

## Quiz 1

- Do not open this quiz booklet until you are directed to do so. Read all the instructions first.
- The quiz contains **seven** multi-part problems. You have 120 minutes to earn 120 points.
- This quiz booklet contains **12** pages, including this one and an extra sheet of scratch paper, which is included for your convenience.
- This quiz is closed book. You may use one double sided Letter ( $8\frac{1}{2}'' \times 11''$ ) or A4 crib sheet. No calculators or programmable devices are permitted.
- Write your solutions in the space provided. Extra scratch paper may be provided if you need more room, although your answer should fit in the given space.
- Do not waste time and paper re-deriving facts that we have studied. It is sufficient to cite known results.
- 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. Generally, a problem's point value is an indication of how much time to spend on it.
- 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.
- Good luck!

Problem	Points	Grade	Initials
1	2		
2	8		
3	40		
4	15		
5	15		
6	20		
7	20		
Total	120		

Name: \_\_\_\_\_  
Circle your recitation instructor:

Matthew Webber (F 11, F1)      Kevin Matulef (F2)      Huy Nguyen (F3)

**Problem 1.** [2 points] Write your name on every page!

**Problem 2. Recurrences** [8 points] (2 parts)

Solve the following recurrences by giving tight  $\Theta$ -notation bounds. You do not need to justify your answers, but any justification that you provide will help when assigning partial credit. As usual, assume that for  $n \leq 10$ ,  $T(n) = O(1)$ .

(a) [4 points]  $T(n) = 9T(\sqrt[3]{n}) + \Theta(\log(n)).$

**Solution:** Let  $n = 2^m$ . Then the recurrence becomes  $T(2^m) = 9T(2^{m/3}) + \Theta(m)$ . Setting  $S(m) = T(2^m)$  gives us  $S(m) = 9S(m/3) + \Theta(m)$ . Using case 1 of the Master Method gives us  $S(m) = \Theta(m^2)$  or  $T(n) = \Theta(\log^2 n)$

(b) [4 points]  $T(n) = T(2n/7) + T(5n/7) + \Theta(n).$

**Solution:** The Master Theorem doesn't apply here. Draw recursion tree. At each level, do  $\Theta(n)$  work. Number of levels is  $\log_{7/5} n = \Theta(\lg n)$ , so guess  $T(n) = \Theta(n \lg n)$  and use the substitution method to verify guess.

**Problem 3. True or False, and Justify [40 points] (8 parts)**

Circle **T** or **F** for each of the following statements, and briefly explain why. The better your argument, the higher your grade, but be brief. Your justification is worth more points than your true-or-false designation.

- (a) T F** [5 points] Suppose that  $H$  is a finite family of universal hash functions of range (table) size 999.  $|H|$  is divisible by 9.

**Solution:** False. Let  $h$  be the identical function that maps the key universe  $\{1, 2, \dots, 999\}$  into the range (table)  $\{1, 2, \dots, 999\}$  such that  $h(x) = x$ . Then the hashing function  $h$  does not have any collision.  $H = \{h\}$  is a universal hashing family since for any  $x \neq y$ ,  $\Pr[h(x) = h(y)] = 0 < 1/999$ .  $|H| = 1$  is not divisible by 9.

Note: In this question, we also gave credit to those who used the following definition of universal hashing: “ $H$  is a universal hashing family if for any  $x \neq y$  and a hash function  $h$  chosen randomly from  $H$ , the probability that  $h(x) = h(y)$  is **equal to**  $1/m$ ” (although in the correct definition, this probability is **at most**  $1/m$ ). The answer is then *True* since for any pair  $x \neq y$ ,  $|\{h \in H | h(x) = h(y)\}| = |H|/999$  is an integer, therefore,  $|H|$  is divisible by 999.

- (b) T F** [5 points] Let  $H$  be a family of universal hash functions that map the universe  $K$  of keys into the range  $\{0, 1, \dots, n - 1\}$ . For a given  $x \in K$ , and  $h$  is a function chosen randomly from  $H$ ,  $\Pr[h(x) = 0] = \Pr[h(x) = 1] = \dots = \Pr[h(x) = n - 1] = \frac{1}{n}$ .

**Solution:** False.

Let  $\mathcal{H} = \{h_1, h_2, h_3\}$ , where the three hash functions map the universe  $\{A, B, C, D\}$  of keys into the range  $\{0, 1, 2\}$  according to the following table:

$x$	$h_1(x)$	$h_2(x)$	$h_3(x)$
$A$	1	0	2
$B$	0	1	2
$C$	0	0	0
$D$	1	1	0

$H$  is a universal hashing family. However, for  $x = C$  and any  $h \in H$ ,  $\Pr[h(x) = 0] = 1 > \Pr[h(x) = 1] = \Pr[h(x) = 2] = 0$ .

- (c) **T F** [5 points] A rotate operation on balanced tree always increases the depth of at least one node and decreases the depth of at least one node.

**Solution:** TRUE. Every rotate operation demotes the root of a subtree and promotes a new node to that position. Promotion decreases a node's depth. See CLRS 13.2 or Lecture 7 for a description/illustration of rotation.

- (d) **T F** [5 points] In a B-tree of minimum parameter  $t$ , every node contains at least  $t - 1$  elements.

**Solution:** FALSE, Normally nodes in a B-tree of parameter  $t$  must have between  $t - 1$  and  $2t - 1$  elements, but the root node is exempted from this rule to accommodate trees with fewer than  $t - 1$  elements. Consider a B-tree of parameter 3 that contains one element: the root node contains 1 element, but here  $t - 1 = 2$ .

- (e) **T F** [5 points] The number of leaves (leaf nodes) in every B-tree is at least 1/2 the total number of nodes.

**Solution:** TRUE. Every non-leaf in a B-tree must have at least 2 children. The number of nodes in the level above the leaves is at most 1/2 the number of leaves. The number of nodes in the level above that is at most 1/4 the number of leaves, and so on up the tree. This sum cannot exceed the total number of leaves.

- (f) **T F** [5 points] The difference between the depth of the deepest and least deep node in a 2-3-4 tree is  $\Theta(\log(n))$ , where  $n$  is the number of nodes in the tree.

**Solution:** True. The difference in depths is one less than the height of the tree, so it is  $\Theta(\log(n))$ .

- (g) **T F** [5 points] Consider a dynamic table that doubles in size when an insert operation causes the table to overflow, and halves when a delete operation causes the table to be less than 1/4 full. If we assign an amortized cost of 4 per insert (with deletes free), then for every sequence of  $n$  consecutive operations, amortized costs serve as an upper bound on true costs.

**Solution:** False. As an example, consider the cost of inserting 17 elements and then deleting those 17 elements. The cost is

- $1*17$  for the inserts.
- $1*17$  for the deletes.
- $1+2+4+8+16 = 31$  for table expansions.
- $7+3+1 = 11$  for table contractions.
- Total =  $17+17+31+11 = 76$ .

Note: Due to the subtleties of the problem, we decided to disregard this question and give everyone 5 points.

- (h) **T F** [5 points] A sequence of  $n$  operations is performed, so that the  $i^{th}$  operation costs  $\lg(i)$  if  $i$  is an exact power of 2, and 1 otherwise. Then the amortized cost per operation is  $\Theta(1)$ .

**Solution:** True. Let  $c(i)$  be the cost of the  $i^{th}$  operation

$$c(i) = \begin{cases} \lg i & \text{if } i = 2^k, k \text{ integer} \\ 1 & \text{otherwise} \end{cases}$$

For any  $n$ , the total cost of  $n$  operations is

$$\begin{aligned} \sum_{i=1}^n c(i) &= n - \lfloor \lg n \rfloor + \sum_{i=1}^{\lfloor \lg n \rfloor} i \\ &= n + \Theta(\lg^2(n)) = \Theta(n) \end{aligned}$$

Therefore, the amortized cost per operation is  $\Theta(1)$ .

**Problem 4. Polynomial Interpolation** [15 points] Suppose you are given numbers  $r_1, r_2, \dots, r_n$  and want to compute the coefficients of the degree  $n$  polynomial with exactly those roots, i.e.  $\prod_{i=1}^n (x - r_i)$ . Give an  $O(n \log^2 n)$  algorithm.

**Solution:** A simple recursive algorithm suffices. Compute  $\prod_{i=1}^{\lfloor n/2 \rfloor} (x - r_i)$  and  $\prod_{i=\lfloor n/2 \rfloor + 1}^n (x - r_i)$  and then multiply the two degree- $n/2$  polynomials together using fast polynomial multiplication via the FFT. The base case is clearly constant time and the combine step takes  $O(n \log n)$  time. Thus the running time is given by the recurrence  $T(n) = 2T(n/2) + \Theta(n \log n)$ . By case 2 of the generalized Master Theorem (as in the lecture),  $T(n)$  is  $O(n \log^2 n)$ .

**Problem 5. Chemical testing [15 points]**

A chemistry lab is given  $n$  samples, with the goal of determining which of the samples contain traces of a foreign substance. It is assumed that only few (say, at most  $t$ ) samples test positive. The tests are very sensitive, and can detect even the slightest trace of the substance in a sample. However, each test is very expensive. Because of that, the lab decided to test "sample pools" instead. Each pool contains a mixture of some of the samples (each sample can participate in several pools). A test of a pool returns positive if any of the samples contributing to the pool contains a trace of the substance.

Design a testing method that correctly determines the positive samples using only  $O(t \log n)$  tests. The method can be **adaptive**, i.e., the choice of the next test can depend on the outcomes of the previous tests.

**Solution:** There exist several related algorithms that solve this problem. The simplest one proceeds as follows: we divide the samples into  $2t$  groups of size  $\frac{n}{2t}$  each. We pool and test each group. Since at most  $t$  groups are positive, we can label at least  $n/2$  samples as negative. Then we recurse on the remaining  $n/2$  samples. It is easy to see that the number of recursion levels is  $O(\log n)$ . Since  $2t$  tests are performed at each level, the total number of tests is at most  $O(t \log n)$ .

A different algorithm divide the samples into two groups of size  $n/2$ . Both groups are tested, and the algorithm recurses on group(s) that test positive. As before, the recursion tree has depth  $\log n$ , since we divide the group size by 2 at each level. Moreover, the recursion tree contains at most  $t$  leaves. Therefore the total number of tree nodes (and therefore tests) is  $O(t \log n)$ .

**Problem 6. Two-array hashing [20 points]**

Alyssa P. Hacker runs an internet company that sells  $n$  different products. In order to quickly access information about the  $n$  products for sale, each product is hashed to a size- $n$  hash table using a simple uniform hash function, with collisions resolved by chaining. Alyssa is happy with this approach because in expectation, a query takes  $\mathcal{O}(1)$  time. However, the downside of the approach is that it is quite likely some slot of the table will have many items hashing to it.<sup>1</sup>

To solve this problem, Alyssa comes up with an idea for ***two-array hashing***, which is defined as follows. Given  $n$  items, allocate *two* arrays  $A_1$  and  $A_2$ , each of size  $n^{1.5}$ . When inserting a new item, map it to one slot in each of the arrays using two different simple uniform hash functions  $h_1$  and  $h_2$ . Place the item only in the less crowded of the two slots. We say that a ***collision*** occurs if both of the two slots are already nonempty.

- (a) [8 points] Consider the  $k$ th items inserted into the two-array hash table. Let  $C_k$  be an indicator random variable with

$$C_k = \begin{cases} 1 & : \text{ if the } k\text{th insert causes a collision} \\ 0 & : \text{ otherwise.} \end{cases}$$

Show that  $E[C_k] \leq (k-1)^2/n^3$ .

**Solution:** The  $k$ th insert causes a collision if and only if  $h_1(k)$  is a nonempty slot of  $A_1$  and  $h_2(k)$  is a nonempty slot of  $A_2$ . At the time of the  $k$ th insert, both  $A_1$  and  $A_2$  have at most  $k-1$  nonempty slots (possibly fewer). Since we are assuming  $h_1$  and  $h_2$  hash uniformly and independently, the probability that they both hash to nonempty slots is at most  $(k-1)/n^{1.5} \cdot (k-1)/n^{1.5} = (k-1)^2/n^3$ .

Partial credit was given for correctly bounding the probability of the  $k$ th item hashing to a nonempty slot of a single array as  $\leq (k-1)/n^{1.5}$ . Other attempts at a solution received a small number of points.

---

<sup>1</sup>One can show that with high probability some slot of the table has  $\Theta(\log n / \log \log n)$  items hashing to it, but you do not need to know or be able to prove this fact.

- (b)** [12 points] Define the random variable  $C = \sum_{k=1}^n C_k$ . What does the variable  $C$  represent? Show that  $E[C] = O(1)$ , and conclude that the fullest slot in the hash table contains  $O(1)$  items in expectation.

**Solution:** The variable  $C$  represents the number of elements that collide with other elements (note this is slightly different than the number of pairs of elements that collide).

To solve for  $E[C]$ , we just need linearity of expectation and part (a) above:

$$\begin{aligned} E[C] &= E\left[\sum_{k=1}^n C_k\right] \\ &= \sum_{k=1}^n E[C_k] \\ &\leq \sum_{k=1}^n \frac{(k-1)^2}{n^3} \\ &\leq \frac{n(n-1)^2}{n^3} \\ &\leq 1 \end{aligned}$$

The expected size of the fullest slot in the hash table is upper bounded by the expected number of elements that collide with other elements. Since the latter is bounded by a constant, so is the former. Thus Alyssa's scheme works.

To receive full credit on this problem it should have been made clear that linearity of expectation was being used. Sloppiness here usually received a minor point deduction.

**Problem 7. Broadcast channel [20 points]**

A set of up to  $n$  processors attempt to communicate over a network. The communication process is deemed successful if *any* of the processors manages to broadcast its information (since the successful processor can then lead the remainder of the communication process). However, the only means of communication is through a common broadcast channel. At any given time step (we assume the time is discrete), any subset of the processors can attempt to communicate through the channel by sending a message. The channel operates as follows:

- If *none* of the processors attempts to send a message, then all processors receive a special "none" message.
- If *only one* of the processors attempts to send a message, then all processors receive that message, and the communication process is deemed successful.
- If *two or more* processors attempt to send a message, then all processors receive a special "collision" message.

Suppose that the number of processors is at least  $n/2$ . Design a randomized protocol that, if followed by all processors, will result in successful communication. The expected number of time steps used by the protocol should be  $O(1)$ .

You can assume all processors know the upper bound  $n$  and the lower bound  $n/2$ .

**Solution:** We assume  $n > 1$ , since otherwise the problem is trivial. The algorithm is as follows: at each time step, each processor sends its message with probability  $1/n$ . If exactly one of the processors manages to broadcast its message, the whole process stops. Otherwise, the protocol is repeated in the next time step.

The analysis is as follows: we will prove that the expected number of time steps used by the protocol is constant. Since each processor sends a message with probability  $1/n$ , the number of messages sent in each time step follows the binomial distribution. In particular, if there are  $k$  processors, the probability that exactly one message is sent at a given time step is

$$P = \binom{k}{1} \frac{1}{n} \left(\frac{n-1}{n}\right)^{k-1} \geq 1/2 \cdot (1 - 1/n)^{k-1} \geq 1/2 \cdot (1 - 1/n)^n,$$

Since  $(1 - 1/n)^n \rightarrow 1/e$  as  $n \rightarrow \infty$  and  $(1 - 1/n)^n > 0$  for  $n > 1$ , it follows that  $(1 - 1/n)^n \geq \delta$  for some absolute constant  $\delta > 0$  (in fact, we can take  $\delta = 1/4$ ). This implies that  $P \geq \delta/2$ . The expected number of time steps used by the protocol is at most  $1/P \leq 2/\delta = O(1)$ .

Some students proposed related randomized algorithms, with running times of  $n$  steps or more. We gave partial credit for those. Also, some students observed that the above procedure guarantees that the expected number of processors that broadcast at each step is  $O(1)$ . This is correct, but not sufficient to give an  $O(1)$  bound for the expected number of *steps*. Again, partial credit was given.

## SCRATCH PAPER

## SCRATCH PAPER

## Practice Quiz 1

- Do not open this quiz booklet until you are directed to do so. Read all the instructions first.
- When the quiz begins, write your name on every page of this quiz booklet.
- The quiz contains five multi-part problems. You have 80 minutes to earn 80 points.
- This quiz booklet contains **11** pages, including this one and an extra sheet of scratch paper, which is included for your convenience.
- This quiz is closed book. You may use one handwritten Letter ( $8\frac{1}{2}'' \times 11''$ ) or A4 crib sheet. No calculators or programmable devices are permitted.
- Write your solutions in the space provided. Extra scratch paper may be provided if you need more room, although your answer should fit in the given space.
- Do not waste time and paper re-deriving facts that we have studied. It is sufficient to cite known results.
- 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. Generally, a problem's point value is an indication of how much time to spend on it.
- 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.
- Good luck!

Problem	Points	Grade	Initials
1	??		
2	??		
3	??		
4	??		
5	??		
Total	80		

Name: \_\_\_\_\_

**Problem 1. Recurrences [?? points] (3 parts)**

For each of the following recurrences, give an asymptotically tight ( $\Theta(\cdot)$ ) bound. Justify your answer by naming the particular case of the Master's Method, by iterating the recurrences, or by using the substitution method. As usual, assume that for  $n \leq 10$ ,  $T(n) = O(1)$ .

**Example:** [0 points] BINARY SEARCH

Recurrence:  $T(n) = T(n/2) + c$

Solution by iteration:

$$T(n) = T(n/4) + c + c = \sum_{i=0}^{\log n} c = c \log n = \Theta(\log n)$$

(a) [?? points]  $T(n) = 8T(n/2) + \Theta(n).$

**Solution:**  $T(n) = \Theta(n^3)$  by part 1 of the Master's Method.

(b) [?? points]  $T(n) = 9T(n/9) + \Theta(n\sqrt{n}).$

**Solution:**  $T(n) = \Theta(n\sqrt{n})$  by part 3 of the Master's Method.

- (c) [?? points]  $T(n) = T(\sqrt{n}) + \log n$ . (It is fine to assume that  $n$  is of the form  $2^{2^k}$  in order to avoid floor and ceiling notation.)

**Solution:** Let  $n = 2^k$ . Then the recurrence becomes  $T(2^k) = T(2^{k/2}) + k$ , or, if we set  $L(k) = T(2^k)$ ,  $L(k) = L(k/2) + k$ . This solves to  $L(k) = \Theta(k)$  by part 3 of the Master's Method. Thus,  $T(n) = \Theta(k) = \Theta(\log n)$ .

Alternative solution is note that  $T(n) \geq \log n$  and, for the upper bound, iterate the recurrence:

$$T(n) = T(\sqrt{n}) + \log n = T(n^{1/4}) + \log \sqrt{n} + \log n \leq \sum_{i=0}^{\infty} \log n^{1/2^i} = \log n \sum_{i=0}^{\infty} \frac{1}{2^i} \leq 2 \log n.$$

Many students made mistakes on this problem, but partial credit (2-3 points) was given where solutions were on the right track, but made a math mistake or used the wrong case of the Master Method (e.g. finding  $L(k) = \Theta(k \log k)$  instead of  $L(k) = \Theta(k)$ ).

**Problem 2. True or False, and Justify** [?? points] (4 parts)

Circle **T** or **F** for each of the following statements, and briefly explain why. The better your argument, the higher your grade, but be brief. Your justification is worth more points than your true-or-false designation.

- (a) **T F** [?? points]  $f(n) = \Theta(g(n))$  is equivalent to  $g(n) = \Theta(f(n))$ .

**Solution:** True.  $f(n) = \Theta(g(n))$  means there exists some  $n_0$  and constants  $c_1, c_2 > 0$  such that for  $n \geq n_0$  we have  $c_1 g(n) \leq f(n) \leq c_2 g(n)$ . But this is equivalent to  $\frac{1}{c_2} f(n) \leq g(n) \leq \frac{1}{c_1} f(n)$ , or  $g(n) = \Theta(f(n))$ .

Many people simple argued that if  $f(n) = \Theta(g(n))$  then  $f(n)$  “grows asymptotically as fast as”  $g(n)$ , which implies that  $g(n)$  also grows as fast as  $f(n)$ . This argument is clearly not enough since we expected that you use the definitions, so no points were given for that justification.

Some people also assumed the fact that  $f(n) = O(g(n))$  implies that  $g(n) = \Omega(f(n))$  to conclude (which, although it is true, needed a line of justification and so one or two points were taken off in those cases).

- (b) **T F** [?? points] There is a deterministic algorithm to sort  $n$  numbers in the comparison model running in time  $\log F_n$ , where  $F_n$  is the  $n$ th Fibonacci number. (Recall that the Fibonacci numbers are defined as follows:  $F_0 = 0, F_1 = 1$  and for  $i > 1$ ,  $F_i = F_{i-1} + F_{i-2}$ .)

**Solution:** False. Remember that  $F_n \leq c^n$ , where  $c$  is some constant. Then,  $\log F_n = n \log c = O(n)$ . In the comparison model, we know that sorting takes  $\Omega(n \log n)$  time.

Some people also justified this part by saying that it is impossible to sort 2 numbers using  $0 = \log 1 = \log F_2$  comparisons in the comparison model. Although the argument is correct we were expecting an asymptotic argument using the lower bound for sorting in the comparison model. Also, a common mistake was to confuse the magnitude of  $F_n$  with the time needed to compute the  $n$ -th Fibonacci number, which is exponentially smaller.

**Problem 3. Hashing**

Design a data structure called `DISTINCT` which maintains a *multi-set* of integers in the range  $\{1, 2, \dots, n^3\}$  under insertions and deletions. A multi-set is a set of elements where duplicate keys are allowed (e.g.,  $\{1, 2, 1, 3\}$  is a multi-set). At any time, the data structure should provide the current number of *distinct* elements in the multi-set. For example, the multi-set  $\{1, 2, 1, 3\}$  contains 3 distinct elements.

Design a randomized data structure that supports the above operations in  $O(1)$  expected time. You can assume that at any time the total number of distinct elements in the set is at most  $n$ . Moreover, you can assume that you are given an  $O(n)$ -size empty block of memory at the beginning.

**Solution:** We maintain a hash table of size  $n$  to stored integers in the multiset along with their duplication counts. For each insertion, we lookup the inserted integer in the hash table. If the number is already there, we increase its duplication count by 1. If it is not, we add that it to the table with duplication count 1. For each deletion, we look up the deleted number in the table, decrease its duplication count by 1 and remove it from the table if the duplication count is 0. We also need to maintain a distinct count for answer queries. This count is increased by 1 whenever new number is added into the table and decreased by 1 whenever a number is removed from the table.

By choosing a hash function  $h$  randomly from a universal hashing family, the expected time for each table lookup is  $O(1)$ , and therefore,  $O(1)$  for each operation.

**Problem 4. Plural Elements** [?? points] (2 parts)

You are given an array of  $n$  integers. For each of the questions below, remember to give a brief, clear explanation of why the algorithm works.

- (a) [?? points] Suppose there exists an integer that appears more than  $n/2$  times in the array. Give a linear time deterministic algorithm to find such an integer.

**Solution:** Find the median of the array – it will be the majority element. Since, if the median wouldn't be the majority element, then the majority element would be smaller or larger than the median, which is impossible (e.g., there are only  $n/2$  elements smaller than the median).

Many people thought an algorithm using the COUNTING idea, which was used in COUNTING SORT. But, it need an assumption for the range of elements, otherwise it is impossible to implement and it may not run in  $O(n)$  time. Also, some people designed an algorithm essentially using the Sorting algorithm, which runs in  $\Theta(n \log n)$  time. Both answers got approximately half of full points.

- (b) [?? points] Now suppose there exists an integer that appears more than  $n/k$  times, where  $k > 2$  is a constant (suppose  $n$  is divisible by  $k$ ). Give a linear time deterministic algorithm to find all such integers.

**Solution:** Let's call a common element an integer that appears more than  $n/k$  times. Let  $a_1, a_2, \dots, a_k$  be the elements with ranks respectively  $n/k, 2n/k, \dots, (k-1)n/k, n$ . Then we check each of the elements  $a_1, a_2, \dots, a_k$  whether it is a common element (just a linear scan per element).

The correctness of the algorithm follows from the claim that any common element must be among  $a_1, a_2, \dots, a_k$ . To see that, consider the sorted array. Any common element is a contiguous block of size  $> n/k$ . Since our “probes”  $a_1, a_2, \dots, a_k$  are at distance  $n/k$ , they must strand the block of a common element.

**Problem 5. Min and Max Revisited** [?? points] (3 parts) In this problem we will investigate a divide and conquer approach for *simultaneously* finding the minimum and maximum of an array of  $n$  integers. in the *comparison* model (i.e., your algorithm is only allowed to compare elements, but not add/subtract/index with them).

Assume for simplicity that  $n$  is a power of 2.

For each of the questions below, remember to give a brief and clear justification.

- (a) [?? points] Suppose you compute the maximum separately and the minimum separately, and output both. How many comparisons does this require?

**Solution:** Computing the maximum takes  $n - 1$  comparisons and the minimum takes another  $n - 1$  comparisons. Total number of comparisons is  $2n - 2$ .

(Full points were given for just saying  $\Theta(n)$ . A few students thought we were asking for lower bounds and gave lower bounds of  $\Omega(n)$  or  $\Omega(\log n)$ . Both answer got full points as well.)

- (b) [?? points] For the rest of this problem, we would like to reduce the leading constant factor in the number of comparisons. We want a result that is not necessarily better asymptotically, but in terms of the exact number of comparisons.

Let's consider an idea for designing a divide and conquer solution for simultaneously computing the minimum and maximum element. The plan is to break the problem into two sublists, compute the min and max of each sublist and then combine the results. An outline of such an algorithm is given below. In the following page, you will be given the opportunity to fill in some details:

$\text{MINMAX}(A, n);$

```
If  $n = 2$  then  $\langle \dots \rangle$  fill in base case  
else Let  $A1 =$  left half and  $A2 =$  right half  
     $(a, b) = \text{MINMAX}(A1, n/2)$   
     $(c, d) = \text{MINMAX}(A2, n/2)$   
 $\langle \dots \rangle$  fill in combine stage
```

- (i) Fill in the details for the case  $n = 2$ .

**Solution:** If  $A[1] < A[2]$  return( $A[1], A[2]$ ), else return( $A[2], A[1]$ );  
A few solutions said “Return  $\min(A[1], A[2]), \max(A[1], A[2])$ ”. Since the goal is to minimize actual number of comparisons this is wasteful. Usually (depending on clarifications in Part (c)), two to four points were taken off for this error.

- (ii) Fill in the details for the combine step here.

**Solution:** If  $a < c$  let  $a' = a$ , else  $a' = c$ . If  $b < d$  then  $b' = d$  else  $b' = b$ .  
Return( $a', b'$ )

(c) [?? points]

Let  $T(n) = c_1n - c_2$  be the number of comparisons used by the algorithm. Find an exact expression for  $T(n)$  (i.e., the best values of the constants  $c_1$  and  $c_2$ ). (You may do this by writing out and solving the recurrence for  $T(n)$  using the substitution method. ) Is this really better than the naive algorithm?

**Solution:** The recurrence for the number of comparisons is  $T(n) = 2T(n/2) + 2$  with the base case  $T(2) = 1$ .

We guess that the solution is of the form  $T(n) = c_1n - c_2$ . Then we have  $c_1n - c_2 = 2(c_1n/2 - c_2) + 2$ , giving  $c_2 = 2$ . Now plugging in  $T(2) = 2c_1 - 2 = 1$ , we get  $c_1 = 3/2$ .

Thus the number of comparisons is  $(3/2)n - 2$ , which has a better leading constant than the naive solution.

Alternate solutions expanded the recurrence as  $2+4+\dots+n/2+n/2T(2) = 3/2n-2$ , using  $T(2) = 1$ . Such a solution also received full credit.

Common mistakes in this part were to ignore the base case, or to say  $T(n) = 2T(n/2) + \Theta(1)$ .

**Problem 6. Matrix Multiplication [?? points] (2 parts)**

In this problem we are given three matrices and the question is to compute their product. For each of the two cases below, report the fastest algorithm you know to compute the product of the given three matrices. You need to prove correctness of the algorithm and argue its running time.

- (a) [?? points] Given matrices  $A, B, C$ , each of dimension  $n \times n$ , give a fast algorithm to compute  $A \cdot B \cdot C$ ? What is its asymptotic running time? (You may use algorithms mentioned in lectures without describing them.)

**Solution:** The solution depends on what's the fastest way to compute the product of just two matrices. Suppose it takes  $T$  time to compute the product of two  $n \times n$  matrices. Then, computing  $A \cdot B \cdot C$  takes  $2T$  time – just compute  $A \cdot B$  first, and then multiply the result, also an  $n \times n$  matrix, by  $C$ .

What is the best known  $T$ ? Strassen's algorithm gives a  $T = O(n^{\log_2 7})$  time algorithm for computing the product of two matrices. The currently best known algorithm has  $T = O(n^{2.376})$ , and is due to Coppersmith and Winograd. We gave full points for the answer  $O(n^{\log_2 7})$ , and four points for knowing there is a faster algorithm than the straightforward  $O(n^3)$  algorithm, but not remembering the runtime.

Several solutions forgot to mention the runtime, two to three points were taken off for this type of error.

- (b) [5 points] Now, suppose  $C$  is of dimension  $n \times 1$  (i.e.,  $C$  is a vertical vector), and  $A$  and  $B$  are still of dimension  $n \times n$ . Give a fast algorithm for computing  $A \cdot B \cdot C$ ? What is its asymptotic running time?

**Solution:** Here the trick is to perform the computations in the right order, using the fact that multiplying an  $n \times n$  matrix by an  $n \times 1$  vector can be done in  $O(n^2)$  time with the straight-forward algorithm. First compute  $B \cdot C$ , which takes  $O(n^2)$  time. Then compute  $A \cdot (B \cdot C)$ , which is also a multiplication of a  $n \times n$  matrix by a  $n \times 1$  vector, and takes  $O(n^2)$  times. So, the total running time is  $O(n^2)$ .

Some people thought that matrix multiplication is commutative, please check to convince yourself that it is not. Three to four points were given for the right algorithm, but wrong runtime analysis.

**Problem 7.** Joining and Splitting 2-3-4 Trees

The JOIN operator takes as input two 2-3-4 trees,  $T_1$  and  $T_2$ , and an element  $x$  such that for any  $y_1 \in T_1$  and  $y_2 \in T_2$ , we have  $\text{key}[y_1] < \text{key}[x] < \text{key}[y_2]$ . As output JOIN returns a 2-3-4 tree  $T$  containing the node  $x$  and all the elements of  $T_1$  and  $T_2$ .

The SPLIT operator is like an “inverse” JOIN: given a 2-3-4 tree  $T$  and an element  $x \in T$ , SPLIT creates a tree  $T_1$  consisting of all elements in  $T - \{x\}$  whose keys are less than  $\text{key}[x]$ , and a tree  $T_2$  consisting of all elements in  $T - \{x\}$  whose keys are greater than  $\text{key}[x]$ .

In this problem, we will efficiently implement JOIN and SPLIT. For convenience, you may assume that all elements have unique keys.

- (a) Suppose that in every node  $x$  of the 2-3-4 tree there is a new field  $\text{height}[x]$  that stores the height of the subtree rooted at  $x$ . Show how to modify INSERT and DELETE to maintain the  $\text{height}$  of each node while still running in  $O(\log n)$  time. Remember that all leaves in a 2-3-4 tree have the same depth.

**Solution:** Let leaf nodes have a  $\text{height}$  of 1 and internal nodes have  $\text{height}[x] = 1 + \text{height}[\text{child}(x)]$ . A node affected by INSERT or DELETE operations will simply recalculate their  $\text{height}$  value by looking at the  $\text{height}$  of their children. In both INSERT and DELETE, at most  $O(\log n)$  nodes positions will be affected and each of their  $\text{height}$  values can be updated in  $O(1)$  time. Therefore, the added calculation cost of maintaining  $\text{height}$  fields is  $O(\log n)$ .

A slight caveat in using this method is that we must ensure that heights are calculated from the bottom-up, otherwise there could be a case where a parent computes its height from an out of date child. Fortunately, both INSERT and DELETE recursively work from the bottom of the tree upward, so this is not an issue.

- (b) Using part (a), give an  $O(1 + |h_1 - h_2|)$ -time JOIN algorithm, where  $h_1$  and  $h_2$  are the heights of the two input 2-3-4 trees.

**Solution:** Find the heights of  $T_1$  and  $T_2$ . If  $h_1 > h_2$ , find the node  $z$  with depth  $h_1 - h_2$  on the rightmost path of  $T_1$  and insert  $x$  into  $z$ . If  $z$  is full, it will be split with a key floating up as in INSERT. Set the rightmost child of the node containing  $x$  to be the root of  $T_2$ . Now every leaf in the resulting 2-3-4 Tree has depth  $h_1$ , and the branching constraint is obeyed. The case for  $h_1 < h_2$  is similar.

If  $h_1 = h_2$ , merge the two root nodes along with  $x$  into a “fat” node, and split the node if it is overloaded.

It takes  $O(1 + |h_1 - h_2|)$  time to find the node  $z$  and insert  $x$  into  $z$ , and  $O(1)$  time to join the smaller tree to the larger tree. Therefore the total running time is  $O(1 + |h_1 - h_2|)$ .

- (c) Give an  $O(\log n)$ -time SPLIT algorithm. Your algorithm will take a 2-3-4 tree  $T$  and key  $k$  as input. To write your SPLIT algorithm, you should take advantage of the search path from  $T$ 's root to the node that would contain  $k$ . This path will consist of a set of keys  $\{k_1, \dots, k_m\}$ . Consider the left and right subtrees of each key  $k_i$  and their relationship to  $k$ . You may use your JOIN procedure from part (b) in your solution.

**Solution:**

1. Initialize two empty trees  $T_1$  and  $T_2$ .
2. Search for the element  $k$  in the tree  $T$ .
3. If the search path at node  $k_i$  traverses right, INSERT  $k_i$  into  $T_1$  and JOIN  $k_i$ 's left subtrees with  $T_1$ .
4. If the search path at node  $k_i$  traverses left, INSERT  $k_i$  into  $T_2$  and JOIN  $k_i$ 's right subtrees with  $T_2$ .
5. If  $k$  is found JOIN  $k$ 's left child with  $T_1$  and its right child with  $T_2$ .
6. If a leaf node is encountered, insert any remaining elements into their appropriate tree.

Let  $k_l$  be some key less than  $k$ . Then  $k_l$  will either: (1) be a node which the search path turned right on, (2) be less than some node that the search path turned right on, or (3) be a left child of  $k$ . In all three cases,  $k_l$  will be INSERTed or JOINed into  $T_1$ . Similarly, any nodes greater than  $k$  will be placed in  $T_2$ .

The algorithm joins the subtrees as it walks down the 2-3-4 tree along the search path. Therefore the height of subtrees never increases. In other words, we have  $height[T_{i-1}] \geq height[T_i]$ . Searching for  $k$  takes  $O(\log n)$  time. Let  $h_i$  denote the height of subtree  $T_i$ . The running time for the iterative JOIN takes:

$$\begin{aligned} O\left(\sum_{i=1}^m (1 + |h_{i-1} - h_i|)\right) &= O\left(\sum_{i=1}^m (1 + h_{i-1} - h_i)\right) \\ &= O(m + h_0 - h_m) \\ &= O(m + \log n). \end{aligned}$$

Since a 2-3-4 tree has at most 4 branches, the algorithm can join at most 3 subtrees before the search path goes down 1 level in the 2-3-4 tree. Therefore, the number of subtrees  $m$  joined is at most 3 times the depth of the key  $k$ . Therefore  $m = O(\log n)$  and the time complexity of the SPLIT operation is  $O(\log n)$ .

## SCRATCH PAPER

## 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		
2	33		
3	15		
4	10		

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

Total	160		
-------	-----	--	--

Name: \_\_\_\_\_  
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  $T(n)/n$ .

**Solution:** True. This is the aggregate method.

- (b) **T F** [4 points] Fix some integers  $m \gg n > 0$ . 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  $h(i) \neq 0$ .

**Solution:** If  $\mathcal{H}$  is the set of all functions (which is a universal hashing family), then  $h$  can be a zero function.

- (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.

**Solution:** False. Any problem in P is also in NP and can be solved in polynomial time. However, it is not known that all problems in NP can be solved in polynomial time.

- (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  $y \notin L$ .)

**Solution:** True. If  $P = NP$ , then every NP-complete problem reduces in polynomial time from any other problem in NP, by solving the problem and then outputting a canonical yes or no instance.

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

**Solution:** True. We can simply use BFS (breadth-first search) to achieve this. Note that the question asks about finding a spanning tree, not a minimum spanning tree.

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

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

This algorithm runs in polynomial time.

**Solution:** False. To run in polynomial-time, this algorithm would have to run in time polynomial in  $\lg x$  (input size). This algorithm uses  $\Theta(x)$  multiplications in the worst case.

- (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.

**Solution:** True. Consider the minimum cut. It is made up of edges with capacities that are multiples of 3, so the capacity of the cut (sum of capacities of edges in the cut) must be a multiple of 3. By the Maxflow-Mincut theorem, the maximum flow has the same value.

- (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  $w(e) \in \{0, 1, \dots, V^3\}$ , 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.

**Solution:** True. We can achieve this by modifying the Dijkstra algorithm to use Van Emde Boas data structure in place of the priority queue.

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

**Solution:** False. It can be done in  $\Theta(\log n)$  queries.

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

**Solution:** False; in order to decide whether a undirected graph is connected, all nodes in the graph must be checked.

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

**Solution:** False. An adversary cannot influence the outcome of coin tosses.

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

**Solution:** False. There is a polynomial-time approximation algorithm for metric TSP, but no approximation algorithm for the general case (consider the hard special case where there is a 0-weight cycle) assuming  $P \neq NP$ .

**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.

<input type="checkbox"/>	Map folding	A. Polynomial reduction
<input type="checkbox"/>	Integer multiplication	B. Ford-Fulkerson algorithm
<input type="checkbox"/>	Finding a minimum spanning tree	C. Dynamic programming
<input type="checkbox"/>	All-pairs shortest paths	D. General matching
<input type="checkbox"/>	Polynomial identity testing	E. Divide and conquer
<input type="checkbox"/>	SubsetSum is NP-hard	F. Johnson algorithm
<input type="checkbox"/>	Chinese postman tour	G. Dijkstra
<input type="checkbox"/>	Finding a minimum cut	H. Greedy algorithm
<input type="checkbox"/>	Single-source shortest paths	I. Monte Carlo algorithm

**Solution:**

- |                                    |                             |
|------------------------------------|-----------------------------|
| C. Map folding                     | A. Polynomial reduction     |
| E. Integer Multiplication          | B. Ford-Fulkerson algorithm |
| H. Finding a minimum spanning tree | C. Dynamic programming      |
| F. All pair shortest path          | D. General Matching         |
| I. Polynomial identity testing     | E. Divide and Conquer       |
| A. SubsetSum is NP-hard            | F. Johnson Algorithm        |
| D. Chinese postman tour            | G. Dijkstra                 |
| B. Finding Mincut                  | H. Greedy Algorithm         |
| G. Single source shortest path     | I. Monte Carlo algorithm    |

- (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.

**Solution:** We use universal hashing to solve this problem. Create a hash table of  $2n$  elements, and for each element  $x$  in  $A$ , search for  $x$  in the table and insert it only if the search fails. Then walk down the slots of the table and output every element. The searches and insertions each take  $O(1)$  expected time, and walking down the table takes  $O(n)$  time, for a total expected runtime of  $O(n)$ .

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

**Solution:** Prove by contradiction. Assume that there exists an  $(1 + \frac{1}{2n})$ -approximation algorithm  $\mathcal{A}$  for Vertex Cover. For a given graph  $G = (V, E)$ , let  $S$  be a minimum vertex cover for  $G$ . Then  $\mathcal{A}$  can decide if  $G$  has a vertex cover of size at least

$$|S|/(1 + \frac{1}{2n}) \geq |S| \times (1 - \frac{1}{2n}) > |S| - 1$$

. Therefore,  $\mathcal{A}$  can solve Vertex Cover in polynomial time. Contradiction.

- (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

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	
B	
C	
D	
E	
F	

**Solution:**

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

**Solution:** LP.

**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  $k > 0$ ,  $H_k = \begin{bmatrix} H_{k-1} & H_{k-1} \\ H_{k-1} & -H_{k-1} \end{bmatrix}$ .

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.

**Solution:**

HADAMARD-MULT( $k, v$ )

```

1  if  $k = 0$ 
2    then
3      return  $v$ 
4  else
5    Let  $v_1, v_2$  be the first and second half of  $v$ . (each  $v_i$  has length  $n_i = 2^{k-1}$ ).
6     $a \leftarrow$  HADAMARD-MULT( $k - 1, v_1$ )
7     $b \leftarrow$  HADAMARD-MULT( $k - 1, v_2$ )
8    return  $((a + b)^T, (a - b)^T)^T$ 
```

Let  $T(n)$  be the running time.

$$T(n) = 2T(n/2) + O(n)$$

By the Master Theorem case 2,

$$T(n) = O(n \log n)$$

**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( $G, s, t$ )

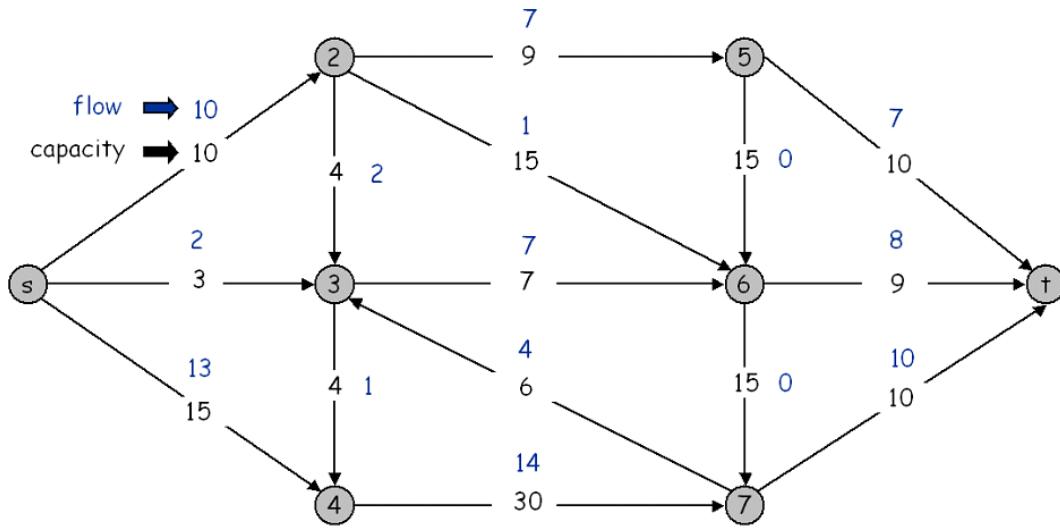
```

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- $s$ - $t$ -CUT of  $s$  and  $t$ .
```

**Solution:** Each time we add a new vertex to the cut, the number of edges removed is smaller than the number of edges added. Therefore, the size of the cut is at least half the total number of edges.

**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.

**Solution:**

$$s \rightarrow 3 \rightarrow 2 \rightarrow 5 \rightarrow t$$

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

**Solution:** 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.

**Solution:** No.

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

**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.

**Solution:** Easy to check that both problems are in NP.

$P_2$  is NP-hard. Let us construct a graph  $G' = (V', E')$  from  $G$  as follow.

$$V' = \{u_1, u_2, u_3\} \cup V$$

$$E' = \{(u_1, u_2), (u_2, u_3), (u_3, u_1)\} \cup \{(u_1, v) : v \in V\} \cup E$$

$G'$  always has a cycle of length 3 -  $(u_1, u_2, u_3)$ . Also, if there exists any Hamiltonian path  $P$  in  $G$ , then  $(u_3, u_2, u_1, P)$  is also a Hamiltonian path in  $G'$ . Conversely, if  $P'$  is a Hamiltonian path in  $G'$ . Then  $P'$  must be has one the following patterns

$$(u_2, u_3, u_1, P), (u_3, u_2, u_1, P), (P, u_1, u_2, u_3), (P, u_1, u_3, u_2)$$

In any case,  $P$  is a Hamiltonian path in  $G$ . Therefore, solving HAM-PATHfor  $G'$  is equivalent to solving HAM-PATHfor  $G$ .

$P_1$  is decidable. Checking whether a graph has cycle can be done in polynomial time using DFS. If the graph has a cycle, accept and return. In case the graph does not have a cycle, checking if it has a Hamiltonian path is easy since the graph is acyclic.

**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.

**Solution:** Let  $A(G)$  be output of the black box on input  $G$ . The algorithm works as follow.

```
1   $S \leftarrow \emptyset$ 
2   $k = A(G)$ 
3  for each vertex  $v \in V$ 
4    do
5       $G' \leftarrow G \setminus \{v\}$ 
6      if  $A(G') < k$ 
7        then
8           $S \leftarrow S \cup \{v\}$ 
9           $k \leftarrow k - 1$ 
10     else
11        $G \leftarrow G'$ 
12   return the subgraph of  $G$  induced by  $S$ .
```

**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:  $c_1, c_2, \dots, c_n$ . 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  $c_1, c_2, \dots, c_n$  is a sequence of  $n$  poses  $p_1, p_2, \dots, p_n$  such that pose  $p_i$  places exactly one finger on each note in  $c_i$ , for all  $1 \leq i \leq n$ ,

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  $p_1, p_2, \dots, p_n$  is the sum  $\sum_{i=2}^n D[p_{i-1}, p_i]$ . Give an  $O(n)$ -time algorithm that, given a song  $c_1, c_2, \dots, c_n$ , finds a fingering  $p_1, p_2, \dots, p_n$  of the song with minimum possible difficulty.

**Solution:**

**Dynamic programming.** Subproblems  $\text{OPT}(i, p_i) =$  the minimum possible difficulty for the song  $c_1, c_2, \dots, c_i$  and the  $i^{th}$  pose is  $p_i$ . There are  $O(n) \cdot 6^4 = O(n)$  such subproblems.

Base case:  $\text{OPT}(1, p_1) = 0$ . For  $i > 1$ , in order to compute  $\text{OPT}(i, p_i)$ , we guess pose  $p_{i-1}$  (there are  $6^4 = O(1)$  choices):

$$\text{OPT}(i, p_i) = \min\{\text{OPT}(i - 1, p_{i-1}) + D[p_{i-1}, p_i] \mid p_{i-1} \text{ is a valid pose for } c_{i-1}\}$$

Running time is  $O(n)$  since there are  $O(n)$  subproblems and it takes  $O(1)$  to compute the solution for each of them.

**Other solution :** You could define an  $n$ -stage graph with  $6^4$  nodes in each stage, and find a shortest path. Since this graph is acyclic, we can find the shortest path in linear time (using topological sort).

SCRATCH PAPER

SCRATCH PAPER

SCRATCH PAPER

SCRATCH PAPER

SCRATCH PAPER

## 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		
2	33		
3	15		
4	10		

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

Total	160		
-------	-----	--	--

Name: \_\_\_\_\_  
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  $T(n)/n$ .

**Solution:** True. This is the aggregate method.

- (b) **T F** [4 points] Fix some integers  $m \gg n > 0$ . 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  $h(i) \neq 0$ .

**Solution:** If  $\mathcal{H}$  is the set of all functions (which is a universal hashing family), then  $h$  can be a zero function.

- (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.

**Solution:** False. Any problem in P is also in NP and can be solved in polynomial time. However, it is not known that all problems in NP can be solved in polynomial time.

- (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  $y \notin L$ .)

**Solution:** True. If  $P = NP$ , then every NP-complete problem reduces in polynomial time from any other problem in NP, by solving the problem and then outputting a canonical yes or no instance.

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

**Solution:** True. We can simply use BFS (breadth-first search) to achieve this. Note that the question asks about finding a spanning tree, not a minimum spanning tree.

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

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

This algorithm runs in polynomial time.

**Solution:** False. To run in polynomial-time, this algorithm would have to run in time polynomial in  $\lg x$  (input size). This algorithm uses  $\Theta(x)$  multiplications in the worst case.

- (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.

**Solution:** True. Consider the minimum cut. It is made up of edges with capacities that are multiples of 3, so the capacity of the cut (sum of capacities of edges in the cut) must be a multiple of 3. By the Maxflow-Mincut theorem, the maximum flow has the same value.

- (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  $w(e) \in \{0, 1, \dots, V^3\}$ , 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.

**Solution:** True. We can achieve this by modifying the Dijkstra algorithm to use Van Emde Boas data structure in place of the priority queue.

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

**Solution:** False. It can be done in  $\Theta(\log n)$  queries.

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

**Solution:** False; in order to decide whether a undirected graph is connected, all nodes in the graph must be checked.

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

**Solution:** False. An adversary cannot influence the outcome of coin tosses.

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

**Solution:** False. There is a polynomial-time approximation algorithm for metric TSP, but no approximation algorithm for the general case (consider the hard special case where there is a 0-weight cycle) assuming  $P \neq NP$ .

**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.

<input type="checkbox"/>	Map folding	A. Polynomial reduction
<input type="checkbox"/>	Integer multiplication	B. Ford-Fulkerson algorithm
<input type="checkbox"/>	Finding a minimum spanning tree	C. Dynamic programming
<input type="checkbox"/>	All-pairs shortest paths	D. General matching
<input type="checkbox"/>	Polynomial identity testing	E. Divide and conquer
<input type="checkbox"/>	SubsetSum is NP-hard	F. Johnson algorithm
<input type="checkbox"/>	Chinese postman tour	G. Dijkstra
<input type="checkbox"/>	Finding a minimum cut	H. Greedy algorithm
<input type="checkbox"/>	Single-source shortest paths	I. Monte Carlo algorithm

**Solution:**

- |                                    |                             |
|------------------------------------|-----------------------------|
| C. Map folding                     | A. Polynomial reduction     |
| E. Integer Multiplication          | B. Ford-Fulkerson algorithm |
| H. Finding a minimum spanning tree | C. Dynamic programming      |
| F. All pair shortest path          | D. General Matching         |
| I. Polynomial identity testing     | E. Divide and Conquer       |
| A. SubsetSum is NP-hard            | F. Johnson Algorithm        |
| D. Chinese postman tour            | G. Dijkstra                 |
| B. Finding Mincut                  | H. Greedy Algorithm         |
| G. Single source shortest path     | I. Monte Carlo algorithm    |

- (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.

**Solution:** We use universal hashing to solve this problem. Create a hash table of  $2n$  elements, and for each element  $x$  in  $A$ , search for  $x$  in the table and insert it only if the search fails. Then walk down the slots of the table and output every element. The searches and insertions each take  $O(1)$  expected time, and walking down the table takes  $O(n)$  time, for a total expected runtime of  $O(n)$ .

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

**Solution:** Prove by contradiction. Assume that there exists an  $(1 + \frac{1}{2n})$ -approximation algorithm  $\mathcal{A}$  for Vertex Cover. For a given graph  $G = (V, E)$ , let  $S$  be a minimum vertex cover for  $G$ . Then  $\mathcal{A}$  can decide if  $G$  has a vertex cover of size at least

$$|S|/(1 + \frac{1}{2n}) \geq |S| \times (1 - \frac{1}{2n}) > |S| - 1$$

. Therefore,  $\mathcal{A}$  can solve Vertex Cover in polynomial time. Contradiction.

- (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

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	
B	
C	
D	
E	
F	

**Solution:**

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

**Solution:** LP.

**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  $k > 0$ ,  $H_k = \begin{bmatrix} H_{k-1} & H_{k-1} \\ H_{k-1} & -H_{k-1} \end{bmatrix}$ .

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.

**Solution:**

HADAMARD-MULT( $k, v$ )

```

1  if  $k = 0$ 
2    then
3      return  $v$ 
4  else
5    Let  $v_1, v_2$  be the first and second half of  $v$ . (each  $v_i$  has length  $n_i = 2^{k-1}$ ).
6     $a \leftarrow$  HADAMARD-MULT( $k - 1, v_1$ )
7     $b \leftarrow$  HADAMARD-MULT( $k - 1, v_2$ )
8    return  $((a + b)^T, (a - b)^T)^T$ 
```

Let  $T(n)$  be the running time.

$$T(n) = 2T(n/2) + O(n)$$

By the Master Theorem case 2,

$$T(n) = O(n \log n)$$

**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( $G, s, t$ )

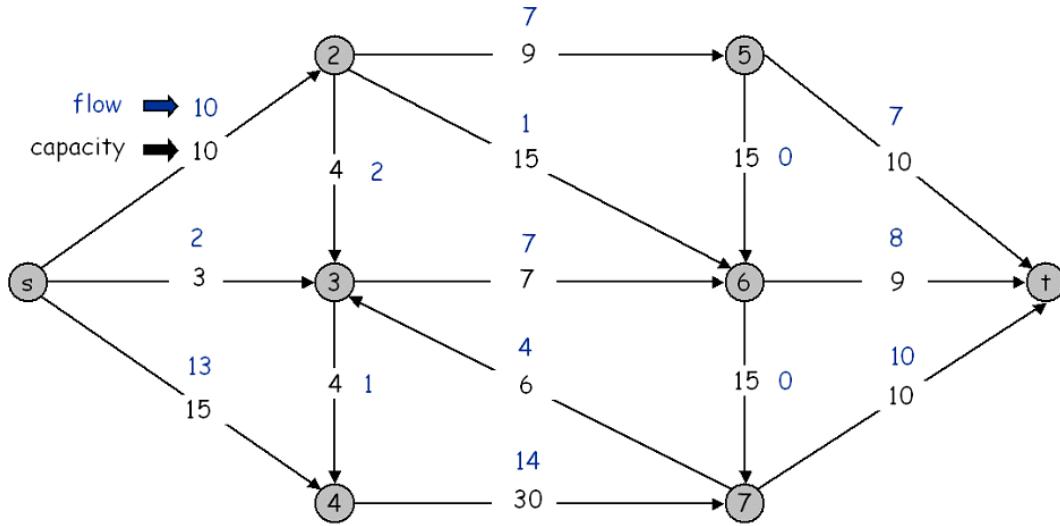
```

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- $s$ - $t$ -CUT of  $s$  and  $t$ .
```

**Solution:** Each time we add a new vertex to the cut, the number of edges removed is smaller than the number of edges added. Therefore, the size of the cut is at least half the total number of edges.

**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.

**Solution:**

$$s \rightarrow 3 \rightarrow 2 \rightarrow 5 \rightarrow t$$

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

**Solution:** 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.

**Solution:** No.

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

**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.

**Solution:** Easy to check that both problems are in NP.

$P_2$  is NP-hard. Let us construct a graph  $G' = (V', E')$  from  $G$  as follow.

$$V' = \{u_1, u_2, u_3\} \cup V$$

$$E' = \{(u_1, u_2), (u_2, u_3), (u_3, u_1)\} \cup \{(u_1, v) : v \in V\} \cup E$$

$G'$  always has a cycle of length 3 -  $(u_1, u_2, u_3)$ . Also, if there exists any Hamiltonian path  $P$  in  $G$ , then  $(u_3, u_2, u_1, P)$  is also a Hamiltonian path in  $G'$ . Conversely, if  $P'$  is a Hamiltonian path in  $G'$ . Then  $P'$  must be has one the following patterns

$$(u_2, u_3, u_1, P), (u_3, u_2, u_1, P), (P, u_1, u_2, u_3), (P, u_1, u_3, u_2)$$

In any case,  $P$  is a Hamiltonian path in  $G$ . Therefore, solving HAM-PATHfor  $G'$  is equivalent to solving HAM-PATHfor  $G$ .

$P_1$  is decidable. Checking whether a graph has cycle can be done in polynomial time using DFS. If the graph has a cycle, accept and return. In case the graph does not have a cycle, checking if it has a Hamiltonian path is easy since the graph is acyclic.

**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.

**Solution:** Let  $A(G)$  be output of the black box on input  $G$ . The algorithm works as follow.

```
1   $S \leftarrow \emptyset$ 
2   $k = A(G)$ 
3  for each vertex  $v \in V$ 
4    do
5       $G' \leftarrow G \setminus \{v\}$ 
6      if  $A(G') < k$ 
7        then
8           $S \leftarrow S \cup \{v\}$ 
9           $k \leftarrow k - 1$ 
10     else
11        $G \leftarrow G'$ 
12   return the subgraph of  $G$  induced by  $S$ .
```

**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:  $c_1, c_2, \dots, c_n$ . 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  $c_1, c_2, \dots, c_n$  is a sequence of  $n$  poses  $p_1, p_2, \dots, p_n$  such that pose  $p_i$  places exactly one finger on each note in  $c_i$ , for all  $1 \leq i \leq n$ ,

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  $p_1, p_2, \dots, p_n$  is the sum  $\sum_{i=2}^n D[p_{i-1}, p_i]$ . Give an  $O(n)$ -time algorithm that, given a song  $c_1, c_2, \dots, c_n$ , finds a fingering  $p_1, p_2, \dots, p_n$  of the song with minimum possible difficulty.

**Solution:**

**Dynamic programming.** Subproblems  $\text{OPT}(i, p_i) =$  the minimum possible difficulty for the song  $c_1, c_2, \dots, c_i$  and the  $i^{th}$  pose is  $p_i$ . There are  $O(n) \cdot 6^4 = O(n)$  such subproblems.

Base case:  $\text{OPT}(1, p_1) = 0$ . For  $i > 1$ , in order to compute  $\text{OPT}(i, p_i)$ , we guess pose  $p_{i-1}$  (there are  $6^4 = O(1)$  choices):

$$\text{OPT}(i, p_i) = \min\{\text{OPT}(i - 1, p_{i-1}) + D[p_{i-1}, p_i] \mid p_{i-1} \text{ is a valid pose for } c_{i-1}\}$$

Running time is  $O(n)$  since there are  $O(n)$  subproblems and it takes  $O(1)$  to compute the solution for each of them.

**Other solution :** You could define an  $n$ -stage graph with  $6^4$  nodes in each stage, and find a shortest path. Since this graph is acyclic, we can find the shortest path in linear time (using topological sort).

SCRATCH PAPER

SCRATCH PAPER

SCRATCH PAPER

SCRATCH PAPER

SCRATCH PAPER

## Final Exam

- Do not open this exam booklet until you are directed to do so. Read all the instructions first.
- The quiz contains **6** multi-part problems. You have 180 minutes to earn 120 points.
- This quiz booklet contains **11 double-sided** pages, including this one and a double-sided sheet of scratch paper; there should be 18 (numbered) pages of problems.
- This quiz is closed book. You may use **three** double sided Letter ( $8\frac{1}{2}'' \times 11''$ ) or A4 crib sheet. No calculators or programmable devices are permitted. Cell phones must be put away.
- Write your solutions in the space provided. Extra scratch paper may be provided if you need more room, although your answer should fit in the given space.
- Do not waste time re-deriving facts that we have studied. It is sufficient to cite known results.
- Do not spend too much time on any one problem. Generally, a problem's point value is an indication of how much time to spend on it.
- 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.
- Good luck!

Problem	Points	Grade	Initials
1	42		
2	42		
3	9		
4	9		
5	8		
6	10		
Total	120		

Name: \_\_\_\_\_

Circle your recitation:

R01	R02	R03	R04	R05	R06
F10	F11	F12	F1	F2	F3
Joe	Joe	Khanh	Khanh	Emily	Emily

R07	R08	R09	R10
F11	F12	F1	F2
Matt	Matt	Geoff	Geoff



**Problem 1. True or False, and Justify [42 points] (14 parts)**

Circle **T** or **F** for each of the following statements, and briefly explain why. Your justification is worth more points than your true-or-false designation. If you need to make a reasonable assumption in order to answer a question (for example, if you need to assume that  $P \neq NP$ ), please state that assumption explicitly.

- (a) **T F** [3 points] If problem  $A$  can be reduced to 3SAT via a deterministic polynomial-time reduction, and  $A \in NP$ , then  $A$  is NP-complete.

**Solution:** False. We need to reduce in the other direction (reduce an NP-hard problem to  $A$ ).

- (b) **T F** [3 points] Let  $G = (V, E)$  be a flow network, i.e., a weighted directed graph with a distinguished source vertex  $s$ , a sink vertex  $t$ , and non-negative capacity  $c(u, v)$  for every edge  $(u, v)$  in  $E$ . Suppose you find an  $s$ - $t$  cut  $C$  which has edges  $e_1, e_2, \dots, e_k$  and a capacity  $f$ . Suppose the value of the maximum  $s$ - $t$  flow in  $G$  is  $f$ .

