-fix form "y 3 =", we do not put the value of y in stack (as in the case of processing "y 3 +").
- Other special indicators (called ’lvalue’ are added).

TREE OF A STRUCTURE-DEFINITION AND THE ADDRESS ASSIGNMENT PROBLEM

```
typedef struct {
        int id;
        char flag, name[14];
        double val;
} IdName;
typedef struct ListNodeDummy {
        IdName idName;
        struct ListNodeDummy *next, *prev;
} ListNode;
ListNode x;
```

Number of Bytes for Basic Types:

- size(int) = 4, size(char) = 1, size(double) = 8.
- size(x) = 40, not 4 + 1 + 14 + 8 + 4 + 4 = 35.

	flag ↓		5 bytes wasted			
id		name[0..13]		val	next	prev

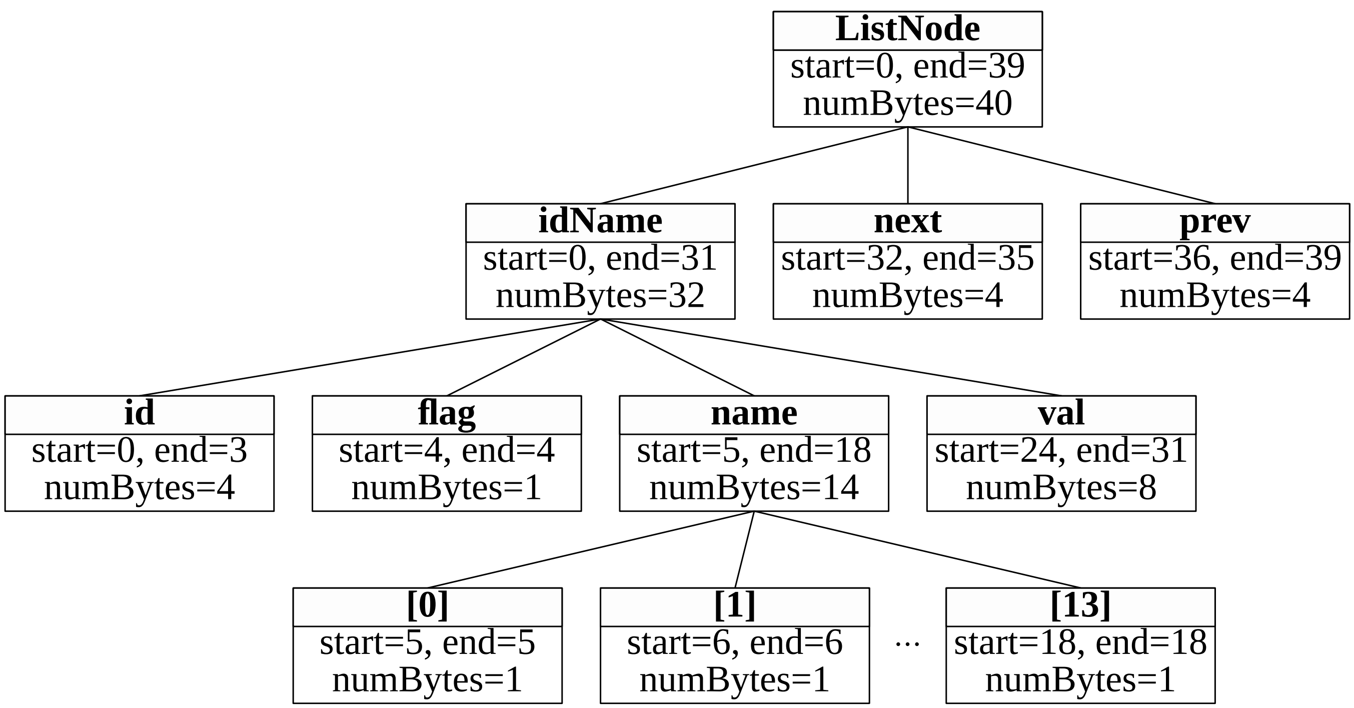
- An actual address allocation of the components of x:

```
                  x = 268439696
           x.idName = 268439696
        x.idName.id = 268439696
      x.idName.flag = 268439700
      x.idName.name = 268439701
   x.idName.name[0] = 268439701
   x.idName.name[1] = 268439702
  x.idName.name[13] = 268439714
       x.idName.val = 268439720
             x.next = 268439728
             x.prev = 268439732
```

- Start-address(x) is a multiple of 8; because displacement(val) = 24 within x, start-address(val) is a multiple of 8.
- It makes start-adrress of id, next, and prev multiples of 4.

CONTD.

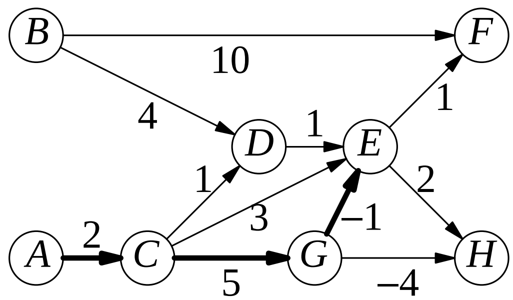
```
typedef struct {
        int id;
        char flag, name[14];
        double val;
} IdName;
typedef struct ListNodeDummy {
        IdName idName;
        struct ListNodeDummy *next, *prev;
} ListNode;
ListNode x;
```



EXERCISE

1. Give a pseudocode for determining start-address, end-address, and numBytes for all nodes of an arbitrary structure-tree. Assume you know the type of each terminal node and you have the structure-tree. (Hint: Your pseudocode must indicate: (1) the order in which the start, end, and numBytes at each node of the structure-tree are computed. and (2) how each of these is computed based on values of various quantities at some other nodes.)

LONGEST-PATHS IN AN ACYCLIC DIGRAPH

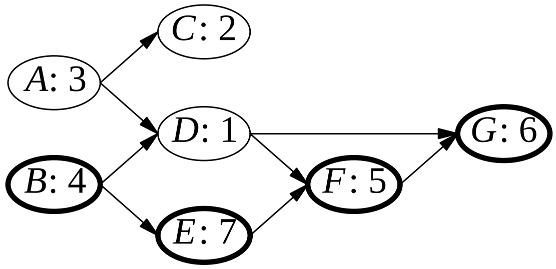


Paths from A to E and their lengths
(1) $\langle A, C, E\rangle$; length = 2+3 = 5
(2) $\langle A, C, D, E\rangle$; length = 2+1+1 = 4
(3) $\langle A, C, G, E\rangle$; length = 2 + 5 – 1 = 6

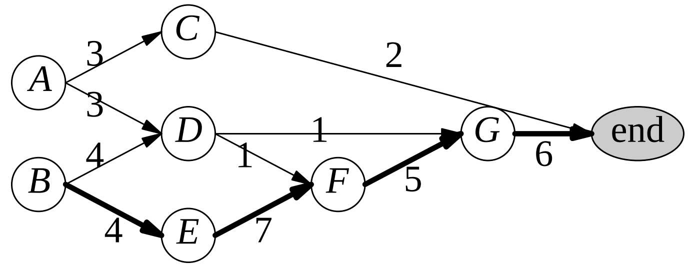
- $w(x, y)$ = length (cost or weight) of link (x, y); it can be negative.
- Length of a path = sum of the lengths of its links.
- LongestPathFromTo(A, E): $\langle A, C, G, E\rangle$; length = 6.

Application:

- Critical-path/critical-task analysis in project scheduling.
- Assume unlimited resources for work on tasks in parallel.
- The new acyclic digraph for critical-path analysis:
  - Add a new "end"-node and connect each sink node to it.
  - The length of each link (x, y) = time to complete task x.

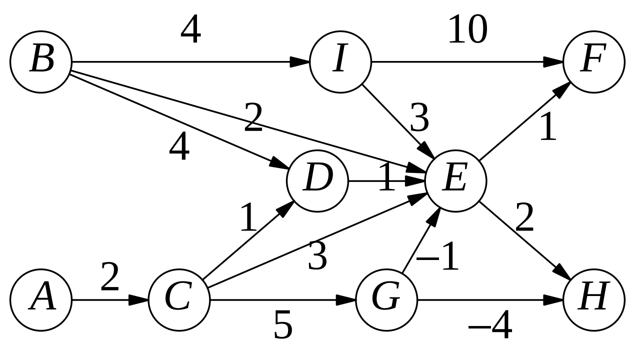


The number next to each car is its repair time.



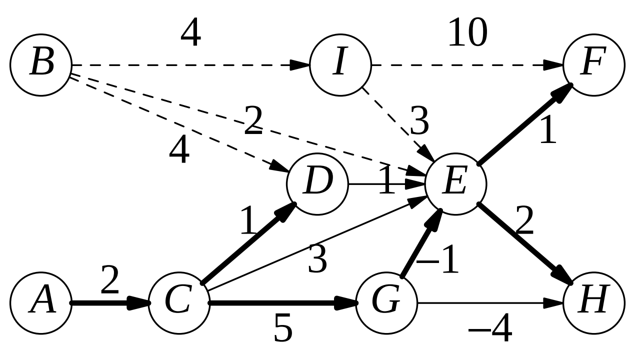
The digraph for critical-path analysis. The longest-path: $\langle B, E, F, G, \text{"end"}\rangle$.

TREE OF LONGEST-PATHS



Tree of Longest Paths From startNode = *A*:

- First, we can reduce the digraph so that the only source-node is the startNode.



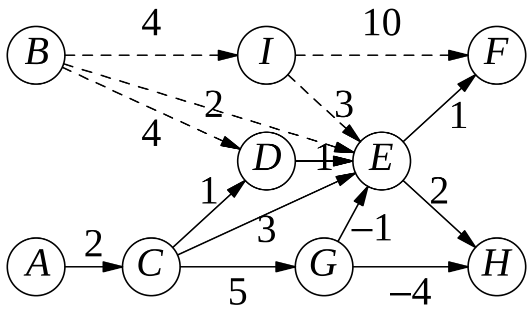
- The tree contains *one* longest path from startNode to each node *x* which can be *reached* from startNode. (It is not a binary tree or an ordered tree.)
- To obtain the reduced digraph (which is a must for the algorithm given later to work properly) we can successively delete source-nodes $x \neq$ startNode and links from those x.

Question:

•? Show the reduced digraph to compute longest paths from node *B*; also show a tree of longest paths from node *B*.

DIGRAPH REDUCTION

- We actually don't delete any nodes/links or modify adjaceny-lists.
- We pretend deletion of a link (x, y) by reducing inDegree of y.



Reductions for statrtNode = A:
- inDegree(D) = 2 – 1 = 1
- inDegree(E) = 5 – 2 = 3
- inDegree(F) = 2 – 1 = 1

Algorithm ReduceAcyclicDigraph(startNode):

Input: An acyclic digraph in adjacency-list form
Output: Reduced indegrees.

1. Compute indegrees of all nodes.
2. While (there is a node $x \neq$ startNode and inDegree(x) = 0) do:

   if (x is not processed)
   then for each $y \in$ adjList(x) deduce inDegree(y) by 1.

Notes:

- Use a stack to hold the nodes x with inDegree(x) = 0 and which have not been processed yet. Initialize stack with all $x \neq$ startNode and inDegree(x) = 0.
- We do not modify the adjList(x) of any node, and thus the digraph is actually not changed.
- The longest-path algorithm works with the reduced indegrees.

LONGEST-PATH COMPUTATION

Array Data-Structures Used:

$d(x)$ = current longest path to x from startNode.
parent(x) = the node previous to x on the current longest path to x; parent(startNode) = startNode.
inDegree(x) = number of links to x yet to be looked at.

Stack Data-structure Used:

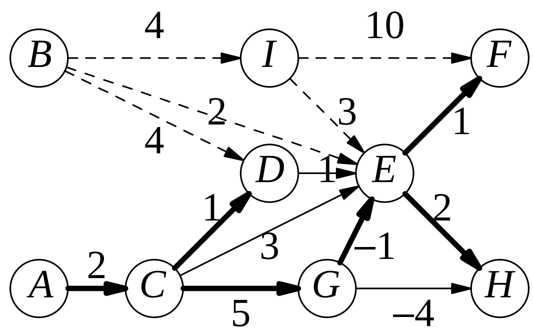
- Stack holds all nodes to which the longest-path is known, but links from which have not been processed yet.

Algorithm LongestPathsFrom(startNode):

Input: An acyclic digraph in adjacency-list form and startNode.
Output: A tree of longest paths to each x reachable from startNode.

1. Apply ReduceAcyclicDigraph(startNode).
2. Initialize a stack with startNode, let $d(x) = -\infty$ and parent(x) = –1 for each node x with indegree(x) > 0, and finally let $d(startNode)$ = 0 and parent(startNode) = startNode.
3. While (stack $\neq$ empty) do the following:
   (a) Let x = top(stack); remove x from stack.
   (b) For (each $y \in$ adjList(x)) do:
      (i) If $(d(x) + w(x, y) > d(y))$, then let $d(y) = d(x) + w(x, y)$ and parent(y) = x.
      (ii) Reduce inDegree(y) by 1 and if it equals 0 then add y to stack and print the longest-path to y from startNode (using the successive parent-links) and $d(y)$.

ILLUSTRATION OF LONGEST-PATH COMPUTATION



StartNode = A.

Stack	Node x	For each node y, inDegree(y) and ($d(y)$, parent(y))						
		A; 0 (0, A)	C; 1 ($-\infty$, ?)	D; 1 ($-\infty$, ?)	E; 3 ($-\infty$, ?)	F; 1 ($-\infty$, ?)	G; 1 ($-\infty$, ?)	H; 2 ($-\infty$, ?)
$\langle A \rangle$	A		$0+2>-\infty$ (2, A) $1 \to 0$					
$\langle C \rangle$	C			$2+1>-\infty$ (3, C) $1 \to 0$	$2+3>-\infty$ (5, C) $3 \to 2$		$2+5>-\infty$ (7, C) $1 \to 0$	
$\langle D, G \rangle$	G				$7-1>5$ (6, G) $2 \to 1$			$7-4>-\infty$ (3, G) $2 \to 1$
$\langle D \rangle$	D				$3+1 \leq 6$ $1 \to 0$			
$\langle E \rangle$	E					$6+1>-\infty$ (7, E) $1 \to 0$		$6+2>3$ (8, E) $1 \to 0$
$\langle F, H \rangle$	H							
$\langle F \rangle$	F							

- We can use minus the sum of all positive link-weights as $-\infty$.

EXERCISE

1. Show the complete executions of RreduceAcyclicDigraph(B) and LongestPathsFrom(B) in the suitable table forms.
2. How many times a link (x, y) is processed during the longest-path computation and when?
3. What can change as we process a link (x, y) and how long does it take to all those computations?
4. Why is it that the longest-path to a node y cannot be computed untill all remaining links to y (after the digraph reduction) have been processed? (For example, we must look at the links (C, E), (D, E), and (G, E) before we can compute the longest-path to C?)

PROGRAMMING EXERCISE

1. Develop a function void longestPathsFrom(int startNode). (Use $-\sum|w(x, y)|$, summed over all links (x, y), instead of $-\infty$.) Show the following outputs for startNode B using the example digraph discussed.

   (a) Print the input digraph, with node name, nodeIndex, node's outDegee in parenthesis, adjacency-list (with weight of the link in parenthesis) in the form:

   C, 2 (3): 3(1), 4(3), 6(5)

   Put the information for each node on a separate line. There should be an appropriate header-line (like "Acyclic digraph: node name, nodeIndex, outdegree, and adjList with link-costs").

   (b) Show the successive stacks (one per line) every time it is changed during the digraph reduction process. As usual give an appropriate heading before printing the stacks. Use the node names when you print the stack.

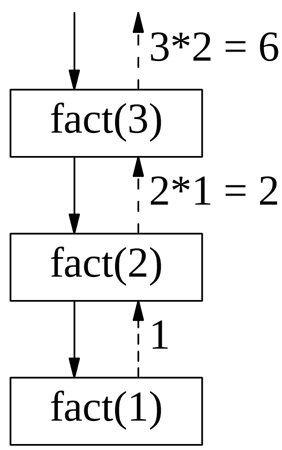
   (c) Next, when the longest-paths are computed, for each link (x, y) processed, show the link (x, y); also, if there is a change in $d(y)$ then shown the new $d(y)$ and parent(y), and when inDegree(y) becomes 0 show the final values of $d(y)$ and parent(y). For example, for startNode = A, the processing of the links (C, E), (G, E), and (D, E) should generate output lines

   link (C, E)**:** $d(E)$ **= 5, parent(**E**) =** C
   link (G, E)**:** $d(E)$ **= 6, parent(**E**) =** G
   link (D, E)**:** $d(E)$ **= 6, parent(**E**) =** G**, final value**

CALL-RETURN TREE OF FUNCTION-CALLS

Example.

```
int factorial(int n) //n >= 0
{ if ((n == 0) || (n == 1))
      return(1);
 else return(n*factorial(n-1));
}
```



EXERCISE

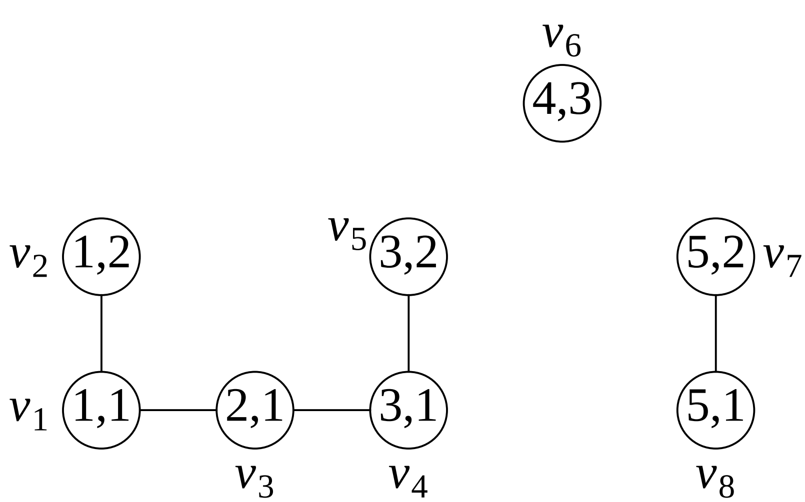
1. Show the call-return tree for the initial call Fibonacci(4), given the definition below; also show the return values from each call. Is the resulting tree a binary tree? If not, what kind of tree is it?

```
int Fibonacci(int n) //n >= 0
{if ((n == 0) || (n == 1))
    return(1);
 else return(Fibonacci(n-2) + Fibonacci(n-1));
}
```

A PROBLEM IN WIRELESS NETWORK

Problem: Given the coordinates (x_i, y_i) of the nodes v_i, $1 \le i \le N$, find the minimum transmission-power that will suffice to form a connected graph on the nodes.

- A node with transmission power P can communicate with all nodes within distance $r = c.\sqrt{P}$ from it ($c > 0$ is a constant).
- Let $r_{\min}$ be the minimum r for which the links $E(r) = \{(v_i, v_j): d(v_i, v_j) \le r\}$ form a connected graph on the nodes. Then, $P_{\min} = (r_{\min}/c)^2$ gives the minimum transmission power to be used by each node.



The links $E(1)$ corresponding to $P = 1/c^2$

Question:

1? What is $r_{\min}$ for the set of nodes above? Give an example to show that $r_{\min} \ne \max$ {distance of a node nearest to v_i: $1 \le i \le N$}. (If $r_{\min}$ were always equal to the maximum , then what would be an Algorithm to determine $r_{\min}$?)

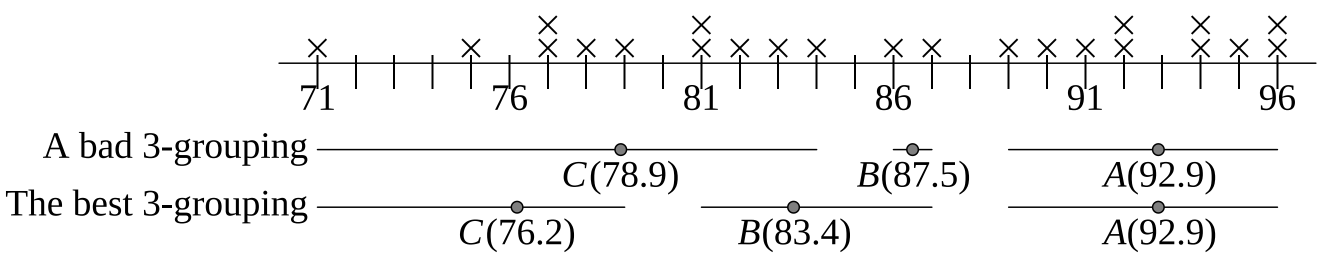
GROUPING NUMERICAL SCORES INTO CLASSES

Problem: Find the best grade-assignment A, B, C, etc to the student-scores x_i, $1 \leq i \leq N$, on a test. That is, find the best grouping of the scores into classes A, B, $\cdots$.

Interval-property of a group:

- If $x_i < x_k$ are two scores in the same group, then all in-between scores x_j ($x_i < x_j < x_k$) are in the same goup.
- Thus, we only need to find the group boundaries.

Example. Scores of 23 students in a test (one '×' per student).



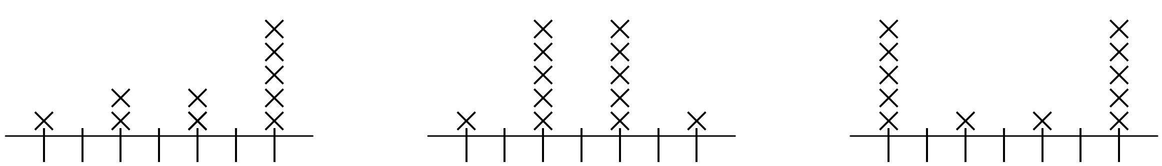
Closest-Neighbor Property (CNP) for Optimal Grouping:

- Each x_i is closest to the average of the particular group containing it compared to the average of other groups.

