Design and Analysis of Algorithms — Dec 14, 2009
Massachusetts Institute of Technology — 6.046J/18.410J
Profs. Ronald L. Rivest and Shafi Goldwasser — Final Examination

Final Examination

- Do not open this exam booklet until you are directed to do so.
- This exam ends at 4:30 P.M. It contains 8 problems, some with several parts. You have 180 minutes to earn 160 points.
- This exam is closed book, but you may use two double-sided 8 1/2″ × 11″ or A4 crib sheets.
- When the exam begins, write your name in the space below and on the top of every page in this exam. Circle your recitation instructor.
- Write your solutions in the space provided. If you need more space, use the scratch paper at the end of the exam booklet. Please write your name on any extra pages that you use.
- **Do not spend too much time on any one problem.** Read them all through first, and attack them in the order that allows you to make the most progress.
- Do not waste time rederiving algorithms and facts that we have studied. It suffices to cite known results.
- Show your work, as partial credit will be given. You will be graded on the correctness and efficiency of your answers and also on your clarity. Please be neat.
- When giving an algorithm, sketch a proof of its correctness and analyze its running time using an appropriate measure.
- Good luck!

Problem	Points	Grade	Initials
1	48	40	
2	33	19	
3	15	15	
4	10	10	

Problem	Points	Grade	Initials
5	10	10	
6	15	15	
7	10	10	
8	19	19	

Total	160	138	86.1%

NOTE FROM SCOTT: For ~~some~~ questions I used a different, although I believe completely valid appr as for the question 2(d) However, even if these were omitted, my score is still well above 50.1%.

Name: SCOTT YOUNG

Circle your recitation instructor:

TB (F10, F11) Ammar (F12,F1) Angelina (F2,F3)

Problem 1. True/False and Justify [48 points] (12 parts)

Circle **T** or **F** for each of the following statements to indicate whether the statement is true or false, respectively. If the statement is correct, briefly state why. If the statement is wrong, explain why. Your justification is worth more than your true or false designation.

(a) [✓ **T**] **F** [4 points] If a sequence of n operations on a data structure cost $T(n)$, then the amortized runtime of each operation in this sequence is $T(n)/n$.

The definition of amortization is the cost of n operations divided by n.

✓

(b) [✓ **T**] **F** [4 points] Fix some integers $m >> n > 0$. For every function $h : [m] \rightarrow [n]$ in a universal hashing family $\mathcal{H}$, there exists an integer $0 \leq i \leq m - 1$ such that $h(i) \neq 0$.

~~In~~ order to be universal, h must map ~~values~~ of m uniformly distributed on n, therefore, if $n > 0$, there must be some value $h(i) \neq 0$.

(c) **T** [✓ **F**] [4 points] If a problem in NP can be solved in polynomial time, then it is known that all problems in NP can be solved in polynomial time.

$P \subseteq NP$ is known already, it does not imply $P = NP$

✓

(d) [✓ **T**] **F** [4 points] If P $=$ NP, then every nontrivial decision problem $L \in$ P is NP-complete. (A decision problem L is ***nontrivial*** if there exist some x, y such that $x \in L$ and $y \notin L$.)

If $P=NP$, then all NP-Complete problems have polynomial time solutions, and therefore all problems in NP are of the same complexity class. NP-Complete indicates the "hardest" problems in NP, which includes P if they are all equally ~~to~~ easy.

✓

(e) [✓ **T**] **F** [4 points] A spanning tree of a given undirected, connected graph $G = (V, E)$ can be found in $O(E)$ time.

First, if $|V| > |E|$, return false, no spanning tree is possible.

Otherwise maintain a set of unconnected vertices and connect them one at a time to the growing tree (Prim's w/o minimum requireme

(f) **T** [✓ **F**] [4 points] Consider the following algorithm for computing the square root of an n-bit integer x:

```
SQUARE-ROOT($x$)
    For $i = 1, 2, \ldots, \lfloor x/2 \rfloor$:
        If $i^2 = x$, then output $i$.
```

This algorithm runs in polynomial time.

$x/2$ is exponential in the size of n.

(g) [✓ **T**] **F** [4 points] If all edge capacities in a flow network are integer multiples of 3, then the maximum flow value is a multiple of 3.

All the flows could be divided by 3 and ~~the~~ augmenting path ~~formula~~ in the residual network would never add a fractional flow.

(h) **T** [✓ **F**] [4 points] Given a connected directed graph $G = (V, E)$ and a source vertex $s \in V$ such that each every $e \in E$ has an integer weight $w(e) \in \{0, 1, \ldots, V^3\}$, there is an algorithm to compute single-source shortest-path weights $\delta(s, v)$ for all $v \in V$ in $O(E \lg \lg V)$ time.