Now let  $H$  be the flow network obtained by adding 1 to the capacity of each edge in  $C$ . Then the value of the maximum  $s$ - $t$  flow in  $H$  is  $f + k$ .

**Solution:** False. There could be multiple min-cuts. Consider the graph s-v-t where the edges have capacity 1; either edge in itself is a min-cut, but adding capacity to that edge alone does not increase the max flow.

- (c) **T F** [3 points] Let  $A$  and  $B$  be optimization problems where it is known that  $A$  reduces to  $B$  in polynomial time. Additionally, it is known that there exists a polynomial-time 2-approximation for  $B$ . Then there must exist a polynomial-time 2-approximation for  $A$ .

**Solution:** False; approximation factor is not (necessarily) carried over in poly-time reduction. See e.g. set cover vs. vertex cover.

- (d) **T F** [3 points] There exists a polynomial-time 2-approximation algorithm for the Traveling Salesman Problem.

**Solution:** False, assuming  $P \neq NP$ . There is an approximation algorithm in the special case where the graph obeys the triangle inequality, but we don't know of one in general.

- (e) **T F** [3 points] A dynamic programming algorithm that solves  $\Theta(n^2)$  subproblems could run in  $\omega(n^2)$  time.

**Solution:** True. It could take  $\omega(1)$  time per subproblem.

- (f) **T F** [3 points] If  $A$  is a Monte Carlo program computing a predicate  $f(x)$ , and  $B$  is a Las Vegas program computing a predicate  $g(x)$ , then

```
if A(x) then
    return B(x)
else
    return False
```

is a Monte Carlo program computing  $(f(x) \wedge g(x))$ .

**Solution:** False.  $B$  has a chance of taking arbitrarily long, so the algorithm is not Monte Carlo.

- (g) T F [3 points] Dynamic programming programs require space at least proportional to the number of subproblems generated (in order to “memoize” the solution to each subproblem).

**Solution:** False. Under some circumstances, we can reuse the same space for multiple subproblems; for example, if each subproblem of size  $k$  only looks at subproblems of size  $k - 1$ , then when calculating bottom-up we need not store subproblems of size  $k - 2$  once all subproblems of size  $k - 1$  have been calculated. (See the discussion of longest common subsequence in CLRS.)

- (h) T F [3 points] Let  $H = \{h_i : \{1, 2, 3\} \rightarrow \{0, 1\}\}$  be a hash family defined as follows.

	1	2	3
$h_1$	0	1	0
$h_2$	1	0	1
$h_3$	1	1	0

(For example,  $h_1(3) = 0$ .)

Then  $H$  is a universal hash family.

**Solution:** False. Consider elements 1 and 3:  $h_1$  and  $h_2$  both cause a collision between them, so in particular a uniformly random hash function chosen from  $H$  causes a collision between 1 and 3 with probability  $2/3$ , greater than the  $1/2$  allowed for universal hashing (since there are 2 hash buckets).

- (i) **T F** [3 points] If we use a max-queue instead of a min-queue in Kruskal's MST algorithm, it will return the spanning tree of maximum total cost (instead of returning the spanning tree of minimum total cost). (Assume the input is a weighted connected undirected graph.)

**Solution:** True. The proof is essentially the same as for the usual Kruskal's algorithm. Alternatively, this is equivalent to negating all the edge weights and running Kruskal's algorithm.

- (j) **T F** [3 points] Define a graph as being *tripartite* if its vertices can be partitioned into three sets  $X_1, X_2, X_3$  such that no edge in the graph has both vertices in the same set. (That is, all edges are between vertices in different sets.) Then deciding whether a graph is tripartite can be done in polynomial time.

**Solution:** False, assuming  $P \neq NP$ . This is exactly 3-colorability; partitioning the vertices into three sets with no internal edges is the same as coloring them with three colors such that no edge has two endpoints of the same color. As seen in lecture, 3-coloring is NP-complete.

- (k) **T F** [3 points] A randomized algorithm for a decision problem with one-sided-error and correctness probability  $1/3$  (that is, if the answer is YES, it will always output YES, while if the answer is NO, it will output NO with probability  $1/3$ ) can always be amplified to a correctness probability of 99%.

**Solution:** True. Since the error is one-sided, it in fact suffices for the correctness probability to be any constant  $> 0$ . We can then repeat it, say,  $k$  times, and output NO if we ever see a NO, and YES otherwise. Then, if the correct answer is YES, all  $k$  repetitions of our algorithm will output YES, so our final answer is also YES, and if the correct answer is NO, each of our  $k$  repetitions has a  $1/3$  chance of returning NO, in which case our final answer is, correctly, NO, with probability  $1 - (2/3)^k$ , so  $k = \log_{3/2} 100$  repetitions suffice.

- (l) **T F** [3 points] Let  $B_0, B_1, B_2, \dots$  be an infinite sequence of decision problems, where  $B_0$  is known to be NP-hard and

$$B_i \leq_P B_{i+1} \text{ for all } i \geq 0.$$

Then it must be the case that  $B_i$  is NP-hard for all  $i \geq 0$ .

**Solution:** True. This can be seen by induction; if  $B_i$  is NP-hard, and there is a polynomial-time reduction from  $B_i$  to  $B_{i+1}$ , then  $B_{i+1}$  is NP-hard.

- (m) **T F** [3 points] Let  $L$  be a decision problem. If there exists an interactive proof for  $L$  where the verifier is deterministic, then  $L \in NP$ .

**Solution:** True. If the verifier is deterministic, the transcript of the interactions between the prover and verifier will always be the same. Thus, the transcript itself is a polynomial-sized certificate for  $L$ .

- (n) **T F** [3 points] Let  $L$  be a decision problem. If there exists an interactive proof for  $L$  where the prover runs in polynomial time, then  $L \in P$ .

**Solution:** False. Either the prover or the verifier could be randomized, which would allow them to prove a larger class of problems (assuming  $P \neq BPP$ ).

**Problem 2. Short Answer [41 points] (9 parts)**

Give *brief*, but complete, answers to the following questions.

- (a) [7 points] Suppose that  $n$  women check their coats at a concert. However, at the end of the night, the attendant has lost the claim checks and doesn't know which coat belongs to whom. All of the women came dressed in black coats that were nearly identical, but of different sizes. The attendant can have a woman try a coat, and find out whether the coat fits (meaning it belongs to that woman), or the coat is too big, or the coat is too small. However, the attendant cannot compare the sizes of two coats directly, or compare the sizes of two women directly. Describe how the attendant can determine which coat belongs each woman in expected  $O(n \log n)$  time. Give a brief analysis of the running time of your algorithm.

**Solution:** This can be solved with a modification of randomized Quicksort.

- Pick a random coat (the “pivot coat”), and have all the women try it on. Now we have the woman who fits the coat (the “pivot woman”), and a partition of the rest of the women into those who are smaller than the pivot and those who are larger.
- Have the woman who fit the coat try on all the coats. Now we have a partition of the rest of the coats into those that are smaller and those that are larger than the pivot.
- Recursively solve the two smaller subproblems – matching the women and coats that are smaller than the matched pair, and matching the women and coats that are larger than the matched pair.

This algorithm makes twice as many comparisons as randomized Quicksort does on an array of size  $n$ . Therefore, the asymptotic expected running time is the same as that of randomized Quicksort –  $O(n \log n)$ .

- (b)** [4 points] Let  $F_1, F_2, \dots = 1, 1, 2, 3, 5, 8, \dots$  denote the usual sequence of Fibonacci numbers (defined by  $F_1 = 1$ ,  $F_2 = 1$ , and  $F_i = F_{i-1} + F_{i-2}$  for  $i > 2$ ).

Suppose that a file to be compressed contains  $k$  different symbols  $a_1, a_2, \dots, a_k$  and that it contains  $F_i$  occurrences of  $a_i$  for each  $i$ . Thus, if  $k = 4$ , the string has length 7 and contains 2 occurrences of  $a_3$ .

Assume the file is encoded with Huffman encoding. How many bits will be used to encode  $a_i$ , as a function of  $i$  and/or  $k$ ? State your answer concisely. You do not need to provide a proof.

**Solution:**  $(k+1) - i$  for  $i > 1$ ;  $k-1$  for  $i = 1$

- (c)** [4 points] As a final project for one of your other Course 6 classes, you have a massive program to run. After much effort, you are able to parallelize 90% of your code. The computer lab has two systems on which you could run your program:

- a cluster of 90 single-core computers each running at 1GHz, and
- a computer with 9 cores each running at 2GHz.

Which one should you choose to complete your project as quickly as possible?

**Solution:** Compared to a single 1GHz single-core machine, the first option offers a speedup factor of

$$\frac{1}{.1 + .9/90} = \frac{1}{.11} \approx 9,$$

while the second offers a speedup factor of

$$\frac{2}{.1 + .9/9} = \frac{2}{.2} = 10.$$

So you should go with the second.

- (d) [5 points] Recall the clique problem from lecture: Given an undirected graph  $G = (V, E)$  and a positive integer  $k$ , is there a subset  $C$  of  $V$  of size at least  $k$  such that every pair of vertices in  $C$  has an edge between them?

Ben Bitdiddle thinks he can solve the clique problem in polynomial time using linear programming.

- Let each variable in the linear program represent whether or not each vertex is a part of our clique. Add constraints stating that each of these variables must be nonnegative and at most one.
- We go through the graph  $G$  and consider each pair of vertices. For every pair of vertices where there is *not* an edge in  $G$ , add a constraint stating that the sum of the variables corresponding to the endpoint vertices must be at most one. This ensures that both of them cannot be part of a clique if there is no edge between them.
- The objective function is the sum of the variables corresponding to the vertices. We wish to maximize this function.

Ben argues that the value of the optimum must be the size of the maximum-size clique in  $G$ , and we can then simply compare this value to  $k$ . Explain the flaw in Ben's logic.

**Solution:** This is an integer program, not a linear program, and therefore we don't know how to solve it in polynomial time. (Alternatively, if we don't add the integrality constraints, we can solve it in polynomial time but will likely get fractional values back; it's unclear what fractional values of the variables mean with regard to the clique.)

- (e) [4 points] In a weighted connected undirected graph that might have negative-weight edges but no negative-weight cycles, how would you find a triple of distinct vertices  $x, y, z$  that minimizes  $f(x, y, z) = d(x, y) + d(y, z) + d(z, x)$  where  $d(u, v)$  is the length of the shortest path from  $u$  to  $v$ ?

The running time of your algorithm should be  $O(n^3)$ , where  $n$  is the number of vertices in the graph.

**Solution:** Run Johnson's all-pairs shortest-paths algorithm to find all of the shortest paths  $d(u, v)$ . This takes  $O(V^2 \log V + VE) = O(n^3)$  time, since  $E = O(n^2)$ . Then calculate  $f$  for all triples of vertices in the graph, and take the minimum. There are  $O(n^3)$  triples, and  $f$  can be calculated in  $O(1)$  time given the  $d(u, v)$  values, so this step also takes  $O(n^3)$  time.

- (f) [4 points] Suppose you are using RSA and you change your public key  $(e, N)$  every so often, where  $N = pq$  is the product of your two large secret primes.

Why is it *not* a good idea to leave  $p$  the same and just replace  $q$  with a different secret prime  $q'$  (so your new  $N'$  is just  $pq'$ )?

**Solution:** Anyone could compute the GCD of two of your public keys (using the Euclidean algorithm, which is polynomial-time) to find  $p$ , and thus factor  $N$  and  $N'$ .

- (g) [7 points] You are working at a hospital trying to diagnose patients; you may assume that each patient has exactly one disease. You know of  $m$  different diseases  $d_1, d_2, \dots, d_m$ . You have  $n$  different tests you can run (labeled  $T_1, T_2, \dots, T_n$ ), each of which comes up positive for some set of diseases and negative for other diseases. You would like to correctly diagnose all patients while giving them the minimum necessary number of tests—or, at least, close to the minimum number. Since you must send the tests to the lab for processing, all tests must be performed in parallel.

We say that a set of tests  $T \subseteq \{T_1, T_2, \dots, T_n\}$  is *comprehensive* if, for every pair of diseases  $(d_i, d_j)$ , there is some test  $T_k \in T$  that distinguishes them—that is, it returns positive for one and negative for the other. The minimum-comprehensive-set problem (MCS) is the problem of finding a comprehensive set of tests of minimum cardinality. MCS is known to be NP-hard.

Describe a polynomial-time  $\alpha$ -approximation algorithm for the MCS problem, where  $\alpha = \ln(m(m - 1)/2)$ .

**Solution:** For every pair of diseases, there is at least one of the tests that distinguishes them, and we want a minimum-cardinality set of the tests that between them distinguish all diseases. This is simply the Set Cover problem operating on *pairs* of diseases; we can use the standard approximation for Set Cover seen in CLRS/lecture.

- (h) [3 points] State the three properties a trapdoor function should have.

**Solution:** Easy to compute, hard to invert without the trapdoor information, easy to invert with the trapdoor information.

- (i) [4 points] Suppose you are given a polynomial time algorithm DECISION-FACTOR that, given two integers  $k$  and  $n$ , returns YES if  $n$  has a prime factor less than  $k$ , and NO if  $n$  does not. Give a polynomial time algorithm for computing a single prime factor of  $n$ .

**Solution:** Use DECISION-FACTOR in a binary search to find the smallest prime factor  $p$  of  $n$ : for every  $m \leq p$  we have  $\text{DECISION-FACTOR}(m) = \text{NO}$  and for every  $m > p$  we have  $\text{DECISION-FACTOR}(m) = \text{YES}$ . This takes  $O(\log n)$  calls to DECISION-FACTOR, which is polynomial (linear, in fact) in the length of the input  $n$ , so the overall running time is polynomial as well.

**Problem 3. More Spy Games [9 points]**

An enemy country, Elbonia, has  $n$  transmitter/receiver pairs  $(t_i, r_i)$ . You can model the position of each  $t_i$  and each  $r_i$  as a point in the plane. Enemy communications travel along the straight-line segment from  $t_i$  to  $r_i$ . You can place eavesdrop units at any point in the plane, but a unit must be on the line segment from  $t_i$  to  $r_i$  in order to eavesdrop successfully. If you put a unit at the intersection of two such segments, that unit can eavesdrop on both transmitter/receiver pairs. Assume no three such segments intersect at a point.

Your intelligence agency has given you a list of the coordinates of all  $n$  enemy transmitter/receiver pairs. Briefly describe a polynomial time algorithm for finding the minimum number of eavesdrop units required to eavesdrop on all  $n$  transmitter/receiver pairs. (No proof needed.)

**Solution:** Construct a graph  $G$  consisting of a vertex  $v_i$  for each transmitter/receiver pair  $(t_i, r_i)$ , and an edge between vertices  $v_i$  and  $v_j$  if the corresponding line segments intersect. This can be done in  $O(n^2)$  time. Now, any vertex that is isolated must be eavesdropped on by its own dedicated unit, and we can remove it from consideration. The problem then reduces to finding a minimum edge cover of  $G$ , that is, the minimum number of edges such that every vertex in  $G$  is incident on at least one.

We can do this by first finding a maximum matching in  $G$  (using any of several matching algorithms covered in class, all of which run in polynomial time), and adding an edge to cover each of the remaining uncovered vertices. To see why this indeed achieves a minimum edge cover, observe that any edge cover contains a matching, each edge of which covers two vertices, together with some additional edges, each of which covers a single additional vertex. Thus the smallest edge cover we can hope to obtain comprises, in this manner, of a maximum matching of  $G$  together with an edge for each remaining unmatched vertex. But this is indeed what we construct, so it must be the minimum edge cover, and we are done.

**Problem 4. Almost Sorted [9 points]**

A sequence  $x_1, x_2, \dots, x_n$  of real numbers is said to be **sorted** if

$$x_1 \leq x_2 \leq \dots \leq x_n.$$

We say that  $x_1, x_2, \dots, x_n$  is **D-almost-sorted** for a non-negative real number  $D$  if there exists another sequence  $y_1, y_2, \dots, y_n$  of real numbers such that  $y_1, y_2, \dots, y_n$  is sorted, and  $\sum_i |x_i - y_i| \leq D$ . (That is, by “shifting” values  $x_i$  to new values  $y_i$ , such that the total amount of shifting is at most  $D$ , the new set of numbers is sorted.)

Describe concisely a polynomial-time algorithm which, given an input sequence  $x_1, x_2, \dots, x_n$  and a non-negative real number  $D$ , determines whether  $x$  is  $D$ -almost-sorted.

**Solution:** We solve this problem using linear programming. To determine whether a sequence  $x_1, x_2, \dots, x_n$  is  $D$ -almost-sorted, check whether the following LP is feasible:

$$\begin{array}{ll} \text{minimize} & 0 \\ \text{subject to} & \\ & x_i = y_i + c_i - d_i & \text{for } i = 1, \dots, n \\ & y_i \leq y_{i+1} & \text{for } i = 1, \dots, n-1 \\ & \sum_i (c_i + d_i) \leq D & \end{array}$$

(The objective function is irrelevant.)

Alternatively, we could solve the following LP, and then check whether the optimal value of the objective function is at most  $D$ .

$$\begin{array}{ll} \text{minimize} & \sum_i (c_i + d_i) \\ \text{subject to} & \\ & x_i = y_i + c_i - d_i & \text{for } i = 1, \dots, n \\ & y_i \leq y_{i+1} & \text{for } i = 1, \dots, n-1 \end{array}$$

Linear programming can be solved in worst-case polynomial time by the ellipsoid algorithm or interior-point methods.

**Problem 5. Randomized 3-Coloring** [8 points] (3 parts)

In an undirected graph  $G = (V, E)$ , a *coloring* is a mapping  $c$  which assigns colors to vertices. We denote the color of vertex  $v$  by  $c(v)$ .

We say a coloring  $c$  *satisfies* an edge  $e = (u, v)$  if  $c(u) \neq c(v)$  (that is, the endpoints of the edges are assigned different colors). Let the function  $s(c)$  count the number of satisfied edges under a coloring  $c$ .

Define the *3-coloring optimization problem* as follows: Given an undirected graph  $G = (V, E)$ , output a coloring  $c$  such that  $c(v) \in \{R, W, B\}$  for all  $v \in V$ , such that  $s(c)$  is maximized.

Here is one very simple randomized algorithm:

RANDOMIZED-COLOR( $G$ )

- 1 **for** each  $v \in V$
- 2     Pick a color uniformly at random in  $\{R, W, B\}$
- 3     Let  $c(v) =$  color picked
- 4 **return**  $c$

- (a) [2 points] Let  $e$  be any edge. What is the probability that the coloring picked satisfies  $e$ ?

**Solution:** 2/3

- (b) [2 points] What is the expected number of edges satisfied by the coloring produced by  $c$ ? Justify.

**Solution:**  $2|E|/3$ , due to linearity of expectation over all edges.

- (c) [4 points] Show that RANDOMIZED-COLOR is a polynomial-time randomized  $(3/2)$ -approximation algorithm for the 3-coloring optimization problem. That is, show that  $E(s(c)) \geq (2/3)s(c^*)$  where  $c^*$  is the optimal coloring.

**Solution:** The optimal coloring can satisfy at most  $|E|$  edges, so  $s(c^*) \leq |E|$ . From (b),  $E[s(c)] = 2|E|/3$ . Thus,  $E(s(c)) \geq (2/3)s(c^*)$ .

**Problem 6. Sublinear-Time Unimodal Testing [10 points]**

We say that an array  $A[1..n]$  of real numbers is **unimodal** if there exists an integer  $k$  such that  $1 \leq k \leq n$ ,  $A[1..k]$  is monotonically non-decreasing, and  $A[k..n]$  is monotonically non-increasing.

We say that  $A$  is  $\epsilon$ -far from being unimodal if you have to remove more than  $\epsilon n$  elements from  $A$  in order for the remaining sequence to be unimodal.

Give a sublinear-time property tester that, given an array  $A[1..n]$  of *distinct* real numbers:

- if  $A$  is unimodal, outputs YES with probability 1, and
- if  $A$  is  $\epsilon$ -far from being unimodal, outputs NO with probability at least 2/3.

**Solution:** We first use binary search to find a candidate  $k$ : each query will be of two consecutive indices, to see if we are to the left of  $k$  or to the right. Since  $A$  consists of all distinct values, we will never get a tie. Then, once we have such a  $k$ , we run our monotonicity tester with parameter  $\epsilon$  on  $A[1..k]$  and  $A[k..n]$ . If both return YES, we return YES, otherwise we return NO.

To see that this is correct, suppose that  $A$  is unimodal. Then our search for  $k$  must return the correct  $k$ , and  $A[1..k]$  and  $A[k..n]$  must both be monotone, so our subroutines both return YES and we return YES as well, as required. On the other hand, if  $A$  is  $\epsilon$ -far from being unimodal, meaning we need to remove  $\epsilon n$  elements from  $A$  to make it unimodal, then no matter what  $k$  we pick, either  $A[1..k]$  or  $A[k..n]$  must be  $\epsilon$ -far from being monotone. Specifically, suppose, to get a contradiction, that  $A[1..k]$  and  $A[k..n]$  are both at most  $\epsilon'$ -far from being monotone, for some  $\epsilon' < \epsilon$ . This means we can remove  $\epsilon' k$  elements from  $A[1..k]$  to make it monotone, and  $\epsilon' (n - k)$  elements from  $A[k..n]$  to make it monotone. But then we could remove these same  $\epsilon' k + \epsilon' (n - k) = \epsilon' n < \epsilon n$  elements from  $A[1..n]$  to make it unimodal, contradicting the fact that  $A$  is  $\epsilon$ -far from being unimodal. It follows that at least one of our subroutines must return NO with probability at least 2/3, so we also return NO with probability at least 2/3.

Note that our monotonicity tester allows for a specification of the direction of monotonicity (i.e., increasing or decreasing).

## SCRATCH PAPER

## SCRATCH PAPER

## Final Exam Solutions

- Do not open this quiz booklet until you are directed to do so. Read all the instructions first.
- The quiz contains 12 problems, several with multiple parts. You have 180 minutes to earn 150 points.
- This quiz booklet contains 15 pages, including this one, and a sheet of scratch paper.
- This quiz is closed book. You may use two double-sided letter ( $8\frac{1}{2}'' \times 11''$ ) or A4 crib sheet. No calculators or programmable devices are permitted. Cell phones must be put away.
- Write your solutions in the space provided. If you run out of space, continue your answer on the back of the same sheet and make a notation on the front of the sheet.
- Do not waste time deriving facts that we have studied. Just cite results from class.
- When we ask you to “give an algorithm” in this quiz, describe your algorithm in English or pseudocode, and provide a short argument for correctness and running time. You do not need to provide a diagram or example unless it helps make your explanation clearer.
- Do not spend too much time on any one problem. Generally, a problem’s point value is an indication of how many minutes to spend on it.
- 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. Please be neat.
- Good luck!

Problem	Points	Parts	Grade	Problem	Points	Parts	Grade
0	3	1		7	10	1	
1	22	20		8	20	1	
2	10	2		9	10	1	
3	10	2		10	5	1	
4	15	3		11	10	2	
5	10	1		12	15	2	
6	10	1		Total	150		

Name: \_\_\_\_\_

Circle your recitation:

F10	F11	F11	F12	F12	F1	F2	F3
R01	R02	R07	R03	R08	R04	R05	R06
Yotam	Boon Teik	Aizana	Annie	Aizana	Annie	Katherine	Heejung

**Problem 0. Name.** [3 points] Write your name on every page of this exam booklet! Don't forget the cover.

**Problem 1. True or False.** [22 points] (20 parts)

Circle **T** or **F** for each of the following statements to indicate whether the statement is true (T) or false (F). **No explanation is needed.**

- (a) **T F** Las Vegas algorithms are a class of randomized algorithms that always give the correct answer.

**Solution:** True.

- (b) **T F** A Monte Carlo algorithm runs in worst-case polynomial time.

**Solution:** True.

- (c) **T F** Prim's algorithm on a graph  $G(V, E)$  takes  $O(|V| \log |V|)$  time if implemented with a binary heap.

**Solution:** False. It takes  $O(|E| \log |V|)$  time.

- (d) **T F** If a flow  $f$  on a graph  $G$  has no augmenting paths, then  $|f|$  is equal to the capacity of the minimum cut in  $G$ .

**Solution:** True.

- (e) **T F** Paths along a minimum spanning tree are shortest paths.

**Solution:** False.

- (f) **T F** Solving a linear program whose variables are constrained to be real numbers is NP-hard.

**Solution:** False. ILP is NP-hard, while (real-valued) LP is in P.

- (g) **T F** The simplex algorithm is a polynomial-time algorithm.

**Solution:** False. The worse-case running time is exponential time, although it is often more efficient than known polynomial-time LP algorithms in practice.

- (h) T F [3 points] Consider the following snapshot of the simplex algorithm, where  $z$  denotes the objective value, and  $x_1, \dots, x_5$  are variables constrained to be nonnegative. The current basic variables are  $x_4$  and  $x_5$ . If  $x_2$  is chosen as the entering variable for the next pivoting operation, then  $x_4$  is the leaving variable.

$$\begin{aligned} z &= x_1 + 4x_2 + x_3 \\ x_4 &= 6 - x_1 - 2x_2 - 3x_3 \\ x_5 &= 10 - 2x_1 - 5x_2 - x_3 \end{aligned}$$

**Solution:** False.  $x_5$  is the leaving variable. The second equation gives  $x_2 \leq 3$ , while the third equation gives  $x_2 \leq 2$ , a tighter bound.

- (i) T F For NP-complete decision problems, there always exist certificates that can be verified in polynomial time.

**Solution:** True.

- (j) **T F** NP-hard problems are problems that are strictly harder than all NP problems.

**Solution:** False. NP-hard problems are at least as hard as all NP problems.

- (k) **T F** If problem  $A$  is NP-hard and can be reduced by Karp reduction to problem  $B$ , then  $B$  is also NP-hard.

**Solution:** True.

- (l) **T F** Let  $A$  and  $B$  be optimization problems where it is known that  $A$  reduces to  $B$  in quadratic time. If there exists a polynomial-time 2-approximation for  $B$ , then there also exists a polynomial-time 2-approximation for  $A$ .

**Solution:** False. Approximation factor is not necessarily carried over in polynomial-time reduction. e.g., set cover vs. vertex cover.

- (m) **T F** Assume  $P \neq NP$ . The general Traveling Salesman Problem has a polynomial-time  $\alpha$ -approximation algorithm for some constant  $\alpha > 1$ .

**Solution:** False. Assuming  $P \neq NP$ , there is a polynomial-time approximation algorithm for TSP with the triangular inequality, but not the general TSP. For example, the case where all weights are 0 corresponds to the Hamiltonian Cycle problem, which is NP-complete.

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

**Solution:** True. This is the aggregate method.

- (o) **T F** The amortized cost of any single operation should always be greater than or equal to the actual cost of that operation, in order to bound the total actual cost for any sequence of operations.

**Solution:** False. It is true for the total cost of a sequence of operations, but not necessarily for any single operation.

- (p) **T F** Leader election in a bidirectional ring of  $n$  processes can be performed in  $O(\log n)$  rounds.

**Solution:** True. Use the hierarchical algorithm from lecture.

- (q) **T F** Using the simplified Luby's algorithm presented in lecture, a maximal independent set of vertices of an arbitrary undirected graph  $G(V, E)$  can be found in expected  $O(|V| \log |V|)$  time.

**Solution:** True. The maximum degree is bounded above by  $|V|$ .

- (r) **T F** A hash function that is one-way (OW), target collision resistant (TCR) and non-malleable (NM) is guaranteed to be collision resistant (CR).

**Solution:** False.

- (s) **T F** There is an existing deterministic polynomial-time algorithm that determines whether or not a number is composite.

**Solution:** True. Test if it is a prime, which takes polynomial time.

- (t) **T F** A computational problem in P is guaranteed to have an interactive proof.

**Solution:** True. The execution of the polynomial-time algorithm itself is a legal proof.

**Problem 2. Selection [10 points]**

Consider the recursive expression for the running time  $T(n)$  of  $\text{SELECT}(A, i)$ , an algorithm that returns the  $i$ th smallest element in an array  $A$  of  $n$  distinct elements:

$$T(n) \leq \begin{cases} O(1) & \text{if } n < 140, \\ T(\lceil n/5 \rceil) + T(7n/10 + 6) + O(n) & \text{if } n \geq 140. \end{cases}$$

For each of the three terms in the latter case where  $n \geq 140$ , specify which step(s) in the following outline of  $\text{SELECT}$  contributed to the term. Briefly explain why.

$\text{SELECT}(A, i)$

1. Divide the  $n$  elements into groups of 5 elements each, sort each group, and find the median within each group.
2. Use  $\text{SELECT}$  recursively to find the median  $x$  of the medians found in step 1.
3. Partition the input array  $A$  around  $x$ . Suppose  $x$  is the  $k$ th largest element, i.e., there are  $k - 1$  elements on the low side of the partition, and  $n - k$  elements on the high side.
4. If  $i = k$ , then return  $x$ . Otherwise, use  $\text{SELECT}$  recursively to find the  $i$ th smallest element on the low side if  $i < k$ , or the  $(i - k)$ th smallest element on the high side if  $i > k$ .

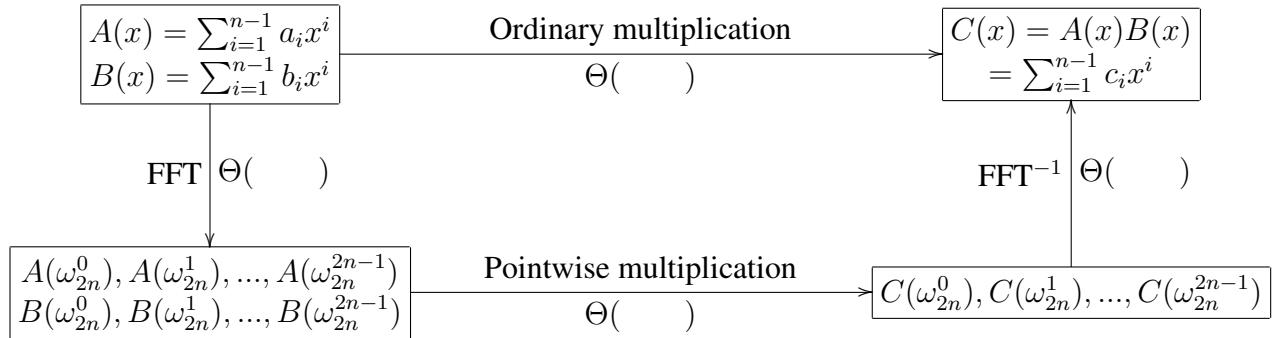
Term	Step(s)	Brief explanation
$T(\lceil n/5 \rceil)$		
$T(7n/10 + 6)$		
$O(n)$		

**Solution:**

- $T(\lceil n/5 \rceil)$ : from step 2, recursively finding the median-of-medians;
- $T(7n/10 + 6)$ : from step 4, recursively performing  $\text{SELECT}$ . The eliminated side contains at least  $3(\lceil \frac{1}{2}\lceil \frac{n}{5} \rceil \rceil - 2) \geq \frac{3n}{10} - 6$  elements, so the chosen side contains at most  $7n/10 + 6$  elements.
- $O(n)$ : from step 1, sorting  $n$  groups of 5 elements each, and from step 3, partitioning the input around the median-of-medians.

**Problem 3. Fast Fourier Transform [10 points] (2 parts)**

The following diagram illustrates the outline of an efficient polynomial-multiplication process, using the Fast Fourier Transform (FFT):



- (a)** Fill in the running times above.

**Solution:** Ordinary multiplication:  $\Theta(n^2)$ , FFT and  $\text{FFT}^{-1}$ :  $\Theta(n \log n)$ , pointwise multiplication:  $\Theta(n)$ .

- (b)** What is  $\omega_{2n}^k$ ,  $k \in \{0, 1, \dots, 2n - 1\}$ , in the lower two boxes? Describe it with a concise verbal explanation or terminology, as well as a mathematical expression. (Caution: Note that the expression involves  $2n$  and not  $n$ .)

**Solution:**  $\omega_{2n}^k$  is one of the  $2n$  complex  $(2n)$ th roots of unity; in particular,

$$\omega_{2n}^k = e^{\frac{2\pi i k}{2n}}.$$

**Problem 4. Modifying Paths [15 points] (3 parts)**

You are given a weighted directed graph  $G = (V, E)$  with weights  $w: (u, v) \rightarrow \mathbb{R}$ . Your friend has already computed the all pairs shortest path problem for this graph.

How long would it take to recompute the all pairs shortest paths if you only changed the weight of the  $(n - 1, n)$  edge in the following circumstances.

- (a) Your friend gives you the answers to the subproblems from the  $O(n^4)$  dynamic programming algorithm.

**Solution:**  $O(n^4)$  time. While the first two iterations (filling out  $d_{uv}^{(1)}$  and  $d_{uv}^{(2)}$ ) can be done without recalculating most of the entries, the third iteration onwards may have many entries differing from the original problem. Thus, it will still take  $O(n^4)$  time.

Using Johnson's algorithm for a time of  $O(nm + n^2 \log n)$  or Floyd-Warshall for a time of  $O(n^3)$  was also acceptable.

- (b) Your friend gives you the intermediate matrices from the fast matrix multiplication algorithm.

**Solution:**  $O(n^3 \log n)$  time. After taking the matrix to the fourth power, all entries may be different, which means that all the matrices have to be redone.

Using Johnson's algorithm for a time of  $O(nm + n^2 \log n)$  or Floyd-Warshall for a time of  $O(n^3)$  was also acceptable.

- (c) Your friend gives you the answers to the subproblems from the Floyd-Warshall algorithm.

**Solution:**  $O(n^2)$  time. For all  $u \in V$ ,  $C_{u,n}^{(n-1)}$  may use edge  $(n - 1, n)$ . This changes  $O(n)$  calculations. Subsequently, in calculating  $C_{uv}^{(n)}$ , all pairs  $u, v \in V$  may use the result from  $C_{u,n}^{(n-1)}$ . This changes  $O(n^2)$  calculations. Finally, the distance between  $n - 1$  and  $n$  may be different. Since no paths may use vertex  $n - 1$  or  $n - 1$  in the first  $n - 2$  iterations, the algorithm can skip calculating  $C_{n-1,n}$  for these calculations. It takes  $O(n)$  time to loop through the other vertices and find some intermediate value for the path from  $n - 1$  to  $n$ .

**Problem 5. Edge connectivity** [10 points] (1 parts)

The *edge connectivity* is the minimum number  $k$  of edges that must be removed to disconnect the graph into two or more components. Given an undirected, unweighted graph  $G$  with vertices  $V$  and edges  $E$ , design an algorithm which computes the edge connectivity. Provide the runtime analysis for your algorithm. You may give your runtime in terms of  $T(n, m)$ , the best algorithm for computing maximum flow values on a graph of  $n$  nodes and  $m$  edges.

**Solution:** This can be solved by making a flow network out of the graph, where all edges have edge capacity of 1. Choose one of the vertices  $V$  as a source node  $S$ , and iterate through the remaining vertices and set it as the sink node  $T$ . Running a maximum-flow algorithm with the given source and sink yields the minimum cut of the flow network which is equivalent to minimum number of edges that must be removed to disconnect the source from sink. Take the minimum of the max flow value over the sink node choices. Running the Ford-Fulkerson algorithm takes  $O(|E| * k)$ , which is bounded by  $O(|E| * |V|)$ . We run  $|V - 1|$  iterations of the algorithm, so the entire algorithm has time complexity of  $O(|E| * |V|^2)$ .

**Problem 6. Linear Program** [10 points] (1 parts)

Consider the following linear program:

$$\begin{aligned} \max_{x,y} \quad & 5x - 3y \\ \text{subject to} \quad & x - y \leq 1 \\ & 2x + y \leq 2 \\ & x, y \geq 0 \end{aligned}$$

Show that  $x = 1, y = 0$  is an optimal solution by writing down the dual problem and using duality (i.e., select a dual feasible solution to provide an upper bound on the optimal primal objective value).

**Solution:** The dual problem is

$$\begin{aligned} \min_{u,v} \quad & u + 2v \\ \text{subject to} \quad & u + 2v \geq 5 \\ & v - u \geq -3 \\ & u, v \geq 0 \end{aligned}$$

The solution  $u = 1, v = 2$  gives an objective value of 5. By duality, this objective value 5 of the dual feasible solution is an upper bound on the objective value of any primal feasible solution. The objective value of the primal feasible solution  $x = 1, y = 0$  is also 5. Therefore,  $x = 1, y = 0$  is a primal optimal solution.

**Problem 7. ILP and Total Unimodularity** [10 points] (1 parts)

For any integer  $n \times n$  matrix  $B$  and any integer  $1 \times n$  vector  $c$ , prove that the following ILP (integer linear program) can be solved in polynomial time:

$$\begin{aligned} & \max_x \quad cx \\ \text{subject to} \quad & x_i - x_j \leq B_{ij} \quad \forall i \in \{1, \dots, n\}, j \in \{1, \dots, n\} \\ & x_i \geq 0 \quad \forall i \in \{1, \dots, n\} \\ & x = [x_1 \quad x_2 \quad \cdots \quad x_n]^T \text{ is an integer-valued vector} \end{aligned}$$

where  $B_{ij}$  denotes the  $(i, j)$ -th entry in the matrix  $B$ .

**Solution:** The given linear program, without the integrality constraint, is of the form  $Ax \leq B$ ,  $x \geq 0$ , and can be solved in polynomial time. The matrix  $A$  is of size  $n^2 \times n$  where each row corresponds to one constraint. It's easy to show that  $A^T$  satisfies the sufficiency constraints for total unimodularity. The entries of  $A^T$  are 0, 1 and -1. The columns of  $A^T$  are the rows of  $A$ . Therefore, each column of  $A^T$  has exactly two non-zero entries: 1 and -1. We can partition the rows of  $A^T$  into  $M_1 = \emptyset$  and  $M_2 = M$ , where  $M$  is the set of all rows of  $A^T$ . We have that for each column of  $A^T$ , the sum of entries in each partition equals to 0. Since  $A^T$  is TU,  $A$  also must be TU. This means that the optimal solution is integral and can be found in polynomial time.

**Problem 8. Universal Hashing [20 points] (2 parts)**

For a matrix  $A$  of size  $m \times n$  with  $\{0, 1\}$  entries, define the hash function

$$h_A(x) = Ax$$

where the input  $x$  is a binary column vector of length  $n$  and all operations are done mod 2. In this problem you will show that the hash family  $H$  that consists of all such hash functions (for a fixed  $m$  and  $n$ ) is universal.

- (a) [10 points]** Let  $x$  and  $x'$  be column vectors of length  $n$  where each entry is in  $\{0, 1\}$  and  $x \neq x'$ . Let  $r$  be a random row vector of length  $n$  such that each entry is chosen from  $\{0, 1\}$  uniformly and independently at random. Show that

$$\Pr[rx = rx' \pmod{2}] = \frac{1}{2}$$

**Solution:**

For any two inputs  $x \neq x'$ , there is a difference at least in one entry. Without loss of generality, let there be a difference in the  $j$ th entry, i.e., let the index  $j$  be such that  $x_j \neq x'_j$ . Consider all possible  $1 \times n$  rows with  $\{0, 1\}$  entries. It is possible to pair them by difference in index  $j$ . For any such pair  $(r, r')$ , we have that

$$rx - rx' \neq r'x - r'x'.$$

Therefore, for a randomly drawn row vector  $r$ ,

$$\Pr[rx = rx'] = \frac{1}{2}.$$

- (b) [10 points]** Using the result from part (a), show that  $H$  is a universal hash family.

**Solution:**

Using the result from part (a), for a random matrix  $A$ , we have that

$$\Pr[Ax = Ax'] = \left(\frac{1}{2}\right)^m$$

because  $Ax = Ax'$  only when each bit of the output is the same. Each output bit is determined by a row in  $A$  and the output size is  $m$ . Therefore, we must have that  $H$  is a universal hash family.

**Problem 9. Approximation** [10 points] (1 parts)

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

**Solution:** Solution: Assume that there exists an  $(1 + \frac{1}{2n})$ -approximation algorithm  $A$  for Vertex Cover. For a given graph  $G = (V, E)$ , let  $S$  be a minimum vertex cover for  $G$ . Then  $A$  can decide if  $G$  has a vertex cover of size at most

$$|S| \left(1 + \frac{1}{2n}\right) < |S| + 1$$

That implies that  $A$  can solve Vertex Cover in polynomial time. Contradiction.

**Problem 10. Second Best is Leader [5 points] (1 parts)**

Suppose we have a ring of  $n$  processes connected using bidirectional links. Assume that the processes have unique, strictly positive IDs. We wish to find the process with the *second* highest ID. Give an algorithm that can do this using  $O(n^2)$  messages. You can quote algorithms from lecture.

**Solution: [5 points]**

- 1.In the first phase, find the maximum ID using the algorithm shown in lecture.
- 2.The process with maximum ID should pretend its ID is 0 and merely pass along the IDs it receives in the second phase (Step 3 below).
- 3.Find the process with the maximum ID in the remaining processes.

Since the algorithm given in lecture uses  $O(n^2)$  messages, the above uses  $O(n^2)$  messages.

**Problem 11. Ben Bitdiddle Tries Cryptography** [10 points] (2 parts) Ben Bitdiddle proposes the following public key scheme. Alice chooses a secret key  $x$ , and computes  $g^x \bmod N$  as her public key.  $g$  and  $N$  are public parameters known to all. If Bob wants to encrypt a message  $m$ , he computes  $g^{x \cdot m}$  and sends it to Alice.

- (a) Ignoring questions of efficiency, is this scheme secure? State what computational problems need to be intractable for an adversary who knows  $g$ ,  $N$  and  $g^{x \cdot m}$  to not be able to discover  $m$ . The adversary does not know  $x$ .

**Solution:** [5 points] Very similar assumptions to Diffie-Hellman key exchange.

1. Discrete Log Problem is hard: Can't compute discrete log to find  $x \cdot m$  from  $g^{x \cdot m}$  and  $x$  from  $g^x$ .
2. Diffie-Hellman-like problem is hard: Seeing  $g^x$  and  $g^{x \cdot m}$  the adversary shouldn't be able to discover  $m$ .

- (b) Explain what Alice has to do to discover  $m$ . Indicate whether you think this scheme is efficient or not.

**Solution:** [5 points] Alice has  $x$ , and sees  $g^{x \cdot m}$ . She has to try to find  $m$  by raising  $g^x$  to some power so it equals  $g^{x \cdot m}$  which she sees. This is difficult to do efficiently.

**Problem 12. Duplicate customers [15 points] (2 parts)**

Two servers 1 and 2 each have a list of customers, and they would like to figure out which customers are in the intersection of their lists. In the following, the names of the customers are represented as integers in  $\{1, \dots, D\}$ . Each server has  $n$  distinct customers. Let  $S$  denote the set of customers in the intersection of the two lists.

- (a) Server 1 sends a message to server 2, after which server 2 needs to produce the set  $S$ , based on the message and server 2's list. Show how the servers can do this in such a way that server 1 only sends at most  $O(n \log |D|)$  bits to server 2.

**Solution:** [5 points]

Server 1 sends server 2 the names of his  $n$  customers. Each name needs only  $\log |D|$  bits to specify.

- (b) Server 1 sends a message to server 2, after which server 2 should be able to produce a set  $C$  based on the message and its own list, such that  $S \subseteq C$ , and with probability no less than  $\frac{3}{4}$ ,

$$|C| < |S| + 0.1n.$$

Show how the servers can do this so that the number of bits that server 1 sends to server 2 is as efficient as you can achieve.

Hint: You may use Markov's inequality for a non-negative random variable  $X$  and  $a > 0$ :

$$\Pr[X \geq a] \leq \frac{E[X]}{a}$$

**Solution:** [10 points]

Server 1 should send server 2 a Bloom filter of size  $O(n)$  with a false positive rate of  $\epsilon$ . Let  $X$  be the random variable that denotes the number false positives. Then,

$$E[X] \leq n\epsilon$$

We want  $X < .1n$ . Using the Markov's inequality:

$$\Pr[X \geq .1n] \leq \frac{E[X]}{.1n} \leq \frac{\epsilon}{.1}$$

For  $\epsilon \leq .025$ , we have that  $\frac{\epsilon}{.1} \leq \frac{1}{4}$ . Then, with probability at least  $\frac{3}{4}$ , the server 2 can produce the desired  $C$ .

## SCRATCH PAPER

## Final Exam Review

### True-false questions

- (1) **T F** The **best case** running time for INSERTION SORT to sort an  $n$  element array is  $O(n)$ .

**Answer:** True

- (2) **T F** By the master theorem, the solution to the recurrence  $T(n) = 3T(n/3) + \log n$  is  $T(n) = \Theta(n \log n)$ .

**Answer:** False

- (3) **T F** Given *any* binary tree, we can print its elements in sorted order in  $O(n)$  time by performing an inorder tree walk.

**Answer:** True. For each node in the tree we will be calling INORDER-TREE-WALK recursively exactly twice (once for the left subtree and once for the right subtree).

- (4) **T F** Computing the median of  $n$  elements takes  $\Omega(n \log n)$  time for any algorithm working in the comparison-based model.

**Answer:** False

- (5) **T F** Every binary search tree on  $n$  nodes has height  $O(\log n)$ .

**Answer:** False

- (6) **T F** Given a graph  $G = (V, E)$  with cost on edges and a set  $S \subseteq V$ , let  $(u, v)$  be an edge such that  $(u, v)$  is the minimum cost edge between any vertex in  $S$  and any vertex in  $V - S$ . Then, the minimum spanning tree of  $G$  must include the edge  $(u, v)$ . (You may assume the costs on all edges are distinct, if needed.)

**Answer:** True

- (7) **T F** Computing  $a^b$  takes exponential time in  $n$ , for  $n$ -bit integers  $a$  and  $b$ .

**Answer:** True (Output size is exponential in  $n$ ).

- (8) **T F** There exists a data structure to maintain a dynamic set with operations  $\text{Insert}(x, S)$ ,  $\text{Delete}(x, S)$ , and  $\text{Member?}(x, S)$  that has an expected running time of  $O(1)$  per operation.

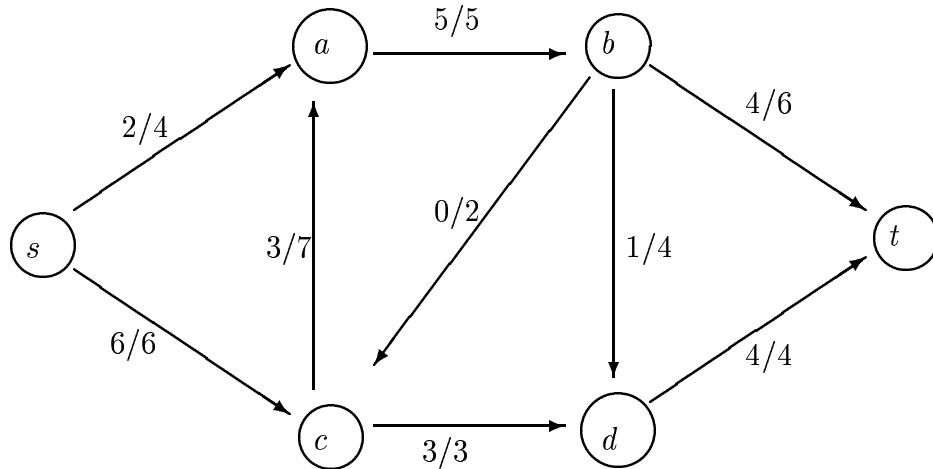
**Answer:** True. Use a hash table.

- (9) **T F** The total amortized cost of a sequence of  $n$  operations (i.e., the sum over all operations, of the amortized cost per operation) gives a lower bound on the total actual cost of the sequence.

**Answer:** False. This only gives an upper bound on the actual cost.

- (10) **T F** The figure below describes a flow assignment in a flow network. The notation  $a/b$  describes  $a$  units of flow in an edge of capacity  $b$ .

True or False: The following flow is a maximal flow.



**Answer:** True. The flow pushes 8 units and the cut  $\{s, a, c\}$  vs.  $\{b, d, t\}$  has capacity 8. The cut must be maximum by the Max-Flow Min-cut theorem.

- (11) **T F** Let  $G = (V, E)$  be a weighted graph and let  $M$  be a minimum spanning tree of  $G$ . The path in  $M$  between any pair of vertices  $v_1$  and  $v_2$  must be a shortest path in  $G$ .

**Answer:** False. Consider the graph in which  $V = \{a, b, c\}$  and the edges are  $(a, b)$ ,  $(b, c)$ ,  $(c, a)$ . The weight of the edges are 3, 3 and 4 respectively. The MST is clearly  $(a, b)$ ,  $(b, c)$ . However the shortest path between  $a$  and  $c$  is of cost 4 not 6 as seen from the MST.

- (12) **T F**  $n \lg n = O(n^2)$

**Answer:** True. Clearly  $n \lg n \leq n^2$ .

- (13) **T F** Let  $P$  be a shortest path from some vertex  $s$  to some other vertex  $t$  in a graph. If the weight of each edge in the graph is increased by one,  $P$  remains a shortest path from  $s$  to  $t$ .

**Answer:** False Consider the graph  $G = (V, E)$  where  $V = \{a, b, c\}$  and  $E = \{(a, b), (b, c), (a, c)\}$ . Let the weight of the edges be  $w(a, b) = 1, w(b, c) = 1, w(a, c) = 2.5$ . The shortest path from  $a$  to  $c$  is  $a-b-c$ . However if the weights are increased by 1, the new shortest path is  $a-c$ .

- (14) **T F** Suppose we are given  $n$  intervals  $(l_i, u_i)$  for  $i = 1, \dots, n$  and we would like to find a set  $S$  of non-overlapping intervals maximizing  $\sum_{i \in S} w_i$ , where  $w_i$  represents the weight of interval  $(l_i, u_i)$ . Consider the following greedy algorithm. Select (in the set  $S$ ) the interval, say  $(l_i, u_i)$  of maximum weight  $w_i$ , remove all intervals that overlap with  $(l_i, u_i)$  and repeat. This algorithm always provides an optimum solution to this interval selection problem.

**Answer:** False. Consider the following 3 intervals  $(1,10), (2,5)$  and  $(6,9)$  with weights 10, 6 and 6. The above greedy strategy would choose the intervals  $\{(1, 10)\}$  which has total weight 10 while the optimal set of intervals is  $\{(2, 5), (6, 9)\}$  which has total weight 12.

- (15) **T F** Given a set of  $n$  elements, one can output in sorted order the  $k$  elements following the median in sorted order in time  $O(n + k \log k)$ .

**Answer:** True. Find the median and the partition about it to obtain all the elements greater than it. Now find the  $k$ th largest element in this set and partition about it and obtain all the elements less than it. Output the sorted list of the final set of elements. Clearly, this operation costs  $O(n + k \lg k)$  time.

- (16) **T F** Consider a graph  $G = (V, E)$  with a weight  $w_e > 0$  defined for every edge  $e \in E$ . If a spanning tree  $T$  minimizes  $\sum_{e \in T} w_e$  then it also minimizes  $\sum_{e \in E} w_e^2$ , and vice versa.

**Answer:** True. Only the relative order of the weights matter, the actual weights do not matter for the MST. And as if  $w(e_1) \leq w(e_2)$ , then  $(w(e_1))^2 \leq (w(e_2))^2$ , the result holds.

- (17) **T F** The breadth first search algorithm makes use of a stack.

**Answer:** False. BFS uses a queue.

- (18) **T F** In the worst case, merge sort runs in  $O(n^2)$  time.

**Answer:** True. Merge sort runs in  $O(n \lg n)$  time which is  $O(n^2)$ .

- (19) **T F** A heap can be constructed from an unordered array of numbers in linear worst-case time.

**Answer:** True. A heap can always be constructed in  $O(n)$  time using the BUILD-HEAP procedure.

- (20) **T F** No adversary can elicit the  $\Theta(n^2)$  worst-case running time of randomized quicksort.

**Answer:** True. No adversary has control over which random numbers the algorithm will use, and no adversary can determine which random numbers the algorithm will use. Therefore, although it is true that RANDOMIZED-QUICKSORT runs in  $\Theta(n^2)$  time in the worst case, no adversary can force this behavior.

- (21) **T F** Radix sort requires an “in place” auxiliary sort in order to work properly.

**Answer:** False. Radix sort needs a stable sorting algorithm.

- (22) **T F** A longest path in a dag  $G = (V, E)$  can be found in  $O(V + E)$  time.

**Answer:** True

- (23) **T F** The Bellman-Ford algorithm is not suitable if the input graph has negative-weight edges.

**Answer:** False. Bellman Ford is used when there are negative weight edges, it's Dijkstra's algorithm that cannot be used.

- (24) **T F** Memoization is the basis for a top-down alternative to the usual bottom-up version of dynamic programming.

**Answer:** True

- (25) **T F** Given a weighted, directed graph  $G = (V, E)$  with no negative-weight cycles, the shortest path between every pair of vertices  $u, v \in V$  can be determined in  $O(V^3)$  worst-case time.

**Answer:** True. Floyd-Warshall's algorithm performs exactly this in  $O(V^3)$  time.

- (26) **T F** For hashing an item into a hash table in which collisions are resolved by chaining, the worst-case time is proportional to the load factor of the table.

**Answer:** False. Even with a low load factor (say  $\frac{1}{2}$ ) in the worst case you can get all  $n$  elements to hash to the same slot

- (27) **T F** A red-black tree on 128 keys must have at least 1 red node.

**Answer:** True

- (28) **T F** The move-to-front heuristic for self-organizing lists runs no more than a constant factor slower than any other reorganization strategy.

**Answer:** True. In recitation we've seen that it's *4-competitive*, which means that the best algorithm that can exist ("God's algorithm") could only do better by a factor of 4.

- (29) **T F** Depth-first search of a graph is asymptotically faster than breadth-first search.

**Answer:** False. They both run in time  $\Theta(V + E)$ .

- (30) **T F** Dijkstra's algorithm is an example of a greedy algorithm. **Answer:** True

- (31) **T F** Fibonacci heaps can be used to make Dijkstra's algorithm run in  $O(E + V \lg V)$  time on a graph  $G = (V, E)$ .

**Answer:** True

- (32) **T F** The Floyd-Warshall algorithm solves the all-pairs shortest-paths problem using dynamic programming.

**Answer:** True

- (33) **T F** A maximum matching in a bipartite graph can be found using a maximum-flow algorithm.

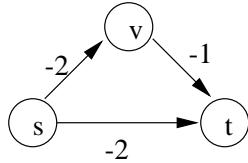
**Answer:** True. We've seen this in recitation.

- (34) **T F** For any directed acyclic graph, there is only one topological ordering of the vertices.

**Answer:** False. Consider a graph with two vertices and no edges. Either order is a topological ordering.

- (35) **T F** If some of the edge weights in a graph are negative, the shortest path from  $s$  to  $t$  can be obtained using Dijkstra's algorithm by first adding a large constant  $C$  to each edge weight, where  $C$  is chosen large enough that every resulting edge weight will be nonnegative.

**Answer:** False. Consider the following figure. The shortest path from  $s$  to  $t$  goes through  $v$ . But if we add 2 to each edge, so that we make the costs nonnegative, the edge from  $s$  to  $t$  becomes a shorter path.



- (36) **T F** If all edge capacities in a graph are integer multiples of 5 then the maximum flow value is a multiple of 5.

**Answer:** True. Consider the minimum cut. It is made up of edges with capacities that are multiples of 5, so the capacity of the cut (sum of capacities of edges in the cut) must be a multiple of 5. By the maxflow-mincut theorem, the maximum flow has the same value.

- (37) **T F** For any graph  $G$  with edge capacities and vertices  $s$  and  $t$ , there always exists an edge such that increasing the capacity on that edge will increase the maximum flow from  $s$  to  $t$  in  $G$ . (Assume that there is at least one path in the graph from  $s$  to  $t$ .)

**Answer:** False. There may be more than one minimum cut. For example, consider the following graph. There is no single edge you can increase to increase the flow.



- (38) **T F** Heapsort, quicksort, and mergesort are all asymptotically optimal, stable comparison-based sort algorithms.

**Answer:** False

- (39) **T F** If each operation on a data structure runs in  $O(1)$  amortized time, then  $n$  consecutive operations run in  $O(n)$  time in the worst case.

**Answer:** True. Recall that amortized analysis does not make any assumptions about the possible distribution of inputs (that is, it doesn't take into account what happens "on average")

- (40) **T F** A graph algorithm with  $\Theta(E \log V)$  running time is asymptotically better than an algorithm with a  $\Theta(E \log E)$  running time for a connected, undirected graph  $G(V, E)$ .

**Answer:** False. Compare  $\log V$  and  $\log E$ . We know that  $E \leq V^2/2$  (one edge for each pair of vertices), which means  $\log E \leq 2 \log V$ . That factor of 2 won't do us any good asymptotically

- (41) **T F** In  $O(V + E)$  time a matching in a bipartite graph  $G = (V, E)$  can be tested to determine if it is maximum.

**Answer:** True: use DFS to determine there is an augmenting path in the corresponding flow graph, which can be done in linear time

- (42) **T F**  $n$  integers each of value less than  $n^{100}$  can be sorted in linear time.

**Answer:** True. Use radix sort, for example

- (43) **T F** For any network and any maximal flow on this network there always exists an edge such that increasing the capacity on that edge will increase the network's maximal flow.

**Answer:** False. There may be *two* min-cuts, with the edge in question increasing the capacity of only one of them. The other one will remain as it is, preventing the max-flow from increasing any further

- (44) **T F** If the depth-first search of a graph  $G$  yields no back edges, then the graph  $G$  is acyclic.

**Answer:** True

- (45) **T F** Insertion in a binary search tree is “commutative”. That is, inserting  $x$  and then  $y$  into a binary search tree leaves the same tree as inserting  $y$  and then  $x$ .

**Answer:** False. Consider inserting  $x, y$  in an empty binary search tree to see why.

- (46) **T F** A heap with  $n$  elements can be converted into a binary search tree in  $O(n)$  time.

**Answer:** False. Building a heap takes  $O(n)$  time. If we could convert that heap into a binary search tree in  $O(n)$  time, then we could perform an *inorder tree walk* (which we know takes  $O(n)$  time) and thus sort  $n$  elements in a total of  $O(n)$  time. This contradicts the fact that any comparison-based sorting algorithm takes  $\Omega(n \log n)$  time to sort  $n$  elements

## Practice Final Exam—From Fall 2004

### Problem 1. Recurrences (4 parts) [8 points]

For each of the recurrences below, do the following:

- Give the solution using  $\Theta$ -notation. You need not provide a proof or other justification.
- Name a recursive algorithm we've seen during the term whose running time is described by that recurrence.

(a)  $T(n) = T(n/2) + \Theta(1)$

**Solution:**  $\Theta(\log n)$ . Binary search.

(b)  $T(n) = 2T(n/2) + \Theta(n)$

**Solution:**  $\Theta(n \log n)$ . MERGE-SORT.

(c)  $T(n) = T(n/5) + T(7n/10) + \Theta(n)$

**Solution:**  $\Theta(n)$ . SELECT

(d)  $T(n) = 7T(n/2) + \Theta(n^2)$

**Solution:**  $\Theta(n^{\lg 7})$ . Strassen's matrix-multiplication algorithm.

### Problem 2. Design Techniques and Data Structures (5 parts) [10 points]

For each of the following design techniques and data structures, name an algorithm covered this term that uses it.

(a) Divide and conquer:

**Solution:** MERGE-SORT uses divide and conquer. It divides the problem into two problems of half the size (the left and right halves of the array), conquers the subproblems by running MERGE-SORT recursively, and combines the results by merging the subarrays together.

(b) Dynamic programming:

**Solution:** The typesetting problem used dynamic programming.

(c) Greedy:

**Solution:** Prim's algorithm for minimum spanning tree is a greedy algorithm.

(d) Binary search tree:

**Solution:** The dynamic maximum-prefix problem from problem set 4 used an augmented red-black tree.

(e) FIFO queue:

**Solution:** Breadth-first search uses a FIFO queue.

**Problem 3. True or False, and Justify** (12 parts) [84 points]

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. The more content you provide in your justification, the higher your grade, but be brief. Your justification is worth more points than your true-or-false designation.

- (a) **T F** If  $f(n)$  is asymptotically positive, then  $f(n) + o(f(n)) = \Theta(f(n))$ .

