University of New Mexico
Department of Computer Science

Midterm Examination

CS 561 Data Structures and Algorithms
Fall, 2011

Name:
Email:

- *"Nothing is true. All is permitted"* - Friedrich Nietzsche. Well, not exactly. **You are not permitted to discuss this exam with any other person.** If you do so, you will surely be smitten: collusion on any problem will result in a 0 on the entire exam. However, you may consult any non-human sources including books, papers, web pages, computational devices, animal entrails, etc. in your quest for truth. Please acknowledge your sources.
- *Show your work!* You will not get full credit if we cannot figure out how you arrived at your answer. A numerical solution obtained via a computer program is unlikely to get much credit, if any, without a correct mathematical derivation.
- Write your solution in the space provided for the corresponding problem.
- If any question is unclear, ask for clarification.

Question	Points	Score	Grader
1	20		
2	20		
3	15		
4	20		
5	25		
Total	100		

1. **Recurrences**

   Remember that when the base case for a recurrence is not explicitly given, assume that it is constant for inputs of constant size.

   (a) (5 points) Solve the following recurrence using annihilators: $f(n) = 5f(n-1) - 6f(n-2) + 2^n$. Do not solve for the constant coefficients *Solution:* $(L^2 - 5L + 6) = (L-2)(L-3)$ *annihilates the homogeneous part and* $(L-2)$ *annihilates the non-homogeneous part. So the annihilator is* $(L-3)(L-2)^2$ *and hence the solution is* $c_1 3^n + (c_2 n + c_3)2^n$

   (b) (5 points) Solve the following recurrence using a transformation and the Master method: $f(n) = 10f(\sqrt{n}) + n$. Do not solve for the constant coefficients. If an algorithm's runtime is given by this recurrence, how would it compare with algorithms with runtimes of $\theta(2^n)$, $\theta(n)$, $\theta(\sqrt{n})$, $\theta(\log n)$? *Solution: Let* $2^i = n$ *and* $F(i) = f(2^i)$. *Then we get a transformed recurrence:* $F(i) = 10F(i/2) + 2^i$. *Using the Master method, we can see that the solution to this is* $F(i) = \theta(2^i)$ *(the root node dominates). Reverse transforming this solution, and using the fact that* $i = \lg n$ *we get* $f(n) = n$.

Imagine that there is a computer virus where infected machines slowly ramp up the rate at which they infect other machines. In particular, in a given round, each machine that has been infected for exactly i rounds infects exactly $i-1$ new machines. Let $f(n)$ be the number of machines infected at round n - the base case is $f(1) = 1$. One of your coworkers claims that the recurrence relation for $f(n)$ is $f(n) = \sum_{i=1}^{n-1} f(i)$. Another coworker claims that the correct recurrence relation is $f(n) = \sum_{i=1}^{n-1} i * f(i)$.

(c) (3 points) Which coworker is correct and why? *Solution: The first coworker is correct. There are exactly $f(n-1)$ machines infected in previous rounds. Then, this relation counts each machine infected for i rounds exactly $i-1$ times.*

(d) (7 points) Find an exact solution to the first recurrence relation. Prove your answer is correct via induction. *Solution: The solution is $f(n) = 2^{n-2}$ for all $n \geq 2$.*
B.C.: $f(2) = 1 = 2^0$
I.H.: $\forall_{2 \leq j < n} f(j) = 2^{j-2}$.
I.S.: $f(n) = \sum_{i=1}^{n-1} f(i) = \sum_{i=0}^{n-3} 2^i + 1 = 2^{n-2}$, *where the second to last step holds through repeated application of the I.H.*

2. **Data Structures**

(a) (4 points) Imagine that you are storing n items in a Bloom Filter with m bits. You want the probability of a false positive to be less than 1/10000. Approximately how large must m be as a function of n? Approximately how many hash functions will you need (i.e. how big is k). Show your work! *Solution: The false positive rate is about* $.62^{m/n}$. *Setting* $.62^{m/n} = 1/10000$, *and taking the log of both sides, we see that we need for* m *to be at least* $19.3 * n$ *and for* k, *the number of hash functions, to bigger than* 13.4 *or* 14.