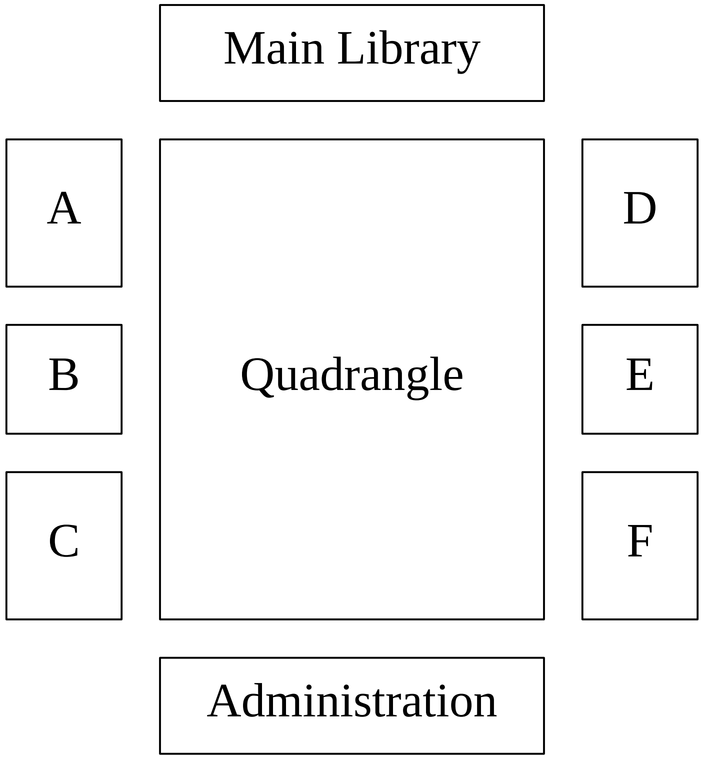
Question:

1? Give an application of such grouping for weather-data, say.

2? Find the best 2-grouping using CNP for each data-set below. Do these groupings match your intuition?



TWO EXAMPLES OF BAD ALGORITHMS



Algorithm#1 FindBuildingA:

1. Go to Main Library.
2. When you come out of the library, it is on your right.

Algorithm#2 FindBuildingA:

1. Go to the north-west corner of Quadrangle.

Questions:

1? Which Algorithm has more clarity?

2? Which one is better (more effi cient)?

3? What would be a better Algorithm?

WHAT IS WRONG IN THIS ALGORITHM

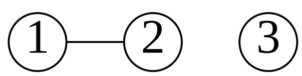
Algorithm GenerateRandomTree(n): //nodes = $\{1, 2, \cdots, n\}$

Input: n = #(nodes); $n \geq 2$.
Output: The edges (i, j), $i < j$, of a random tree.

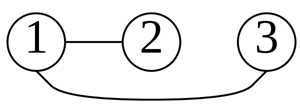
1. For (each $j = 2, 3, \cdots, n\}$, choose a random $i \in \{1, 2, \cdots, j - 1)$ and output the edge (i, j).

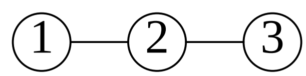
Successive Edges Produced for n = 3:

- $j = 2$: the only possible $i = 1$ and the edge is (1, 2).

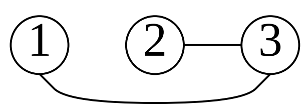


- $j = 3$; i can be 1 or 2, giving the edge (1, 3) or (2, 3).





Cannot generate the tree:



Always test your Algorithm.

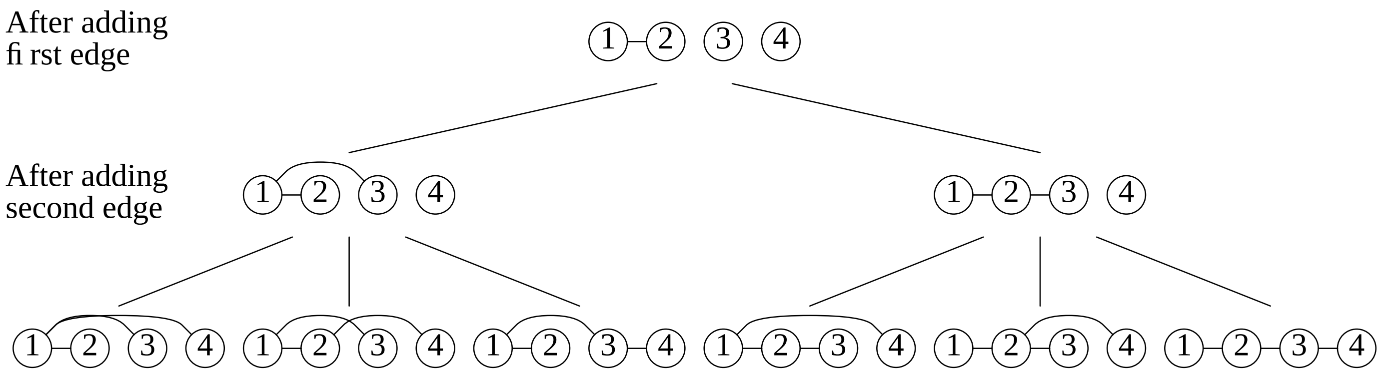
Question:

1? Does the above Algorithm always generate a tree (i.e., a connected acyclic graph)? Show all graphs generated for n = 4.

2? How do you modify GenerateRandomTree(n) so that all trees with n nodes can be generated (i.e., no one is excluded)?

3? Why would we want to generate the trees (randomly or all of them in some order) - what would be an application?

TREES GENERATED BY GenerateRandomTree(4)



Only 6 different trees are generated, each with degree(4) = 1.

Question:

1? Does the following Algorithm generate all trees on n nodes? What is the main inefficiency in this Algorithm?

1. Let $E = \varnothing$ (empty set).
2. For ($k = 1, 2, \cdots, n-1$), do the following:
   (a) Choose random i and j, $1 \le i < j \le n$ and $(i, j) \notin E$.
   (b) If $\{(i, j)\} \cup E$ does not contain a cycle (how do you test it?), then add (i, j) to E; else goto step (a).

2? Give a recursive Algorithm for generating random trees on nodes $\{1, 2, \cdots, n\}$. Does it generating each of n^{n-2} trees with the same probability?

3? Do we get a random tree (each tree with the same probability) by applying a random permutation to the nodes of a tree obtained by GenerateRandomTree(4)?

4? Give a pseudocode for generating a random permutation of $\{1, 2, \cdots, n\}$. Create a program and show the output for $n = 3$ for 10 runs and the time for 10 runs for $n = 100{,}000$.

PSEUDOCODES ARE SERIOUS THINGS

Pseudocode is a High-Level Algorithm Description:

- It *must* be unambiguous (clear) and concise, with sufficient details to allow
  - correctness proof and.
  - performance efficiency estimation
- It is *not* a "work-in-progress" or a "rough" description.

Describing Algorithms in pseudocode forms requires substantial skill and practice.

TYPES OF ALGORITHMS

Problem: (1) Input (= given)
(2) Output (= to find)

↓

Algorithm Design

↓

Pseudocode: (1) Key steps in the solution method
(2) Key data-structures

- Choose a proper solution method first and then select a data-structure to fit the solution method.

Exploit Input/Output Properties:

- Exploit properties/structures among the different parts of the problem-input.
- Exploit properties/structures of the solution-outputs, which indirectly involves properties of input-output relationship.

Method of Extension (problem size N to size $N+1$, recursion)
Successive Approximation (numerical Algorithms)
Greedy Method (a special kind of search)
Dynamic Programming (a special kind of search)
Depth-first and other search methods

Programming tricks alone are not sufficient for efficient solutions.

USE OF OUTPUT-STRUCTURE

Problem: Given an array of N numbers $nums[1..N]$, compute $partialSums[i] = nums[1] + nums[2] + \cdots + nums[i]$ for $1 \le i \le N$.

Example. $nums[1..5]$: 2, -1, 5, 3, 3
$partialSums[1..5]$: 2, 1, 6, 9, 12

- There is no input-structure to exploit here.

Two Solutions. Both can be considered method of extension.

(1) A brute-force method.

$partialSums[1] = nums[1]$;
for (i=2 to N) do the following:
  $partialSums[i] = nums[1]$;
  for (j=2 to i) add $nums[j]$ to $partialSums[i]$;

#(additions involving $nums[.]$) $= 0 + 1 + \cdots + (N-1) = N(N-1)/2 = O(N^2)$.

(2) Use the property "$partialSums[i+1] = partialSums[i] + nums[i+1]$" among output items.

$partialSums[1] = nums[1]$;
for (i=2 to N)
  $partialSums[i] = partialSums[i-1] + nums[i]$;

#(additions involving $nums[.]$) $= N - 1 = O(N)$.

- The $O(N)$ Algorithm is optimal because we must look at each $nums[i]$ at least once.

ANOTHER EXAMPLE OF THE USE OF OUTPUT-STRUCTURE

Problem: Given a binary-matrix $vals[1..M, 1..N]$ of 0's and 1's, obtain $counts(i, j)$ = #(1's in $vals[.,.]$ in the range $1 \le i' \le i$ and $1 \le j' \le j$) for all i and j.

Example.

$$\text{vals} = \begin{bmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 1 \end{bmatrix} \qquad \text{counts} = \begin{bmatrix} 1 & 1 & 1 & 2 \\ 1 & 2 & 2 & 4 \\ 1 & 3 & 4 & 7 \end{bmatrix}$$

- Since $vals[i, j]$'s can be arbitrary, there is no relevant input property/structure.
- The outputs $counts(i, j)$ have many properties as shown below; the first one does not help in computing $counts(i, j)$.

$$counts(i, j) \le \begin{cases} counts(i, j+1) \\ counts(i+1, j) \end{cases}$$

$$\begin{aligned} counts(1, j+1) &= counts(1, j) + vals[1, j+1] \\ counts(i+1, 1) &= counts(i, 1) + vals[i+1, 1] \\ counts(i+1, j+1) &= counts(i+1, j) + counts(i, j+1) \\ &\quad - counts(i, j) + vals[i+1, j+1] \end{aligned}$$

Not all input/output properties may be equally exploitable in a given computation.

Algorithm:

1. Let *counts*$(1, 1)$ = *vals*$[1, 1]$; compute the remainder of first row *counts*$(1, j)$, $2 \le j \le N$, using *counts*$(1, j+1)$ = *counts*$(1, j)$ + *vals*$[1, j+1]$.
2. Compute the first column *counts*$(i, 1)$, $1 \le i \le M$, similarly.
3. Compute the remainder of each row ($i + 1 = 2, 3, \cdots, M$), from left to right, using the formula for *counts*$(i+1, j+1)$ above.

Exploiting the output-properties includes choosing a proper order of computing different parts of output.

Complexity Analysis:

We look at the number of additions/subtractions involving *counts*(i, j) and *vals*$[i', j']$.

Step 1: $N - 1 = O(N)$
Step 2: $M - 1 = O(M)$
Step 3: $3(M-1)(N-1) = O(MN)$
Total: $O(MN)$; this is optimal since we must look at each item *vals*$[i, j]$ at least once.

Brute-force method::

1. For each $1 \le i \le M$ and $1 \le j \le N$, start with *counts*$(i, j) = 0$ and add to it all *vals*$[i', j']$ for $1 \le i' \le i$ and $1 \le j' \le j$.

Complexity: #(additions) = $\sum_{i=1}^{M} \sum_{j=1}^{N} ij = (\sum_{i=1}^{M} i)(\sum_{j=1}^{N} j) = O(M^2 N^2)$

MAXIMIZING THE SUM OF CONSECUTIVE ITEMS IN A LIST

Problem: Given an array of numbers $nums[1..N]$, find the maximum M of all $S_{ij} = \sum nums[k]$ for $i \le k \le j$.

Example: For the input $nums[1..15] =$

$[-2, \underline{7, 3, -1, -4, 3, -4, 9}, -5, 3, 1, -20, 11, -3, -1]$,

the maximum is $7 + 3 - 1 - 4 + 3 - 4 + 9 = 13$.

Brute-Force Method:

- For ($j = 1$ to N), compute S_{ij}, $1 \le i \le j$, using the method of partial-sums and let $M(j) = \max\{S_{ij}: 1 \le i \le j\}$.
- $M = \max\{M(j): 1 \le j \le N\}$.

Question: What is the complexity?

Observations (assume that at least one $nums[i] > 0$):

- Eliminate items equal to 0.
- The initial (terminal) –ve items are not used in a solution.
- If a solution S_{ij} uses a +ve item, then S_{ij} also uses the immediate +ve neighbors of it. This means we can replace each group of consecutive +ve items by their sum.
- If a solution S_{ij} uses a –ve item, then S_{ij} uses the whole group of consecutive –ve items containing it and also the group of +ve items on immediate left and right sides. This means we can replace consecutive –ve items by their sum.

Simplify Input: It is an array of alternate +ve and –ve items.

$$nums[1..9] = [10, -5, 3, -4, 9, -5, 4, -20, 11].$$

ADDITIONAL OBSERVATIONS

Another Observation: There are three possibilities:

(1) M = *nums*[1].

(2) *nums*[1] is combined with others to form M. Then we can replace *nums*[1. . 3] by *nums*[1]+*nums*[2]+*nums*[3].

(3) *nums*[1] is not part of an optimal solution. Then we can throw away *nums*[1. . 2].

- A similar consideration applies to *nums*[N].

Search For a Solution for *nums*[] = [10, –5, 3, –4, 9, –5, 4, –20, 11]:

(a) 10 or solution from [8, –4, 9, –5, 4, –20, 11]
or solution from [3, –4, 9, –5, 4, –20, 11],
i.e., 10 or solution from [8, –4, 9, –5, 4, –20, 11].

(b) 10 or 8 or solution from [13, –5, 4, –20, 11]
or solution from [9, –5, 4, –20, 11],
i.e., 10 or solution from [13, –5, 4, –20, 11].

(c) 10 or 13 or solution from [12, –20, 11]
or solution from [4, –20, 11],
i.e., 13 or solution from [12, –20, 11].

(d) 13 or 12 or solution from [3] or solution from [11].

(e) Final solution: $M = 13 = 8 - 4 + 9 = 10 - 5 + 3 - 4 + 9$.

Question:

•? Is this a method of extension (explain)?

•? Can we formulate a solution method by starting at the middle +ve item (divide and conquer method)?

A RECURSIVE ALGORITHM

Algorithm MAX_CONSECUTIVE_SUM: //initial version

Input: An array $nums[1..N]$ of alternative +ve/-ve numbers, with $nums[1]$ and $nums[N] > 0$.
Output: Maximum sum M for a set of consecutive items.

1. Let $M_1 = nums[1]$.
2. If (N >= 3) then do the following:
   (a) Let $nums[3] = nums[1] + nums[2] + nums[3]$ and let M_2 be the solution obtained by applying the Algorithm to $nums[i]$, $3 \le i \le N$.
   (b) Let M_3 be the solution obtained by applying the Algorithm to $nums[i]$, $3 \le i \le N$. (M_3 is the best solution when none of $nums[1]$ and $nums[2]$ are used.)

   else let $M_2 = M_3 = M_1$.
3. Let $M = \max\{M_1, M_2, M_3\}$.

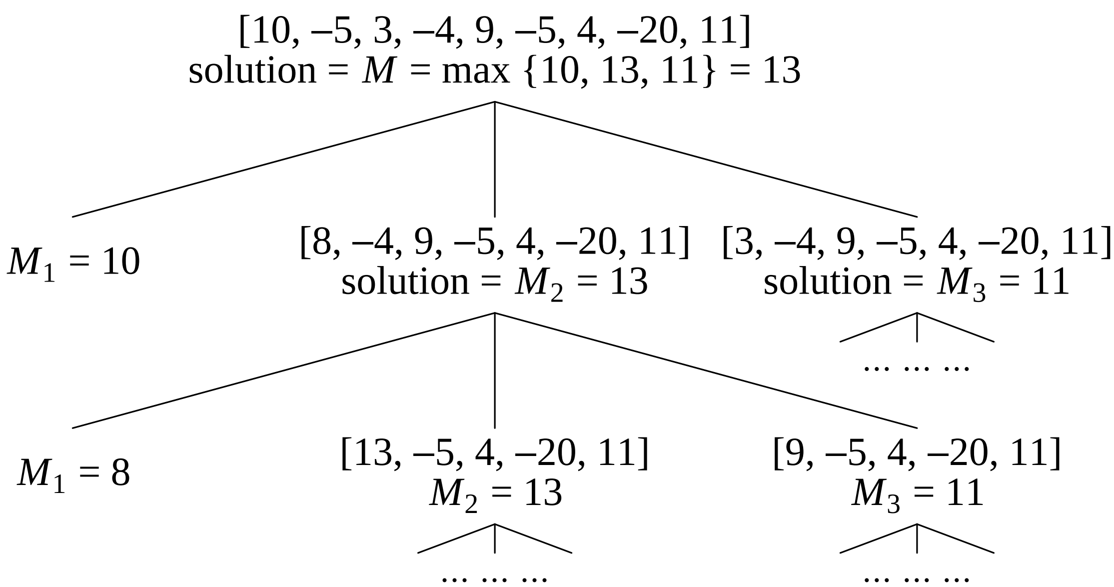
Question:

•? Characterize the solution M_2 (in a way similar to that of M_3).

•? How does this show that the Algorithm is correct?

•? How do you show that we make $2^{(N+1)/2} - 1$ recursive-calls for an input $nums[1..N]$?

AN EXAMPLE OF THE CALL-TREE IN THE RECURSION



Question:

•? Complete the above call-tree, examine it carefully, identify the redundant computations, and then restate the simplified and improved form of MAX_CONSECUTIVE_SUM. How many recursive-calls are made in the simplified Algorithm for $nums[1..N]$?

•? Let $T(N)$ = #(additions involving $nums[i]$ in the new Algorithm for an input array of size N). Show that $T(N) = T(N-2) + 2$ and T(1) = 0. (This gives $T(N) = N - 1 = O(N)$.)

•? Let $T(N)$ = #(comparisons involving $nums[i]$ in the new Algorithm for an input array of size N), Show the relationship between $T(N)$ and $T(N-1)$.

A DYNAMIC PROGRAMMING SOLUTION

Let $M(j) = \max\{S_{ij}: 1 \le i \le j\}$; here, both $i,\ j \in \{1, 3, \cdots, N\}$.

Example. For *nums*[] = [10, –5, 3, –4, 9, –5, 4, –20, 11],

	$j = 1$	$j = 3$	$j = 5$	$j = 7$	$j = 9$
	$S_{11} = 10$	$S_{13} = 8$ $S_{33} = 3$	$S_{15} = 13$ $S_{35} = 8$ $S_{55} = 9$	$S_{17} = 12$ $S_{37} = 7$ $S_{57} = 8$ $S_{77} = 4$	$S_{19} = 3$ $S_{39} = -2$ $S_{59} = -1$ $S_{79} = -5$ $S_{99} = 11$
$M(j)$	10	8	13	12	11

Observations:

$$\begin{aligned} M(1) &= nums[1]. \\ M(j+2) &= \max\{M(j)+nums[j+1]+nums[j+2],\ nums[j+2]\}. \\ M &= \max\{M(j):\ j = 1, 3, \cdots, N\}. \end{aligned}$$

Pseudocode (it does not "extend a solution" - why?):

1. $M = M(1) = nums[1]$.
2. For (j = 3, 5, ⋯, N) let $M(j) = \max\{nums[j],\ M(j-2) + nums[j-1] + nums[j]\}$ and finally $M = \max\{M, M(j)\}$.

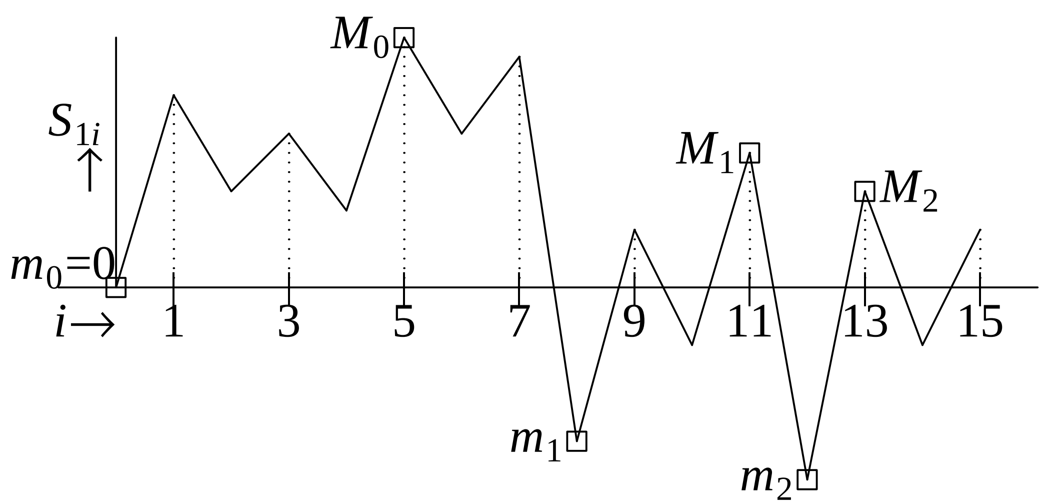
Complexity: $O(N)$.

$$\begin{aligned} \#(\text{additions involving } nums[]) &= N-1 \\ \#(\text{comparisons in computing } M(j)\text{'s}) &= (N-1)/2 \\ \#(\text{comparisons in computing } M) &= (N-1)/2 \end{aligned}$$

ANOTHER $O(N)$ METHOD

Observation:

- For $1 \le i \le j \le N$, $S_{i,j} = S_{1,j} - S_{1,(i-1)}$; here $S_{1,0} = 0$ for $i = 1$.
- If $S_{ij} = M$, then $S_{1,(i-1)} = \min\{S_{1,(i'-1)}: i' \le j\}$.



Solution Method: There are three steps.

1. Find $(i-1)$'s which can possibly give maximum S_{ij}.
   - Find the successive decreasing items $m_0 > m_1 > m_2 > \cdots > m_n$ among $S_{1,i-1}$, $i = 1, 3, \cdots, N$. (That is, m_k is the first partial-sum $< m_{k-1}$ to the right of m_{k-1}; $m_0 = 0 = S_{1,0}$.)
   - For each m_k, let i_k be corresponding i, i.e., $m_k = S_{1,(i_k-1)}$.
2. For each $i = i_k$, find the associated $j = j_k$.
   - Let $M_{k-1} = \max\{S_{1,j}: i_{k-1} \le j < i_k\} = S_{1,j_k}$ for $1 \le k \le n$; let $M_n = \max\{S_{1,j}: j \ge i_n\}$.
3. Let $M = \max\{M_k - m_k: 0 \le k \le n\}$.

(CONTD.)

A Slightly Larger Example.

$nums[i]$:	10	–5	3	–4	9	–5	4	–20	11	–6	10	–17	14
$i,\ j$:	1		3		5		7		9		11		13
$S_{1,i-1}$:	0		5		4		8		–8		–3		–10
m_k:	m_0								m_1				m_2
i_k:	1								9				13
$S_{1,j}$:	10		8		13		12		3		7		4
M_k:					M_0=13						M_1=7		M_2=5
j_k:					5						11		13

$$
\begin{array}{lll}
i_1 = 1, & i_2 = 9, & i_3 = 13 \\
j_1 = 5, & j_2 = 7, & j_3 = 13
\end{array}
$$

$$M = \max\{13 - 0,\ 7 - (-8),\ 4 - (-10)\} = 15 = S_{i_2, j_2} = S_{9,11}.$$

Question:

•? Why can’t we call this method a "method of extension"?