**Solution:** True.

Clearly,  $f(n) + o(f(n))$  is  $\Omega(f(n))$ , so let us prove that  $f(n) + o(f(n)) = O(f(n))$ . Let  $g(n) \in o(f(n))$ . Then for any  $c > 0$ , there exists  $n_0$  such that  $g(n) \leq c(f(n))$  for all  $n \geq n_0$ . Hence,  $f(n) + g(n) \leq (c+1)f(n)$  for all  $n \geq n_0$ , which means that  $f(n) + g(n) = O(f(n))$ .

- (b) **T F** An adversary can provide an input to randomized quicksort that will elicit<sup>1</sup> its  $\Theta(n^2)$  worst-case running time.

**Solution:** False. The worst-case behavior of quicksort happens due to bad coin flips; it has nothing to do with the adversary's choice of inputs.

- (c) **T F** Any comparison sort of 5 elements requires at least 7 comparisons in the worst case.

**Solution:** True. The decision tree for sorting 5 elements has  $5! = 120$  leaves. Any comparison sort can only distinguish at most  $2^6 = 64$  different elements with fewer than 7 comparisons and is therefore unable to sort 7 elements correctly all the time.

- (d) **T F** Consider a sequence of  $n$  operations on an initially empty dynamic set. Suppose that the amortized running time of each operation is  $O(1)$ . Then, the  $n$  operations take  $O(n)$  time in the worst case.

**Solution:** True.

- (e) **T F** A good heuristic for a simple dynamic set implemented as a linked list is to move an item to the front of the list whenever it is accessed.

**Solution:** True. We showed this heuristic to be 2-competitive.

- (f) **T F** Prim's algorithm, Dijkstra's algorithm, and the Bellman-Ford algorithm are all examples of greedy algorithms.

**Solution:** False. Not BF.

- (g) **T F** For the all-pairs shortest-paths problem on an edge-weighted graph  $G = (V, E)$  with  $E = \Theta(V^{3/2})$ , the Floyd-Warshall algorithm is asymptotically at least as fast as Johnson's algorithm.

**Solution:** False. Floyd-Warshall runs in  $O(V^3)$  time. Johnson's runs in  $O(V^2 \lg V + VE)$ . If  $E = \Theta(V^{3/2})$ , then Johnson's runs in  $O(V^2 \lg V + V^{5/2}) = o(V^3)$ .

---

<sup>1</sup>**elicit** transitive verb **1** : to draw forth or bring out (something latent or potential) *(hypnotism elicited his hidden fears)* **2** : to call forth or draw out (as information or a response) *(her performance elicited wild applause)* — *Merriam-Webster's Collegiate Dictionary*, Tenth Edition, 1993.

- (h) T F** Suppose that the constraint graph  $G = (V, E)$  of a linear-programming system of difference constraints is acyclic. Then, a solution always exists and can be found in  $O(V + E)$  time.

**Solution:** True. Dag shortest paths.

- (i) T F** Let  $G = (V, E)$  be an edge-weighted digraph, where edge weights are given by the function  $w : E \rightarrow \mathbb{R}$ . Define another edge-weight function  $w' : E \rightarrow \mathbb{R}$  by

$$w'(u, v) = w(u, v) - \text{out-degree}(u) + \text{out-degree}(v).$$

Then,  $G$  contains a negative-weight cycle under  $w$  if and only if  $G$  contains a negative-weight cycle under  $w'$ .

**Solution:** True. The total weight on a cycle  $C$  under  $w'$  is

$$W_C = \sum_{(u,v) \in C} (w(u, v) - \text{out-degree}(u) + \text{out-degree}(v)).$$

Since this summation telescopes, we have  $W_C = \sum_{(u,v) \in C} w(u, v)$ .

- (j) T F** Suppose that all edge capacities in a flow network are integer multiples of 3, but that the value of a flow between the source  $s$  and the sink  $t$  is not a multiple of 3. Then, an augmenting path from  $s$  to  $t$  exists.

**Solution:** True. Consider the minimum cut. It is made up of edges with capacities that are multiples of 3, so the capacity of the cut (sum of capacities of edges in the cut) must be a multiple of 3. By the Maxflow-Mincut theorem, the maximum flow has the same value.

- (k) T F** Given a maximum flow  $f$  on a flow graph  $G = (V, E)$  with source  $s$  and sink  $t$ , a minimum cut separating  $s$  from  $t$  can be found in  $O(V + E)$  time.

**Solution:** True. BFS or DFS.

- (l) T F** The Karp-Rabin algorithm always reports a match of a pattern in a text string if one exists.

**Solution:** True. It may have false positives, but no false negatives.

#### Problem 4. Set Equality [15 points]

Let  $S$  and  $T$  be two sets of numbers represented as unordered lists of distinct numbers. All you have are pointers to the heads of the lists, but you do not know the list lengths. Describe an  $O(\min\{|S|, |T|\})$ -expected-time algorithm to determine whether  $S = T$ . You may assume that any operation on one or two numbers can be performed in constant time.

**Solution:** First, check that both sets are the same size. If they are not, then they cannot be equal. To do this check in  $O(\min\{|S|, |T|\})$  time, just iterate over both lists in parallel. That is, advance one step in  $S$  and one step in  $T$ . If both lists end, the lengths are the same. If one list ends before the other, they have different lengths.

If both lists are the same size, then we want to check whether the elements are the same. We create a hash table of size  $\Theta(|S|)$  using universal hashing with chaining. We iterate over  $S$ , adding each element from  $S$  to the hash table. Then we iterate over  $T$ . For each element  $x \in T$ , we check whether  $x$  belongs to the hash table (that is, whether it is also in  $S$ ). If not, then we return that the sets are not identical. If so, then continue iterating over  $T$ .

Any sequence of  $|S| = |T|$  INSERT and SEARCH operations in the table take  $O(|S|)$  time in expectation (see CLRS p.234), so the total runtime is  $O(\min\{|S|, |T|\})$  in expectation.

### Problem 5. Minimum Spanning Tree [15 points]

Let  $G = (V, E)$  be a connected undirected graph with edge-weight function  $w : E \rightarrow \mathbb{R}$ . Consider the following algorithm:

- 1 **while** there exists a cycle  $C$  in  $G$
- 2     **do** find an edge  $e \in C$  such that  $w(e) = \max_{e' \in C} \{w(e')\}$
- 3          $G \leftarrow (V, E - \{e\})$

Prove that when this algorithm terminates,  $G$  forms a minimum spanning tree of the original input graph.

**Solution:** This algorithm repeatedly removes a maximum-weight edge from a cycle.

First, let's argue that the graph  $G$  forms a spanning tree. That is, that  $G$  is acyclic and connects all the vertices. The first part is obvious—since we continue removing edges until there are no cycles, the resulting  $G$  must be acyclic. Since the only edges removed are in cycles, removing an edge never disconnects any two vertices. Thus, the output  $G$  remains connected as well, and forms a spanning tree.

Now we argue that the spanning tree  $G$  is minimum. If it is not minimum, then there is some edge  $(u, v) \in E$  that can be removed to partition the graph into two connected components such that  $(u, v)$  is not a minimum-weight edge crossing the cut. Suppose that  $(x, y)$  is the minimum-weight edge crossing this cut. The algorithm removed  $(x, y)$ , so there must have been some cycle such that  $(x, y)$  was a maximum-weight edge on the cycle. Since any cycle including  $(x, y)$  crosses the cut in two places, the other edge that was left behind must have had weight at most  $w(x, y)$ . Thus, by induction, we must have  $w(u, v) \leq w(x, y)$ .

### Problem 6. Woody the Woodcutter (3 parts) [15 points]

Given a log of wood of length  $k$ , Woody the woodcutter will cut it once, in any place you choose, for the price of  $k$  dollars. Suppose you have a log of length  $L$ , marked to be cut in  $n$  different locations labeled  $1, 2, \dots, n$ . For simplicity, let indices 0 and  $n + 1$  denote the left and right endpoints of the original log of length  $L$ . Let the distance of mark  $i$  from the left end of the log be  $d_i$ , and assume that  $0 = d_0 < d_1 < d_2 < \dots < d_n < d_{n+1} = L$ . The **wood-cutting problem** is the problem of determining the sequence of cuts to the log that will (1) cut the log at all the marked places, and (2) minimize your total payment to Woody.

- (a) Give a small example illustrating that two different sequences of cuts to the same marked log can result in two different costs.

**Solution:** Suppose that we have a log of length 4, and we want to cut at  $d_1 = 1$ ,  $d_2 = 2$ , and  $d_3 = 3$ . Then cutting  $d_1, d_2, d_3$  results in a total cost of  $4 + 3 + 2 = 9$ . Cutting at  $d_2, d_1, d_3$  results in a total cost of  $4 + 2 + 2 = 8$ .

Let  $c(i, j)$  be the minimum cost of cutting a log with left endpoint  $i$  and right endpoint  $j$  at all its marked locations.

- (b) Complete the following recursive definition, and *briefly* justify your answer:

**Solution:**

$$c(i, j) = \min_{i < k < j} \{c(i, k) + c(k, j) + (d_j - d_i)\} .$$

First off, the length of the log segment is  $d_j - d_i$ , so Woody charges  $\$d_j - d_i$  for the cut regardless of the location.

As for the rest of the answer, the problem exhibits optimal substructure. If the optimal answer involves cutting the log at position  $k$ , then we use the optimal answers for segment  $i$  to  $k$  and segment  $k$  to  $j$  to perform the rest of the cuts (use a cut & paste argument for completeness). We just try all possible locations  $k$  for the next cut and choose the best one.

- (c) Using part (b), describe an efficient algorithm to solve the wood-cutting problem. What is the running time of your algorithm?

**Solution:** Build a table  $C$  of size  $(n+1) \times (n+1)$  to hold the values  $C[i][j] = c(i, j)$ . We initialize the table first by setting  $C[i][i] \leftarrow 0$  and  $C[i][i+1] \leftarrow 0$ . Then we just fill along the diagonals to fill in the  $C[i][j]$  for segments whose endpoints are two positions away, then three, etc. We are filling the table in in order of size of log segments.

Pseudocode for this algorithm looks like

```

1  for  $i \leftarrow 0$  to  $n$ 
2      do  $C[i][i] \leftarrow 0$ 
3       $C[i][i+1] \leftarrow 0$ 
4  for  $s \leftarrow 2$  to  $n$ 
5      do for  $i \leftarrow 0$  to  $n$ 
6          do  $j \leftarrow i + s$ 
7           $C[i][j] \leftarrow \min_{i < k < j} \{c(i, k) + c(k, j) + d_j - d_i\}$ 

```

The important part is that when we fill compute  $C[i][j]$ , all the necessary values to max over have already been computed. This is because we max over smaller-size log segments, and we compute entries in order of log-segment size. The answer is found at  $C[0][n]$ .

This algorithm takes  $O(n^3)$  time.

**Problem 7. Edge Covering [15 points]**

Given an undirected graph  $G = (V, E)$  with no isolated vertices (vertices with degree 0), an **edge cover** is a set  $C \subseteq E$  of edges such that for all  $u \in V$ , there exists a  $v \in V$  such that  $(u, v) \in C$ . The **edge-covering problem** is the problem of finding an edge cover of minimum cardinality.

Describe an  $O(1)$ -approximation algorithm for the edge-covering problem. Analyze your algorithm's running time and its ratio bound (by what factor worse than the optimal is the approximation your algorithm produces?).

**Solution:** We assume that the graph is in the adjacency-list representation. We keep auxiliary information associated with each vertex to check whether a vertex is already covered. In the beginning, no vertex is covered. Iterate over all vertices. When considering vertex  $u$ , check if it is already covered. If so, ignore it and move to the next vertex. If not, let  $v$  be the first element of  $\text{Adj}[u]$ . Add  $(u, v)$  to  $C$  and mark both  $u$  and  $v$  as being covered. Then continue on to the next vertex.

Each vertex is considered once and marked as covered once. Total amount of work spent on a vertex is  $O(1)$ , so the total time of the algorithm is  $O(V)$ .

This algorithm finds a 2-approximation. The optimal covering must have cardinality at least  $|V| / 2$ , because each edge covers at most 2 new vertices. Our covering has at most  $|V| - 1$  edges, which is off by less than a factor of 2.

## Practice Final Exam

### Problem 1. Recurrences (4 parts)

For each of the recurrences below, do the following:

- Give the solution using  $\Theta$ -notation. You need not provide a proof or other justification.
- Name a recursive algorithm we've seen during the term whose running time is described by that recurrence.

(a)  $T(n) = T(n/2) + \Theta(1)$

**Solution:**  $\Theta(\log n)$ . Binary search.

(b)  $T(n) = 2T(n/2) + \Theta(n)$

**Solution:**  $\Theta(n \log n)$ . MERGE-SORT.

(c)  $T(n) = T(n/5) + T(7n/10) + \Theta(n)$

**Solution:**  $\Theta(n)$ . SELECT

(d)  $T(n) = 7T(n/2) + \Theta(n^2)$

**Solution:**  $\Theta(n^{\lg 7})$ . Strassen's matrix-multiplication algorithm.

### Problem 2. Algorithms and running times (5 parts)

Match each algorithm below with the tightest asymptotic upper bound for its worst-case running time by inserting one of the letters A, B, . . . , into the corresponding box. **Some running times may be used multiple times or not at all.**

You need not justify your answers.

Dijkstra's implemented using Fibonacci heap

A:  $O(E + V \lg V)$

Dijkstra's implemented using binary heap

B:  $O((V + E) \lg V)$

Building a BST

C:  $O(n)$

BUILD-HEAP

D:  $O(n \lg n)$

Viterbi decoding in HMM with  $n^{1/2}$  states and  $n$  emissions

E:  $O(n^3)$

F:  $O(n^2)$

### Problem 3. Design Techniques and Data Structures (5 parts)

For each of the following design techniques and data structures, name an algorithm covered this term that uses it.

(a) Divide and conquer:

**Solution:** MERGE-SORT uses divide and conquer. It divides the problem into two problems of half the size (the left and right halves of the array), conquers the subproblems by running MERGE-SORT recursively, and combines the results by merging the subarrays together.

(b) Dynamic programming:

**Solution:** The typesetting problem used dynamic programming.

(c) Greedy:

**Solution:** Prim's algorithm for minimum spanning tree is a greedy algorithm.

(d) Binary search tree:

**Solution:** The dynamic maximum-prefix problem from problem set 4 used an augmented red-black tree.

(e) FIFO queue:

**Solution:** Breadth-first search uses a FIFO queue.

**Problem 4. True or False, and Justify** (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. The more content you provide in your justification, the higher your grade, but be brief. Your justification is worth more points than your true-or-false designation.

- (a) **T F** If  $f(n)$  is asymptotically positive, then  $f(n) + o(f(n)) = \Theta(f(n))$ .

**Solution:** True.

Clearly,  $f(n) + o(f(n))$  is  $\Omega(f(n))$ , so let us prove that  $f(n) + o(f(n)) = O(f(n))$ . Let  $g(n) \in o(f(n))$ . Then for any  $c > 0$ , there exists  $n_0$  such that  $g(n) \leq c(f(n))$  for all  $n \geq n_0$ . Hence,  $f(n) + g(n) \leq (c+1)f(n)$  for all  $n \geq n_0$ , which means that  $f(n) + g(n) = O(f(n))$ .

- (b) **T F** An adversary can provide an input to randomized quicksort that will elicit<sup>1</sup> its  $\Theta(n^2)$  worst-case running time.

**Solution:** False. The worst-case behavior of quicksort happens due to bad coin flips; it has nothing to do with the adversary's choice of inputs.

- (c) **T F** Any comparison sort of 5 elements requires at least 7 comparisons in the worst case.

**Solution:** True. The decision tree for sorting 5 elements has  $5! = 120$  leaves. Any comparison sort can only distinguish at most  $2^6 = 64$  different elements with fewer than 7 comparisons and is therefore unable to sort 7 elements correctly all the time.

- (d) **T F** Consider a sequence of  $n$  operations on an initially empty dynamic set. Suppose that the amortized running time of each operation is  $O(1)$ . Then, the  $n$  operations take  $O(n)$  time in the worst case.

**Solution:** True.

- (e) **T F** In an HMM, let  $x_j$  be the emission observed at time  $j$ . Given a series of observed emissions  $x_1, x_2, \dots, x_n$ , the most likely state at time  $i$  is independent of emissions  $x_{i+1}, x_{i+2}, \dots, x_n$ .

**Solution:** False

- (f) **T F** Prim's algorithm, Dijkstra's algorithm, and the Bellman-Ford algorithm are all examples of greedy algorithms.

**Solution:** False. Not BF.

- (g) **T F** For the all-pairs shortest-paths problem on an edge-weighted graph  $G = (V, E)$  with  $E = \Theta(V^{3/2})$ , the Floyd-Warshall algorithm is asymptotically at least as fast as Johnson's algorithm.

---

<sup>1</sup>**elicit** transitive verb 1 : to draw forth or bring out (something latent or potential) *⟨hypnotism elicited his hidden fears⟩* 2 : to call forth or draw out (as information or a response) *⟨her performance elicited wild applause⟩* — *Merriam-Webster's Collegiate Dictionary*, Tenth Edition, 1993.

**Solution:** False. Floyd-Warshall runs in  $O(V^3)$  time. Johnson's runs in  $O(V^2 \lg V + VE)$ . If  $E = \Theta(V^{3/2})$ , then Johnson's runs in  $O(V^2 \lg V + V^{5/2}) = o(V^3)$ .

- (h) **T F** Suppose that the constraint graph  $G = (V, E)$  of a linear-programming system of difference constraints is acyclic. Then, a solution always exists and can be found in  $O(V + E)$  time.

**Solution:** True. Dag shortest paths.

- (i) **T F** Let  $G = (V, E)$  be an edge-weighted digraph, where edge weights are given by the function  $w : E \rightarrow \mathbb{R}$ . Define another edge-weight function  $w' : E \rightarrow \mathbb{R}$  by

$$w'(u, v) = w(u, v) - \text{out-degree}(u) + \text{out-degree}(v).$$

Then,  $G$  contains a negative-weight cycle under  $w$  if and only if  $G$  contains a negative-weight cycle under  $w'$ .

**Solution:** True. The total weight on a cycle  $C$  under  $w'$  is

$$W_C = \sum_{(u,v) \in C} (w(u, v) - \text{out-degree}(u) + \text{out-degree}(v)).$$

Since this summation telescopes, we have  $W_C = \sum_{(u,v) \in C} w(u, v)$ .

- (j) **T F** Suppose that all edge capacities in a flow network are integer multiples of 3, but that the value of a flow between the source  $s$  and the sink  $t$  is not a multiple of 3. Then, an augmenting path from  $s$  to  $t$  exists.

**Solution:** True. Consider the minimum cut. It is made up of edges with capacities that are multiples of 3, so the capacity of the cut (sum of capacities of edges in the cut) must be a multiple of 3. By the Maxflow-Mincut theorem, the maximum flow has the same value.

- (k) **T F** Given a maximum flow  $f$  on a flow graph  $G = (V, E)$  with source  $s$  and sink  $t$ , a minimum cut separating  $s$  from  $t$  can be found in  $O(V + E)$  time.

**Solution:** True. BFS or DFS.

- (l) **T F** The Karp-Rabin algorithm always reports a match of a pattern in a text string if one exists.

**Solution:** True. It may have false positives, but no false negatives.

### Problem 5. Set Equality

Let  $S$  and  $T$  be two sets of numbers represented as unordered lists of distinct numbers. All you have are pointers to the heads of the lists, but you do not know the list lengths. Describe an  $O(\min\{|S|, |T|\})$ -expected-time algorithm to determine whether  $S = T$ . You may assume that any operation on one or two numbers can be performed in constant time.

**Solution:** First, check that both sets are the same size. If they are not, then they cannot be equal. To do this check in  $O(\min\{|S|, |T|\})$  time, just iterate over both lists in parallel. That is, advance

one step in  $S$  and one step in  $T$ . If both lists end, the lengths are the same. If one list ends before the other, they have different lengths.

If both lists are the same size, then we want to check whether the elements are the same. We create a hash table of size  $\Theta(|S|)$  using universal hashing with chaining. We iterate over  $S$ , adding each element from  $S$  to the hash table. Then we iterate over  $T$ . For each element  $x \in T$ , we check whether  $x$  belongs to the hash table (that is, whether it is also in  $S$ ). If not, then we return that the sets are not identical. If so, then continue iterating over  $T$ .

Any sequence of  $|S| = |T|$  INSERT and SEARCH operations in the table take  $O(|S|)$  time in expectation (see CLRS p.234), so the total runtime is  $O(\min\{|S|, |T|\})$  in expectation.

### Problem 6. Minimum Spanning Tree

Let  $G = (V, E)$  be a connected undirected graph with edge-weight function  $w : E \rightarrow \mathbb{R}$ . Consider the following algorithm:

- 1 **while** there exists a cycle  $C$  in  $G$
- 2     **do** find an edge  $e \in C$  such that  $w(e) = \max_{e' \in C} \{w(e')\}$
- 3          $G \leftarrow (V, E - \{e\})$

Prove that when this algorithm terminates,  $G$  forms a minimum spanning tree of the original input graph.

**Solution:** This algorithm repeatedly removes a maximum-weight edge from a cycle.

First, let's argue that the graph  $G$  forms a spanning tree. That is, that  $G$  is acyclic and connects all the vertices. The first part is obvious—since we continue removing edges until there are no cycles, the resulting  $G$  must be acyclic. Since the only edges removed are in cycles, removing an edge never disconnects any two vertices. Thus, the output  $G$  remains connected as well, and forms a spanning tree.

Now we argue that the spanning tree  $G$  is minimum. If it is not minimum, then there is some edge  $(u, v) \in E$  that can be removed to partition the graph into two connected components such that  $(u, v)$  is not a minimum-weight edge crossing the cut. Suppose that  $(x, y)$  is the minimum-weight edge crossing this cut. The algorithm removed  $(x, y)$ , so there must have been some cycle such that  $(x, y)$  was a maximum-weight edge on the cycle. Since any cycle including  $(x, y)$  crosses the cut in two places, the other edge that was left behind must have had weight at most  $w(x, y)$ . Thus, by induction, we must have  $w(u, v) \leq w(x, y)$ .

### Problem 7. Woody the Woodcutter (3 parts)

Given a log of wood of length  $k$ , Woody the woodcutter will cut it once, in any place you choose, for the price of  $k$  dollars. Suppose you have a log of length  $L$ , marked to be cut in  $n$  different locations labeled  $1, 2, \dots, n$ . For simplicity, let indices 0 and  $n + 1$  denote the left and right endpoints of the original log of length  $L$ . Let the distance of mark  $i$  from the left end of the log be  $d_i$ , and assume that  $0 = d_0 < d_1 < d_2 < \dots < d_n < d_{n+1} = L$ . The **wood-cutting problem** is the

problem of determining the sequence of cuts to the log that will (1) cut the log at all the marked places, and (2) minimize your total payment to Woody.

- (a) Give a small example illustrating that two different sequences of cuts to the same marked log can result in two different costs.

**Solution:** Suppose that we have a log of length 4, and we want to cut at  $d_1 = 1$ ,  $d_2 = 2$ , and  $d_3 = 3$ . Then cutting  $d_1, d_2, d_3$  results in a total cost of  $4 + 3 + 2 = 9$ . Cutting at  $d_2, d_1, d_3$  results in a total cost of  $4 + 2 + 2 = 8$ .

Let  $c(i, j)$  be the minimum cost of cutting a log with left endpoint  $i$  and right endpoint  $j$  at all its marked locations.

- (b) Complete the following recursive definition, and *briefly* justify your answer:

**Solution:**

$$c(i, j) = \min_{i < k < j} \{c(i, k) + c(k, j) + (d_j - d_i)\} .$$

First off, the length of the log segment is  $d_j - d_i$ , so Woody charges  $\$d_j - d_i$  for the cut regardless of the location.

As for the rest of the answer, the problem exhibits optimal substructure. If the optimal answer involves cutting the log at position  $k$ , then we use the optimal answers for segment  $i$  to  $k$  and segment  $k$  to  $j$  to perform the rest of the cuts (use a cut & paste argument for completeness). We just try all possible locations  $k$  for the next cut and choose the best one.

- (c) Using part (b), describe an efficient algorithm to solve the wood-cutting problem. What is the running time of your algorithm?

**Solution:** Build a table  $C$  of size  $(n+1) \times (n+1)$  to hold the values  $C[i][j] = c(i, j)$ . We initialize the table first by setting  $C[i][i] \leftarrow 0$  and  $C[i][i+1] \leftarrow 0$ . Then we just fill along the diagonals to fill in the  $C[i][j]$  for segments whose endpoints are two positions away, then three, etc. We are filling the table in in order of size of log segments.

Pseudocode for this algorithm looks like

```

1  for  $i \leftarrow 0$  to  $n$ 
2      do  $C[i][i] \leftarrow 0$ 
3           $C[i][i+1] \leftarrow 0$ 
4  for  $s \leftarrow 2$  to  $n$ 
5      do for  $i \leftarrow 0$  to  $n$ 
6          do  $j \leftarrow i + s$ 
7               $C[i][j] \leftarrow \min_{i < k < j} \{c(i, k) + c(k, j) + d_j - d_i\}$ 

```

The important part is that when we fill compute  $C[i][j]$ , all the necessary values to max over have already been computed. This is because we max over smaller-size log

segments, and we compute entries in order of log-segment size. The answer is found at  $C[0][n]$ .

This algorithm takes  $O(n^3)$  time.

### Problem 8. Edge Covering

Given an undirected graph  $G = (V, E)$  with no isolated vertices (vertices with degree 0), an **edge cover** is a set  $C \subseteq E$  of edges such that for all  $u \in V$ , there exists a  $v \in V$  such that  $(u, v) \in C$ . The **edge-covering problem** is the problem of finding an edge cover of minimum cardinality.

Describe an  $O(1)$ -approximation algorithm for the edge-covering problem. Analyze your algorithm's running time and its ratio bound (by what factor worse than the optimal is the approximation your algorithm produces?).

**Solution:** We assume that the graph is in the adjacency-list representation. We keep auxiliary information associated with each vertex to check whether a vertex is already covered. In the beginning, no vertex is covered. Iterate over all vertices. When considering vertex  $u$ , check if it is already covered. If so, ignore it and move to the next vertex. If not, let  $v$  be the first element of  $\text{Adj}[u]$ . Add  $(u, v)$  to  $C$  and mark both  $u$  and  $v$  as being covered. Then continue on to the next vertex.

Each vertex is considered once and marked as covered once. Total amount of work spent on a vertex is  $O(1)$ , so the total time of the algorithm is  $O(V)$ .

This algorithm finds a 2-approximation. The optimal covering must have cardinality at least  $|V|/2$ , because each edge covers at most 2 new vertices. Our covering has at most  $|V| - 1$  edges, which is off by less than a factor of 2.

### Problem 9. Radix Sort

We have seen that COUNTING-SORT requires  $O(n+k)$  time to sort  $n$  numbers in the range  $1, \dots, k$ , while RADIX-SORT requires  $O(nd)$  time to sort  $n$  number of  $d$  digits each. By a judicious combination of these algorithms, we can get a linear running time, as in COUNTING-SORT, when operating on elements from a wider range, as in RADIX-SORT.

- (a) Given a number  $x$ , show how to get the  $i$ th digit of its base- $r$  representation in  $O(1)$  time. (The 1st digit is the least-significant one.)

**Solution:** The  $i$ th digit is  $\lfloor x/r^{i-1} \rfloor \bmod r$ . To see this, note that dividing by  $r^{i-1}$  and taking the floor is just “shifting right” the base- $r$  representation of  $x$  by  $i-1$  digits, and taking that value mod  $r$  is just keeping the least significant remaining digit. If we can compute  $r^{i-1}$  in  $O(1)$  time, then an algorithm for computing the  $i$ th digit is straightforward. Alterately, we can get the digits in sequence by computing  $1, r, r^2, r^3, \dots$ , which yields  $O(1)$  time per digit.

- (b) What is the running time of RADIX-SORT on an array of  $n$  numbers in the range  $0, \dots, n^5 - 1$ , when using base-10 representations?

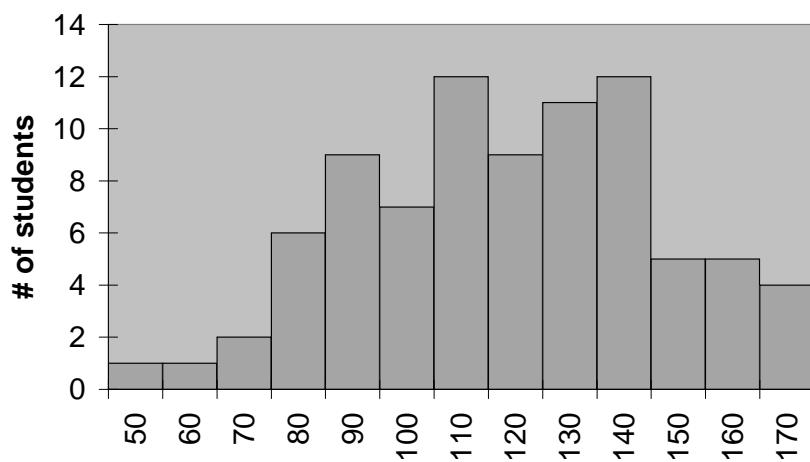
**Solution:** Using base 10, the numbers have  $d = \log n^5 = 5 \log n$  digits. Each COUNTING-SORT call takes  $\Theta(n+10) = \Theta(n)$  time, so the running time of RADIX-SORT is  $\Theta(nd) = \Theta(n \log n)$ .

- (c) Give an algorithm to sort an array of  $n$  numbers in the range  $0, \dots, n^5 - 1$  in only  $O(n)$  time. Does your technique extend to wider ranges of elements? Explain.

**Solution:** If we use base  $n$ , the numbers have  $d = \log_n n^5 = 5$  digits, and by part (a) we can compute each digit in  $O(1)$  time. Therefore each COUNTING-SORT call takes  $\Theta(n + n) = \Theta(n)$  time, so the running time of RADIX-SORT is  $\Theta(nd) = \Theta(n)$ . This method extends to numbers in the range  $0, \dots, n^c - 1$  for any constant  $c$ .

## Final Exam Solutions

**Final Exam Score Distribution**



**Problem 1. Recurrences [15 points] (3 parts)**

Give a tight asymptotic upper bound ( $O$  notation) on the solution to each of the following recurrences. You need not justify your answers.

(a)  $T(n) = 2T(n/8) + \sqrt[3]{n}.$

**Solution:**  $\Theta(n^{1/3} \lg n)$  by Case 2 of the Master Method.

(b)  $T(n) = T(n/3) + T(n/4) + 5n$

**Solution:**  $\Theta(n).$

(c)  $T(n) = \begin{cases} 8T(n/2) + \Theta(1) & \text{if } n^2 > M, \\ M & \text{if } n^2 \leq M; \end{cases}$

where  $M$  is a variable independent from  $n$ .

**Solution:**  $\Theta(n^3/\sqrt{M}).$  The recursion tree has approximately  $\lg n - \lg \sqrt{M} = \lg(n/\sqrt{M})$  levels. In the tree, every internal node has 8 children, each with cost  $O(1).$  At the bottom of the tree, there are approximately  $8^{\lg(n/\sqrt{M})} = (n/\sqrt{M})^3$  leaves. Since each leaf costs  $M,$  and the total cost is dominated by the leaves, the solution is  $\Theta(M(n/\sqrt{M})^3) = \Theta(n^3/\sqrt{M}).$

**Problem 2. Algorithms and running times [9 points]**

Match each algorithm below with the tightest asymptotic upper bound for its worst-case running time by inserting one of the letters A, B, . . . , I into the corresponding box. For sorting algorithms,  $n$  is the number of input elements. For matrix algorithms, the input matrix has size  $n \times n$ . For graph algorithms, the number of vertices is  $n$ , and the number of edges is  $\Theta(n)$ .

You need not justify your answers. Some running times may be used multiple times or not at all. Because points will be deducted for wrong answers, do not guess unless you are reasonably sure.

Insertion sort

A:  $O(\lg n)$ 


Heapsort

B:  $O(n)$ 


BUILD-HEAP

C:  $O(n \lg n)$ 


Strassen's

D:  $O(n^2)$ 


Bellman-Ford

E:  $O(n^2 \lg n)$ 


Depth-first search

F:  $O(n^{2.5})$ 


Floyd-Warshall

G:  $O(n^{\lg 7})$ 


Johnson's

H:  $O(n^3)$ 


Prim's

I:  $O(n^3 \lg n)$ 

**Solution:** From top to bottom: D, C, B, G, D, B, H, E, C.

**Problem 3. Substitution method [10 points]**

Use the substitution method to prove a tight asymptotic lower bound ( $\Omega$ -notation) on the solution to the recurrence

$$T(n) = 4T(n/2) + n^2.$$

**Solution:** By the master method, we know  $T(n) = \Theta(n^2 \lg n)$ . Therefore, our induction hypothesis is  $T(m) \geq cm^2 \lg m$  for all  $m < n$ .

For  $m = 1$ , we have the base case that  $T(1) = 1 > c1^2 \lg 1$  for all  $c > 0$ .

For the inductive step, assume for all  $m < n$ ,  $T(m) \geq cm^2 \lg m$ . The induction hypothesis yields

$$\begin{aligned} T(n) &= 4T(n/2) + n^2 \\ &\geq 4c \left(\frac{n}{2}\right)^2 \lg \left(\frac{n}{2}\right) + n^2 \\ &= cn^2 \lg n - cn^2 \lg 2 + n^2 \\ &= cn^2 \lg n + (1 - c)n^2. \end{aligned}$$

For  $c < 1$ , this quantity is always greater than  $cn^2 \lg n$ . Therefore,  $T(n) = \Omega(cn^2 \lg n)$ .

**Problem 4. True or False, and Justify [35 points] (7 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. The more content you provide in your justification, the higher your grade, but be brief. Your justification is worth more points than your true-or-false designation.

- T F** Let  $A_1$ ,  $A_2$ , and  $A_3$  be three sorted arrays of  $n$  real numbers (all distinct). In the comparison model, constructing a balanced binary search tree of the set  $A_1 \cup A_2 \cup A_3$  requires  $\Omega(n \lg n)$  time.

**Solution: False.** First, merge the three arrays,  $A_1$ ,  $A_2$ , and  $A_3$  in  $O(n)$  time. Second, construct a balanced binary search tree from the merged array: the median of the array is the root; recursively build the left subtree from the first half of the array and the right subtree from the second half of the array. The resulting running time is  $T(n) = 2T(n/2) + O(1) = O(n)$ .

- T F** Let  $T$  be a complete binary tree with  $n$  nodes. Finding a path from the root of  $T$  to a given vertex  $v \in T$  using breadth-first search takes  $O(\lg n)$  time.

**Solution: False.** Breadth-first search requires  $\Omega(n)$  time. Breadth-first search examines each node in the tree in breadth-first order. The vertex  $v$  could well be the last vertex explored. (Also, notice that  $T$  is not necessarily sorted.)

**T F** Given an unsorted array  $A[1 \dots n]$  of  $n$  integers, building a max-heap out of the elements of  $A$  can be performed asymptotically faster than building a red-black tree out of the elements of  $A$ .

**Solution:** **True.** Building a heap takes  $O(n)$  time, as described in CLRS. On the other hand, building a red-black tree takes  $\Omega(n \lg n)$  time, since it is possible to produce a sorted list of elements from a red-black tree in  $O(n)$  time by doing an in-order tree walk (and sorting requires  $\Omega(n \lg n)$  time in a comparison model).

**T F** Suppose we use a hash function  $h$  to hash  $n$  distinct keys into an array  $T$  of length  $m$ . Assuming simple uniform hashing, the expected number of colliding pairs of elements is  $\Theta(n^2/m)$ .

**Solution:** **True.** Let  $X_{i,j}$  be an indicator random variable equal to 1 if elements  $i$  and  $j$  collide, and equal to 0 otherwise. Simple uniform hashing means that the probability of element  $i$  hashing to slot  $k$  is  $1/m$ . Therefore, the probability that  $i$  and  $j$  both hash to the same slot  $\Pr(X_{i,j}) = 1/m$ . Hence,  $E[X_{i,j}] = 1/m$ . We now use linearity of expectation to sum over all possible pairs  $i$  and  $j$ :

$$\begin{aligned} E[\text{number of colliding pairs}] &= E\left[\sum_{i=1}^n \sum_{j=i+1}^n X_{i,j}\right] \\ &= \sum_{i=1}^n \sum_{j=i+1}^n E[X_{i,j}] \\ &= \sum_{i=1}^n \sum_{j=i+1}^n 1/m \\ &= \frac{n(n+1)}{2m} \\ &= \Theta(n^2/m) \end{aligned}$$

**T F** Every sorting network with  $n$  inputs has depth  $\Omega(\lg n)$ .

**Solution: True.** Let  $d$  be the depth of the network. Since there are  $n$  inputs to the network, there can be at most  $nd$  comparators. (One way of seeing this is by the pigeon-hole principle: if there are  $n$  wires and more than  $nd$  comparators, then some wire must traverse more than  $d$  comparators, resulting in a depth  $> d$ .)

In the comparison-based model, it is possible to simulate the sorting network one comparator at a time. The running time is equal to the number of comparators (times a constant factor overhead). Therefore, every sorting network must have at least  $\Omega(n \lg n)$  comparators, by the lower-bound for sorting in the comparison-based model.

Therefore,  $nd > \Omega(n \lg n)$ , implying that the depth  $d$  is  $\Omega(\lg n)$ .

**T F** If a dynamic-programming problem satisfies the optimal-substructure property, then a locally optimal solution is globally optimal.

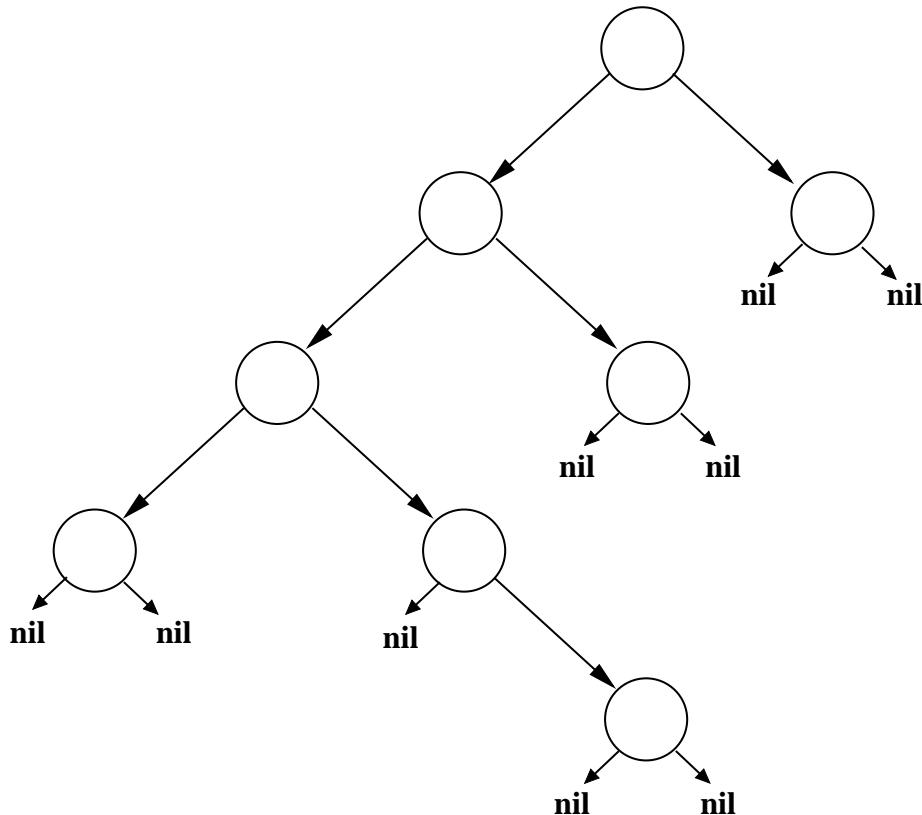
**Solution: False.** The property which implies that locally optimal solutions are globally optimal is the *greedy-choice property*. Consider as a counterexample the edit distance problem. This problem has optimal substructure, as was shown on the problem set, however a locally optimal solution—making the best edit next—does not result in a globally optimal solution.

**T F** Let  $G = (V, E)$  be a directed graph with negative-weight edges, but no negative-weight cycles. Then, one can compute all shortest paths from a source  $s \in V$  to all  $v \in V$  faster than Bellman-Ford using the technique of reweighting.

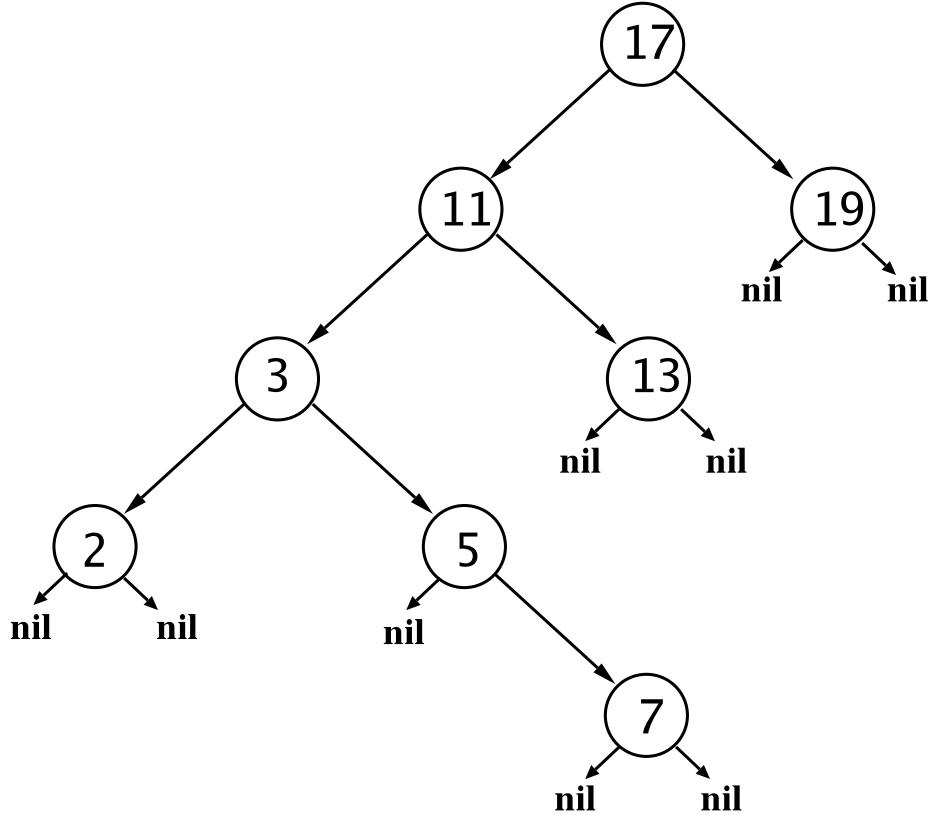
**Solution:** **False.** The technique of reweighting preserves the shortest path by assigning a value  $h(v)$  to each vertex  $v \in V$ , and using this to calculate new weights for the edges:  $\tilde{w}(u, v) = w(u, v) + h(u) - h(v)$ . However, to determine values for  $h(v)$  such that the edge weights are all non-negative, we use Bellman-Ford to solve the resulting system of difference constraints. Since the technique of reweighting relies on Bellman-Ford, it cannot run faster than Bellman-Ford.

**Problem 5. Red-black trees [15 points] (3 parts)**

- (a) Assign the keys 2, 3, 5, 7, 11, 13, 17, 19 to the nodes of the binary search tree below so that they satisfy the binary-search-tree property.



**Solution:** 5 points for the correct answer:



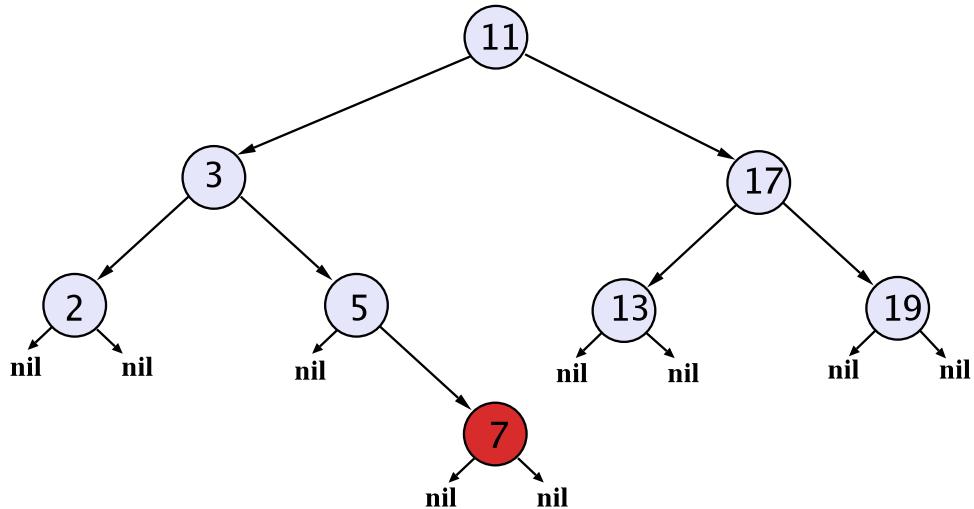
- (b) Explain why this binary search tree cannot be colored to form a legal red-black tree.

**Solution:** We prove this by contradiction. Suppose that a valid coloring exists. In a red-black tree, all paths from a node to descendant leaves contain the same number of black nodes. The path (17, 19, NIL) can contain at most three black nodes. Therefore the path (17, 11, 3, 5, 7, NIL) must also contain at most three black nodes and at least three red nodes. By the red-black tree properties, the root 17 must be black and the NIL node must also be black. This means that there must be three red nodes in the path (11, 3, 5, 7), but this would mean there are two consecutive red nodes, which violates the red-black tree properties. This is a contradiction, therefore the tree cannot be colored to form a legal red-black tree.

5 points for correct answer. 2-4 points for stating the red-black tree rules, but not proving why the tree does not satisfy the rules. 1 point for stating that the tree is unbalanced.

- (c) The binary search tree can be transformed into a red-black tree by performing a single rotation. Draw the red-black tree that results, labeling each node with “red” or “black.” Include the keys from part (a).

**Solution:** Rotate right around the root. There are several valid ways to color the resulting tree. Here is one possible answer (all nodes are colored black except for 7).



5 points for the correct answer. 2 points for the correct rotation but incorrect coloring.

**Problem 6. Wiggly arrays [10 points]**

An array  $A[1 \dots 2n + 1]$  is **wiggly** if  $A[1] \leq A[2] \geq A[3] \leq A[4] \geq \dots \leq A[2n] \geq A[2n + 1]$ . Given an unsorted array  $B[1 \dots 2n+1]$  of real numbers, describe an efficient algorithm that outputs a permutation  $A[1 \dots 2n + 1]$  of  $B$  such that  $A$  is a wiggly array.

**Solution:** There are several ways to solve this problem in  $\Theta(n)$  time. You can find the median  $k$  of  $B$  using the deterministic  $\Theta(n)$  select algorithm and partition  $B$  around the median  $k$  into two equal sized sets  $B_{low}$  and  $B_{high}$ . Assign  $A[1] \leftarrow k$ . Then for each  $i > 1$ , if  $i$  is even, assign an element from  $B_{high}$  to  $A[i]$ , otherwise assign an element from  $B_{low}$  to  $A[i]$ . Since all elements in  $B_{high}$  are greater than or equal to all elements in  $B_{low}$  and the median is less than or equal to all elements in  $B_{high}$ , the array is wiggly. The overall running time is  $\Theta(n)$ .

10 points for correct  $\Theta(n)$  solution and correct analysis. 8-9 points for correct solution but missing a minor step in the analysis.

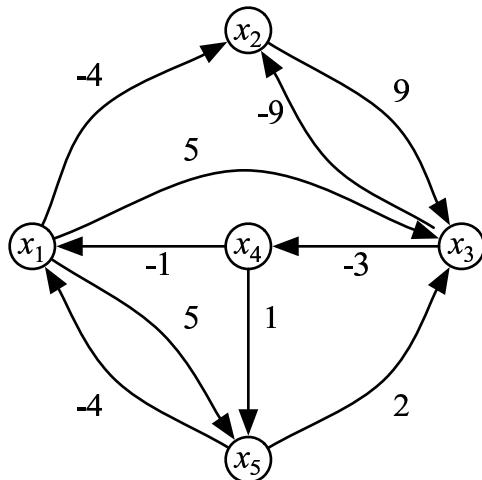
5 points for correct  $\Theta(n \lg n)$  solution and correct analysis. 2-4 points for correct solution but incomplete analysis.

**Problem 7. Difference constraints [12 points] (2 parts)**

Consider the following linear-programming system of difference constraints (note that one constraint is an equality):

$$\begin{aligned}x_1 - x_4 &\leq -1 \\x_1 - x_5 &\leq -4 \\x_2 - x_1 &\leq -4 \\x_2 - x_3 &= -9 \\x_3 - x_1 &\leq 5 \\x_3 - x_5 &\leq 2 \\x_4 - x_3 &\leq -3 \\x_5 - x_1 &\leq 5 \\x_5 - x_4 &\leq 1\end{aligned}$$

- (a) Draw the constraint graph for these constraints.



**Solution:** The equality constraint can be written as two inequality constraints,  $x_2 - x_3 \leq -9$ , and  $x_3 - x_2 \leq 9$ . A completely correct constraint graph received 6 points. Solutions that had edges in the wrong direction received only 4 points, and solutions that did not handle the equality constraint received 3 points.

- (b) Solve for the unknowns  $x_1, x_2, x_3, x_4$ , and  $x_5$ , or explain why no solution exists.

**Solution:** No solution to this system exists because the constraint graph has a negative-weight cycle. For example,  $x_1 \rightsquigarrow x_3 \rightsquigarrow x_4 \rightsquigarrow x_5 \rightsquigarrow x_1$  has weight  $5 - 3 + 1 - 4 = -1$ . A full-credit solution (6 points) must exhibit the negative weight cycle.

**Problem 8. Amortized increment [12 points]**

An array  $A[0 \dots k-1]$  of bits (each array element is 0 or 1) stores a binary number  $x = \sum_{i=0}^{k-1} A[i] \cdot 2^i$ .

To add 1 (modulo  $2^k$ ) to  $x$ , we use the following procedure:

```

INCREMENT( $A, k$ )
1  $i \leftarrow 0$ 
2 while  $i < k$  and  $A[i] = 1$ 
3   do  $A[i] \leftarrow 0$ 
4      $i \leftarrow i + 1$ 
5 if  $i < k$ 
6   then  $A[i] \leftarrow 1$ 

```

Given a number  $x$ , define the potential  $\Phi(x)$  of  $x$  to be the number of 1's in the binary representation of  $x$ . For example,  $\Phi(19) = 3$ , because  $19 = 10011_2$ . Use a potential-function argument to prove that the amortized cost of an increment is  $O(1)$ , where the initial value in the counter is  $x = 0$ .

**Solution:**  $\Phi(x)$  is a valid potential function the number of 1's in the binary representation of  $x$  is always nonnegative, i.e.,  $\Phi(x) \geq 0$  for all  $x$ . Since the initial value of the counter is 0,  $\Phi_0 = 0$ .

Let  $c_k$  be the real cost of the operation  $\text{INCREMENT}(A, k)$ , and let  $d_k$  be the number of times the while loop body in Lines 3 and 4 execute (i.e.,  $d_k$  is the number of consecutive 1's counting from the least-significant bit of the binary representation of  $x$ ). If we assume that executing all of Lines 1, 2, 5, and 6 require unit cost, and executing the body of the while loop requires unit cost, then the real cost is  $c_k = 1 + d_k$ ,

The potential decreases by one every time the while loop is executed. Therefore, the change in potential,  $\Delta\Phi = \Phi_k - \Phi_{k-1}$  is at most  $1 - d_k$ . More specifically,  $\Delta\Phi = 1 - d_k$  if we execute Line 6, and  $\Delta\Phi = -d_k$  if we do not.

Thus, using the formula for amortized cost, we get

$$\begin{aligned}
\hat{c}_k &= c_k + \Delta\Phi \\
&= 1 + d_k + \Delta\Phi \\
&\leq 1 + d_k + 1 - d_k \\
&= 2.
\end{aligned}$$

Therefore, the amortized cost of an increment is  $O(1)$ .

Only solutions that explicitly calculate/explain the real cost  $c_k$  of a single increment received full credit. Solutions that give an aggregate analysis or any other method that did not use the potential function received at most 6 points. One or two points were deducted from correct solutions that did not explain why the potential function is valid or calculate  $\Phi_0$ .

**Problem 9. Minimum spanning trees [12 points]**

Let  $G = (V, E)$  be a connected, undirected graph with edge-weight function  $w : E \rightarrow \mathbb{R}$ , and assume all edge weights are distinct. Consider a cycle  $\langle v_1, v_2, \dots, v_k, v_{k+1} \rangle$  in  $G$ , where  $v_{k+1} = v_1$ , and let  $(v_i, v_{i+1})$  be the edge in the cycle with the largest edge weight. Prove that  $(v_i, v_{i+1})$  does *not* belong to the minimum spanning tree  $T$  of  $G$ .

**Solution:** Proof by contradiction. Assume for the sake of contradiction that  $(v_i, v_{i+1})$  does belong to the minimum spanning tree  $T$ . Removing  $(v_i, v_{i+1})$  from  $T$  divides  $T$  into two connected components  $P$  and  $Q$ , where some nodes of the given cycle are in  $P$  and some are in  $Q$ . For any cycle, at least two edges must cross this cut, and therefore there is some other edge  $(v_j, v_{j+1})$  on the cycle, such that adding this edge connects  $P$  and  $Q$  again and creates another spanning tree  $T'$ . Since the weight of  $(v_j, v_{j+1})$  is less than  $(v_i, v_{i+1})$ , the weight of  $T'$  is less than  $T$  and  $T$  cannot be a minimum spanning tree. Contradiction.

**Problem 10. Multithreaded scheduling [10 points]**

A greedy scheduler runs a multithreaded computation in 260 seconds on 4 processors and in 90 seconds on 32 processors. Is it possible that the computation has work  $T_1 = 1024$  and critical-path length  $T_\infty = 64$ ? Justify your answer.

**Solution:** For greedy schedulers, we know that  $\min\{T_1/P, T_\infty\} \leq T_P \leq T_1/P + T_\infty$ . The numbers given above satisfy all of the above inequalities for both values of  $P$ . Therefore, it is possible that the work and critical path are correct.

**Problem 11. Transposing a matrix [40 points] (4 parts)**

Let  $X$  be an  $N \times N$  matrix, where  $N$  is an exact power of 2. The following code computes  $Y = X^T$ :

```
TRANS( $X, Y, N$ )
1 for  $i \leftarrow 1$  to  $N$ 
2   do for  $j \leftarrow 1$  to  $N$ 
3     do  $Y[j, i] \leftarrow X[i, j]$ 
```

Consider the cache-oblivious two-level memory model with a cache of  $M$  elements and blocks of  $B$  elements. Assume that both matrices  $X$  and  $Y$  are stored in row-major order, that is, the linear order of  $X$  in memory is  $X[1, 1], X[1, 2], \dots, X[1, N], X[2, 1], X[2, 2], \dots, X[2, N], \dots, X[N, 1], X[N, 2], \dots, X[N, N]$ , and similarly for  $Y$ .

- (a) Analyze the number  $MT(N)$  of memory transfers incurred by TRANS when  $N \gg M$ .

**Solution:** Since the loop scans through matrix  $X$  one row at a time, the number of memory transfers required for accessing  $X$  is  $O(N^2/B)$ . We incur  $O(N^2)$  memory transfers to access  $Y$ , however, because  $Y$  is accessed one column at a time. When we access the block containing  $Y[j, i]$ , since  $N \gg M$  and we scan through all of column  $i$  before accessing  $Y[j + 1, i]$ , each access to  $Y$  may incur a memory transfer.

Therefore,  $MT(N) = O(N^2)$ .

Now, consider the following divide-and-conquer algorithm for computing the transpose:

```
R-TRANS( $X, Y, N$ )
1  if  $N = 1$ 
2    then  $Y[1, 1] \leftarrow X[1, 1]$ 
3    else Partition  $X$  into four  $(N/2) \times (N/2)$  submatrices  $X_{11}, X_{12}, X_{21}$ , and  $X_{22}$ .
4      Partition  $Y$  into four  $(N/2) \times (N/2)$  submatrices  $Y_{11}, Y_{12}, Y_{21}$ , and  $Y_{22}$ .
5      R-TRANS( $X_{11}, Y_{11}, N/2$ )
6      R-TRANS( $X_{12}, Y_{21}, N/2$ )
7      R-TRANS( $X_{21}, Y_{12}, N/2$ )
8      R-TRANS( $X_{22}, Y_{22}, N/2$ )
```

Assume that the cost of partitioning is  $O(1)$ .

- (b)** Give and solve a recurrence for the number  $MT(N)$  of memory transfers incurred by R-TRANS when  $N \gg M$ .

**Solution:** If we make either the tall-cache assumption (i.e.,  $M = \Omega(B^2)$ ), or we assume that the matrix is stored in the recursive block layout, then

$$MT(n) = \begin{cases} 4T(n/2) + \Theta(1) & \text{if } cn^2 > M, \\ M/B & \text{if } cn^2 \leq M \end{cases}.$$

The recursion tree has approximately  $\lg N - \lg(\sqrt{M/c})$  levels. Since every level has 4 times as many nodes as the previous level, the solution to the recurrence is dominated by the cost of the leaves. The tree has  $4^{\lg(N\sqrt{c}/\sqrt{M})}$  leaves, each of cost  $M/B$ . Therefore,  $MT(N)$  is

$$\begin{aligned} MT(N) &= \frac{M}{B} \left( \frac{N\sqrt{c}}{\sqrt{M}} \right)^2 \\ &= \frac{cN^2}{B} \\ &= O\left(\frac{N^2}{B}\right). \end{aligned}$$

The following multithreaded algorithm computes the transpose in parallel:

```
P-TRANS( $X, Y, N$ )
1  if  $N = 1$ 
2    then  $Y[1, 1] \leftarrow X[1, 1]$ 
3    else Partition  $X$  into four  $(N/2) \times (N/2)$  submatrices  $X_{11}, X_{12}, X_{21}$ , and  $X_{22}$ .
4      Partition  $Y$  into four  $(N/2) \times (N/2)$  submatrices  $Y_{11}, Y_{12}, Y_{21}$ , and  $Y_{22}$ .
5      spawn P-TRANS( $X_{11}, Y_{11}, N/2$ )
6      spawn P-TRANS( $X_{12}, Y_{21}, N/2$ )
7      spawn P-TRANS( $X_{21}, Y_{12}, N/2$ )
8      spawn P-TRANS( $X_{22}, Y_{22}, N/2$ )
9      sync
```

- (c) Give and solve recurrences describing the work  $T_1(N)$  and critical-path length  $T_\infty(N)$  of P-TRANS. What is the asymptotic parallelism of the algorithm?

**Solution:**

$$\begin{aligned} T_1(N) &= 4T_1\left(\frac{N}{2}\right) + O(1) \\ &= \Theta(N^2). \\ T_\infty(N) &= T_\infty\left(\frac{N}{2}\right) + O(1) \\ &= \Theta(\lg N). \end{aligned}$$

The parallelism of the algorithm is  $T_1/T_\infty$ , or  $\Theta(N^2/\lg N)$ .

Professor Kellogg inadvertently places two additional **sync** statements into his code as follows:

```

K-TRANS( $X, Y, N$ )
1  if  $N = 1$ 
2    then  $Y[1, 1] \leftarrow X[1, 1]$ 
3    else Partition  $X$  into four  $(N/2) \times (N/2)$  submatrices  $X_{11}, X_{12}, X_{21}$ , and  $X_{22}$ .
4      Partition  $Y$  into four  $(N/2) \times (N/2)$  submatrices  $Y_{11}, Y_{12}, Y_{21}$ , and  $Y_{22}$ .
5      spawn K-TRANS( $X_{11}, Y_{11}, N/2$ )
6      sync
7      spawn K-TRANS( $X_{12}, Y_{21}, N/2$ )
8      sync
9      spawn K-TRANS( $X_{21}, Y_{12}, N/2$ )
10     spawn K-TRANS( $X_{22}, Y_{22}, N/2$ )
11     sync
```

- (d) Give and solve recurrences describing the work  $T_1(N)$  and critical-path length  $T_\infty(N)$  of K-TRANS. What is the asymptotic parallelism of this algorithm?