No such algorithm exists.

(i) **T** [✓ **F**] [4 points] Given a constant $\varepsilon > 0$, probabilistic property testing whether a sequence is ε-close (as defined in the lecture) to monotone requires $\Omega(n)$ queries.

*

We can sample sequences $\leq \Omega(n)$ to create a probabalistic ~~[illegible]~~ guess that a sequence is $\mathcal{E}$-close in sublinear time ✓

(j) **T** [✓ **F**] There is a sublinear-time algorithm that decides whether a given undirected graph is connected.

To prove a graph is connected, we would need, at the very least, to examine each vertex to ensure it connects. ✓ This bound may not apply to Monte Carlo algorithms

(k) **T** [✓ **F**] [4 points] An adversary can force a skip-list insertion to take $\Omega(n)$ time.

Skip-list is randomized, therefore an adversary can never select an input to force an $\Omega(n)$ insertion time ✓

(l) **T** [✓ **F**] [4 points] Assume P $\neq$ NP. The Traveling Salesman Problem has a polynomial-time α-approximation algorithm for some constant $\alpha > 1$.

Only if the triangle inequality is preserved, otherwise any $1+\epsilon$ approximation scheme could be used to solve the Hamiltonian Cycle problem, which is NP-Complete.

Problem 2. Short answer [33 points] (5 parts)

Give brief answers to the following problems.

(a) [9 points] Match up each application with an algorithm or data structure that we used to solve it in this course. Use each answer exactly once.

C $\boxed{D}$ ✗	Map folding	~~A.~~	Polynomial reduction
$\boxed{E}$ ✓	Integer multiplication	~~B.~~	Ford-Fulkerson algorithm
$\boxed{H}$ ✓	Finding a minimum spanning tree	C.	Dynamic programming
$\boxed{F}$ ✓	All-pairs shortest paths	D.	General matching
$\boxed{I}$ ✓	Polynomial identity testing	~~E.~~	Divide and conquer
$\boxed{A}$ ✓	SubsetSum is NP-hard	~~F.~~	Johnson algorithm
D $\boxed{C}$ ✗	Chinese postman tour	~~G.~~	Dijkstra
$\boxed{B}$ ✓	Finding a minimum cut	~~H.~~	Greedy algorithm
$\boxed{G}$ ✓	Single-source shortest paths	~~I.~~	Monte Carlo algorithm

7/9

(b) [6 points] Suppose that you are given an unsorted array A of n integers, some of which may be duplicates. Explain how you could "uniquify" the array (that is, output another array containing each unique element of A exactly once) in $O(n)$ expected time.

Create a hash table for storing found A[i] values, Loop through the array and for each index, do a table lookup to see if the element is in the ~~array~~ (O(1)), if it is, skip to the next index. If it isn't, insert it into the table (O(1)) and add the element to A!

Total algorithm runtime O(n)

6/6

(c) [6 points] Prove that there is no polynomial-time $(1 + \frac{1}{2n})$-approximation algorithm for Vertex Cover (unless P = NP).

*

0/6

X

(d) [6 points] The following table gives the frequencies of the characters of an alphabet.

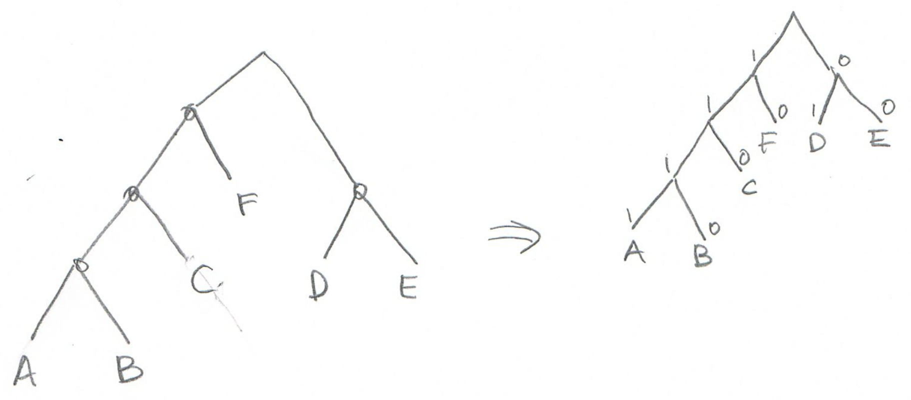
Character	Frequency
A	1/20
B	2/20
C	2/20
D	4/20
E	4/20
F	7/20

−5

Show a tree that Huffman's algorithm could produce for these characters and frequencies, and fill in the table below with the codeword for each character in the alphabet produced by this tree.

