Massachusetts Institute of Technology
6.046J/18.410J: Introduction to Algorithms
Professors Piotr Indyk and Bruce Tidor

Quiz 2
15 April 2003

Quiz 2

- Do not open this quiz booklet until you are directed to do so.
- This quiz ends at 3:55 P.M.
- When the quiz begins, write your name on the top of ***EVERY*** page in this quiz booklet, because the pages will be separated for grading.
- Write your solutions in the space provided. If you need more space, write on the *back* of the sheet containing the problem. Do not put part of the answer to one problem on the back of the sheet for another problem.
- Plan your time wisely. Do not spend too much time on any one problem. Read through all of them first and attack them in the order that allows you to make the most progress.
- Show your work, as partial credit will be given. You will be graded not only on the correctness of your answer, but also on the clarity with which you express it. Be neat.
- When describing an algorithm, describe the main idea in English. Use pseudocode only to the extent that it helps clarify the main ideas.
- Good luck!

Problem	Points	Grade
1	30	28
2	15	10
3	15	15
4	20	10
Total	80	63

Name: DAN PORTS

Please circle your TA's ~~name and~~ recitation:

Nitin **Brian** **Mihai** **Yoav**

10am **11am** **12pm** **1pm** **2pm**

Problem 1. True or False, and Justify **[30 points]** (6 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 points than your true-or-false designation.

T **F** Assume you are given a magical priority-queue data structure, which performs find-min, insert and decrease-key in constant time. Then you can implement Dijkstra's algorithm to run in $O(E+V)$ time.

5 Dijkstra's algorithm requires E extract-min calls and V decrease-key calls and V insert calls. If they can be done in O(1) time, then Dijkstra's runs in O(E+V)

T **F** Yoav claims the following modification to Dijkstra's shortest paths algorithm will allow it to work for negative edge weights as well. The idea is to find the smallest edge weight, say $w < 0$, and add $-w$ to all the edges, thus making all of them non-negative. Now run Dijkstra's algorithm on these new edge weights. To argue correctness, Yoav claims that the addition doesn't change the relative order between edges, ie, if $a < b$ then $a + w < b + w$.

5

Yoav's algorithm for the shortest paths problem is correct.

Consider a graph w/ a negative-weight cycle:

The shortest path found by Yoav's algorithm has length -3, but an infinitely short path can be found by repeatedly traversing the cycle.

T **F** Yoav claims that this approach works for MST as well. I.e., after making the above changes to the weights and running any MST algorithm, the algorithm outputs correct MST for the original graph.

Yoav's algorithm for finding MST is correct.

3

Kruskal's algorithm depends only on the ordering of edges, not their actual weights. Yoav's approach does not change the ordering, so Kruskal produces the correct output.

T **F** Consider the family $\mathcal{H}$ of hash functions that contains all functions $h_{a,b} : \{0 \ldots 10q\} \to \{0 \ldots q-1\}$ of the form $h_{a,b}(x) = x + a + b^2 \mod q$, where $a, b \in \{0 \ldots q-1\}$, q prime.

The family $\mathcal{H}$ is universal.

5

$h(x) = h(y)$

iff $x + a + b^2 \mod q = y + a + b^2 \mod q$

$\Rightarrow x + a + b^2 - (y + a + b^2) \mod q = 0$

$\Rightarrow x - y \equiv 0 \mod q$

This does not depend on a or b, so it cannot be universal. q and q^2 map to the same value in every hash function.

T **F** Consider a complete directed graph $G(V, E)$ over the vertex set $V = \{1 \ldots 100\}$. That is, all edges $(i, j), i, j \in V, i \neq j$ are present in G. Assume that the capacity of each edge is equal to 1.

The maximum flow from the source vertex 1 to the sink vertex 2 has value 100.

5 There are 100 vertices total. There are 99 edges leading from the source to every other vertex. We can saturate those w/ 1 unit of flow each, then transmit that flow to the sink through the edge from the intermediate node to the sink. This creates a flow of 99, which is the max because the residual net has no edges leaving the source and thus no augmenting paths.

T F Assume you are given two binary search trees T_1 and T_2. Let E_1 be the set of elements in T_1, and E_2 be the set of elements in T_2. The *union* of T_1 and T_2 is a binary search tree T that contains the elements from $E_1 \cup E_2$.

There is a linear time algorithm for computing the union of T_1 and T_2.

5 We can use an in-order walk to convert T_1 and T_2 to sorted arrays A_1 and A_2 in $\Theta(n)$ time. Then we can merge A_1 and A_2 into a new array A in $\Theta(n)$ time using the Merge algorithm. We can then convert A into a BST in $\Theta(n)$ time since A is already sorted. (Trivially, by assigning A_2 to be A_1's right child, A_3 to be A_2's right child, etc. Also possible to do this in a more complex way to maintain some semblance of balance). Then T contains $E_1 \cup E_2$.

