STANFORD UNIVERSITY
CS 161, Summer 2004
Final Examination

Question	Points
1 Algorithm Design	/ 64
2 Augmenting Skip Lists	/ 22
3 Shortest Paths	/ 21
4 Boolean Chains	/ 13
5 Dynamic Hash Tables	/ 12
6 Dynamic Transitive Closure	/ 18
7 Short Answer	/ 30
Total	/180

Name (print): _____

Honor Code Statement:

I attest that I have not given or received aid in this examination, and that I have done my share and taken an active part in seeing to it that others as well as myself uphold the spirit and letter of the Honor Code. Specifically, I attest that I have not had any advance knowledge of the questions on this examination, and I have not enabled anyone to gain such knowledge.

Signed: _____

Answer Guidelines

- These guidelines cover the default assumptions that should be made when answering a question. **In all cases, explicit instructions given in the question supersede these guidelines.**
- Try to keep your answers to the space provided. If you need additional space, use the blank page at the end of the exam and/or the back side of normal pages. If you do, please tell us where to find the continuation of your answer and clearly label the answer text with the corresponding question number and part.
- Your answers need to be complete and **concise**. Unless explicitly told otherwise, you must provide justification for your answers, but are not required to formally prove them. If you're asked to give an algorithm and its running time, you do not need to prove correctness or do a formal running time analysis; a brief explanation for both is sufficient.
- You may use anything proven in class, in the textbook, or on the homework without proof, unless explicitly asked to prove it.
- When your solution refers to an algorithm or data structure we covered in class, please be sure to specify all the relevant parameters. For instance, specify whether you're using randomized or deterministic Select, Quicksort; MIN or MAX heap; chaining or open addressing collision resolution for hash tables, number of slots in the hash table; skip list propagation parameter; and so on. Every factor that effects performance (or correctness) should be addressed, although it's acceptable to say, for instance, "using universal hashing" without explicitly specifying a universal hash function family.
- When answering with a running time or space bound please be sure to specify whether it's a worst-case, expected, or amortized bound. Unless stated otherwise, any type of bound is acceptable. If giving an expected-case bound, be careful not to make any assumptions about the distribution of the input(s) - the expectation should be over randomization internal to the algorithm or data structure.
- You may assume that all elements' keys are comparable using $\leq$. **Other than that, you should NOT assume any properties of elements, keys, or their distribution**, unless explicitly specified.
- Assume graphs are given using the adjacency list representation unless specified otherwise. If you are asked to formulate a graph, you may use whichever representation is convenient.

1. **[64 points]** Algorithm Design

   Please read these directions **carefully**, they contain several hints and are crucial to approaching this question effectively.

   In this question, you will be given six optimization problems and asked to give a polynomial time algorithm to determine the value of the optimal solution, if possible. Note that your algorithm only needs to find the **value** of the best solution. **Two of the problems can be solved using a greedy approach, three can be solved using dynamic programming, and one problem is NP-Complete.** For each of the five problems that have a polytime algorithm, you need to **give an algorithm** (pseudocode is more than enough) and **state its running time** (no formal analysis needed). You should try to **give the most efficient algorithm possible**; assume that whenever it is correct, a greedy algorithm will be more efficient than DP.

   Any algorithm you give should run in time polynomial in n, under the assumption that all other variables are polynomially bounded as a function of n. Give the running time in terms of the relevant variables.

   If your approach is greedy, prove that your greedy algorithm is optimal. The proof does not need to be formal, but you should clearly explain why the greedy choice property holds. **If your approach is dynamic programming, argue informally why the optimal substructure exploited by your algorithm is present.** You do NOT need to explain why the greedy approach fails, although it may be helpful to think about that when analyzing the problem initially. **If you believe the problem is NP-Complete, just state so.** No proof or justification is necessary; do not spend your time trying to come up with a reduction (in fact, the formal reduction for this particular NP-Complete problem is nontrivial).

   You may use any result proven in class, in the textbook, on the homeworks, or on the midterm without proof. In particular, you may use any of the algorithms we've covered as a subroutine. In fact, at least one of the problems has a simple solution that takes advantage of this.

   Several of the problems are variations of problems you've seen in this class. Be careful not to assume that the same technique still applies, or that a more efficient technique does not. Pay close attention to the differences, they are important.

