Computer algorithms test review

Format
- True and false
- Pseudo code

Topic
- ~~Fibonacci Sequence~~
  - ~~Recursive method~~
    - ~~Case~~
    - ~~Pseudo code~~
  - ~~Polynomial method~~
    - ~~Case~~
    - ~~Pseudocode~~
  - ~~Compare the two methods~~

- ~~Basic knowledge of graphs~~
- ~~Basic knowledge of modular~~

- ~~Divide and conquer~~
  - ~~General idea~~
  - ~~Steps~~

- ~~Binary search~~
  - ~~Pseudocode~~
- ~~Merge sort~~
  - ~~Pseudocode~~

- ~~Greedy algorithm~~
  - ~~General idea~~
  - ~~Pseudocode~~
  - ~~Case~~

- ~~Kruskal's algorithm for MST~~

BOOK

Divide and conquer
- Chapter 2

Greedy algorithm
- Chapter 5

Algorithm with numbers
- Chapter 1

Resources
- Merge sort
  - http://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-006-introduction-to-algorithms-fall-2011/lecture-videos/lecture-3-insertion-sort-merge-sort/

- Binary search
  - http://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-006-introduction-to-algorithms-fall-2011/lecture-videos/lecture-5-binary-search-trees-bst-sort/

- List of algorithms that you already know (not really)
  - General
    - Recursive algorithm
    - Polynomial algorithm
  - Searching algorithms
    - Linear search
    - Binary search
    - Search with hashing
  - Sorting algorithms
    - Quick sort
    - Selection sort
    - Bubble sort
    - Insertion sort
    - Heap sort
    - Merge sort
  - Graph related algorithms
    - Bread-first Search/traversal
    - Depth first search/traversal
    - Single-source shortest path
  - Binary tree related algorithm
    - Bread-first traversal
    - Depth-first traversal
      - In-order
      - Pre-order
      - Post-order
- Algorithm
  - Sequence of well-defined computational steps that transform the input into the output
    - Steps are precise, unambiguous, mechanical, efficient and correct
  - A tool for solving well-specified computational problems
  - 3 questions to ask about any algorithm
    - Is it correct?
    - How much time does it take, as a function of n?
    - And can we do better?

- Fibonacci numbers
  - Generated by a simple rule
    - $F_n = F_{n-1} + F_{n-2}$
  - Grows almost as fast as the power of 2
    - $F_n = 2^{0.694n}$
  - Recursive algorithm
    - Function fib1(n)
      - If n == 0: return 0;
      - If n == 1: return 1;
      - else
        - return fib1(n-1) + fib1(n-2)
    - Efficiency
      - T(n) is the number of computer steps to compute fib1(n)
        - T(n) <= 2 for n <= 1
        - T(n) = T(n-1) + T(n-2) + C for n > 1
        - T(n) >= F_n (exponential though $F_n = 2^{0.694n}$)
      - Recursive algorithm is too slow except for very small values of n
        - Why?
          - Many computations are repeated
          - None are stored
  - Polynomial algorithm
    - Function fib2(n)
      - If n == 0 return 0
      - Create array f[0 .... n]
      - F[0] = 0, f[1] = 1
      - for i = 2 ... n{
        - F[i] = f[i-1] + f[i-2] }
      - Return f[n]
    - Array is used to store intermediate results
    - Inner loop consists of a single computer step and is executed n-1 time
      - Thus the algorithm is linear in n

- Evaluating running time
  - Processor's instruction set has a variety of basic primitives
    - Branching
    - Storing to memory
    - Comparing numbers
    - Simple arithmetic
    - More-
  - Estimating running time of algorithm
    - Considered as one category of basic computer steps
  - Architecture-specific and hardware-specific details are ignored

Big-o Notation

Running time (number of basic computer steps) of an algorithm is expressed as a function of the size of input n

EXP: $T(n) = 5n^3 + 4n + 3$

Lower-order terms and details of coefficient in the leading term are ignored

(because as n is generally rather large they become insignificant quickly)