PSEUDOCODE vs. CODE

Characteristics of Pseudocode:

± Shows key concepts and computation steps of the Algorithm, avoiding details as much as possible.

– Avoids dependency on any specific programming language.

\+ Allows determining correctness of the Algorithm.

\+ Allows choice of proper data-structures for efficient implementation and complexity analysis.

Example. The pseudocodes below for computing the number of positive and negative items in *nums*$[1..N]$, where each *nums*$[i] \neq 0$, do not use the array-bounds. The pseudocode in (B) is slightly more efficient than the one in (A).

(A)

```
positiveCount = negativeCount = 0;
for (i=0; i<n; i++) //each nums[i] > 0 or < 0
    if (0 < nums[i]) positiveCount++;
    else negativeCount++;
```

---

1. Initialize positiveCount = negativeCount = 0.
2. Use each *nums*$[i]$ to increment one of the counts by one.

(B)

```
positiveCount = 0;
for (i=0; i<n; i++) //each nums[i] > 0 or < 0
    if (0 < nums[i]) positiveCount++;
negativeCount = n - positiveCount;
```

---

1. Initialize positiveCount = 0.
2. Use each *nums*$[i] > 0$ to increment positiveCount by one.
3. Let negativeCount = numItems – positiveCount.

Writing a pseudocode requires skills to express an Algorithm in a concise and yet clear fashion.

ANOTHER EXAMPLE OF PSEUDOCODE

Problem. Compute the size of the largest block of non-zero items in *nums*[1. . *N*].

Pseudocode:

1. Initialize maxNonZeroBlockSize = 0.
2. while (there are more array-items to look at) do:
   (a) skip zero's. //keep this
   (b) find the size of next non-zero block
   and update maxNonZeroBlockSize.

Code:

```
i = 1; maxNonZeroBlockSize = 0;
while (i <= N) {
      for (; (i<=N) && (nums[i]==0); i++); //skip 0's
      for (blockStart=i; (i<=N) && (nums[i]!=0); i++);
      if (i - blockStart > maxNonZeroBlockSize)
         maxNonZeroBlockSize = i - blockStart;
}
```

Question:

•? If there are *m* non-zero blocks, then what is the maximum and minimum number of tests involving the items nums[*i*]?

•? Rewrite the code to reduce the number of such comparisons. What is reduction achieved?

•? Generalize the code and the pseudocode to compute the largest size same-sign block of items.

ALWAYS TEST YOUR METHOD AND YOUR ALGORITHM

(a) Create a few general examples of input and the corresponding outputs.

- Select some input-output pairs based on your understanding of the problem and before you design the Algorithm.
- Select some other input-output pairs based on your Algorithm.

  Include a few cases of input that require special handling in terms of specific steps in the Algorithm.

(b) Use these input-output pairs for testing (but not proving) correctness of your Algorithm.

(c) Illustrate the use of data-structures by showing the "state" of the data-structures (lists, trees, etc.) at various stages in the Algorithm's execution for some of the example inputs.

Always use one or more carefully selected example to illustrate the critical steps in your method/Algorithm.

A DATA-STRUCTURE DESIGN PROBLEM

Problem:

- We have N switches[1..N]; initially, they are all "on".
- They are turned "off" and "on" in a random fashion, one at a time and based on the last-off-first-on policy: if switches[i] changed from "on" to "off" before switches[j], then switches[j] is turned "on" before switches[i].
- Design a data-structure to support following operations:

Print:	print the "on"-switches (in the order 1, 2, ..., N) in time proportional to M = #(switches that are "on").
Off(k):	turn switches[k] from "on" to "off"; if switches[k] is already "off", nothing happens. It should take a constant time (independent of M and N).
On:	turn "on" the most recent switch that was turned "off"; if all switches are currently "on", then nothing happens. It should take a constant time.

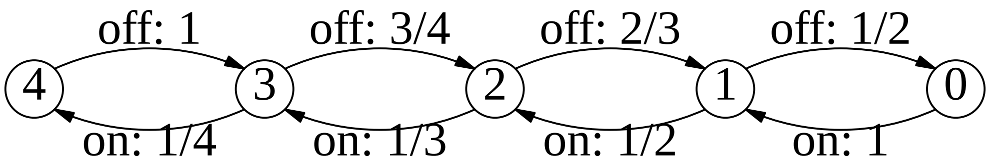
Example: Shown below are some on/off-operations (1 = on and 0 = off).

Switches[1..9]:	1	2	3	4	5	6	7	8	9
	0	1	1	0	1	0	1	1	1
Off(3):	0	1	0	0	1	0	1	1	1
Off(5):	0	1	0	0	0	0	1	1	1
On:	0	1	0	0	1	0	1	1	1

AVERAGE-TIME ANALYSIS FOR ALL SWITCHES TO BECOME OFF

Assume: If #(on-switches) = m and $0 < m < N$, then there are m+1 switches that can change their on-off status. One of them is arbitrarily chosen with equal probability to change its on-off status.

State-diagram for N = 4: state = #(on-switches).



At state m = 2:
Prob(a switch going from "on" to "off") = 2/(1+2) = 2/3.
Prob(a switch going from "off" to "on") = 1/(1+2) = 1/3.

Analysis: Let E_k = Expected time to reach state 0 from state k.

- The following equations follow from the state-diagram:

(1) $E_4 = 1 + E_3$
(2) $E_3 = (1 + E_2).3/4 + (1 + E_4).1/4 = 1 + 3.\,E_2/4 + (1+E_3)/4$
i.e., $E_3 = 1 + 2/3 + E_2$
(3) $E_2 = (1 + E_1).2/3 + (1 + E_3).1/3 = 1 + 2.\,E_1/\,3 + E_3/3$
i.e., $E_2 = 1 + 2/2 + 2/(2.3) + E_1$
(4) $E_1 = 1 + 2/1 + 2/(1.2) + 2/(1.2.3) + E_0$
i.e., $E_1 = 1 + 2/1 + 2/(1.2) + 2/(1.2.3)$ because $E_0 = 0$

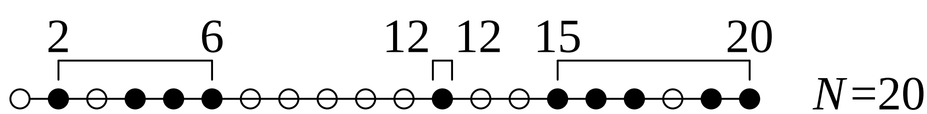
- Thus, $E_4 = 1 + (1+2/3) + (1+2/2+2/6) + (1+2/1+2/2+2/6) = 9\frac{1}{3}$.

OPTIMUM PAGE-INDEX SET FOR A KEYWORD IN A DOCUMENT

A Covering-Problem: D is a document with N pages.

- $D[i]$ = 1 means page i of the document contains one or more occurrences of a keyword; we say page i is *non-empty*. Otherwise $D[i]$ = 0 and we say page i is empty.
- m = Maximum number of references allowed in the index for the keyword. Each reference is an interval of consecutive pages; the interval $[k, k]$ is equivalent to the single page k.
- We want to find an optimal set of reference page-intervals $PI = \{I_1, I_2, \cdots, I_k\}$, $k \leq m$, where I_j's are disjoint, $\cup I_j$, $1 \leq j \leq k$, covers all non-empty pages, and $|\cup I_j|$ is minimum.

Example. The solid dots below correspond to non-empty pages. For m = 3, the optimal PI = {2-6, 12-12, 15-20}. There are two optimal solutions for m = 4 (what are they?) and one for $m \geq 5$.



Solution by Greedy Elimination:

1. Scan $D[1..N]$ to determine all 0-blocks.
2. If ($D[1]$ = 0), throw away the 0-block containing $D[1]$.
3. If ($D[N]$ = 0), throw away the 0-block containing $D[N]$.
4. Successively throw away the largest size 0-blocks until we are left with $\leq m$ blocks.

A VARIATION OF
PAGE-INDEX SET PROBLEM

- $\cup I_j$ need not cover all non-empty pages.
- Maximize Val(PI) = #(non-empty pages covered by $\cup I_j$) – #(empty pages covered by $\cup I_j$) = $|\cup I_j|$ – 2.#(empty pages covered by $\cup I_j$).

Example. Let $D[1..20]$ be as before.

○●○●●●○○○○○●○○●●●○●● N=20

- For m = 1, the optimal PI = {15-20}, with value 6 – 2.1 = 4. (For the original problem and m = 1, optimal PI = {2-20}.)
- For m = 2, there are two optimal solutions: PI = {2-6, 15-20} or PI = {4-6, 15-20}, both with value 3+4 = 7.

Algorithm?

- Finding an optimal PI is now considerably more difficult and requires a substantially different approach. (This problem can be reduced to a shortest-path problem in a digraph.)

A slight variation in the problem-statement may require a very different solution method.

Question:

•? What is the connection between this modified keyword-index problem and the consecutive-sum problem when m = 1?

•? What are some possible approaches to modify the solution method for m = 1 for the case of m = 2?

AN EXAMPLE OF THE USE OF INPUT-STRUCTURE

Problem: Find minimum and maximum items in an array $nums[1..N]$ of distinct numbers where the numbers are initially increasing and then decreasing. (For $nums[]$ = [10, 9, 3, 2], the increasing part is just 10.)

Example. For $nums[]$ = [1, 6, 18, 15, 10, 9, 3, 2], minimum = 1 and maximum = 18.

Algorithm:

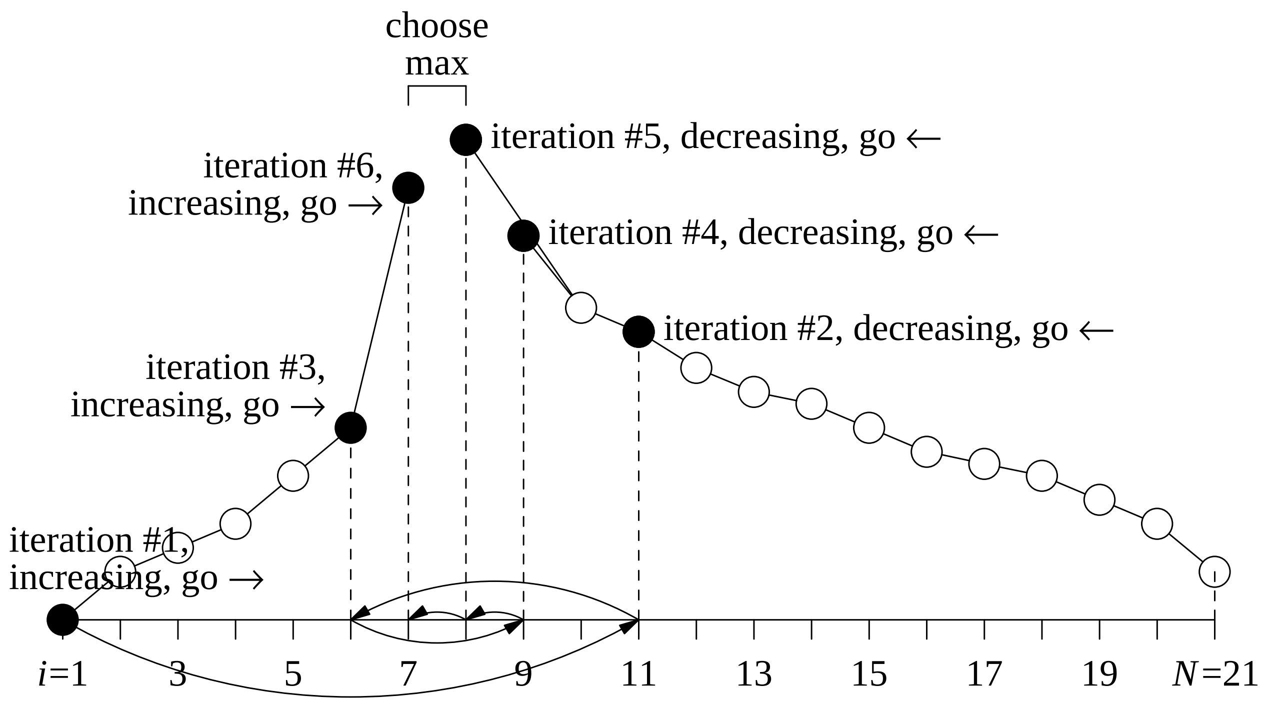
1. minimum = min $\{nums[1], nums[N]\}$.
2. If ($nums[N-1] < nums[N]$) then maximum = $nums[N]$.
3. Otherwise, starting with the initial range $1..N$ and position 1, do a binary search. In each step, we move to the mid-point i of the current range and then select the right-half of the range if the numbers are increasing ($nums[i] < nums[i+1]$) at i and otherwise select the left-half, until $nums[i]$ is larger than its each neighbor.
4. Maximum = $nums[i]$.

Complexity: #(comparisons involving $nums[]$) = $O(1)$ for minimum and $O(\log N)$ for maximum.

- This is better than $O(N)$, if we do not use the input structure.

Question: How will you use the input structure to sort the numbers $nums[1..N]$? How long will it take?

ILLUSTRATION OF BINARY SEARCH



Test for "increasing" at i: $nums[i] < nums[i+1]$

- Strictly speaking, this is *not a successive approximations* because at $(i+1)$th iteration we may be further away from the maximum than at kth (though we are closer to the maximum at $(k+2)$th iteration than at kth iteration).
- To compute maximum by the principle of *extending* the solution from the case N to $N+1$, we would proceed as:

  (1) If $(nums[N+1] > nums[N])$ then max = $nums[N+1]$.

  (2) Otherwise, apply the same method to $nums[1..N]$.

  This can take $N - 1 = O(N)$ comparisons for $nums[1..N]$.

BALANCED *be*-STRINGS

Balanced *be*-string: b = begin or '(' and e = end or ')'.

$$x = b\ b\ e\ b\ b\ e\ e\ e\ b\ e\ b\ e$$

The unique matching of each b to an e on its right without crossing

A matching with crossing

- For each initial part (prefix) x' of x, $\#(b, x') \geq \#(e, x')$, with equality for $x' = x$. In particular, x starts with b and ends with e.

  This means every b has a *matching* e to its right, and conversely every e has a matching b to its left. (Why?)

Two basic structural properties:

(1) *Nesting:*

If x is balanced, then bxe (with the additional starting b and ending e) is balanced.

(2) *Sequencing:*

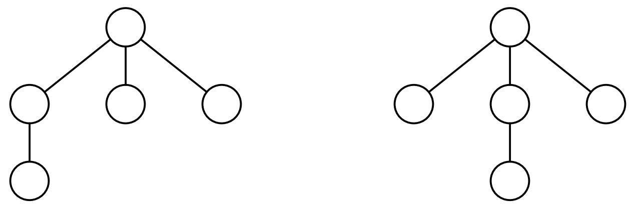
If both x and y are balanced, then xy is balanced.

All balanced *be*-strings are obtained in this way starting from λ (empty string of length 0).

Question: If x_1 and x_2 are balanced *be*-strings, $x = x_1x_2$, and $n(x)$ = #(matchings with or without crossing for x), then how do you show that $n(x_1x_2) = n(x_1)n(x_2)$?

ORDERED ROOTED TREES

- The children of each node are ordered from left to right.

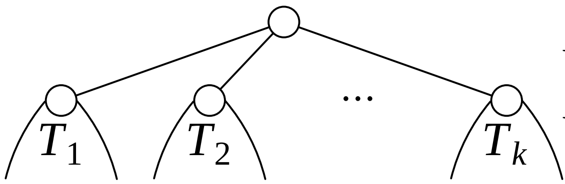


Two different ordered rooted trees; as unordered rooted trees, they are considered the same.

- The ordered rooted trees have the same two structural characteristics of *nesting* and *sequencing* as the balanced *be*-strings:
  - The subtrees correspond to nesting, and
  - The left to right ordering of children of a node (or, equivalently, the subtrees at the child nodes) corresponds to sequencing.

MAPPING ORDERED ROOTED TREES TO BALANCED *be*-STRINGS

 balString(T) = λ

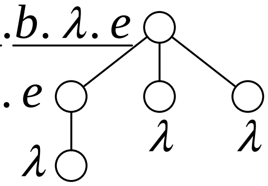


balString(T) = $bx_1e.\, bx_2e. \cdots .\, bx_ke$
where x_i = balString(T_i)

Example. Build the string *balString(T)* bottom-up.

$$bbeebebe = \underline{b.\, be.\, e}.\underline{b.\, \lambda.\, e}.\underline{b.\, \lambda.\, e}$$

$$be = b.\, \lambda.\, e$$



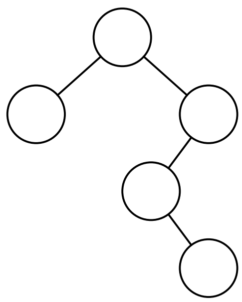
Question:

- ? What would be wrong if for the one-node tree we take beString(T) = be (instead of λ)?
- ? How will you show that balString(T_1) $\neq$ balString(T_2) for $T_1 \neq T_2$, and that balString(T) is always balanced?
- ? How will you show that for every balanced *be*-string x there is a tree T with balString(T) = x?

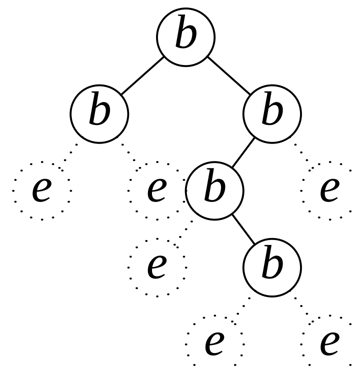
$$\boxed{\begin{array}{l}\#(\text{ordered rooted trees with } (n+1) \text{ nodes}) \\ = \#(\text{balanced } be\text{-strings of length } 2n) = \dfrac{1.\,3\cdots(2n-1)}{(n!)} \cdot \dfrac{2^n}{(n+1)}\end{array}}$$

- For length = $2n$, $\dfrac{\#(\text{balanced } be\text{-strings})}{\#(\text{all } be\text{-strings})} \to 0$ as $n \to \infty$.

MAPPING BINARY TREES TO BALANCED *be*-STRINGS



(i) A binary tree T.



(ii) After adding a child "*e*" for each null-pointer (or missing child) and labeling each original node as "*b*".

beString(T): Delete the rightmost *e* of the pre-order listing of the labels *b* and *e* in the extended tree.

For the above T, the pre-order listing gives *bbeebbebeee* and beString(T) = *bbeebbebee*.

Question:

•? If n = #(nodes in T), then how many news nodes are added?

•? What is the special property of the new binary tree?

•? In what sense the pre-order listing *bbeebbebeee* is almost balanced? How will you prove it?

•? How is beString(T) related to beString(T_1) and beString(T_2), where T_1 and T_2 are the left and right subtrees of T?

•? How is the notion of nesting and sequencing accounted in beString(T)?

GENERATING BALANCED *be*-STRINGS

Problem: Compute all *balanced be*-strings of length $N = 2k \geq 2$.

Example: Input: $N = 4$; Output: {bbee, bebe}.

~~bbbb~~	~~bbbe~~	~~bbeb~~	bbee
~~bebb~~	bebe	~~beeb~~	~~beee~~
~~ebbb~~	~~ebbe~~	~~ebeb~~	~~ebee~~
~~eebb~~	~~eebe~~	~~eeeb~~	~~eeee~~

Only 2 out of $2^N = 16$ strings of $\{b, e\}$ are balanced.

Idea: Generate all 2^N *be*-strings of length N and eliminate the unbalanced ones.

Algorithm BRUTE-FORCE:

Input: $N \geq 2$ and even.
Output: All balanced *be*-strings of length N.

1. Generate all strings of $\{b, e\}$ of length N.
2. Eliminate the *be*-strings that are not balanced.

Complexity:

- $O(N.\, 2^N)$ for step (1).
- $O(N)$ to verify balancedness of each *be*-string in step (2).
- Total = $O(N.\, 2^N)$.

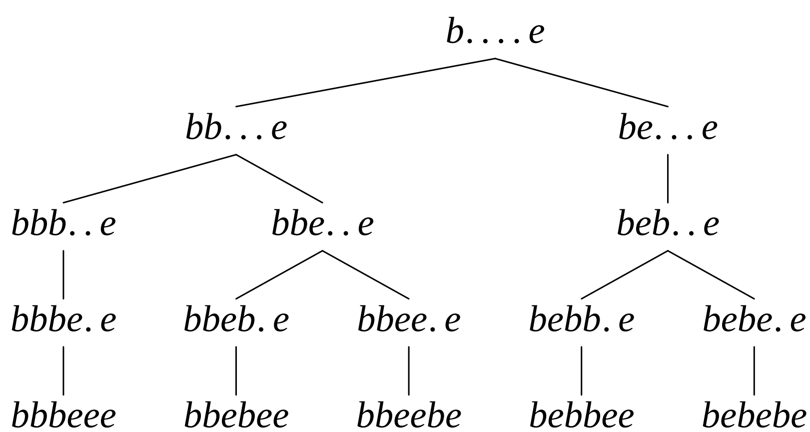
A BETTER METHOD BY USING THE OUTPUT-STRUCTURE

Idea: Generate only the balanced *be*-strings using their structure.

(1) Structure within a balanced *be*-string

(2) Structure among balanced *be*-strings of a given length N.

Ordered-Tree of Balanced *be*-strings: For $N = 6$.



This structure is suitable to compute all balanced *be*-strings of a given length by recursion, where the recursive call-tree follows the above tree-structure.

- The string at a non-terminal node is the part common to all balanced *be*-strings below it.
- The children of a non-terminal node correspond to filling the left-most empty position by b or e.
- A node has a single child = b if number of b's and e's to the left of the position are equal; a node has a single child = e if all b's are used up.
- Otherwise, it has two children (one for b and one for e).
- Terminal nodes are balanced *be*-strings in the lexicographic (dictionary) order from left to right.

DEVELOPING THE PSEUDOCODE

General Idea:

(1) Recursive Algorithm; each call generates a subtree of the balanced *be*-strings and prints those at its terminal nodes.

(2) The initial call starts with the *be*-string having its first position = '*b*' and the last position = '*e*'.

Data-structure: $beString[1..N]$
Initial Parameters: *beString*

Initial Pseudocode for GenBalStrings(beString):

1. If (no child exist, i.e., no blanks in *beString*), then print *beString* and stop.
2. Otherwise, create each childString of *beString* and call GenBalStrings(childString).

Additional Parameters: firstBlankPosn (= 2 in initial call)

First refinement for GenBalStrings(*beString*, firstBlankPosn):

1. If (firstBlankPosn = N), then print *beString* and stop.

2.1. Let numPrevBs = #(*b*'s before firstBlankPosn) and numPrevEs = #(*e*'s before firstBlankPosn).

2.2. If (numPrevBs < $N/2$), then *beString*[*firstBlankPosn*] = '*b*' and call GenBalStrings(*beString*, firstBlankPosn+1).