   Each of the five algorithms is worth 12 points; you will receive 4 points for correctly identifying the NP-Complete problem. Even if you are unable to come up with an algorithm for a problem, you should at least try to determine which approach is likely to work. We will give partial credit for correctly labeling each problem as "Greedy", "DP", or "NP-Complete".

   This question is meant to be long on thinking, short on writing. Your explanations should be as brief as possible. **Please do not try to fill up all the space provided.**

(a) Consider the following variation of the gas station problem. You are an eccentric billionaire planning a cross-country trip along a straight highway with $n+1$ gas stations. You start at gas station 0, and the distance to gas station i is d_i miles ($0 = d_0 < d_1 < \ldots < d_n$). Your car's gas tank holds $G > 0$ gallons and your car travels $m > 0$ miles on each gallon of gas. You start with an empty tank of gas (at gas station 0), and your destination is gas station n. You may not run out of gas, although you may arrive at a gas station with an empty tank.

As you are very rich, rather than trying to plan the cheapest trip, you want to minimize the total number of stops you need to make (you stop at a gas station only if you need to buy gas there).

Assume that G, m, and all d_i are positive integers (except $d_0 = 0$) and are polynomially bounded as a function of n. Assume all d_i are distinct.

If possible, give a polynomial time algorithm to determine the value of the optimal solution (i.e. the minimum number of stops).

(b) You have n books on your bookshelf, and you've decided to alphabetize them by their title. A *move* consists of pulling out some book $B[i]$ and inserting it between any two books on the shelf.

You want to compute the minimum number of such moves needed to alphabetize the books on the shelf.

Assume each book $B[i]$ has a distinct title and two titles can be compared in constant time.

If possible, give a polynomial time algorithm to determine the value of the optimal solution (i.e. the minimum number of moves).

(c) You have a sequence S of n characters from an alphabet of size $k \leq n$; each character may occur many times in the sequence. You want to find the longest subsequence of S where all occurrences of the same character are together in one place; for example, if $S = aaaccaaaccbccbbbab$, then the longest such subsequence is $aaaaaaccccbbbb = aaa\_aaacc\_ccbbb\_b$. In other words, any alphabet character that appears in S may only appear in one contiguous block in the subsequence. If possible, give a polynomial time algorithm to determine the value of the optimal solution (i.e. the length of the longest such subsequence).

(d) Consider the following variation of the subset sum problem. You have an array A of n distinct nonnegative integers and you wish to determine the smallest subset of A (if any) that adds up to some positive integer $x > 0$. Each element of A may be used at most once in the sum.

Assume x is polynomially bounded as a function of n. Assume that all $A[i]$ are distinct **nonnegative** integers.

If possible, give a polynomial time algorithm to determine the value of the optimal solution (i.e. the size of the minimum subset, or ∞ if none exists).

(e) You have a rod of adamantium metal of length L meters and you want to cut it in n specific places: $L_1, L_2, \ldots, L_n$ meters from the left end. A mad scientist charges x dollars to cut an x-meter rod any place you want.

Assume L is polynomially bounded as a function of n. Assume that all L_i are distinct real numbers strictly between 0 and L, i.e. $0 < L_1 < \ldots < L_n < L$. You should **not** assume L_i are integers.

If possible, give a polynomial time algorithm to determine the value of the optimal solution (i.e. the minimum cost in dollars).

(f) You are designing an online dating website and currently there are n boys and n girls that need to be matched up. Unfortunately, due to a database crash the only information you have about them is their heights: an array B specifying the heights of the n boys and an array G specifying the heights of the n girls.

You want to match them up in a way that will minimize the largest height difference within any couple. Formally, you want to **minimize** the quantity $\max_{j=1}^{n} |b_j - g_j|$, where (b_j, g_j) is the j^{th} couple.

Assume that all the $B[i]$ and $G[i]$ are positive real numbers that are polynomially bounded as a function of n. You should **not** assume they are integers.

If possible, give a polynomial time algorithm to determine the value of the optimal solution (i.e. the smallest possible maximum height difference).

2. **[22 points]** Augmenting Skip Lists

   In class we saw how to augment red-black trees to efficiently maintain dynamic order statistics. In this problem, you will augment skip lists to accomplish the same purpose. Throughout this problem, assume that the propagation parameter $p = 1/2$.

   Recall that in order-statistic trees, each node kept a field $size[x]$ that equals the size of the subtree rooted at x. In an order-statistic skip list, we will keep a similar field for each cell in the skip list.