EXP: $T(n) = O(n^3)$

f(n) and g(n) are functions from positive integers to positive real numbers

f = O(g) (f grows no faster than g)

If there is a constant $c > 0$ such that $f(n) <= c * g(n)$

EXP: $10n = O(n)$

When comparing two algorithms, we care about the rate of growth of n

$f_1(n) = n^2$, $f_2(n) = 2n + 20$

$f_2(n) / f_1(n) =$

$(2n + 20) / n^2$

$<= 22$

For all n $f_2 = O(f_1)$

Common rules

Multiplicative constants can be omitted

EXP: $14n^2$ ---> n^2

n^a dominates n^b if $a > b$

EXP: n2 dominates n

Any exponential dominates any polynomial

EXP: 3^n dominates n^5; 3^n dominates 2^n

Any polynomial dominates any logarithm

EXP: n dominates $(\log n)^3$; n^2 dominates $n(\log n)$

What is a logarithm

Answers question

How many of one number do we multiply to get another number

$\log_{(Base)}(integer)$ = (number of times Base must be multiplies to get to integer)

$\log_{(2)}(8) = 3$

Insertion sort

Insertion-sort(A)

```
for j <- 2 to length[a]
    do key <- a[j]
        Insert A[j] into the sorted sequence a[1..j-1]
        i <- j -1
        while i > 0 and A[i] > key
            do A[i+1] <- A[i]
                i <- i-1
        A[i+1] <- key
```

Input: A sequence of n numbers (a_1, a_2, ..., a_n)

Output: a permutation (reordering) (a_1', a_2', ..., a_n') of the input sequence such that $a_1' <= a_2' <= ... <= a_n'$

Efficiency

$T(n) = c_1(n-1)[c_2 + c_4 + (t_j -1)(c_5 + c_6 + c_7) + c_5 + c_8]$ -- t_j epsilon [1, n-1]

c1 cost n = 1

c2 cost n-1

c4 n - 1

c5 Sigma n j= 2^{t_j}

c6 sigma n j = $2^{(t_j - 1)}$

c7 sigma n j - $2^{(t_j -1)}$

c8 n - 1

Best case : array already sorted, t_j =1:

$T(n) = c_1(n-1) [c_2+c_4+c_5+c_8]$

$= an+b$

Worst case: array is in reverse sorted order, $t_j = n-1$

$T(n) = an^2 +bn+ c$

Comparison of sorting algorithms

Sort	Best case	average case	worst case
Selection sort	$O(N^2)$	$O(N^2)$	$O(N^2)$
Bubble sort	$O(N^2)$	$O(N^2)$	$O(N^2)$
Shortbubble	O(N)(*)	$O(N^2)$	$O(N^2)$
Insertion sort	O(N)(*)	$O(N^2)$	$O(N^2)$
Merge sort	$O(N\log_2 N)$	$O(N\log_2 N)$	$O(N\log_2 N)$
Quick sort	$O(N\log_2 N)$	$O(N\log_2 N)$	$O(N^2)$(depends on split)
Heap sort	$O(N\log_2 N)$	$O(N\log_2 N)$	$O(N\log_2 N)$

- Addition
  - Basic property of decimal numbers (binary number)
    - Sum of any three single-digit numbers is at most two digits long
  - Add two numbers in any base
    - Align their right hand ends
      - Compute the sum of digit by digit in a single right to left pass
      - Maintain overflow as a carry
    - Since each individual sum is a two-digit number
      - The carry is always a single digit
      - At any given step three single digit numbers are added
  - Efficiency of adding
    - Two n-bit long binary numbers x and y
      - How long to add them expressed as a function of the size of the input n
    - Sum of x and y is n+1bits at most
      - Each individual bit of this sum gets computed in a fixed amount of time
        - $T(n) = c_0 + c_1 n$ where c_0 and c_1 are some constants
        - Running time is linear O(n)