**Solution:** The work for the algorithm remains the same.

$$\begin{aligned} T_1(N) &= 4T_1\left(\frac{N}{2}\right) + O(1) \\ &= \Theta(N^2). \end{aligned}$$

The critical path increases, however, because we only solve one of the subproblems in parallel.

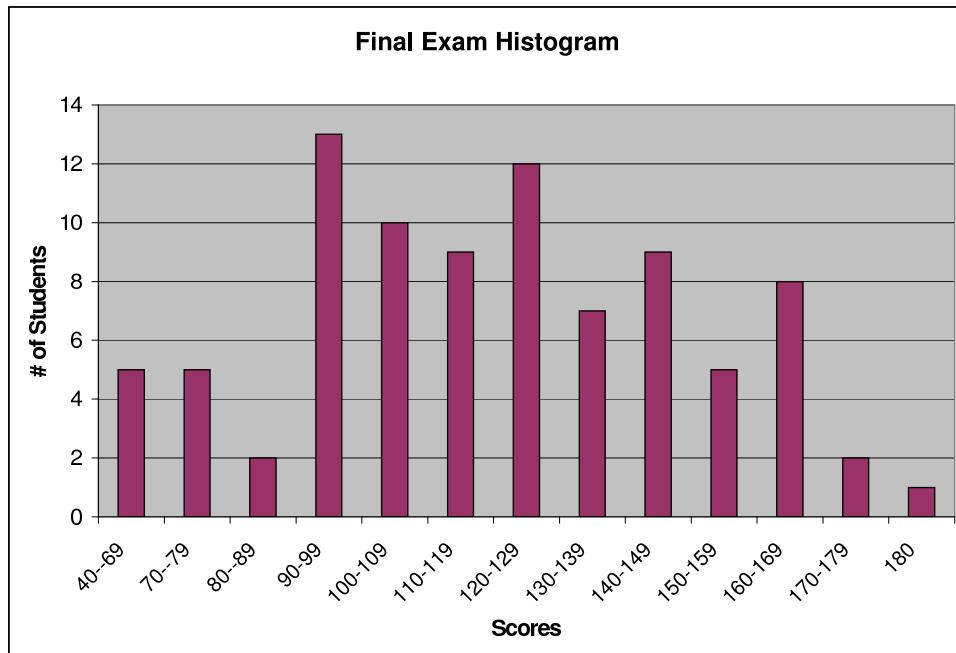
$$\begin{aligned} T_\infty(N) &= 3T_\infty\left(\frac{N}{2}\right) + O(1) \\ &= \Theta(N^{\lg 3}). \end{aligned}$$

The parallelism of the algorithm is  $T_1/T_\infty$ , or  $\Theta(N^{2-\lg 3})$ .

**SCRATCH PAPER** — Please detach this page before handing in your exam.

**SCRATCH PAPER** — Please detach this page before handing in your exam.

## Final Exam Solutions



Problem	Parts	Points	Grade	Grader
1	16	80		
2	3	20		
3	4	30		
4	4	30		
5	4	20		
Total		180		

**Problem 1. True or False, and Justify [80 points] (16 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. The more content you provide in your justification, the higher your grade, but be brief. Your justification is worth more points than your true-or-false designation.

**T F** For every two positive functions  $f$  and  $g$ , if  $g(n) = O(n)$ , then  $f(g(n)) = O(f(n))$ .

**Solution: False.**

Let  $f(n) = 2^n$  and  $g(n) = 2n = O(n)$ . Then  $f(g(n)) = 2^{2n} = (2^n)^2 = (f(n))^2 \neq O(f(n))$ .

**T F** Suppose  $f(n) = 4 f(n/4) + n$  for  $n > 8$ , and  $f(n) = O(1)$  for  $n \leq 8$ . Similarly, suppose  $g(n) = 3 g(n/4) + n \lg n$  for  $n > 8$ , and  $g(n) = O(1)$  for  $n \leq 8$ . Then  $f(n) = \Theta(g(n))$ .

**Solution: True.**

By the Master Method (Cases 1 and 3), both are  $\Theta(n \lg n)$ .

- T F** Suppose that a randomized algorithm  $A$  has expected running time  $\Theta(n^2)$  on any input of size  $n$ . Then it is possible for some execution of  $A$  to take  $\Omega(2^n)$  time.

**Solution:** **True.**

Consider a (somewhat dumb) sorting algorithm that first sorts  $n$  items using mergesort, in time  $\Theta(n \lg n)$ , and then flips  $n$  (fair) coins. If at least one coin turns up heads, the algorithm waits  $n^2$  additional units of time before terminating. If all coins turn up tails, it waits  $2^n$  units of time. Although in the latter case the running time is  $\Omega(2^n)$ , the expected running time is only  $\Theta(n \lg n) + (1 - 2^{-n})(n^2) + 2^{-n}(2^n) = \Theta(n^2)$ .

- T F** Suppose we maintain a hash table with  $m$  slots using chaining and a hash function chosen from a universal hash family. If we insert  $n > m$  keys into this (initially empty) hash table, then the total number of collisions is  $O(n/m)$  in expectation. (Recall that a **collision** is a pair of distinct keys that hash to the same slot.)

**Solution:** **False.**

In expectation, any particular key  $x$  collides with  $\Theta(n/m)$  other keys. Thus, summing over all  $n$  keys, we have a total of  $\Theta(n^2/m) \neq O(n/m)$  collisions.

**T F** Building an  $n$ -element heap requires  $\Theta(n \lg n)$  time.

**Solution: False.**

It takes  $\Theta(n)$  time. See CLRS, pages 133–135.

**T F** Given an unsorted array  $A$  of  $n$  integers, let  $x_i$  denote the  $2^i$ th smallest element in  $A$ . Then

we can compute  $\sum_{i=0}^{\lfloor \lg n \rfloor} x_i$  in  $O(n)$  time.

**Solution: True.**

Initialize  $i \leftarrow \lfloor \lg n \rfloor$ ,  $\tilde{A} \leftarrow A$ , and  $s \leftarrow 0$ , and then repeat the following until  $i < 0$ : compute the  $2^i$ th element  $x_i$  of  $\tilde{A}$  using SELECT, set  $s \leftarrow s + x_i$ , update  $\tilde{A}$  to be those elements of  $\tilde{A}$  that are smaller than  $x_i$ , and update  $i$  to be  $i - 1$ . At the end, return  $s$ . Each iteration takes time linear in the size of  $\tilde{A}$ . In the first iteration (in which  $i = \lfloor \lg n \rfloor$ ),  $\tilde{A}$  has  $n$  elements, so this iteration takes  $c n$  time (for some appropriate constant  $c > 0$ ); in each subsequent iteration,  $\tilde{A}$  has  $2^{i+1} - 1$  elements, so the time taken is  $c(2^{i+1} - 1)$ . Thus the total time taken is

$$c n + \sum_{i=0}^{\lfloor \lg n \rfloor - 1} c(2^{i+1} - 1) = \Theta(n).$$

**T F** Suppose that you have a 2-3-4 tree  $T_1$  and a red-black tree  $T_2$ , each storing the same set of keys. Then in-order traversals of  $T_1$  and  $T_2$  can result in different sequences of keys.

**Solution:** **False.**

Both 2-3-4 trees and red-black trees are search trees. Thus, an in-order traversal of either produces the (same) sorted sequence of keys.

**T F** There are at least two distinct red-black trees containing keys 1, 2, 3, 4, 5.

**Solution:** **True.**

For example, any of the  $\binom{4}{2}$  trees with a black root, two black children, and two red grandchildren.

**T F** Graduating from MIT requires passing  $n$  specified classes. You decide to take each class every semester until you pass it. Suppose that, every semester you take a class, you have a 50% chance of passing it and a 50% chance of having to drop it. Then you will graduate in  $O(\lg n)$  semesters with high probability.

**Solution:** **True.**

For each of the  $n$  courses, the probability that do not pass the course after  $k \lg n$  semesters is  $(1/2)^{k \lg n} = 1/n^k$ . Therefore, the probability that there exists a course that you do not pass after  $k \lg n$  semesters is at most  $1/n^{k-1}$ . Thus you graduate with probability at least  $1 - 1/n^{k-1}$ . Choosing  $k$  (and thus  $k-1$ ) to be an arbitrarily large constant, the number of semesters is  $O(\lg n)$  with high probability. (This derivation is essentially the same as Problem 2 on Problem Set 5.)

**T F** If a sequence of  $m$  operations has amortized cost  $c$  per operation, then the actual (not amortized) cost of the  $m$ th operation is always  $O(cm)$ .

**Solution:** **True.**

Because the amortized cost of each operation is  $c$ , the total cost of a sequence of  $m$  operations is at most  $cm$ . Hence, in particular, the cost of  $m$ th operation is  $O(cm)$ .

**T F** Suppose that you have two deterministic online algorithms,  $A_1$  and  $A_2$ , with competitive ratios  $c_1$  and  $c_2$ , respectively. Consider the randomized online algorithm  $A^*$  that flips a fair coin once at the beginning; if the coin comes up heads, it runs  $A_1$  from then on; if the coin comes up tails, it runs  $A_2$  from then on. Then the expected competitive ratio of  $A^*$  is at least  $\min\{c_1, c_2\}$ .

**Solution:** **False.**

Randomization can help a lot. In particular, the expected competitive ratio of  $A^*$  can be smaller than  $\min\{c_1, c_2\}$ . In class we have seen several examples of problems where this happens. For instance, see Problem 2 on Problem Set 6.

**T F** In a connected undirected graph  $G = (V, E, w)$  with nonnegative edge weights, the shortest-path tree from any source vertex  $s \in V$  is a minimum spanning tree of  $G$ . (Recall that the **shortest-path tree** from  $s$  consists of the edges  $\{(\pi(v), v) : v \in V - \{s\}\}$  where, for each  $v \in V - \{s\}$ ,  $(\pi(v), v)$  is the last relaxed incoming edge at  $v$  in an execution of Dijkstra's single-source shortest-paths algorithm from  $s$ .)

**Solution:** **False.**

Consider a graph with three nodes:  $s$ ,  $v$ , and  $t$ . Let  $w(s, v) = w(v, t) = 3$  and  $w(s, t) = 4$ . The shortest-path tree from  $s$  consists of two edges  $(s, v)$  and  $(s, t)$ , while the minimum spanning tree is  $\{(s, v), (v, t)\}$ .

**T F** Reweighting a graph with negative edge weights but no negative-weight cycles, as in Johnson's algorithm, can be used to solve the single-source shortest-paths problem more efficiently than Bellman-Ford.

**Solution: False.**

Johnson's reweighting uses the Bellman-Ford algorithm as a subroutine, to solve a system of difference constraints. Therefore Johnson's reweighting cannot be more efficient than Bellman-Ford.

**T F** Every problem in NP can be solved in exponential time.

**Solution: True.**

The brute-force search algorithm can solve every such problem in exponential time. This algorithm guesses a certificate  $y$  of polynomial length  $n^c$ , and runs the verification algorithm on the input  $x$  and certificate  $y$  in  $n^{c'}$  time. If the verification algorithm outputs 1 for any certificate, the brute-force algorithm returns 1; otherwise, it returns 0. The running time of this algorithm is  $O(2^{n^c} n^{c'})$ , which is  $O(2^{n^{c''}})$  for a suitable constant  $c''$ .

- T F** For any decision problem  $\pi$  in NP, define the input size  $n$  as the parameter  $k$  to fix. Then  $\pi$  is fixed-parameter tractable with respect to  $n$ .

**Solution:** **True.**

Recall that a parameterized problem is fixed-parameter tractable with respect to a parameter  $k$  if the problem can be solved in  $f(k) \cdot n^c$  time for some constant  $c$ . Clearly, problems in NP are FPT with respect to  $n$ , because by the previous part they can be solved in  $f(n) = O(2^{n^c})$  time for some constant  $c$ .

- T F** Define an *independent set* of a graph  $G = (V, E)$  to be a subset  $S \subseteq V$  of vertices such that  $V - S$  is a vertex cover of  $G$ . Every 2-approximation algorithm for finding a minimum vertex cover is also a 2-approximation algorithm for finding a maximum independent set.

**Solution:** **False.**

Let  $G$  be a cycle on six vertices. Clearly, both the maximum independent set and the minimum vertex cover of  $G$  are of size 3. A 2-approximation to minimum vertex cover may have size up to 6, but its complement will not necessarily be a 2-approximation to maximum independent set, having size as low as 0.

**Problem 2. One, One Room; Two, Two Rooms; Ah Ha Ha!** [20 points] (3 parts)

You are maintaining a hotel room reservation system for an  $n$ -day period, with dates labeled  $1, 2, \dots, n$ . Your reservation system must support two operations:

- $\text{RESERVE}(i, j)$  makes a room reservation for the dates  $i, i + 1, \dots, j$ .
- $\text{COUNT}(i)$  computes how many rooms are currently reserved on day  $i$ .

Your goal is to construct a data structure that supports both operations in  $O(\lg n)$  time. Assume that  $n$  is an exact power of two.

You decide to maintain your data in the form of a perfectly balanced binary tree. The root corresponds to the entire interval  $[1 \dots n]$ ; the root's left child corresponds to the interval  $[1 \dots n/2]$ ; the root's right child corresponds to the interval  $[(n/2 + 1) \dots n]$ ; etc. At the (bottom) leaf level, each leaf corresponds to a single day, and the leaf order matches the day order. Thus there are exactly  $n$  leaves and  $1 + \lg n$  levels.

- (a) What additional information would you maintain in the nodes in order to support the updates and queries efficiently?

**Solution:** For each node  $x$  representing some interval  $I_x$  of dates, we store a non-negative integer  $c[x]$ . This value  $c[x]$  counts the number of reservation intervals  $[i, j]$  that wholly contain  $I_x$  but do not wholly contain the parent interval  $I_{p[x]}$ .

- (b) Briefly describe how you would implement the  $\text{COUNT}(i)$  operation. Briefly justify why the running time is  $O(\lg n)$ .

**Solution:**  $\text{COUNT}(i)$  walks the path  $x_0, x_1, x_2, \dots, x_{\lg n}$  from the leaf  $x_0$  corresponding to date  $i$  up to the root  $x_{\lg n}$  of the tree. It returns the sum of the  $c$  values along this path:  $c[x_0] + c[x_1] + c[x_2] + \dots + c[x_{\lg n}]$ . This computation takes  $O(\lg n)$  time because we spend  $O(1)$  time at each node along the path of length  $1 + \lg n$ .

We claim that this sum equals the number of reservation intervals containing date  $i$ . To see this, consider a reservation interval  $I$  containing  $i$ . The intervals  $I_0, I_1, I_2, \dots, I_{\lg n}$  along the leaf-to-root path are nested:  $I_0 \subseteq I_1 \subseteq I_2 \subseteq \dots \subseteq I_{\lg n}$ . Hence, there is a unique  $k$  such that  $I$  wholly contains  $I_k$  but  $I$  does not wholly contain the parent interval  $I_{k+1}$ . Thus,  $c[x_k]$  counts  $I$ , and no other  $c[x_{k'}]$  counts  $I$ .

- (c) Briefly describe how you would implement the  $\text{RESERVE}(i, j)$  operation. Briefly justify why the running time is  $O(\lg n)$ .

**Solution:**  $\text{RESERVE}(i, j)$  simultaneously walks up the tree from the two leaves  $x_0$  and  $y_0$  corresponding to  $i$  and  $j$ , visiting nodes  $x_0, x_1, x_2, \dots$  and  $y_0, y_1, y_2, \dots$ , until the two paths meet at some node  $x_k = y_k$ , the lowest common ancestor of  $x_0$  and  $y_0$ . We define a **test** operation on a node  $x$ : if  $[i, j]$  wholly contains  $x$ 's interval but not  $x$ 's parent's interval, then we increment  $c[x]$ . Then  $\text{RESERVE}(i, j)$  tests  $x_k = y_k$ , and tests each child of every node  $x_0, y_0, x_1, y_1, \dots, x_{k-1}, y_{k-1}, x_k = y_k$ . It is easy to see that these  $O(\lg n)$  nodes are all that we need to test, and that the running time is thus  $O(\lg n)$ .

**Problem 3. Amortization is as Easy as 2-3-4** [30 points] (4 parts)

In this problem, you will analyze the amortized number of node modifications caused by inserting a key into a 2-3-4 tree. To do so, define the potential function  $\Phi$  to be the number of nodes in the tree with exactly three keys.

Also, we introduce the following notation. Define the **cost**  $c_i$  of the  $i$ th insertion to be the number of nodes it splits or otherwise modifies. Define  $\Delta\Phi_i$  to be the change in potential during the  $i$ th insertion (positive if  $\Phi$  goes up, negative if  $\Phi$  goes down). Define  $\hat{c}_i = c_i + \Delta\Phi_i$ .

- (a) Suppose you do a key insertion into a leaf that currently has less than three keys. What values can  $c_i$ ,  $\Delta\Phi_i$ , and  $\hat{c}_i$  have?

**Solution:** First,  $c_i = 1$  because the insertion modifies just the leaf.

Second,  $\Delta\Phi_i \in \{0, 1\}$ . If the leaf initially had only one key, so after the insertion it has two keys, then  $\Delta\Phi_i = 0$ . If the leaf initially had two keys, so after the insertion it has three keys, then  $\Delta\Phi_i = 1$ . The leaf could not have three keys, because that would have caused a split.

Hence,  $\hat{c}_i = c_i + \Delta\Phi_i \in \{1, 2\}$ .

- (b) Suppose you do a key insertion that causes  $k$  nodes to split. What values can  $c_i$ ,  $\Delta\Phi_i$ , and  $\hat{c}_i$  have?

**Solution:** First,  $c_i = k + 1$ . By definition, the insertion causes  $k$  splits. In addition, the last split causes a modification to the parent of that node (or, if the root was split, the creation of a new root), incurring the additive 1 in cost.

Second,  $\Delta\Phi_i \in \{-k, -k + 1\}$ . Each of the  $k$  nodes that split must have previously had exactly three keys, and after splitting, all of the resulting nodes have at most two keys, causing the main potential drop of  $-k$ . However, the parent of the last split node gains a new child, so if it previously had two keys, it will then have three keys, causing a potential increase of  $+1$ . On the other hand, if the parent of the last split node had only one key (the only other possibility), there is no such potential increase.

Hence,  $\hat{c}_i = c_i + \Delta\Phi_i \in \{1, 2\}$ .

- (c) What else do you need to show about the potential function  $\Phi$  to get an upper bound on the number of node modifications caused by a sequence of  $n$  key insertions into an initially empty 2-3-4 tree?

**Solution:** First, we need that  $\Phi_i \geq 0$  for all  $i$ . This is true by definition of  $\Phi$ : there cannot be a negative number of nodes with any property.

Second, we need that  $\Phi_0 = 0$ . This is true because the stated initial tree is empty, which has no nodes whatsoever.

- (d) Give an upper bound on the number of node modifications caused by a sequence of  $n$  key insertions into an initially empty 2-3-4 tree. Is your answer asymptotically tight?

**Solution:** Because  $\hat{c}_i \leq 2$  for all  $i$  by part (b), and by the properties of  $\Phi$  from part (c), we obtain an upper bound of  $2n$ .

This bound is tight up to a factor of 2, because every insertion modifies at least one node (the leaf into which it inserts), so the number of node modifications is at least  $n$ .

**Problem 4. Forty Two** [30 points] (4 parts)

Professor Hackermann has finally cracked the meaning of life, the universe, and everything: it is  $H(42, 42)$  where the function  $H(m, n)$  is defined by the following unusual recurrence:

$$\begin{aligned} H(m, 1) &= m^2; \\ H(1, n) &= n^3; \\ H(m, n) &= H(\text{FOO}(m, n), n) + H(m, \text{BAR}(m, n)) \quad \text{for all other values of } m, n \geq 1. \end{aligned}$$

Professor Hackermann knows how to compute  $\text{FOO}(m, n)$  and  $\text{BAR}(m, n)$  in  $O(1)$  time for given values of  $m, n \geq 1$ . The catch is that  $\text{FOO}(m, n)$  and  $\text{BAR}(m, n)$  can sometimes be larger than  $m$  and  $n$ , so the recurrence does not obviously terminate. Nonetheless, both  $\text{FOO}(m, n)$  and  $\text{BAR}(m, n)$  have value 1 often enough that the recursive formula may allow computing  $H(m, n)$ .

- (a) The professor hires you to compute  $H(3, 4)$  by hand, using the following information about  $\text{FOO}$  and  $\text{BAR}$ :

$$\begin{array}{lll} \text{FOO}(3, 4) & = & 7; & \text{BAR}(3, 4) & = & 1; \\ \text{FOO}(7, 4) & = & 10; & \text{BAR}(7, 4) & = & 2; \\ \text{FOO}(10, 4) & = & 1; & \text{BAR}(10, 4) & = & 1; \\ \text{FOO}(7, 2) & = & 1; & \text{BAR}(7, 2) & = & 1. \end{array}$$

Show your work.

**Solution:**

$$\begin{aligned} H(7, 2) &= H(1, 2) + H(7, 1) = 8 + 49 = 57, \\ H(10, 4) &= H(1, 4) + H(10, 1) = 64 + 100 = 164, \\ H(7, 4) &= H(10, 4) + H(7, 2) = 164 + 57 = 221, \\ H(3, 4) &= H(7, 4) + H(3, 1) = 221 + 9 = 230. \end{aligned}$$

To understand whether  $H(42, 42)$  can be computed with the recursive formula, Professor Hackermann sets out to understand which pairs  $(m', n')$  arise from the recursion. The professor defines  $\text{descendants}(m, n)$  to be the set of pairs  $(m', n')$  for which  $H(m', n')$  is required to compute  $H(m, n)$ , i.e.,

$$\begin{aligned}\text{descendants}(m, n) &= \left\{ \left( \text{FOO}(m, n), n' \right), \left( m, \text{BAR}(m, n) \right) \right\} \\ &\cup \text{descendants}\left( \text{FOO}(m, n), n' \right) \cup \text{descendants}\left( m, \text{BAR}(m, n) \right).\end{aligned}$$

Note that  $\text{descendants}(m, n)$  does not necessarily include  $(m, n)$ .

- (b) To thwart critics who claim that pairs  $(m', n')$  in  $\text{descendants}(m, n)$  can grow without bound, Professor Hackermann makes the following conjecture:

**Conjecture 1** For every  $m, n \geq 1$ , and for every pair  $(m', n') \in \text{descendants}(m, n)$ , we have both  $m' \leq (m + n)^3$  and  $n' \leq (m + n)^3$ .

Give an algorithm that, on input  $m, n \geq 1$ , determines whether Conjecture 1 is true for this pair of integers, i.e., whether  $m', n' \leq (m + n)^3$  for every pair  $(m', n') \in \text{descendants}(m, n)$ . Your algorithm must run in time polynomial in  $m + n$ .

**Solution:** To verify Conjecture 1 for a given pair of integers  $m$  and  $n$ , we recursively compute the pairs that belong to  $\text{descendants}(m, n)$ , using an  $(m+n)^3 \times (m+n)^3$  table  $A$  to track the pairs  $(m', n')$  we have already visited. When expanding the recursive definition of  $\text{descendants}(m, n)$ , if we ever discover a pair  $(m', n')$  out of range, (i.e., either  $m' > (m + n)^3$  or  $n' > (m + n)^3$ ), we know that Conjecture 1 is false. On the other hand, if all elements we check are in range, then Conjecture 1 holds for  $(m, n)$ . We fill in the table recursively, but use memoization to avoid recomputing repeated subproblems.

```

1 for  $i \leftarrow 1$  to  $(m + n)^3$        $\triangleright$  Initialize table
2   do for  $j \leftarrow 1$  to  $(m + n)^3$ 
3     do  $A[m, n] \leftarrow \text{FALSE}$ 
4 return CHECK-C1( $m, n, m, n$ )

```

```

CHECK-C1( $m', n', m, n$ )       $\triangleright$  Check Conjecture 1
1   if  $m' > (m + n)^3$  or  $n' > (m + n)^3$ 
2     then return FALSE
3   if  $A[m', n']$ 
4     then return TRUE
5   else  $A[m', n'] \leftarrow \text{TRUE}$        $\triangleright$  Do work only if we have not visited  $(m', n')$ 
6      $t_1 \leftarrow \text{CHECK-C1}(\text{FOO}(m', n'), n', m, n)$ 
7      $t_2 \leftarrow \text{CHECK-C1}(m', \text{BAR}(m', n'), m, n)$ 
8   return ( $t_1$  and  $t_2$ )

```

The amortized cost to fill any single element in the table  $A$  is  $O(1)$ , because we can charge the cost of making recursive calls (Lines 7–9) and the cost of checking the base cases (Lines 1–4) to each table element of  $A$  that is filled in (Line 6). Because we have  $(m + n)^6$  table elements, the running time of the algorithm is  $O((m + n)^6)$ .

- (c) More critics claim that the professor's recursive formula is useless because it could be cyclic: the computation of  $H(m, n)$  could require the computation of  $H(m, n)$  itself, leading to an infinite recursion. To thwart these critics, Professor Hackermann makes another conjecture:

**Conjecture 2** For every  $m, n \geq 1$ , we have  $(m, n) \notin \text{descendants}(m, n)$ .

Give an algorithm that, on input  $m, n \geq 1$ , determines whether Conjecture 2 is true for this pair of integers, i.e., whether  $(m, n) \notin \text{descendants}(m, n)$ . Your algorithm must run in time polynomial in  $m + n$ , and it may assume that Conjecture 1 holds.

**Solution:** Assuming Conjecture 1 holds, all pairs  $(m', n') \in \text{descendants}(m, n)$  are represented in the  $(m+n)^3 \times (m+n)^3$  table  $A$ . Thus, we can use a memoized recursive algorithm similar to the one in part (b), except with the added check for whether we encounter  $m' = m$  and  $n' = n$  after the first step.

```

1 for  $i \leftarrow 1$  to  $(m + n)^3$        $\triangleright$  Initialize table
2     do for  $j \leftarrow 1$  to  $(m + n)^3$ 
3         do  $A[m, n] \leftarrow \text{FALSE}$ 
4      $t_1 \leftarrow \text{CHECK-C2}(\text{FOO}(m, n), n, m, n)$ 
5      $t_2 \leftarrow \text{CHECK-C2}(m, \text{BAR}(m, n), m, n)$ 
6     return ( $t_1$  and  $t_2$ )
```

CHECK-C2( $m', n', m, n$ )  $\triangleright$  Check Conjecture 2

1 **if**  $(m', n') = (m, n)$ 
2 **then return** FALSE
3 **if**  $A[m', n']$ 
4 **then return** TRUE
5 **else**  $A[m', n'] \leftarrow \text{TRUE}$   $\triangleright$  Do work only if we have not visited  $(m', n')$ 
6  $t_1 \leftarrow \text{CHECK-C2}(\text{FOO}(m', n'), n', m, n)$ 
7  $t_2 \leftarrow \text{CHECK-C2}(m', \text{BAR}(m', n'), m, n)$ 
8 **return** ( $t_1$  and  $t_2$ )

As before, the running time is  $O((m + n)^6)$ , proportional to the size of the table.

- (d) Assuming Conjectures 1 and 2 hold, give an algorithm to compute  $H(m, n)$  with running time polynomial in  $m + n$ . What is the asymptotic running time of your algorithm?

**Solution:** As in parts (b) and (c), we use a recursive memoized algorithm to compute  $H(m, n)$ . By Conjecture 1, we know we only need a table of size  $(m+n)^3 \times (m+n)^3$  to store all  $H$  values we might need to compute  $H(m, n)$ . By Conjecture 2, we know that none of the  $H(m', n')$  are defined in terms of themselves. Thus, the following algorithm computes  $H(m, n)$ :

```

1 for  $i \leftarrow 1$  to  $(m+n)^3$        $\triangleright$  Initialize table
2     do for  $j \leftarrow 1$  to  $(m+n)^3$ 
3         do  $A[m, n] \leftarrow \text{NIL}$ 
4     return COMPUTE-H( $m, n$ )

```

```

    COMPUTE-H( $m', n'$ )       $\triangleright$  Compute  $H(m', n')$ 
1   if  $A[m', n'] = \text{NIL}$        $\triangleright$  Do work only if we have not computed  $H(m', n')$  before.
2     then    $v_1 \leftarrow \text{COMPUTE-H}(\text{FOO}(m', n'), n')$ 
3            $v_2 \leftarrow \text{COMPUTE-H}(m', \text{BAR}(m', n'))$ 
4            $A[m', n'] \leftarrow v_1 + v_2$ 
5   return  $A[m', n']$ 

```

As before, the running time is  $O((m+n)^6)$ , proportional to the size of the table.

**Problem 5. Cliquish Behavior [20 points] (4 parts)**

Prof. Vernon has come up with the following divide-and-conquer algorithm, BREAKFAST, for finding a clique in an undirected graph  $G = (V, E)$ :

1. Number the vertices in  $V$  as  $1, 2, \dots, n$ , where  $n = |V|$ .
2. If  $n = 1$ , return  $V$ .
3. Partition the vertices into the two sets  $V_1 = \{1, 2, \dots, \lfloor n/2 \rfloor\}$  and  $V_2 = \{\lfloor n/2 \rfloor + 1, \dots, n\}$ .
4. Let  $G_1$  be the subgraph of  $G$  induced by  $V_1$ , and similarly let  $G_2$  be the subgraph of  $G$  induced by  $V_2$ . (In other words, the edges of  $G_1$  are all edges of  $G$  that connect pairs of vertices in  $V_1$ , and the edges of  $G_2$  are those of  $G$  that connect pairs of vertices in  $V_2$ .)
5. Recursively find cliques  $C_1 = \text{BREAKFAST}(G_1)$  and  $C_2 = \text{BREAKFAST}(G_2)$ .
6. Combine these two cliques as follows:
  - Initialize  $C_1^+ \leftarrow C_1$  and  $C_2^+ \leftarrow C_2$ .
  - For every vertex  $v \in C_2$ , if  $v$  is adjacent to every vertex of  $C_1^+$ , then add  $v$  to  $C_1^+$ .
  - For every vertex  $u \in C_1$ , if  $u$  is adjacent to every vertex of  $C_2^+$ , then add  $u$  to  $C_2^+$ .
  - Return the larger of  $C_1^+$  and  $C_2^+$ .

- (a)** Briefly argue that the BREAKFAST algorithm always returns a clique of  $G$ .

**Solution:** We induct on  $n$ , the number of nodes in the graph.

**Base case:** For  $n = 1$ , the algorithm returns a single vertex, which is a clique.

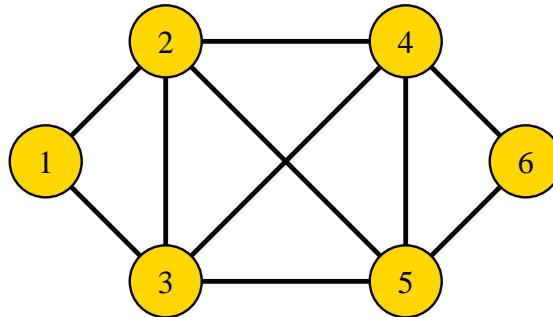
**Induction step:** Let  $n > 1$ , and assume the claim for all graphs with less than  $n$  vertices. Let  $G$  be a graph with  $n$  vertices. Clearly, both  $G_1$  and  $G_2$  have less than  $n$  vertices each. By the induction hypothesis,  $C_1$  and  $C_2$  are cliques of  $G_1$  and  $G_2$ , respectively; as a consequence, they are both cliques of  $G$  as well. Thus  $C_1^+$  and  $C_2^+$  are both initialized to cliques of  $G$  in Step 6. Now, a vertex  $v \in C_2$  is added to  $C_1^+$  if and only if  $v$  is adjacent to every vertex of  $C_1^+$ . Thus,  $C_1^+$  remains a clique of  $G$ . Similarly,  $C_2^+$  remains a clique of  $G$ . Because the algorithm returns either  $C_1^+$  or  $C_2^+$ , it always returns a clique of  $G$ .

- (b)** Give an asymptotically tight upper bound on the running time of the BREAKFAST algorithm.

**Solution:** The worst-case running time  $T(n)$  of the algorithm on an input graph with  $n$  nodes satisfies the recurrence  $T(n) = 2T(n/2) + \Theta(n^2)$  because the combination step in Step 6 takes  $\Theta(n^2)$  time in the worst case. By Case 3 of the Master Theorem,  $T(n) = \Theta(n^2)$ .

- (c) Give an example of a graph  $G$  where the algorithm produces a clique of less than maximum size.

**Solution:** There are many such counterexamples. Our favorite (found by many students) is the following:



The maximum clique in this graph is  $\{2, 3, 4, 5\}$ , but the BREAKFAST algorithm returns either  $\{1, 2, 3\}$  or  $\{4, 5, 6\}$ .

- (d) If the professor could modify algorithm BREAKFAST so as to find the largest clique without increasing the asymptotic running time, what would this tell you about the classes P and NP? Briefly explain your answer.

**Solution:** This would imply that  $P = NP$ . If the BREAKFAST algorithm could be modified to correctly find a maximum clique in  $O(n^2)$  time, we would have a polynomial-time algorithm for the CLIQUE decision problem: given a graph  $G$  and integer  $k$ , run the BREAKFAST algorithm to find the largest clique, and check whether the clique returned is of size  $\geq k$ . Because CLIQUE is NP-complete, this would imply that all NP problems can be solved in polynomial time, i.e., that  $P = NP$ .

---

## Final Exam Solutions

Problem	Parts	Points	Grade	Grader
1	16	80		
2	3	20		
3	4	30		
4	4	30		
5	4	20		
Total		180		

**Problem 1. True or False, and Justify [80 points] (16 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. The more content you provide in your justification, the higher your grade, but be brief. Your justification is worth more points than your true-or-false designation.

**T F** For every two positive functions  $f$  and  $g$ , if  $g(n) = O(n)$ , then  $f(g(n)) = O(f(n))$ .

**Solution: False.**

Let  $f(n) = 2^n$  and  $g(n) = 2n = O(n)$ . Then  $f(g(n)) = 2^{2n} = (2^n)^2 = (f(n))^2 \neq O(f(n))$ .

**T F** Suppose  $f(n) = 4 f(n/4) + n$  for  $n > 8$ , and  $f(n) = O(1)$  for  $n \leq 8$ . Similarly, suppose  $g(n) = 3 g(n/4) + n \lg n$  for  $n > 8$ , and  $g(n) = O(1)$  for  $n \leq 8$ . Then  $f(n) = \Theta(g(n))$ .

**Solution: True.**

By the Master Method (Cases 1 and 3), both are  $\Theta(n \lg n)$ .

- T F** Suppose that a randomized algorithm  $A$  has expected running time  $\Theta(n^2)$  on any input of size  $n$ . Then it is possible for some execution of  $A$  to take  $\Omega(2^n)$  time.

**Solution:** **True.**

Consider a (somewhat dumb) sorting algorithm that first sorts  $n$  items using mergesort, in time  $\Theta(n \lg n)$ , and then flips  $n$  (fair) coins. If at least one coin turns up heads, the algorithm waits  $n^2$  additional units of time before terminating. If all coins turn up tails, it waits  $2^n$  units of time. Although in the latter case the running time is  $\Omega(2^n)$ , the expected running time is only  $\Theta(n \lg n) + (1 - 2^{-n})(n^2) + 2^{-n}(2^n) = \Theta(n^2)$ .

- T F** Suppose we maintain a hash table with  $m$  slots using chaining and a hash function chosen from a universal hash family. If we insert  $n > m$  keys into this (initially empty) hash table, then the total number of collisions is  $O(n/m)$  in expectation. (Recall that a **collision** is a pair of distinct keys that hash to the same slot.)

**Solution:** **False.**

In expectation, any particular key  $x$  collides with  $\Theta(n/m)$  other keys. Thus, summing over all  $n$  keys, we have a total of  $\Theta(n^2/m) \neq O(n/m)$  collisions.

**T F** Building an  $n$ -element heap requires  $\Theta(n \lg n)$  time.

**Solution: False.**

It takes  $\Theta(n)$  time. See CLRS, pages 133–135.

**T F** Given an unsorted array  $A$  of  $n$  integers, let  $x_i$  denote the  $2^i$ th smallest element in  $A$ . Then

we can compute  $\sum_{i=0}^{\lfloor \lg n \rfloor} x_i$  in  $O(n)$  time.

**Solution: True.**

Initialize  $i \leftarrow \lfloor \lg n \rfloor$ ,  $\tilde{A} \leftarrow A$ , and  $s \leftarrow 0$ , and then repeat the following until  $i < 0$ : compute the  $2^i$ th element  $x_i$  of  $\tilde{A}$  using SELECT, set  $s \leftarrow s + x_i$ , update  $\tilde{A}$  to be those elements of  $\tilde{A}$  that are smaller than  $x_i$ , and update  $i$  to be  $i - 1$ . At the end, return  $s$ . Each iteration takes time linear in the size of  $\tilde{A}$ . In the first iteration (in which  $i = \lfloor \lg n \rfloor$ ),  $\tilde{A}$  has  $n$  elements, so this iteration takes  $c n$  time (for some appropriate constant  $c > 0$ ); in each subsequent iteration,  $\tilde{A}$  has  $2^{i+1} - 1$  elements, so the time taken is  $c(2^{i+1} - 1)$ . Thus the total time taken is

$$c n + \sum_{i=0}^{\lfloor \lg n \rfloor - 1} c(2^{i+1} - 1) = \Theta(n).$$

**T F** Suppose that you have a 2-3 tree  $T_1$  and AA-tree (from problem set 4)  $T_2$ , each storing the same set of keys. Then in-order traversals of  $T_1$  and  $T_2$  can result in different sequences of keys.

**Solution:** **False.**

Both 2-3 trees and AA trees are search trees. Thus, an in-order traversal of either produces the (same) sorted sequence of keys.

**T F** There are at least two distinct 2-3 trees containing keys 1, 2, 3, 4, 5.

**Solution:** **True.** E.g., root node could have either 2 or 3 children.

**T F** Graduating from MIT requires passing  $n$  specified classes. You decide to take each class every semester until you pass it. Suppose that, every semester you take a class, you have a 50% chance of passing it and a 50% chance of having to drop it. Then you will graduate in  $O(\lg n)$  semesters with high probability.

**Solution:** **True.**

For each of the  $n$  courses, the probability that do not pass the course after  $k \lg n$  semesters is  $(1/2)^{k \lg n} = 1/n^k$ . Therefore, the probability that there exists a course that you do not pass after  $k \lg n$  semesters is at most  $1/n^{k-1}$ . Thus you graduate with probability at least  $1 - 1/n^{k-1}$ . Choosing  $k$  (and thus  $k - 1$ ) to be an arbitrarily large constant, the number of semesters is  $O(\lg n)$  with high probability. (This derivation is essentially the same as Problem 2 on Problem Set 5.)

**T F** Suppose that you have two deterministic online algorithms,  $A_1$  and  $A_2$ , with competitive ratios  $c_1$  and  $c_2$ , respectively. Consider the randomized online algorithm  $A^*$  that flips a fair coin once at the beginning; if the coin comes up heads, it runs  $A_1$  from then on; if the coin comes up tails, it runs  $A_2$  from then on. Then the expected competitive ratio of  $A^*$  is at least  $\min\{c_1, c_2\}$ .

**Solution:** **False.**

Randomization can help a lot. In particular, the expected competitive ratio of  $A^*$  can be smaller than  $\min\{c_1, c_2\}$ . In class we have seen several examples of problems where this happens. For instance, see Problem 2 on Problem Set 6.

**T F** In a connected undirected graph  $G = (V, E, w)$  with nonnegative edge weights, the shortest-path tree from any source vertex  $s \in V$  is a minimum spanning tree of  $G$ . (Recall that the **shortest-path tree** from  $s$  consists of the edges  $\{(\pi(v), v) : v \in V - \{s\}\}$  where, for each  $v \in V - \{s\}$ ,  $(\pi(v), v)$  is the last relaxed incoming edge at  $v$  in an execution of Dijkstra's single-source shortest-paths algorithm from  $s$ .)

**Solution:** **False.**

Consider a graph with three nodes:  $s$ ,  $v$ , and  $t$ . Let  $w(s, v) = w(v, t) = 3$  and  $w(s, t) = 4$ . The shortest-path tree from  $s$  consists of two edges  $(s, v)$  and  $(s, t)$ , while the minimum spanning tree is  $\{(s, v), (v, t)\}$ .

**T F** Reweighting a graph with negative edge weights but no negative-weight cycles, as in Johnson's algorithm, can be used to solve the single-source shortest-paths problem more efficiently than Bellman-Ford.

**Solution: False.**

Johnson's reweighting uses the Bellman-Ford algorithm as a subroutine, to solve a system of difference constraints. Therefore Johnson's reweighting cannot be more efficient than Bellman-Ford.

**T F** Every problem in NP can be solved in exponential time.

**Solution: True.**

The brute-force search algorithm can solve every such problem in exponential time. This algorithm guesses a certificate  $y$  of polynomial length  $n^c$ , and runs the verification algorithm on the input  $x$  and certificate  $y$  in  $n^{c'}$  time. If the verification algorithm outputs 1 for any certificate, the brute-force algorithm returns 1; otherwise, it returns 0. The running time of this algorithm is  $O(2^{n^c} n^{c'})$ , which is  $O(2^{n^{c''}})$  for a suitable constant  $c''$ .

- T F** For any decision problem  $\pi$  in NP, define the input size  $n$  as the parameter  $k$  to fix. Then  $\pi$  is fixed-parameter tractable with respect to  $n$ .

**Solution:** **True.**

Recall that a parameterized problem is fixed-parameter tractable with respect to a parameter  $k$  if the problem can be solved in  $f(k) \cdot n^c$  time for some constant  $c$ . Clearly, problems in NP are FPT with respect to  $n$ , because by the previous part they can be solved in  $f(n) = O(2^{n^c})$  time for some constant  $c$ .

- T F** Define an *independent set* of a graph  $G = (V, E)$  to be a subset  $S \subseteq V$  of vertices such that  $V - S$  is a vertex cover of  $G$ . Every 2-approximation algorithm for finding a minimum vertex cover is also a 2-approximation algorithm for finding a maximum independent set.

**Solution:** **False.**

Let  $G$  be a cycle on six vertices. Clearly, both the maximum independent set and the minimum vertex cover of  $G$  are of size 3. A 2-approximation to minimum vertex cover may have size up to 6, but its complement will not necessarily be a 2-approximation to maximum independent set, having size as low as 0.

**Problem 2. One, One Room; Two, Two Rooms; Ah Ha Ha!** [20 points] (3 parts)

You are maintaining a hotel room reservation system for an  $n$ -day period, with dates labeled  $1, 2, \dots, n$ . Your reservation system must support two operations:

- $\text{RESERVE}(i, j)$  makes a room reservation for the dates  $i, i + 1, \dots, j$ .
- $\text{COUNT}(i)$  computes how many rooms are currently reserved on day  $i$ .

Your goal is to construct a data structure that supports both operations in  $O(\lg n)$  time. Assume that  $n$  is an exact power of two.

You decide to maintain your data in the form of a perfectly balanced binary tree. The root corresponds to the entire interval  $[1 \dots n]$ ; the root's left child corresponds to the interval  $[1 \dots n/2]$ ; the root's right child corresponds to the interval  $[(n/2 + 1) \dots n]$ ; etc. At the (bottom) leaf level, each leaf corresponds to a single day, and the leaf order matches the day order. Thus there are exactly  $n$  leaves and  $1 + \lg n$  levels.

- (a) What additional information would you maintain in the nodes in order to support the updates and queries efficiently?

**Solution:** For each node  $x$  representing some interval  $I_x$  of dates, we store a non-negative integer  $c[x]$ . This value  $c[x]$  counts the number of reservation intervals  $[i, j]$  that wholly contain  $I_x$  but do not wholly contain the parent interval  $I_{p[x]}$ .

- (b) Briefly describe how you would implement the  $\text{COUNT}(i)$  operation. Briefly justify why the running time is  $O(\lg n)$ .

**Solution:**  $\text{COUNT}(i)$  walks the path  $x_0, x_1, x_2, \dots, x_{\lg n}$  from the leaf  $x_0$  corresponding to date  $i$  up to the root  $x_{\lg n}$  of the tree. It returns the sum of the  $c$  values along this path:  $c[x_0] + c[x_1] + c[x_2] + \dots + c[x_{\lg n}]$ . This computation takes  $O(\lg n)$  time because we spend  $O(1)$  time at each node along the path of length  $1 + \lg n$ .

We claim that this sum equals the number of reservation intervals containing date  $i$ . To see this, consider a reservation interval  $I$  containing  $i$ . The intervals  $I_0, I_1, I_2, \dots, I_{\lg n}$  along the leaf-to-root path are nested:  $I_0 \subseteq I_1 \subseteq I_2 \subseteq \dots \subseteq I_{\lg n}$ . Hence, there is a unique  $k$  such that  $I$  wholly contains  $I_k$  but  $I$  does not wholly contain the parent interval  $I_{k+1}$ . Thus,  $c[x_k]$  counts  $I$ , and no other  $c[x_{k'}]$  counts  $I$ .

- (c) Briefly describe how you would implement the  $\text{RESERVE}(i, j)$  operation. Briefly justify why the running time is  $O(\lg n)$ .

**Solution:**  $\text{RESERVE}(i, j)$  simultaneously walks up the tree from the two leaves  $x_0$  and  $y_0$  corresponding to  $i$  and  $j$ , visiting nodes  $x_0, x_1, x_2, \dots$  and  $y_0, y_1, y_2, \dots$ , until the two paths meet at some node  $x_k = y_k$ , the lowest common ancestor of  $x_0$  and  $y_0$ . We define a **test** operation on a node  $x$ : if  $[i, j]$  wholly contains  $x$ 's interval but not  $x$ 's parent's interval, then we increment  $c[x]$ . Then  $\text{RESERVE}(i, j)$  tests  $x_k = y_k$ , and tests each child of every node  $x_0, y_0, x_1, y_1, \dots, x_{k-1}, y_{k-1}, x_k = y_k$ . It is easy to see that these  $O(\lg n)$  nodes are all that we need to test, and that the running time is thus  $O(\lg n)$ .

**Problem 3. Forty Two** [30 points] (4 parts)

Professor Hackermann has finally cracked the meaning of life, the universe, and everything: it is  $H(42, 42)$  where the function  $H(m, n)$  is defined by the following unusual recurrence:

$$\begin{aligned} H(m, 1) &= m^2; \\ H(1, n) &= n^3; \\ H(m, n) &= H(\text{FOO}(m, n), n) + H(m, \text{BAR}(m, n)) \quad \text{for all other values of } m, n \geq 1. \end{aligned}$$

Professor Hackermann knows how to compute  $\text{FOO}(m, n)$  and  $\text{BAR}(m, n)$  in  $O(1)$  time for given values of  $m, n \geq 1$ . The catch is that  $\text{FOO}(m, n)$  and  $\text{BAR}(m, n)$  can sometimes be larger than  $m$  and  $n$ , so the recurrence does not obviously terminate. Nonetheless, both  $\text{FOO}(m, n)$  and  $\text{BAR}(m, n)$  have value 1 often enough that the recursive formula may allow computing  $H(m, n)$ .

- (a) The professor hires you to compute  $H(3, 4)$  by hand, using the following information about  $\text{FOO}$  and  $\text{BAR}$ :

$$\begin{array}{lll} \text{FOO}(3, 4) & = & 7; & \text{BAR}(3, 4) & = & 1; \\ \text{FOO}(7, 4) & = & 10; & \text{BAR}(7, 4) & = & 2; \\ \text{FOO}(10, 4) & = & 1; & \text{BAR}(10, 4) & = & 1; \\ \text{FOO}(7, 2) & = & 1; & \text{BAR}(7, 2) & = & 1. \end{array}$$

Show your work.

**Solution:**

$$\begin{aligned} H(7, 2) &= H(1, 2) + H(7, 1) = 8 + 49 = 57, \\ H(10, 4) &= H(1, 4) + H(10, 1) = 64 + 100 = 164, \\ H(7, 4) &= H(10, 4) + H(7, 2) = 164 + 57 = 221, \\ H(3, 4) &= H(7, 4) + H(3, 1) = 221 + 9 = 230. \end{aligned}$$

To understand whether  $H(42, 42)$  can be computed with the recursive formula, Professor Hackermann sets out to understand which pairs  $(m', n')$  arise from the recursion. The professor defines  $\text{descendants}(m, n)$  to be the set of pairs  $(m', n')$  for which  $H(m', n')$  is required to compute  $H(m, n)$ , i.e.,

$$\begin{aligned}\text{descendants}(m, n) &= \left\{ \left( \text{FOO}(m, n), n' \right), \left( m, \text{BAR}(m, n) \right) \right\} \\ &\cup \text{descendants}\left( \text{FOO}(m, n), n' \right) \cup \text{descendants}\left( m, \text{BAR}(m, n) \right).\end{aligned}$$

Note that  $\text{descendants}(m, n)$  does not necessarily include  $(m, n)$ .

- (b) To thwart critics who claim that pairs  $(m', n')$  in  $\text{descendants}(m, n)$  can grow without bound, Professor Hackermann makes the following conjecture:

**Conjecture 1** For every  $m, n \geq 1$ , and for every pair  $(m', n') \in \text{descendants}(m, n)$ , we have both  $m' \leq (m + n)^3$  and  $n' \leq (m + n)^3$ .

Give an algorithm that, on input  $m, n \geq 1$ , determines whether Conjecture 1 is true for this pair of integers, i.e., whether  $m', n' \leq (m + n)^3$  for every pair  $(m', n') \in \text{descendants}(m, n)$ . Your algorithm must run in time polynomial in  $m + n$ .

**Solution:** To verify Conjecture 1 for a given pair of integers  $m$  and  $n$ , we recursively compute the pairs that belong to  $\text{descendants}(m, n)$ , using an  $(m+n)^3 \times (m+n)^3$  table  $A$  to track the pairs  $(m', n')$  we have already visited. When expanding the recursive definition of  $\text{descendants}(m, n)$ , if we ever discover a pair  $(m', n')$  out of range, (i.e., either  $m' > (m + n)^3$  or  $n' > (m + n)^3$ ), we know that Conjecture 1 is false. On the other hand, if all elements we check are in range, then Conjecture 1 holds for  $(m, n)$ . We fill in the table recursively, but use memoization to avoid recomputing repeated subproblems.

```

1 for  $i \leftarrow 1$  to  $(m + n)^3$        $\triangleright$  Initialize table
2   do for  $j \leftarrow 1$  to  $(m + n)^3$ 
3     do  $A[m, n] \leftarrow \text{FALSE}$ 
4 return CHECK-C1( $m, n, m, n$ )

```

```

CHECK-C1( $m', n', m, n$ )       $\triangleright$  Check Conjecture 1
1   if  $m' > (m + n)^3$  or  $n' > (m + n)^3$ 
2     then return FALSE
3   if  $A[m', n']$ 
4     then return TRUE
5   else  $A[m', n'] \leftarrow \text{TRUE}$        $\triangleright$  Do work only if we have not visited  $(m', n')$ 
6      $t_1 \leftarrow \text{CHECK-C1}(\text{FOO}(m', n'), n', m, n)$ 
7      $t_2 \leftarrow \text{CHECK-C1}(m', \text{BAR}(m', n'), m, n)$ 
8   return ( $t_1$  and  $t_2$ )

```

The amortized cost to fill any single element in the table  $A$  is  $O(1)$ , because we can charge the cost of making recursive calls (Lines 7–9) and the cost of checking the base cases (Lines 1–4) to each table element of  $A$  that is filled in (Line 6). Because we have  $(m + n)^6$  table elements, the running time of the algorithm is  $O((m + n)^6)$ .

- (c) More critics claim that the professor's recursive formula is useless because it could be cyclic: the computation of  $H(m, n)$  could require the computation of  $H(m, n)$  itself, leading to an infinite recursion. To thwart these critics, Professor Hackermann makes another conjecture:

**Conjecture 2** For every  $m, n \geq 1$ , we have  $(m, n) \notin \text{descendants}(m, n)$ .

Give an algorithm that, on input  $m, n \geq 1$ , determines whether Conjecture 2 is true for this pair of integers, i.e., whether  $(m, n) \notin \text{descendants}(m, n)$ . Your algorithm must run in time polynomial in  $m + n$ , and it may assume that Conjecture 1 holds.

**Solution:** Assuming Conjecture 1 holds, all pairs  $(m', n') \in \text{descendants}(m, n)$  are represented in the  $(m+n)^3 \times (m+n)^3$  table  $A$ . Thus, we can use a memoized recursive algorithm similar to the one in part (b), except with the added check for whether we encounter  $m' = m$  and  $n' = n$  after the first step.

```

1 for  $i \leftarrow 1$  to  $(m + n)^3$        $\triangleright$  Initialize table
2     do for  $j \leftarrow 1$  to  $(m + n)^3$ 
3         do  $A[m, n] \leftarrow \text{FALSE}$ 
4      $t_1 \leftarrow \text{CHECK-C2}(\text{FOO}(m, n), n, m, n)$ 
5      $t_2 \leftarrow \text{CHECK-C2}(m, \text{BAR}(m, n), m, n)$ 
6     return ( $t_1$  and  $t_2$ )
```

CHECK-C2( $m', n', m, n$ )  $\triangleright$  Check Conjecture 2

1 **if**  $(m', n') = (m, n)$ 
2 **then return** FALSE
3 **if**  $A[m', n']$ 
4 **then return** TRUE
5 **else**  $A[m', n'] \leftarrow \text{TRUE}$   $\triangleright$  Do work only if we have not visited  $(m', n')$ 
6  $t_1 \leftarrow \text{CHECK-C2}(\text{FOO}(m', n'), n', m, n)$ 
7  $t_2 \leftarrow \text{CHECK-C2}(m', \text{BAR}(m', n'), m, n)$ 
8 **return** ( $t_1$  and  $t_2$ )

As before, the running time is  $O((m + n)^6)$ , proportional to the size of the table.

- (d) Assuming Conjectures 1 and 2 hold, give an algorithm to compute  $H(m, n)$  with running time polynomial in  $m + n$ . What is the asymptotic running time of your algorithm?

**Solution:** As in parts (b) and (c), we use a recursive memoized algorithm to compute  $H(m, n)$ . By Conjecture 1, we know we only need a table of size  $(m+n)^3 \times (m+n)^3$  to store all  $H$  values we might need to compute  $H(m, n)$ . By Conjecture 2, we know that none of the  $H(m', n')$  are defined in terms of themselves. Thus, the following algorithm computes  $H(m, n)$ :

```

1 for  $i \leftarrow 1$  to  $(m+n)^3$        $\triangleright$  Initialize table
2     do for  $j \leftarrow 1$  to  $(m+n)^3$ 
3         do  $A[m, n] \leftarrow \text{NIL}$ 
4     return COMPUTE-H( $m, n$ )

```

```

    COMPUTE-H( $m', n'$ )       $\triangleright$  Compute  $H(m', n')$ 
1   if  $A[m', n'] = \text{NIL}$        $\triangleright$  Do work only if we have not computed  $H(m', n')$  before.
2       then  $v_1 \leftarrow \text{COMPUTE-H}(\text{FOO}(m', n'), n')$ 
3            $v_2 \leftarrow \text{COMPUTE-H}(m', \text{BAR}(m', n'))$ 
4            $A[m', n'] \leftarrow v_1 + v_2$ 
5   return  $A[m', n']$ 

```

As before, the running time is  $O((m+n)^6)$ , proportional to the size of the table.

**Problem 4. Cliquish Behavior [20 points] (4 parts)**

Prof. Vernon has come up with the following divide-and-conquer algorithm, BREAKFAST, for finding a clique<sup>1</sup> in an undirected graph  $G = (V, E)$ :

1. Number the vertices in  $V$  as  $1, 2, \dots, n$ , where  $n = |V|$ .
2. If  $n = 1$ , return  $V$ .
3. Partition the vertices into the two sets  $V_1 = \{1, 2, \dots, \lfloor n/2 \rfloor\}$  and  $V_2 = \{\lfloor n/2 \rfloor + 1, \dots, n\}$ .
4. Let  $G_1$  be the subgraph of  $G$  induced by  $V_1$ , and similarly let  $G_2$  be the subgraph of  $G$  induced by  $V_2$ . (In other words, the edges of  $G_1$  are all edges of  $G$  that connect pairs of vertices in  $V_1$ , and the edges of  $G_2$  are those of  $G$  that connect pairs of vertices in  $V_2$ .)
5. Recursively find cliques  $C_1 = \text{BREAKFAST}(G_1)$  and  $C_2 = \text{BREAKFAST}(G_2)$ .
6. Combine these two cliques as follows:
  - Initialize  $C_1^+ \leftarrow C_1$  and  $C_2^+ \leftarrow C_2$ .
  - For every vertex  $v \in C_2$ , if  $v$  is adjacent to every vertex of  $C_1^+$ , then add  $v$  to  $C_1^+$ .
  - For every vertex  $u \in C_1$ , if  $u$  is adjacent to every vertex of  $C_2^+$ , then add  $u$  to  $C_2^+$ .
  - Return the larger of  $C_1^+$  and  $C_2^+$ .

- (a) Briefly argue that the BREAKFAST algorithm always returns a clique of  $G$ .

**Solution:** We induct on  $n$ , the number of nodes in the graph.

**Base case:** For  $n = 1$ , the algorithm returns a single vertex, which is a clique.

**Induction step:** Let  $n > 1$ , and assume the claim for all graphs with less than  $n$  vertices. Let  $G$  be a graph with  $n$  vertices. Clearly, both  $G_1$  and  $G_2$  have less than  $n$  vertices each. By the induction hypothesis,  $C_1$  and  $C_2$  are cliques of  $G_1$  and  $G_2$ , respectively; as a consequence, they are both cliques of  $G$  as well. Thus  $C_1^+$  and  $C_2^+$  are both initialized to cliques of  $G$  in Step 6. Now, a vertex  $v \in C_2$  is added to  $C_1^+$  if and only if  $v$  is adjacent to every vertex of  $C_1^+$ . Thus,  $C_1^+$  remains a clique of  $G$ . Similarly,  $C_2^+$  remains a clique of  $G$ . Because the algorithm returns either  $C_1^+$  or  $C_2^+$ , it always returns a clique of  $G$ .

- (b) Give an asymptotically tight upper bound on the running time of the BREAKFAST algorithm.

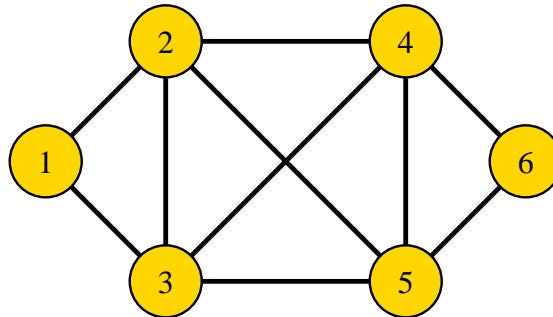
**Solution:** The worst-case running time  $T(n)$  of the algorithm on an input graph with  $n$  nodes satisfies the recurrence  $T(n) = 2T(n/2) + \Theta(n^2)$  because the combination step in Step 6 takes  $\Theta(n^2)$  time in the worst case. By Case 3 of the Master Theorem,  $T(n) = \Theta(n^2)$ .

---

<sup>1</sup>For a graph  $G$ , a clique  $C \subset V$  is a subset of vertices that are all interconnected by edges.

- (c) Give an example of a graph  $G$  where the algorithm produces a clique of less than maximum size.