   To reestablish the notation we used in the handout and on the problem sets, recall that a skip list consists of L levels, each of which is a linked list. The bottom level is level 0, and contains all the elements in the list. Each cell contains the pointers *next*, *below*, and *above* as well as the field *key*. In this problem, we will also assume all cells contain the pointer *prev*, so that each list is in effect doubly linked. Finally, each linked list also has a dummy cell at the beginning and end to indicate the beginning and end of the list. These cells have key $-\infty$ and ∞ respectively. We assume all elements in the list have distinct keys.

   For every cell c, we add a new field $c.size$ which equals the number of elements in the skip list whose key is less than $c.key$ but greater than $c.prev.key$; in other words, if c is a node in level h, then $c.size$ represents the **number of elements in the skip list between** $c.prev$ **and** c **that were not propagated to level** h.

   In answering the questions below, you do not need to give a detailed implementation of each function; the only things that should be specified precisely are operations that involve the new *size* field. **You should NOT give complete pseudocode**; in particular, feel free to ignore boundary cases (involving the dummy cells) if that simplifies the explanation. The main idea behind each implementation can be explained using a clear and informative diagram and a few English sentences. Be sure to justify the (expected) time bounds.

(a) **[6 points]** Sketch how you would implement the OS-SELECT(i) operation, which returns the i^{th} smallest element in the skip list. Explain why your algorithm runs in expected time $\Theta(\lg n)$.

(b) **[6 points]** Sketch how you would implement the OS-RANK(x) operation, which returns the rank of (number of elements less than or equal to) a given element x in the list. Explain why your algorithm runs in expected time $\Theta(\lg n)$.

(c) **[10 points]** Sketch how you would maintain the *size* field without affecting the asymptotic running time of the INSERT and DELETE operations.

3. **[21 points]** Shortest Paths

   You are traveling in a galaxy whose planets are positioned on a $n \times n \times n$ cubic grid: there is a planet at every lattice (integer) location (a, b, c) for $1 \leq a, b, c \leq n$. You are trying to get from planet Earth located at $(1, 1, 1)$ to planet Hoth located at (n, n, n) as quickly as possible.

   Travel is possible between planets that are exactly one unit apart, i.e. you may travel between (a, b, c) and $(a, b-1, c)$ but NOT between (a, b, c) and $(a-1, b+1, c)$. Travel between such neighboring planets takes exactly one day. However, there are two additional factors to consider.

   First, some planets have worm holes that allow you to jump to other planets, potentially far far away. Assume your pet robot has a list of W existing worm holes, where $W = O(1)$. Each worm hole w allows you to travel from its start location to its end location in time t_w. We assume this travel time is an integer number of days, although it **may be negative or zero**. Also, we assume that it is impossible to travel back in time infinitely by using these worm holes.

   Second, some planets are controlled by the evil empire and therefore travel through them is restricted. The empire maintains very tight control within its borders, and therefore travel between any two empire-controlled planets is impossible without getting caught. However, you may travel from a free planet to an empire-controlled planet (or vice versa) in one day as usual. Assume that your pet robot's galactic database tells you whether each planet is free.

   Your goal is to get from Earth to Hoth in the shortest amount of time possible, given the travel constraints imposed by the evil empire, and possibly taking advantage of the worm holes.

(a) **[6 points]** Formulate this problem as a single-source shortest-path problem in a graph G. State precisely what the nodes and edges of G are, whether G is weighted or unweighted, directed or undirected. If your graph is weighted, all edge weights should be integer and finite. Also, your formulation should not be a multigraph; be sure that there is only one edge between u and v if G is undirected, or only one edge from u to v if G is directed. Finally, state which graph representation (adjacency list or adjacency matrix) you would use in this problem, and why.

(b) **[4 points]** Which algorithm would you use to solve the problem you formulated in part (a)? Justify your choice. Give a tight bound on its running time in terms of n.

(c) **[3 points]** Your pet robot just integrated the worm hole information with the list of empire-controlled planets, and concluded that the empire controls the start and end planets of all worm holes with $t_w < 0$. How does this change your answer to part (b)? Justify and give the new running time in terms of n, if it changed.

(d) **[4 points]** You examine the data and realize things are even worse than you thought. In fact, the empire controls the start and end planets of **all** worm holes. How does this change your answer to part (c)? Justify and give the new running time in terms of n, if it changed.