Problem 2. Making money [15 points]

10 You have \$1000 and want to invest it for n months. At the beginning of the t-th month, $t \geq 1$, you must choose from the following three options:

- Purchase a savings certificate from the local bank. Your money will be tied up for one month. If you buy it at the beginning of the t-th month, there will be a fee of $BankFees(t)$ and after a month, it will return $BankRate(t)$ for every dollar invested. That is, if you have \$$c$ at time t, then you will have \$$(c - BankFees(t)) \cdot BankRate(t)$ at time $t+1$.
- Purchase a state treasury bond. Your money will be tied up for six months. If you buy it at the beginning of the t-th month, there will be a fee of $BondFees(t)$ and after six months, it will return $BondRate(t)$ for every dollar invested. That is, if you have \$$c$ at time t, then you will have \$$(c - BondFees(t)) \cdot BondRate(t)$ at time $t+6$.
- Store the money in a sock under your mattress for a month. That is, if you have \$$c$ at time t, then you will have \$$c$ at time $(t+1)$.

Suppose you have predicted values for BankFees, BankRate, BondFees, BondRate for the next n months. Devise an algorithm that computes the maximum amount of money that you can have after n months. Your algorithm should run in time $O(n)$.

Use a dynamic programming approach ✓ w/ recurrence:

$$M(t) = \max \begin{cases} [M(t-1) - BankFees(t-1)] \cdot BankRate(t-1) & \text{(savings certificate) requires } t \geq 2 \\ [M(t-6) - BondFees(t-6)] \; BondRate(t-6) & \text{(bond) requires } t \geq 7 \\ M(t-1) & \text{(mattress option} \end{cases}$$

Base case is $M(0) = 1000$. ✓

We can generate a solution by building an array, initializing $M[0] = \$1000$. ✓ Then iterate through i from 1 to n, filling in $M[i]$ using the above recurrence. ✓ At each step, maintain a backpointer to the

Problem 3. Scheduling [15 points]

Prof. Tidor is in charge of 26-100. There are many lecturers who want to lecture in it. He is given a list of n lectures in the following format:

"lecture i needs to run from S_i to E_i (start time and end time)"

For all lectures we have $E_i > S_i$. Note that different lectures can have different length.

Prof. Tidor wants to come up with a schedule that maximizes the number of lectures on a given day. Help the professor by designing an $O(n \log n)$-time algorithm that finds the maximum number of lectures that can be given that day. Justify the correctness of your algorithm.

Create two arrays of pointers to lecture objects. One is sorted by start time (call it S[1..n]) and one is sorted by ending time (call it E[1..n]). We use a greedy approach. Maintain a variable k (initialize it to 0) that keeps track of lectures that can't be scheduled: the lectures that appear before index k in S cannot be scheduled. Then, for i from 1 to n, find the lecture E[i]. If it is not in S[1..k-1] (we maintain pointers so we can do this check in $\Theta(1)$ time), we add lecture i to the schedule. We then use binary search on S[k..n] to find the minimum element of S such that that lecture's start time is after the end time of the lecture we just added. Set k equal

0 can be transformed to 6,
or 0 was not optimal, so 6 is optimal.

Excellent!

15/15

Problem 4. Bottleneck paths [20 points] 10

Consider the following data structure problem. Initially, you are given a threshold $T > 0$, and an undirected graph G with a set of vertices $V = \{1 \dots n\}$ and no edges. Then, you are given a sequence of requests of the following two types:

- ADD(u, v, c): add an undirected edge $\{u, v\}$ to G; the edge has capacity $c > 0$.
- TEST(u, v): check if there is a path between u and v in G such that each edge in the path has capacity at least T

Your goal is to design efficient data structure(s) for this problem. You are allowed to spend polynomial time to initialize your data structure (i.e., before any ADD or TEST operation is performed).

(a) [14 points] Assume that each edge is added at most once. Design an efficient data structure for this problem, with low *amortized* running times. For full credit, your data structure should perform both operations in amortized $o(\log n)$ time (i.e., better than $\Theta(\log n)$-time). Partial credit for less efficient data structures will be given.

(b) [6 points] Assume now that each edge can be added several times, and the capacities sum up. I.e., if we add an edge $\{u, v\}$ k times with capacities $c_1 \dots c_k$, then the capacity of the edge is equal to $\sum_i c_i$. Design an efficient data structure for this case. For full credit, the amortized time per operation should be again $o(\log n)$. Partial credit for less efficient data structures will be given.

a. For each edge, maintain a list of reachable edges, and the capacity. When adding an edge (u, v) w/ capacity c, add v to u's list w/ capacity c, then for every vertex k reachable from u, if the capacity from k to some j is less than the min of capacities from k to u, u to v, and v to j, raise the capacity from j to k to the minimum of these. Then do the same for all vertices reachable