**Solution:** There are many such counterexamples. Our favorite (found by many students) is the following:



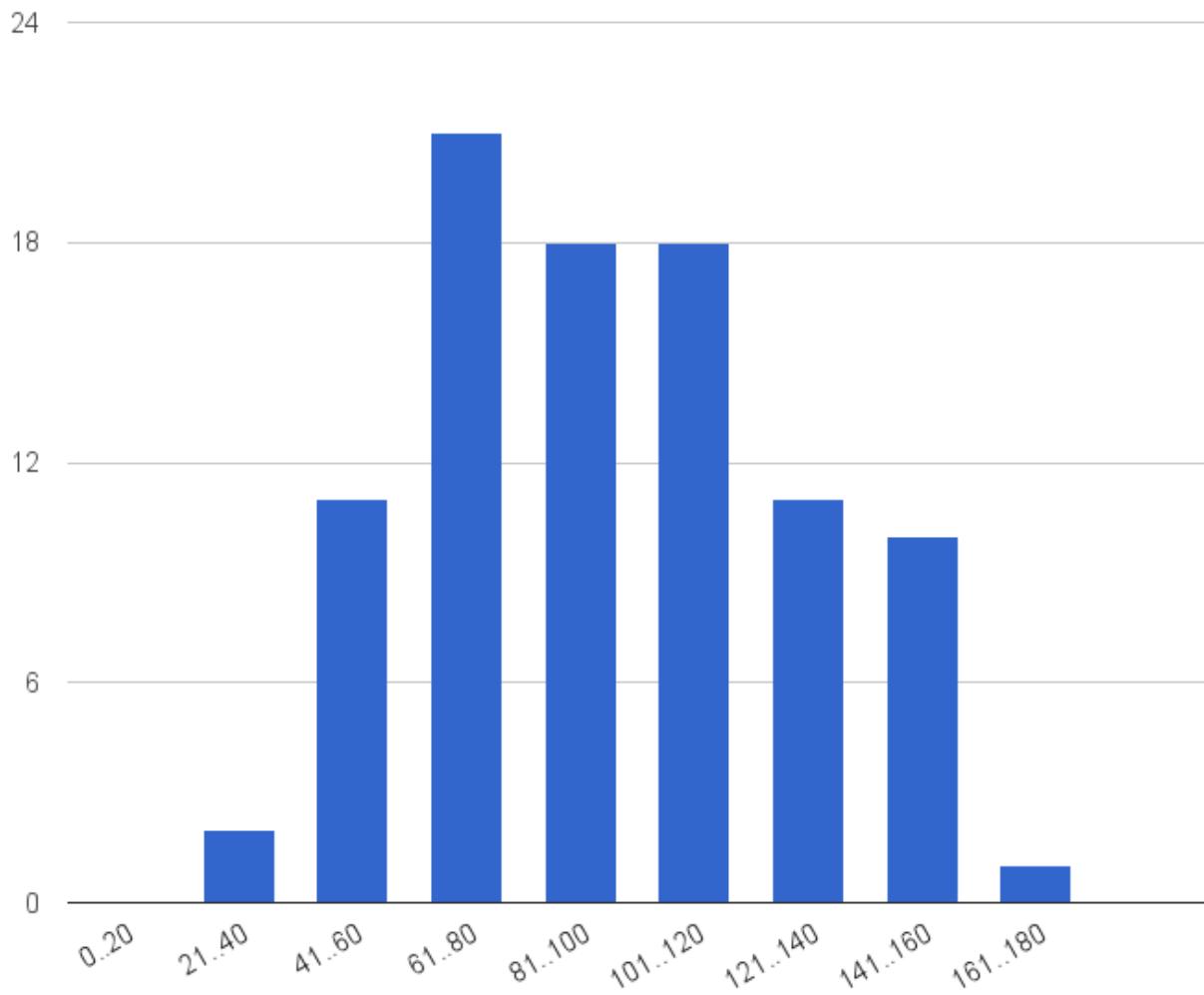
The maximum clique in this graph is  $\{2, 3, 4, 5\}$ , but the BREAKFAST algorithm returns either  $\{1, 2, 3\}$  or  $\{4, 5, 6\}$ .

- (d) If the professor could modify algorithm BREAKFAST so as to find the largest clique without increasing the asymptotic running time, what would this tell you about the classes P and NP? Briefly explain your answer.

**Solution:** This would imply that  $P = NP$ . If the BREAKFAST algorithm could be modified to correctly find a maximum clique in  $O(n^2)$  time, we would have a polynomial-time algorithm for the CLIQUE decision problem: given a graph  $G$  and integer  $k$ , run the BREAKFAST algorithm to find the largest clique, and check whether the clique returned is of size  $\geq k$ . Because CLIQUE is NP-complete, this would imply that all NP problems can be solved in polynomial time, i.e., that  $P = NP$ .

## Final Exam

**Final Exam Grades**



**Problem 0. Name.** [1 point] (19 parts)

Write your name on every page of this exam booklet! Don't forget the cover.

**Problem 1. True or False.** [40 points] (10 parts)

Circle **T** or **F** for each of the following statements to indicate whether the statement is true or false, respectively, and briefly explain why (one or two sentences). Your justification is worth more points than your true-or-false designation. *Careful:* Some problems are straightforward, but some are tricky!

- (a) **T F** Suppose that every operation on a data structure runs in  $O(1)$  amortized time. Then the running time for performing a sequence of  $n$  operations on the data structure is  $O(n)$  in the worst case.

**Solution:** True:  $\sum_{i=1}^n \hat{c}_i \geq \sum_{i=1}^n c_i$ , assuming one starts from an empty data structure.

- (b) **T F** Suppose that a Las Vegas algorithm has expected running time  $\Theta(n)$  on inputs of size  $n$ . Then there may still be an input on which it always runs in time  $\Omega(n \lg n)$ .

**Solution:** False. Definition of Las Vegas means that it runs in  $\Theta(n)$  expected time for *any* input.

- (c) **T F** If there is a randomized algorithm that solves a decision problem in time  $t$  and outputs the correct answer with probability 0.5, then there is a randomized algorithm for the problem that runs in time  $\Theta(t)$  and outputs the correct answer with probability at least 0.99.

**Solution:** False. Every decision problem has an algorithm that produces the correct answer with probability 0.5 just by flipping a coin to determine the answer.

- (d) **T F** Let  $\mathcal{H} : \{0, 1\}^n \rightarrow \{1, 2, \dots, k\}$  be a universal family of hash functions, and let  $S \subseteq \{0, 1\}^n$  be a set of  $|S| = k$  elements. For  $h$  chosen at random from  $\mathcal{H}$ , let  $E$  be the event that for all  $y \in \{1, 2, \dots, k\}$ , the number of elements in  $S$  hashed to  $y$  is at most 100, that is,  $|h^{-1}(y) \cap S| \leq 100$ . Then we have  $\Pr\{E\} \geq 3/4$ .

**Solution:** False.  $|h^{-1}(y) \cap S|$  is likely to be  $\Theta(\log k / \log \log k)$  for some  $y$ . Only its expectation is  $O(1)$ .

- (e) **T F** Let  $\Sigma = \{a, b, c, \dots, z\}$  be a 26-letter alphabet, and let  $s \in \Sigma^n$  and  $p \in \Sigma^m$  be strings of length  $n$  and  $m < n$  respectively. Then there is a  $\Theta(n)$ -time algorithm to check whether  $p$  is a substring of  $s$ .

**Solution:** True. E.g., using suffix trees.

- (f) **T F** If an iteration of the Ford-Fulkerson algorithm on a network places flow 1 through an edge  $(u, v)$ , then in every later iteration, the flow through  $(u, v)$  is at least 1.

**Solution:** False: A later augmenting path may pass through  $(v, u)$ , causing the flow on  $(u, v)$  to be decreased.

- (g) **T F** Consider the linear programming problem of finding a vector  $x$  of  $n$  unknowns maximizing  $c^T x$  such that  $Ax \leq b$ , where  $A$  is an  $m \times n$  matrix,  $c$  is a vector of length  $n$ , and  $b$  is a vector of length  $m$ . Given a feasible solution for this program, one can check whether the solution is optimal in  $O(mn)$  time.

**Solution:** True: The assignment to the slack variables produces a solution that must be feasible for the dual.

- (h) **T F** Consider a linear program having  $m$  constraints on  $n$  unknowns  $x_1, x_2, \dots, x_n$ , where each constraint is of the form  $x_i - x_j \leq c_{ij}$ , where each  $c_{ij}$  is a constant. Suppose that this linear program is feasible. Then a feasible assignment to the  $x_i$  can be found in  $O(mn)$  time.

**Solution:** True: Using Bellman-Ford to solve a system of difference constraints.

- (i) **T F** There exists a minimization problem such that (i) assuming  $P \neq NP$ , there is no polynomial-time 1-approximation algorithm for the problem; and (ii) for any constant  $\epsilon > 0$ , there is a polynomial-time  $(1 + \epsilon)$ -approximation algorithm for the problem.

**Solution:** True. There are NP-hard optimization problems with a PTAS, such as PARTITION, as we saw in class.

- (j) **T F** A particular  $n$ -input comparison network  $N$  is alleged to be capable of sorting bitonic sequences. There exists a set of just  $O(n^2)$  different bitonic sequences, each of size  $n$ , which, if  $N$  sorts them correctly, guarantees that  $N$  can indeed sort any bitonic sequence of length  $n$ .

**Solution:** True. Using the 0-1 Principle, we only have to test for bitonic 0-1 strings, which consists of strings of the form  $0^i 1^j 0^k$  or of the form  $1^i 0^j 1^k$ , where  $i + j + k = n$ . There are  $O(n^2)$  such strings.

**Problem 2. Substitution Method.** [14 points]

Use the substitution method to show that the recurrence

$$T(n) = \sqrt{n} T(\sqrt{n}) + n$$

has solution  $T(n) = O(n \lg \lg n)$ .

**Solution:** Assume  $T(k) \leq ck \lg \lg k$  for some  $c > 0$  for all  $n_0 \leq k < n$ . Then:

$$T(n) = \sqrt{n} T(\sqrt{n}) + n \leq \sqrt{n} c \sqrt{n} \lg \lg \sqrt{n} + n$$

We now find  $c > 0$  so that

$$\begin{aligned} \sqrt{n} c \sqrt{n} \lg \lg \sqrt{n} + n &\leq cn \lg \lg n \\ nc(\lg \lg n - \lg 2) + n &\leq cn \lg \lg n \\ -c \lg 2 + 1 &\leq 0 \\ 1 &\leq c \end{aligned}$$

Hence, for any  $c \geq 1$ , we have  $T(n) \leq cn \lg \lg n$  for all  $n \geq n_0$ , provided that  $T(n_0) \leq cn_0 \lg \lg n_0$ . Checking  $n_0 = 4$  as the base case, we need  $c \geq \min\{T(4), 1\}$ .

**Problem 3. Updating a Flow Network [25 points] (3 parts)**

Let  $G = (V, E)$  be a flow network with source  $s$  and sink  $t$ , and suppose that each edge  $e \in E$  has capacity  $c(e) = 1$ . Assume also, for convenience, that  $|E| = \Omega(V)$ .

- (a) Suppose that we implement the Ford-Fulkerson maximum-flow algorithm by using depth-first search to find augmenting paths in the residual graph. What is the worst-case running time of this algorithm on  $G$ ?

**Solution:** Since the capacity out of the source is  $|V| - 1$ , a mincut has value at most  $|V| - 1$ . Thus the running time is  $O(VE)$ .

- (b) Suppose that a maximum flow for  $G$  has been computed, and a new edge with unit capacity is added to  $E$ . Describe how the maximum flow can be efficiently updated. (*Note:* It is not the value of the flow that must be updated, but the flow itself.) Analyze your algorithm.

**Solution:** (Assume that the flow on every edge is integer.) Simply run one more BFS to find one augmenting path in the residual flow network. This costs  $O(E)$  time. One path suffices because the max flow value cannot increase by more than the capacity of the added edge (see min cut). Since all edges have capacity 1, any augmenting path will have capacity 1 as well. If no augmenting path exists, the flow remains unchanged.

- (c) Suppose that a maximum flow for  $G$  has been computed, but an edge is now removed from  $E$ . Describe how the maximum flow can be efficiently updated. Analyze your algorithm.

**Solution:** (Once again, assume that the flow on every edge is integer.) If the removed edge is  $(u, v)$  and  $f(u, v) = 0$ , do nothing. In the residual flow network, find a path from sink to source via residual of the edge to be removed,  $(v, u)$ . To do so, find a path from sink to  $v$  and a second segment from  $u$  to source. Then diminish the flow by sending one unit along the path. Since all edges have capacity 1, this removes the flow from the edge to be removed. After removing the edge, search for augmenting path in the residual flow network. This is needed in case the edge was not in all minimum cuts hence the flow can be redirected. One augmentation is enough because the value of max flow cannot increase beyond the original value. The total cost is  $O(E)$ .

**Problem 4. Candy Land.** [20 points]

The Candy Corporation of Candy Land owns 3 candy factories  $f_1, f_2$ , and  $f_3$  that manufacture 20 kinds of candy  $c_1, c_2, \dots, c_{20}$ . Each factory  $f_i$  manufactures  $m_{ij}$  pounds of each candy  $c_j$ . The Candy Corporation has 103 local stores  $s_1, s_2, \dots, s_{103}$  that sell candy. Each store  $s_k$  requires  $d_{kj}$  pounds of candy  $c_j$ . The factories only produce what is needed, and hence we have

$$\sum_{i=1}^3 \sum_{j=1}^{20} m_{ij} = \sum_{k=1}^{103} \sum_{j=1}^{20} d_{kj} .$$

A directed graph  $G = (V, E)$  represents the road map of Candy Land, where  $V$  represents road intersections and  $E$  represents the roads themselves. Factories and stores are located at intersections:  $f_i \in V$  for  $i = 1, 2, 3$  and  $s_k \in V$  for  $k = 1, 2, \dots, 103$ . King Kandy's tax collector, Lord Licorice, has designated a tax rate  $c(e)$  for each road  $e \in E$ : if  $x$  pounds of candy (of any kind) are shipped along road  $e$ , Lord Licorice must be paid  $x \cdot c(e)$  gumdrops.

Princess Lolly, the supply manager of the Candy Corporation, wants to determine how to deliver the candy from the factories to the local stores while minimizing the tax paid to Lord Licorice. Give a linear-programming formulation of this problem.

**Solution:**  $\forall j \in \{1, \dots, 20\}$  and  $\forall (u, v) \in E$ , let  $x_j(u, v)$  be the amount of candy  $c_j$  delivered through road  $(u, v) \in E$ . Let  $N(v) = \{u | (u, v) \in E\}$  denote the neighborhood of node  $v$ . The delivery of candy is optimized by the linear program

$$\text{minimize} \quad \sum_{(u,v) \in E} x_j(u, v) c(u, v) \quad (\text{total tax})$$

subject to

$$\forall j \in \{1, \dots, 20\}, \forall v \in V,$$

$$\sum_{u \in N(v)} x_j(v, u) - x_j(u, v) = \begin{cases} m_{ij} & : v = f_i \in \{f_1, f_2, f_3\} \\ -d_{kj} & : v = s_k \in \{s_1, \dots, s_{103}\} \\ 0 & : \text{otherwise} \end{cases} \quad \begin{array}{l} (\text{supply constraints}) \\ (\text{demand constraints}) \\ (\text{no buildup of candy}) \end{array}$$

$$\forall j \in \{1, \dots, 20\}, \forall u, v \in V,$$

$$x_j(u, v) \geq 0$$

**Problem 5. Combined-Signal Problem.** [25 points] (4 parts)

Consider a square  $m \times m$  region containing  $n$  radio-transmitting stations at integer grid points. For  $i = 1, 2, \dots, n$ , station  $i$  is located at grid point  $(x_i, y_i)$ , where  $x_i$  and  $y_i$  are integers in the range  $0 \leq x_i, y_i < m$ . Station  $i$  emits a signal of strength  $s_i$ . For each grid point  $(x, y)$  not occupied by a station, the combined signal received at  $(x, y)$  is

$$S(x, y) = \sum_{i=1}^n \frac{s_i}{(x - x_i)^2 + (y - y_i)^2}.$$

The **combined-signal problem** is to find, for each of the  $m^2 - n$  points on the grid that are not occupied by a station, the signal received at that point.

- (a) Describe a simple  $O(m^2n)$  time algorithm to solve the combined-signal problem.

**Solution:** Use the given formula.

- (b)** Let  $\mathbb{Z}_{2m-1} = \{0, 1, \dots, 2m-2\}$ . Find two functions  $f, g : \mathbb{Z}_{2m-1} \times \mathbb{Z}_{2m-1} \rightarrow \mathbb{R}$  such that the signal at  $(x, y)$  equals the convolution of  $f$  and  $g$ :

$$\begin{aligned} S(x, y) &= (f \otimes g)(x, y) \\ &= \sum_{x'=0}^{2m-2} \sum_{y'=0}^{2m-2} f(x', y') \cdot g(x - x', y - y') . \end{aligned}$$

Importantly, in this definition  $x - x'$  and  $y - y'$  are computed in the additive group  $\mathbb{Z}_{2m-1}$ , which is a fancy way of saying they are computed modulo  $2m - 1$ .

**Solution:** Let  $f(x, y) = s_i$ , if  $(x, y)$  is occupied by the  $i$ th station for  $1 \leq i \leq n$ , and 0 otherwise. Define  $h : \mathbb{Z}_{2m-1} \rightarrow \mathbb{R}$  as follows: for  $0 \leq x \leq m - 1$ , let  $h(x) = x$ ; for  $1 \leq x \leq m - 1$ , let  $h(2m - 1 - x) = x$ . Let  $g(x, y) = 1/(h(x)^2 + h(y)^2)$ .

- (c) The **discrete Fourier transform** of a function  $h : \mathbb{Z}_{2m-1} \times \mathbb{Z}_{2m-1} \rightarrow \mathbb{R}$  is the function  $\widehat{h} : \mathbb{Z}_{2m-1} \times \mathbb{Z}_{2m-1} \rightarrow \mathbb{C}$  defined as follows:

$$\widehat{h}(a, b) = \frac{1}{(2m-1)^2} \sum_{x=0}^{2m-2} \sum_{y=0}^{2m-2} h(x, y) \omega_{2m-1}^{-ax-by},$$

where  $\omega_{2m-1}$  is a  $(2m-1)$ th root of unity. Argue that for any two functions  $f, g : \mathbb{Z}_{2m-1} \times \mathbb{Z}_{2m-1} \rightarrow \mathbb{R}$  and for any point  $(a, b) \in \mathbb{Z}_{2m-1} \times \mathbb{Z}_{2m-1}$ , we have

$$\widehat{(f \otimes g)}(a, b) = (2m-1)^2 \cdot \widehat{f}(a, b) \cdot \widehat{g}(a, b).$$

**Solution:** Note that for every fixed  $0 \leq x' \leq 2m-2$ , if we take  $x$  to go from 0 to  $2m-2$ , we get that  $(x - x' \bmod 2m-1)$  goes over all numbers between 0 and  $2m-2$ . Therefore, for any  $0 \leq x', y' \leq 2m-2$  we can write:

$$\widehat{g}(a, b) = \frac{1}{(2m-1)^2} \sum_{x=0}^{2m-2} \sum_{y=0}^{2m-2} g(x - x', y - y') \omega_{2m-1}^{-a(x-x')-b(y-y')}.$$
 (1)

We write:

$$\begin{aligned} \widehat{f}(a, b) \cdot \widehat{g}(a, b) &= \left( \frac{1}{(2m-1)^2} \sum_{x'=0}^{2m-2} \sum_{y'=0}^{2m-2} f(x', y') \omega_{2m-1}^{-ax'-by'} \right) \cdot \widehat{g}(a, b) \\ &= \frac{1}{(2m-1)^2} \sum_{x'=0}^{2m-2} \sum_{y'=0}^{2m-2} f(x', y') \omega_{2m-1}^{-ax'-by'} \widehat{g}(a, b) \end{aligned}$$

Substituting equality (1),

$$= \frac{1}{(2m-1)^4} \sum_{x'=0}^{2m-2} \sum_{y'=0}^{2m-2} f(x', y') \omega_{2m-1}^{-ax'-by'} \left( \sum_{x=0}^{2m-2} \sum_{y=0}^{2m-2} g(x - x', y - y') \omega_{2m-1}^{-a(x-x')-b(y-y')} \right)$$

Re-arranging,

$$\begin{aligned} &= \frac{1}{(2m-1)^4} \sum_{x=0}^{2m-2} \sum_{y=0}^{2m-2} \sum_{x'=0}^{2m-2} \sum_{y'=0}^{2m-2} f(x', y') g(x - x', y - y') \omega_{2m-1}^{-ax'-by'} \omega_{2m-1}^{-a(x-x')-b(y-y')} \\ &= \frac{1}{(2m-1)^4} \sum_{x=0}^{2m-2} \sum_{y=0}^{2m-2} \sum_{x'=0}^{2m-2} \sum_{y'=0}^{2m-2} f(x', y') g(x - x', y - y') \omega_{2m-1}^{-ax-by} \\ &= \frac{1}{(2m-1)^4} \sum_{x=0}^{2m-2} \sum_{y=0}^{2m-2} \left( \sum_{x'=0}^{2m-2} \sum_{y'=0}^{2m-2} f(x', y') g(x - x', y - y') \right) \omega_{2m-1}^{-ax-by} \\ &= \frac{1}{(2m-1)^4} \sum_{x=0}^{2m-2} \sum_{y=0}^{2m-2} (f \otimes g)(x, y) \omega_{2m-1}^{-ax-by} \\ &= \frac{1}{(2m-1)^2} \widehat{(f \otimes g)}(a, b) \end{aligned}$$

- (d) Describe an  $O(m^2 \lg m)$ -time algorithm to solve the combined-signal problem.

**Solution:** The algorithm is as follows:

1. Run the FFT algorithm to compute  $\hat{f}$  and  $\hat{g}$ . This takes  $\Theta(m^2 \lg m)$  time.
2. Compute  $(\widehat{f \otimes g})$  by pointwise multiplication. This takes  $\Theta(m^2)$  time.
3. Run the FFT algorithm to compute  $(f \otimes g)$ . This takes  $\Theta(m^2 \lg m)$  time.

**Problem 6. Eleanor's Problem Revisited.** [25 points] (4 parts)

Recall the following problem abstracted from the take-home quiz, which we shall refer to as **Eleanor's optimization problem**. Consider a directed graph  $G = (V, E)$  in which every edge  $e \in E$  has a length  $\ell(e) \in \mathbb{Z}$  and a cost  $c(e) \in \mathbb{Z}$ . Given a source  $s \in V$ , a destination  $t \in V$ , and a cost  $x$ , find the shortest path in  $G$  from  $s$  to  $t$  whose total cost is at most  $x$ .

We can reformulate Eleanor's optimization problem as a decision problem. **Eleanor's decision problem** has the same inputs as Eleanor's optimization problem, as well as a distance  $d$ . The problem is to determine if there exists a path in  $G$  from  $s$  to  $t$  whose total cost is at most  $x$  and whose length is at most  $d$ .

- (a) Argue that Eleanor's decision problem has a polynomial-time algorithm if and only if Eleanor's optimization problem has a polynomial-time algorithm.

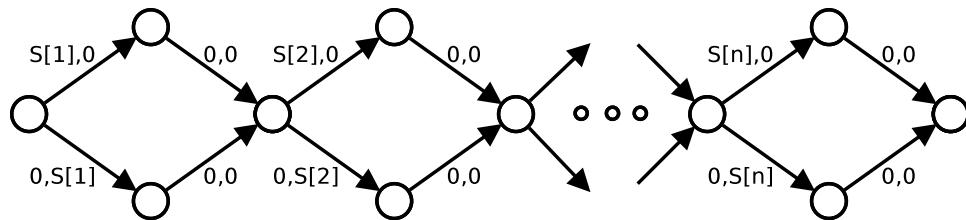
**Solution:** Reduce decision to optimization: solve optimization, output "YES" iff length of found path is less than  $d$ .

Reduce optimization to decision: Compute sum of all edge lengths which gives  $d_{\max}$ . If the decision algo returns "NO" for  $d_{\max}$  then there is no solution. Now binary search for the smallest  $d$  in the range  $[0, d_{\max}]$  on which the decision algo returns "YES". Note  $\log d_{\max}$  is polynomial in input size.

Recall the NP-complete PARTITION problem: Given an array  $S[1 \dots n]$  of natural numbers and a value  $r \in \mathbb{N}$ , determine whether there exists a set  $A \subseteq \{1, 2, \dots, n\}$  of indices such that

$$\max \left\{ \sum_{i \in A} S[i], \sum_{i \notin A} S[i] \right\} \leq r .$$

- (b) Prove that the Eleanor's decision problem is NP-hard. (*Hint:* Consider the graph illustrated below, where the label of each edge  $e$  is “ $c(e), \ell(e)$ ”.)



**Solution:** We reduce PARTITION to Eleanor's decision problem. Label the left-hand vertex in the hint graph as  $s$  and the right-hand vertex  $t$ . Run the algo for Eleanor's decision problem with this graph and  $x = c$  and  $d = c$  to finish the reduction. Any  $s - t$  path in the graph corresponds to a set  $A$  of the indices of those "widgets" where the path follows the upper branch. The cost of this path is  $\sum_{i \in A} S[i]$ , while the length of this path is  $\sum_{i \notin A} S[i]$ . The algo says YES iff there exists a path of both length and cost less than  $c$ , that is, the answer to the PARTITION problem is YES. Construction of the graph takes time linear in  $n$ , so  $\text{PARTITION} \leq_p$  Eleanor's decision problem.

- (c) Argue on the basis of part (b) that Eleanor's decision problem is NP-complete.

**Solution:** The witness is the path which can be checked in time linear in the number of its edges, which is less than  $|V|$ . Since the decision problem belongs to NP and is NP-hard, it is NP-complete.

- (d) The solutions to the take-home quiz showed that there is an efficient  $O(xE + xV \lg V)$  algorithm for Eleanor's optimization problem. Why doesn't this fact contradict the NP-completeness of Eleanor's decision problem?

**Solution:** The value of  $x$  is not polynomial in the input size (e.g., number of bits used to express  $x$ ), so this "efficient" algorithm is no proof that Eleanor's optimization problem is in P.

**Problem 7. Majority Rules.** [30 points] (5 parts)

The  $n$ -input **majority function** takes as input  $n$  binary values  $x_1, x_2, \dots, x_n$  and produces a binary output

$$y = \begin{cases} 1 & \text{if } \sum_{i=1}^n x_i \geq n/2, \\ 0 & \text{otherwise.} \end{cases}$$

We shall be interested in developing combinational (no storage elements) circuits that implement the  $n$ -input majority function and which have small size and depth.

- (a) Explain how a 2-input, 2-output comparator of 1-bit numbers can be implemented using Boolean logic. Then show how the sorting network described in lecture (which uses bitonic sorters) can be employed to compute the  $n$ -input majority function. Analyze the asymptotic depth and size of this sorting-based majority circuit.

**Solution:** The min output of the comparator is the logical AND of the inputs, and the max output is the logical OR. Sort, and the majority is the  $n/2$  output. Depth =  $\Theta(\lg^2 n)$ . Size =  $\Theta(n \lg^2 n)$ .

Another way to compute the majority function is to add up all  $n$  bits of the input and see whether the value is at least  $n/2$ , just as the definition suggests. Assume that  $n$  is an exact power of 2. We can implement such a circuit using a complete binary tree in which each leaf is an input and each internal node is a ripple-carry adder whose number of stages depends on its height in the tree. The output is the sum bit produced by the high-order full adder in the root's ripple-carry adder.

- (b) Consider the ripple-carry adder at the root of a subtree of height  $k$ . How many full adders does this ripple-carry adder require in terms of  $k$ ?

**Solution:**  $k + 1 = \Theta(k)$ . (An answer of  $k$  received full credit.)

- (c) Analyze the asymptotic depth and size of this tree-based majority circuit in terms of  $n$ .  
*(Hint:  $\sum_{k=0}^{\infty} kx^k = x/(1-x)^2$  for  $|x| < 1$ .)*

**Solution:** Size =  $\Theta(\sum_{k=0}^{\lg n} k(n/2^k)) = \Theta(n)$ . Depth =  $O(\lg^2 n)$  received full credit.  
In fact, depth =  $O(\lg n)$ , by observing that each full adder either increases the depth of the circuit by 1 or increases the bit position by 1.

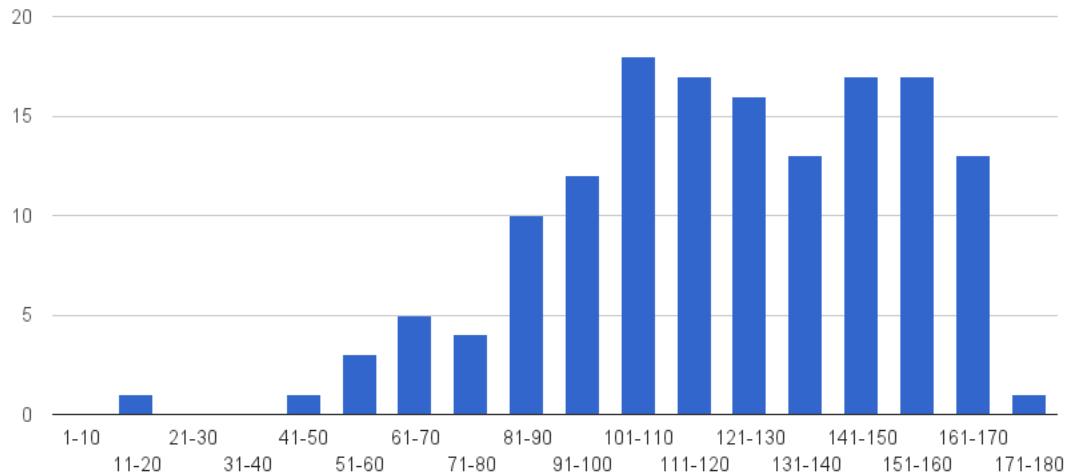
- (d) Suppose that you replace each ripple-carry adder with a carry-lookahead adder. What are the asymptotic depth and size of the resulting circuit?

**Solution:** Depth =  $\Theta(\lg n \lg \lg n)$ . Size =  $\Theta(n)$ .

- (e) Describe an asymptotically efficient majority circuit inspired by Wallace trees. Analyze the asymptotic depth and size of this Wallace-tree-inspired circuit.

**Solution:** Use  $n/3$  carry-save adders, each of which adds 3 inputs and produces 2 outputs. Recurse on the resulting  $\lceil 2n/3 \rceil$  results. At the end, perform a ripple-carry addition of the two final values. Depth =  $\Theta(\lg n)$ . Size =  $\Theta(n)$ . (Math is easier if you use two levels of carry-save to add 4 inputs and produce 2 outputs, because then you have a real tree.)

## Final Exam



Mean: 121.3; Median: 122; Standard deviation: 30.8

**Problem 1. True/False/Justify [40 points] (8 parts)**

Circle **T** or **F** for each of the following statements to indicate whether the statement is true or false, and briefly explain why. Your justification is worth more points than your true-or-false designation.

- (a) **T F** If a problem has an algorithm that is correct for  $2/3$  fraction of the inputs, then it also has an algorithm that is correct for 99.9% of the inputs.

**Solution: False.** If an algorithm is correct for only  $2/3$  of the *inputs*, then it is not necessarily possible to amplify the results to make it correct for those inputs. (Consider, for example, a problem that is easy on  $2/3$  of the inputs but hard/undecidable for the remaining  $1/3$  of the inputs.) An algorithm must be correct for  $2/3$  of the possible sequences of random choices for amplification to apply.

- (b) **T F** If a problem has a randomized algorithm that runs in time  $t$  and returns the correct answer with probability at least  $2/3$ , then the problem also has a deterministic algorithm that runs in time  $t$  and always returns the correct answer.

**Solution: False.** Sublinear time algorithms are counterexamples, as a deterministic algorithm would not be able to read all of the input in the time it would take for the sublinear time algorithm to return an answer.

- (c) **T F** A perfect hash table that is already built requires 2 hash functions, one for the first level hash, and another for the second level hash.

**Solution: False.** In the implementation of perfect hashing which we discussed in recitation, the second level hash table requires a different hash function for every bucket. Note that if we relax the requirement for taking only  $O(n)$  space and allow the hash table to instead take  $O(n^2)$  space, then we can use just one hash function.

- (d) **T F** If  $\Phi$  is a potential function associated with a data structure  $S$ , then  $2\Phi$  is also a potential function that can be associated with  $S$ . Moreover, the amortized running time of each operation with respect to  $2\Phi$  is at most twice the amortized running time of the operation with respect to  $\Phi$ .

**Solution: True.** By definition, the amortized cost  $\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$ . If we use the potential function  $2\Phi$ , then because the actual cost is positive (and the same), the new amortized cost is bounded by 2 times the original.

- (e) **T F** If we use van Emde Boas (vEB) to sort  $n$  elements, it is possible to achieve  $O(n \lg \lg n)$  running time. Thus, whenever we need to use traditional  $O(n \lg n)$  sorting, we can replace it with vEB sort and achieve a better asymptotic running time (ignore setup time).

**Solution: False.** In order to use van Emde Boas, we must have a restriction on the input (it is within a universe of integers size 1 through  $n$ ).

- (f) **T F** The “union-by-rank” and “path-compression” heuristics do not improve the running time of MAKE-SET in the union-find data structure.

**Solution: True.** The running time of MAKE-SET is always  $O(1)$ .

- (g) **T F** There is an NP-hard problem with a known polynomial time randomized algorithm that returns the correct answer with probability  $2/3$  on all inputs.

**Solution: False.** We currently don't know whether there are NP-hard problems that can be solved by polynomial time randomized algorithms, and conjecture that these do not exist.

- (h) **T F** Every special case of an NP-hard problem is also NP-hard.

**Solution: False.** Consider the MST problem, which is a special case of the Steiner Tree problem.

**Problem 2.** Short Answer [20 points] (4 parts)

- (a) Recall the forest-of-trees solution to the disjoint-set problem. Suppose that the only heuristic we use is a variant of the union-by-rank heuristic: when merging two roots  $u$  and  $v$ , we compare the number of descendants (rather than the rank), and make  $u$  a child of  $v$  if and only if  $u$  has fewer descendants. Is this asymptotically worse than the original union-by-rank heuristic? Explain why or why not.

**Solution:** No, it is not asymptotically worse. The way that heuristics affect the runtime is by affecting the height of the trees in the data structure. Consider some item  $x$  in the data structure. When will the path from  $x$  to the root increase in length? Well, it can only increase in length when the root of  $x$  changes. Let  $u$  be the old root of  $x$ , and let  $v$  be the new root of  $x$ . Let  $d_{old}(\cdot)$  be the number of descendants before the merge, and let  $d_{new}(\cdot)$  be the number of descendants after the merge.

Because of the union-by-descendants heuristic, the root could only change from  $u$  to  $v$  if  $d_{old}(u) \leq d_{old}(v)$ . This means that  $d_{new}(v) = d_{old}(v) + d_{old}(u) \geq 2d_{old}(u)$ . So every time the path from  $x$  to the root increases by 1, the number of descendants of the root at least doubles. This means that the depth of  $x$  is at most  $O(\lg n)$ , just as with the union-by-rank heuristic.

- (b) Suppose we apply Huffman coding to an alphabet of size 4, and the resulting tree is a perfectly balanced binary tree (one root with two children, each of which has two children of its own). Find the maximum frequency of any letter.

**Solution:** The maximum frequency of any letter is  $2/5$ . To see why, we must prove that this is both feasible, and as large as possible.

- Suppose that we have four characters  $a, b, c, d$  with frequencies  $f_a = 2/5$  and  $f_b = f_c = f_d = 1/5$ . Without loss of generality, suppose that Huffman's algorithm merges  $c$  and  $d$ . Then  $f_{cd} = f_c + f_d = 2/5$ . This introduces a tie<sup>1</sup>, which we may break by assuming that Huffman's algorithm chooses to merge  $b$  with  $a$ . This yields a perfectly balanced tree.
- To see that any frequency greater than  $2/5$  is impossible, let  $f_a \geq f_b \geq f_c \geq f_d$  be the frequencies in decreasing order. We define these frequencies so that  $f_a + f_b + f_c + f_d = 1$ . For the sake of contradiction, suppose that  $f_a > 2/5$ . This means that  $f_b + f_c + f_d < 3/5$ . Because  $f_b > f_c, f_d$ , we can conclude that  $f_c + f_d < 2/5$ . But this means that when  $c$  and  $d$  are merged, their combined frequency will be strictly less than  $f_a$ , so we will not get a perfectly balanced tree.

---

<sup>1</sup>To avoid a tie, we must use  $2/5 - \epsilon$  as the maximum frequency instead.

- (c) In lecture, we saw min-radius clustering, in which the goal was to pick a subset of  $k$  points such that each point formed the center of a cluster of radius  $r$ . Suppose instead that the center of the cluster can be a point not in the original set.

Give an example set of points where it is possible to find  $k$  clusters of radius  $r$  centered around arbitrary points, but impossible to find  $k$  clusters of radius  $r$  centered around points in the set.

**Solution:** Given arbitrary values of  $k$  and  $r$ , we can set up a counterexample in  $\mathbb{R}^2$  as follows. For each  $i \in \{1, \dots, k\}$ , create two points:  $p_{i,1} = (4ri, 0.9r)$  and  $p_{i,2} = (4ri, -0.9r)$ . For any pair  $p_{i,1}$  and  $p_{i,2}$  the distance is equal to  $1.8r$ , so any cluster centered on one of the points cannot contain the other. For all other pairs of points, their  $x$ -coordinates must differ by at least  $4r$ , and so they cannot belong to the same cluster. So there's no way to create  $k$  clusters centered on  $k$  points in the set. However, there is a way to create  $k$  clusters centered on  $k$  arbitrary points: for each  $i \in \{1, \dots, k\}$ , the point  $(4ri, 0)$  has distance  $\leq r$  to both  $p_{i,1}$  and  $p_{i,2}$ .

- (d) Consider the following algorithm, which is intended to compute the shortest distance among a collection of points in the plane:

- 1 Sort all points by their  $y$ -coordinate.
- 2 **for** each point in the sorted list:
- 3     Compute the distance to the next 7 points in the list.
- 4 **return** the smallest distance found.

Give an example where this algorithm will return a distance that is not in fact the overall shortest distance.

**Solution:** To construct a counterexample, it is sufficient to construct an example where the shortest distance is between two points that have at least 7 points between them in the ordering of  $y$ -coordinates. The example we use is as follows:

$$(0, -1) \quad (0, 5) \quad (0, 10) \quad (0, 15) \quad (0, 20) \quad (0, 25) \quad (0, 30) \quad (0, 35) \quad (0, 1)$$

In any ordering of these points by  $y$ -coordinate, the seven points with  $y$ -coordinate 0 will lie between  $(0, -1)$  and  $(0, 1)$ . Hence, the algorithm will never compute the Euclidean distance between  $(0, -1)$  and  $(0, 1)$ , and so it cannot find the true shortest distance.

**Problem 3. Estate Showing.** [30 points] (3 parts)

Trip Trillionaire is planning to give potential buyers private showings of his estate, which can be modeled as a weighted, directed graph  $G$  containing locations  $V$  connected by one-way roads  $E$ . To save time, he decides to do  $k$  of these showings at the same time, but because they were supposed to be private, he doesn't want any of his clients to see each other as they are being driven through the estate.

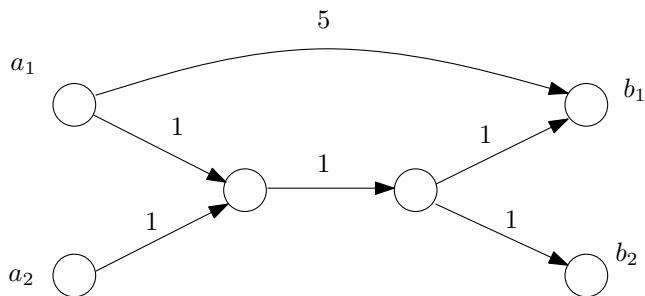
Trip has  $k$  grand entrances to his estate,  $A = \{a_1, a_2, \dots, a_k\} \subset V$ . He wants to take each of his buyers on a path through  $G$  from their starting location  $a_i$  to some ending location in  $B = \{b_1, b_2, \dots, b_k\} \subset V$ , where there are spectacular views of his private mountain range.

Because of your prowess with algorithms, he hires you to help him plan his buyers' visits. His goal is to find a path for each buyer  $i$  from the entrance they take,  $a_i$ , to any ending location  $b_j$  such that no two paths share any edges, and no two buyers end in the same location  $b_j$ .

- (a) Trip tells you his idea: find all-pairs shortest paths on  $G$ , and then try to select  $k$  of those shortest paths  $a_i \rightsquigarrow b_j$  such that all  $k$  paths start and end at different vertices and no two paths share any edges.

Give a graph where there exists a set of paths satisfying the requirements, but where Trip's strategy won't find it.

**Solution:** Consider this graph:



The all-pairs shortest paths algorithm would find all shortest paths from  $a_i$  to  $b_j$  ( $a_1 \rightsquigarrow b_1$ ,  $a_1 \rightsquigarrow b_2$ ,  $a_2 \rightsquigarrow b_1$ , and  $a_2 \rightsquigarrow b_2$ ), which all go through the same edge. Trip's algorithm, which considers only those paths, would find no solution. There are two completely disjoint paths using the shortest path  $a_2 \rightsquigarrow b_2$  and the direct edge  $(a_1, b_1)$ , but Trip's algorithm would not find this combination because one of them is not a shortest path.

- (b)** Rather than using shortest paths, you think that perhaps you can formulate this as a flow problem. Find an algorithm to find  $k$  paths  $a_i \rightsquigarrow b_j$  that start and end at different vertices and that share no edges, and briefly justify the correctness and running time of your algorithm.

**Solution:** The algorithm is as follows:

1. Create a flow network  $G'$  containing all vertices in  $V$ , all directed edges in  $E$  with capacity 1, and additionally a source vertex  $s$  and a sink vertex  $t$ . Connect the source to each starting location with a directed edge  $(s, a_i)$  and each ending location to the sink with a directed edge  $(b_j, t)$ , all with capacity 1.
2. Run Ford-Fulkerson on this network to get a maximum flow  $f$  on this network. If  $|f| = k$ , then there is a solution; if  $|f| < k$ , then there is no solution, so we return FALSE. We will later show that it is always the case that  $|f| \leq k$ .
3. To extract the paths from  $a_i$  to  $b_j$  (as well as which starting location ultimately connects to which ending location), run a depth-first search on the returned max flow  $f$  starting from  $s$ , tracing a path to  $t$ . Remove these edges and repeat  $k$  times until we have the  $k$  disjoint paths.

The running time for this algorithm is  $O(k|E|)$ : in the modified graph  $G'$ , we have  $|E| + 2k = O(|E|)$  edges, and the maximum possible flow value is  $k$ ; therefore, running Ford-Fulkerson takes  $O(k|E|)$  time. Extracting paths takes at most  $O(|E|)$  time, as we traverse each edge at most once. In total, then, the algorithm takes  $O(k|E|)$  time.

Some students used Edmonds-Karp, which runs in  $O(VE^2)$  time. In this case, because  $k = O(|V|)$ , Ford-Fulkerson provides a better bound.

To show correctness, we must show that there is a flow of size  $k$  if and only if there are  $k$  disjoint paths.

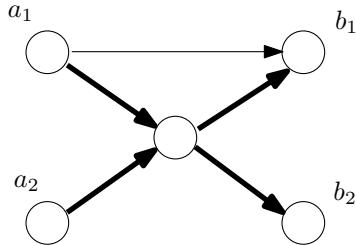
( $\rightarrow$ ) **Claim.** An integral flow of size  $f$  in a network with unit capacities can be decomposed into a unit flow over a simple path  $p : s \rightsquigarrow t$  and a flow of size  $f - 1$  which does not use  $p$ .

**Proof.** Pick any simple path  $p : s \rightsquigarrow t$  over edges with non-zero flow. One must exist (otherwise, we would have a cut with no flow). In the residual graph, that path has capacity 1; send an augmenting flow of -1 over that path. We then have a resulting flow of size  $f - 1$ , and the flow on each of the edges of  $p$  is 0.

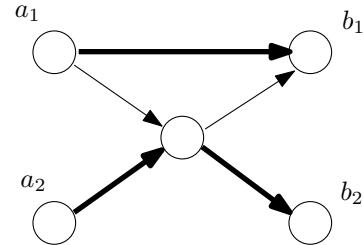
( $\leftarrow$ ) If there is a set of  $k$  disjoint paths from  $a_i$  to  $b_j$ , then the flow with one unit of flow on each of the edges contained in those paths defines a flow of size  $k$ . Because the cut  $(s, V' - s)$  has capacity  $k$ , this is a maximum flow on the network.

A common incorrect solution involved using the augmenting paths found by Ford-Fulkerson as the  $k$  paths for the buyers to use. This approach doesn't work because the augmenting paths are not guaranteed to be the final paths along which there are units of flow.

- (c) Trip, after trying out the paths found by your algorithm, realizes that making sure the  $k$  paths don't share edges isn't enough: it's possible that some paths will share vertices, and his buyers might run into each other where their paths intersect.



$a_1 \rightsquigarrow b_1$  and  $a_2 \rightsquigarrow b_2$  share a vertex



$a_1 \rightsquigarrow b_1$  and  $a_2 \rightsquigarrow b_2$  share neither vertices nor edges

Modify your algorithm to find  $k$  paths  $a_i \rightsquigarrow b_j$  that start and end in different locations, and that share neither vertices nor edges.

**Solution:** Duplicate each vertex  $v$  into two vertices  $v_{in}$  and  $v_{out}$ , with a directed edge between them. All edges  $(u, v)$  now become  $(u, v_{in})$ ; all edges  $(v, w)$  now become  $(v_{out}, w)$ . Assign the edge  $(v_{in}, v_{out})$  capacity 1.

With this transformation, we now have a graph in which there is a single edge corresponding to each vertex, and thus any paths that formerly shared vertices would be forced to share this edge.

Now, we can use the same algorithm as in part (b) on the modified graph to find  $k$  disjoint paths sharing neither edges nor vertices, if they exist.

The transformation of the graph takes  $O(|E|)$  time, as we are adding that many new vertices and edges to the graph. The new graph has  $O(|E| + |V| + 2k) = O(|E|)$  edges, and therefore the running time is unchanged.

Many students excluded the starting and ending locations  $a_i$  and  $b_i$  from the vertices duplicated. This approach, however, would overlook any paths that might have gone through some  $a_i$  or  $b_i$  before reaching their final destinations. (There was nothing in the previous problem statement preventing this from occurring.)

As in part (b), some students also tried to modify the operation of Ford-Fulkerson by removing all edges incident to vertices in the augmenting paths found, thus not allowing any other augmenting paths to go through those vertices. Again, this approach doesn't work because it is too greedy: the augmenting paths found by Ford-Fulkerson are not necessarily the final paths along which there are units of flow, and therefore this approach may miss valid solutions.

**Problem 4. Credit Card Optimization** [30 points] (3 parts)

Zelda Zillionaire is planning on making a sequence of purchases with costs  $x_1, \dots, x_n$ . Ideally, she would like to make all of these purchases on one of her many credit cards. Each credit card has credit limit  $\ell$ . Zelda wants to minimize the number of credit cards that she needs to use to make these purchases, without exceeding the credit limit on any card. More formally, she wants to know the smallest  $k$  such that there exists  $\pi : \{1, \dots, n\} \rightarrow \{1, \dots, k\}$  assigning each purchase  $i$  to a credit card  $j$ , where  $\pi$  must satisfy the following constraint:

$$\forall j \in \{1, \dots, k\} : \sum_{\substack{i \in \{1, \dots, n\} \\ \text{s.t. } \pi(i)=j}} x_i \leq \ell.$$

Zelda is thinking of using the following algorithm to distribute her purchases:

```

1 Create an empty list of credit cards  $L$ .
2 for each purchase  $x_i$ :
3   for each credit card  $j \leq |L|$ :
4     if  $L[j] + x_i \leq \ell$ :
5       Purchase  $x_i$  with credit card  $j$ .
6       Set the balance for card  $j$  to  $L[j] = L[j] + x_i$ .
7       Skip to the next purchase.
8     if no existing credit card has enough credit left:
9       Purchase  $x_i$  with a new credit card.
10      Append a new credit card to  $L$ .
11      Set the balance of the new credit card to  $x_i$ .
12 return  $k = |L|$ 
```

- (a) Give an example where Zelda's algorithm will not use the optimal number of credit cards.

**Solution:**

Let the credit limit  $\ell$  be 10. Consider the following sequence of purchases as input to Zelda's algorithm: 2, 1, 9, 8. The optimal solution would only use 2 credit cards by grouping purchases {2, 8} and {9, 1}. Zelda's algorithm would split the charges into 3 cards: {2, 1}, {9}, {8}, which is suboptimal.

This problem part was worth 5 points. Almost everyone got this problem right. Some students didn't read carefully the pseudocode and inverted the order of the two loops.

- (b) Show that Zelda's algorithm gives a 2-approximation for the number of credit cards.

**HINT:** Let  $OPT$  be the optimal number of credit cards. In order for the algorithm to add a new credit card  $j > OPT$ , it must reach  $x_i$  such that  $x_i + \min_{j' \in \{1, \dots, OPT\}} L[j'] > \ell$ . It will then set  $L[j] = x_i$ .

**Solution:**

First, we show that the following loop invariant holds. At no point are there two distinct cards  $i$  and  $j$ , where  $i < j$  such that the balance on each card  $L[j] \leq \frac{\ell}{2}$  and  $L[i] \leq \frac{\ell}{2}$ . Suppose to the contrary that there are two such cards. Then the remaining credit on card  $i$  was sufficient to transfer all of the purchases  $x_i$  from the new credit card to the old credit card  $j$ , instead of opening up a new credit card.

We prove that Zelda's algorithm gives a 2-approximation by contradiction. Namely, we assume that there is a case where at least  $2OPT + 1$  cards are needed. By the loop invariant we know that no 2 credit cards have less than  $\frac{\ell}{2}$  as a balance. So, assuming that there are no zero purchase charges, the total amount of purchases must be more than  $2OPT \cdot \frac{\ell}{2}$ , which is the maximum amount that the  $OPT$  cards in the optimal solution can hold. This is a contradiction. So, we can always find at most  $2OPT$  cards to pay for all of the purchases.

This problem part was worth 10 points.

- (c) Show that minimizing the number of credit cards used is NP-hard.

**Solution:**

We prove that minimizing the number of credit cards is NP-hard by reducing SET-PARTITION to the Zelda's problem in polynomial time.

First, we recast Zelda's optimization problem as a decision problem. Let the decision problem be formulated as follows: For a given target number of cards  $k$ , we are able to come up with an assignment of purchases  $x_1 \dots x_n$  to  $k$  credit cards, where the cumulative balance on each credit card can not exceed the limit  $l$ .

Recall that we defined SET-PARTITION as follows:

Given set  $S = \{s_1, \dots, s_n\}$  we can partition them into 2 disjoint subsets  $A$  and  $B$  such that  $\max\{\sum_{i \in A} s_i, \sum_{j \in B} s_j\}$  is minimized.

The decision version of the SET-PARTITION problem can be reformulated as the question whether the numbers can be partitioned into two sets  $A$  and  $B = S - A$  such that  $\sum_{i \in A} s_i = \sum_{j \in B} s_j$ .

Given an input set  $S = \{s_1, \dots, s_n\}$  to the SET-PARTITION problem, we can use the elements in the set as the purchases in Zelda's problem, set the number of cards to be 2 and the credit card limit  $l = \frac{1}{2} \sum_{i \in S} s_i$ . Clearly, if the answer to the Zelda's decision problem is in the affirmative, then each credit card will have balance of exactly  $\frac{l}{2}$ , and we can take the disjoint subsets of purchases on each card to be the two sets  $A$  and  $B$  that would satisfy SET-PARTITION. The reduction can be executed in linear time.

We also accepted the reduction from SUBSET-SUM to Zelda.

We gave 5 points for correctly identifying an NP-hard problem from which one could reduce to Zelda's problem to prove that the problem is NP-hard (the direction of the reduction needed to be correct to receive all the points). 10 points was given for a correct reduction and analysis.

**Problem 5.** Well-Separated Clusters [30 points] (3 parts)

Let  $S = \{(x_1, y_1), \dots, (x_n, y_n)\}$  be a set of  $n$  points in the plane. You have been told that there is some  $k$  such that the points can be divided into  $k$  clusters with the following properties:

1. Each cluster has radius  $r$ .
2. The center of each cluster has distance at least  $5r$  from the center of any other cluster.
3. Each cluster contains at least  $\epsilon n$  elements.

Your goal is to use this information to construct an algorithm for computing the number of clusters.

- (a) Give an algorithm that takes as input  $r$  and  $\epsilon$ , and outputs the exact value of  $k$ .

**Solution:**

**Executive Summary.** We find all points sufficiently close to a given point, remove all such points and increment the number of clusters, and repeat until we run out of points. This procedure is exact and takes  $O(nk)$  time.

**Algorithm** Initialize the number of clusters  $k \equiv 0$ , and the set  $S' \equiv S$ . Choose a point  $p \in S'$ , and compute the distance from  $p$  to every other point in  $S'$ . Remove  $p$  and all points within distance  $2r$  from  $S'$ , and increment  $k$ . Repeat this procedure until  $S'$  is empty; at that point, return  $k$ .

**Correctness** We first note that if the point  $p$  chosen in an iteration lies in a cluster with center  $c$ , then all points within this cluster will be removed in the same iteration. Indeed, if  $q$  is any other point in this cluster, by the triangle inequality,

$$d(p, q) \leq d(p, c) + d(c, q) \leq r + r = 2r.$$

Furthermore, no points from other clusters will be removed in the same iteration. Indeed, if  $q'$  is in another cluster with center  $c'$ , then by the triangle inequality

$$\begin{aligned} d(p, q') &= (d(c, p) + d(p, q') + d(q', c')) - d(c, p) - d(q', c') \\ &\geq d(c, c') - d(c, p) - d(q', c') \\ &\geq 5r - r - r = 3r. \end{aligned}$$

**Runtime Analysis** Each iteration computes at most  $n$  distances, and removes exactly 1 cluster. Hence there are  $k$  iterations total, so the overall runtime is  $O(nk)$ . Since each cluster contains at least  $\epsilon n$  points, there are at most  $1/\epsilon$  clusters, so the runtime can also be phrased as  $O(n/\epsilon)$ .

**Grading comments** This part was worth 12 marks total, with 6 marks allotted for a correct and efficient algorithm, 4 for justification of correctness, and 2 for runtime analysis. Some students omitted the latter two parts and were duly penalized, since both are expected in this course.

Correct algorithms which were slightly inefficient (e.g  $O(n^2)$  or similar) were deducted one mark independent of analysis; this often occurred if all pairs of distances were computed, or a disjoint-set structure used to find the actual clusters (both were unnecessary). More inefficient or incorrect algorithms were given at most 3 marks.

Several students noted that this was a variant of Hierarchical Agglomerative Clustering as discussed in class, which is true; however, justification of correctness was still necessary since the latter is an approximation algorithm. This observation was not required for full credit.

- (b) Given a particular cluster  $C_i$ , if you sample  $t$  points uniformly at random (with replacement), give an upper bound in terms of  $\epsilon$  for the probability that none of these points are in  $C_i$ .

**Solution:**

**Solution** The number of points in  $C_i$  is at least  $\epsilon n$ , so the probability that a point sampled u.a.r. lies within  $C_i$  is at least  $\epsilon n/n = \epsilon$ . Therefore the probability that a point sampled u.a.r. *doesn't* lie within  $C_i$  is at most  $1 - \epsilon$ . Since the points are chosen independently, the probability that  $t$  points sampled u.a.r. don't lie within  $C_i$  is at most  $(1 - \epsilon)^t$ .

**Grading comments** This problem was relatively straightforward; the full 8 marks were given for the correct answer. An alternative (slightly weaker) bound using Chernoff was also given full credit as long as it was computed correctly. When the answer was incorrect, part marks were allotted for similarity to the solution and evidence of work.

- (c) Give a sublinear-time algorithm that takes as input  $r$  and  $\epsilon$ , and outputs a number of clusters  $k$  that is correct with probability at least  $2/3$ .

**Solution:**

**Executive Summary** Use the algorithm from part (a) on a subset of the points chosen u.a.r. The size of the subset is chosen so that the algorithm is correct with the desired probability.

**Algorithm** Initialize  $S'$  to be a subset of  $S$  of size  $t$  (to be determined below), where the points are chosen uniformly at random. Run the same algorithm as in part (a) on this subset.

**Correctness** From part (b), the probability that  $t$  points sampled u.a.r. don't lie in a given cluster is  $(1 - \epsilon)^t$ . Hence, by the union bound, the probability that  $t$  points sampled u.a.r. don't lie in *any* cluster is at most  $k(1 - \epsilon)^t$ ; since  $k \leq 1/\epsilon$ , this can be upper bounded by  $\frac{1}{\epsilon}(1 - \epsilon)^t$ . The algorithm is incorrect iff the  $t$  sampled points miss any cluster, so we want this probability to be at most one third:

$$\begin{aligned} \frac{1}{\epsilon}(1 - \epsilon)^t &\leq \frac{1}{3} \\ (1 - \epsilon)^t &\leq \frac{\epsilon}{3} \\ t \log(1 - \epsilon) &\leq \log \frac{\epsilon}{3} \\ t &\geq \frac{\log \frac{\epsilon}{3}}{\log(1 - \epsilon)} \end{aligned}$$

(the sign is reversed in the last inequality since  $\log(1 - \epsilon)$  is negative). The rest of justification of correctness follows from part (a).

**Runtime analysis** The runtime of this algorithm is  $O(tk)$  or  $O(t/\epsilon)$ ; the analysis is similar to that for part (a). Since neither  $t$  nor  $\epsilon$  depend on  $n$ , this algorithm is sublinear.

**Grading comments** This part was worth 10 marks total, with 6 marks for achieving the correct approach to the sublinear algorithm and 4 marks for computing the size of the subset correctly.

Some students failed to realize that part (b) gives the probability of missing a *fixed* cluster, whereas this part requires the probability of missing *any* cluster; this resulted in a deduction of 2 marks. Others erroneously assumed that the events of missing each cluster were independent; this resulted in a deduction of 1 mark. Incorrect algorithms were given partial credit depending on their similarity with the correct solution.

Several students noted that this was a variant of the sublinear Connected Components algorithm as discussed in class; again, while true, this was not necessary and a correct calculation of the subset size was still required.

**Problem 6. Looking for a Bacterium [30 points] (2 parts)**

Imagine you want to find a bacterium in a one dimensional region divided into  $n$   $1\text{-}\mu\text{m}$  regions. We want to find in which  $1\text{-}\mu\text{m}$  region the bacterium lives. A microscope of resolution  $r$  lets you query regions 1 to  $r$ ,  $r + 1$  to  $2r$ , etc. and tells you whether the bacteria is inside that region. Each time we operate a microscope to query one region, we need to pay  $(n/r)$  dollars for electricity (microscopes with more precise resolution take more electricity). We also need to pay an  $n$ -dollar fixed cost for each type of microscope we decide to use (independent of  $r$ ).

- (a) Suppose that you decide to purchase  $\ell = \lg n$  microscopes, with resolutions  $r_1, \dots, r_\ell$  such that  $r_i = 2^{i-1}$ . Give an algorithm for using these microscopes to find the bacterium, and analyze the combined cost of the queries and microscopes. **Solution:**

**Executive Summary** We will use all the microscope to perform a binary search. Starting with the lowest resolution microscope, we use successively more powerful microscopes and eventually using the microscope of resolution 1 to locate exactly where the bacterium is.

**Algorithm** We will assume  $n$  is a power of 2, if not, we can imagine expanding to a larger region that is a power of 2. This will not affect the asymptotic cost. We first use a microscope of resolution  $n/2$  to query the region  $(1, n/2)$ . Regardless of whether the answer is yes or no, we can eliminate a region of size  $n/2$ . We then do the same with a microscope of resolution  $n/4$ , and so on.

**Correctness** This is essentially the same as a binary search, we use each type of microscope exactly once, and eventually we will pin point the location of the bacteria to a region of size 1.

**Cost analysis** We are only interested in the total electricity cost for this problem. Some students provided runtime analysis, but their correctness were not graded. We use each microscope exactly once, starting from the the one with the lowest resolution, it costs us  $2, 4, 8, \dots, n$  to use each microscope. The last cost of  $n$  comes from using the microscope of resolution 1. This geometric series sums to a total of  $2n - 2$ , thus the electricity cost is  $O(n)$ . The total cost of purchasing the microscope is  $n \lg n$ , for a combined cost of  $O(n \lg n)$ .

**Grading comments** The maximum points for this part is 10. 3 points are awarded for people who can roughly describe the algorithm. 5 to 7 points are awarded for people who fail to correctly analyze the cost of electricity to various degrees. Points are also deducted if you simply assert the total cost of electricity is  $O(n)$ , we are looking for at least a mentioning of things such as this geometric sum converges.