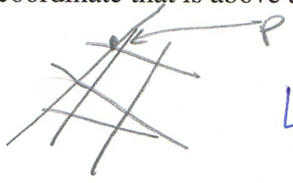
Character	Codeword
A	1111
B	1110
C	110
D	01
E	00
F	10

✓ 6/6

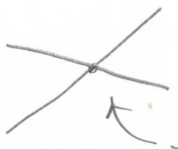


(e) [6 points] A nonvertical line L in the plane can be represented by an equation $y = m_L x + b_L$ for real numbers m_L, b_L. A point $P = (x_P, y_P)$ is ***above*** a line L if $y_P \geq m_L x_P + b_L$.

Given n nonvertical lines $L_1, L_2, \ldots, L_n$ in the plane, describe how to find in linear time the point P of minimum y coordinate that is above all n lines.



Linear Programming



the highest point must be on the intersection of two lines

Maintain 4 bounds and the ~~[illegible]~~ high point so-far

1) largest slope (+) → L_{m+}
2) smallest slope (−) → L_{m-}
3) largest b (+) → L_{b+}
4) smallest b (−) → L_{b-}

We then traverse the list of lines, if a line has a larger slope than L_+ and a smaller b, compare the point to current P. If it has a smaller slope than L_- and a larger b, compare the point to current P. If it has a larger b than b+ and a smaller slope, compare the point to current P. If it has a smaller b than b− and a larger slope, compare the point to current P. In each case, update the 4 benchmarks and P with their respective max/min and return the resulting value, This takes linear tim as each line is accessed a constant number of times.

Problem 3. Hadamard chronicles IV: Divide and conquer [15 points]

For each nonnegative integer k, the ***Hadamard matrix*** H_k is the $2^k \times 2^k$ matrix defined recursively as follows:

- $H_0 = [1]$.
- For $k > 0$, $H_k = \left[\begin{array}{c|c} H_{k-1} & H_{k-1} \\ \hline H_{k-1} & -H_{k-1} \end{array}\right]$.

Let $\vec{v}$ be a column vector of length $n = 2^k$. Describe an algorithm that computes the product $H_k\vec{v}$ in $O(n \log n)$ arithmetic operations (additions, subtractions, multiplications, or divisions). Show that your algorithm achieves the stated complexity.

$$\left[\begin{array}{c|c} H_{k-1} & H_{k-1} \\ \hline H_{k-1} & -H_{k-1} \end{array}\right]\left[\begin{array}{c} v_1 \\ \hline v_2 \end{array}\right] = \begin{array}{l} [H_{k-1}][v_1] + [H_{k-1}][v_2] = [v_1+v_2][H \\ [H_{k-1}][v_1] + [-H_{k-1}][v_2] = [v_1-v_2][H \end{array}$$

Algorithm:

Divide the problem into two subproblems

$$[H_{k-1}][v_1+v_2] \text{ and } [H_{k-1}][v_1-v_2]$$

✓

Merge by appending the arrays

$$\begin{bmatrix} [H_{k-1}][v_1+v_2] \\ [H_{k-1}][v_1-v_2] \end{bmatrix}$$

$\frac{15}{15}$

The algorithm represents the recurrence:

$$T(n) = \underbrace{2T(n/2)}_{\text{solve each subproblem}} + \underbrace{2n}_{\text{add the } v_1 +/- v_2 \text{ vector halves}}$$

✓

which by the master method simplifies to

$O(n \lg n)$ ✓

Problem 4. Biggest. Cut. Ever. [10 points]

Given an undirected graph $G = (V, E)$ and two vertices $s, t \in V$, a ***maximum s-t cut*** is a cut (S, T) satisfying the following conditions:

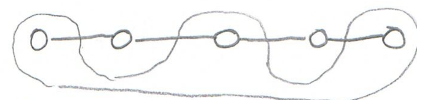
i. (S, T) is a cut: $S, T \subset V$, $S \cap T = \emptyset$, and $S \cup T = V$.

ii. $s \in S$ and $t \in T$.

iii. The number of edges $(u, v) \in E$ with $u \in S$ and $v \in V \setminus S$ is the maximum possible.