2.3. If (numPrevBs > numPrevEs), then *beString*[*firstBlankPosn*] = '*e*' and call GenBalStrings(*beString*, firstBlankPosn+1).

FURTHER REFINEMENT

Additional Parameters: numPrevBs
Second refinement:
GenBalStrings(*beString*, firstBlankPosn, numPrevBs):

1. If (firstBlankPosn = N), then print beString and stop.

2.1. Let numPrevEs = #(*e*'s before firstBlankPosn).

2.2. If (2*numPrevBs < N) then *beString*[*firstBlankPosn*] = '*b*' and call GenBalStrings(*beString*, firstBlankPosn+1, numPrevBs+1).

2.3. If (numPrevBs > numPrevEs), then *beString*[*firstBlankPosn*] = '*e*' and call GenBalStrings(*beString*, firstBlankPosn+1, numPrevBs).

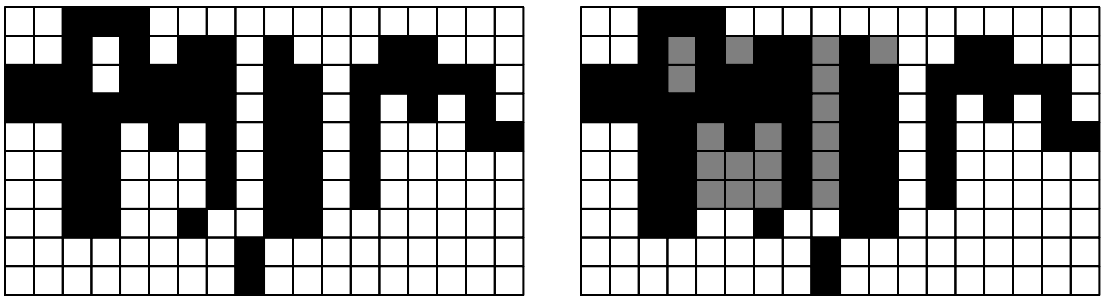
Implementation Notes:

- Make *beString* a static-variable in the function instead of passing as a parameter.
- Eliminate the parameters firstBlankPosn and numPrevB by making them static variable in the function, and use the single parameter length.
- Eliminate the variable numPrevEs (how?).
- Update firstBlankPosn and numPrevBs before and after each recursive call as needed. Initialize the array *beString* when firstBlankPosn = 1 and free the memory for *beString* before returning from the first call.

```
//cc genBalBeStrings.c (contact kundu@csc.lsu.edu for
//comments/questions)
//This program generates all balanced be-strings of a given
//length using recursion.  One can improve it slightly to
//eliminate the recursive calls when "length == 2*numPrevBs".

 #include <stdio.h>

void GenBalBeStrings(int length) //length > 0 and even
{ static char *beString;
  static int firstBlankPosn, numPrevBs;
  if (NULL == beString) {
     beString = (char *)malloc(length+1, sizeof(char));
     beString[0] = ’b’; beString[length-1] = ’e’;
     beString[length] = ’\0’; //helps printing
     firstBlankPosn = numPrevBs = 1;
  }
  if (length-1 == firstBlankPosn)
     printf("beString = %s\n", beString);
  else { if (2*numPrevBs < length) {
            beString[firstBlankPosn++] = ’b’;
            numPrevBs++;
            GenBalBeStrings(length);
            firstBlankPosn--; numPrevBs--;
         }
         if (2*numPrevBs > firstBlankPosn) {
            beString[firstBlankPosn++] = ’e’;
            GenBalBeStrings(length);
            firstBlankPosn--;
         }
       }
  if (1 == firstBlankPosn)
     { free(beString); beString = NULL; }
}

int main()
{ int n;
  printf("Type the length n (even and positive) ");
  printf("of balanced be-strings: ");
  scanf("%d", &n);
  if ((n > 0) && (0 == n%2))
     { GenBalBeStrings(n); GenBalBeStrings(n+2); }
}
```

FINDING A BEST RECTANGULAR APPROXIMATION TO A BINARY IMAGE

Example. Black pixels belong to objects; others belong to background. Let B = Set of black pixels.



(i) An image I. (ii) An approximation R.

- R covers $|R - B| = 18$ white pixels (shown in grey).
- R fails to cover $|B - R| = 29$ black pixels.
- $Val(R) = 29+18 = 47$.

$$
\begin{aligned}
R &= \text{The rectangular approximation.}\\
B\Delta R &= (B - R) \cup (R - B)\text{, the symmetric-difference.}\\
Val(R) &= |(B\Delta R)|\text{, Value of } R.\\
Val(\emptyset) &= |B| = 65;\ Val(\mathrm{I}) = \#(\text{white pixels}) = 115
\end{aligned}
$$

Question: Is there a better R (with smaller $Val(R)$)?

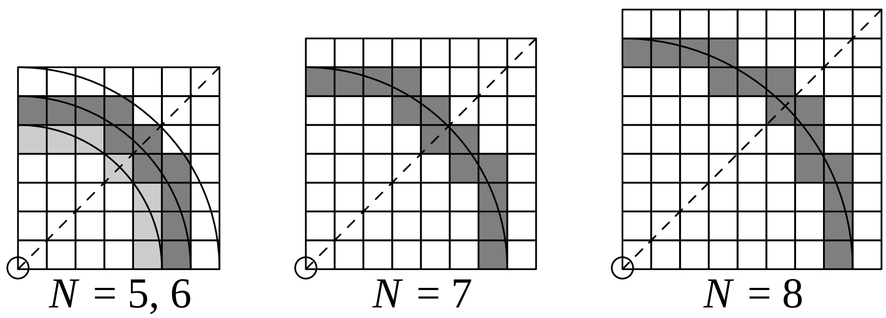
EXERCISE

1. Suppose we fix the top-row r_t and the bottom-row $r_b \geq r_t$ of R. How do you convert the problem of finding an optimal R to a maximum consecutive-sum problem?

FINDING THE BINARY IMAGE OF A CIRCLE

Problem: Find the pixels in the first quadrant belonging to the circular arc of radius N centered at (0, 0).

Example. Shown below are the binary images for N = 6 to 8.



Each circular arc is entirely contained in the pixels representing the circle.

Some Properties of Output:

(1) The lower and upper halves of the quadrant are *symmetric*.

(2) The lower-half has *at most* 2 pixels in a row (why?).

(3) For radius N, there are at most $(2N - 1)$ pixels in the first quadrant.

Notes on Designing An Algorithm:

- Exploit the output-properties (1)-(2) to find the required pixels; we need to use only integer operations.
- Some pixels that are not in the final set will be examined.

Complexity: $O(N)$;

Brute-Force Method: Complexity $O(N^2)$.

THE O-NOTATION FOR ASYMPTOTIC UPPER BOUND

Meaning of $O(n)$:

- The class of all functions $g(n)$ which are *asymptotically bounded above* by $f(n) = n$, i.e.,

$$O(n) = \{g(n): g(n) \leq c.\, n \text{ for } \textit{some} \text{ constant } c \text{ and } \textit{all large } n\}$$

  – c may depend on $g(n)$; $c > 0$.
  – "all large n" means "all $n \geq N$ for some $N > 0$"; N may depend on both c and $g(n)$.

Example. We show $g(n) = 7 + 3n \in O(n)$.

We find appropriate c and N, which are not unique.

(1) For $c = 4$, $7 + 3n \leq 4.\, n$ holds for $n \geq 7 = N$.

(2) For $c = 10$, $7 + 3n \leq 10.n$ or $7 \leq 7n$ holds for $n \geq 1 = N$.

A smaller c typically requires larger N; if c is too small, there may not exist a suitable N.

(3) For $c = 2$, $7 + 3n \leq 2.\, n$ holds only for $n \leq -7$, i.e., there is no N. This does not say $7 + 3n \notin O(n)$.

Each linear function $g(n) = A + Bn \in O(n)$.

Example. We show $g(n) = A.\, n^2 \notin O(n)$.

For any $c > 0$, $A.\, n^2 < c.\, n$ is false for all $n > c/A$ and hence there is no N.

MEANING OF $O(n^2)$

- The class of all functions $g(n)$ which are *asymptotically bounded above* by $f(n) = n^2$, i.e.,

$$O(n^2) = \{g(n)\text{: } g(n) \leq c.\, n^2 \text{ for } \textit{some} \text{ constant } c \text{ and } \textit{all large } n\}$$

  – As before, c may depend on $g(n)$ and N may depend on both c and $g(n)$.

Example. We show $g(n) = 7 + 3n \in O(n^2)$.

We find appropriate c and N; again, they are not unique.

(1) For $c = 1$, $7 + 3n \leq n^2$, i.e., $n^2 - 3n - 7 \geq 0$ holds for $n \geq (3 + \sqrt{9+28})/2$ or for $n \geq 5 = N$.

(2) In this case, there is an N for each $c > 0$.

Example. We show $g(n) = 7 + 3n + 5n^2 \in O(n^2)$.

We find appropriate c and N.

(1) For $c = 6$, $7 + 3n + 5n^2 \leq 6.\, n^2$, i.e., $n^2 - 3n - 7 \geq 0$ holds for $n \geq 5 = N$.

(2) For $c = 4$, $7 + 3n + 5n^2 \leq 4.\, n^2$, i.e., $-n^2 - 3n - 7 \geq 0$ does not hold for any $n \geq 1$. This does not say $7 + 3n + 5n^2 \notin O(n^2)$.

Each quadratic function $g(n) = A + Bn + Cn^2 \in O(n^2)$; $g(n) = n^3 \notin O(n^2)$.

SOME GENERAL RULES FOR $O(\cdot)$

(O1) The constant function $g(n) = C \in O(n^0) = O(1)$.

(O2) If $g(n) \in O(n^p)$ and c is a constant, then $c.\,g(n) \in O(n^p)$.

(O3) If $g(n) \in O(n^p)$ and $p < q$, then $g(n) \in O(n^q)$.

The pair (c, N) that works for $g(n)$ and n^p also works for $g(n)$ and n^q.

(O4) If $g_1(n), g_2(n) \in O(n^p)$, then $g_1(n) + g_2(n) \in O(n^p)$.

This can be proved as follows. Suppose that $g_1(n) \leq c_1.\,n^p$ for all $n \geq N_1$ and $g_2(n) \leq c_2.\,n^p$ for all $n \geq N_2$.

Then, $g_1(n) + g_2(n) \leq (c_1 + c_2).\,n^p$ for all $n \geq \max\{N_1, N_2\}$. So, we take $c = c_1 + c_2$ and $N = \max\{N_1, N_2\}$.

A similar argument proves the following.

(O5) If $g_1(n) \in O(n^p)$ and $g_2(n) \in O(n^q)$, then $g_1(n)g_2(n) \in O(n^{p+q})$.

Also, $\max\{g_1(n), g_2(n)\} \in O(n^q)$ assuming $p \leq q$.

Question: If $g_1(n) \leq g_2(n)$ and $g_2(n) \in O(n^p)$, then is it true $g_1(n) \in O(n^p)$?

MEANING OF $g(n) \in O(f(n))$

$$O(f(n)) = \{g(n): g(n) \le cf(n) \text{ for some } constant\ c \text{ and all large } n\}$$
$$= \{g(n): \limsup_{n \to \infty} \frac{g(n)}{f(n)} = U < \infty\}.$$

All other $\frac{g(n)}{f(n)}$ are on left $\leftarrow$ | $\rightarrow$ Only finitely many $\frac{g(n)}{f(n)}$ are on right

U $\quad$ $c = \text{U}+\varepsilon,\ \varepsilon > 0$

- We sometimes write $g(n)$ is $O(f(n))$ or $g(n) = O(f(n))$, by abuse of notation.

Examples:

(1) $7 + 3n = O(n)$ since $\limsup \frac{g(n)}{n} = \limsup \frac{7+3n}{n} = 3 < \infty$.

(2) If $g(n) \le 7 + 3\log_2 n$, then $g(n) = O(\log_2 n)$ since $\limsup \frac{g(n)}{\log_2 n} \le \limsup \left[\frac{7}{\log_2 n} + 3\right] = 3 < \infty$.

(3) If $g(n) = 7 + 3n + 5n^2$, then $g(n) = O(n^2)$ since $\limsup \frac{g(n)}{n^2} = \limsup \left[\frac{7}{n^2} + \frac{3}{n} + 5\right] = 5 < \infty$.

(4) $g(n) = 2^n \notin O(n^p)$ for any $p = 1, 2, \cdots$.

ASYMPTOTIC LOWER BOUND $\Omega(f(n))$

- We say $g(n) \in \Omega(f(n))$ if

$$\liminf_{n \to \infty} \frac{g(n)}{f(n)} = L > 0 \ (L \text{ maybe } +\infty)$$

  i.e, $\dfrac{g(n)}{f(n)} > L - \varepsilon$ or $g(n) > (L - \varepsilon) f(n)$ for all large n

  i.e, $g(n) \geq cf(n)$ for *some* constant $c > 0$ for all large n.

- We also write in that case

$$g(n) \text{ is } \Omega(f(n)) \quad \text{or} \quad g(n) = \Omega(f(n)).$$

Examples.

(1) $g(n) = 7 + 3n \in \Omega(n) \cap \Omega(1)$, but $g(n) \notin \Omega(n^2)$.

(2) $g(n) = 7 + 3n + 5n^2 \in \Omega(n^2) \cap \Omega(n) \cap \Omega(1)$, but $g(n) \notin \Omega(n^3)$.

(3) $g(n) = \log_2 n \in \Omega(1)$ but $g(n) \notin \Omega(n)$.

Question:

•? If $g(n) \in O(f(n))$, then which of the following is true: $f(n) \in O(g(n))$, $f(n) \in \Omega(g(n))$, and $g(n) \in \Omega(f(n))$?

•? If $g(n) \in \Omega(f(n))$, can we say $f(n) \in O(g(n))$?

•? State appropriate rules (Ω1)-(Ω5) similar to (O1)-(O5).

ASYMPTOTIC EXACT ORDER $\Theta(f(n))$

- We say $g(n) \in \Theta(f(n))$ if $g(n) \in O(f(n)) \cap \Omega(f(n))$

Question: Why does $g(n) \in \Theta(f(n))$ imply $f(n) \in \Theta(g(n))$?

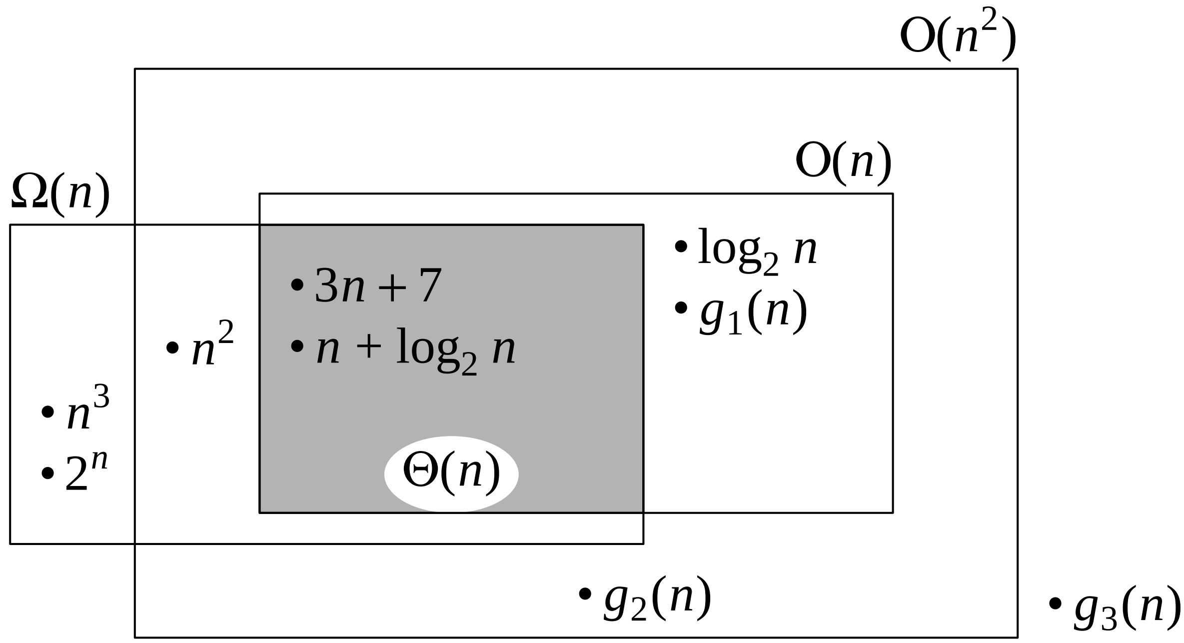
Example.

(1) $g(n) = 7 + 3n + 5n^2 \in \Theta(n^2)$, but not in $\Theta(n)$ or $\Theta(n^3)$.

(2) If $\log_2(1+n) \leq g(n) \leq 1 + \log_2 n$, then $g(n) = \Theta(\log_2 n)$.

Question: If $g_1(n) = \Theta(n^p)$, $g_2(n) = \Theta(n^q)$, and $p \leq q$, then what can you say for $g_1(n) + g_2(n)$ and $g_1(n)g_2(n)$?

COMPARISON OF VARIOUS ASYMPTOTIC CLASSES



$$g_1(n) = \begin{cases} \log_2 n, \text{ for } n \text{ even} \\ n, \text{ for } n \text{ odd} \end{cases} \qquad g_2(n) = \begin{cases} \log_2 n, \text{ for } n \text{ even} \\ n^2, \text{ for } n \text{ odd} \end{cases}$$

$$g_3(n) = \begin{cases} \log_2 n, \text{ for } n \text{ even} \\ n^3, \text{ for } n \text{ odd} \end{cases}$$

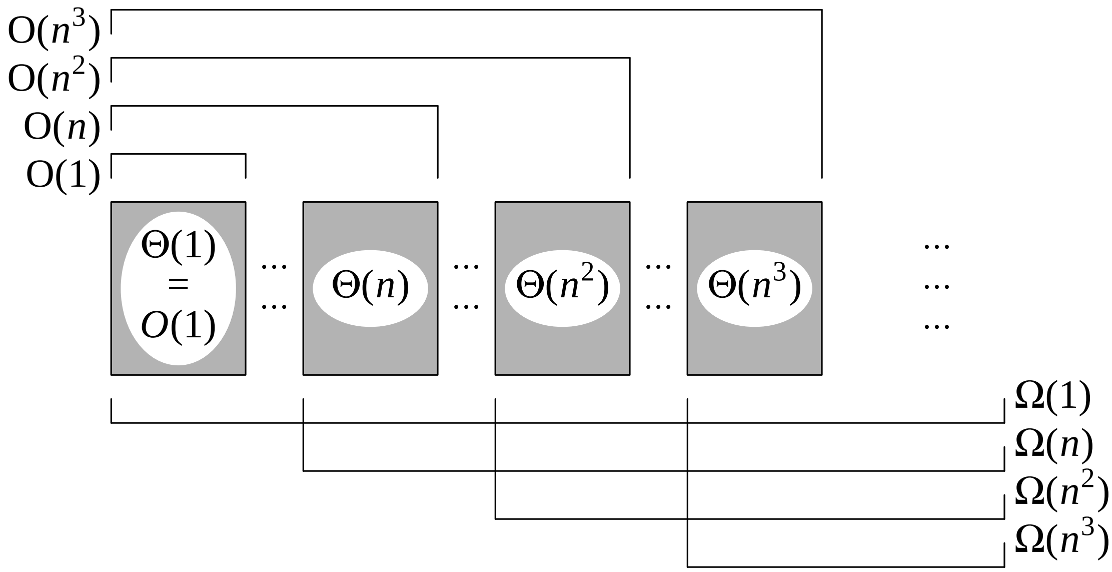
Question:

•? Place the boxes for $\Omega(n^2)$ and $\Theta(n^2)$ in the diagram above.

•? Now, place the function $g_4(n) = \begin{cases} n^{1.5}, \text{ for } n \text{ even} \\ n^{2.5}, \text{ for } n \text{ odd} \end{cases}$

Always give the best possible bound using O or Ω notation as appropriate, or give the exact order using Θ.

(CONTD.)



- There are infinitely many $\Theta(f(n))$ between $\Theta(1)$ and $\Theta(n)$ above; for example, we can have

$$f(n) = n^p,\ 0 < p < 1$$
$$f(n) = (\log n)^p,\ 0 < p$$
$$f(n) = \log^m(n),\ m = 1, 2, \cdots$$

- For each $\Theta(f(n))$ between $\Theta(1)$ and $\Theta(n)$, $\Theta(n^k.\ f(n))$ is between $\Theta(n^k)$ and $\Theta(n^{k+1})$ and vice-versa.
- $O(f(n)) = \bigcup_{g(n) \in O(f(n))} \Theta(g(n))$
- $\Omega(f(n)) = \bigcup_{g(n) \in \Omega(f(n))} \Theta(g(n))$

Question: Why don't we talk of $O(1/n)$?

ALGORITHM DESIGN vs. ANALYSIS

Four (3+1) Basic Questions on an Algorithm:

(1) What does A do – inputs, outputs, and their relationship?

(2) How does A do it – the method for computing $f(x)$.

(3) Any special data-structures used in implementing the method?

(4) What is its performance?

- Time $T(n)$ required for an input of size n (measured in some way).

  If different inputs of size n require different computation times, then we can consider:

  $T_w(n)$: the worst case (maximum) time
  $T_b(n)$: the best case (minimum) time
  $T_a(n)$: the average case time

- Similar questions on the use of memory-space.

  Since the amount of memory in use during the time $T(n)$ may vary, one can also talk about the maximum (and similarly, the minimum and the average) memory over the period $T(n)$.

1. Show the first quadrant for $N = 9$.
2. Is it true that the circles obtained in this way for various $N \geq 1$ have no pixels in common?
3. Is it true that they fill-up all the pixels?
4. Give an efficient Algorithm in a pseudocode form using the properties/structures identified above to determine the pixels on the circle of radius N. It should use, in particular, only integer arithmetic. How many pixels do you test (not all of which may be part of your answer) in determining the first quadrant of the circle?
5. Show that the number of pixels on the perimeter of the circle in the first quadrant is $2N - 1$. (Hint: if there are many pixels in a column as is the case on the right side of the first quadrant, then there are many columns with few pixels as is the case on the left of the first quadrant. Note that if we bent the line $i + j = N$ slightly, then it takes $2N - 1$ pixels to cover it.)
6. How will you create the three dimensional image of the surface of the sphere of radius N in a similar way? (Each pixel is now a small cube.)

IMPROVE THE LOGIC/EFFICIENCY IN THE FOLLOWING CODE SEGMENTS

Ignore language-specifi c issues (such as "and" vs. "&&").