(e) **[4 points]** For your success, you've been put in charge of a galaxy-wide smuggling operation. You now need to compute the fastest way to smuggle goods from any planet A to any planet B. Which algorithm would you use to solve this problem and why? Justify your answer by comparing the running times of the algorithms we've seen in class for this problem. **Do NOT make assumptions from parts (c) and (d) about inaccessiblity of worm holes.**

4. **[13 points]** Boolean Chains

   A boolean chain is a sequence of n boolean random variables $X_1, \ldots, X_n$ such that the value of X_i depends only on the value of X_{i-1}. Graphically we can represent the chain as shown:

   $$X_1 \longrightarrow X_2 \longrightarrow \ldots \longrightarrow X_n$$

   Each arrow represents a probabilistic dependence of X_i on X_{i-1}; this dependence consists of four probabilities:

   - $Pr[X_i = T \mid X_{i-1} = T]$
   - $Pr[X_i = T \mid X_{i-1} = F]$
   - $Pr[X_i = F \mid X_{i-1} = T]$
   - $Pr[X_i = F \mid X_{i-1} = F]$

   For the first node in the chain we simply have $Pr[X_1 = T]$ and $Pr[X_1 = F]$.

   Notice that the probability of any assignment $(X_1 = x_1, X_2 = x_2, \ldots, X_n = x_n)$, where each x_i is either T or F, is therefore equal to

   $$Pr[X_1 = x_1] \cdot Pr[X_2 = x_2 \mid X_1 = x_1] \cdot \ldots \cdot Pr[X_n = x_n | X_{n-1} = x_{n-1}]$$

   (a) **[2 points]** Give the basic constraints on these probabilities, beyond the fact that all probabilities are between 0 and 1.

(b) **[6 points]** Consider the problem of finding the most likely truth assignment to the variables $X_1, \ldots, X_n$. Formulate this problem as a single-source shortest path problem in some graph G. Be explicit about what the nodes and edges of G are, and what the weights on these edges should be. What algorithm would you use to find the shortest path and what is its running time as a function of n?

Hint: It may be helpful to introduce an auxilliary variable X_0 which is always true, so that X_1 may be treated almost uniformly with $X_2, \ldots, X_n$.

(c) **[5 points]** Give a dynamic programming algorithm to find the most likely assignment in time $\Theta(n)$.

5. [**12 points**] Dynamic Hash Tables

   In order to maintain expected $O(1)$ performance for hash table operations in an m-bucket hash table that holds n entries, the load factor $\alpha = n/m$ must be $O(1)$. However, what if we don't know in advance how many items the table needs to hold? Choosing a large m means wasting memory, choosing a small m means poor performance when $n = \omega(m)$. Thus, m will have to grow dynamically.

   We will analyze a hash table that uses chaining to resolve collisions. Thus, the table is an array of buckets, each of which is a linked list. Elements are inserted at the beginning of the list corresponding to the bucket they hash to; empty buckets are initialized to NIL. We assume the hash function $h(x, m)$ satisfies the assumptions of simple uniform hashing.

   Throughout this problem, assume $h(x, m)$ takes $c_h = O(1)$ units of time to compute, while arithmetic operations, memory accesses, and memory allocations take 1 unit.

   (a) [**2 points**] Explain why we cannot simply grow the bucket array every time we need to expand the hash table, without rehashing all the elements in the table.

   Our goal will be to maintain the property that $\alpha \leq 2$. We will not worry about reclaiming memory if too many elements are deleted.

   We say the hash table is *full* when $n = 2m$. If we try to insert into a full hash table, we need to grow the table in order to maintain the constraint on α. Thus, prior to inserting we will do the following:

   - allocate a new array of buckets with $m' = 2m$
   - for each element x in the hash table, insert x into new bucket $h(x, m')$
   - free the old hashtable (ignore this cost in your analysis)

   The second and third steps above are done by iterating over the linked list within each bucket.

(b) [**1 point**] What is α' right after the reallocation but prior to inserting the next element? No explanation required.

(c) [**7 points**] Analyze the performance of the INSERT operation (including reallocation when the table is full) using the amortized analysis method of your choice. Your analysis does not need to be completely formal, but should be detailed. Ignore the cost of deallocating memory in your analysis.

(d) [**2 points**] Are we restricted to using the same function h when growing the hash table, or can we switch from $h(x, m)$ to some other hash function $h'(x, m')$, if $h'(x, m')$ satisfies the assumptions of simple uniform hashing? Explain.