The MAXIMUM-s-t-CUT problem is to find a maximum cut for a given pair of vertices. Unlike its counterpart, the MINIMUM-s-t-CUT problem, MAXIMUM-s-t-CUT is NP-hard. Analyze the following algorithm and show that it is a 2-approximation algorithm for MAXIMUM-s-t-CUT.

```
MAX-CUT($G, s, t$)
 1  $S \leftarrow \{s\}$
 2  $T \leftarrow \{t\}$
 3  for each vertex $v \in V - \{s, t\}$
 4        do
 5            $a \leftarrow$ the number of edges $(u, v)$ with $u \in S$.
 6            $b \leftarrow$ the number of edges $(v, w)$ with $w \in T$.
 7            if $a > b$
 8              then
 9                   $T \leftarrow T \cup \{v\}$
10              else
11                   $S \leftarrow S \cup \{v\}$
12  return $(S, T)$ as the approximation for the MAXIMUM-$s$-$t$-CUTof $s$ and $t$.
```

The maximum possible cut would be for all edges to lie along the cut (i.e.: 

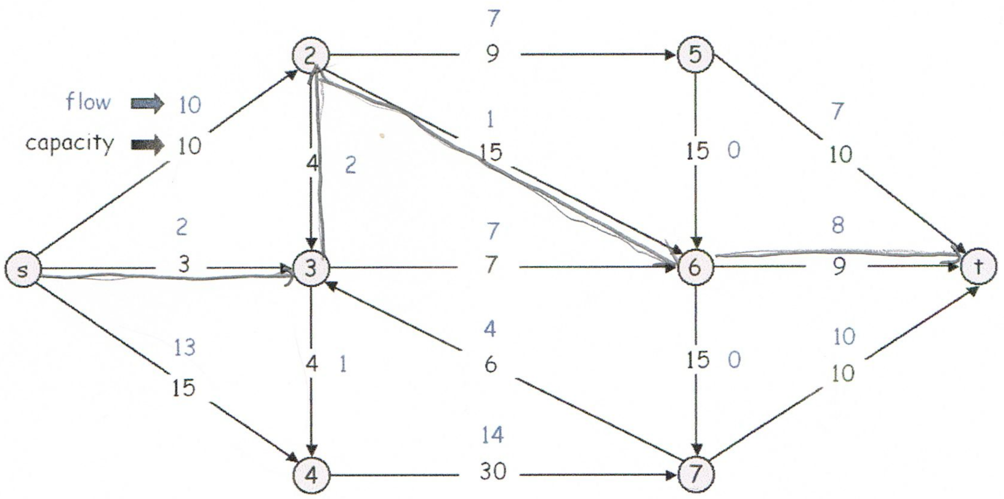
This algorithm places at least half of the edges into the cut since if ~~the~~ number of edges which connect to S are greater than the number which connect to T we put it in T (and vice versa ensuring each vertex added to one side of the cut contributes a majority of edges.

✓

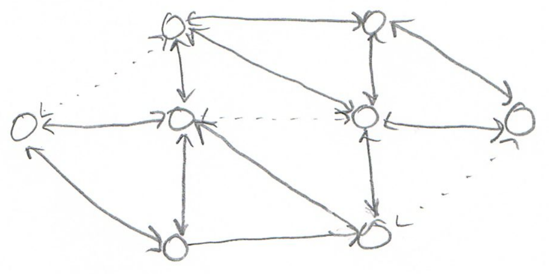
Because it is at worst half as many as the max cut, this algorithm is a 2-approximatio algorithm.

Problem 5. Be the computer [10 points]

Starting from the following flow (printed above or to the right of the capacities), perform one iteration of the Edmonds-Karp algorithm.



(a) [4 points] Write down your shortest augmenting path, that is, the augmenting path with the fewest possible edges.



Shortest augmenting path:

$s \to 3 \to 2 \to 6 \to t$

with an augmented flow of 1

(b) [3 points] Perform the augmentation. What is the value of the resulting flow?

Flow $= 10 + 9 + 7 = 26$

✓

(c) [3 points] Is the resulting flow optimal? If so, give a cut whose capacity is equal to the value of the flow. If not, give a shortest augmenting path.

No.

$s \to 4 \to 3 \to 2 \to 5 \to t$

can be augmented with another 1 unit of flow.

✓

Problem 6. Graphs and paths and cycles, oh my! [15 points]

Given a directed graph $G = (V, E)$, a ***Hamiltonian path*** is a path that visits each vertex in G exactly once. Consider the following properties for a directed graph G:

- $P_1(G)$: G contains either a cycle (not necessarily Hamiltonian) *or* a Hamiltonian path (or both).
- $P_2(G)$: G contains both a cycle (not necessarily Hamiltonian) *and* a Hamiltonian path.

Given that the problem HAM-PATH (which decides whether a graph G has a Hamiltonian path) is NP-complete, prove that one of the two properties above is decidable in polynomial time, while the other property is NP-complete.