1.
```
if (nums[i] >= max) max = nums[i];
```

2.
```
if (x and y) z = 0;
else if ((not x) and y) z = 1;
else if (x and (not y)) z = 2;
else z = 3;
```

3.
```
if (x > 0) z = 1;
if ((x > 0) && (y > 0)) z = 2;
```

4.
```
for (i=1; i<n; i++)
    if (i < j) sum = sum + nums[i]; //sum += nums[i]
```

5.
```
for (i=0; i<n; i++)
    if (i == j) items[i] = 0;
    else items[i] = 1;
```

6.
```
for (i=1; i<n; i++)
    for (j=1; j<n; j++) {
        diff = nums[i] - nums[j];
        if (i ≠ j) sumOfSquares += diff*diff;
    }
```

7.
```
for (i=1; i<n; i++)
    for (j=1; j<n; j++) {
        if (i == j) A[i][j] = -1;
        else if (M[i][j] >= M[j][i]) A[i][j] = 1;
        else A[i][j] = 0;
    }
```

8.
```
for (i=0; i<3*length; i++)
    printf(" ");
```

9.
```
for (i=0; i<10; i++) {
    char stringOfBlanks[3*10+1] = "";
    for (j=0; j<i; j++)
        strcat(stringOfBlanks, "   ");
    if (...) printf("%s: %d\n", stringOfBlanks, i);
    else printf("%s: ...", stringOfBlanks, ...);
}
```

TOPICS TO BE COVERED

Introductory Material:

- (1) Solution method before Algorithm - necessary & sufficient condition in rectangle inclusion

Sorting:

- (1) Review and close look at some sorting Algorithms.
- (1) Sorting non-numerical things (strings, trees, flowcharts, digraphs)
- (1) Some non-trivial application of sorting.
- (2) Heap-data structure for efficient implementation of selection-sort.

--------------------- Quiz #1 ---------------------------------

- (1) 2-3 trees: a generalization of heap.

Application of Stack: Topological Sorting:

- (1) Sorting nodes of an acyclic digraph. and finding all topological sorting.
- (1) Counting the number of topological sorting.
- (1) Converting an infix-expression to a postfix-expression using a stack and evaluating a postfix-expression using stack.
- (1) Finding longest paths

--------------------- Quiz #2 ---------------------------------

- (1) Longest increasing subsequence
- (2) Depth first search and depth first tree

Minimum Weight Spanning Tree:

- (2) Finding minimum weight spanning tree

Shortest and Longest Paths:

- (1) Find all acyclic paths and cycles from a node (undirected graph)
- (2) Finding shortest paths - Dijkstra; connection between shortest and longest paths

---------------------- Quiz #3 ----------------------------------

- (2) Finding shortest paths - Floyd

String Matching:

- (2) String matching

Huffman tree:

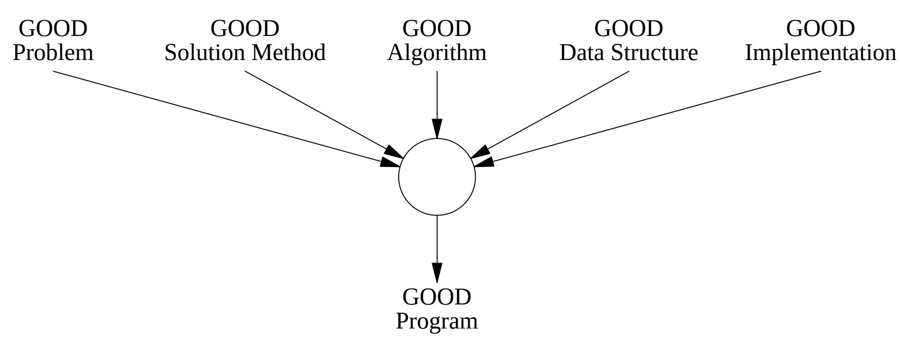
- (1) Prefix free coding and Huffman tree

---------------------- Quiz #4 ----------------------------------

DATA-STRUCTURE AND ALGORITHM ANALYSIS: APPLICATION DRIVEN

Jan 12

- I am Kundu. I want this course to be a rewarding and enjoyable experience for you so that you have a renewed sense of confidence in and love for computer science. This also means that I expect you to put a lot of effort, a full 120%.
- One of your goals for being here, I believe, is that by the end of the semester you want to become a good/expert programmer in terms of using proper data-structures and Algorithms, and you are ready to compete with other CS graduates from any other University in US or elsewhere.
- Good programmers write good (efficient and clear, not just programs that somehow produce the right output) programs, but what goes into a good program?



Good Implementation:

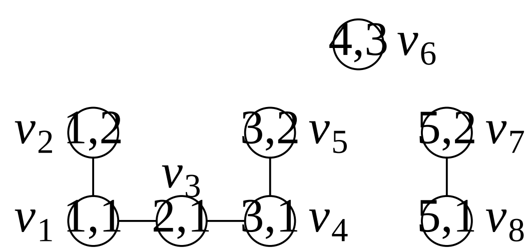
- Good choice of names for variables, functions, parameters, and files.
- Good choice of local and global variables.
- Good choice of conditions for branch-point and loops.

- To do all these good selections, you need to know some example of good Algorithms and their implementations. (We indeed learn from experience.) In this course, we are going to: (1) learn a number of interesting Algorithms and (2) practice solving some new problems using those Algorithms and their variations.

Difference between a good program/software and a good product: solves a useful problem and good interface.

- Give some example problems that the students will be able to solve by the end of semester
  - Take them from MUM-lectures; minimum energy nodes to form a connected sensor network

    Let $r_{\min}$ be the minimum r where the links $E(r) = \{(v_i, v_j) \colon d(v_i, v_j) \le r\}$ form a connected graph on the nodes.



The links $E(1)$.

  - **Question:** What is $r_{\min}$ for the set of nodes above? Give an example where $r_{\min} \ne \max\{$distance of a node nearest to v_i: $1 \le i \le N\}$. (If $r_{\min}$ always equals the maximum, then what would be an Algorithm to determine $r_{\min}$?)
  - Find the largest number of points $P_i = (x_i, y_i)$ that can be roped in with a rope of length L.

Some Critical-Thinking Questions On Selection Sort:

For the questions below, it suffices to consider the input to be a permutation of {1, 2, ⋯, *numItems*}.

- •? Is it true that the number of upward data-movements are always the same as the number of downward data-movements?
- •? If we know that *n* of the data-items are out of order, what is the maximum and minimum number of data-movements? Show the example inputs in which this maximum and minimum are achieved.
- •? In what sense the Selection Sort minimizes data-movement?
- •? How many data-comparisons are made in finding the *i*th smallest item? What is the total number of data-comparisons? Does it depend on the input?
- •? Suppose a series of related exchanges are of the form items[i1] and items[i2], items[i2] and items[i3], ... , items[i(k-1)] and items[ik]. Then argue that the indices {i1, i2, ..., ik} form a cycle in the permutation. Note that the exchange operations in the different cycles may be interleaved.

An Example of Creative Thinking Related to Selection Sort:

- •? If we view Selection Sort as a way of "filling the places by the right items", then give a high level pseudocode of an Algorithm that fits the description "finding and putting each item in the right place".
- •? Can you think of another variant of selection-sort?

In bubble sort is it true that if a data-item moved up, then it is never moved down? How abot if we interchange "up" and "down" in the above sentence?

- Concept of Sorting
  - An example: $\langle 7, 2, 6, 1\rangle$ becomes $\langle 1, 2, 6, 7\rangle$ after sorting in increasing order. Lexicographic ordering of {bat, but, cap, happy, life}.

    Sort names in a printed voter/airline-passenger list to quickly locate if a given name is in the list. (For electronic copy, it is not necessary to sort it; a binary search list is more suitable.) The words in a dictionary are sorted as are index-words at the end of a book.
  - How do you define the sorting problem?

    > Given a set of n things t_j, $1 \le j \le n \cdots$, which are mutually comparable in some way (i.e., there is a linear order among them), find the arrangement as in: $t_1 < t_2 < \cdots < t_n$, i.e., find the smallest item, the second smallest item, and so on.

  - Strings have linear ordering among them (the lexicographic ordering), they can be sorted: but < cat < cup < heavy < life.
  - What kinds of things cannot be sorted? If there is no linear ordering as in the case of subsets of a set. For $S_1 = \{a, b\}$ and $S_2 = \{b, c\}$, we have both S_1 and $S_2 \subset S = \{a, b, c\}$ but $S_1 \not\subset S_2$ and $S_2 \not\subset S_1$. Thus, $\{S_1, S_2\}$ cannot be sorted under the subset-relation. (Indeed, we can simply declare that $S_1 < S_2$ is the sorting, but others need not accept this.)
  - What is an application (distinction between "use" and "application").

Jan 14:

- How do we compute the partial sums d_1, $(d_1 + d_2)$, $(d_1 + d_2 + d_3)$, $\cdots$, $(d_1 + d_2 + \cdots + d_n)$ most efficiently?
- How would we modify the code below to count the number of time the condition C is evaluated and likewise read and write counts of x and y (use variables xReadCount, xWriteCount, etc)?

```
...
if (C) z = 0;
else z = 1;
...
```

```
...
if (x < 3)
   y = x + 5;
...
```

- Discussion on the program below for generating successive binary string and its variations with numOnes (see the other file binString-prog.t).
  - The successive calls to NextBinString(3) produces 000, 001, 010, 011, 100, 101, 110, 111, and NULL.
  - The next binary string of 0110001011 is 0110001100, and its next is 0110001101.
  - Pseudocode:

1. Find the rightmost 0 (finding from right is faster since most change take
2. If (0 is found) then make that 0 to 1 and all 1's to its right 0.
3. Otherwise stop.

  - The two key issues needed to develop the Algorithm are (this is true for this case, and the case where the number of 1's is fixed and also in the case generating next permutation):

    (1) where do we start making the change, and
    (2) what is the change

    This abstraction ties together all three next-item generation Algorithms.

– NextBinString program

```
//use this function with same length repeatedly to generate all binary strings of that length
//until the return value is NULL; only then use a different length, if desired, or use the same
//length to repeat the cycle.
char *NextBinString(int length) //length > 0
{ static char *binString=NULL;  //arraySize=length+1; 1 for end-of-string to help print binString
  int i;
  if (!binString) {
     binString = (char *)malloc((length+1) * sizeof(char));
     for (i=0; i<length; i++)
         binString[i] = '0';
     binString[length] = '\0';
  }
  else { for (i=length-1; i>=0; i--) //find position of rightmost 0
             if ('0' == binString[i]) break;
         if (i >= 0) { //update binString
            binString[i] = '1';
            for (i=i+1; i<length; i++) binString[i] = '0';
         }
         else binString = NULL; //reset for next call of NextBeString
       }
  if (binString)
     printf("binString: %s\n", binString);
  return(binString);
}
```

– Pseudocode for finding the next binary string of given length and number of ones.

1. Find the rightmost 01 (finding from right is faster since most change take place on the rightside).
2. If (found) then make that 01 to 10 and all move 1's to its right to rightmost places.
3. Otherwise stop.

• Show a pseudocode and a piece of C/Java-code for finding the rightmost "00" in a binaryString[0..(length-1)]. Keep things as clean and efficient as possible.

1. Find rightmost 0.
2. If (the previous item is 1), then go back to step (1) and start the search from the left of the current position.

The implementation below, is cleaner than the one following it in terms of logic and is equally efficient.

```
i = length;
do { for (i=i-1 ; i>0; i--)
         if (0 == binString[i]) break;
} while (1 == binString[--i]);
```

```
for (i=length-1; i>0; i--) //warning: body of for-loop updates i
    if (0 == binString[i]) && (0 == binString][--i]) break;
```

1. **Bonus:** Let $R(W, H)$, where $W \geq H > 0$, denote a rectangle with width W and height H. How will you determine if a rectangle $R_1(W_1, H_1)$ can be placed completely inside another rectangle $R_2(W_2, H_2)$, and if so how can you find at least one an actual placement (there can be more than one ways to place R_1 inside R_2). (Note that the problems of placing a circle inside a rectangle and of placing a rectangle inside a circle are easy.) First, show that if $D_1 = D_2$, where D_i is the length of the diagonal of R_i, then the only way R_1 can be placed inside R_2 is $R_1 = R_2$, i.e., $W_1 = W_2$ (and hence $H_1 = H_2$).

2. **Homework:** Consider again the car-repair problem, where now we have two repair-men. Suppose we have four cars C_1, C_2, C_3, and C_4 with the repair-times 7, 2, 6, and 1 respectively. Show all possible repair-schedules (who repairs which cars and in what order) which has the minimum total lost-service time; the person who repairs C_1, call him A and call the other person B.

   – What do you think (guess) is the general rule for creating the best repair-schedule?

   – If there are $2n$ cars and two repair men, what is the number of optimal repair-schedules?

3. **Homework:** How to compute the successive permutations of $\{1, 2, \cdots, n\}$ in the lexicographic order?

   Given two permutations $p = (p_1, p_2, \cdots, p_n)$ and $q = (q_1, q_2, \cdots, q_n)$, we say $p < q$ if for the leftmost position i where $p_i \neq q_i$, we have $p_i < q_i$. The lexicographic ordering of the permutations for $n = 3$ is

   $$(1, 2, 3) < (1, 3, 2) < (2, 1, 3) < 2, 3, 1) < (3, 1, 2) < (3, 2, 1)$$

   For $n = 9$, what is the first permutation p that starts with (4, 3, 1, 9, 6, ...) and what is the one next to it, and the one next to that? Also, what is the one previous to p? Show the pseudocode for computing the permutation which is next to a given permutation $(p_1, p_2, \cdots, p_n)$.

Jan 21

- Discuss homework problems for NextPermutation(numItems), two-person car repair scheduling, rectangle placement, and programming of NextBinString(length, numOnes).
- The Algorithms for NextBinString(length), NextBinString(length, numOnes), and NextPermutation(numItems) have the following common form although they differ in the details of each of the three steps.

  1. Find the rightmost place where a change occurs.
  2. Make the change at that place
  3. Make the change to its right.

- Problem random generation of a binary string of length n:

  1. Save all the strings in a file.
  2. Create a random number $0 \leq k <$ numStrings.
  3. Select kth string.

  Problem too much time to compute all of them and too much storage to save. Better approach

  Compute successive bits of the string with suitable probability.

- Algorithm for random permutation;

  1. For (each $0 \leq i <$ numItems) choose randomly an item from $\{0, 1, 2, \cdots, n-1\}$ which is different from previous items.

  An implementation (very inefficient):

```
permutation[0] = random()%numItems;
for (i=1; i<numItems; i++) {
    do { item = random()%numItems;
         for (j=0; j<i; j++)
             if (permutation[j] == item) break;
       } while (j < i);
    permutation[i] = item;
}
```

  Better idea: keep track of remaining items and choose one at random from the remaining items.

- **Homework+Program:** Find a better way and compare the average number of times random() is called for generating 10^6 cases of random permutations for numItems = 50. Also, show the details for numItems = 4 and 5 different runs of RandomPermutation(4), show the sequence of random items generated by the brute-force method as each new permutation[i] is determined, the final permutation, and the counts of random() in each case.

- A variation of car-repair problem that can be solved in the same way: we have customers lined up in a shop to get some service, and we want to serve them in a way that reduce their total weight time.

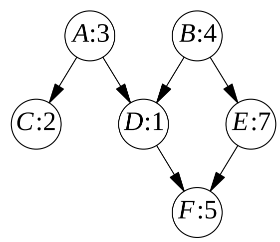
  Now we can introduce some probability that a customer may leave at any time based on an (say) exponential distribution, i.e., a customer leaves within a time period t with probability $1 - x^t$ and the probability x^t that he does not leave (where $x = e^{-\lambda}$ for some $\lambda > 0$, i.e., $0 < x < 1$). Then what is the best order-of-service to maximize the profi t, i.e., the amount of service that can be provided.

  - If we have just two customers with $d_1 = 2$ and $d_2 = 6$, then the processing order $\langle C_2, C_1\rangle$ is optimal with the expected extra return $[8x^6 + 6.(1 - x^6)] - [8x^2 + 2.(1 - x^2)] \geq 0$ for all $0 < x = e^{-\lambda} < 1$.

- If you have two repair-men, then what is the optimal distribution of the work between them for the d_i-values {2, 6, 7, 11, 13}?

- A generalization to the case of a precedence constraints among the tasks.

  Suppose I have 6 pieces of tools $\{A, B, \cdots, F\}$ in my machine shops which need repair. Also, some of the tools themselves are needed to repair some of the other tools as shown below; here, tool A is needed to repair both the tools C and D (as indicated by the links (A, C) and (A, D) respectively). The number next to each node is the time needed to repair that tool.



  Here two of the many possible repair-sequence are: $\langle A, B, C, D, E, F\rangle$ and $\langle B, A, C, D, E, F\rangle$.

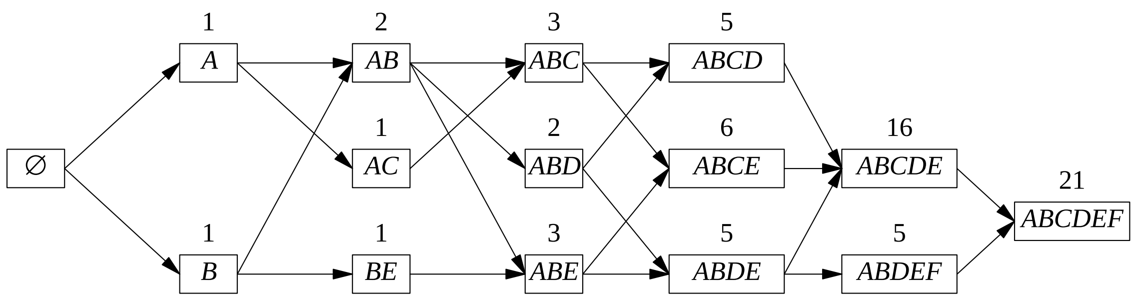
  Here, the best repair-sequence is: $\langle A, C, B, D, E, F\rangle$.

  You always repair the tool which has no precedence constraint (i.e., is not waiting for some other tool to be repaired) and which has the smallest repair time.

Set of tools ready for repair	A: 3, B: 4	B: 4, C: 2	B: 4	D: 1, E: 7	E: 7	F: 5
Best choice	A	C	B	D	E	F

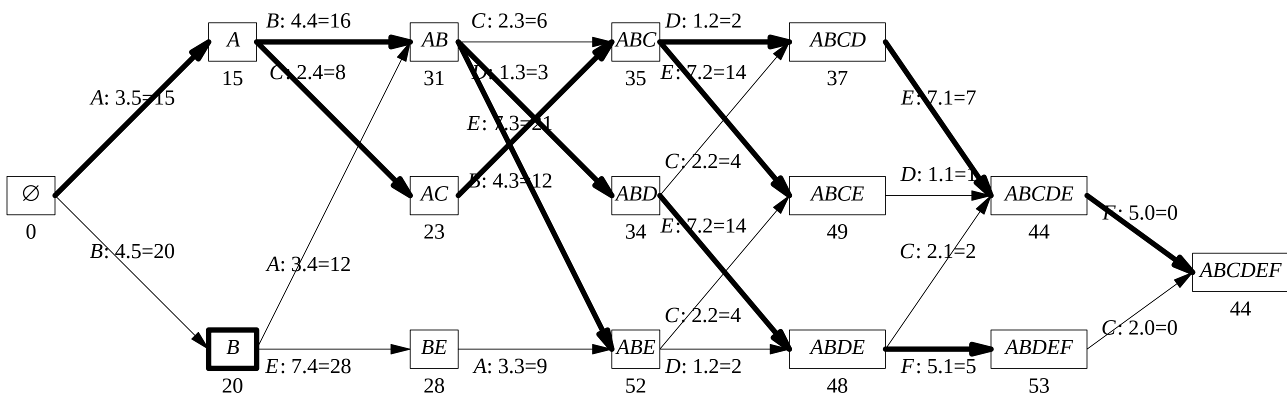
- **Homework:** Find 5 different repair-sequences and the associated total lost-time for each of them. How many repair-sequences are there?

  - How do you compute the number of possible repair-sequences for a general precedence digraph;



  - We can use a shortest-path computation on the digraph below to get the best repair-sequence. The link (S_i, S_j) connecting node S_i to S_j corresponds to the repair job for tool $T_k \in S_j - S_i$, and the cost of the link is $d_k.(N - |S_j|)$, which is the total contribution to the delay for repair of the remaining $N - |S_j|$ tools.

Below each node we show the shortest-path length from the node ∅.



- What is the basic assumption in sorting: there is a linear order among the items to be sorted.
  - We have seen linear ordering og numbers, strings, and permutations.
  - Can we use the linear order of binary strings of length 3 to provide a linear order on subsets of $\{a, b, c\}$? What happens if we associate a with the leftmost bit, b with middle bit, and c with rightmost bit and map $010 \to \{b\}$, $101 \to \{a, c\}$, and so on giving

$$\{c\} < \{b\} < \{a\} < \{b, c\} < \{a, c\} < \{a, b\} < \{a, b, c\}.$$

- Following is a pseudocode for Insertion-sort Algorithm, where we have used recursion; here, numItems = #(items to be sorted) = size(input array). Here, you know nothing of the final result until the very end.

1. If (numItems = 1) then stop.
2. Otherwise, sort the first (numItems-1) items from the input and insert the last item.

For the initial input array [7, 2, 6, 1], the recursion proceeds as follows:

[7, 2, 6, 1] $\to$ insert 1 in [2, 6, 7]: [2, 6, 7, 1] $\to$ [2, 6, 1, 7] $\to$ [2, 1, 6, 7] $\to$ [1, 2, 6, 7]

↓ ↑[2, 6, 7]

[7, 2, 6] $\to$ insert 6 in [2, 7]: [2, 7, 6] $\to$ [2, 6, 7]

↓ ↑[2, 7]

[7, 2] $\to$ insert 2 in [7]: [7, 2] $\to$ [2, 7]

↓ ↑[7]

[7]

Lots of data-movements: [7, 2, 6, 1] $\to$ [2, 7, 6, 1] $\to$ [2, 6, 7, 1] $\to$ [2, 6, 1, 7] $\to$ [2, 1, 6, 7] $\to$ [1, 2, 6, 7].

Worst case: $1 + 2 + 3 + \cdots + (n-1) = \dfrac{n(n-1)}{2}$, arising for input [7, 6, 2, 1]; same for the number of comparisons. Best case: #(data movements) = 0 and #(comparisons) = $n - 1$.

Indeed, you can use a for loop:

```
1. For (i = 1 to numItems – 1)
       insert nums[i] among nums[0..i-1] so that nums[0..i] are sorted.
```

Insertion: pseudocode and implementation (where steps (1)-(2) are combined):

Pseudocode:

1. Find the position $0 \le j \le i$ for nums[i].
2. If ($j < i$) then move items in nums[$j..(i-1)$] one position right (save nums[i] before this) and place nums[i] in position j.