(b) (8 points) Now imagine that you are keeping a blacklist of n items and you want to determine quickly if any given item is in this blacklist. However, you need the false positive rate to be no more than $1/n$ (i.e. the probability of a false positive must quickly go to 0 as n gets large). If you use a Bloom filter to store the blacklist, what are the approximate values of m and k you need as a function of n? What is the time and space cost of using a Bloom filter with this error rate? Is there any other data structure we've discussed in class that would have asymptotically the same time and space costs as a Bloom filter, and perhaps smaller error probability for this problem? *Solution: To get a false positive rate of* $1/n$, *you need* $.62^{m/n} = 1/n$, *which occurs when* $m/n = \log_{.62} 1/n$ *or* $m = O(n \log n)$. *The number of hash functions needed* k *is also* $O(logn)$. *Red-black trees or skip lists would achieve these time and space costs and reduce probability of error to* 0*!*

(c) (8 points) Imagine that in a red-black tree, each internal node has exactly 3 children. All rules for red-black trees remain the same, so for example each red node must have all black children; for each node, all paths from that node to all descendant leaves must contain the same number of black nodes; all leaf nodes (NIL) are black; etc. Recall that for standard red-black trees, we proved that: "The subtree rooted at the node x contains at least $2^{bh(x)} - 1$ internal nodes" (Lecture 5). Show that the number of nodes (not just internal) in the subtree rooted at x is now at least $3^{bh(x)}$. Prove this bound via induction - don't forget to include the BC, IH and IS. *Solution: We'll show this by induction on the height of x BC: if the height of x is* 0, *x is a leaf node and the number of nodes under x is indeed* $3^0 = 1$. *IH: For all heights j less than h, if x has height j, then its subtree has at least* $3^{bh(x)}$ *nodes. IS: Consider a node x with height $h > 0$ and let q, r, s be x's children. All of these children have height less than x, so we can apply the IH to them. Moreover, the black height of each child must be* $\geq bh(x) - 1$. *Thus, by the IH, the number of nodes in the subtrees rooted at each child is* $\geq 3^{bh(x)-1}$. *Hence, the total number of nodes in the subtree rooted at x is at least* $3 * 3^{bh(x)-1} = 3^{bh(x)}$

3. **Chips**

In the following problem, you are trying to detect illegal copies of items made at a company. For concreteness, assume that the company makes "chips". Unfortunately, there are reports of illegal copies of the company's chips. It is possible to detect these copies because they are completely identical.[1] For any pair of chips, you can place them in a "tray" that will test the two chips and tell you whether or not they are identical. However, this is the only possible operation you can perform on two chips, you can not for example determine if one chip is "less than" another! Also, it is a time consuming operation to use the tray, so you want to minimize the number of times that you use the tray. Your goal is to determine if more than half the chips in some collection are identical.

Thus, you have the following algorithmic problem. You are given a collection of n chips. If there is no set of $n/2$ that are all identical, you should answer NO. If there is a set of more than $n/2$ that are identical, you should answer YES and return one chip that is a member of this set of identical chips. Sketch an algorithm to solve this problem that minimizes the number of times you use the tray. Argue that your algorithm is correct and determine how many times your algorithm uses the tray.

Hint: Use recursion! You must use the tray $O(n \log n)$ times for full credit (15 points).

Solution: Here is a $O(n \log n)$ solution that is easy - you can actually get down to $O(n)$. Divide the chips into two equal piles. Note that if there are more than $n/2$ identical chips at the start, then at least one of the piles will have more than half of its chips be identical. Solve the problem recursively on each pile. Now if more than half the chips in the left (right) pile are identical, a chip x (resp. y) will be returned. Test x and y against all other chips. If either is identical to more than $n/2$ chips than return YES and return the appropriate chip. If neither is, then return NO.

[1] For example the company may use a watermark or a digital signature so it is possible to detect if two chips are equal.

3. **Chips , continued.**

4. **Dynamic Programming**

In this problem, you have a n wireless sensors located in a network, and you must assign each sensor one of two possible channels. When two adjacent nodes, x and y are assigned the same channel, there is a certain amount of interference, $w(x, y)$, which is given by the weight on the edge between the adjacent nodes. In this problem, the weight on an edge can be either positive or negative. An assignment for the network is an assignment of one of the two channels to each node in the network, and the total cost of an assignment is the sum of the interference costs for all adjacent nodes. Your goal in the problems below is to find an assignment that minimizes total cost.

For each of the variants below, 1) give a recurrence relation for the desired value; 2) describe a dynamic program; and 3) give an analysis of the runtime of your dynamic program.

(a) (6 points) Assume the n sensors are connected in a line, and that for all $1 \leq i < n$, you are given edge weight $w(i-1, i)$. Hint: Let $c(i, 1)$ be the minimum cost for assigning channels to nodes 1 through i when node i is assigned channel 1. Let $c(i, 2)$ be the minimum cost of assigning channels to nodes 1 through i when node i is assigned channel 2.