- Multiplication
  - Multiplying two binary numbers x and y
    - Creates an array of intermediate sums,
      Each representing the product of x by a single digit of y
    - Values left shifted then added up
  - Efficiency of multiplication
    - X and y are both n bits
    - There are n intermediate rows
      With lengths of up to 2n bits (cause of shift)
    - Total time take to add up these rows two numbers at a time
      - n-1 times of O(n)
      - Which is $O(n^2)$ and thus quadratic in the size of the input n
    - Division also $O(n^2)$

- Modular Arithmetic
  - A system for dealing with restricted ranges of integers
  - X modulo N
    - defined to be the remainder when x is divided by N;
      - If $x = qN + r : 0 <= r <= N$
      - Then x modulo N is equal to r
  - It limits numbers to a pre-defined range
    - {0, 1, ..., N-1} wraps around whenever the number leaves this range
  - It deals with all integers but divides them into N equivalence classes
    - Of the form : { i + kN : k epsilon Z} for some i between 0 and N-1
  - Equivalence between numbers
    - X and y are congruent modulo N if they differ by a multiple of N
      - X congruent y (mod N) if N divides (x-y);
    - Example
      - 253 minutes is 4 hours and 13 minutes
        - 253 congruent 13(mod 60)
      - Can also be negative
        - 59 congruent 1 (mod60): when it is 59 minutes past the hour it is also one minute short the hour

Divide and conquer

- Top-down technique for designing algorithms
  - Divide
    - Split the problem into a number of sub problems that are similar to the originl problem but smaller in size
  - Conquer
    - Solve the subproblem recursively (successively and independently)
  - Combine
    - Merge the solution to the subproblems into the solution for the original problem

Binary search

- Uses Divide and conquer
- Searching problem
  - MUST BE SORTED FIRST
  - Given A[1...n] an array of non decreasing sorted order
    - A[i] <= A[j] whenever 1 <= i <= j <= n
  - Let q be the query point
  - The problem consist of finding q in the array A
- Divide
  - Split A into two segments from the middle
    - A[0] to A[A.length/2] -> Array B
    - A[A.length/2] to A[A.length] -> Array C
- Conquer (recursively)
  - Compare q with the middle element (A[A.length/2])
    - If q is less than A[A.length/2]
      - Continue to Array B
    - If q is more than A[A.length/2]
      - Continue to Array C
    - Repeat the Divide step in the appropriate Array and then repeat the Compare step in the appropriate array until the we can safely conclude that q is or is not in the Array
- Combine
  - The conclusion is that q is found / not found
- Binary search can be accomplished in logarithmic time
  - $T(n) = O(\log_2 n)$

Merge Sort

- Closely follows the divide-and-conquer paradigm
- Sorting problem
  - Given $A=\{a_1, a_2, a_3, ..., a_n\}$ a sequence of n numbers
    - Reorder elements of A such that
      - $A' = \{ a_1', a_2', a_3', ..., a_n'\} : a_1' <= a_2' <= a_3' <= ... <= a_n'$
- Divide
  - Split A into two subsequences of n/2 elements each
    - A[0] to A[A.length/2] -> Array B
    - A[A.length/2] to A[A.length] -> Array C
- Conquer
  - Sort the two subsequences recursively using the merge sort
- Combine
  - Merge the two sorted subsequences to produce A'

Merge sort implementation

Merge-sort (Array, Starting_point, Ending_point)

If Starting_point < Ending_point

Mid = ((Starting_point + Ending_point)/2)

Merge-sort (Array, Starting_point, Mid)

Merge-sort (Array, Mid + 1, Ending_point)

Merge (Array, Starting_point, q, Ending_point)

Recurrence

An equation describes the overall running time on a problem of size n in terms of the running time on smaller inputs

If the problem of size n is divided into c subproblems

The size of each subproblem is n/b and it rakes f(n) = O(nd) time to divide and combine the solutions, then recurrence is

$T(n) = cT(n/b) + O(n^d)$, while n is large

(c > 0, b>1 and d => 0)