Implementation:

```
for (j=i-1; j>=0; j--)
    if (nums[j+1] > nums[j]) break; //>=
    else interchange nums[j+1] and nums[j];
```

- Selection Sort: Here, you do know part of the final output at the intermediate phases (unlike insertion-sort). This is iterative from the output point of view while insertion-sort iterative from an approximation viewpoint). The recursive form below applies recursion after some preliminary computation (cf. insertion-sort)

  1. If (numItems = 1) do nothing.
  2. Otherwise, Find the largest item and interchange it with the items[numItems-1], if necessary, and then apply the method recursively to items[0..numItems-2].

  For input array [2, 7, 1, 6], the recursion proceeds as shown below.

  [2, 7, 6, 1] $\rightarrow$ max-item = nums[1] = 7
  $\rightarrow$ after interchange [2, 1, 6, 7] $\qquad\qquad \rightarrow$ [1, 2, 6, 7]

  $\downarrow \qquad\qquad \uparrow$ [1, 2, 6]

  [2, 1, 6] $\rightarrow$ max-item = nums[2] = 6
  $\rightarrow$ (no interchange) [2, 1, 6]

  $\downarrow$ ...

  Few data-movements here: maximum of 1 per each recursion's own direct computation. Worst case: $n-1$. The number of comparisons is always $(n-1) + (n-2) + \cdots + 3 = 2 + 1 = \dfrac{n(n-1)}{2}$.

- Merge sort:

  1. If (numItems == 1) do nothing.
  2. Otherwise divide input into two equal (or close to equal) halves (first half size ≤ second half size). and sort each part.
  3. Merge the two sorted part.

  Show with an example of 8 items that merging may take longer if we divide into 2/3 and 1/3 parts instead of into 1/2 and 1/2.

  An extreme case of this division into first $n-1$ and the last item gives insertion sort.

- **Homework.** For the input nums[0..3] = [7, 2, 6, 1], show the sequence of successive value-pairs compared in the insertion-sort Algorithm (instead of writing the pair as (nums[0], nums[1]), write (7,2) and not (2, 7)). Also, show the whole nums-array every time some data-movement takes place in the array. In what input situation, we have the maximum number of data-movements (give an example for an array of 5 items)? In what input situation, we have the maximum number of comparisons (give example)?

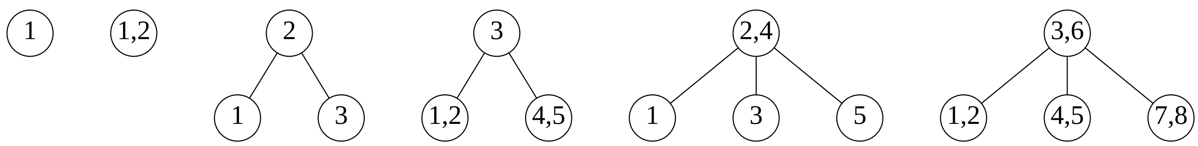
- **Homework.** Give a recursion-based pseudocode (not C-code) for insertion-sort. Imagine that you are doing this to develop a program later for the function InsertionSort(int *nums, int numItems). Show the successive calls that will be made for the initial input nums[0..3] = [7, 2, 6, 1].

- **ONUS.** Use the above piece of code to create a function GenRandomPermutation(int numItems), which prints all the successive random items generated and putting a '*' next to an item when it becomes part of the permutation (you can put all the values of item in a line). It should also count the total number of random numbers generated in creating a random permutation. Show the detailed output for 5 calls to the function for numItems = 4. Finally, show the average value of count for 5 calls to the function for numItems = 100000 (don't show the details of random items generated for these permutations).

- **Homework:** Show a similar pseudocode for a recursive form of Selection-sort Algorithm and show its call-return tree and the computations for the input [7, 2, 6, 1].

Feb 09

- 2-3 tree: An ordered rooted tree, whose nodes are labeled by items from a linear ordered set (like numbers) with the following properties (T.1()-(T.3) and (L.1)-(L.3). Shown below are few small 2-3 trees.



min number of labels = 3 | max number of labels = 5 | min number of labels = 5 | max number of labels = 8

(T.1) Each node has exactly one parent, except the root

(T.2) It is height balanced: all terminal nodes are at the same distance from the root.

(T.3) Each non-terminal node has either 2 children or 3 children.

(L.1) A node x with 2 children has one label, $label_1(x)$, with the properties:

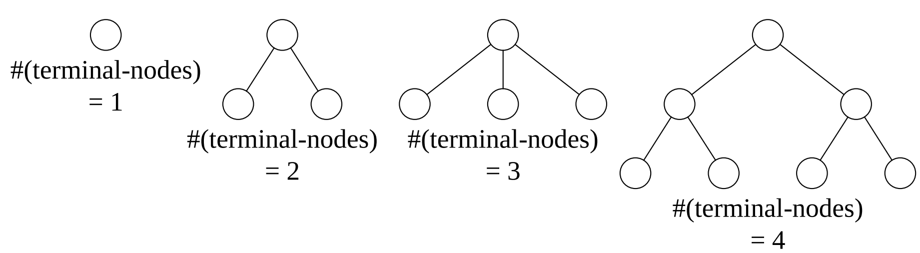
$$\text{labels}(T_L(x)) < label_1(x) \quad \text{where } T_L(x) \text{ is left-subtree at } x,$$
$$label_1(x) < \text{labels}(T_R(x)) \quad \text{where } T_R(x) \text{ is right-subtree at } x$$

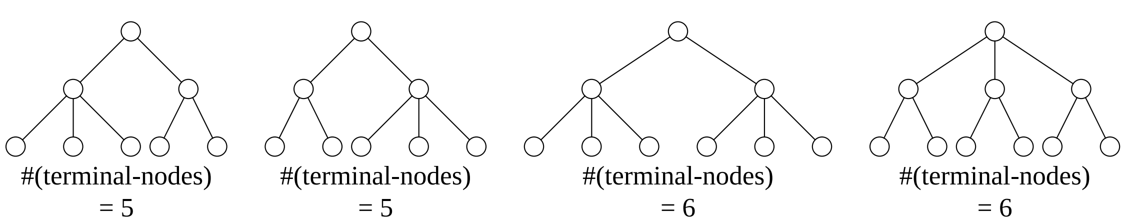
(L.2) A node x with 3 children has two labels, $label_1(x) < label_2(x)$, with the properties:

$$\text{labels}(T_L(x)) < label_1(x) \quad \text{where } T_L(x) \text{ is left-subtree at } x,$$
$$label_1(x) < \text{labels}(T_M(x)) < label_2(x) \quad \text{where } T_M(x) \text{ is middle-subtree at } x$$
$$label_2(x) < \text{labels}(T_R(x)) \quad \text{where } T_R(x) \text{ is right-subtree at } x$$

(L.3) A terminal node may have one label or two labels.

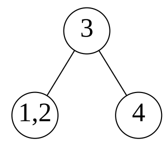
- Example of 2-3 trees with different number of terminal nodes:



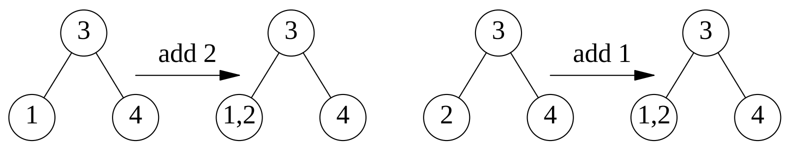


Feb 11

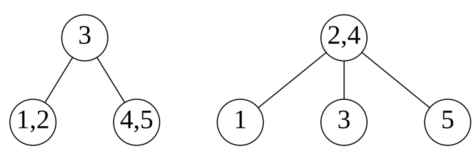
- How many ways can the 2-3 tree on left can arise? There are 12 ways, i.e., 12 possible input sequences (permutations of {1, 2, 3, 4}) that gives this 2-3 tree. The only other 2-3 tree with the labels {1, 2, 3, 4} is also obtained in 12 ways, covering 12 + 12 = 24 = 4! permutations of {1, 2, 3, 4}.



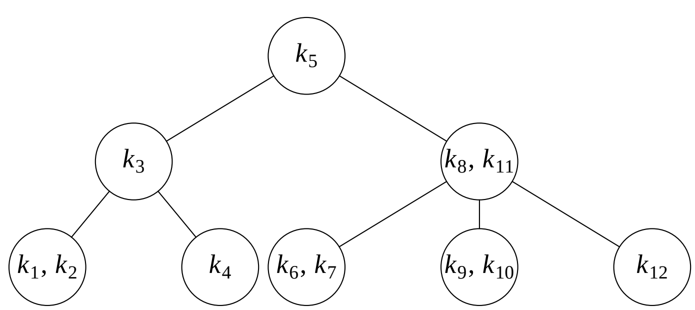
- It came from a 3 node 2-3 tree (of the same shape) – why? The 3-node 2-3tree can be only one of the following, and by adding 2 to the first tree and 1 to the second tree we get the above tree.



- How many ways we get the first 2-3 tree above? there are 6 ways, i.e, from 6 different permutations of {1, 3, 4} and they all come from 3 different one-node 2-3 tree.
- **Homework:** Show all possible structure of 2-3 tree with 5 terminal nodes and 6 terminal nodes. Also, label the nodes of each with the numbers 1, 2, 3, ⋯ for the case of minimum number of data items in the nodes and also for the case of maximum number of data items in the nodes.
- **Homework.** Show that the following 2-3 trees arise from 48 and 72 (total = 120 = 5!) permutations of {1, 2, ⋯, 5}. In each case, they come from a 3-node 2-3 tree.



- **Homework.** What additional information we could at each node of 2-3 tree if we want to quickly find the key-value of the *i*th smallest item? Show how you will use that to determine the 9th item in the following 2-3 tree ($k_1 < k_2 < \cdots$).

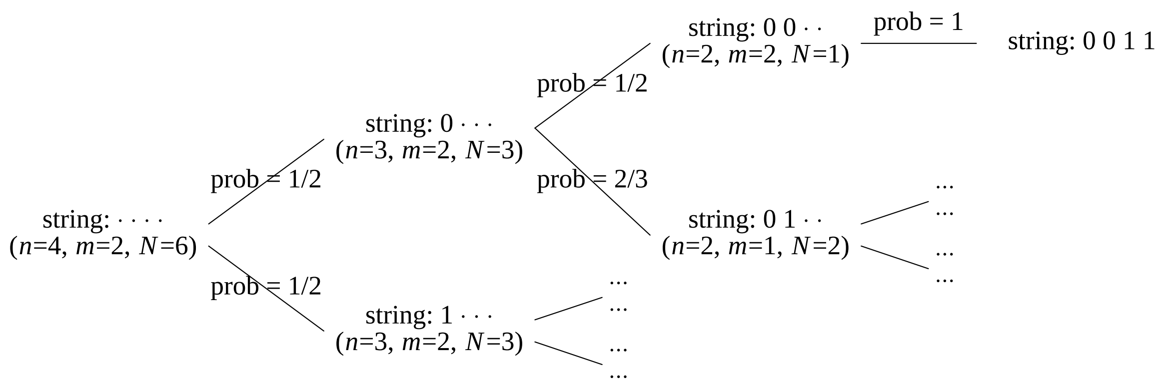


- How to choose the probability for successive bits in the binary string of length n and numOnes m?

Probability	Problem

1.	Prob(0) = 1/2 for each position	All binary strings of a given length
2.	Prob(0) depends on position n' = remainingLength, and m' = remainingNumOnes (prob(0) = $C^{n'-1}_{m'}/C^{n'}_{m'}$)	Binary strings of a given length and numOnes
3.	Depends on position n' = #(remaining symbols) prob(s) = $1/n'$ for each remaining symbols	Permutations

The case of length $n = 4$, numOnes $m = 2$, and numStrings $N = 6$:



Feb 18 CA: circle at (0,0) CB: circle at CA+(x,0); line -> from CA to CB chop CC: circle at CA+(x/2,-y); line -> from CA to CC chop # CA: circle at CA+(x2,0) CB: circle at CA+(x,0); line -> from CA to CB chop CC: circle at CA+(x/2,-y); line -> from CC to CA chop "(i) The three acyclic digraphs on" "n = 3 nodes and 2 links." at CC.s-(0,z) # CA: circle at CA+(x2,0) CB: circle at CA+(x,0); line -> from CB to CA chop CC: circle at CA+(x/2,-y); line -> from CC to CA chop # CA: circle at CA+(x2+x,0) CB: circle at CA+(x,0); line -> from CA to CB chop CC: circle at CA+(x/2,-y); line -> from CA to CC chop; line -> from CB to CC chop "(ii) The acyclic digraphs on" "n = 3 nodes and maximum number links 3." at CC.s-(0,z)

- Given an acyclic digraph, finding #(paths from x to y).

  Method #1: Assume that we have computed indegree of each node.

  (1) Initialize the stack by adding each source-node to it.

  (2) For each node z, initialize $p(z)$ = #(paths from source-nodes to z) = 0. Also, initialize $p(x) = 1$.

  (3) Do the following until indegree(y) = 0:

  (a) Let z = top(stack); remove z from stack.

  (b) For each node w in adjList(z), reduce indegree(w) by 1 and if indegree(w) = 0 then add it to stack. Also, add $p(z)$ to $p(w)$.

- **Homework.** Show in the table form how the topological sorting would proceed on the same digraph with the nodes {A, B, ..., G} (which we looked at before Mardi Gras holidays) when we use a queue instead of a stack to keep the current nodes of indegree 0 that have not been processed yet. (This might give a different topological sorting/ordering than the one using a stack.)

  Suppose we write a queue in the form <A, B, C>, where C is the head of the queue and A is the tail. Then adding D to the queue would give <D, A, B, C>, D being the new tail. If we want to take an item of the queue out, then we have to take the head-item C out and this would make the new queue <D, A, B>.

  Your table should show the queue (with head on right and tail on left), the node selected, the updated indegrees, and the new topological ordering. This is similar to the table we made using the stack for topological ordering.

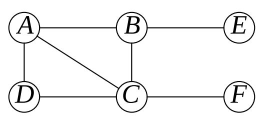
Depth-First Search

- Depth-first search of a graph and its applications:

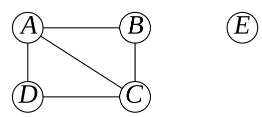
  (1) finding an xy-path,
  (2) finding if the graph is connected,
  (3) finding a cut-vertex,
  (4) finding a bicomponent, etc.

- Given any spanning tree of a connected graph and having chosen any node as the root, the non-tree edges can be classified as back-edges and cross-edges.
  - If there are no cross-edges then we can think of the tree as a depth-first tree.
  - If there are no back-edges then we can think of the tree as a breadth-first tree. (This is also the tree of shortest paths from the root, with 0/1 weights for the edges; some of the cross edges may represent alternative shortest paths.)
  - If we disregard the ordering of the children of a node, then there is just one df-tree and one bf-tree for each choice of root node.
  - Thus, all but $n + n$ spanning trees are neither df-trees and nor bf-trees.
  - A df-tree is a bf-tree if and only if the graph has no cycles.

- Connected graph: there is a path between any pair of nodes x and y ($y \neq x$).



(i) A connected graph on nodes $\{A, B, \cdots, E\}$.



(ii) A disconnected graph on nodes $\{A, B, \cdots, F\}$.

- **Homework.** Is it true that "if there is path from some node z to every other node, then there is a path between every pair of nodes"? Why is this result important (in determining connectivity of a graph)?

- Cut-vertex x:

  removal of x and its adjacent edges destroys all paths (one or more) between some pair of nodes y and z; we say x separates y and z.

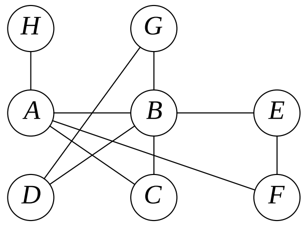
  In this case every path from y to z has to go through x, and thus #(acyclic path from y to z) = #(acyclic paths from y to x) $\times$ #(acyclic paths from x to z).

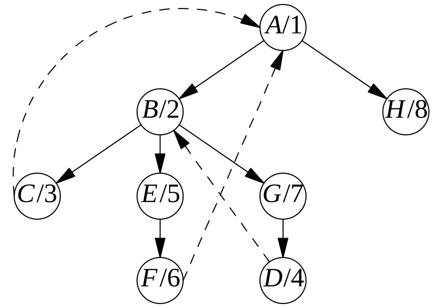
- B and C are the only cut-vertices in the first graph; the other graph has no cut-vertex.

- **Homework.** What is the minimum edges that need to be added to the first graph so that it has no cut-vertex.

- Depth first search of a connected graph:

  (1) Depends on the start-vertex and the ordering of nodes in the adjacency-list of nodes.
  (2) Produces an ordered rooted tree, with root = start-vertex; it is called the depth first tree.
  The children of a node are ordered from left to right in the order they are visited.
  (3) Each non-tree edge creates a cycle in the graph.
  (4) Each edge (x, y) of the graph is visited twice:
  once in the direction x to y and once in the direction y to x.





Stack (top on right)	Current node	df label	Edge processed	back/tree and visit#
$\langle A \rangle$	A	1	(A, B)	tree, visit #1
$\langle A, B \rangle$	2	B	(B, C)	tree, visit #1

- Cross-edge and back-edge:

  There are no cross-edges in the df-tree; each edge joins a a node with a parent or with an ancestor. (x, y) is a back edge if dfLabel(x) > dfLabel(y) and $y \neq$ parent(x)

- The start-vertex is a cut-vertex if and only if it has more than one child.

- **Homework.** Show in a similar table form the result of depth first processing when each adjacency-list is ordered in the reverse of alphabetical-list.

- **Homework.** For the graph below, show all possible depth-first trees that may arise if we change the stat-vertex and order the adjacency list in different ways.

- **BONUS** Consider the depth-first tree shown above. Show the maximum possible number of back edges. Is there any cut-vertices if all those edges are present in the graph?

Mar 09

- Algorithm DepthFirstTraverse:

  Use the following local data structures and variables in the function. (You could add parent-information to the structure GraphNode if the depth-first tree is to be used later for some other purpose.)

lastDfLabel:	0 initially; it is incremented by one before assigning to a node.
dfLabels[0..numNodes-1]:	each dfLabels[i] = 0 initially.
nextToVisit[0..numNodes-1]:	each nextToVisit[i] = 0 initially; nextToVisit[i] gives the position of the item in adjList of node i that is to be visited next from node i, i.e., the next link to visit from node i is link (i, j), where j = nodes[i].adjList[nextToVisit[i]].
stack[0..numNodes-1]:	initialized with the startNode; recall that this gives the path in the depth-first tree from the root to the current node.
parents[0..numNodes-1]:	parents[i] is the parent of node i.

Pseudocode: //it has a little bug; find this out as you create the program and test it, and then fix the bug.

1. Initialize lastDfLabel, dfLabels-array, parents-array, nextToVisit-array, the stack; also, let parent[currentNode] = currentNode (or -1).
2. While (stack $\neq$ empty) do the following:
   (a) Let currentNode = top(stack); update lastDfLabel and let dfLabels[currentNode] = lastDfLabel.
   (b) If (nextToVisit[currentNode] = degree[currentNode]) then backtrack by throwing away top of stack and go back to step (2).

(c) Otherwise, let nextNode = the node in position nextToVisit[currentNode] in adjList of currentNode, and update nextToVisit[currentNode].

(c) [Classify the type of the link (currentNode, nextNode) as follows

(1) tree-edge: if dfLabels[nextNode] = 0; in this case, let parent[nextNode] = currentNode and add nextNode to stack.

(2) back-edge: if (dfLabels[nextNode] < dfLabels[currentNode]) and (nextNode ≠ parents[currentNode])

(3) second visit: otherwise.

- **Program.** Create the function DepthFirstTraverse(int startNode) and show the output for the graph considered in the class with startNode 0 = A and startNode 1 = B. Create your datafile using the format we used for digraph, except that now node j will appear in the adjacency list of i if i appears in the adjacency list of j; keep the adjacency lists sorted in increasing order. For a graph, inDegree(i) = outDegree(i) = degree(i) for each node i. The function DepthFirstTraverse should produce one line of output for each link processed, and a separate line from backtracking and every time stack is modified. A possible output may look like:

stack = [0], node 0, dfLabel = 1
link = (0, 1), tree-edge
stack = [0 1], node = 1, dfLabel = 2
link = (1, 0), 2nd-visit
link = (1, 2), tree-edge
stack = [0 1 2], node = 2, dfLabel = 3
link = (2, 0), back-edge
link = (2, 1),‘2nd-visit
backtrack from 2 to parent(2) = 1
stack = [0 1]
···

Mar 11

- 3rd quiz.
- Breadth first traversal of a connected graph

Breadth first	Depth first
breadth-first spanning tree (BFT)	depth-first spanning tree (DFT)
rooted ordered tree	rooted ordered tree
tree-edges and cross-edges	tree-edges and back-edges
cross-edges limited to levels differing by ≤ 1	back-edges between levels differing by ≥ 2
no backtracking	backtracking
whole tree need to be maintained	backtracked nodes can be deleted from the tree
BFT tree tends to be "wide"	DFT tends to be "tall"
each edge visited twice	each edge visited twice
$O(\|E\|)$	$O(\|E\|)$

Mar 16

- Computing all paths in a graph from a start-node (reset dfLabel(x) = 0 when you backtrack from $x \neq$ start-node and reset the nextItemSeenFromAdjListToProcess(x) at the beginning of adjList(x)).

(1) For $x \neq$ start-node, #(occurrences of x in the new dfTree) = #(acyclic paths from start-node to x).

(2) $P = \#(\text{path from } i \text{ to } j \text{ in } K_n) = (n-2)!\left[1 + \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + \cdots + \frac{1}{(n-2)!}\right] \approx e(n-2)!$.

(3) #(occurrences of a node i in the new dfTree(1)) = P, except for $i = 1$ = root.

(4) #(tree edges in the new dfTree(1)) = $T(n) = (n-1)P = (n-1)T(n-1) + (n-1)$, with $T(1) = 0$ and $T(2) = 1$. This gives, $T(n) = (n-1)! + n(n-1)/2 = O((n-1)!)$ for $n \geq 2$.

- Check if there is a hamiltonian cycle by depth first search
- Compute the number of topological sorting.
- Minimum spanning tree by **Prim's** Algorithm.

Mar 18

- Minimum weight spanning tree of a weighted graph.
  - Number of trees on n nodes is n^{n-2}, too large to create them, find their weights, and choose the minimum.
  - Need a more direct way.
    + Start with a spanning tree and keep modifying it when its weight cannot be reduced any more.
    + Build a spanning tree slowly by adding a edge to an existing tree so that it ends up with a MST.
- The first approach:
  1. Build a spanning tree T (start at any node and do a depth-first traversal).
  2. Sort the edges in increasing (non-decreasing) link weights: $e_1, e_2, \cdots, e_m$.
  3. For each edge $e_1, e_2, \cdots$ do the following:
     (a) If e_i is not in the current spanning tree T and its weight is the not least weight in the cycle C in $T + e_i$, then add e_i and remove the maximum weight link in C.