Solution: The recurrence relation is the following:
$c(1, 1) = c(1, 2) = 0$
$c(i, 1) = min\ (c(i-1, 1) + w(i-1, i), c(i-1, 2))$
$c(i, 2) = min\ (c(i-1, 1), c(i-1, 2) + w(i-1, i))$ *The dynamic program just keeps an array of size n with all these values and fills it in from left to right. The final value returned is the minimum of* $c(n, 1)$ *and* $c(n, 2)$. *Runtime is* $O(n)$.

(b) (7 points) Now assume that the sensors are connected in a binary tree. Hint: for node v in the tree, let $c(v, 1)$ be the min cost for assigning channels to all nodes in the subtree rooted at v when v is assigned channel 1, and define $c(v, 2)$ similarly. Let left(v) (right(v)) be the left (reps. right) child of v if they exist or NIL otherwise; let $w(x, y) = 0$ if either x or y is NIL.

Solution: Base Case is when v is a leaf node or when v is NIL. Then $c(v, 1) = c(v, 2) = 0$. *When v is not a leaf node, we have the following recurrence:*
$c(v, 1) = min\ (c(left(v), 2) + w(v, left(v))), c(left(v), 1)) +$
$min\ (c(right(v), 2) + w(v, right(v))), c(right(v), 1))$
$c(v, 2) = min\ (c(left(v), 1) + w(v, left(v))), c(left(v), 2)) +$
$min\ (c(right(v), 1) + w(v, right(v))), c(right(v), 2))$ *Dynamic program just stores these values for each node in the tree, working from the leaf nodes up. Runtime is* $O(n)$.

(c) (7 points) Finally, assume that the sensors are connected in a binary tree and that at least a 2/3 fraction must be assigned channel 1. Hint: Add another parameter to the function c. For this problem, you only need to write down the recurrence relation for c and give a very brief sketch of the algorithm and its runtime. *Solution: Now we let* $c(v, x, c)$ *be the min cost for assigning channels to all nodes in subtree rooted at* v *if 1)* v *is assigned channel* c*; and 2) exactly* x *nodes in the subtree are assigned channel* c*. We will keep a* 3 *dimensional table that is of size* n *by* $n+1$ *by* 2*. Initially, all entries of the table are set to infinity. Base Case is when* v *is a leaf node:* $c(v, 1, 1) = 0$*,* $c(v, 0, 1) = 0$*. When* v *is not a leaf, we have the following recurrence, for all values* $0 \leq k \leq n$*:*
$c(v, k, 1) = min\ _{i,j:i+j=k-1}((min\ (c(left(v), i, 2) + w(v, left(v))), c(left(v), i, 1))$
$+ min\ (c(right(v), j, 2) + w(v, right(v))), c(right(v), j, 1))$
$c(v, i, 2) = min\ _{i,j:i+j=k} min\ (c(left(v), i, 1) + w(v, left(v))), c(left(v), i, 2))$
$+ min\ (c(right(v), j, 1) + w(v, right(v))), c(right(v), j, 2))$
The dynamic program just stores these values for each node in the tree, working from the leaf nodes up. Runtime is now $O(n^3)$ *since there are now* $O(n^2)$ *entries and each takes at most* $O(n)$ *time to fill in. The minimum cost returned is the minimum over all* $k \leq (2/3)n$ *and* $1 \leq x \leq 2$ *of* $c(v, k, x)$*, where* v *is the root of the tree.*

5. **Randomized Algorithms**

Consider a situation where we have n servers and n clients. The servers all know a message m and the clients want to learn that message. Our goal is to design an algorithm that ensures that all clients learn m, while sending the smallest number of messages possible. We have access to a global random number generator R that generates a number uniformly at random between 1 and $\sqrt{n}$, and which all the servers can read.

Consider the following algorithm:

(a) Each client chooses a subset S of $k\sqrt{n}$ of the n servers, uniformly at random from all such subsets. The client then generates $k\sqrt{n}$ requests by choosing independently for each $s \in S$, a tag t that is an integer distributed uniformly at random between 1 and $\sqrt{n}$. The client sends each such request (s, t) to the server s

(b) A random number r is generated by the global random number generator and all servers read that number

(c) Every server s considers the requests they have received of the form (s, r). If there are less than $k\sqrt{n}$ such requests, then s sends m to each client that it received such a request from. k is a parameter to be determined later.