Master theorem

For $T(n) = cT(n/b) + f(n)$, when $f(n) = O(n^d)$

If $d > \log_b c$, then $T(n) = O(n^d)$

If $d = \log_b c$, then $T(n) = O(n^d \log_2 n)$

If $d < \log_b c$, then $T(n) = O(n^{\log_b c})$

Analysis of merge sort

We assume the original problem size n is a power of 2

Divide

Each divide step yields 2 subsequences of size n/2

A sequence of n will be divided $\log_2 n$ times to reach 1 element sequence

Combine

Merge n elements into a sorted array

$T(n) = cT(n/b) + f(n)$

Divide

It takes constant time to divide a sequence -- O(1)

Conquer

We recursively solve two subproblems of size n/2

Contributes 2T(n/2) to the running time (c = 2 and b =2)

Combine

Merge n element subarray takes time O(n) (d=1)

Master theorem case No. 2

($d = \log_b c$) : $T(n) = O(n^d \log_2 n) = O(n \log_2 n)$

Median, min. & Max

The ith order statistic of a set of n elements is the ith smallest element

The minimum of a set is the first order statistic

i = 1

The maximum is the nth order statistic

i = n

A median is the halfway point of the set

i = (n+1)/2 when n is odd

i = n/2 and i = n/2+1 when n is even

Matrix multiplication

- Product of two n x n matrices X and Y is a third n x n matrix Z = XY: with (i, j)th entry
  - Z_{ij} = summation n start at 1 for $X_{ik}Y_{kj}$
- Square matrix-multiply
  - $O(n^3)$
  - Input A and B n x n matrices
  - C is a new n x n matrix

```
For i = 1 to n
    For j = 1 to n
        Cij = 0
        For k = 1 to n
            Cij = Cij + Aik * Bkj
```

  - Return C
- Partition matrix
  - If n is the exact power of 2, then each of A, B and C matrices could be divided into four n/2 X m/2 matrices

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \quad B = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix} \quad C = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}$$

  - We could write C = A * B as

$$\begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} * \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}$$

  - Where

$$C_{11} = A_{11} * B_{11} + A_{12} * B_{12}$$

$$C_{12} = A_{11} * B_{12} + A_{12} * B_{22}$$

$$C_{21} = A_{21} * B_{11} + A_{22} * B_{21}$$

$$C_{22} = A_{21} * B_{12} + A_{22} * B_{22}$$

Greedy algorithms

- They build up a solution piece by piece
  - Always choosing the next piece that offers the most obvious immediate benefit
- Local optimization
  - Making a move seems best at the moment
  - Not worrying too much about future consequences

- Graph
  - Data structure that consists of a set of nodes and a set of edges that relate the nodes to each other
    - Vertex
      - Node in a graph
    - Edge (arc)
      - Connection between two nodes in a graph, represented by a pair of vertices
        - Undirected graph
          - All edges are bidirectional
        - Directed graph
          - All edges point in a direction
    - Formally a graph G is defined as G = (V,E)
      - V(G) is a finite nonempty set of vertices
      - E(G) is a set of edges (written as paris of vertices)
  - Adjacent vertices
    - Two vertices in a graph that are connected by an edge
  - Path
    - A sequence of vertices that connects two nodes in a graph
    - Cycle
      - In a directed graph it is a circular path where V0 -> V1 -> ... Vk -> V0
    - Acyclic
      - Graph with no cycles
    - (DAG) directed acyclic Graph (DAG)
      - A directed graph with no cycles
  - Complete graph
    - A graph in which every vertex is directly connected to every other vertex;
      - For a graph with N nodes
        - N*(N-1) edges in a complete directed graph
        - N*(N-1)/2 edges in a complete undirected graph
  - Weighted graph
    - A graph in which each edge carries a value
  - Martix representation of graph
    - Adjacency Matrix
      - For a graph with N nodes, an N by N table that shows the existence (and weights) of all edges in the graph
    - Mapping array
      - An array that maps vertex names into array indexes
    - Marking
      - Associate a boolean variable with each vertex: true if visited, false if not yet visited
  - Linked-list representation of Graph
    - Adjacency list
      - A linked list that identifies all the vertices to which a particular vertex is connected
      - Each vertex has its own adjacency list
  - Graph traversal
    - Visit all the vertices, beginning with a specified start vertex
    - No vertex is visited more than once and vertices are only visited if they can be reached
      - If there is a path from the start vertex
    - Breadth first search algorithm
      - Visit all the nodes on one level before going to the next level