Most students did pretty well on this part, some students attempted to use recurrence to solve for the cost of electricity, but most of them made mistakes.

- (b) Give a set of microscopes and an algorithm to find the bacterium with total cost  $O(n \lg \lg n)$ .

**Solution:**

**Executive Summary** We borrow our idea from the vEB data structure. We will purchase microscopes of resolutions  $n^{1/2}, n^{1/4}, n^{1/8}, \dots, n^{1/2^i} = 2$ . Clearly,  $i = \lg \lg n$ . So we spend  $n \lg \lg n$  for buying microscopes. We do a linear search at each level, and we will show that the total cost of electricity is also  $n \lg \lg n$ .

**Algorithm** We perform a search for the bacterium as follows. First we use the microscope with resolution  $\sqrt{n}$  to locate which region of size  $\sqrt{n}$  the bacterium is in. This will take at most  $\sqrt{n}$  applications of this microscope. We can then narrow our search in the region of size  $\sqrt{n}$  and use microscopes of resolution  $n^{1/4}$  to continue this process, eventually we can narrow down the location of the bacterium to a region of size 2. From there, it takes one simple application of a microscope of resolution 1 to find the bacterium.

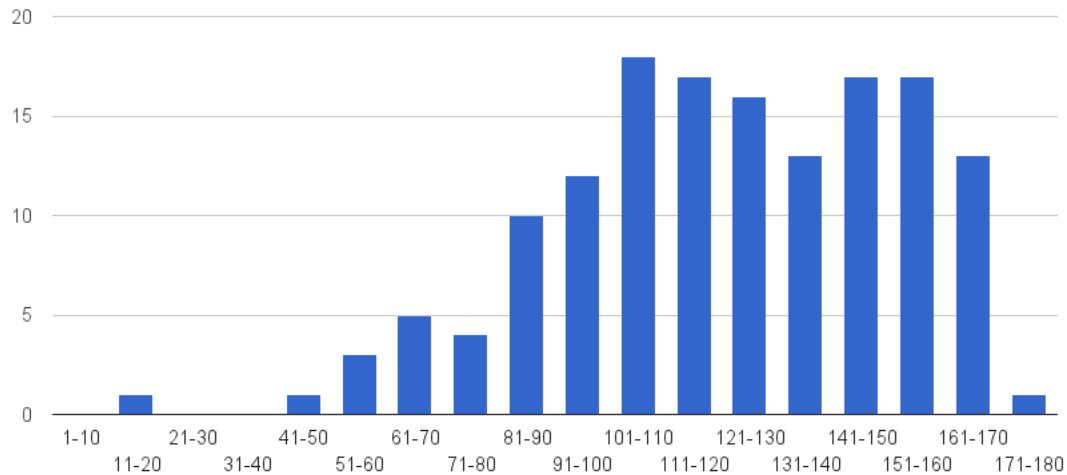
**Correctness** This collection of microscopes allows us to search every region exhaustively, thus we can always find the bacterium.

**Cost analysis** The total purchase of  $\lg \lg n$  microscopes will cost us  $n \lg \lg n$  dollars. At stage 1, we use the microscope of resolution  $\sqrt{n}$  a total of  $\sqrt{n}$  times. Each use of this microscope costs  $\frac{n}{\sqrt{n}}$  dollars, for a total of  $n$  dollars. In fact, at every subsequent step, we always apply the microscope of resolution  $\sqrt{r}$  a total of  $\sqrt{r}$  times to completely scan a region of size  $r$ . The total cost of this is always  $n$ . Thus, the total electricity cost associated with using each type of microscope is at most  $n$ . Therefore, the total electricity cost is also  $n \lg \lg n$ . Our combined total cost is thus  $O(n \lg \lg n)$ .

**Grading comments** The maximum points for this part is 20. 2 points are given for some very vague mentioning of vEB, demonstrating some level of understanding. 5 points are given for the correct choice of microscope resolutions (but not how to use them to search for the bacterium). 10 points are given if the algorithm is correct (correct choice of resolutions as well as a description of how to use them to search for the bacterium). 13 points are awarded if the electricity cost is analyzed incorrectly, but has some correct ideas. 16 to 18 points are awarded if there are some minor mistakes in the electricity cost analysis. Many students attempted to analyze the electricity cost using recurrence, arriving at  $C(n) = C(\sqrt{n}) + O(n)$ . The idea is that the cost to search a region of size  $n$  is equal to spending  $O(n)$  cost to apply the microscope of resolution  $\sqrt{n}$  a total of  $\sqrt{n}$  times, and then recurse on a region of size  $\sqrt{n}$ . This

problem with this approach is that the  $O(n)$  term is a constant. Every level costs at most  $n$  dollars, but this cost does not scale down with region size. Therefore, the correct way to setup the recurrence is to distinguish the variables from the constants, as  $C(k) = C(\sqrt{k}) + n$ . this indeed gives us an answer of  $C(k) = n \lg \lg k$ .

## Final Exam



Mean: 121.3; Median: 122; Standard deviation: 30.8

**Problem 1. True/False/Justify [40 points] (8 parts)**

Circle **T** or **F** for each of the following statements to indicate whether the statement is true or false, and briefly explain why. Your justification is worth more points than your true-or-false designation.

- (a) **T F** If a problem has an algorithm that is correct for  $2/3$  fraction of the inputs, then it also has an algorithm that is correct for 99.9% of the inputs.

**Solution: False.** If an algorithm is correct for only  $2/3$  of the *inputs*, then it is not necessarily possible to amplify the results to make it correct for those inputs. (Consider, for example, a problem that is easy on  $2/3$  of the inputs but hard/undecidable for the remaining  $1/3$  of the inputs.) An algorithm must be correct for  $2/3$  of the possible sequences of random choices for amplification to apply.

- (b) **T F** If a problem has a randomized algorithm that runs in time  $t$  and returns the correct answer with probability at least  $2/3$ , then the problem also has a deterministic algorithm that runs in time  $t$  and always returns the correct answer.

**Solution: False.** Sublinear time algorithms are counterexamples, as a deterministic algorithm would not be able to read all of the input in the time it would take for the sublinear time algorithm to return an answer.

- (c) **T F** A perfect hash table that is already built requires 2 hash functions, one for the first level hash, and another for the second level hash.

**Solution: False.** In the implementation of perfect hashing which we discussed in recitation, the second level hash table requires a different hash function for every bucket. Note that if we relax the requirement for taking only  $O(n)$  space and allow the hash table to instead take  $O(n^2)$  space, then we can use just one hash function.

- (d) **T F** If  $\Phi$  is a potential function associated with a data structure  $S$ , then  $2\Phi$  is also a potential function that can be associated with  $S$ . Moreover, the amortized running time of each operation with respect to  $2\Phi$  is at most twice the amortized running time of the operation with respect to  $\Phi$ .

**Solution: True.** By definition, the amortized cost  $\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$ . If we use the potential function  $2\Phi$ , then because the actual cost is positive (and the same), the new amortized cost is bounded by 2 times the original.

- (e) **T F** If we use van Emde Boas (vEB) to sort  $n$  elements, it is possible to achieve  $O(n \lg \lg n)$  running time. Thus, whenever we need to use traditional  $O(n \lg n)$  sorting, we can replace it with vEB sort and achieve a better asymptotic running time (ignore setup time).

**Solution: False.** In order to use van Emde Boas, we must have a restriction on the input (it is within a universe of integers size 1 through  $n$ ).

- (f) **T F** The “union-by-rank” and “path-compression” heuristics do not improve the running time of MAKE-SET in the union-find data structure.

**Solution: True.** The running time of MAKE-SET is always  $O(1)$ .

- (g) **T F** There is an NP-hard problem with a known polynomial time randomized algorithm that returns the correct answer with probability  $2/3$  on all inputs.

**Solution: False.** We currently don't know whether there are NP-hard problems that can be solved by polynomial time randomized algorithms, and conjecture that these do not exist.

- (h) **T F** Every special case of an NP-hard problem is also NP-hard.

**Solution: False.** Consider the MST problem, which is a special case of the Steiner Tree problem.

**Problem 2.** Short Answer [20 points] (4 parts)

- (a) Recall the forest-of-trees solution to the disjoint-set problem. Suppose that the only heuristic we use is a variant of the union-by-rank heuristic: when merging two roots  $u$  and  $v$ , we compare the number of descendants (rather than the rank), and make  $u$  a child of  $v$  if and only if  $u$  has fewer descendants. Is this asymptotically worse than the original union-by-rank heuristic? Explain why or why not.

**Solution:** No, it is not asymptotically worse. The way that heuristics affect the runtime is by affecting the height of the trees in the data structure. Consider some item  $x$  in the data structure. When will the path from  $x$  to the root increase in length? Well, it can only increase in length when the root of  $x$  changes. Let  $u$  be the old root of  $x$ , and let  $v$  be the new root of  $x$ . Let  $d_{old}(\cdot)$  be the number of descendants before the merge, and let  $d_{new}(\cdot)$  be the number of descendants after the merge.

Because of the union-by-descendants heuristic, the root could only change from  $u$  to  $v$  if  $d_{old}(u) \leq d_{old}(v)$ . This means that  $d_{new}(v) = d_{old}(v) + d_{old}(u) \geq 2d_{old}(u)$ . So every time the path from  $x$  to the root increases by 1, the number of descendants of the root at least doubles. This means that the depth of  $x$  is at most  $O(\lg n)$ , just as with the union-by-rank heuristic.

- (b) Suppose we apply Huffman coding to an alphabet of size 4, and the resulting tree is a perfectly balanced binary tree (one root with two children, each of which has two children of its own). Find the maximum frequency of any letter.

**Solution:** The maximum frequency of any letter is  $2/5$ . To see why, we must prove that this is both feasible, and as large as possible.

- Suppose that we have four characters  $a, b, c, d$  with frequencies  $f_a = 2/5$  and  $f_b = f_c = f_d = 1/5$ . Without loss of generality, suppose that Huffman's algorithm merges  $c$  and  $d$ . Then  $f_{cd} = f_c + f_d = 2/5$ . This introduces a tie<sup>1</sup>, which we may break by assuming that Huffman's algorithm chooses to merge  $b$  with  $a$ . This yields a perfectly balanced tree.
- To see that any frequency greater than  $2/5$  is impossible, let  $f_a \geq f_b \geq f_c \geq f_d$  be the frequencies in decreasing order. We define these frequencies so that  $f_a + f_b + f_c + f_d = 1$ . For the sake of contradiction, suppose that  $f_a > 2/5$ . This means that  $f_b + f_c + f_d < 3/5$ . Because  $f_b > f_c, f_d$ , we can conclude that  $f_c + f_d < 2/5$ . But this means that when  $c$  and  $d$  are merged, their combined frequency will be strictly less than  $f_a$ , so we will not get a perfectly balanced tree.

---

<sup>1</sup>To avoid a tie, we must use  $2/5 - \epsilon$  as the maximum frequency instead.

- (c) In lecture, we saw min-radius clustering, in which the goal was to pick a subset of  $k$  points such that each point formed the center of a cluster of radius  $r$ . Suppose instead that the center of the cluster can be a point not in the original set.

Give an example set of points where it is possible to find  $k$  clusters of radius  $r$  centered around arbitrary points, but impossible to find  $k$  clusters of radius  $r$  centered around points in the set.

**Solution:** Given arbitrary values of  $k$  and  $r$ , we can set up a counterexample in  $\mathbb{R}^2$  as follows. For each  $i \in \{1, \dots, k\}$ , create two points:  $p_{i,1} = (4ri, 0.9r)$  and  $p_{i,2} = (4ri, -0.9r)$ . For any pair  $p_{i,1}$  and  $p_{i,2}$  the distance is equal to  $1.8r$ , so any cluster centered on one of the points cannot contain the other. For all other pairs of points, their  $x$ -coordinates must differ by at least  $4r$ , and so they cannot belong to the same cluster. So there's no way to create  $k$  clusters centered on  $k$  points in the set. However, there is a way to create  $k$  clusters centered on  $k$  arbitrary points: for each  $i \in \{1, \dots, k\}$ , the point  $(4ri, 0)$  has distance  $\leq r$  to both  $p_{i,1}$  and  $p_{i,2}$ .

- (d) Consider the following algorithm, which is intended to compute the shortest distance among a collection of points in the plane:

- 1 Sort all points by their  $y$ -coordinate.
- 2 **for** each point in the sorted list:
- 3     Compute the distance to the next 7 points in the list.
- 4 **return** the smallest distance found.

Give an example where this algorithm will return a distance that is not in fact the overall shortest distance.

**Solution:** To construct a counterexample, it is sufficient to construct an example where the shortest distance is between two points that have at least 7 points between them in the ordering of  $y$ -coordinates. The example we use is as follows:

$$(0, -1) \quad (0, 5) \quad (0, 10) \quad (0, 15) \quad (0, 20) \quad (0, 25) \quad (0, 30) \quad (0, 35) \quad (0, 1)$$

In any ordering of these points by  $y$ -coordinate, the seven points with  $y$ -coordinate 0 will lie between  $(0, -1)$  and  $(0, 1)$ . Hence, the algorithm will never compute the Euclidean distance between  $(0, -1)$  and  $(0, 1)$ , and so it cannot find the true shortest distance.

**Problem 3. Estate Showing.** [30 points] (3 parts)

Trip Trillionaire is planning to give potential buyers private showings of his estate, which can be modeled as a weighted, directed graph  $G$  containing locations  $V$  connected by one-way roads  $E$ . To save time, he decides to do  $k$  of these showings at the same time, but because they were supposed to be private, he doesn't want any of his clients to see each other as they are being driven through the estate.

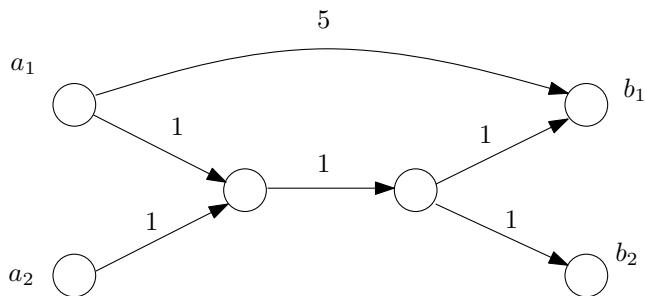
Trip has  $k$  grand entrances to his estate,  $A = \{a_1, a_2, \dots, a_k\} \subset V$ . He wants to take each of his buyers on a path through  $G$  from their starting location  $a_i$  to some ending location in  $B = \{b_1, b_2, \dots, b_k\} \subset V$ , where there are spectacular views of his private mountain range.

Because of your prowess with algorithms, he hires you to help him plan his buyers' visits. His goal is to find a path for each buyer  $i$  from the entrance they take,  $a_i$ , to any ending location  $b_j$  such that no two paths share any edges, and no two buyers end in the same location  $b_j$ .

- (a) Trip tells you his idea: find all-pairs shortest paths on  $G$ , and then try to select  $k$  of those shortest paths  $a_i \rightsquigarrow b_j$  such that all  $k$  paths start and end at different vertices and no two paths share any edges.

Give a graph where there exists a set of paths satisfying the requirements, but where Trip's strategy won't find it.

**Solution:** Consider this graph:



The all-pairs shortest paths algorithm would find all shortest paths from  $a_i$  to  $b_j$  ( $a_1 \rightsquigarrow b_1$ ,  $a_1 \rightsquigarrow b_2$ ,  $a_2 \rightsquigarrow b_1$ , and  $a_2 \rightsquigarrow b_2$ ), which all go through the same edge. Trip's algorithm, which considers only those paths, would find no solution. There are two completely disjoint paths using the shortest path  $a_2 \rightsquigarrow b_2$  and the direct edge  $(a_1, b_1)$ , but Trip's algorithm would not find this combination because one of them is not a shortest path.

- (b)** Rather than using shortest paths, you think that perhaps you can formulate this as a flow problem. Find an algorithm to find  $k$  paths  $a_i \rightsquigarrow b_j$  that start and end at different vertices and that share no edges, and briefly justify the correctness and running time of your algorithm.

**Solution:** The algorithm is as follows:

1. Create a flow network  $G'$  containing all vertices in  $V$ , all directed edges in  $E$  with capacity 1, and additionally a source vertex  $s$  and a sink vertex  $t$ . Connect the source to each starting location with a directed edge  $(s, a_i)$  and each ending location to the sink with a directed edge  $(b_j, t)$ , all with capacity 1.
2. Run Ford-Fulkerson on this network to get a maximum flow  $f$  on this network. If  $|f| = k$ , then there is a solution; if  $|f| < k$ , then there is no solution, so we return FALSE. We will later show that it is always the case that  $|f| \leq k$ .
3. To extract the paths from  $a_i$  to  $b_j$  (as well as which starting location ultimately connects to which ending location), run a depth-first search on the returned max flow  $f$  starting from  $s$ , tracing a path to  $t$ . Remove these edges and repeat  $k$  times until we have the  $k$  disjoint paths.

The running time for this algorithm is  $O(k|E|)$ : in the modified graph  $G'$ , we have  $|E| + 2k = O(|E|)$  edges, and the maximum possible flow value is  $k$ ; therefore, running Ford-Fulkerson takes  $O(k|E|)$  time. Extracting paths takes at most  $O(|E|)$  time, as we traverse each edge at most once. In total, then, the algorithm takes  $O(k|E|)$  time.

Some students used Edmonds-Karp, which runs in  $O(VE^2)$  time. In this case, because  $k = O(|V|)$ , Ford-Fulkerson provides a better bound.

To show correctness, we must show that there is a flow of size  $k$  if and only if there are  $k$  disjoint paths.

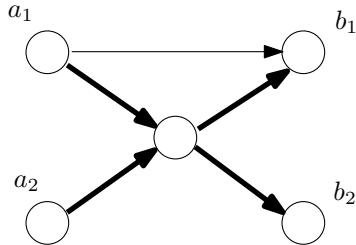
( $\rightarrow$ ) **Claim.** An integral flow of size  $f$  in a network with unit capacities can be decomposed into a unit flow over a simple path  $p : s \rightsquigarrow t$  and a flow of size  $f - 1$  which does not use  $p$ .

**Proof.** Pick any simple path  $p : s \rightsquigarrow t$  over edges with non-zero flow. One must exist (otherwise, we would have a cut with no flow). In the residual graph, that path has capacity 1; send an augmenting flow of -1 over that path. We then have a resulting flow of size  $f - 1$ , and the flow on each of the edges of  $p$  is 0.

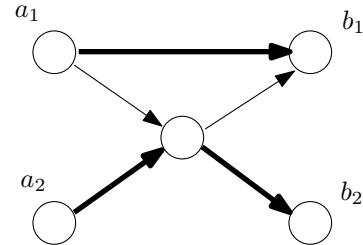
( $\leftarrow$ ) If there is a set of  $k$  disjoint paths from  $a_i$  to  $b_j$ , then the flow with one unit of flow on each of the edges contained in those paths defines a flow of size  $k$ . Because the cut  $(s, V' - s)$  has capacity  $k$ , this is a maximum flow on the network.

A common incorrect solution involved using the augmenting paths found by Ford-Fulkerson as the  $k$  paths for the buyers to use. This approach doesn't work because the augmenting paths are not guaranteed to be the final paths along which there are units of flow.

- (c) Trip, after trying out the paths found by your algorithm, realizes that making sure the  $k$  paths don't share edges isn't enough: it's possible that some paths will share vertices, and his buyers might run into each other where their paths intersect.



$a_1 \rightsquigarrow b_1$  and  $a_2 \rightsquigarrow b_2$  share a vertex



$a_1 \rightsquigarrow b_1$  and  $a_2 \rightsquigarrow b_2$  share neither vertices nor edges

Modify your algorithm to find  $k$  paths  $a_i \rightsquigarrow b_j$  that start and end in different locations, and that share neither vertices nor edges.

**Solution:** Duplicate each vertex  $v$  into two vertices  $v_{in}$  and  $v_{out}$ , with a directed edge between them. All edges  $(u, v)$  now become  $(u, v_{in})$ ; all edges  $(v, w)$  now become  $(v_{out}, w)$ . Assign the edge  $(v_{in}, v_{out})$  capacity 1.

With this transformation, we now have a graph in which there is a single edge corresponding to each vertex, and thus any paths that formerly shared vertices would be forced to share this edge.

Now, we can use the same algorithm as in part (b) on the modified graph to find  $k$  disjoint paths sharing neither edges nor vertices, if they exist.

The transformation of the graph takes  $O(|E|)$  time, as we are adding that many new vertices and edges to the graph. The new graph has  $O(|E| + |V| + 2k) = O(|E|)$  edges, and therefore the running time is unchanged.

Many students excluded the starting and ending locations  $a_i$  and  $b_i$  from the vertices duplicated. This approach, however, would overlook any paths that might have gone through some  $a_i$  or  $b_i$  before reaching their final destinations. (There was nothing in the previous problem statement preventing this from occurring.)

As in part (b), some students also tried to modify the operation of Ford-Fulkerson by removing all edges incident to vertices in the augmenting paths found, thus not allowing any other augmenting paths to go through those vertices. Again, this approach doesn't work because it is too greedy: the augmenting paths found by Ford-Fulkerson are not necessarily the final paths along which there are units of flow, and therefore this approach may miss valid solutions.

**Problem 4. Credit Card Optimization [30 points] (3 parts)**

Zelda Zillionaire is planning on making a sequence of purchases with costs  $x_1, \dots, x_n$ . Ideally, she would like to make all of these purchases on one of her many credit cards. Each credit card has credit limit  $\ell$ . Zelda wants to minimize the number of credit cards that she needs to use to make these purchases, without exceeding the credit limit on any card. More formally, she wants to know the smallest  $k$  such that there exists  $\pi : \{1, \dots, n\} \rightarrow \{1, \dots, k\}$  assigning each purchase  $i$  to a credit card  $j$ , where  $\pi$  must satisfy the following constraint:

$$\forall j \in \{1, \dots, k\} : \sum_{\substack{i \in \{1, \dots, n\} \\ \text{s.t. } \pi(i)=j}} x_i \leq \ell.$$

Zelda is thinking of using the following algorithm to distribute her purchases:

```

1 Create an empty list of credit cards  $L$ .
2 for each purchase  $x_i$ :
3     for each credit card  $j \leq |L|$ :
4         if  $L[j] + x_i \leq \ell$ :
5             Purchase  $x_i$  with credit card  $j$ .
6             Set the balance for card  $j$  to  $L[j] = L[j] + x_i$ .
7             Skip to the next purchase.
8         if no existing credit card has enough credit left:
9             Purchase  $x_i$  with a new credit card.
10            Append a new credit card to  $L$ .
11            Set the balance of the new credit card to  $x_i$ .
12 return  $k = |L|$ 

```

- (a) Give an example where Zelda's algorithm will not use the optimal number of credit cards.

**Solution:**

Let the credit limit  $\ell$  be 10. Consider the following sequence of purchases as input to Zelda's algorithm: 2, 1, 9, 8. The optimal solution would only use 2 credit cards by grouping purchases {2, 8} and {9, 1}. Zelda's algorithm would split the charges into 3 cards: {2, 1}, {9}, {8}, which is suboptimal.

This problem part was worth 5 points. Almost everyone got this problem right. Some students didn't read carefully the pseudocode and inverted the order of the two loops.

- (b) Show that Zelda's algorithm gives a 2-approximation for the number of credit cards.

**HINT:** Let  $OPT$  be the optimal number of credit cards. In order for the algorithm to add a new credit card  $j > OPT$ , it must reach  $x_i$  such that  $x_i + \min_{j' \in \{1, \dots, OPT\}} L[j'] > \ell$ . It will then set  $L[j] = x_i$ .

**Solution:**

First, we show that the following loop invariant holds. At no point are there two distinct cards  $i$  and  $j$ , where  $i < j$  such that the balance on each card  $L[j] \leq \frac{\ell}{2}$  and  $L[i] \leq \frac{\ell}{2}$ . Suppose to the contrary that there are two such cards. Then the remaining credit on card  $i$  was sufficient to transfer all of the purchases  $x_i$  from the new credit card to the old credit card  $j$ , instead of opening up a new credit card.

We prove that Zelda's algorithm gives a 2-approximation by contradiction. Namely, we assume that there is a case where at least  $2OPT + 1$  cards are needed. By the loop invariant we know that no 2 credit cards have less than  $\frac{\ell}{2}$  as a balance. So, assuming that there are no zero purchase charges, the total amount of purchases must be more than  $2OPT \cdot \frac{\ell}{2}$ , which is the maximum amount that the  $OPT$  cards in the optimal solution can hold. This is a contradiction. So, we can always find at most  $2OPT$  cards to pay for all of the purchases.

This problem part was worth 10 points.

- (c) Show that minimizing the number of credit cards used is NP-hard.

**Solution:**

We prove that minimizing the number of credit cards is NP-hard by reducing SET-PARTITION to the Zelda's problem in polynomial time.

First, we recast Zelda's optimization problem as a decision problem. Let the decision problem be formulated as follows: For a given target number of cards  $k$ , we are able to come up with an assignment of purchases  $x_1 \dots x_n$  to  $k$  credit cards, where the cumulative balance on each credit card can not exceed the limit  $l$ .

Recall that we defined SET-PARTITION as follows:

Given set  $S = \{s_1, \dots, s_n\}$  we can partition them into 2 disjoint subsets  $A$  and  $B$  such that  $\max\{\sum_{i \in A} s_i, \sum_{j \in B} s_j\}$  is minimized.

The decision version of the SET-PARTITION problem can be reformulated as the question whether the numbers can be partitioned into two sets  $A$  and  $B = S - A$  such that  $\sum_{i \in A} s_i = \sum_{j \in B} s_j$ .

Given an input set  $S = \{s_1, \dots, s_n\}$  to the SET-PARTITION problem, we can use the elements in the set as the purchases in Zelda's problem, set the number of cards to be 2 and the credit card limit  $l = \frac{1}{2} \sum_{i \in S} s_i$ . Clearly, if the answer to the Zelda's decision problem is in the affirmative, then each credit card will have balance of exactly  $\frac{l}{2}$ , and we can take the disjoint subsets of purchases on each card to be the two sets  $A$  and  $B$  that would satisfy SET-PARTITION. The reduction can be executed in linear time.

We also accepted the reduction from SUBSET-SUM to Zelda.

We gave 5 points for correctly identifying an NP-hard problem from which one could reduce to Zelda's problem to prove that the problem is NP-hard (the direction of the reduction needed to be correct to receive all the points). 10 points was given for a correct reduction and analysis.

**Problem 5.** Well-Separated Clusters [30 points] (3 parts)

Let  $S = \{(x_1, y_1), \dots, (x_n, y_n)\}$  be a set of  $n$  points in the plane. You have been told that there is some  $k$  such that the points can be divided into  $k$  clusters with the following properties:

1. Each cluster has radius  $r$ .
2. The center of each cluster has distance at least  $5r$  from the center of any other cluster.
3. Each cluster contains at least  $\epsilon n$  elements.

Your goal is to use this information to construct an algorithm for computing the number of clusters.

- (a) Give an algorithm that takes as input  $r$  and  $\epsilon$ , and outputs the exact value of  $k$ .

**Solution:**

**Executive Summary.** We find all points sufficiently close to a given point, remove all such points and increment the number of clusters, and repeat until we run out of points. This procedure is exact and takes  $O(nk)$  time.

**Algorithm** Initialize the number of clusters  $k \equiv 0$ , and the set  $S' \equiv S$ . Choose a point  $p \in S'$ , and compute the distance from  $p$  to every other point in  $S'$ . Remove  $p$  and all points within distance  $2r$  from  $S'$ , and increment  $k$ . Repeat this procedure until  $S'$  is empty; at that point, return  $k$ .

**Correctness** We first note that if the point  $p$  chosen in an iteration lies in a cluster with center  $c$ , then all points within this cluster will be removed in the same iteration. Indeed, if  $q$  is any other point in this cluster, by the triangle inequality,

$$d(p, q) \leq d(p, c) + d(c, q) \leq r + r = 2r.$$

Furthermore, no points from other clusters will be removed in the same iteration. Indeed, if  $q'$  is in another cluster with center  $c'$ , then by the triangle inequality

$$\begin{aligned} d(p, q') &= (d(c, p) + d(p, q') + d(q', c')) - d(c, p) - d(q', c') \\ &\geq d(c, c') - d(c, p) - d(q', c') \\ &\geq 5r - r - r = 3r. \end{aligned}$$

**Runtime Analysis** Each iteration computes at most  $n$  distances, and removes exactly 1 cluster. Hence there are  $k$  iterations total, so the overall runtime is  $O(nk)$ . Since each cluster contains at least  $\epsilon n$  points, there are at most  $1/\epsilon$  clusters, so the runtime can also be phrased as  $O(n/\epsilon)$ .

**Grading comments** This part was worth 12 marks total, with 6 marks allotted for a correct and efficient algorithm, 4 for justification of correctness, and 2 for runtime analysis. Some students omitted the latter two parts and were duly penalized, since both are expected in this course.

Correct algorithms which were slightly inefficient (e.g  $O(n^2)$  or similar) were deducted one mark independent of analysis; this often occurred if all pairs of distances were computed, or a disjoint-set structure used to find the actual clusters (both were unnecessary). More inefficient or incorrect algorithms were given at most 3 marks.

Several students noted that this was a variant of Hierarchical Agglomerative Clustering as discussed in class, which is true; however, justification of correctness was still necessary since the latter is an approximation algorithm. This observation was not required for full credit.

- (b) Given a particular cluster  $C_i$ , if you sample  $t$  points uniformly at random (with replacement), give an upper bound in terms of  $\epsilon$  for the probability that none of these points are in  $C_i$ .

**Solution:**

**Solution** The number of points in  $C_i$  is at least  $\epsilon n$ , so the probability that a point sampled u.a.r. lies within  $C_i$  is at least  $\epsilon n/n = \epsilon$ . Therefore the probability that a point sampled u.a.r. *doesn't* lie within  $C_i$  is at most  $1 - \epsilon$ . Since the points are chosen independently, the probability that  $t$  points sampled u.a.r. don't lie within  $C_i$  is at most  $(1 - \epsilon)^t$ .

**Grading comments** This problem was relatively straightforward; the full 8 marks were given for the correct answer. An alternative (slightly weaker) bound using Chernoff was also given full credit as long as it was computed correctly. When the answer was incorrect, part marks were allotted for similarity to the solution and evidence of work.

- (c) Give a sublinear-time algorithm that takes as input  $r$  and  $\epsilon$ , and outputs a number of clusters  $k$  that is correct with probability at least  $2/3$ .

**Solution:**

**Executive Summary** Use the algorithm from part (a) on a subset of the points chosen u.a.r. The size of the subset is chosen so that the algorithm is correct with the desired probability.

**Algorithm** Initialize  $S'$  to be a subset of  $S$  of size  $t$  (to be determined below), where the points are chosen uniformly at random. Run the same algorithm as in part (a) on this subset.

**Correctness** From part (b), the probability that  $t$  points sampled u.a.r. don't lie in a given cluster is  $(1 - \epsilon)^t$ . Hence, by the union bound, the probability that  $t$  points sampled u.a.r. don't lie in *any* cluster is at most  $k(1 - \epsilon)^t$ ; since  $k \leq 1/\epsilon$ , this can be upper bounded by  $\frac{1}{\epsilon}(1 - \epsilon)^t$ . The algorithm is incorrect iff the  $t$  sampled points miss any cluster, so we want this probability to be at most one third:

$$\begin{aligned} \frac{1}{\epsilon}(1 - \epsilon)^t &\leq \frac{1}{3} \\ (1 - \epsilon)^t &\leq \frac{\epsilon}{3} \\ t \log(1 - \epsilon) &\leq \log \frac{\epsilon}{3} \\ t &\geq \frac{\log \frac{\epsilon}{3}}{\log(1 - \epsilon)} \end{aligned}$$

(the sign is reversed in the last inequality since  $\log(1 - \epsilon)$  is negative). The rest of justification of correctness follows from part (a).

**Runtime analysis** The runtime of this algorithm is  $O(tk)$  or  $O(t/\epsilon)$ ; the analysis is similar to that for part (a). Since neither  $t$  nor  $\epsilon$  depend on  $n$ , this algorithm is sublinear.

**Grading comments** This part was worth 10 marks total, with 6 marks for achieving the correct approach to the sublinear algorithm and 4 marks for computing the size of the subset correctly.

Some students failed to realize that part (b) gives the probability of missing a *fixed* cluster, whereas this part requires the probability of missing *any* cluster; this resulted in a deduction of 2 marks. Others erroneously assumed that the events of missing each cluster were independent; this resulted in a deduction of 1 mark. Incorrect algorithms were given partial credit depending on their similarity with the correct solution.

Several students noted that this was a variant of the sublinear Connected Components algorithm as discussed in class; again, while true, this was not necessary and a correct calculation of the subset size was still required.

**Problem 6. Looking for a Bacterium [30 points] (2 parts)**

Imagine you want to find a bacterium in a one dimensional region divided into  $n$   $1\text{-}\mu\text{m}$  regions. We want to find in which  $1\text{-}\mu\text{m}$  region the bacterium lives. A microscope of resolution  $r$  lets you query regions 1 to  $r$ ,  $r + 1$  to  $2r$ , etc. and tells you whether the bacteria is inside that region. Each time we operate a microscope to query one region, we need to pay  $(n/r)$  dollars for electricity (microscopes with more precise resolution take more electricity). We also need to pay an  $n$ -dollar fixed cost for each type of microscope we decide to use (independent of  $r$ ).

- (a) Suppose that you decide to purchase  $\ell = \lg n$  microscopes, with resolutions  $r_1, \dots, r_\ell$  such that  $r_i = 2^{i-1}$ . Give an algorithm for using these microscopes to find the bacterium, and analyze the combined cost of the queries and microscopes. **Solution:**

**Executive Summary** We will use all the microscope to perform a binary search. Starting with the lowest resolution microscope, we use successively more powerful microscopes and eventually using the microscope of resolution 1 to locate exactly where the bacterium is.

**Algorithm** We will assume  $n$  is a power of 2, if not, we can imagine expanding to a larger region that is a power of 2. This will not affect the asymptotic cost. We first use a microscope of resolution  $n/2$  to query the region  $(1, n/2)$ . Regardless of whether the answer is yes or no, we can eliminate a region of size  $n/2$ . We then do the same with a microscope of resolution  $n/4$ , and so on.

**Correctness** This is essentially the same as a binary search, we use each type of microscope exactly once, and eventually we will pin point the location of the bacteria to a region of size 1.

**Cost analysis** We are only interested in the total electricity cost for this problem. Some students provided runtime analysis, but their correctness were not graded. We use each microscope exactly once, starting from the the one with the lowest resolution, it costs us  $2, 4, 8, \dots, n$  to use each microscope. The last cost of  $n$  comes from using the microscope of resolution 1. This geometric series sums to a total of  $2n - 2$ , thus the electricity cost is  $O(n)$ . The total cost of purchasing the microscope is  $n \lg n$ , for a combined cost of  $O(n \lg n)$ .

**Grading comments** The maximum points for this part is 10. 3 points are awarded for people who can roughly describe the algorithm. 5 to 7 points are awarded for people who fail to correctly analyze the cost of electricity to various degrees. Points are also deducted if you simply assert the total cost of electricity is  $O(n)$ , we are looking for at least a mentioning of things such as this geometric sum converges.

Most students did pretty well on this part, some students attempted to use recurrence to solve for the cost of electricity, but most of them made mistakes.

- (b) Give a set of microscopes and an algorithm to find the bacterium with total cost  $O(n \lg \lg n)$ .

**Solution:**

**Executive Summary** We borrow our idea from the vEB data structure. We will purchase microscopes of resolutions  $n^{1/2}, n^{1/4}, n^{1/8}, \dots, n^{1/2^i} = 2$ . Clearly,  $i = \lg \lg n$ . So we spend  $n \lg \lg n$  for buying microscopes. We do a linear search at each level, and we will show that the total cost of electricity is also  $n \lg \lg n$ .

**Algorithm** We perform a search for the bacterium as follows. First we use the microscope with resolution  $\sqrt{n}$  to locate which region of size  $\sqrt{n}$  the bacterium is in. This will take at most  $\sqrt{n}$  applications of this microscope. We can then narrow our search in the region of size  $\sqrt{n}$  and use microscopes of resolution  $n^{1/4}$  to continue this process, eventually we can narrow down the location of the bacterium to a region of size 2. From there, it takes one simple application of a microscope of resolution 1 to find the bacterium.

**Correctness** This collection of microscopes allows us to search every region exhaustively, thus we can always find the bacterium.

**Cost analysis** The total purchase of  $\lg \lg n$  microscopes will cost us  $n \lg \lg n$  dollars. At stage 1, we use the microscope of resolution  $\sqrt{n}$  a total of  $\sqrt{n}$  times. Each use of this microscope costs  $\frac{n}{\sqrt{n}}$  dollars, for a total of  $n$  dollars. In fact, at every subsequent step, we always apply the microscope of resolution  $\sqrt{r}$  a total of  $\sqrt{r}$  times to completely scan a region of size  $r$ . The total cost of this is always  $n$ . Thus, the total electricity cost associated with using each type of microscope is at most  $n$ . Therefore, the total electricity cost is also  $n \lg \lg n$ . Our combined total cost is thus  $O(n \lg \lg n)$ .

**Grading comments** The maximum points for this part is 20. 2 points are given for some very vague mentioning of vEB, demonstrating some level of understanding. 5 points are given for the correct choice of microscope resolutions (but not how to use them to search for the bacterium). 10 points are given if the algorithm is correct (correct choice of resolutions as well as a description of how to use them to search for the bacterium). 13 points are awarded if the electricity cost is analyzed incorrectly, but has some correct ideas. 16 to 18 points are awarded if there are some minor mistakes in the electricity cost analysis. Many students attempted to analyze the electricity cost using recurrence, arriving at  $C(n) = C(\sqrt{n}) + O(n)$ . The idea is that the cost to search a region of size  $n$  is equal to spending  $O(n)$  cost to apply the microscope of resolution  $\sqrt{n}$  a total of  $\sqrt{n}$  times, and then recurse on a region of size  $\sqrt{n}$ . This

problem with this approach is that the  $O(n)$  term is a constant. Every level costs at most  $n$  dollars, but this cost does not scale down with region size. Therefore, the correct way to setup the recurrence is to distinguish the variables from the constants, as  $C(k) = C(\sqrt{k}) + n$ . this indeed gives us an answer of  $C(k) = n \lg \lg k$ .

## Practice Quiz 1

### Problem 1. Algorithms and running times (5 parts) [5 points]

Match each algorithm below with the tightest asymptotic upper bound for its worst-case running time by inserting one of the letters A, B, . . . , E into the corresponding box. **Some running times may be used multiple times or not at all.** For sorting algorithms,  $n$  is the number of input elements. For matrix algorithms, the input matrix has size  $n \times n$ .

You need not justify your answers. Because points will be deducted for wrong answers, do not guess unless you are reasonably sure.

Insertion sort

A:  $O(\lg n)$

Binary Search

B:  $O(n)$

BUILD-HEAP

C:  $O(n \lg n)$

Strassen's

D:  $O(n^3)$

Randomized Quicksort

E:  $O(n^2)$

F:  $O(n^{\lg 7})$

**Solution:** From top to bottom: E, A, B, F, E.

**Problem 2. Recurrences** (3 parts) [9 points]

Solve the following recurrences by giving tight  $\Theta$ -notation bounds. You do not need to justify your answers, but any justification that you provide will help when assigning partial credit.

(a)  $T(n) = T(\sqrt{n}) + \Theta(\lg \lg n)$

**Solution:** We start by using a change of variable,  $n = 2^m$ , and we define  $S(m) = T(2^m)$ . This translates the recurrence into  $S(m) = S(m/2) + \Theta(\lg m)$ . By the master method,  $S(m) = \Theta((\lg m)^2)$ . By changing the variable back to  $n$ , we obtain  $T(n) = \Theta((\lg \lg n)^2)$ .

(b)  $T(n) = T(n/2 + \sqrt{n}) + \sqrt{6046}$

**Solution:** We neglect the lower order term  $\sqrt{n}$  and we use the master method to obtain  $T(n) = \Theta(\lg n)$ . We may use the substitution method to verify this answer.

(c)  $T(n) = T(n/5) + T(4n/5) + \Theta(n)$

**Solution:** We use a recursion tree. The sum of the terms at each level is equal to  $n$  and the height of the tree is  $\lg n$ . This gives  $T(n) = \Theta(n \lg n)$ .

**Problem 3. Short Answers (4 parts) [16 points]**

Give *brief*, but complete, answers to the following questions.

- (a) Argue that you cannot have a Priority Queue in the comparison model with both the following properties.

- EXTRACT-MIN runs in  $\Theta(\lg \lg n)$  time.
- BUILD-HEAP runs in  $\Theta(n)$  time.

**Solution:** If such a priority queue exists, then, we can sort in  $\Theta(n \lg \lg n)$  in the comparison model by using BUILD-HEAP and applying EXTRACT-MIN  $n$  times. This contradicts the lower bound on the running time of sorting algorithms in the comparison model.

- (b) A sequence of  $n$  operations is performed on a data structure. The  $i$ th operation costs  $i$  if  $i$  is a power of two, and one otherwise. Determine the amortized cost per operation.

**Solution:** Over the course of  $n$  operations, we have at most  $\lceil \lg n \rceil$  operations that cost  $i$  and at most  $n - \lfloor \lg n \rfloor = \Theta(n)$  operations that cost 1. Thus, we have a total cost over  $n$  operations of

$$\Theta(n) + \sum_{i=0}^{\lceil \lg n \rceil} 2^i = \Theta(n) + \left( \frac{2^{\lceil \lg n \rceil} - 1}{2 - 1} \right) = \Theta(n)$$

Because we are averaging over  $n$  operations, we have an amortized cost of  $\Theta(n)/n = \Theta(1)$ .

- (c) What does it mean to sort *in place*, and what is one advantage of sorting in place? Which of the following algorithms sort in place?

- INSERTION-SORT
- MERGE-SORT
- HEAPSORT
- COUNTING-SORT

**Solution:** An in place sorting algorithm does not use auxiliary array and sorts items by moving them around in the same array. One advantage of in place sorting is that it

uses less memory, and has better cache performance. INSERTION-SORT and HEAP-SORT work in place, while MERGE-SORT and COUNTING-SORT do not.

- (d) If an algorithm has running time  $T(m) \leq 2^m$  for all  $m$  which are powers of 2, and  $T(n)$  is monotonically increasing, then can we conclude that  $T(n) = O(2^n)$  by using the sloppiness lemma?

**Solution:** Since  $f(n) = 2^n$  is an exponential function, it does not grow slowly, and therefore sloppiness lemma does not apply.

- (e) Consider the following collection  $\mathcal{H} = \{h_1, h_2, h_3\}$  of hash functions, where the three hash functions map the universe  $\{A, B, C, D\}$  of keys into the range  $\{0, 1, 2\}$  according to the following table:

$x$	$h_1(x)$	$h_2(x)$	$h_3(x)$
$A$	1	0	2
$B$	1	2	0
$C$	2	2	2
$D$	2	0	0

Is this collection of hash functions universal?

**Solution:** Yes. A hash family  $\mathcal{H}$  that maps a universe of keys  $U$  into  $m$  slots is *universal* if for each pair of distinct keys  $x, y \in U$ , the number of hash functions  $h \in \mathcal{H}$  for which  $h(x) = h(y)$  is exactly  $|\mathcal{H}|/m$ . In this problem,  $|\mathcal{H}| = 3$  and  $m = 3$ . Therefore, for any pair of the four distinct keys, exactly 1 hash function should make them collide. By consulting the table above, we have:

$$\begin{aligned}
 h(A) &= h(B) && \text{only for } h_1 \text{ mapping into slot 1} \\
 h(A) &= h(C) && \text{only for } h_3 \text{ mapping into slot 2} \\
 h(A) &= h(D) && \text{only for } h_2 \text{ mapping into slot 0} \\
 h(B) &= h(C) && \text{only for } h_2 \text{ mapping into slot 2} \\
 h(B) &= h(D) && \text{only for } h_3 \text{ mapping into slot 0} \\
 h(C) &= h(D) && \text{only for } h_1 \text{ mapping into slot 2}
 \end{aligned}$$

**Problem 4. True or False, and Justify** (7 parts) [28 points]

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. The more content you provide in your justification, the higher your grade, but be brief. Your justification is worth more points than your true-or-false designation.

- T F** There exists a pivot selection algorithm such that quicksort on  $n$  numbers runs in  $O(n \lg n)$  time in the worst case.

**Solution:** **True.** In  $O(n)$  time we deterministically find the median and we use this median as the pivot.

- T F** Let  $f$  and  $g$  be asymptotically nonnegative functions. Then, at least one relationship of  $f(n) = O(g(n))$  and  $g(n) = O(f(n))$  must always hold.

**Solution:** **False.** For  $f(n) = 1$  and  $g(n) = \|n * \sin(n)\|$  it is false.

- T F** Suppose we use a hash function  $h$  to hash  $n$  distinct keys into an array  $T$  of length  $m$ . Assuming simple uniform hashing, the expected number of colliding pairs is  $\Omega((\log n)/m)$ .

- T F** Suppose that an array contains  $n$  numbers, each of which is  $-1$ ,  $0$ , or  $1$ . Then, the array can be sorted in  $O(n)$  time in the worst case.

**Solution:** **True.** We may use counting sort. We first add 1 to each of the elements in the input array such that the precondition of counting sort is satisfied. After running counting sort, we subtract 1 from each of the elements in the sorted output array.

A solution based on partitioning is as follows. Let  $A[1 \dots n]$  be the input array. We define the invariant

- $A[1 \dots i]$  contains only  $-1$ ,
- $A[i + 1 \dots j]$  contains only  $0$ , and
- $A[h \dots n]$  contains only  $+1$ .

Initially,  $i = 0$ ,  $j = 0$ , and  $h = n + 1$ . If  $h = j + 1$ , then we are done; the array is sorted. In the loop we examine  $A[j + 1]$ . If  $A[j + 1] = -1$ , then we exchange  $A[j + 1]$  and  $A[i + 1]$  and we increase both  $i$  and  $j$  with 1 (as in partition in quicksort). If  $A[j + 1] = 0$ , then we increase  $j$  with 1. Finally, if  $A[j + 1] = +1$ , then we exchange  $A[j + 1]$  and  $A[h - 1]$  and we decrease  $h$  by 1.

**T F** Suppose that a hash table of  $m$  slots contains a single element with key  $k$  and the rest of the slots are empty. Suppose further that we search  $r$  times in the table for various other keys not equal to  $k$ . Assuming simple uniform hashing, the probability is  $r/m$  that at least one of the  $r$  searches probes the slot containing the single element stored in the table.

**Solution:** **False.** The probability  $p$  that one of the  $r$  searches collides with the single element stored in the table is equal to 1 minus the probability that none of the  $r$  searches collides with the single element stored in the table. That is,  $p = 1 - (1 - 1/m)^r$ .

**T F** On all input arrays consisting of more than a 1000 elements, QUICKSORT performs at most as many comparisons as INSERTION-SORT.

**Solution:** **False.** If the input is already sorted, then QUICKSORT performs  $n \lg n$  comparisons and insertion sort only needs  $n$  comparisons.

**T F** Bucket sort can be used to sort an arbitrary list of real numbers in  $O(n)$  expected time.

**Solution:** **False.** Violates the lower bound on comparison sorting. Bucket sort assumes a distribution on the inputs. It is not a linear time algorithm for arbitrary lists of real

numbers since all numbers can be chosen to fall in the same bucket.

**Problem 5. Sorting a partially-sorted array (3 parts) [10 points]**

In this problem, more efficient algorithms will be given more credit. Partial credit will be given for correct but inefficient algorithms.

Let  $A_0$  be a numerical array of length  $n$ , originally sorted into ascending order. Assume that  $k$  entries of  $A_0$  are overwritten with new values, producing an array  $A$ . Furthermore assume you have an array  $B$  containing  $n$  boolean values, where  $B[i]$  is true if  $A[i]$  is one of the  $k$  values that was overwritten, and false otherwise.

- (a) Give a fast algorithm to sort  $A$  into ascending order, with time complexity better than  $O(nk)$ . [5 points]

**Solution:** A straightforward solution is: (i) Separate out  $A$  into two lists,  $A_1$  consisting of all elements of  $A$  where the corresponding element of  $B$  is false, and  $A_2$  where the corresponding element is true. (ii) Sort  $A_2$  using mergesort or heapsort. (iii) Perform a linear merge of  $A_1$  and  $A_2$ , writing the result back into  $A$ .

Partial credit was given for solutions based on insertion sort.

- (b) Give the time complexity of your algorithm in big-O notation, as a function of  $n$  and  $k$ . [3 points]

**Solution:** Separation of lists:  $O(n)$ . Sorting new items:  $O(k \lg k)$ . Merging back together again:  $O(n)$ . Total time:  $O(n + k \lg k)$ . Note that if the algorithm given is correct, then any correctly-demonstrated time bound less than  $O(nk)$  for the algorithm may be given here.

- (c) Give the space complexity of your algorithm in big-O notation, as a function of  $n$  and  $k$ . (Do not include the space required for  $A$  and  $B$ .) [2 points]

**Solution:**  $O(n)$ . There are more efficient implementations that overwrite parts of  $A$  and  $B$  as they go, but some of these approaches sacrifice time efficiency for space efficiency, and optimization of space was not asked for in this question. As long as the algorithm is correct and runs in time less than  $O(nk)$ , the correct space complexity for the algorithm is all that is required here.

**Problem 6. Tree Ancestors**

Suppose you are given a complete binary tree of height  $h$  with  $n = 2^h$  leaves, where each node and each leaf of this tree has an associated “value”  $v$  (an arbitrary real number).

If  $x$  is a leaf, we denote by  $A(x)$  the set of ancestors of  $x$  (including  $x$  as one of its own ancestors). That is,  $A(x)$  consists of  $x$ ,  $x$ ’s parent, grandparent, etc. up to the root of the tree.

Similarly, if  $x$  and  $y$  are distinct leaves we denote by  $A(x, y)$  the ancestors of *either*  $x$  or  $y$ . That is,

$$A(x, y) = A(x) \cup A(y) .$$

Define the function  $f(x, y)$  to be the sum of the values of the nodes in  $A(x, y)$ .

Give an algorithm (pseudo-code not necessary) that efficiently finds two leaves  $x_0$  and  $y_0$  such that  $f(x_0, y_0)$  is as large as possible. What is the running time of your algorithm?

**Solution:** There are several different styles of solution to this problem. Since we studied divide-and-conquer algorithms in class, we just give a divide-and-conquer solution here. There were also several different quality algorithms, running in  $O(n)$ ,  $O(n \lg n)$ , and  $O(n^2 \lg n)$ . These were worth up to 11, 9, and 4 points, respectively. A correct analysis is worth up to 4 points.

First, let us look at an  $O(n \lg n)$  solution then show how to make it  $O(n)$ . For simplicity, the solution given here just finds the maximum value, but it is not any harder to return the leaves giving this value as well.

We define a recursive function  $\text{MAX1}(z)$  to return the maximum value of  $f(x)$ —the sum of the ancestors of a single node—over all leaves  $x$  in  $z$ ’s subtree. Similarly, we define  $\text{MAX2}(z)$  to be a function returning the maximum value of  $f(x, y)$  over all pairs of leaves  $x, y$  in  $z$ ’s subtree. Calling  $\text{MAX2}$  on the root will return the answer to the problem.

First, let us implement  $\text{MAX1}(z)$ . The maximum path can either be in  $z$ ’s left subtree or  $z$ ’s right subtree, so we end up with a straightforward divide and conquer algorithm given as:

**MAX1( $z$ )**

1 **return** ( $\text{value}(z) + \max \{\text{MAX1}(\text{left}[z]), \text{MAX1}(\text{right}[z])\}$ )

For  $\text{MAX2}(z)$ , we note that there are three possible types of solutions: the two leaves are in  $z$ ’s left subtree, the two leaves are in  $z$ ’s right subtree, or one leaf is in each subtree. We have the following pseudocode:

**MAX2( $z$ )**

1 **return** ( $\text{value}(z) + \max \{\text{MAX2}(\text{left}[z]), \text{MAX2}(\text{right}[z]), \text{MAX1}(\text{left}[z]) + \text{MAX1}(\text{right}[z])\}$ )

**Analysis:**

For MAX1, we have the following recurrence

$$\begin{aligned} T_1(n) &= 2T_1\left(\frac{n-1}{2}\right) + \Theta(1) \\ &= \Theta(n) \end{aligned}$$

by applying the Master Method.

For MAX2, we have

$$\begin{aligned} T_2(n) &= 2T_2\left(\frac{n-1}{2}\right) + 2T_1\left(\frac{n-1}{2}\right) + \Theta(1) \\ &= 2T_2\left(\frac{n-1}{2}\right) + \Theta(n) \\ &= \Theta(n \lg n) \end{aligned}$$

by case 2 of the Master Method.

To get an  $O(n)$  solution, we just define a single function, MAXBOTH, that returns a pair—the answer to MAX1 and the answer to MAX2. With this simple change, the recurrence is the same as MAX1

**SCRATCH PAPER** — Please detach this page before handing in your exam.

**SCRATCH PAPER** — Please detach this page before handing in your exam.

## Practice Final Solutions

- Do not open this exam booklet until you are directed to do so. Read all the instructions first.
- When the exam begins, write your name on every page of this exam booklet.
- The exam contains seven multi-part problems. You have 180 minutes to earn 180 points.
- This exam booklet contains 17 pages, including this one. An extra sheet of scratch paper is attached. Please detach it before turning in your exam.
- This exam is closed book. You may use three handwritten A4 or  $8\frac{1}{2}'' \times 11''$  crib sheets. No calculators or programmable devices are permitted.
- 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, since the pages may be separated for grading.
- Do not waste time and paper rederiving facts that we have studied. It is sufficient to cite known results.
- 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.
- 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.
- Good luck!

Problem	Points	Grade	Initials
1	12		
2	56		
3	20		
4	25		
5	27		
6	20		
7	20		
Total	180		

Name: **Solutions** \_\_\_\_\_  
Circle your recitation letter and the name of your recitation instructor:

David      A      B      Steve      C      D      Hanson      E      F

**Problem 1. Algorithm Design Techniques [12 points]**

The following are a few of the design strategies we followed in class to solve several problems.

1. Dynamic programming.
2. Greedy strategy.
3. Divide-and-conquer.

For each of the following problems, mention which of the above design strategies was used (in class) in the following algorithms.

1. Longest common subsequence algorithm

**Solution:** Dynamic Programming

2. Minimum spanning tree algorithm (Prim's algorithm)

**Solution:** Greedy

3.Select

**Solution:** Divide-and-Conquer

4.Fast Fourier Transform

**Solution:** Divide-and-Conquer

**Problem 2. True or False, and Justify [56 points]**

Circle **T** or **F** for each of the following statements, and briefly explain why. The better your argument, the higher your grade, but be brief. Your justification is worth more points than your true-or-false designation. Each part is 4 points.

- (a) **T F** An inorder traversal of a Min Heap will output the values in sorted order.

**Solution:** False. Consider the Min Heap with 1 at the root and 3 as left child and 2 as right child.

- (b) **T F** A Monte-Carlo algorithm is a randomized algorithm that always outputs the correct answer and runs in expected polynomial time.

**Solution:** False. This is definition of a Las Vegas Algorithm.

- (c) **T F** Two distinct degree- $d$  polynomials with integer coefficients can evaluate to the same value in as many as  $d + 1$  distinct points.

**Solution:** False. They have same value on at most  $d$  distinct points.

- (d) **T F** The right subtree of an  $n$ -node 2-3-4 tree contains  $\Omega(n)$  nodes.

**Solution:** False. A tree with all degree-3 nodes on one subtree and degree-2 nodes on the other will have depth  $h = \log_3 n$ . There will be  $2^{\log_3 n} = n^{\log_3 2} = o(n)$  nodes on the sparse subtree. *Note: The prior version of the solutions incorrectly stated that this problem was true.*

- (e) **T F** If a problem in NP can be solved in polynomial time, then all problems in NP can be solved in polynomial time.

**Solution:** False. The decision version of MST is in NP, but this doesn't mean that all problems in NP can be solved in polynomial time.

- (f) **T F** If an NP-complete problem can be solved in linear time, then all NP-complete problems can be solved in linear time.

**Solution:** False. The reductions are *polynomial-time* but not necessarily *linear* time.

- (g) **T F** If single digit multiplication can be done in  $O(1)$  time, then multiplying two  $k$ -digit numbers can be done in  $O(k \log k)$  time.

**Solution:** True. Use FFT.

- (h) **T F** In a  $k$ -bit binary counter (that is initialized to zero and is always non-negative), any sequence of  $n < 2^k$  increments followed by  $m \leq n$  decrements takes  $O(m + n)$  total bit flips in the worst case, where a bit flip changes one bit in the binary counter from 0 to 1 or from 1 to 0.

**Solution:** True. As shown in lecture the amortized cost of an increment in a sequence of increments is  $O(1)$  and by the same argument, the amortized cost of a decrement in a sequence of decrements is  $O(1)$ .

- (i) **T F** Consider a directed graph  $G$  in which every edge has a positive edge weight. Suppose you create a new graph  $G'$  by replacing the weight of each edge by the negation of its weight in  $G$ . For a given source vertex  $s$ , you compute all shortest path from  $s$  in  $G'$  using Dijkstra's algorithm.

True or false: the resulting paths in  $G'$  are the longest (i.e., highest cost) simple paths from  $s$  in  $G$ .

**Solution:** False. Dijkstra's algorithm may not necessarily return the minimum weight path because this new graph may contain a negative weight cycle.

- (j) **T F** A spanning tree of a given undirected, connected graph  $G = (V, E)$  can be found in  $O(E)$  time.

**Solution:** True. Perform a walk on the graph starting from an arbitrary node.

- (k) T F Consider the following algorithm for computing the square root of a number  $x$ :

```
SQUARE-ROOT( $x$ )
For  $i = 1, \dots, x/2$ :
    if  $i^2 = x$  then output  $i$ .
```

True or False: This algorithm runs in polynomial time.

**Solution:** False. To run in polynomial-time, this algorithm would have to run in time polynomial in  $\lg x$ .

- (l) T F An efficient max-flow algorithm can be used to efficiently compute a maximum matching of a given bipartite graph.

**Solution:** True. Add a “supersource” and a “supersink”, each connected to a partition of the graph by unit capacity edges.

- (m) **T F** The following recurrence has solution  $T(n) = \Theta(n \lg(n^2))$ .

$$T(n) = 2T(n/2) + 3 \cdot n$$

**Solution:** True.  $\Theta(n \lg n^2) = \Theta(n \lg n)$ .

- (n) **T F** Computing the convolution of two vectors, each with  $n$  entries, requires  $\Omega(n^2)$  time.

**Solution:** False. Use the standard FFT algorithm.

**Problem 3. Placing Gas Stations Along a Highway [20 points]**

Give a dynamic programming algorithm that on input  $S$ , where  $S = \{s_0 = 0 \leq s_1 \leq \dots \leq s_n = m\}$  is a finite set of positive integers, determines whether it is possible to place gas stations along an  $m$ -mile highway such that:

1. A gas station can only be placed at a distance  $s_i \in S$  from the start of the highway.
2. There must be a gas station at the beginning of the highway ( $s_0 = 0$ ) and at the end of the highway ( $s_n = m$ ).
3. The distance between every two consecutive gas stations on the highway is between 15 and 25 miles.

For example, suppose the input is  $\{0, 15, 40, 50, 60\}$ . Then your algorithm should output “yes”, because we can place gas stations at distances  $\{0, 15, 40, 60\}$  from the beginning of the highway.

However, if the input is  $\{0, 25, 30, 55, 70\}$ , then your algorithm should output “no”, because there is no subset of the distances that satisfies the conditions listed above.

Remember to analyze the running time of your algorithm.

**Solution:** Check to see if 0 and  $m$  are in  $S$ . If not, output “no”.

Let  $G[0] = \text{“yes”}$  if  $s_0 = 0$ .

For  $i$  from 1 to  $n$ , let  $j = s_i$ :

Let  $G[j]_{j \in S, j > 0} = \text{“yes”}$  if  $G[k] = \text{“yes”}$ , for some  $k \in S$ ,  $k < j$ , and  $15 \leq |s_k - s_j| \leq 25$ .

Return  $G[m]$ .