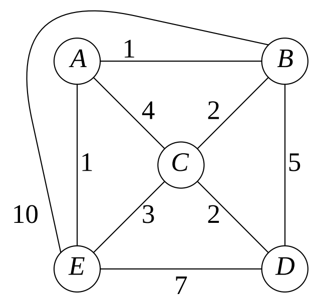
Problem: takes too much computation for detecting the cycles for various e_i (although each time we can detect the cycle in $T + e_i$).

- **Homework.** If $e_i = (x_i, y_i)$ where will you begin depth-first search of $T + e_i$ to detect the cycle?
- Pseudocode for second approach: Prim's Algorithm.
  1. Choose a start-node x_0 and let T consists of just this node.
  2. Repeat the following $n - 1$ times:
     (a) Add a new node x_i ($i = 1, 2, \cdots, n-1$) and connect it to T via an edge (x_i, y_i), where $y_i \in T$ such that this is the least cost edge connecting T to the outside.

  Selecting x_i and (x_i, y_i):
  1. For each $x_i \notin T$, find the best link (x_i, y_i) connecting x_i to T.
  2. Find the link with minimum weight among all (x_i, y_i). This gives both x_i and (x_i, y_i).

Mar 23

- **Homeworks.**
  1. Show in a table form (as indicated below) the steps and the trees in Prim's Algorithm; here, the second column shows the starting node. Note that once a node is added to T the column for that node for the remainder of the table will not have any entry (indicated by '–' below). Use the following input graph.



Node added to T	Best link connecting current T to nodes not in T and weight of that link					
	A=startNode	B	C	D	E	···
A	–					
···	–					
···	–					
···	–					
···	–					

2. What effects do we have on an MST (minimum weight spanning tree) when we reduce each link-weight by some constant c (which might make some link-weights < 0)?

- **Program:**

1. Write a function PrimMinimumSpanningTree(startNode) to construct an MST for a weighted graph. The output should show the following, with #(output lines) = #(nodes in the connected input graph).

   (a) The start-node.

   (b) For each successive line, a list of the triplets of the form $(x_i, y_i, w(x_i, y_i))$ for each node x_i not in the current tree T, where (x_i, y_i) is the current best link connecting x_i to T.

   Follow this by the node selected for adding to T.

Pseudocode for processing the links from the node x added to T:

1. For each y in adjList(x) do the following:

   (a) If y is not in T, then update bestLinkFrom(y) = x if $w(y, bestLinkFrom(y)) > w(y, x)$.

Notes:

(a) Use an array bestLinkFrom[0..(n – 1)], where n = #(nodes), and initialize each bestLinkFrom[i] = –1 to indicate that the best link is not known. For the start-node, let bestLinkFrom[startNode] = startNode.

This is the array that is returned by the function.

(b) Use another array inTree[0..(n – 1)], with inTree[i] = 1 meaning that i is in T and = 0 otherwise.

(c) The input-file graph.dat now should give the link weights as indicated below, where each item in the adjacency-list is followed by the link-weight in parentheses.

0 (3): 1(1) 2(4) 4(1) /for node A = 0 in the graph shown above

- Questions on Prim's Algorithm:
  - When do we process a link (x, y)?
  - What does the processing of (x, y) involve?
  - What is the complexity of processing (x, y)?
  - What is the complexity of Prim's Algorithm?
  - What is the main data structures needed for implementing Prim's Algorithm?

- Shortest paths in a weighted digraph, with $w(x, y) \geq 0$ for Dijkstra's Algorithm.

Apr 01

- Longest path in a acyclic weighted digraph (weights can be –ve):
  - Comparison with Dijkstra's shortest-path algo.
    + Unlike Dijkstra's algo, we need to look at all incoming links to y before we can find a longest-path to y.
    + It process a link (x, y) only after it finds a longest path to x
    + Subpath of a longest-path is also a longest-path between its end points.
  - It has complexity $O(|E|)$, similar to topological sorting Algorithm.
  - It is in many ways similar (with some variation) to topological sorting.

- Pseudocode for longestPath(startNode).

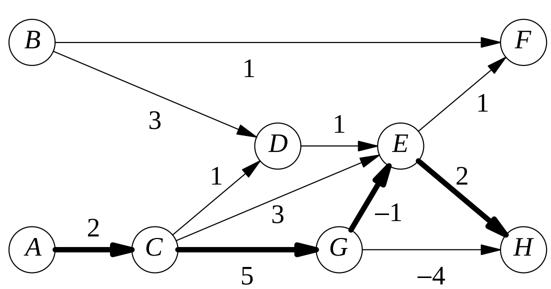
  It use following array data-structures:

$d(x)$ =	current longest path to x from startNode
parent(x)	= the node previous to x on the current longest path to x; parent(startNode) = startNode
indegree(x)	= number of links to x not yet looked at; it changes during the Algorithm

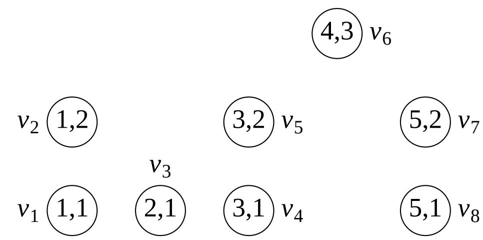
1. Preprocess the input digraph to make the startNode the only source-node:
   (a) Compute indegree(x) for each node x.
   (b) Initialize a stack with all source-nodes, if any, which are different from startNode (which may or may not be a source-node).
   (c) While (stack $\neq$ empty) do the following:
      (i) Let x = top(stack); remove x from stack.
      (ii) For (each $y \in$ adjList(x)) reduce indegree(y) by 1 and if it equals 0 then add y to stack.

2. Initialize a stack with startNode, let $d(x) = -\infty$ and parent(x) = −1 for each node x with indegree(x) > 0, and fi nally let $d(startNode) = 0$ and parent(startNode) = startNode. (You can take $-\infty$ to be a number which is minus of the sum of absolute values of all link-costs.)
3. While (stack $\neq$ empty) do the following:
   (a) Let x = top(stack); remove x from stack.
   (b) For (each $y \in$ adjList(x)) do:
      (i) If $(d(x) + w(x, y) > d(y))$, then let $d(y) = d(x) + w(x, y)$ and parent(y) = x.
      (ii) Reduce indegree(y) by 1 and if it equals 0 then add y to stack and also print the longest-path to y from startNode using the successive parent-links and print the cost of this path.

- **Program.** Develop a function LongestPath(int startNode) and test it with the digraph below. Show the output in a reasonable form (you have seen enough examples of proper outputs) for startNode = A. In particular, every time $d(y)$ for some node y is updated, print a separate line of the form "d(3) = 2, parent(2) = 0" to show the new $d(y)$ and its parent. (You can start with your topological sorting program and modify it appropriately.)



- **Homework.** Show the details (in the table form) the computations in Prim's Algorithm to construct an MST for the graph on the nodes shown below (given next to each node v_i are its x and y coordinates in the plane), where the link (v_i, v_j) has cost equal to the Euclidean distance between v_i and v_j. Assume the start-node is v_1. (Most of you did not do this problem right in the Quiz.)



- Find a suitable acyclic weighted digraph so that if we compute the longest between some pairs of nodes of this digraph then we will get the longest increasing subsequence (LIS) for the input sequence <4, 1, 3, 8, 5, 7, 13, 6>. Your method for constructing the digraph must be general enough that it will can be used for any input sequence for finding an LIS. Show your digraph, the longest path in your digraph, and the associated longest increasing subsequence.

Apr 15

- Huffman tree/Huffman code: assigning prefix-free codes to a set of symbols with given probabilities.
  - *Alphabet* Σ = a non-empty finite set of symbols; *word* is a finite non-empty string of symbols in Σ.
  - Code(x) = code of symbol $x \in \Sigma$ = a binary string; code($x_1 x_2 \cdots x_n$) = code(x_1).code(x_2)$\cdots$code(x_n).
  - Example. Let $\Sigma = \{A, B, C, D, E\}$.

A	B‘	C	D	E		Prefix-property
000	001	010	011	100	code(AAB) = 000000001; easy to decode	yes
0	01	001	0001	00001	code(C) = code(AB) = 001; not always possible to uniquely decode	no
1	01	001	0001	00001		yes
1	10	100	1000	10000		no

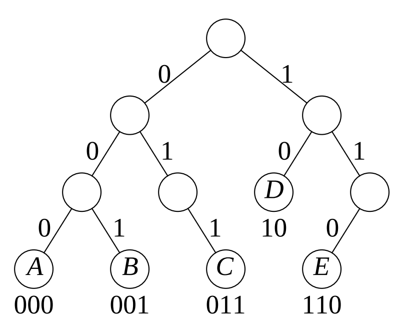
  - Some requirements:

1.	Each binary string has at most one possible decoding.
2.	It should be possible to do the decoding from the left, i.e. as the symbols are received.

  - A sufficient condition for both (1)-(2) the that the codes satisfy *prefix* property:

    No code(x) is the prefix of another code(y) for x and $y \in \Sigma$.
    In particular, code(x) $\neq$ code(y).

  - A code with prefix-property can be represented as the terminal nodes of a binary tree with 0 = label(left branch) and 1 = label(right branch).



- **Homework.** Consider the codes shows below.

A	B	C	D	E
000	001	011	10	110

(a) Arrange the codes in a binary tree form, with 0 = label(leftbranch) and 1 = label(rightbranch).

(b Is it true that the codes has the prefix-property? How do you decode the string 10110001000?

(c) Modify the above code (keeping the prefix property) so that the new code will have less average length no matter what the probabilities of the symbols are. Show the binary tree for the new code.

(d) What are the two key properties of the new binary tree (hint: compare with your answer for part (a))?

(e) Give a suitable probability for the symbols such that prob(A) < prob(B) < prob(C) < prob(D) < prob(E) and the new code in part (c) is optimal (minimum aver. length) for those probabilities.

Apr 20

- Floyd's Algorithm for shortest-path computation for all (x_i, x_j) node pairs.
  - The digraph may have -ve link costs; in that case, Dijkstra's Algorithm cannot be used.

    If there is a cycle with -ve cost, then shortest-paths between nodes in the cycle are not defined.
  - Total complexity is $O(N^3)$ for all node-pairs, which is comparable to $O(N^2)$ for shortest-path from a fixed start-point to all other nodes in Dijkstra's Algorithm.
  - Number of path-lengths computed = $O(N^3)$, one corresponding to the computation of $F^{k-1}(i, k) + F^{k-1}(k, j)$ for each $1 \le i, j \le N$ and $0 \le k \le N$.

    Per node pair (i, j), we compute $O(N) = N + 1$ path lengths including the path $\langle x_i, x_j \rangle$.

    This means most of the loop-free $e(N-2)$ $x_i x_j$-paths are not looked at.

- $F^k(i, j)$ = the shortest $x_i x_j$-path length where only intermediate nodes are $\{x_1, x_2, \cdots, x_k\}$.

$$
\begin{aligned}
(1)\ F^0(i, j) &= c(x_i, x_j) \\
(2)\ F^k(i, j) &= \min\,\{F^{k-1}(i, j), F^{k-1}(i, k) + F^{k-1}(k, j)\} \\
(3)\ F^N(i, j) &= \text{the final shortest } x_i x_j\text{-path length.}
\end{aligned}
$$

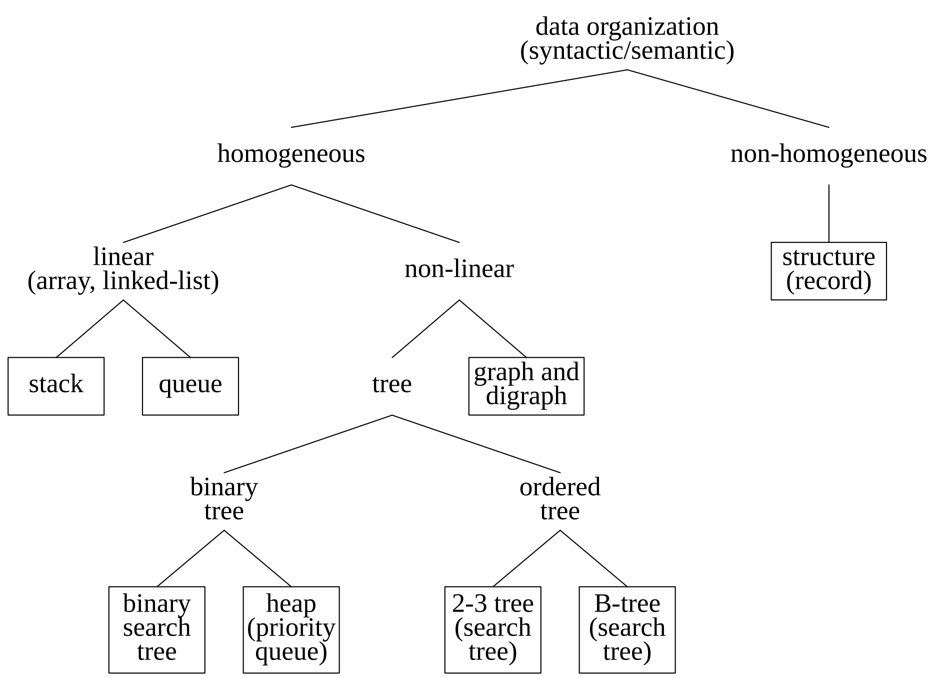
- How will you create a sorted list of the key in a 2-3 tree? Preorder traversal where at a node with one label you do

  list-left-subtree, list-node-label, list-right-subtree

  and for a node with two labels do

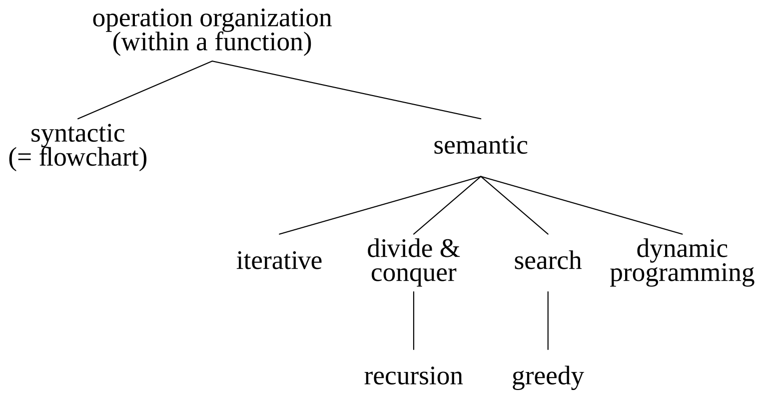
  list-left-subtree, list-first-node-label, list-middle-tree, list-second-node-label, list-right-subtree

- What is the connection between variance and the sum $(a_i - a_j)^2$, summed over all $1 \le i,\ j \le n$ for a given collection of numbers a_i?
- Find the next binary string of a given length n.
- **Homework** Find the smallest pair of numbers from nums[1..n] whose average is closest to 0.
- **Homework** Find three numbers from nums[1..n] whose standard deviation is minimum.
- Syntactic and semantic organization of data and operations.



  – Lists and arrays are of homogeneous data-units, where that data-unit can be any thing (homogeneous or not).

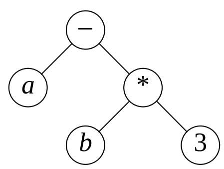
    This covers the case of lists of pointers to different classes in a common hierarchy in C++ because all those pointers are in a sense considered of the same type, namely, a pointer for the top record in the hierarchy.

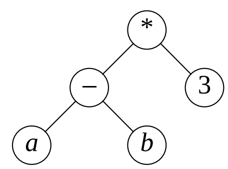


- What doe the following equal to

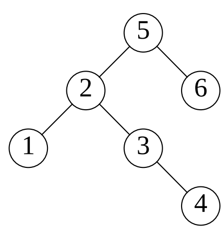
$$247801 \times 7125 - 247801 \times 7025$$

- How do you represent an arithmetic expression like $a - b * 3$ and $(a - b) * 3$, how do you build the tree, and how do you systematically simplify (bottom-up) it for given values of the variables a and b?

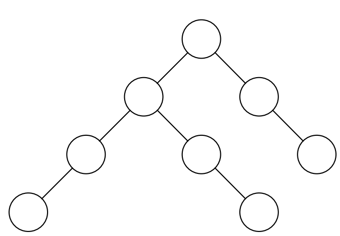




- What do you call a tree of the type shown below?



- Why do we call it binary? What is a non-binary tree – have we seen any yet? Why do we call it a search-tree?
- So how would you define a binary search tree?
- What is the main use of such a tree?
- Can you label the nodes of the binary tree below with the numbers 1, 2, ···, 8 to make it a binary search-tree? Is the labeling unique?



- Show two different inputs that can give rise to this tree? How many inputs are there?
- What are the most basic elements that we compute?

  numbers, strings, images (colors and positions of dots), other displays (strings and images).

  Each of them may have different meanings; number = age, weight, salary, temperature, height of a binary tree, length of a string.
- What is an Algorithm?

  A finite sequence of basic computation-steps and three other operations:
  inputs, outputs, and control-flow.
- What are the steps in computing the average of three input numbers a, b, and c.
- Are there different ways (Algorithms, methods) of the computing average?
- In how many ways can one method be better than the other?

  time-wise, memory-wise, simplicity-wise.
- Algorithm Design: organizing computations for maximum efficiency and the best solution.
- **In-Class:** Give an Algorithm for new International Students to go to Allen Hall from Student Union.
- Since computation needs data, organization of data for efficient access becomes important.
- Consider a program P using the data-organization on the left below. If we replace the data-organization by the one on the right, do we have to make any change in P? Is there then any reason to prefer one to the other? (Yes, the left one takes 4 + 3*8 = 28 and right one takes 3*(4+8) = 36) Why?

```
typedef struct {
        char grade, grade2, grade3;
        double score, score2, score3;
} First;
```

```
typedef struct {
        char grade;
        double score;
        char grade2;
        double score2;
        char grade3;
        double score3;
} Second;
```

- How many different structure definitions are there involving three chars and three doubles that would give different memory mappings? How many of them give total size 36 bytes (note that every structure address begins at a multiple of 4 bytes and is of size a multiple of 4 bytes)?
- This course will emphasize data-structure concepts and their applications in efficient program development.
  - Data Structure for better efficiency (linked lists of different kinds, trees) and better organization of data for visibility and naming (struct-construction).
  - Want clear program, with pseudocodes; main-functions is to primarily call other functions and set values of global variables.
  - Use for-loop when the control variable is updated in a regular fashion.
- Write the code for firstPositiveItemIndex(int *items, int numItems); if there are no positive items then it returns -1.

```
1. look at items[0], items[1], ... and stop as soon as
   a positive item is found.
2. if found then return index of the item
   else return -1.

for (i=0; i<numItems; i++)
    if (items[i] > 0) break
if (i < numItems) return(i);
```

```
else return(-1);
```

What is an alternate way of writing the if-then-else statement? (replace "break" by "return(i)")

- Modify it so that each call will find the successive positive item's index, and call the new function nextPositiveItemIndex; if we call it after it returns -1, then it should again restart the cycle by finding the first positive item's index. Note that if there is any change in items or numItems, then the search will start with items[0]. Should we find all the positive items and save it in a separate array?

– The complexity of computing partial sums of items[.] and items[.][.].

- Measuring efficiency via instrumentation of InsertionSort.

  – Need to generate random permutation or all permutations. How to do it?

    1. Find the term to be increased, find the new value, and adjust values to its right.
    2. Repeat the above till the sequence is $\langle n, n-1, \cdots, 3, 2, 1\rangle$

  – Measure average number of comparisons and data-movements

- Finding a subset of $m \le n$ items from a list of n (distinct) items which are most closely packed, i.e., have smallest variance.

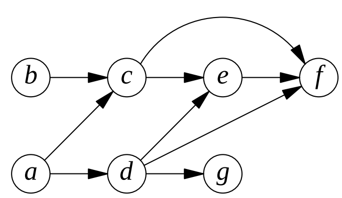
Jan 14

- Acyclic digraphs, source-nodes, sink-nodes, and topological sorting, pseudocode.

  Homeworks: how many ways can you top-sort; tree of all possibilities (not a binary tree); draw the tree with all terminal nodes placed on a line with equal spacing between them.

  – each node of the tree shows the nodes that can be laid off (including the the most recent child to be created).

  – each link of the tree shows what is being laid off.



- Input file design.
- **Program:** Write a program to obtain topological sort.

Jan 19

- Comparison of tree and digraph (digraph instead of graph because direction of links being a common feature between them).

	Rooted Tree T	Digraph G
1.	Made of nodes and directed links	Made of nodes and directed links
2.	For n nodes, #(links) = $n-1$	For n nodes, $0 \le$ #(links) $\le n(n-1)$
3.	Children $C(x)$ of node x	Nodes $N^+(x)$ that are adjacent from x
	– $C(x) \cap C(y) = \varnothing$ for $x \ne y$	– this need not hold
	– Terminal node x has $C(x) = \varnothing$	– Sink node x has $N^+(x) = \varnothing$
4.	Unique parent $par(x)$, except for root	$\lvert N^-(x)\rvert$ can be arbitrary
	– Root-node x has no-parent	– Source nodes x has $N^-(x) = \varnothing$
5.	Has no cycle	For acyclic digraph, #(links) $\le n(n-1)/2$
	– Unique path from root to all nodes	– #(paths between two nodes) $\le e(n-2)!$ for acyclic digraphs

	– Minimum connectivity from root to all nodes	
6.	Subtree $T(x)$ at a node x	Subdigraph $G(x)$ of nodes reachable from x
7.	$S(x) = \{x\}$, strong component of x	Strong component $S(x)$ of x can be as large as G – Merging each $S(x)$ into a node gives an acyclic digraph
8.	Already transitively reduced	Need not be already transitively reduced.

Jan 21

•

Jan 26

- **Iterative solution:** When the solution has many parts, and we compute each part in the same way on a slightly different part of the original input-data, part of which might be modified in the computation of previous parts.
  - Sorting by iteration:

    1. Find ith smallest items among S −{1st, 2nd, ⋯, $(i-1)$th smallest element}
    2. Repeat (1) for $n-1$ times, where $|S = n$.

    Bubble-sort is an iterative method, which finds successive largest number, where on completion of the ith iteration, more than i items might have properly placed.

    It is a refined implementation of the above pseudocode in some sense, but it may perform too many exchanges for some inputs.

    Insertion-sort can be thought of as an iterative (but more appropriately as a recursion) based on the size of the input-data:

    1. Successively sort first i items, $1 \le i \le n$.

  Iterative-approximation is a technique common numerical analysis (such as finding roots), where iterations are performed until some error limit is obtained.