Depth first search algorithm
Visit all the nodes in a branch to its deepest point before moving up

Breadth first traversal
"neighbors-first"
Queue data structure is needed
Holds a list of vertices which have not been visited yet but which should be visited soon
While visit a vertex involves adding its neighbors to the queue.
Neighbors are not added to the queue if they are already in the queue, or have already been visited
Queue FIFO

Depth first traversal
A stack data structure is needed
Holds a list of vertices which have not been visited yet but should be soon
While visit a vertex involves adding its neighbors to the stack
Neighbors are not added to the stack if they are already in the stack or have already been visited
Stack LIFO

(MST) Minimum Spanning Tree (MST)
Given an undirected graph G = (V,E)
Weight of edge e is W_e, to find an undirected acyclic graph (tree) T = (V,E')
With E' member of E that minimizes
Weight (T) = summation of e is a member of E' for W_e
Application
Build cheapest network for a group of computers
Go through all vertices in the undirected graph while discarding all edges that would create a cycle
Making the lowest weighted edges graphed first

Kruskal's algorithm
Is a greedy algorithm and MST
Min spanning tree is acyclic
Property 1
Removing a cycle edge cannot disconnect a graph
The algorithm starts with the empty graph and then selects edges from E according to the following rule
Repeatedly add the next lightest edge that doesn't produce a cycle
Property 2
A tree on n nodes has n-1 edges
Property 3
Any connected undirected graph G = (V,E) with |E| = |V| - 1 is a tree
Property 4
An undirected graph is a tree if and only if there is a unique path between any pair of nodes

Cut property
A cut is any partition of the vertices into two groups, S and V-S
Supposed edges X are part of a minimum spanning tree of G = (V;E)
Pick any subset of nodes S for which X does not cross between S and V-S and let e be the lightest edge across this partition
Then X Union {e} is part of some MST
It is always safe to add the lightest edge across any cut
Provided X has no edges across the cut

Kruscals algorithm

- Implementation
  - Procedure kruskal (G,w)
    - Input : a connected undirected graph G = (V,E) with edge weights w_e
    - Output: a minimum spanning tree defined by the edges X
      - For all u member of V
        - Makeset(u) // makeset(u) : create a singleton set containing just u
      - X = {}
      - Sort the edges E by weight
      - For all edges {u,v} member of E, in increasing order of weight
        - If find(u) does not equal find(v):
          - Add edge {u,v} to X // find (v) to which set does v belong
          - Union(u,v) // union(u,v) merge the sets containing u and v
- Implementation of kruskal's algorithm
  - The algorithm's state is modeled as a collection of disjoint sets, each of which contains the nodes of a particular component (a cut)
  - Initially each node is in a component by itself
    - Makeset(x) : create singleton set containing jsut x
  - Pairs of nodes are tested repeatedly to see if they belong to the same set
    - Find(x) : to which set does x belong to
  - When an edge is added two components are merged
    - Union(x,y) : merge the sets containing x and y
  - The algorithm uses |V| makeset, 2|E| find, and |V|-1 union operations
- Time complexity
  - Kruskal's algorithm
    - Sorting the edges: $O(|E| \log_2 |V|)$ (remember $\log_2 |E| = \log_2 |V|$)

Data structure for disjoint sets

- A set could be stored as a directed tree
  - Nodes are elements of set, not in order
  - Root is a representative, or name, for the set
- Each node has a parent pointer TT that eventually leads up to the root of the tree
  - The parent pointer of root points to itself
- Each node also has a rank - the height of the subtree hanging from that node
  - Create a singleton set containing just x
    - Procedure makeset(x)
      - Pie(x) = x
      - Rank(x) =0
  - To which set does x belong? It returns the root of its set
    - While x does not equal pie (x) : x = pie(x)
    - Return x

Minimum spanning tree

Property 1

Removing a cycle edge cannot disconnect a graph