**Problem 4. Independent Set and Vertex Cover [25 points]**

For a graph  $G = (V, E)$ , we say  $S \subseteq V$  is an independent set in  $G$  if there are no edges between any two vertices in  $S$ .

We say a subset  $T \subseteq V$  is a vertex cover of  $G$  if for every edge  $(u, v)$  in  $E$  at least one of  $u$  or  $v$  is in  $T$ .

- (a) **[10 points]** Show  $S$  is an independent set in  $G$  if and only if  $V - S$  is a vertex cover of  $G$ .

**Solution:**  $\Rightarrow$  Assume  $S$  is an Independent Set, but that  $V - S$  is not a Vertex Cover. Then there exists an edge whose endpoints are both not in  $V - S$ , namely, there is an edge whose endpoints are in  $S$ . That is a contradiction, so  $V - S$  must be a Vertex Cover.

$\Leftarrow$  Now assume that  $V - S$  is a Vertex Cover, but that  $S$  is not an Independent Set. Then there exists an edge with both endpoints in  $S$ . But then that edge would not be touched by  $V - S$ , so  $V - S$  could not be a Vertex Cover. This contradicts our assumption, so  $S$  must be an Independent Set.

Therefore,  $S$  is an Independent Set iff  $V - S$  is a Vertex Cover.

- (b) [5 points] Show that the decision problem  $\text{Independent Set} = \{(G, k) \mid G \text{ contains an independent set of size at least } k\}$  is in NP.

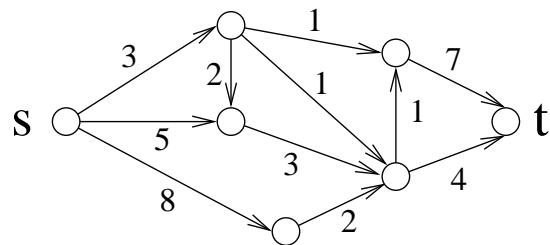
**Solution:** A set of  $k$  vertices forming an independent set is a proof that a given graph is an instance of Independent Set language. This proof can be verified trivially in polynomial time.

(c) [10 points] We showed in class that the decision problem  $\text{Vertex Cover} = \{(G, k) | G \text{ contains a vertex cover of size at most } k\}$  is NP-complete. Use this to show that *Independent Set* is NP-complete.

**Solution:** We need to show that Vertex Cover reduces to Independent Set. By part (a), if a graph has a Vertex Cover of size  $k$ , then it has an Independent Set of size  $n - k$ . So, given an instance  $(G, k)$ , we can trivially make an instance of Independent Set  $(G, n - k)$ .

**Problem 5. Flows [27 points]**

Consider the following graph:



- (a) [10 points] What is the maximum flow in this graph? Give the actual flow as well as its value. Justify your answer.

**Solution:** The maximum flow is 6. From S, we route 3 along both the 3-capacity edge and the 5-capacity edge. The resultant flow will inundate the 1, 1 and 4-capacity edges that form a min-cut for the graph. By the Max-Flow/Min-Cut algorithm, this is a Max-Flow.

- (b) [5 points] True or false: For any flow network  $G$  and any maximum flow on  $G$ , there is always an edge  $e$  such that increasing the capacity of  $e$  increases the maximum flow of the network. Justify your answer.

**Solution:** False. A counterexample is a graph with two unit capacity edges in a chain. Increasing the capacity of a single edge will not increase the max flow, since the other edge is at capacity.

- (c) [5 points] Suppose you have a flow network  $G$  with integer capacities, and an integer maximum flow  $f$ . Suppose that, for some edge  $e$ , we increase the capacity of  $e$  by one. Describe an  $O(|E|)$ -time algorithm to find a maximum flow in the modified graph.

**Solution:** Add the new flow to the residual flow graph in  $O(|E|)$  time. Perform a tree traversal from the source node to detect whether a path now exists to the sink. If so, augment along that path and increase the maximum flow by one.

(d) [7 points] Consider the decision problem:  $\text{Flow} = \{(G, s, t, k) \mid G = (V, E) \text{ is a flow network, } s, t \in V, \text{ and the value of an optimal flow from } s \text{ to } t \text{ in } G \text{ is } k\}$ .

Is  $\text{Flow}$  in NP? Why or why not?

**Solution:** Yes. We can explicitly compute the max-flow using Edmonds-Karp or another polynomial time max-flow algorithm. Since  $P \subseteq NP$ ,  $\text{Flow}$  must be in NP.

**Problem 6. Comparing Sets [20 points]**

Given two sets of integers  $S_1$  and  $S_2$ , each of size  $n$ , your job is to determine if  $S_1$  and  $S_2$  are identical.

Give a  $O(n)$  time randomized algorithm that always outputs “yes” if  $S_1 = S_2$  and outputs “no” with probability at least  $1 - 1/n$  if  $S_1 \neq S_2$ .

You may assume that we have a probabilistic model of computation in which generating a random number takes  $O(1)$  time and a comparison, a multiplication, and an addition of two numbers each takes  $O(1)$  time, regardless of the size of the two numbers involved.

**Hint:** Reduce the problem of comparing sets to the problem of comparing polynomials.

**Solution:** Reduce the problem of comparing sets to that of comparing two degree- $n$  polynomials  $P_1(x) = \prod_{r_1 \in M_1} (x - r_1)$  and  $P_2(x) = \prod_{r_2 \in M_2} (x - r_2)$  with  $n$  integer roots, where each polynomial is specified as a list of these  $n$  integer roots, i.e.  $P_1(x) = \{r_{11}, r_{12}, \dots, r_{1n}\}$ . The following randomized algorithm for comparing  $P_1(x)$  and  $P_2(x)$  which runs in  $O(n)$  time.

```
COMPARE-POLYNOMIALS( $P_1(x), P_2(x)$ )
1 Choose a random integer  $a \in \{1, \dots, n^2\}$ 
2 Compute  $P_1(a) = \prod_{i=1}^n (a - r_{1i})$  and  $P_2(a) = \prod_{i=1}^n (a - r_{2i})$ 
3 if  $P_1(a) = P_2(a)$ 
4   then output  $M_1 = M_2$ 
5   else output  $M_1 \neq M_2$ 
```

Line 2 of COMPARE-POLYNOMIALS( $P_1(x), P_2(x)$ ) runs in  $O(n)$  time, since each multiplication takes  $O(1)$  time. All other lines take  $O(1)$  time under the assumptions stated above. Now we will analyze the probability that COMPARE-POLYNOMIALS( $P_1(x), P_2(x)$ ) outputs the correct answer. If  $P_1(x) = P_2(x)$ , then COMPARE-POLYNOMIALS( $P_1(x), P_2(x)$ ) outputs the correct answer with probability 1.

A degree- $n$  polynomial  $P(x)$  has at most  $n$  distinct roots. Furthermore,  $P(a) \neq 0$  if  $a$  is not a root of  $P(x)$ . Thus,  $P(x)$  has at most  $n$  integer zeroes in the range  $\{1, \dots, n^2\}$ .

Let  $P_3(x) = P_1(x) - P_2(x)$ . Note that  $P_1(a) = P_2(a)$  exactly when  $P_3(a) = 0$ . Since the maximum degree of  $P_3(x)$  is  $n$ ,  $P_3(x)$  has at most  $n$  distinct roots. Thus, if we choose a random integer  $x$  from the range  $\{1, \dots, n^2\}$ , the probability that  $P_3(x)$  evaluates to 0 is at most  $1/n$ .

Therefore, if  $P_1(x) \neq P_2(x)$ , the probability that we choose  $a$  such that  $P_1(a) - P_2(a) = 0$  is at most  $\frac{1}{n}$ . Thus, the probability that COMPARE-POLYNOMIALS( $P_1(x), P_2(x)$ ) outputs the correct answer is at least  $1 - \frac{1}{n}$ .

**Problem 7. Finding the Topological Sort of a Complete Directed Acyclic Graph** [20 points]

Let  $G = (V, A)$  be a directed acyclic graph that has an edge between every pair of vertices and whose vertices are labeled  $1, 2, \dots, n$ , where  $n = |V|$ . To determine the direction of an edge between two vertices in  $V$ , you are only allowed to ask a *query*. A query consists of two specified vertices  $u$  and  $v$  and is answered with:

“from  $u$  to  $v$ ” if  $(u, v)$  is in  $A$ , or

“from  $v$  to  $u$ ” if  $(v, u)$  is in  $A$ .

Give matching upper and lower bounds (as functions of  $n$ ) for the number of queries required to find a topological sort of  $G$ .

**Solution:** This problem reduces to sorting. A query establishes an ordering between two nodes, i.e.  $(u, v)$ ’s existence can be interpreted as  $u > v$  and  $(v, u)$  as  $v < u$ . Since the graph is complete, there exists an ordering between every pair of nodes, in other words, we have a total ordering on the graph.

We can simply run a comparison-based sort to output a topological sort. The “max” element will be a source node with out-going edges to all other nodes. Similarly, the “min” element will be a sink node. Since *query* is effectively a comparison operator, there is a tight  $\Omega(n \log n)$  lower bound on performing a topological sort in this model.

## Practice Quiz 1

- Do not open this quiz booklet until you are directed to do so. Read all the instructions first.
- When the quiz begins, write your name on every page of this quiz booklet.
- The quiz contains five multi-part problems. You have 80 minutes to earn 80 points.
- This quiz booklet contains 13 pages, including this one. An extra sheet of scratch paper is attached. Please detach it before turning in your quiz.
- This quiz is closed book. You may use one handwritten A4 or  $8\frac{1}{2}'' \times 11''$  crib sheet. No calculators or programmable devices are permitted.
- 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, since the pages may be separated for grading.
- Do not waste time and paper rederiving facts that we have studied. It is sufficient to cite known results.
- 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.
- 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.
- Good luck!

Problem	Points	Grade	Initials
1	11		
2	19		
3	10		
4	20		
5	20		
Total	80		

Name: Solutions \_\_\_\_\_  
Circle your recitation letter and the name of your recitation instructor:

Brian

11

12

2

Jen

12

1

**Problem 1. Recurrences [11 points]**

Solve the following recurrences. Give tight, i.e.  $\Theta(\cdot)$ , bounds.

(a)  $T(n) = 81T\left(\frac{n}{9}\right) + n^4 \lg n$  [2 points]

**Solution:**  $T(n) = \Theta(n^4 \lg n)$  by Case 3 of the Master Method. We have:  $n^{\log_b a} = n^2$  and  $f(n) = n^4 \lg n$ . Thus, if  $\epsilon = 1$ , then  $f(n) = \Omega(n^{2+\epsilon})$ . The regularity condition,  $a \cdot f(n/b) < c \cdot f(n)$  holds here for  $c = .99$ .

You received one point if you mentioned Case 3 of the Master Method and showed that it complied with the regularity condition and one point if you gave the correct answer.

(b)  $T(n) = 2T\left(\frac{n}{2}\right) + \frac{n}{\lg n}$  [2 points]

**Solution:** Note that we can not use the Master Method to solve this recurrence. If we expand this recurrence, we obtain:

$$\begin{aligned} T(n) &= \frac{n}{\lg n} + \frac{n}{\lg \frac{n}{2}} + \frac{n}{\lg \frac{n}{4}} \cdots \frac{n}{\lg \frac{n}{n}} = n \cdot \sum_{i=1}^{\lg n} \frac{1}{\lg \frac{n}{2^i}} = \\ &n \cdot \sum_{i=1}^{\lg n} \frac{1}{\lg n - i} = n \cdot \sum_{i=1}^{\lg n} \frac{1}{i} = \Theta(n \lg \lg n). \end{aligned}$$

(c)  $T(n) = 4T(n/3) + n^{\log_3 4}$  [2 points]

**Solution:**  $T(n) = \Theta(n^{\log_3 4} \lg n)$  by Case 2 of the Master Method.

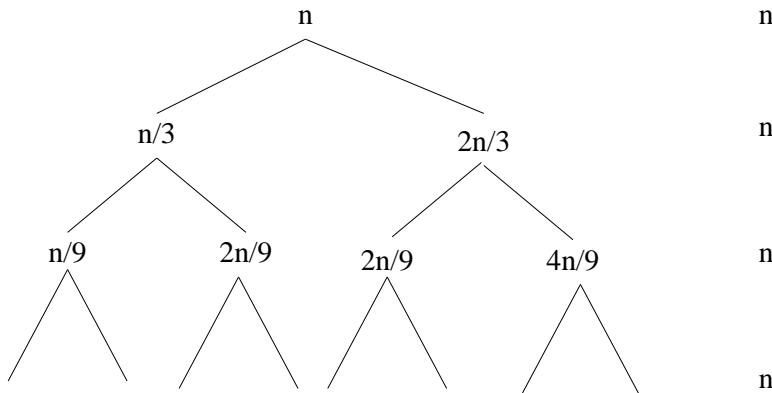
You received one point if you mentioned the correct case of the Master Method and one point for the correct answer. A common error was to not cite which case of the Master Method you used.

- (d) Write down and solve the recurrence for the running time of the DETERMINISTIC SELECT algorithm using groups of 3 (rather than 5) elements. The code is provided on the last page of the exam. [5 points]

**Solution:**  $T(n) = T(n/3) + T(2n/3) + cn$ . We solve this recurrence using a recursion tree, see Figure 1. The height of the tree is at least  $\log_3 n$  and is at most  $\log_{3/2} n$  and the sum of the costs in each level is  $n$ . Hence  $T(n) = \Theta(n \lg n)$ .

A correct recurrence received 3 points. One point was awarded for solving whichever recurrence you gave correctly and one point was awarded for justification. An incorrect recurrence received 1 point if it was almost correct.

A common error was omitting the  $T(2n/3)$  term in the recurrence.



**Figure 1:** Recursion tree

**Problem 2. True or False, and Justify [19 points]**

Circle **T** or **F** for each of the following statements, and briefly explain why. The better your argument, the higher your grade, but be brief. Your justification is worth more points than your true-or-false designation.

- (a) **T F** Given a set of  $n$  integers, the number of comparisons necessary to find the maximum element is  $n - 1$  and the number of comparisons necessary to find the minimum element is  $n - 1$ . Therefore the number of comparisons necessary to simultaneously find the smallest and largest elements is  $2n - 2$ . **[3 points]**

**Solution:** False. In recitation, we gave an algorithm that uses  $3n/2$  comparisons to simultaneously find the maximum and minimum elements.

A common error was to say, "You can just keep track of min and max at the same time", which does not say how many comparisons that would take or how it would work.

- (b) **T F** There are  $n$  people born between the year 0 A.D. and the year 2003 A.D. It is possible to sort all of them by birthdate in  $o(n \lg n)$  time.

**Solution:** True. We can represent a birthdate using  $c$  digits. This database can then be sorted in  $O(c \cdot n) = O(n)$  time using RADIX SORT.

Common errors included saying, "Use Mergesort" or "Use Radix Sort" (with no explanation).

- (c) **T F** There is some input for which RANDOMIZED QUICKSORT always runs in  $\Theta(n^2)$  time. [3 points]

**Solution:** False. The expected running time of RANDOMIZED QUICKSORT is  $\Theta(n \lg n)$ . This applies to any input.

A common error was to say, "RANDOMIZED QUICKSORT will take  $O(n^2)$  only if it's very unlucky", but you needed to say what it's expected runtime is.

- (d) **T F** The following array  $A$  is a Min Heap:

2    5    9    8    10    13    12    22    50

[3 points]

**Solution:** True.

- (e) **T F** If  $f(n) = \Omega(g(n))$  and  $g(n) = O(f(n))$  then  $f(n) = \Theta(g(n))$ . [3 points]

**Solution:** False. For example  $f(n) = n^2$  and  $g(n) = n$ .  
A common error was to mix up  $f$  and  $g$  or  $O$  and  $\Omega$ .

- (f) **T F** Let  $k, i, j$  be integers, where  $k > 3$  and  $1 \leq i, j \leq k$ . Let  $h_{ij}^k$  be the hash function mapping a  $k$ -bit integer  $b_1 b_2 \dots b_k$  to the 2-bit value  $b_i b_j$ . For example,  $h_{31}^8(00101011) = 10$ . The set  $\{h_{ij}^k : 1 \leq i, j \leq k\}$  is a universal family of hash functions from  $k$ -bit integers into  $\{00, 01, 10, 11\}$ . [4 points]

We have not covered Hashing yet this semester. Expect a problem of equivalent length and difficulty.

**Solution:** False. Take  $x = 0, y = 1$ . Then all of  $x$ 's binary digits are the same as  $y$ 's, except for the least significant bit. Thus, if we choose one of the  $k$  hash functions at random,  $\Pr[h(x) = h(y)] = (k^2 - k)/k^2 = (k - 1)/k > 1/4$ . If this were a universal class of hash functions, then this probability should be at most  $1/4$ .

A common error was to show a single counter-example without explanation.

**Problem 3. Short Answer [10 points]**

Give brief, but complete, answers to the following questions.

- (a) Explain the differences between average-case running time analysis and expected running time analysis. For each type of running time analysis, name an algorithm we studied to which that analysis was applied.

**[5 points]**

**Solution:** The average-case running time does not provide a guarantee for the worst-case, i.e. it only applies to a specific input distribution, while the expected running time provides a guarantee (in expectation) for every input.

In the average-case running time, the probability is taken over the random choices over an input distribution. In the expected running time, the probability is taken over the random choices made by the algorithm.

The average-case running time of BUCKET SORT is  $\Theta(n)$  when the input is chosen at random from the uniform distribution. The expected running time of QUICK SORT is  $\Theta(n \lg n)$ .

- (b) Suppose we have a hash table with  $2n$  slots with collisions resolved by chaining, and suppose that  $n/8$  keys are inserted into the table. Assume each key is equally likely to be hashed into each slot (**simple uniform hashing**). What is the expected number of keys for each slot? Justify your answer. [5 points]

*We have not covered Hashing yet this semester. Expect a problem of equivalent length and difficulty.*

**Solution:** Define  $X_j$  (for  $j = 1, \dots, n$ ) to be the indicator which is 1 if element  $j$  hashes to slot  $i$  and 0 otherwise. Then  $E[X_j] = Pr[X_j = 1] = 1/(2n)$ . Then the expected number of elements in slot  $i$  is  $E[\sum_{j=1}^{n/8} X_j] = \sum_{j=1}^{n/8} E[X_j] = n/(8(2n)) = 1/16$  by linearity of expectation.

**Problem 4. Checking Properties of Sets [20 points]**

In this problem, more efficient algorithms will be given more credit. Let  $S$  be a finite set of  $n$  positive integers,  $S \subset \mathbb{Z}^+$ . You may assume that all basic arithmetic operations, i.e. addition, multiplication, and comparisons, can be done in unit time. In this problem, partial credit will be given for correct but inefficient algorithms.

- (a) Design an  $O(n \lg n)$  algorithm to verify that:

$$\forall T \subseteq S, \quad \sum_{t \in T} t \geq |T|^3.$$

In other words, if there is some subset  $T \subseteq S$  such that the sum of the elements in  $T$  is less than  $|T|^3$ , the your algorithm should output “no”. Otherwise, it should output “yes”. Argue (informally) that your algorithm is correct and analyze its running time. **[10 points]**

**Solution:** Sort the set  $S$  in time  $O(n \lg n)$  using  $O(n \lg n)$  sorting algorithm such as MERGE SORT or HEAP SORT. For each  $k$  between 1 and  $n$ , verify that  $(\sum_{i=0}^k s_i) \geq k^3$ . By maintaining a running sum, checking this sum for each value of  $k$  requires only one addition and one comparison operation. Thus, the total running time is  $O(n \lg n) + O(n) = O(n \lg n)$ .

Correctness: The key observation is that every set of  $k$  elements is at least  $k^3$  iff the sum of the smallest  $k$  elements is at least  $k^3$ . Thus, if we sort the elements, the sum of the first  $k$  elements, namely the  $k$  smallest elements, will be at least  $k^3$  iff every set of size  $k$  has sum at least  $k^3$ .

The following procedure CHECKALLSUMS takes as input an array  $A$  containing  $S$  and an integer  $n$  indicating the size of  $S$ .

```
CHECKALLSETS( $A, n$ )
1 Sort  $A$  using MERGESORT
2 sum  $\leftarrow 0$ 
3 from  $i \leftarrow 1$  to  $n$ 
4     sum  $\leftarrow$  sum +  $A[i]$ 
5     if sum <  $k^3$ 
6         return “no”
7 return “yes”
```

A common error was to use COUNTING SORT, which would not necessarily be efficient for an arbitrary set of  $n$  integers.

Many people got this problem backwards, i.e. they tried to find some set such that its sum was at least its size cubed. This was not heavily penalized.

- (b) In addition to  $S$  and  $n$ , you are given an integer  $k$ ,  $1 \leq k \leq n$ . Design a more efficient (than in part (a)) algorithm to verify that:

$$\forall T \subseteq S, |T| = k, \sum_{t \in T} t \geq k^3.$$

In other words, if there is some subset  $T \subseteq S$  such that  $T$  contains exactly  $k$  elements and the sum of the elements in  $T$  is less than  $k^3$ , then your algorithm should output “no”. Otherwise, it should output “yes”. Argue (informally) that your algorithm is correct and analyze its running time. [10 points]

**Solution:** Find the  $k$ th smallest element using the linear time DETERMINISTICSELECT algorithm. Then find the  $k$  smallest elements using the PARTITION procedure. Check that the sum of these  $k$  smallest elements is greater than  $k^3$ . The total runtime is  $O(n) + O(n) + O(k) = O(n)$ .

CHECKSETOSIZEK( $A, n, k$ )

- 1 Run SELECT( $A, n, k$ ) to find  $k^{th}$  smallest element
- 2 Run PARTITION( $A, n, k$ ) to put  $k$  smallest elements in  $A[1 \dots k]$
- 3 sum  $\leftarrow 0$
- 4 from  $i \leftarrow 1$  to  $k$
- 5     sum  $\leftarrow$  sum +  $A[i]$
- 6 if sum  $< k^3$
- 7     return “no”
- 8 return “yes”

Correctness: Again, as in part (a), the key observation is that every set of  $k$  elements has sum at least  $k^3$  iff the sum of the  $k$  smallest elements is at least  $k^3$ .

**Problem 5. Finding the Missing Number [20 points]**

Suppose you are given an unsorted array  $A$  of all integers in the range 0 to  $n$  except for one integer, denoted the *missing number*. Assume  $n = 2^k - 1$ .

- (a) Design a  $O(n)$  Divide and Conquer algorithm to find the missing number. Partial credit will be given for non Divide and Conquer algorithms. Argue (informally) that your algorithm is correct and analyze its running time. **[12 points]**

**Solution:** We can use SELECT to find the median element and check to see if it is in the array. If it is not, then it is the missing number. Otherwise, we PARTITION the array around the median element  $x$  into elements  $\leq x$  and  $> x$ . If the first one has size less than  $x + 1$ , then we recurse on this subarray. Otherwise we recurse on the other subarray.

The procedure MISSINGINTEGER( $A, n, [i, j]$ ) takes as input an array  $A$  and a range  $[i, j]$  in which the missing number lies.

```
MISSINGINTEGER( $A, [i, j]$ )
1 Determine median element  $x$  in range  $i \dots j$ 
2 Check to see if  $x$  is in  $A$ 
3 PARTITION  $A$  into  $B$ , elements  $< x$ , and  $C$ , elements  $\geq x$ 
4 If SIZE( $B$ )  $< x + 1$ 
   5   MISSINGINTEGER( $B, [i, x]$ )
6 Else MISSINGINTEGER( $C, [x + 1, j]$ )
```

The running time is  $O(n)$  because the recurrence for this algorithm is  $T(n) = T(n/2) + n$ , which is  $O(n)$  by the Master Method.

Common errors included using a randomized, instead of deterministic, partitioning scheme and using COUNTING SORT and then stepping through the array to find adjacent pairs that differ by two, which is not a Divide and Conquer approach.

- (b) Suppose the integers in  $A$  are stored as  $k$ -bit binary numbers, i.e. each bit is 0 or 1.

For example, if  $k = 2$  and the array  $A = [01, 00, 11]$ , then the missing number is 10.

Now the only operation to examine the integers is  $\text{BIT-LOOKUP}(i, j)$ , which returns the  $j$ th bit of number  $A[i]$  and costs unit time. Design an  $O(n)$  algorithm to find the missing number. Argue (informally) that your algorithm is correct and analyze its running time. [8 points]

**Solution:** We shall examine bit by bit starting from the least significant bit to the most significant bit. Make a count of the number of 1's and 0's in each bit position, we can find whether the missing number has a 0 or 1 at the bit position being examined. Having done this, we have reduced the problem space by half as we have to search only among the numbers with that bit in that position. We continue in this manner till we have exhausted all the bit-positions (ie.,  $k$  to 1).

The algorithm is as follows. For convenience, we have 3 sets  $S, S_0, S_1$  which are maintained as link-lists.  $S$  contains the indices of the elements in  $A$  which we are going to examine while  $S_0$  ( $S_1$ ) contains the indices of the elements in  $A$  which have 0 (1) in the bit position being examined.

```
MISSING-INTEGER( $A, n$ )
1   $k \leftarrow \lg n$        $\triangleright n = 2^k - 1$ 
2   $S \leftarrow \{1, 2, \dots, n\}$ 
3   $S_0 \leftarrow S_1 \leftarrow \{\}$      $\triangleright$  Initialized to Empty List
4   $count0 \leftarrow count1 \leftarrow 0$ 
5  for  $posn \leftarrow k$  downto 1 do
6    for each  $i \in S$  do
7       $bit \leftarrow \text{BIT-LOOKUP}(i, posn)$ 
8      if  $bit = 0$ 
9        then  $count0 \leftarrow count0 + 1$ 
10       Add  $i$  to  $S_0$ 
11      else  $count1 \leftarrow count1 + 1$ 
12       Add  $i$  to  $S_1$ 
13      if  $count0 > count1$ 
14        then  $missing[posn] \leftarrow 1$ 
15         $S \leftarrow S_1$ 
16        else  $missing[posn] \leftarrow 0$ 
17         $S \leftarrow S_0$ 
18       $S_0 \leftarrow S_1 \leftarrow \{\}$ 
19       $count0 \leftarrow count1 \leftarrow 0$ 
20  return  $missing$ 
```

It can be noted that the following invariant holds at the end of the loop in Step 5-19.

- The bits of the missing integer in the bit position ( $posn$  to  $k$ ) is given by  $missing[posn] \dots missing[k]$ .
- $S = \{i : j^{\text{th}} \text{ bit of } A[i] = missing[j] \text{ for } posn \leq j \leq k\}$

This loop invariant ensures the correctness of the algorithm.

Each loop iteration (Step 5-19) makes  $|S|$   $\text{BIT-LOOKUP}$  operations. And the size of  $S$  is halved in each iteration. Hence total number of  $\text{BIT-LOOKUP}$  operations is  $\sum_{i=0}^{k-1} \frac{n}{2^i}$ , which is  $O(n)$ .

The grading for this problem was (-6) points for an algorithm that runs in  $\Theta(nk)$ , like RADIX SORT (or any of a number of built-from-scratch radix-sort-like approaches). Another point was deducted

DETERMINISTICSELECT( $A, n, i$ )

- 1 Divide the elements of the input array  $A$  into groups of 3 elements.
- 2 Find median of each group of 3 elements and put them in array  $B$ .
- 3 Call DETERMINISTICSELECT( $B, n/3, n/6$ ) to find median of the medians,  $x$ .
- 4 Partition the input array around  $x$  into  $A_1$  containing  $k$  elements  $\leq x$   
and  $A_2$  containing  $n - k - 1$  elements  $\geq x$ .
- 5 If  $i = k + 1$ , then return  $x$ .
- 6 Else if  $i \leq k$ , DETERMINISTICSELECT( $A_1, k, i$ ).
- 7 Else if  $i > k$ , DETERMINISTICSELECT( $A_2, n - k - 1, i - (k + 1)$ ).

## Practice Quiz 2

- Do not open this quiz booklet until you are directed to do so. Read all the instructions first.
- When the quiz begins, write your name on every page of this quiz booklet.
- The quiz contains four multi-part problems. You have 120 minutes to earn 108 points.
- This quiz booklet contains **16** pages, including this one. Extra sheets of scratch paper are attached. Please detach them before turning in your quiz.
- This quiz is closed book. You may use one handwritten A4 or  $8\frac{1}{2}'' \times 11''$  crib sheet. No calculators or programmable devices are permitted.
- 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, since the pages may be separated for grading.
- Do not waste time and paper rederiving facts that we have studied. It is sufficient to cite known results.
- 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.
- 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.
- Good luck!

Problem	Points	Grade	Initials
1	40		
2	25		
3	25		
4	18		
Total	108		

Name: Solutions \_\_\_\_\_  
Circle your recitation letter and the name of your recitation instructor:

David      A      B      Steve      C      D      Hanson      E      F

**Problem 1. True or False, and Justify** [40 points]

Circle **T** or **F** for each of the following statements, and briefly explain why. The better your argument, the higher your grade, but be brief. Your justification is worth more points than your true-or-false designation. Each part is worth **4 points**.

- (a) **T F** A preorder traversal of a binary search tree will output the values in sorted order.

**Solution:** False. Consider a BST with 2 at the root and 1 as left child and 3 as right child. The preorder is 2, 1, 3. An *inorder* traversal of a BST outputs the values in sorted order.

- (b) **T F** The cost of searching in an AVL tree is  $O(\lg n)$ .

**Solution:** True. An AVL tree is a balanced binary search tree. Therefore, searching in an AVL tree costs  $O(\lg n)$ .

- (c) **T F** The expected amount of space used by a skip list on  $n$  elements is  $O(n)$ .

**Solution:** True. Each element appears in  $2 = O(1)$  rows in expectation, and each occurrence requires  $O(1)$  space for the element and pointers. By linearity of expectation, the space taken by all elements is  $O(n)$  in expectation. Alternatively, we expect each level to have half as many elements as the one below it, for space  $O(n + n/2 + n/4 + \dots) = O(2n) = O(n)$  by geometric summation.

It is not correct to answer “false” because the skip list has  $O(\log n)$  levels (w/ high prob.), and each level has  $O(n)$  elements, for a total of  $O(n \log n)$  space. While this bound is true, it is not the tightest possible (i.e., it is too loose of an approximation of the actual space used).

- (d) **T F** The height of a randomly built binary search tree is  $O(\log n)$  with high probability; therefore in randomized quicksort, every element is involved in  $O(\log n)$  comparisons with high probability.

**Solution:** False. In randomized quicksort, the first pivot is compared to *every* other element, and there are  $n - 1 = \Omega(n)$  of them. Therefore every element has at least a  $1/n$  chance of being involved in  $\Omega(n)$  comparisons.

Alternately, the root of the randomly-built binary search tree is compared to every other element. (More generally, an element is compared with every element in its subtree, as well as all its ancestors.) These comparisons are the same ones done by randomized quicksort.

Many solutions regurgitated the fact that there is a connection between the comparisons performed by quicksort and those performed in building a random BST. This is true, but irrelevant. The *height* of a node in the BST has nothing to do with the number of elements it is compared to in the corresponding quicksort.

- (e) **T F** Any mixed sequence of  $m$  increments and  $n$  decrements to a  $k$ -bit binary counter (that is initialized to zero and is always non-negative) takes  $O(m + n)$  time in the worst case.

*Spring 2004: See CLRS pg. 408-409*

**Solution:** False. If we allow decrement operations, then a series of  $n + m$  increment and decrement operations could take  $O(kn + km)$  in the worst case.

- (f) **T F** Consider a directed acyclic graph  $G = (V, E)$  and a specified source vertex  $s \in V$ . Every edge in  $E$  has either a positive or a negative edge weight, but  $G$  has no negative cycles. Dijkstra's Algorithm can be used to find the shortest paths from  $s$  to all other vertices.

**Solution:** False. Counterexample  $G = \{w(s, b) = -3, w(s, c) = 2, w(b, c) = 4\}$ .

- (g) **T F** Given a bipartite graph  $G$  and a matching  $M$ , we can determine if  $M$  is maximum in  $O(V + E)$  time.

**Solution:** True. Modified BFS is used to determine if there is an augmenting path in time  $O(V + E)$ . If there is no augmenting path, then the matching is maximum.

- (h) **T F** Given a random skip list on  $n$  elements in which each element has height  $i$  with probability  $(1/3)^{i-1}(2/3)$ , the expected number of elements with height  $\lg_3 n$  is  $O(1)$ .

**Solution:** True. Let  $X_i$  be the indicator random variable that is a 1 if element  $i$  is in the set of elements with height at least  $\lg_3 n$  and 0 otherwise. An element has height at least  $\lg_3 n$  with probability  $(2/3)(1/3)^{\lg_3 n - 1} = 2/n$ . Thus,  $E[X_i] = 2/n$  and  $E[\sum_{i=1}^n X_i] = \sum_{i=1}^n E[X_i] = 2 = O(1)$ .

- (i) **T F** Suppose you are given an undirected connected graph with integer edge weights in which each vertex is degree 2. An MST can be computed for this graph in  $O(V)$  time.

**Solution:** True. The graph must be a cycle with  $O(V)$  edges. We can remove the edge with the highest weight in  $O(V)$  time.

- (j) **T F** Consider an undirected, weighted graph  $G$  in which every edge has a weight between 0 and 1. Suppose you replace the weight of every edge with  $1 - w(u, v)$  and compute the minimum spanning tree of the resulting graph using Kruskal's algorithm. The resulting tree is a maximum cost spanning tree for the original graph.

**Solution:** True. Let  $G'$  be the graph in which every edge has weight  $1 - w(u, v)$ . Let MaxST denote the maximum spanning tree in  $G$  and let MST denote the minimum spanning tree in  $G'$ . Suppose MST does correspond to MaxST in  $G$ . Then there is a spanning tree in  $G$  with higher weight, implying that if we replace each edge with weight  $1 - w(u, v)$ , we will find a spanning tree in  $G'$  that has less weight than MST, which is a contradiction.

**Problem 2. Short Answer Problems [25 points]**

Give *brief*, but complete, answers to the following questions. Each problem part is worth **5 points**.

**Amortized Analysis**

You are to maintain a collection of lists and support the following operations.

- (i) *insert(item, list)*: insert item into list (cost = 1).
  - (ii) *sum(list)*: sum the items in list, and replace the list with a list containing one item that is the sum (cost = length of list).
- (a) Use the Accounting Method to show that the amortized cost of an *insert* operation is  $O(1)$  and the amortized cost of a *sum* operation is  $O(1)$ .

**Solution:** We will maintain the invariant that every item has one credit. Insert gets 2 credits, which covers one for the actual cost and one to satisfy the invariant. Sum gets one credit, because the actual cost of summing is covered by the credits in the list, but then the result of the sum will need one credit to maintain the invariant.

A common error was not putting a credit on the newly created sum.

- (b) Use the Potential Method to show that the amortized cost of an *insert* operation is  $O(1)$  and the amortized cost of a *sum* operation is  $O(1)$ .

**Solution:** We define  $\Phi(list)$  to be the number of elements in the list. Then the amortized cost of an insert operation is  $\hat{c}_i = c_i + \Phi_i - \Phi_{i-1}$ . The actual cost  $c_i$  is 1. The change in potential is 1. So the amortized cost is 2. For a sum operation, the actual cost,  $c_i$  is  $k$  and the change in potential is  $\Phi_i - \Phi_{i-1} = 1 - (k) = 1 - k$ , so the amortized cost of a sum is 1.

**Balanced Search Trees**

- (c) Show that the number of node splits performed when inserting a single node into a 2-3 tree can be as large as  $\Omega(\lg n)$ , where  $n$  is the number of keys in the tree. (E.g. give a general, worst-case example.)

**Solution:** Suppose that every node in an  $n$ -node 2-3 tree is full – i.e., all internal nodes have three children. Then the height of the tree is  $\log_3 n = \Omega(\log n)$ . [Common error: I don't care that its height is  $O(\log n)!$ ] When we insert into a leaf node, it's full—so we have to have to do something with the fourth element ... that is, bump the median element up to its parent. But its parent is full, too—so we have to bump up an element from there. And so on, all the way up to the root. The numbers of splits is  $\Omega(\text{height}) = \Omega(\log n)$ .

**Faster MST Algorithms**

- (d) Given an undirected and connected graph in which all edges have the same weight, give an algorithm to compute an MST in  $O(E)$  time.

**Solution:** Run DFS and keep only the tree edges.

Some common errors were: (1) not running in linear time (union-find is NOT constant!) and (2) not producing a tree (a graph where every node has degree  $\geq 1$  is not necessarily connected!).

**FFT****(e) Snowball Throwing**

Several 6.046 students hold a team snowball throwing contest. Each student throws a snowball with a distance in the range from 0 to  $10n$ . Let  $M$  be the set of distances thrown by males and  $F$  be the set of distances thrown by females. You may assume that the distance thrown by each student is unique and is an integer. Define a team score to be the combination of one male and one female throw.

Give an  $O(n \log n)$  algorithm to determine every possible team score, as well as how many teams could achieve a particular score. This multi-set of values is called a *cartesian sum* and is defined as:

$$C = \{m + f : m \in M \text{ and } f \in F\}$$

**Solution:**

Represent  $M$  and  $F$  as polynomials of degree  $10n$  as follows:

$$M(x) = x^{a_1} + x^{a_2} + \dots + x^{a_n}, F(x) = x^{b_1} + x^{b_2} + \dots + x^{b_n}$$

Multiply  $M$  and  $F$  in time  $\Theta(n \log n)$  to obtain a coefficient representation  $c_0, c_1, \dots, c_{2n}$ . In other words  $C(x) = c_0 + c_1x + c_2x^2 + \dots + c_{2n}x^{2n}$ . Each pair  $a_i, b_j$  will account for one term  $x^{a_i}x^{b_j} = x^{a_i+b_j} = x^k$ . Therefore,  $c_k$  will be the number of such pairs  $a_i + b_j = k$ .

**Problem 3. Dynamic Programming** [25 points]

Santa Claus is packing his sleigh with gifts. His sleigh can hold no more than  $c$  pounds. He has  $n$  different gifts, and he wants to choose a subset of them to pack in his sleigh. Gift  $i$  has utility  $u_i$  (the amount of happiness gift  $i$  induces in some child) and weight  $w_i$ . We define the weight and utility of a *set of gifts* as follows:

- The weight of a set of gifts is the *sum* of their weights.
- The utility of a set of gifts is the *product* of their utilities.

For example, if Santa chooses two gifts such that  $w_1 = 3, u_1 = 4$  and  $w_2 = 2, u_2 = 2$ , then the total weight of this set of gifts is 5 pounds and the total utility of this set of gifts is 8. All numbers mentioned are positive integers and for each gift  $i$ ,  $w_i \leq c$ . Your job is to devise an algorithm that lets Santa maximize the utility of the set of gifts he packs in his sleigh without exceeding its capacity  $c$ .

- (a) [5 points] A greedy algorithm for this problem takes the gifts in order of increasing weight until the sleigh can hold no more gifts. Give a small example to demonstrate that the greedy algorithm does not generate an optimal choice of gifts.

**Solution:** Suppose that  $c = 2$  and the gifts have weights 1 and 2 and  $u_i = w_i$  for each gift. If Santa chooses gift 1, then he can not fit gift 2, so the total utility he obtains is 1 rather than 2.

- (b) [10 points] Give a recurrence that can be used in a dynamic program to compute the maximum utility of a set of gifts that Santa can pack in his sleigh. Remember to evaluate the base cases for your recurrence.

**Solution:** Let  $H(k, x)$  be the maximal achievable utility if the gifts are drawn from 1 through  $k$  (where  $k \leq n$ ) and weigh at most  $x$  pounds.

For  $1 \leq k \leq n$ :

If  $x - w_k \geq 0$ ,  $H(k, x) = \max\{H(k - 1, x - w_k) \cdot u_k, H(k - 1, x), u_k\}$

Otherwise (if  $x - w_k < 0$ ),  $H(k, x) = H(k - 1, x)$

**Base cases:**  $H(0, x) = 0$  for all integers  $x$ ,  $1 \leq x \leq c$ .

Some common errors were (1) forgetting to include  $u_i$  in the recurrence for the case in which  $H(k - 1, x - w_k)$  is 0, (2) neglecting one dimension, e.g. weight, (3) using + instead of \* for utility, (4) writing the recurrence as  $H(i, c)$  instead of (general)  $H(i, x)$ .

- (c) [7 points] Write pseudo-code for a dynamic program that computes the maximum utility of a set of gifts that Santa can pack in his sleigh. What is the running time of your program?

**Solution:** We give the following pseudo-code for a dynamic program that takes as input a set of gifts,  $\mathcal{G} = \{g_1, g_2, \dots, g_n\}$ , a maximum weight,  $c$ , and outputs the set of gifts with the maximum utility.

MAXIMUM-UTILITY( $\mathcal{G}, c$ )

- 1 For  $1 \leq x \leq c$
- 2      $H(0, x) = 0$
- 3 For  $1 \leq k \leq n$
- 4     For  $1 \leq x \leq c$
- 5         If  $x - w_k < 0$
- 6              $H(k, x) = H(k - 1, x)$
- 7         If  $x - w_k \geq 0$
- 8              $H(k, x) = \max\{H(k - 1, x - w_k) \cdot u_k, H(k - 1, x), u_k\}$
- 9 Return  $H(n, c)$

The running time of this algorithm is  $O(c \cdot n)$ .

- (d) [3 points] Modify your pseudo-code in part (c) to output the actual set of gifts with maximum utility that Santa packs in his sleigh.

**Solution:** The following pseudo-code takes as input the table  $H$  computed in part (c) and variables  $k, n$  and outputs a set of gifts that yield utility  $H(k, n)$ .

OUTPUT-GIFTS( $H, k, x$ )

- 1 If  $k = 0$
- 2     Output  $\emptyset$
- 3 Else If  $H(k, x) = H(k - 1, x - w_k) \cdot u_k$
- 4     Output  $g_k$
- 5     OUTPUT-GIFTS( $H, k - 1, x - w_k$ )
- 6 Else if  $H(k, x) = H(k - 1, x)$
- 7     Output OUTPUT-GIFTS( $H, k - 1, x$ )
- 8 Else if  $H(k, x) = u_k$
- 9     Output  $g_k$

**Problem 4. Reliable distribution**

A communication network consists of a set  $V$  of nodes and a set  $E \subset V \times V$  of directed edges (communication links). Each edge  $e \in E$  has a **weight**  $w(e) \geq 0$  representing the cost of using  $e$ . A **distribution** from a given source  $s \in V$  is a set of directed  $|V| - 1$  paths from  $s$  to each of the other  $|V| - 1$  vertices in  $V - \{s\}$ . The **cost** of a distribution is the sum of the weights of its constituent paths. (Thus, some edges may be counted more than once in the cost of the distribution.)

- (a) Give an efficient algorithm to determine the cheapest distribution from a given source  $s \in V$ . You may assume all nodes in  $V$  are reachable from  $s$ .

**Solution:** This problem can be modelled as a single-source shortest paths problem. A distribution is a tree of paths from  $s$  to every other vertex in the graph. Since the cost of a distribution is the sum of the lengths of its paths, a minimum cost distribution is a set of shortest paths. Since the edge weights are non-negative, we can use Dijkstra's algorithm. The running time of Dijkstra's algorithm can be improved from  $O(|V|^2)$  to  $O((|V| + |E|) \lg |V|)$  using a binary heap. Since we assumed that the graph is connected, the running time is  $O(|E| \lg |V|)$ . If we use a Fibonacci heap, the running time is  $O(|V| \lg |V| + |E|)$ .

In addition, we need to return the minimum cost distribution. Dijkstra's algorithm as given in CLR computes backpointers  $\pi[v]$  to represent the shortest paths. We can use these to represent the distribution; when asked for the shortest path from  $s$  to  $v$ , we trace the backpointers from  $v$  to  $s$  and return the traversed edges (in reverse order).

- (b) One of the edges in the communication network may fail, but we don't know which one. Give an efficient algorithm to determine the maximum amount by which the cost of the cheapest distribution from  $s$  might increase if an adversary removes an edge from  $E$ . (The cost is infinite if the adversary can make a vertex unreachable from  $s$ .)

**Solution:** If an edge is removed from the graph, it is possible that the cost of the minimum cost distribution on the resulting graph may be more than the cost of original chepeast distribution. That is, let  $D$  be the minimum cost distribution found in part (a), and let  $C(D)$  be its cost. If we remove edge  $e$  from the graph, let  $C(e)$  denote the cost of the minimum cost distribution for the new graph (with edge set  $E - \{e\}$ ). We need to compute

$$\max_{e \in E} C(e) - C(D).$$

First recall that cost of a distribution is  $\sum_{v \in V} d[v]$ , where the  $d[v]$  are the distance values returned by Dijkstra.

The straightforward brute-force approach to solve this problem is compute  $C(e)$  by deleting  $e$  from the graph and rerunning the algorithm from (a). However, note that if the deleted edge  $e \notin D$  then  $C(e) = C(D)$ , since the removal of  $e$  does not affect the distribution  $D$ . So we only need to find

$$\max_{e \in D} C(e) - C(D).$$

Since the edges in  $D$  are a set of shortest paths, they form a tree, and a tree has  $|V| - 1$  edges.

To compute  $C(e)$  for an edge  $e \in D$ , we can delete  $e$  and then rerun Dijkstra's algorithm on the resulting graph. It is important to note that removing  $e$  may make some vertices unreachable from  $s$ . To check this, we remove  $e$ , rerun Dijkstra, and then check if any of the  $d'[v]$  distances are  $\infty$ . If so, then  $C(e) = \infty$  and we should halt the algorithm and return  $\infty$  as the maximum possible increase. If not, then  $C(e) = \sum_v d'[v]$ .

The running time of this solution is the cost of  $|V| - 1$  calls to Dijkstra's algorithm. Using Fibonacci heaps, this is  $O(|V|^2 \lg |V| + |V||E|)$ .

## Quiz 1

- Do not open this quiz booklet until you are directed to do so. Read all the instructions first.
- The quiz contains **6** multi-part problems. You have 80 minutes to earn 80 points.
- This quiz booklet contains **7** numbered pages of problems.
- This quiz is closed book. You may use one double sided Letter ( $8\frac{1}{2}'' \times 11''$ ) or A4 crib sheet. No calculators or programmable devices are permitted. Cell phones must be put away.
- Write your solutions in the space provided. Extra scratch paper may be provided if you need more room, although your answer should fit in the given space.
- Do not waste time re-deriving facts covered in lectures, homework or recitations. It is sufficient to cite known results.
- Do not spend too much time on any one problem. Generally, a problem's point value is an indication of how much time to spend on it.
- 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.
- Good luck!

Problem	Points	Grade	Initials
1	1		
2	12		
3	20		
4	12		
5	15		
6	20		
Total	80		

**Problem 1.** [1 points] Write your name on every page! Don't forget the cover.

**Problem 2. True or False, and Justify** [12 points] (4 parts)

Circle **T** or **F** for each of the following statements, and briefly explain why. Your justification is worth more points than your true-or-false designation.

- (a) **T F** (2011 Fall Quiz 1) A dynamic program's sub-problem graph (where there is an edge from sub-problem A to sub-problem B iff solving A requires a solution to B) must be a tree.

**Justification:**

**Solution:** False. The subproblem graph is *directed*. This graph cannot have any cycles: if there was a cycle in the graph, then no subproblem in the cycle can ever be solved first because it depends on other subproblems in the cycle. However, a directed acyclic graph is *not* necessarily a tree. As a simple counterexample, consider the graph with vertices  $A$ ,  $B$  and  $C$  and with edges from  $(A, B)$ ,  $(A, C)$  and  $(B, C)$ .

- (b) **T F** (2011 Fall Quiz 1) You are given a weighted graph  $G = (V, E, W)$  such that (i) the weights  $W$  are non-negative and (ii)  $|E| = |V|^{1.5}$ . Your goal is to compute all-pairs shortest paths in  $G$ .

Among all algorithms seen in the class, Floyd-Warshall algorithm achieves the lowest running time for this problem.

**Justification:**

**Solution:** False. The Floyd-Warshall algorithm takes  $O(|V|^3)$  time to solve this problem. However, since the edges of the graph have non-negative weights, we can solve all-pairs shortest paths by running Dijkstra's algorithm once from each vertex. The total running time of this algorithm is  $O(|V||E| + |V|^2 \log |V|)$ , which is  $O(|V|^{2.5})$  for this particular problem. Note that running Dijkstra's once is insufficient, since it only computes single-source shortest paths.

- (c) **T F** (2011 Spring Practice Quiz 1) Suppose we have computed a minimum spanning tree (MST) and its total weight for some graph  $G = (V, E)$ . If we make a new graph  $G'$  by adding 1 to the weight of every edge in  $G$ , we will need to spend  $\Omega(E)$  time to compute an MST and its total weight for the new graph  $G'$ .

**Justification:**

**Solution:** False. If  $T$  is an MST for  $G$  with weight  $w$ , then it is also an MST for  $G'$  with weight  $w + |V| - 1$ .

- (d) **T F** (2010 Fall Quiz 1) If we use a max-queue instead of a min-queue in Kruskal's MST algorithm, it will return the spanning tree of maximum total cost (instead of returning the spanning tree of minimum total cost). (Assume the input is a weighted connected undirected graph.)

**Justification:**

**Solution:** True. The proof is essentially the same as for the usual Kruskal's algorithm. Alternatively, this is equivalent to negating all the edge weights and running Kruskal's algorithm.

**Problem 3. Short Answer** [20 points] (4 parts)

Give *brief*, but complete, answers to the following questions.

- (a) (2011 Fall Quiz 1) If you run Johnson's all-pair shortest-paths algorithm on a graph with no negative-weight edges, what can you say about the reweighted values  $w_h(u, v)$  compared to the original weight values  $w(u, v)$ ?

**Solution:** Since the graph has no negative-weight edges, the shortest path from the new vertex  $s$  to any vertex has weight 0. Thus,  $h(v) = 0$  for any vertex  $v$ . Therefore,  $w_h(u, v) = w(u, v)$ .

- (b) (2011 Spring Practice Quiz 1) Solve the following recurrence:  $T(n) = 9T(n^{1/3}) + \Theta(\log(n))$ .

**Solution:** Let  $n = 2^m$ . Then the recurrence becomes  $T(2^m) = 9T(2^{m/3}) + \Theta(m)$ . Setting  $S(m) = T(2^m)$  gives us  $S(m) = 9S(m/3) + \Theta(m)$ . Using case 1 of the Master's Theorem, we get  $S(m) = \Theta(m^2)$  or  $T(n) = \Theta(\log^2(n))$ .

- (c) (2010 Fall Quiz 1) Solve the following recurrence:  $T(n) = 4T(n/4) + \Theta(n \log n)$

**Solution:**  $T(n) = \Theta(n \log^2 n)$  by case 2 of the (Extended) Master Theorem.

- (d) (2008 Fall Quiz 1) Suppose you are given numbers  $r_1, r_2, \dots, r_n$  and want to compute the coefficients of the degree  $n$  polynomial with exactly those roots, i.e.  $\prod_{i=1}^n (x - r_i)$ . Give an  $O(n \log^2 n)$  algorithm.

**Solution:** A simple recursive algorithm suffices. Compute  $\prod_{i=1}^{\lfloor n/2 \rfloor} (x - r_i)$  and  $\prod_{i=\lfloor n/2 \rfloor + 1}^n (x - r_i)$  and then multiply the two degree- $n/2$  polynomials together using fast polynomial multiplication via the FFT. The base case is clearly constant time and the combine step takes  $O(n \log n)$  time. Thus the running time is given by the recurrence  $T(n) = 2T(n/2) + \Theta(n \log n)$ . By case 2 of the extended Master Theorem,  $T(n) = O(n \log^2 n)$ .

**Problem 4. Tree Cover (2009 Fall Quiz 1)**

Let  $T$  be a rooted tree with  $n$  nodes  $1, 2, \dots, n$  and parent array  $P[1, 2, \dots, n]$  where  $P[i] = j$  if the node  $j$  is the parent of  $i$ , and  $P[i] = i$  if  $i$  is the root of the tree. Thus the edge set of the tree is  $i, P[i]|P[i] \neq i$ .

We say that a vertex  $c$  \*covers\* an edge  $e = (u, v)$  if  $c$  is one of the endpoints of  $e$ , that is,  $c = u$  or  $c = v$ . The \*vertex cover problem\* is the following: given a tree  $T$ , find a minimum-size subset of vertices  $S \subseteq 1, 2, \dots, n$  such that  $S$  covers all edges, i.e., every edge  $i, P[i]$  in the tree is covered by at least one vertex in  $S$ .

Give a polynomial-time greedy algorithm to solve the problem above and prove its correctness. You may assume that each node  $x$  has an attribute  $x.\text{depth}$  that stores the depth of  $x$  in the tree.

**Solution:** Our algorithm works as follows:

1. Sort all the nodes in a nonincreasing order of  $\text{depth}$ .
2. Set  $S$  to  $\emptyset$ .
3. Scan through all the nodes in the sorted array, for a node  $x$ , if the edge  $(x, P[x])$  has not been covered, add  $P[x]$  ( $x$ 's parent node) into  $S$ .

Running time analysis: This algorithm takes  $O(n \log n)$  to sort, and  $O(n)$  to scan through the sorted array (i.e., each step in the scan can be implemented in  $O(1)$  time). Therefore, the total running time is  $O(n \log n)$ .

Proof of correctness: Observe that at the time when we consider a vertex  $x$  in our algorithm, all edges incident to vertices before  $x$  are already covered. When we consider a vertex  $x$ , if edge  $(x, P[x])$  which has not been covered, in order to cover this edge we have to either pick  $x$  or  $P[x]$  (picking both is wasteful).

1. Greedy choice property: Picking  $P[x]$  is the best option since  $P[x]$  covers at least as many (uncovered) edges as  $x$  does.
2. Optimal substructure:  $P[x]$  covers all (uncovered) edges that  $x$  covers (which is  $(x, P[x])$ ). Therefore, if there is an optimal solution that picks  $x$ , there should be another optimal solution that replaces  $x$  by  $P[x]$ . Therefore, picking  $P[x]$  should lead to an optimal solution.

**Problem 5. Searching Noisy Sequences, generalized (2011 Fall Quiz 1)**

You are given a sequence of symbols  $t = t[0] \dots t[n - 1]$ , and a pattern  $p = p[0] \dots p[m - 1]$ , where the symbols  $p[i], t[i]$  come from the set  $\{0 \dots L\}$ . The goal is to *match*  $p$  and  $t$ . That is, for a specified *symbol scoring function*  $sc(a, b)$  that returns the score of a match between symbols  $a$  and  $b$ , the goal is to find the *match score*

$$score(s) = \sum_{i=0}^{m-1} sc(t[s + i], p[i])$$

for all shifts  $s = 0 \dots n - m$ .

In recitation you have seen an algorithm (for a closely related problem) that works for the binary alphabet, i.e.,  $L = 2$ . Our goal is to extend that algorithm to make it work for larger alphabet sizes.

We consider a scoring function  $sc$  such that  $sc(a, b) = 0$  if  $a = b$  and  $sc(a, b) = 1$  otherwise. That is,  $score(s)$  computes the number of *mismatches* between  $p$  (shifted by  $s$ ) and  $t$ .

Give an algorithm that computes  $score(s)$  for all shifts  $s$ . For full credit your algorithm should run in time  $O(Ln \log n)$ .

**Solution:** Recall from recitation the algorithm using FFT that solves the binary pattern matching problem in  $O(n \log n)$  time. Our strategy is to find the number of *matches* between  $p$  and  $t$  first. This allows us to find the number of mismatches trivially by subtracting from  $m$ , the length of the pattern. We will find the total number of matches by summing the number of matches of all symbols  $a \in \{0 \dots L\}$ . To do so, we replace all the occurrences of symbol  $a$  in  $p$  and  $t$  by 1, and set everything else to 0. Running the algorithm from recitation gives us the number of matches of symbol  $a$  for each offset. We repeat this  $L$  times for every symbol, and sum up the results. This solves the problem in  $O(Ln \log n)$  as required.

Variations:

- The above solution essentially encodes  $a$  into  $L$  bits using unary representation, i.e., the  $a$ -th bits of the encoding is set to 1 and others to 0. One could instead try binary encoding where each symbol  $a$  is mapped to its binary representation of length  $\lceil \log(L + 1) \rceil$ . Unfortunately, this does not preserve the number of mismatches exactly. E.g., for  $L=15$ , the symbol 0 is mapped to 0000 while 15 is mapped to 1111, so the number of mismatched gets multiplied by 4.

**Problem 6. Optimal coins (2011 Fall Quiz 1)**

You are given a sequence of  $n$  coins:  $C_1, \dots, C_n$ .

Each coin is biased: you are told that the  $i$ -th coin  $C_i$  has probability  $p_i$  of coming up heads (call this "1") and probability  $q_i = 1 - p_i$  of coming up tails (call this "0"). You are given the values of  $p_i$  (and of  $q_i$ ) for  $i = 1, 2, \dots, n$ .

Next you are given a sequence  $G_j$ ,  $j = 1, 2, \dots, k$  where each  $G_j$  is either 1 or 0. Here  $1 \leq k \leq n$ .

You are asked to efficiently find an increasing subsequence  $i_1, i_2, \dots, i_k$  of  $1, 2, \dots, n$  such that when all coins are flipped independently

$$\text{Prob}(C_{i_j} = G_j \text{ for all } j = 1, 2, \dots, k)$$

is maximized.

Give an efficient algorithm for finding such a subsequence that maximizes your chance of obtaining the given goal sequence  $G_1, G_2, \dots, G_k$  for the indices your algorithm produces.

**Example:** Consider a sequence of probabilities  $p_1, \dots, p_4$  equal to 0.7, 0.1, 0.2, 0.8, and a sequence  $G_1, G_2$  of coins equal to 0, 1. In this case, the optimal solution  $i_1, i_2$  can be seen to be equal to 2, 4, since the probability

$$\text{Prob}(C_2 = 0 \text{ and } C_4 = 1) = (1 - 0.1) \cdot 0.8 = 0.72$$

is maximized.

**Solution:** We will use dynamic programming.

For  $1 \leq i \leq n$  and  $1 \leq r \leq k$ , let  $P(i, r) = p_i \cdot G_r + q_i \cdot (1 - G_r)$ . In addition, let  $\text{OPT}(i, r)$  be the value of an optimal assignment to  $G_1, \dots, G_r$  using coins from the subsequence  $C_1, \dots, C_i$ . Then, we have that our base cases are:

$$\text{OPT}(i, 1) = \max_{1 \leq j \leq i} P(i, 1)$$

$$\text{OPT}(i, r) = 0, \text{ if } i < r$$

The recurrence is the following (for  $2 \leq r \leq k$  and  $r \leq i \leq n$ ):

$$\text{OPT}(i, r) = \max \{P(i, r) \cdot \text{OPT}(i - 1, r - 1), \text{OPT}(i - 1, r)\}$$

There are  $nk$  subproblems, and based on the recursion it takes  $O(1)$  to solve a problem based on the solutions of the subproblems. Hence, our running time is  $O(nk)$ .

## SCRATCH PAPER

## SCRATCH PAPER

## Practice Final Exam Solutions

**Problem Final-1.** [50 points] (9 parts)

In each part of this problem, circle *all* of the choices that correctly answer the question or complete the statement. Cross out the remaining (incorrect) choices, and in the space provided, briefly explain why the item is wrong. (You need not explain correct answers.) If you circle or cross out incorrectly, you will be penalized, so leave the choice blank unless you are reasonably sure.

- (a) Which of the following algorithms discussed in 6.046 are greedy?
- ① Prim's algorithm for finding a minimum spanning tree
  - ② finding an optimal file merging pattern
  - ✗ finding a longest common subsequence of two strings
  - ④ Dijkstra's algorithm for solving the single-source shortest paths problem
3. *Finding a longest common subsequence of two strings uses a dynamic programming approach.*
- (b) An adversary cannot elicit the worst-case behavior of:
- ✗ ordinary quicksort.
  - ② a hash table where universal hashing is used.
  - ✗ a self-organizing list where the move-to-front (MTF) heuristic is used.
  - ④ RANDOMIZED-SELECT for finding the  $k$ th smallest element of an array.
1. *An adversary can beat a deterministic partitioning scheme with a bad input, such that all elements fall to one side of the pivot.*
3. *An adversary can always access the tail element of the list, regardless of the online heuristic.*
- (c) Which of the following can be performed in worst-case linear time in the input size?
- ① building a binary heap from an unsorted list of numbers
  - ② determining if a graph is bipartite
  - ③ walking a red-black tree inorder
  - ④ solving the single-source shortest paths problem on an acyclic directed graph

(d) Which of the following statements about trees are correct?