Unfortunately, some of the clients are *bad* in that they may disregard the first line of the algorithm, possibly sending out more than $k\sqrt{n}$ requests, and these requests may not necessarily be generated randomly. Note that the number of messages sent by each good client and server is always only $O(\sqrt{n})$. In this problem, you will show that even with the bad clients around, and even with this bound on communication costs, the protocol still has a good chance of ensuring all the good clients will learn m. (The algorithm thus has applications to mitigating situations like denial of service attacks by botnets.)

Call a sever *overloaded* if it receives $\geq k\sqrt{n}$ requests with tag r. Assume that each server receives at most n requests total, since if they receive ≥ 2 requests from the same client, they can throw out these requests, since that client is necessarily bad.

(a) (5 points) Derive an upper bound on the probability that a fixed server is overloaded.

Solution: There are at most n requests total that each server receives and these are distributed over $\sqrt{n}$ possible tags. Thus, at most $\sqrt{n}/k$ of the tags have more than $k\sqrt{n}$ requests. Thus the probability of being overloaded is at most $1/k$

(b) (5 points) Give an upperbound on the expected number of servers that are overloaded. Note: The events that two different servers are overloaded are NOT independent. *Solution: Using linearity of expectation, the expected number of informed processors that are overloaded is at most* n/k

(c) (5 points) Now use Markov's inequality to bound the probability that the number of overloaded servers is greater than or equal to $n/6$. If everything is going well, you should be able to show this probability is no more than $6/k$. *Solution: Let* X *be the number of overloaded servers. By Markov's,* $Pr(X \geq \lambda) \leq E(X)/\lambda$*, which means* $Pr(X \geq n/6) \leq (n/k)/(n/6) = 6/k$.

(d) (5 points) Now assuming that at most $n/6$ servers are overloaded, calculate the probability that a given client fails to send a request to any server that is not overloaded. Note: The servers that a single client sends requests to are NOT chosen independently (since a given server can not be chosen more than once). You may find the following bound from the book helpful:

$$(x/y)^y \leq \binom{x}{y} \leq (xe/y)^y$$

Hint: In how many ways can you choose $k\sqrt{n}$ of the n servers? In how many ways can you choose $k\sqrt{n}$ of only the $n/6$ overloaded servers? *Solution: The probability of sending every request to a server that is overloaded is* $\binom{n/6}{k\sqrt{n}}/\binom{n}{k\sqrt{n}}$. *Using the above inequality, this probability is no more than* $(e/6)^{k\sqrt{n}} \leq (1/2)^{\sqrt{n}}$

(e) (5 points) Finally, use union bounds and the above results to bound the probability that *any* good client does not receive the message. For n large, what is a good approximation to the probability of failure? Hint: Since the tags for good clients are chosen independently, you can use Chernoff bounds (see HW 2, problem 1) to bound the distribution of the number of requests from good clients that have tag r.

Solution: First we need to bound the probability that a good client does not send a request with tag r to a sever that is not overloaded. Fix a good client and let X be the number of requests that client sends out with tag r. Note that $E(X) = k$ by linearity of expectation. Further, since the tags on the requests are chosen independently, we can use Chernoff bounds to bound the random variable X around its expectation. In particular, $Pr(X \leq (1-\epsilon)E(X)) \leq e^{-\epsilon E(X)/2} = e^{-k\epsilon/2}$. Now, choose $\epsilon = 1/2$ and $k = 8 \log n$. Then a Union Bound gives that with probability at least $1 - 1/n$, each good client sends at least $k/2$ requests with tag r. Now, we can bound the probability that every request with tag r is sent to an overloaded server using the technique from the problem above. In particular, consider the $\geq k/2$ set of requests that a fixed client sends out with tag r. The probability that each of these is sent to a server that is overloaded is $\binom{n/6}{k/2}/\binom{n}{k/2}$. Using the inequality from the last problem, this probability is no more than $(e/6)^{k/2} \leq (1/2)^{4 \log n}$.

Thus, the probability of failure for a single client is at most n^{-4} and the probability of failure for any client in this way by union bounds is n^{-3}. Now there are three bad events that we need to take the union bound over 1) the event that the number of overloaded servers is >= $n/6$ (happens with probability 6/k; 2) the event that some good client sends out less than k/2 messages with tag r (happens with probability 1/n); and 3) the event that some good client sends all requests with tag r to servers that are overloaded (happens with probability <= n^{-3}) . Hence, the probability of failure for any client in any way, is at most $6/k + n^{-1} + n^{-3} = O(1/k)$ (provided that $k \geq 8 \log n$.