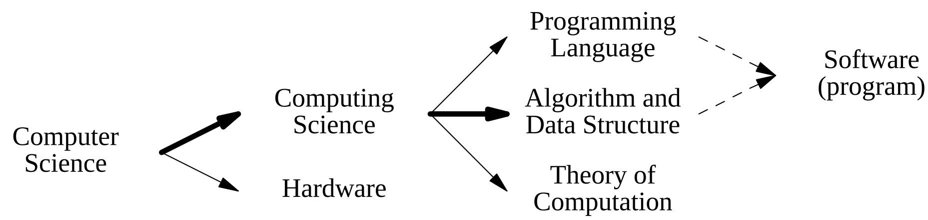
- **Recursion** is different in that the computation of ith call may not be over before starting the $(i+1)$th call, and each call might compute more than one part of the final solution.
- In depth-first, shortest-path, and longest-path, the basic unit of processing is a link (x, y).

Depth-first:	(x, y) is process after processing all (x, y') where $y' < y$ in adj-list(x).
Shortest-path:	Same as above, with the additional restriction that process all links at x before processing links at another x.
Longest-path:	Same as above, but the selection of successive x is different.

- Consider static and dynamic features for comparing Algorithms, unlike comparing concepts (using only static features).

Static features:	(1) Concepts used, basic computations performed in different iterations (recursions).
	(2) Conditions for selecting a unit input element for processing
	(3) Complexity
	(4) Structure of outputs produced: tree, lists, paths, etc.
	(5) Structure of and constraints on input (Floyd vs. Dijkstra).
	(6) Presence of pre-processing (simplifying input to a standard form, as in longest path)
Dynamic features:	(1) Iterative vs. recursive.
	(2) In which order, certain elements are processed.
	(3) Finite-state model and their comparisons

- Computing Science is part of Computer Sc, the latter could include both software and hardware. Data-structure is part of Algorithms, which is part of Software and the latter includes also programming skills.



- Each student introduces him/her-self by stating the name, year, major, where are you from?
- **In-Class:** Describe in (≤ 10) lines a program that you had written and are proud (were excited) about it.
  - Did you state what the input is? How about the output?
  - A name for your program? How long is the program?
  - What language was used?
- **Homework:** Give a short description (< 5 lines) of a programming problem that you would like to be able to solve by the end of this semester? Maybe you have seen something in action and you wondered how to do that sort of things?

ANOTHER EXAMPLE OF PSEUDOCODE

Problem. Compute the size of the largest block of non-zero items in *nums*[0. . *n* – 1].

Example. The underlined part is the largest block.

[2, 0, <u>–1, 3, 1</u>, 0, 0, 5].

Pseudocode:

1. Initialize maxNonZeroBlockSize = 0.
2. while (there are more array-items to look at) do:
   (a) skip zero's. //keep this
   (b) find the size of next non-zero block and update maxNonZeroBlockSize.

Code:

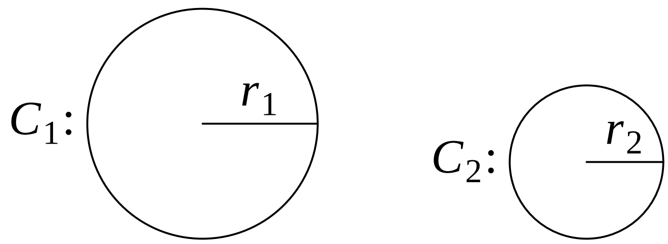
```
i = maxNonZeroBlockSize = 0;
while (i < n) {
      for (; (i<n) && (nums[i]==0); i++); //skip 0's
      for (blockStart=i; (i<n) && (nums[i]!=0); i++);
      if (i - blockStart > maxNonZeroBlockSize)
         maxNonZeroBlockSize = i - blockStart;
}
```

Question:

- ? If there are *m* non-zero blocks, then what is the maximum and minimum number of tests involving the items nums[*i*]?
- ? Rewrite the code to reduce the number of such comparisons. How much reduction is achieved?
- ? Generalize the code and the pseudocode to compute the largest size same-sign block of items.

A GEOMETRIC COMPUTATION PROBLEM

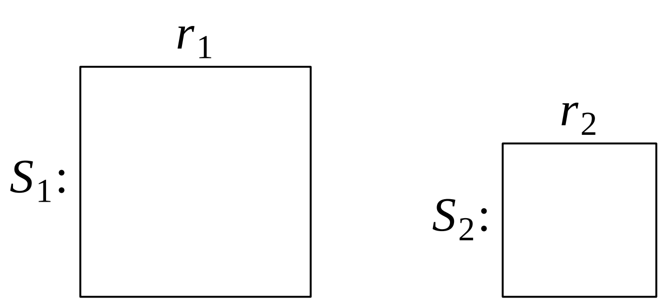
Problem: If C_1 and C_2 are two circles of radii r_1 and r_2, then when can we place C_1 inside C_2?



If C_1 can be placed inside C_2, then can we place it so that the centers of C_1 and C_2 coincide?

Question:

•? If S_1 and S_2 are two squares with sides of length r_1 and r_2, then when can we place S_1 inside S_2?

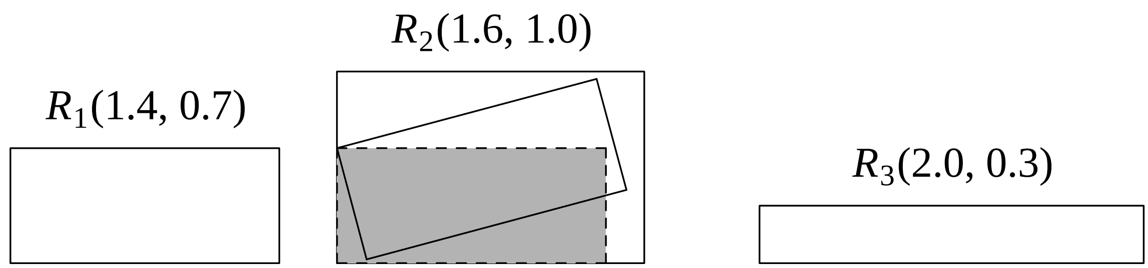


•? If S_1 can be placed inside S_2, then can we place it so that the centers of S_1 and S_2 coincide?

•? If we have a square and a circle, then when can we place one inside the other? (Can we make their centers coincide in that case?)

PLACING ONE RECTANGLE INSIDE ANOTHER

- Let $R_1 = (W_1, H_1)$ and $R_2 = (W_2, H_2)$ be two rectangles, where W_i = width(R_i) ≥ height(R_i) = H_i. When can we place R_1 inside R_2, and if so then how can we find an actual placement?



(i) Two of the infinitely many ways of placing R_1 inside R_2.

(ii) R_3 cannot be placed inside R_2.

Question:

1? What is an application of the rectangle-placement problem?

2? What is a *necessary* condition for placing R_1 inside R_2?

3? What is a *sufficient* condition for placing R_1 inside R_2?

4? Do these conditions lead to a placement-Algorithm (how)?

Generalization of Rectangle-Placement Problem:

- Find a placement that maximizes $R_1 \cap R_2$.

Placing a triangle Δ_1 inside another triangle Δ_2:

- Triangles are more complex objects than rectangles (why?). This makes the triangle-placement problem more difficult.
- What are some special classes of triangles for which the placement problem is easy? Find the placement condition and a particular way of placing.

NECESSARY vs. SUFFICIENT CONDITION

- If a property P implies a property Q, then
  - Q is a *necessary* condition for P, and
  - P is a *sufficient* condition for Q.

Example. Let P = "The integer n is divisible by 4".

- Consider the two conditions below, where $n_1 n_2 \cdots n_k = n$:

  Q_1: "The last digit n_k of n is 0, 2, 4, 6, or 8".
  Q_2: "The integer $n' = n_{k-1} n_k$ comprising the last two digits of n is divisible by 4". (Thus, $n' = n$ if $n < 100$.)

- Clearly, P implies Q_1 and P implies Q_2; so, each of Q_1 and Q_2 is a necessary condition for P.

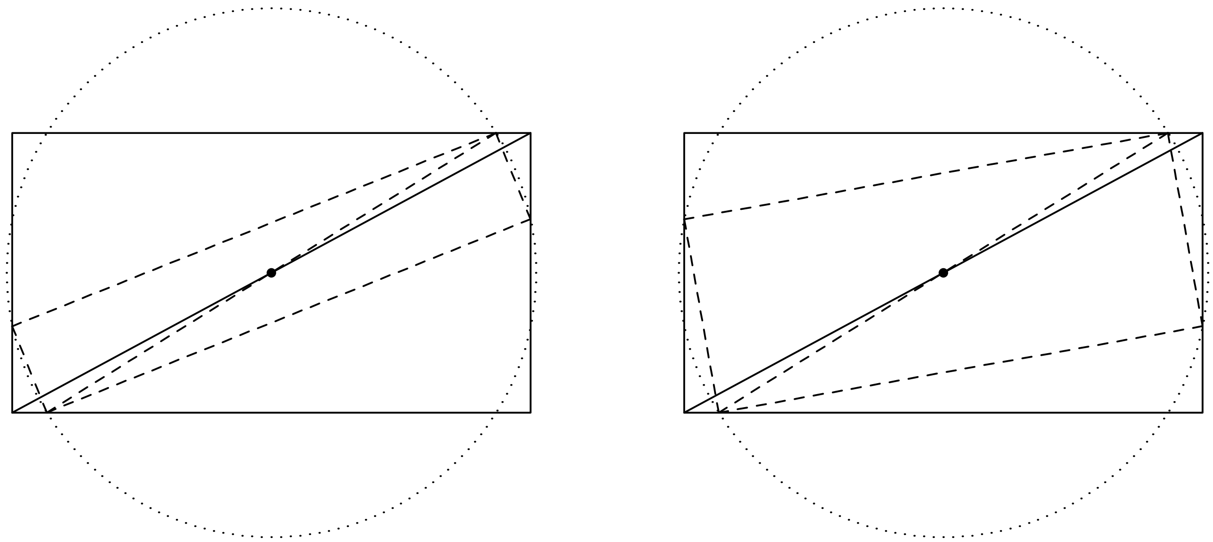
- However, only Q_2 implies P; Q_1 does not imply P (for example, let $n = 6 = n_k$, which makes Q_1 true and P false).

  Thus, only Q_2 is a sufficient condition for P.

If Q is both necessary and sufficient for P then P is both necessary and sufficient for Q. (P and Q are equivalent.)

Question: Are Q_1 and Q_2 above equivalent?

AN EXTREME CASE OF RECTANGLE PLACEMENT PROBLEM



For the case on right, the dashed rectangle R_1 can be slightly rotated and still kept inside the solid rectangle R_2.

Question:

1? Which of the dashed rectangles has the larger area? Can one of them be placed inside the other? Justify your answer.

2? Derive the necessary and sufficient condition for placing R_1 inside R_2 for the following cases:

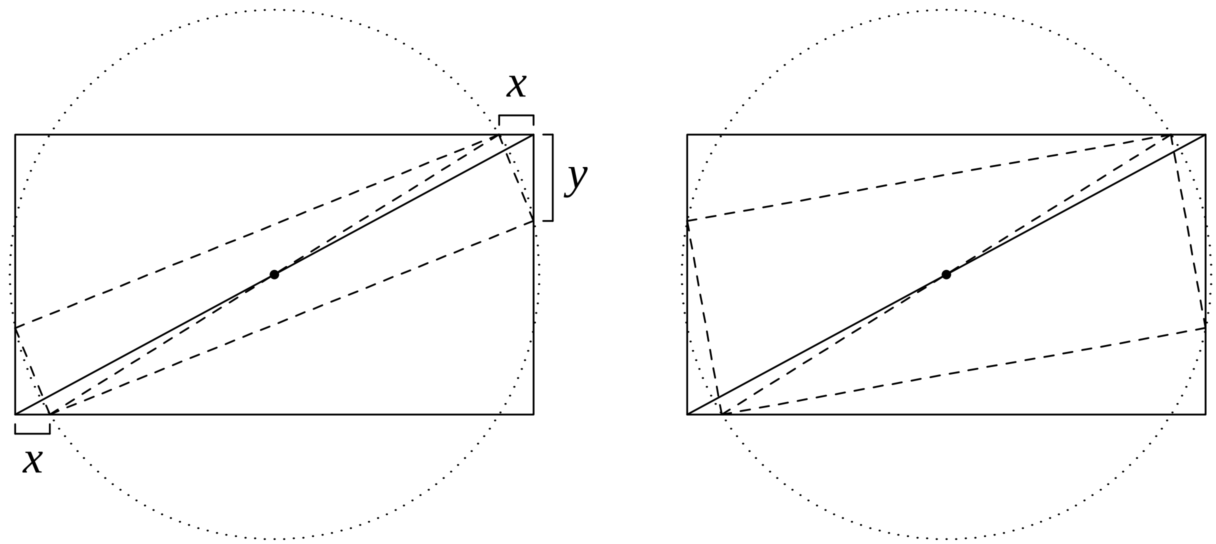
(a) R_1 can be placed inside R_2 without tilting.

(b) R_1 must be tilted to place inside R_2.

(c) R_1 can be placed inside R_2 in essentially only one way as in the lefthand case in the figure (a special case of (a)-(b)).

4? If R_1 can be placed inside R_2, is it true that we can make the placement so that their centers coincide? Explain your answer.

HINT FOR SOLVING THE CASE (c)



For the case on right, the dashed rectangle R_1 can be slightly rotated and still kept inside the solid rectangle R_2.

From similarity of triangles, we get

$$\frac{x}{H_1} = \frac{H_2 - y}{W_1} \text{ and } \frac{y}{H_1} = \frac{W_2 - x}{W_1}.$$

By comparing the length of the diagonals, we get

$$W_1^2 + H_1^2 \leq W_2^2 + H_2^2.$$

We also have $H_1^2 = x^2 + y_2$.

EXERCISE

1. Show that the largest square inside $R_2(W, H)$ is $R_1(H, H)$.
2. If we know that $D_1 = D_2$, where D_i is the length of the diagonal of R_i, then what is a necessary and suffi cient condition hat R_1 can be placed inside R_2.
3. Give an example of R_1 and R_2 such that $D_1 < D_2$ and still R_1 cannot be placed inside R_2

A STRING PROBLEM

Substring: Given a string $x = a_1 a_2 \cdots a_n$, each $x' = a_{i_1} a_{i_2} \cdots a_{i_k}$, where $i_1 < i_2 < \cdots < i_k$, is a k-substring of x.

For $x = abbacd$, $x' = bcd$ is a 3-substring but $x' = dc$ is not a 2-substring.

Question:

•? How many ways can we form k-substrings of $a_1 a_2 \cdots a_n$? When does all k-substrings $(0 < k < n)$ become the same?

•? When do we get the maximum number of distinct substrings?

Projection: If we keep *all* occurrences of some k-subset of the symbols in x (in the order they appear in x), then the resulting substring is a k-projection of x.

Example. For $x = aabcacbbadd$, which is made of four symbols $\{a, b, c, d\}$, we get $6 = C(4, 2)$ many 2-projections as shown below. Note that $x_{ab} = x_{ba}$, $x_{ac} = x_{ca}$, etc.

$$\begin{aligned} x_{ab} &= aababba, & x_{bc} &= bccbb, \\ x_{ac} &= aacaca, & x_{bd} &= bbbdd, \\ x_{ad} &= aaaadd, & x_{cd} &= ccdd. \end{aligned}$$

Question:

•? Give the string y made of the symbols $\{b, c, d\}$ which has the same 2-projection as x above, i.e., $y_{bc} = x_{bc}$, $y_{bd} = x_{bd}$, and $y_{cd} = x_{cd}$.

•? Give an Algorithm to determine the string x from its 2-projections. Explain the Algorithm using $x = aabcacbbadd$.

GENERATING (n, m)-BINARY STRINGS

Problem: Generate all (n, m)-binary strings, with $n - m$ zeros and m ones. There are six $(4, 2)$-binary strings:

Binary strings:	0011	0101	0110	1001	1010	1100
Associated integers:	3	5	6	9	10	12

An Algorithm AllBinaryStrings(n, m): //n=length, m = numOnes

1. For (i = 0, 1, 2, ⋯, $2^n - 1$) do the following:
   (a) Convert i to its binary-string form $s(i)$ of length n.
   (b) Print $s(i)$ if it has exactly m ones.

Problems with AllBinaryStrings(n, m):

- It is very ineffi cient when $m = n/2$. For $n = 4$ and $m = 2$, it generates 16 strings and throws away 16-6 = 10 of them.
- It does not work for $n > 32$ (= word-size in most computers).

Question:

1? What are some diffi culties with the following approach ($0 < m < n$) and how can you get around them:

Start with the string 1^m, then add one 0 in all possible ways, then for each of those strings add one 0 in all possible ways, and so on until each string has $n - m$ zeroes. until all zero's are added (e.g., 11 → {011, 101, 110}).

NEXT (*n*, *m*)-BINARY-STRING GENERATION

Examples of Successive (10,5)-Binary Strings:

A (10,5)-binary string:	0100111100
Next (10,5)-binary string:	$010\underline{10}00111$
Next (10,5)-binary string:	$010100\underline{10}11$
Next (10,5)-binary string:	$0101001\underline{10}1$
···	···
The last (10,5)-binary string:	1111100000

A necessary-and-sufficient condition for string y = next(x):

(1) The rightmost "01" in x is changed to "10" in y.

(2) All 1’s to the right of that "01" in x are moved to the extreme right in y.

Algorithm for Generating next(x) from x:

(1) Locate the rightmost "01" in x and change it to "10".

(2) Move all 1’s to the right of that "10" to the extreme right.

Moving 1’s To Right: $\cdots\underline{01}\overline{111}11000 \rightarrow \cdots\underline{10}00011\overline{111}$

- numOnesToMove = *min*(numEndingZeros, NumPrevOnes – 1)

Questions:

1? What happens when there is no "01"?

2? How will you generate a random (n, m)-binary string, i.e, with what probabilities will you successively determine the bits x_i of a random binary-string $x_1x_2\cdots x_n$? Give the probabilities for successive bits in 01101 (n = 5 and m = 3).

FINDING THE RIGHTMOST "01" IN A BINARY STRING

Pseudocode:

1. Scan the binary string from right-to-left to find the rightmost '1'.
2. Continue right-to-left scan till you find the first '0'.

Question:

1? Why is right-to-left scan is better than left-to-right scan to locate the rightmost "01" (for our application)?

2? Does the following code find the rightmost "01"?

```
for (i=length-1; i>=1; i--)
    if ((1 == binString[i]) &&
        (0 == binString[i-1]))
       break;
```

Explain with an example binary string how the above code wastes unnecessary comparisons of the items in binString[]. Describe the situation that makes the performance of the second code worst.

3? Give a piece of code corresponding to the pseudocode above and which does not have the inefficiencies of the code above.

PROGRAMMING EXERCISE

1. Write a function nextBinString(int length, int numOnes) that can be called again and again to create all binary strings in the lexicographic order with the given length and number of ones. Choose a suitable return value to indicate when the last binary string is created. Use an array binString for the binary-string, and use dynamic memory allocation.

   Your main-function should call nextBinString-function again and again. It should run for large values of length (= 100, say) and all $0 \leq$ numOnes $\leq$ length.

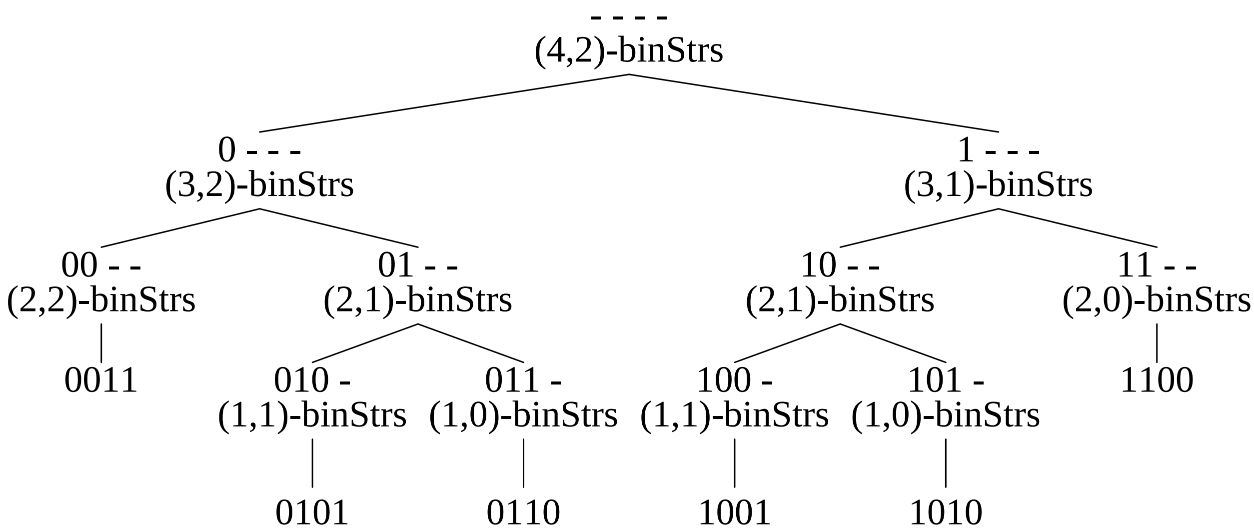
   First, test your program for length = 6 and numOnes = 2 and 3.

   Now modify nextBinString-function to count #(reads) from and #(writes) into the binString-array as you generate each binary string. Call these counts numReads and numWrites. The output should look like the following; show the average numReads and average numWrites upto 2 digits after the decimal point.

```
binString       numReads numWrites
----------------------------------
000111          0        6
001011          ...      ...
...
111000          ...      ...
----------------------------------
averNumReads  = ...  averNumWrites = ...
```

   Submit the paper copy of your code and the outputs for length = 6 and numOnes = 2 and 3..

A RECURSIVE APPROACH FOR GENERATING ALL (n, m)-BINARY STRINGS



Pseudocode for RecAllBinStrings(n, m):

1. If top-level call, then create the array binString[0..$n-1$] and let strLength = n.
2. If ($n = m$) or ($m = 0$), then fill the last n positions in binString with 1's or 0's resp., print binString, and return;

   otherwise, do the following:

   (a) Let binString[strLength – n] = '0' and call RecAllBinStrings($n-1$, m).

   (b) Let binString[strLength – n] = '1' and call RecAllBinStrings($n-1$, $m-1$).

Question:

1? Let $W(n, m)$ = #(total write-operations into binString[]) for generating all (n, m)-binary strings. Give the equation connecting $W(n, m)$, $W(n-1, m)$, and $W(n-1, m-1)$. Show $W(n, m)$ for $1 \le n \le 6$ and $0 \le m \le n$ in Pascal-triangle form.