$P_1(G)$ is decidable in polynomial time. First we check whether G has any cycles by adding weights to each edge of -1 and running Bellman-Ford to see if a negative weight cycle exists. If a cycle doesn't exist, we can find a Hamiltonian path by picking any vertex and traversing in both directions, if we encounter any branching factor, the graph is a tree and therefore can't have a ham-cycle. If we do not, then the Ham-Cycle is simply from one end-point to another.

```
Algorithm :   if (Bellman Ford w/ -1 uniform weights
                 is cycle   return true.
              else (pick vertex and traverse both ends
                    if branching factor encountered
                    return false, otherwise HAM-cyc
                    is end-to-end)
```

$P_2(G)$: This is NP-Complete because if there did exist a P-time algorithm to solve P_2, we could decide all instances of the HAM-PATH problem by running:

```
if G has a cycle (P-time)
   return P2(G)
else
   return P1(G)
```

Problem 7. If I only had a black box... [10 points]

Suppose you are given a magic black box that, in polynomial time, determines the *number of vertices* in the largest complete subgraph of a given undirected graph G. Describe and analyze a polynomial-time algorithm that, given an undirected graph G, computes a complete subgraph of G of maximum size, using this magic black box as a subroutine.

Algorithm:

Go through all vertices in the graph and, for each, add a temporary adjacent vertex (adjacent only to that vertex) and use BlackBox(G'). If it is 1 larger than BlackBox(G), we know the vertex lies on at least one subgraph of maximal size, in which case add it to a separate subgraph H, otherwise do nothing. This will create a graph containing all maximal subgraph So we simply need to pick a random vertex, compute all-pairs shortest paths and discard all vertices which are unreachable to find a single set of connected vertice of maximal size.

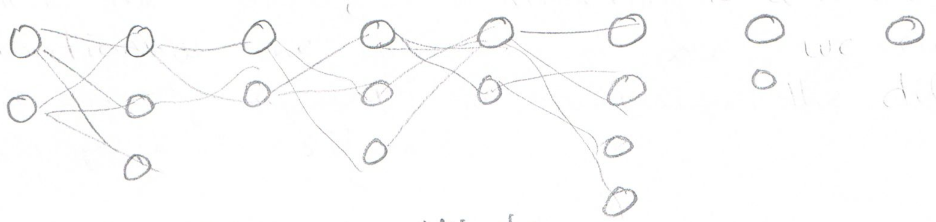
The running time of this algorithm is $O(V)$ to compute H and $O(V^3)$ to pick one subcomponent, therefore running time is O

Problem 8. Hero Training [19 points]

You are training for the World Championship of Guitar Hero World Tour, whose first prize is a *real guitar*. You decide to use algorithms to find the optimal way to place your fingers on the keys of the guitar controller to maximize the ease by which you can play the 86 songs.

Formally, a ***note*** is an element of $\{A, B, C, D, E\}$ (representing the green, red, yellow, blue, and orange keys on the guitar). A ***chord*** is a nonempty set of notes, that is, a nonempty subset of $\{A, B, C, D, E\}$. A ***song*** is a sequence of chords: $c_1, c_2, \ldots, c_n$. A ***pose*** is a function from $\{1, 2, 3, 4\}$ to $\{A, B, C, D, E, \emptyset\}$, that is, a mapping of each finger on your left hand (excluding thumb) either to a note or to the special value $\emptyset$ meaning that the finger is not on a key. A ***fingering*** for a song $c_1, c_2, \ldots, c_n$ is a sequence of n poses $p_1, p_2, \ldots, p_n$ such that pose p_i places exactly one finger on each note in c_i, for all $1 \le i \le n$,

You have carefully defined a real number $D[p, q]$ measuring the difficulty of transitioning your fingers from pose p to q, for all poses p and q. The ***difficulty*** of a fingering $p_1, p_2, \ldots, p_n$ is the sum $\sum_{i=2}^{n} D[p_{i-1}, p_i]$. Give an $O(n)$-time algorithm that, given a song $c_1, c_2, \ldots, c_n$, finds a fingering $p_1, p_2, \ldots, p_n$ of the song with minimum possible difficulty.



We can solve this as a Viterbi lattice where each pose which matches a chord is a vertex in a column and connects to all subsequent poses in the next time interval of the lattice. We then traverse the lattice twice, first we go through computing the best value for each pose (correspond to the easiest transition + sum of all past transition as well as a predecessor table which stores the easiest path up to that pose. Then when we hit the final pose we select the minimum difficu and retrieve the linked list of predecessors to recover the fingering, which requires O(n) time if the number of poses per chord has a constant maximum.