6. **[18 points]** Dynamic Transitive Closure

   Consider the problem of maintaining the dynamic transitive closure of a graph G as edges are inserted into G. As on the homework, assume that each edge is inserted at most once, and we want an $O(V^3)$ bound on the total time needed to maintain the transitive closure G^*.

   Consider the following algorithm. At every point, we keep the adjacency matrix A^* of G^*. For every edge (u, v) added to G, if the edge is already in G^*, we do nothing. Otherwise, we compute the following sets of nodes:

   - $P(u) = \{x : (x, u) \in E^*\}$, i.e. predecessors of u
   - $S(u) = \{x : (u, x) \in E^*\}$, i.e. successors of u
   - $P(v) = \{x : (x, v) \in E^*\}$, i.e. predecessors of v
   - $S(v) = \{x : (v, x) \in E^*\}$, i.e. successors of v

   Now, for each edge (x, y) such that $x \in P(u) - P(v)$ and $y \in S(v) - S(u)$, add the edge (x, y) to G^* if it's not already there.

   (a) **[4 points]** Argue (informally) that this algorithm is correct. That is, show why adding the edge $u \to v$ will only add edges from $P(u)$ to $S(v)$, and then show the edges we don't consider must already be present in G^*.

   (b) **[2 points]** Show how to compute these sets in time $O(V)$ using the adjacency matrix A^*. Make sure to explain why the set difference may be computed in $O(V)$ time here.

Now suppose (u, v) is the edge we are adding to G^*. We want to show that we don't consider too many edges that are already present in G^*.

(c) **[3 points]** Argue that if we consider adding (x, y), then (x, v) and (u, y) are absent from G^* but will be added when we add (u, v).

(d) **[9 points]** Now, use the accounting method to show that this implies that the total time needed to process all the updates is $O(V^3)$. Don't forget the time necessary to compute $P(u) - P(v)$ and $S(v) - S(u)$.

Hint: Charge 0 for considering edges; but charge enough for adding an edge to G^* to account for the actual cost of reconsidering edges. Think about how many edges (x, y) are considered and how many edges are added while adding (u, v) to the graph.

7. **[30 points]** Short Answers

   For true/false questions, be sure to provide a detailed and complete (but not necessarily long!) explanation for your answer. We will only award points for justified answers and explanations in order to discourage random guessing.

   (a) **[2 points]** Give an upper and lower bound on the number of strongly connected components of a directed acyclic graph $G = (V, E)$ in terms of $|V|$ and/or $|E|$. Your bounds should be exact rather than asymptotic, and as tight as possible.

   (b) **[4 points]** Given an unsorted array A of n elements, it is possible to construct a binary search tree T from these elements in time $\Theta(n)$ if we do not require T to be balanced.

   A: True / False **Explanation:**

   (c) **[4 points]** In a binary search tree of height h, we can implement the DECREASE-KEY operation using $O(h)$ rotations and no other operations.

   A: True / False **Explanation:**

(d) **[5 points]** You have a directed acyclic graph $G = (V, E)$. Suppose some subset $V' \subseteq V$ is colored green. We want to color green any ancestor of a node in V'. Give an efficient algorithm to do this and analyze its running time in terms of $|V|$ and/or $|E|$.

(e) **[4 points]** Recall that a vertex cover of an undirected graph $G = (V, E)$ is a subset $V' \subseteq V$ such that for any edge $(u, v) \in E$, we have $u \in V'$ or $v \in V'$ (or both). Suppose that you have an algorithm to solve the vertex cover **decision** problem (deciding whether G has a vertex cover of size $\leq k$) that runs in time in time $f(|V|, |E|)$. How fast can you solve the corresponding **optimization** problem (finding the size of the minimum vertex cover)?

(f) **[5 points]** Recall that in the bipartite matching problem, the Ford-Fulkerson flow algorithm had a running time $O(VE)$ because $|M| = |f^*| \leq |V|/2$. The Edmonds-Karp flow algorithm has running time $O(VE^2)$. Thus, for bipartite matching, Ford-Fulkerson is asymptotically faster than Edmonds-Karp unless $|E| = O(1)$.

A: True / False **Explanation:**

(g) **[6 points]** Suppose you have an undirected graph G where all the edge weights are integers between 0 and $|E|$. How would you find the minimum spanning tree of G? Analyze the running time of your algorithm.

Additional Answer Space