- ① Given a set  $S$  of  $n$  real keys chosen at random from a uniform distribution over  $[a, b]$ , a binary search tree can be constructed on  $S$  in  $O(n)$  expected time.
- ② In the worst case, a red-black tree insertion requires  $O(1)$  rotations.
- ③ Given a connected, weighted, undirected graph  $G$  in which the edge with minimum weight is unique, that edge belongs to every minimum spanning tree of  $G$ .
- ✗ Deleting a node from a binary search tree on  $n$  nodes takes  $O(\lg n)$  time in the worst case.

4. *Deletion on an ordinary binary search tree of height  $h$  takes  $\Theta(h)$  time in worst case, and  $h$  can be  $\Omega(n)$  if the tree with  $n$  nodes is unbalanced.*

(e) Which of the following algorithms discussed in 6.046 employ dynamic programming?

- ✗ Kruskal's algorithm for finding a minimum spanning tree
  - ② the Floyd-Warshall algorithm for solving the all-pairs shortest paths problem
  - ③ optimal typesetting, where the line penalty is the cube of the number of extra spaces at the end of a line
  - ✗ the Bellman-Ford algorithm for solving the single-source shortest paths problem
1. *Kruskal's algorithm uses a greedy strategy.*
  4. *The Bellman-Ford algorithm simply uses repeated edge relaxation.*

(f) Which of the following correctly match a theoretical result from 6.046 with an important technique used in a proof of the result?

- ① correctness of HEAPIFY ... induction on subtree size
  - ✗ 4-competitiveness of the move-to-front (MTF) heuristic for self-organizing lists ... cut-and-paste argument
  - ③ correctness of Dijkstra's algorithm for solving the single-source shortest paths problem ... well ordering and proof by contradiction
  - ④ expected height of a randomly built binary search tree ... indicator random variables and Chernoff bound
2. *This result uses an amortized or “potential-function” argument in its proof.*

(g) On inputs for which both are valid, in the worst case:

- ✗ breadth-first search asymptotically beats depth-first search.

- insertion into an AVL tree asymptotically beats insertion into a 2-3-4 tree.
  - (3) Dijkstra's algorithm asymptotically beats the Bellman-Ford algorithm at solving the single-source shortest paths problem.
  - shellsort with the increment sequence  $\{2^i 3^j\}$  has the same asymptotic performance as mergesort.
1. Both breadth-first and depth-first search are  $O(V + E)$ .
  2. Both AVL trees and 2-3-4-trees are balanced, so insertion take  $O(\lg n)$  time.
  4. Mergesort, a  $\Theta(n \lg n)$  algorithm, beats shellsort, a  $\Theta(n \lg^2 n)$  algorithm.

**Problem Final-2.** [25 points] (5 parts)

In parts (a)–(d), give asymptotically tight upper (big  $O$ ) bounds for  $T(n)$  in each recurrence. Briefly justify your answers.

(a)  $T(n) = 4T(n/2) + n^2$ .

By case 2 of Master Theorem:  $T(n) = \Theta(n^2 \lg n)$

(b)  $T(n) = T(n/2) + n$ .

By case 3 of Master Theorem:  $T(n) = \Theta(n)$ . Note:  $f(n) = n$  is regular.

(c)  $T(n) = 3T(n/3) + \lg n$ .

By case 1 of Master Theorem:  $T(n) = \Theta(n)$

(d)  $T(n) = 2T(n/2) + \lg(n!)$ .

By case 3 of Master Theorem:  $T(n) = \Theta(n \lg^2 n)$ . Note:  $\Theta(\lg(n!)) = \Theta(n \lg n)$

(e) Use the substitution method to prove that the recurrence

$$T(n) = 2T(n/2) + n$$

can be bounded below by  $T(n) = \Omega(n \lg n)$ .

Assume  $T(k) \geq c k \lg k$  for  $k < n$ .

$$\begin{aligned} T(n) &= 2T(n/2) + n \\ &\geq cn \lg(n/2) + n \\ &= cn \lg n - cn + n \\ &\geq cn \lg n \quad \text{if } c \leq 1. \end{aligned}$$

The initial conditions are satisfiable, because  $T(2) \geq 2c$  and  $T(3) \geq 3c \lg 3$ , which can be achieved by making  $c$  sufficiently close to 0.

**Problem Final-3.** [25 points]

Use a potential-function argument to show that any sequence of insertions and deletions on a red-black tree can be performed in  $O(\lg n)$  amortized time per insertion and  $O(1)$  amortized time per deletion. (For substantial partial credit, use an aggregate or accounting argument.)

We define the potential function  $\varphi$  of a red-black tree  $T$  to be

$$\varphi(T) = k \sum_{i=1}^n \lg i,$$

where  $n$  is the number of nodes in  $T$  and  $k > 0$  is a constant.

The initial tree  $T_0$  is an empty red-black tree and we have  $\varphi(T_0) = 0$ . For any tree  $T$  obtained applying insertions and deletions to  $T_0$  the potential function is  $\varphi(T) \geq 0$ . Hence  $\varphi$  is a well-defined potential function.

Next we compute the amortized cost

$$\hat{c} = c + \varphi(T_{\text{after}}) - \varphi(T_{\text{before}})$$

where  $c$  = actual cost of the operation,  $T_{\text{after}}$  = tree after the operation,  $T_{\text{before}}$  = tree before the operation.

Let  $n$  be the number of nodes in  $T$  before the operation.

Amortized cost for insertion (actual cost  $c \leq a_1 \lg n + a_2$ ):

$$\hat{c}_{\text{INSERTION}} = a_1 \lg n + a_2 + k \sum_{i=1}^{n+1} \lg i - k \sum_{i=1}^n \lg i = (a_1 + k) \lg(n+1) + a_2$$

Amortized cost for deletion (actual cost  $c \leq b_1 \lg n + b_2$ ):

$$\hat{c}_{\text{DELETE}} = b_1 \lg n + b_2 + k \sum_{i=1}^{n-1} \lg i - k \sum_{i=1}^n \lg i = (b_1 - k) \lg n + b_2$$

By choosing  $k = b_1$  we get:  $\hat{c}_{\text{DELETE}} = O(1)$  and  $\hat{c}_{\text{INSERTION}} = O(\lg n)$ .

**Problem Final-4.** [25 points]

Bitter about his defeat in the presidential election, Rob Roll decides to hire a seedy photographer to trail Will Clintwood. (The names have been changed to protect the guilty.) The photographer stealthily takes pictures of Clintwood, and he marks each picture with the time  $t$  it was taken. Roll tells the photographer to mark especially scandalous pictures with a big, red X, because these pictures will be used in future negative advertisements. Roll requires a data structure that contains the Clintwood pictures and supports the following operations:

- **INSERT( $x$ )**—Inserts picture  $x$  into the data structure. Picture  $x$  has an integer field  $time[x]$  and a boolean field  $scandalous[x]$ .
- **DELETE( $x$ )**—Deletes picture  $x$  from the data structure.
- **NEXT-PICTURE( $x$ )**—Returns the picture that was taken immediately after picture  $x$ .
- **SCANDAL!( $t$ )**—Returns the first scandalous picture that was taken after time  $t$ .

Describe an efficient implementation of this data structure. Show how to perform these operations, and analyze their running times. Be succinct.

To allow us to implement these operations efficiently, we'll use a red-black tree representation keying photographs by time, and augmenting each node  $x$  with these additional fields:  $scandalous?[x]$  and  $max-scand-time[x]$ . The field  $max-scand-time[x]$  stores the time when the latest scandalous picture in the subtree rooted at  $x$  was taken. It contains either an integer time, or NIL if there are no scandalous pictures in the subtree. We implement the operations (all in  $O(\lg n)$  running time) on this data structure as follows:

**INSERT( $x$ ):** Place  $x$  in the tree, keyed by  $time[x]$ , just like in a regular red-black tree. Update  $max-scand-time$  fields if necessary after rotations. This can be done in  $O(1)$  time.

**DELETE( $x$ ):** Just like **INSERT( $x$ )**; do what's normally done in a red-black tree, taking care to fix  $max-scand-time$  fields after deleting node  $x$ . When node  $n$  is encountered where  $max-scand-time[n] = time[x]$  and  $scandalous?[x] = \text{TRUE}$ , look at the node's children and set  $max-scand-time[n] \leftarrow \max(max-scand-time[right[n]], max-scand-time[left[n]])$ .

**NEXT-PICTURE( $x$ ):** Just like **SUCCESSOR( $x$ )** in a red-black tree.

**SCANDAL!( $t$ ):** We use the  $max-scand-time$  field to search for this element. If there is no such element in the tree, i.e.  $max-scand-time[root] < t$ , then return NIL. Otherwise we recursively descent the tree as follows: at each node  $n$  check whether  $max-scand-time[left[n]] > t$ . If so recursively descent to the left subtree. Otherwise, check whether  $time[n] > t$  and  $scandalous?[n] = \text{TRUE}$ . If so return  $n$ . Otherwise, recursively descent the right subtree.

## Practice Final

- Do not open this exam until you are directed to do so. Read all the instructions first.
- When the exam begins, write your name on every page of this exam booklet.
- The exam contains 8 multi-part problems. You have 180 minutes to earn 160 points.
- This exam booklet contains 14 pages, including this one. Two extra sheets of scratch paper are attached. Please detach them before turning in your exam.
- This exam is closed book. You may use two handwritten A4 or  $8\frac{1}{2}'' \times 11''$  crib sheets. No calculators or programmable devices are permitted.
- 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.
- Good luck!

Problem	Points	Grade	Initials
1	12		
2	18		
3	27		
4	20		
5	16		
6	21		
7	26		
8	20		
Total	160		

Name: Solutions \_\_\_\_\_

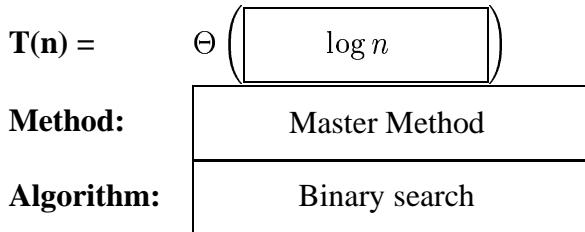
Circle the name of your recitation instructor:

**Problem -1. Recurrences [12 points]**

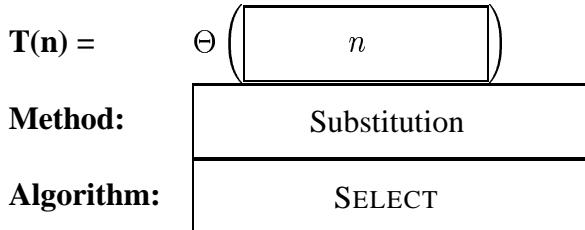
For each of the following recurrences, do the following:

- Give the solution in  $\Theta(\cdot)$  notation.
- Name a method that can be used to solve the recurrence. (**Do not give a proof.**)
- Mention a recursive algorithm we've seen in class whose running time is described by that recurrence.

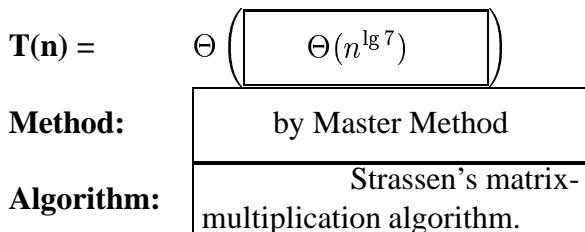
(a)  $T(n) = T(n/2) + \Theta(1)$



(b)  $T(n) = T(n/5) + T(7n/10) + \Theta(n)$



(c)  $T(n) = 7T(n/2) + \Theta(n^2)$



**Problem -2. Design Decisions [18 points]**

The use of algorithms often requires choices. What is efficient in one context may be suboptimal in another. For each of the following pairs, give *one* reason, circumstance, or application for which Choice 1 would be preferable to Choice 2, and vice-versa. Be succinct.

**EXAMPLE:** (1) insertion sort (2) merge sort

(1) Insert sort is preferable on small arrays, or when space is a limited resource.

(2) Merge sort is preferable on large arrays, because of its asymptotically optimal  $O(n \log n)$  running time.

- (a)** (1) randomized quicksort (2) bucket sort

**Solution:** (1) Randomized quicksort gives better worst-case running time when no distribution is known on the input.

(2) Bucket sort is better (linear-time) when the distribution on input elements is known to be uniform, or when randomness is not available.

- (b)** (1) randomized select (2) worst-case  $O(n)$ -time select

**Solution:** (1) Randomized select has smaller constant factors, so it is better suited for daily use when hard, worst-case guarantees on running time may not be needed.

(2) Linear-time select is preferred when a hard upper limit on the running time is necessary (as in a real-time application), or when randomness is not available.

- (c)** (1) red-black tree (2) hash table

**Solution:** (1) Red-black trees are useful for “approximate” searches, augmentation, and guaranteed worst-case running times.

(2) Hash tables support constant time (in expectation) insertions and deletions, and can be designed with guaranteed  $O(1)$ -time operations when the data set is known in advance.

**Problem -3. Short Answer** [27 points]

Give *brief*, but complete, answers to the following questions.

- (a) Suppose 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.

**Solution:** We use universal hashing to solve this problem. Create a hash table of  $2n$  elements, and for each element  $x$  in  $A$ , search for  $x$  in the table and insert it only if the search fails. Then walk down the slots of the table and output every element. The searches and insertions each take  $O(1)$  expected time, and walking down the table takes  $O(n)$  time, for a total expected runtime of  $O(n)$ .

- (b) Given a list of distinct real numbers  $z_0, z_1, \dots, z_{n-1}$ , show how to find the coefficients of the degree- $n$  polynomial  $P(x)$  that evaluates to zero only at  $z_0, z_1, \dots, z_{n-1}$ . Your procedure should run in time  $O(n \log^2 n)$ . (*Hint:* The polynomial  $P(x)$  has a zero at  $z_j$  if and only if  $P(x)$  is a multiple of  $(x - z_j)$ .)

**Solution:** We multiply all the polynomials  $(x - z_j)$  together, in the following divide-and-conquer way: recursively multiply the first  $n/2$  linear terms, then recursively multiply the remaining  $n/2$  linear terms, then combine the two results by multiplying them together using FFT. The running time is expressed by the recurrence  $T(n) = 2T(n/2) + \Theta(n \log n)$ . By the Master Method, this solves to  $T(n) = \Theta(n \log^2 n)$ .

- (c) Consider a generalization of the max-flow problem, in which the network  $G$  may have many sources and many sinks. Explain how to reduce this problem to the conventional max-flow problem. Specifically, describe how you would convert the generalized network  $G$  to a conventional network  $G'$ , and how you would translate a maximum flow in  $G'$  back to a maximum flow in  $G$ .

**Solution:** First, make a copy of  $G$ . Then add a new “supersource” vertex  $s$ , and a new “supersink” vertex  $t$ . Connect  $s$  to every old source with infinite-capacity edges, and connect every old sink to  $t$  with infinite-capacity edges. This graph is  $G'$ . Then run a conventional max-flow algorithm on  $G'$  with  $s$  as the source and  $t$  as the sink.

The flow produced is a max flow in  $G$ , discounting the edges between  $s$  and the old sources, and between the old sinks to  $t$ . This is because it is a valid flow, and the min-cut in  $G'$  is the same as in  $G$  (because only infinite-capacity edges were added).

**Problem -4. True or False, and Justify [20 points]**

Circle **T** or **F** for each of the following statements, and briefly explain why. The better your argument, the higher your grade, but be brief. Your justification is worth more points than your true-or-false designation.

- (a) **T F** If a problem  $L$  is in  $NP$  and  $L$  reduces to  $3SAT$  ( $L \leq_P 3SAT$ ), then  $L$  is  $NP$ -complete. (You may assume  $P \neq NP$ .)

**Solution:** False. Every problem in  $P$  reduces to  $3SAT$ , and if  $P \neq NP$  then those problems are not  $NP$ -complete. The statement *would* be true if the reduction went the other way (if  $3SAT$  reduced to  $L$ ).

- (b) **T F** If a graph has negative-weight edges, there exists a constant  $C$  such that adding  $C$  to every edge weight and running Dijkstra's algorithm produces shortest paths under the original edge weights.

**Solution:** False. Adding the same value to each edge "distorts" the shortest paths, because each path becomes longer in proportion to the number of its edges. For concreteness, suppose the shortest path between  $s$  and  $t$  was 3 "hops," and was 1 unit shorter than the edge directly from  $s$  to  $t$ . Furthermore, if there was an edge somewhere in the graph with weight  $-1$ , we would need  $C \geq 1$  for Dijkstra's algorithm to be correct. Adding  $C$  to each edge causes the 1-hop path from  $s$  to  $t$  to be the shortest.

- (c) **T F** It is possible to compute the convolution of two vectors, each with  $n$  entries, in  $O(n \lg n)$  time.

**Solution:** True. Use polynomial multiplication via FFT.

- (d) **T F** The following procedure produces a minimum spanning tree of  $n$  given points in the plane.

- Sort the points by  $x$ -coordinate. (Assume that all  $x$ -coordinates are distinct.)
- Connect each point to its closest neighbor among all points with smaller  $x$  coordinate.

That is, if  $p_1, p_2, \dots, p_n$  denotes the sorted order of points, then for  $2 \leq i \leq n$ , connect point  $p_i$  to its closest neighbor among  $p_1, p_2, \dots, p_{i-1}$ .

**Solution:** False. Here is a counterexample: take the points  $(0, 0)$ ,  $(10, 100)$ , and  $(11, 0)$  in the  $xy$ -plane. Then in the first step, the edge between  $(0, 0)$  and  $(10, 100)$  is chosen; however, this is the longest of the three edges between points. We know that an MST cannot contain the heaviest edge in a graph, therefore this algorithm is incorrect.

**Problem -5. Average Path Lengths in DAGs [16 points]**

Instead of shortest paths in a graph, you might be interested in the *average* length of all paths from a vertex  $s$  to a vertex  $t$ . In order for this to make sense, we assume that the graph  $G = (V, E)$  is directed and acyclic. Let  $w(u, v)$  denote the weight of edge  $(u, v)$ . It suffices to compute the total length of all paths from  $s$  to  $t$ , and the number of such paths. This problem is well-suited to a dynamic programming approach.

- (a) [8 points]** Define  $count[x]$  to be the number of distinct paths from  $x$  to  $t$ . Define  $sum[x]$  to be the sum of the lengths of all paths from  $x$  to  $t$ . Give recurrences for  $count[x]$  and  $sum[x]$  in terms of the neighbors of  $x$  and the edge weights, and give the base cases as well.

**Solution:** The paths from  $x$  to  $t$  can be partitioned based on the neighbor  $y$  of  $x$  visited first in the path. There are  $count[y]$  distinct paths to take from that point onward. The corresponding paths from  $x$  are each  $w(x, y)$  longer. This yields the recurrences:

$$\begin{aligned} count[x] &= \sum_{y : (x,y) \in E} count[y] \\ sum[x] &= \sum_{y : (x,y) \in E} (count[y] \cdot w(x, y) + sum[y]). \end{aligned}$$

The base cases are  $count[t] = 1$  and  $sum[t] = 0$ .

- (b) [8 points]** Describe a dynamic-programming algorithm to solve the stated problem, and analyze its running time. (*Hint:* In what order should you consider the vertices, so that when considering vertex  $x$ , the values of  $count[y]$  and  $sum[y]$  have already been computed for all neighbors  $y$  of  $x$ ?)

**Solution:** We first compute a topological ordering on the vertices, relabelling them  $1, \dots, |V|$ , so that all edges  $(i, j) \in E$  have  $i < j$ . Then we build tables for  $count$  and  $sum$ , filling them in from  $V$  down to 1. Values in both tables are set to 0, until vertex  $t$  is encountered, at which point we set the table entries according to the base cases. We then use the recurrences to fill in the remaining entries. This is possible because when computing the entries for vertex  $x$ , all neighbors  $y$  of  $x$  are greater than  $x$ , and have had their entries computed already. After filling in the tables, we return  $sum[s]/count[s]$  as the average.

The runtime analysis is as follows: topological sort requires  $\Theta(V + E)$  time. While filling in the tables, each vertex and edge causes a constant number of math operations to be done, for  $\Theta(V + E)$  time overall.

**Problem -6. Amortized 2-3-4 Trees [21 points]**

In this problem, we will analyze the amortized number of modifications made to a 2-3-4 tree during an INSERT operation. For the purpose of this problem, assume that DELETE operations do not occur.

- (a) [3 points] Give a tight asymptotic bound on the number of nodes created or modified during one call to INSERT, in the worst case.

**Solution:**  $\Theta(\log n)$  nodes could be changed, because the height of the tree is  $\Theta(\log n)$  and each node in the insertion path could be full and need to be split.

- (b) [6 points] Suppose we insert an element into a 2-3-4 tree, and the INSERT algorithm splits  $k$  nodes. Give an *exact* (not big-O) upper bound on the number of nodes in the tree that are created or modified in this case.

**Solution:** Each split creates 2 new nodes (by splitting one), and modifies the parent of the old node (by promoting one key). In addition, the leaf node that gets the inserted key is modified (whether or not there are any splits). This yields a total of at most  $3k + 1$  modified nodes (alternatively, we could say  $\max(3k, 1)$ ).

- (c) [6 points] Let  $T$  be a 2-3-4 tree, and define a potential function

$$\phi(T) = 3 \times (\text{number of "full" nodes in } T)$$

(a node is full if it stores 3 keys).

If a call to `INSERT` splits  $k$  nodes, how does  $\phi(T)$  change as a result of this call?

**Solution:** Each node that is split was full before the split, and is not full afterward. No other full nodes are modified; however, the key that is promoted in the final split may fill a node. Therefore the number of full nodes decreases by  $k - 1$ , so  $\phi(T)$  decreases by at least  $3(k - 1)$ . (Note: all of these statements remain true when  $k = 0$  or  $k = 1$ ; in these cases  $\phi$  “decreases” by a small, non-positive amount.)

- (d) [6 points] Prove that the amortized number of nodes created or modified per `INSERT` is  $O(1)$ .

**Solution:** First, we note that  $\phi(T) = 0$  on newly-initialized B-tree, and that  $\phi(T) \geq 0$  by definition. Therefore, the amortized number of modified nodes per `INSERT` is the actual number of modified nodes, plus the change in  $\phi(T)$ . This is at most  $3k + 1 - (3k - 3) = 4 = O(1)$ .

**Problem -7. Maximum Bottleneck Path** [26 points]

In the **maximum-bottleneck-path** problem, you are given a graph  $G$  with edge weights, and two vertices  $s$  and  $t$ , and your goal is to find a path from  $s$  to  $t$  whose minimum edge weight is maximized. In other words, you want to find a path from  $s$  to  $t$  in which *no* edge is light.

- (a) [8 points] Suppose that all edges have nonnegative weights. How would you modify a shortest-path algorithm that we covered in lecture to solve the maximum-bottleneck-path problem?

**Solution:** We would like to use Dijkstra's algorithm, where  $d[v]$  keeps a *lower* bound on the *maximum* bottleneck weight of a path from  $s$  to  $v$ . Therefore, in the initialization step, we set  $d[v] \leftarrow -\infty$  for all  $v$ , and  $d[s] \leftarrow \infty$ . In the relaxation step, we change  $+$  to  $\min$  and change  $>$  to  $<$ . In Dijkstra's algorithm, use a max-heap instead of a min-heap.

- (b) [4 points] Does your solution change if the edges have negative weights? What if there are negative-weight cycles?

**Solution:** Neither of these are a problem. The minimum-weight edge of a path is not changed by going around a cycle several times. Because we are only using  $\min$  instead of addition, the relative values of the edges are all that matter, not whether they are positive or negative.

- (c) [6 points] Suppose we do not need a path which maximizes the minimum edge weight, but we only need a path in which every edge has at least a certain weight. Describe an  $O(V + E)$ -time algorithm for finding a path from  $s$  to  $t$  in which every edge has least a given minimum weight  $w_{\min}$ . (Such an algorithm would have been useful in the movie *Speed*, in which every road traversed had to have a speed limit of at least 50 mph.)

**Solution:** We can find a path using breadth-first or depth-first search, while simply deleting (or ignoring) edges with weight smaller than  $w_{\min}$ .

- (d) [8 points] Describe how you can make  $O(\lg E)$  calls to the algorithm in part (c) to solve the maximum-bottleneck-path problem in  $O((V + E) \lg E)$  time.

**Solution:** First, sort all the edges by weight (there are at most  $|E|$  unique weights). The desired weight must be a particular edge weight, so we binary search for the largest value of  $w_{\min}$  that maintains a path from  $s$  to  $t$  (using the algorithm from the previous part to detect whether such a path exists). Specifically, start with  $w_{\min}$  set to the median edge weight, then try the  $E/4$ th or  $3E/4$ th largest edge weight, etc.

Sorting the edge weights requires  $O(E \log E)$  time. The binary search requires  $O(\log E)$  iterations of an  $O(V + E)$ -time subroutine, so the total running time is  $O((V + E) \log E)$ .

**Problem -8. Cliquependent Graphs [20 points]**

Given a graph  $G = (V, E)$  and nonnegative integers  $c, s$ , we say that  $G$  is  $(c, s)$ -**cliquependent** if both of the following are true:

- there exists a subset  $C \subseteq V$  such that  $|C| = c$  and, for all distinct  $i, j \in C$ ,  $(i, j) \in E$ , and
- there exists a subset  $S \subseteq V$  such that  $|S| = s$  and, for all distinct  $i, j \in S$ ,  $(i, j) \notin E$ .

Given a graph  $G$  and a pair  $(c, s)$ , the **cliquependence decision problem** is to determine whether  $G$  is  $(c, s)$ -cliquependent.

- (a) [3 points]** Define the set CLIQUEPENDENT which contains all “yes” instances to the cliquependence decision problem.

**Solution:**

$$\text{CLIQUEPENDENT} = \{\langle G, c, s \rangle : G \text{ is } (c, s)\text{-cliquependent}\}$$

- (b) [6 points]** Show that CLIQUEPENDENT  $\in \text{NP}$ .

**Solution:** A witness for an instance is a subset  $C$  and a subset  $S$  which have the properties listed above ( $C$  is a clique of size  $c$ , and  $S$  is an independent set of size  $s$ ). The verification algorithm checks that there is an edge between every pair of vertices in  $C$ , and that there are no edges between any pair of vertices in  $S$ , and accepts the witness only if all the conditions are met. Conversely, if the verifier accepts, then  $C$  and  $S$  have the properties describe above, and  $G$  is  $(c, s)$ -cliquependent.

The witness is clearly polynomial-sized in the size of the instance, and the verification algorithm runs in polynomial time.

- (c) [7 points] Show that CLIQUEPENDENT is NP-complete. (*Hint:* Reduce from either CLIQUE or INDEPENDENT-SET.)

**Solution:** We can reduce from CLIQUE: given an instance  $x = \langle G, k \rangle$  of CLIQUE (“is there a clique of size  $k$  in  $G$ ?”), the reduction outputs an instance  $f(x) = \langle G, k, 0 \rangle$  of CLIQUEPENDENT. Trivially,  $G$  has an independent set of size 0, so  $f(x) \in \text{CLIQUEPENDENT}$  if and only if  $G$  has a clique of size  $k$ , i.e. if and only if  $x \in \text{CLIQUE}$ . This reduction is obviously polynomial-time, and because CLIQUE is NP-complete, so is CLIQUEPENDENT.

We can also reduce from INDEPENDENT-SET in a similar way: on input instance  $x = \langle G, s \rangle$  of INDEPENDENT-SET, the reduction outputs  $f(x) = \langle G, 0, s \rangle$ . Because  $G$  has a clique of size 0,  $x \in \text{INDEPENDENT-SET} \iff f(x) \in \text{CLIQUEPENDENT}$ .

- (d) [4 points] Suppose an  $O(n^{100})$ -time algorithm were found for the cliquependence decision problem. What would be the implications, if any, on the “ $P \stackrel{?}{=} NP$ ” question?

**Solution:** If such a procedure exists, then  $P = NP$  with certainty. This is because any language in NP can be reduced to the cliquependence problem, then the algorithm in question would correctly answer “yes” or “no.” Therefore any problem in NP could be solved in polynomial time, implying  $P = NP$ .

**SCRATCH PAPER** — Please detach this page before handing in your exam.

**SCRATCH PAPER** — Please detach this page before handing in your exam.

## Practice Final Exam for Spring 2012

These problems are four of the seven problems from the final exam given in spring 2011, seven out of ten true or false questions, along with the full instructions given with the exam so that you have a sense of what the final exam will be like.

- Do not open this exam booklet until you are directed to do so. Read all the instructions first.
- The exam contains 7 problems, several with multiple parts. You have 180 minutes to earn 180 points.
- This exam booklet contains 10 pages, including this one, and two sheets of scratch paper which can be detached.
- This exam is closed book. You may use two double sided Letter ( $8\frac{1}{2}'' \times 11''$ ) or A4 crib sheets. No calculators or programmable devices are permitted. Cell phones must be put away.
- Write your solutions in the space provided. If you run out of space, continue your answer on the back of the same sheet and make a notation on the front of the sheet.
- Do not waste time deriving facts that we have studied. It is sufficient to cite known results.
- Do not spend too much time on any one problem. Generally, a problem's point value is an indication of how many minutes to spend on it.
- 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. Please be neat.
- Good luck!

Problem	Title	Points	Parts	Grade	Initials
0	Name	1	10		
1	True or False	40	10		
2	Substitution Method	14	1		
3	Updating a Flow Network	25	3		
4	Candy Land	20	1		
5	Combined-Signal Problem	25	4		
6	Eleanor's Problem Revisited	25	4		
7	Majority Rules	30	5		
Total		180			

Name: \_\_\_\_\_

**Problem 0. Name.** [1 point] (10 parts)

Write your name on every page of this exam booklet! Don't forget the cover.

**Problem 1. True or False.** [40 points] (10 parts)

Circle **T** or **F** for each of the following statements to indicate whether the statement is true or false, respectively, and briefly explain why (one or two sentences). Your justification is worth more points than your true-or-false designation. *Careful:* Some problems are straightforward, but some are tricky!

- (a) **T F** Suppose that every operation on a data structure runs in  $O(1)$  amortized time.

Then the running time for performing a sequence of  $n$  operations on an initially empty data structure is  $O(n)$  in the worst case.

**Solution:** True:  $\sum_{i=1}^n \hat{c}_i \geq \sum_{i=1}^n c_i$ .

- (b) **T F** Suppose that a Las Vegas algorithm has expected running time  $\Theta(n)$  on inputs of size  $n$ . Then there may still be an input on which it always runs in time  $\Omega(n \lg n)$ .

**Solution:** False.

- (c) **T F** If there is a randomized algorithm that solves a decision problem in time  $t$  and outputs the correct answer with probability 0.5, then there is a randomized algorithm for the problem that runs in time  $\Theta(t)$  and outputs the correct answer with probability at least 0.99.

**Solution:** False. Every decision problem has an algorithm that produces the correct answer with probability 0.5 just by flipping a coin to determine the answer.

- (d) **T F** Let  $\mathcal{H} : \{0, 1\}^n \rightarrow \{1, 2, \dots, k\}$  be a universal family of hash functions, and let  $S \subseteq \{0, 1\}^n$  be a set of  $|S| = k$  elements. For  $h$  chosen at random from  $\mathcal{H}$ , let  $E$  be the event that for all  $y \in \{1, 2, \dots, k\}$ , the number of elements in  $S$  hashed to  $y$  is at most 100, that is,  $|h^{-1}(y) \cap S| \leq 100$ . Then we have  $\Pr\{E\} \geq 3/4$ .

**Solution:** False.  $|h^{-1}(y) \cap S|$  is likely to be  $\Theta(\log k / \log \log k)$  for some  $y$ . Only its expectation is  $O(1)$ .

- (e) **T F** Let  $\Sigma = \{a, b, c, \dots, z\}$  be a 26-letter alphabet, and let  $s \in \Sigma^n$  and  $p \in \Sigma^m$  be strings of length  $n$  and  $m < n$  respectively. Then there is a  $\Theta(n)$ -time algorithm to check whether  $p$  is a substring of  $s$ .

**Solution:** True. E.g., using suffix trees.

- (f) **T F** If an iteration of the Ford-Fulkerson algorithm on a network places flow 1 through an edge  $(u, v)$ , then in every later iteration, the flow through  $(u, v)$  is at least 1.

**Solution:** False: A later augmenting path may pass through  $(v, u)$ , causing the flow on  $(u, v)$  to be decreased.

- (g) **T F** There exists a minimization problem such that (i) assuming  $P \neq NP$ , there is no polynomial-time 1-approximation algorithm for the problem; and (ii) for any constant  $\epsilon > 0$ , there is a polynomial-time  $(1 + \epsilon)$ -approximation algorithm for the problem.

**Solution:** True. There are NP-hard optimization problems with a PTAS, such as PARTITION, as we saw in class.

**Problem 2. Substitution Method.** [14 points]

Use the substitution method to show that the recurrence

$$T(n) = \sqrt{n} T(\sqrt{n}) + n$$

has solution  $T(n) = O(n \lg \lg n)$ .

**Solution:** First, prove a base case. For  $4 \leq n \leq 16$ , let  $T(n) = O(1) \leq k$  for some  $k > 0$ . For some  $c \geq k/4$ , we have that  $T(n) \leq cn \lg \lg n$ .

Now, prove the inductive case. Assume  $T(n) \leq cn \lg \lg n$  for some  $c > 0$ . Then:

$$T(n) = \sqrt{n} T(\sqrt{n}) + n \leq \sqrt{n} c \sqrt{n} \lg \lg \sqrt{n} + n$$

We now find  $c > 0$  so that

$$\begin{aligned} \sqrt{n} c \sqrt{n} \lg \lg \sqrt{n} + n &\leq cn \lg \lg n \\ nc(\lg \lg n - \lg 2) + n &\leq cn \lg \lg n \\ -c \lg 2 + 1 &\leq 0 \\ 1 &\leq c \end{aligned}$$

Hence, the inductive case holds for any  $c \geq 1$ . Setting  $c = \max\{k/4, 1\}$ , we have  $T(n) \leq cn \lg \lg n$  for all  $4 \leq n$ .

**Problem 3. Updating a Flow Network [25 points] (3 parts)**

Let  $G = (V, E)$  be a flow network with source  $s$  and sink  $t$ , and suppose that each edge  $e \in E$  has capacity  $c(e) = 1$ . Assume also, for convenience, that  $|E| = \Omega(V)$ .

- (a) Suppose that we implement the Ford-Fulkerson maximum-flow algorithm by using depth-first search to find augmenting paths in the residual graph. What is the worst-case running time of this algorithm on  $G$ ?

**Solution:** Since the capacity out of the source is  $|V| - 1$ , a mincut has value at most  $|V| - 1$ . Thus the running time is  $O(VE)$ .

- (b) Suppose that a maximum flow for  $G$  has been computed using Ford-Fulkerson, and a new edge with unit capacity is added to  $E$ . Describe how the maximum flow can be efficiently updated. (*Note:* It is not the value of the flow that must be updated, but the flow itself.) Analyze your algorithm.

**Solution:** Simply run one more BFS to find one augmenting path in the residual flow network. This costs  $O(E)$  time. One path suffices because all edges have capacity 1 so any augmenting path will have capacity 1 as well.

We could also precompute in  $O(E)$  time for each vertex in the graph an edge on a path either to sink or from source in the residual flow network. (Can't have both if the flow is maximum!) Then, given the new edge we can update the flow in  $O(V)$  time.

- (c) Suppose that a maximum flow for  $G$  has been computed using Ford-Fulkerson, but an edge is now removed from  $E$ . Describe how the maximum flow can be efficiently updated. Analyze your algorithm.

**Solution:** In the residual flow network, find a path from sink to source via the edge to be removed. To do so, find a path from sink to the starting endpoint of the directed edge in the residual flow network and similarly the second segment of the path. Then diminish the flow by one unit along the path. Since all edges have capacity 1, this removes the flow from the edge to be removed. After removing the edge, search for augmenting path in the residual flow network. This is needed in case the edge was not in the minimum cut hence the flow can be redirected. The total cost is  $O(E)$ .

**Problem 4. Eleanor's Problem Revisited.** [25 points] (4 parts)

Recall the following problem abstracted from the take-home quiz, which we shall refer to as **Eleanor's optimization problem**. Consider a directed graph  $G = (V, E)$  in which every edge  $e \in E$  has a length  $\ell(e) \in \mathbb{Z}$  and a cost  $c(e) \in \mathbb{Z}$ . Given a source  $s \in V$ , a destination  $t \in V$ , and a cost  $x$ , find the shortest path in  $G$  from  $s$  to  $t$  whose total cost is at most  $x$ .

We can reformulate Eleanor's optimization problem as a decision problem. **Eleanor's decision problem** has the same inputs as Eleanor's optimization problem, as well as a distance  $d$ . The problem is to determine if there exists a path in  $G$  from  $s$  to  $t$  whose total cost is at most  $x$  and whose length is at most  $d$ .

- (a) Argue that Eleanor's decision problem has a polynomial-time algorithm if and only if Eleanor's optimization problem has a polynomial-time algorithm.

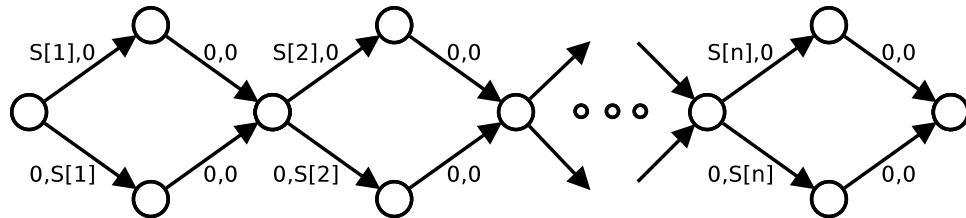
**Solution:** Reduce decision to optimization: solve optimization, output "YES" iff length of found path is less than  $d$ .

Reduce optimization to decision: Compute sum of all edge lengths which gives  $d_{\max}$ . If the decision algo returns "NO" for  $d_{\max}$  then there is no solution. Now binary search for the smallest  $d$  in the range  $[0, d_{\max}]$  on which the decision algo returns "YES". Note  $\log d_{\max}$  is polynomial in input size.

Recall the NP-complete PARTITION problem: Given an array  $S[1 \dots n]$  of natural numbers and a value  $r \in \mathbb{N}$ , determine whether there exists a set  $A \subseteq \{1, 2, \dots, n\}$  of indices such that

$$\max \left\{ \sum_{i \in A} S[i], \sum_{i \notin A} S[i] \right\} \leq r .$$

- (b) Prove that the Eleanor's decision problem is NP-hard. (*Hint:* Consider the graph illustrated below, where the label of each edge  $e$  is “ $c(e), \ell(e)$ ”.)



**Solution:** Label the left-hand vertex in the hint graph as  $s$  and the right-hand vertex  $t$ . Run the algo to the decision problem with this graph and  $x = c$  and  $d = c$  to finish the reduction. Any  $s - t$  path in the graph corresponds to a set  $A$  of the indices of those "widgets" where the path follows the upper branch. The length of this path is  $\sum_{i \in A} S[i]$ , while the cost of this path is  $\sum_{i \notin A} S[i]$ . The decision algo says YES iff there exists a path of both length and cost less than  $c$ , that is, the answer to the PARTITION problem is YES.

- (c) Argue on the basis of part (b) that Eleanor's decision problem is NP-complete.

**Solution:** The witness is the path which can be checked in time linear in the number of its edges, which is less than  $|V|$ . Since the decision problem belongs to NP and is NP-hard, it is NP-complete.

- (d) The solutions to the take-home quiz showed that there is an efficient  $O(xE + xV \lg V)$  algorithm for Eleanor's optimization problem. Why doesn't this fact contradict the NP-completeness of Eleanor's decision problem?

**Solution:** An algorithm with runtime  $O(xE + xV \lg V)$  has runtime polynomial in the **value** of  $x$ . In other words, the runtime is actually pseudo-polynomial, rather than truly polynomial. We showed that Eleanor's optimization problem was hard by a reduction from Partition, which is weakly NP-hard. As a result, we have only shown that Eleanor's Optimization Problem is weakly NP-hard, and so it's not a contradiction to have a pseudo-polynomial algorithm.

## SCRATCH PAPER

## SCRATCH PAPER

## Sample exam (with answers)

- Do not open this quiz booklet until you are directed to do so.
- Quiz ends at 4:30pm. It has 4 problems with multiple parts. You have 180 minutes to earn 90 points. Plan your time wisely.
- This final exam is open-book, open-notes. You may not use other books, bibles, or photocopies.
- 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.
- When describing an algorithm, **please** describe the main idea in English. Use pseudocode only to the extent that it helps clarify the main ideas.
- Good luck!

Problem	Points	Grades	Total
1	??		
2	??		
3	??		
4	??		

Name: \_\_\_\_\_

Please circle your TA's name and recitation:

**Brian   Jon   Josh   Rachel   Yoav**

**10am   11am   12pm   1pm   2pm**

**Problem 1.** Short questions

In the following questions, fill out the blank boxes. In case more than one answer is correct, you should provide the **best known** correct answer. E.g., for a question “Sorting of  $n$  elements in the comparison-based model can be done in  time”, the best correct answer is  $O(n \log n)$ . No partial credit will be given for correct but suboptimal answers. However, you **do not** have to provide any justification.

- (a) Consider a modification to QUICKSORT, such that each time PARTITION is called, the median of the partitioned array is found (using the SELECT algorithm) and used as a pivot.

The worst-case running time of this algorithm is:

**Answer:**  $\Theta(n \log n)$ . Reason: the median can be found in  $O(n)$  time, so we have the running time recurrence  $T(n) = 2T(n/2) + O(n)$ .

- (b) If a data structure supports an operation `foo` such that a sequence of  $n$  `foo`'s takes  $\Theta(n \log n)$  time to perform in the worst case, then the amortized time of a `foo`

operation is  $\Theta\left(\frac{\Theta(n \log n)}{n}\right)$ , while the actual time of a single `foo` operation could be as high as  $\Theta\left(\frac{\Theta(n \log n)}{n}\right)$ .

**Answer:** Amortized time  $\Theta(\log n)$ , worst-case time  $\Theta(n \log n)$ .

- (c) Does there exist a polynomial time algorithm that finds the value of an  $s-t$  minimum cut in a directed graph?  (Yes or No)

**Answer:** Yes. Reason: The value of an  $s - t$  minimum cut is equal to the value of the  $s - t$  maximum flow, and there are many algorithms for computing  $s - t$  maximum flows in polynomial time (e.g. the Edmonds-Karp algorithm).

**Problem 2.** Typesetting

Suppose we would like to neatly typeset a text. The input is a sequence of  $n$  words of lengths  $l_1, l_2, \dots, l_n$  (measured in the number of fixed-size characters they take). Each line can hold at most  $P$  characters, the text is left-aligned, and words cannot be split between lines. If a line contains words from  $i$  to  $j$  (inclusive) then the number of spaces at the end of the line is  $s = P - \sum_{k=i}^j l_k - (j - i)$ . We would like to typeset the text so as to avoid large white spaces at the end of lines; formally, we would like to minimize the sum over all lines of the square of the number of white spaces at the end of the line. Give an efficient algorithm for this problem. What is its running time?

**Answer:**

We solve the problem with dynamic programming. Let  $A[j]$  denote the optimal “cost” (that is, the sum of the square of number of trailing white space characters over all lines) one may achieve by typesetting only the words  $1 \dots j$  (ignoring the remaining words). We can express  $A[j]$  recursively as follows:

$$A[j] = \min_{i < j: T[j] - T[i] \leq P} A[i] + (P - (T[j] - T[i]))^2,$$

where  $T[j] = \sum_{i=1}^j l_i$ . A table of the values  $T[1 \dots n]$  can be initially computed in  $O(n)$  time. The equation above says the following: in order to optimally typeset words  $1 \dots j$ , we must first optimally typeset words  $1 \dots i$  for some  $i < j$ , and then place the remaining words  $i+1 \dots j$  on the final line.

Using dynamic programming, we compute each value  $A[j]$  in sequence for all  $j = 1 \dots n$ . Each value is requires  $O(n)$  time to compute, for a total running time of  $O(n^2)$ . After termination,  $A[n]$  will contain the value of the optimal solution; we can reconstruct the solution itself (that is, the locations where we should insert line breaks) by maintaining “backpointers” as is usually done with dynamic programming.

**Problem 3.** Flows

Let  $G$  be a flow network with integer capacities, and let  $f$  be an integer maximum flow in  $G$ . Suppose that we increase the capacity of an arbitrary edge in  $G$  by one unit. Describe an efficient algorithm to find a maximum flow in the modified flow network. Analyze the worst-case asymptotic running time. Explain why your algorithm is correct.

**Answer:**

## I. Algorithm:

1. Compute the residual network  $G'_f$  from the new graph  $G'$  and flow  $f$ .
2. Run DFS (or BFS) on the residual network  $G'_f$  to find an augmenting path.
3. If an augmenting path is found, increase the flow on the augmenting path by one unit to obtain a new flow  $f'$  which is the maximum flow for  $G'$ .

## II. Analysis

$$\begin{array}{rl}
 \text{step 1} & O(E + V) \\
 \text{step 2} & O(E + V) \\
 \text{step 3} & O(V) \\
 \hline
 \text{Running time} = & O(E + V)
 \end{array}$$

## III. Correctness

Since  $G'$  has integer capacities and  $f$  is an integer max flow, we can augment at most one unit of flow along some path in  $G'_f$ , so one pass of BFS is enough. The rest of the correctness argument is similar to the one for the Ford-Fulkerson algorithm.

**Problem 4.** Princess of Algorithmia (2 parts)

Political upheaval in Algorithmia has forced Princess Michelle X. Goewomans to vacate her royal palace; she plans to relocate to a farm in Nebraska. The princess wants to move all of her possessions from the palace in Algorithmia to the Nebraskan farm using as few trips as possible in her Ferrari. For simplicity, let us assume that the princess's Ferrari has size 1, and that her  $n$  possessions  $x_1, x_2, \dots, x_n$  have real number sizes between 0 and 1. The problem is to divide the princess's possessions into as few carloads as possible, so that all her possessions will be transported from Algorithmia to Nebraska and so that her Ferrari will never be overpacked. It turns out this problem is NP-hard.

Consider the following **first-fit** approximation algorithm. Put item  $x_1$  in the first carload. Then, for  $i = 2, 3, \dots, n$ , put  $x_i$  in the first carload that has room for it (or start a new carload if necessary). For example, if  $x_1 = 0.2$ ,  $x_2 = 0.4$ ,  $x_3 = 0.6$ , and  $x_4 = 0.3$ , the first-fit algorithm would place  $x_1$  and  $x_2$  in the first carload,  $x_3$  in the second carload, and then  $x_4$  in the first carload (where there is still enough room). Note that all decisions are made offline; we divide the princess's  $n$  possessions into carloads before any trips have actually been made.

- (a) The princess's charming husband, Craig, has asserted, "The first-fit algorithm will always minimize the total number of car trips needed." Give a counterexample to Craig's claim.

**Answer:**

Let  $n = 4$  and let  $x_1 = 0.3$ ,  $x_2 = 0.8$ ,  $x_3 = 0.2$ , and  $x_4 = 0.7$ . The optimal number of car trips needed is 2, while the first-fit algorithm produces 3 car trips.

- (b) Prove that the first-fit algorithm has a ratio bound of 2. (*Hint:* How many carloads can be less than half full?)

**Answer:**

Consider a carload  $u$  of size  $p$  which is less than half full, i.e.,  $p < 0.5$ . Let  $x_i$  be the first item to go into the next carload  $v$ . Then, due to the nature of the first-fit algorithm, it follows that  $p + x_i > 1$  and  $x_i > 0.5$ . Hence, carload  $v$  is more than half full. For calculation purposes, we can transfer a portion of the load of item  $x_i$  from carload  $v$  to carload  $u$  to make carload  $u$  exactly half full while keeping carload  $v$  at least half full. This is because  $0.5 - p < x_i - 0.5$ . After we perform this operation, every carload is at least half full. Thus, if  $h$  is the number of carloads produced by the first-fit algorithm, then  $\sum_{i=1}^n x_i \geq 0.5h$ . Note that the optimal number of carloads  $OPT$  is at least  $\sum_{i=1}^n x_i$ . Hence,  $h/OPT \leq 2$ , which proves that the first-fit algorithm has a ratio bound of 2.

## Practice Final Exam

- Do not open this quiz booklet until you are directed to do so. Read all the instructions first.
- The quiz contains 6 problems, several with multiple parts. You have 180 minutes to earn 120 points.
- This quiz booklet contains 15 pages, including this one, and a sheet of scratch paper which can be detached.
- This quiz is closed book. You may use two double sided Letter ( $8\frac{1}{2}'' \times 11''$ ) or A4 crib sheets. No calculators or programmable devices are permitted. Cell phones must be put away.
- Write your solutions in the space provided. If you run out of space, continue your answer on the back of the same sheet and make a notation on the front of the sheet.
- Do not waste time deriving facts that we have studied. It is sufficient to cite known results.
- Do not spend too much time on any one problem. Generally, a problem's point value is an indication of how many minutes to spend on it.
- 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. Please be neat.
- Good luck!

Problem	Title	Points	Parts	Grade	Initials
0	Name	1	15		
1	True or False	44	11		
2	P, NP & Friends	10	1		
3	Taming MAX-CUT	10	3		
4	Spy Games	10	2		
5	Lots of Spanning trees	25	5		
6	Traveling with the salesman	20	3		
Total		120			

Name: \_\_\_\_\_

**Problem 0. Name.** [1 point] Write your name on every page of this exam booklet! Don't forget the cover.

**Problem 1. True or False.** [44 points] (11 parts)

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

- (a) **T F** [4 points] If problem  $A$  can be reduced to 3SAT via a deterministic polynomial-time reduction, and  $A \in \text{NP}$ , then  $A$  is NP-complete.

**Solution:** False. We need to reduce in the other direction (reduce an NP-hard problem to  $A$ ).

- (b) **T F** [4 points] Let  $G = (V, E)$  be a flow network, i.e., a weighted directed graph with a distinguished source vertex  $s$ , a sink vertex  $t$ , and non-negative capacity  $c(u, v)$  for every edge  $(u, v)$  in  $E$ . Suppose you find an  $s$ - $t$  cut  $C$  which has edges  $e_1, e_2, \dots, e_k$  and a capacity  $f$ . Suppose the value of the maximum  $s$ - $t$  flow in  $G$  is  $f$ .

Now let  $H$  be the flow network obtained by adding 1 to the capacity of each edge in  $C$ . Then the value of the maximum  $s$ - $t$  flow in  $H$  is  $f + k$ .

**Solution:** False. There could be multiple min-cuts. Consider the graph  $s$ - $v$ - $t$  where the edges have capacity 1; either edge in itself is a min-cut, but adding capacity to that edge alone does not increase the max flow.

- (c) **T F** [4 points] Let  $A$  and  $B$  be optimization problems where it is known that  $A$  reduces to  $B$  in polynomial time. Additionally, it is known that there exists a polynomial-time 2-approximation for  $B$ . Then there must exist a polynomial-time 2-approximation for  $A$ .

**Solution:** False. approximation factor is not (necessarily) carried over in poly-time reduction. See e.g. set cover vs. vertex cover.

- (d) **T F** [4 points] There exists a polynomial-time 2-approximation algorithm for the general Traveling Salesman Problem.

**Solution:** False, assuming  $P \neq NP$ . There is an approximation algorithm in the special case where the graph obeys the triangle inequality, but we don't know of one in general.

- (e) **T F** [4 points] If we use a max-queue instead of a min-queue in Kruskal's MST algorithm, it will return the spanning tree of maximum total cost (instead of returning the spanning tree of minimum total cost). (Assume the input is a weighted connected undirected graph.)

**Solution:** True. The proof is essentially the same as for the usual Kruskal's algorithm. Alternatively, this is equivalent to negating all the edge weights and running Kruskal's algorithm.

- (f) **T F** [4 points] A randomized algorithm for a decision problem with one-sided-error and correctness probability  $1/3$  (that is, if the answer is YES, it will always output YES, while if the answer is NO, it will output NO with probability  $1/3$ ) can always be amplified to a correctness probability of 99%.

**Solution:** True. Since the error is one-sided, it in fact suffices for the correctness probability to be any constant  $> 0$ . We can then repeat it, say,  $k$  times, and output NO if we ever see a NO, and YES otherwise. Then, if the correct answer is YES, all  $k$  repetitions of our algorithm will output YES, so our final answer is also YES, and if the correct answer is NO, each of our  $k$  repetitions has a  $1/3$  chance of returning NO, in which case our final answer is, correctly, NO, with probability  $1 - (2/3)^k$ , so  $k = \log_{3/2} 100$  repetitions suffice.

- (g) **T F** [4 points] Suppose that a randomized algorithm  $A$  has expected running time  $\Theta(n^2)$  on any input of size  $n$ . Then it is possible for some execution of  $A$  to take  $\Omega(3^n)$  time.

**Solution:** True. Imagine a scenario where the runtime of  $A$  is  $\Theta(3^n)$  with probability  $n^2 3^{-n}$ , and  $\Theta(n^2)$  otherwise. It is apparent that the expected run time of  $A$  is  $\Theta(n^2)$ . But, with non-zero probability some execution may take  $\Omega(3^n)$  time .

- (h) **T F** [4 points] Building a heap on  $n$  elements takes  $\Theta(n \lg n)$  time.

**Solution:** False. It can be done in  $O(n)$  time.

- (i) **T F** [4 points] We can evaluate a polynomial of degree-bound  $n$  at any set of  $n$  points in  $O(n \lg n)$  time.

**Solution:** False. Evaluating a polynomial of degree-bound  $n$  at any (arbitrary) set of  $n$  points takes  $O(n \lg^2 n)$  time. For details, one may refer to problem 2 of pset 8.

- (j) **T F** [4 points]  $P \subseteq \text{co-NP}$ .

**Solution:** True. By definition,  $P \subseteq \text{NP}$ . So,  $\text{co-P} \subseteq \text{co-NP}$ . Recall that  $P = \text{co-P}$ . This follows from the fact that for any problem in  $P$ , one may efficiently determine if it has a solution or not. Hence,  $P \subseteq \text{co-NP}$ .

(k) **T F** [4 points] Suppose that you have two deterministic online algorithms,  $A_1$  and  $A_2$ , with a competitive ratios  $c_1$  and  $c_2$  respectively. Consider the randomized algorithm  $A^*$  that flips a fair coin once at the beginning; if the coin comes up heads, it runs  $A_1$  from then on; if the coin comes up tails, it runs  $A_2$  from then on. Then the expected competitive ratio of  $A^*$  is at least  $\min\{c_1, c_2\}$ .

**Solution:** False. Suppose the problem is to guess which of two cups holds the bean. Algorithm  $A_1$  checks left cup then right cup and has competitive ratio  $c_1 = 2$ . Algorithm  $A_2$  checks right cup then left cup and has competitive ratio  $c_2 = 2$ . The randomized algorithm has competitive ratio  $c = 0.5 \cdot 2 + 0.5 \cdot 1 = 1.5 < \min\{c_1, c_2\}$

**Problem 2. P, NP and Friends.** [10 points] Prove that if  $NP \neq co - NP$ , then  $P \neq NP$ .

**Solution:** We prove this by establishing the contrapositive, which is, if  $P = NP$ , then  $NP = co - NP$ . Observe that  $P = NP \implies co - P = co - NP$ . Now, recall that  $P$  is closed under complement, so  $P = co - P$ . Thus,  $P = co - NP$ . And from the hypothesis,  $P = NP$ . Hence,  $NP = co - NP$ . This completes the proof.

**Problem 3. Taming Max-Cut** [10 points] A CUT, in a graph  $G = (V, E)$ , is a partition of  $V$  into two non-intersecting sets  $A, B$ . An edge is said to be in the cut if one of its end points is in  $A$  and the other is in  $B$ . In the MAX-CUT problem, the objective is to maximize the number of edges in the cut. We intend to design an approximation scheme for MAX-CUT. Consider the following scheme. Every vertex  $v \in V$  is assigned to  $A, B$  uniformly at random.

- (a) What is the probability that  $e \in E$  is in the cut?

**Solution:** With probability  $1/2$ , one vertex of  $e$  is assigned to a set different from the other. Thus, the probability that  $e$  is in the cut is  $1/2$ .

- (b) What is the expected number of edges in the cut?

**Solution:** Use an indicator variable for every edge with probability of success (being in the cut) being  $1/2$ . Since, the variables are identical and independent, each experiment is a Bernoulli experiment. The expected number of successes among the  $|E|$  Bernoulli trials is  $|E|/2$ . Thus, the expected number of edges in the cut is  $|E|/2$ .

- (c) Conclude that the randomized scheme presented above is a 2-approximation to the MAX-CUT.

**Solution:** The maximum number of edges in the cut is  $|E|$ . The randomized scheme has no more than 2 times worse than the optimum in expectation.

**Problem 4. Spy Games [10 points]**

You are an enemy agent inside the borders of the country of Elbonia.

Your assignment is to make it as difficult as possible for the Elbonian rulers to send couriers on horseback from their capital city of *Anar* to its distant outpost of *Chy* along the country's network of roads.

Each road  $r$  in Elbonia connects two cities, and has an associated time  $t_r$  (in minutes) needed to travel it by horseback. A courier will always take the fastest route from *Anar* to *Chy*. You have a budget of  $B$  dollars to hire workers who can covertly degrade the quality of the roads at night (by digging potholes, etc.). Spending  $y_r$  dollars on any road  $r$  will increase the time needed to travel it by  $y_r$  minutes. Given your complete knowledge of Elbonian roads (including their travel times) and your budget  $B$ , how can you efficiently calculate a way to allocate your money so that the fastest route from *Anar* to *Chy* will take as much time as possible?

- (a) Formulate the problem as a linear program.

**Solution:** We represent the map of Elbonian cities and roads as an undirected graph  $G = (V, E)$ , where vertices correspond to cities, and an edge  $r = (u, v)$  corresponds to a road connecting cities  $u$  and  $v$ . The weight of each edge  $r$  will be  $t_r + y_r$  (the original travel time plus the amount by which we degraded the quality of the road). We will assume that travel times and dollar amounts are real, non-negative numbers. Our goal is to find a way to allocate our money (i.e., assign a  $y_r$  to each road  $r$ ) so that the length of the shortest path from *Anar* to *Chy* is maximized, subject to our budget constraint.

We can formulate this problem as the following linear program, similar to the linear program for single-pair shortest paths given in CLRS Section 29.2.

$$\begin{array}{ll} \text{maximize} & d_{Chy} \\ \text{subject to} & \\ & d_v \leq d_u + t_r + y_r \quad \text{for each edge } r = (u, v) \in E \\ & d_u \leq d_v + t_r + y_r \quad \text{for each edge } r = (u, v) \in E \\ & d_{Anar} = 0 \\ & y_r \geq 0 \quad \text{for each edge } r \in E \\ & \sum_{r \in E} y_r \leq B \end{array}$$

- (b) Prove that your linear program is correct. That is, it essentially maximizes the time along the fastest route.

**Solution: Correctness:** For any particular choice of  $y_r$  values, the length of the shortest path from *Anar* to *Chy* is the maximum value of  $d_{Chy}$  subject to the first three constraints above (this is what the linear program for single-pair shortest paths computes). Our linear program maximizes the length of the shortest path from *Anar* to *Chy* over all possible choices of the  $y_r$ 's, subject to the constraints that the  $y_r$ 's are non-negative and add up to at most our budget  $B$ .

**Problem 5. Lots of Spanning Trees.** (5 parts) [25 points] Let  $G = (V, E)$  be a connected undirected graph with edge-weight function  $w : E \rightarrow \mathbb{R}$ . Let  $w_{\min}$  and  $w_{\max}$  denote the minimum and maximum weights, respectively, of the edges in the graph. Do not assume that the edge weights in  $G$  are distinct or nonnegative. The following statements may or may not be correct. In each case, either prove the statement is correct or give a counterexample if it is incorrect.

- (a) If the graph  $G$  has more than  $|V| - 1$  edges and there is a unique edge having the largest weight  $w_{\max}$ , then this edge cannot be part of any minimum spanning tree.

**Solution:** False. This heavy edge could be the only edge connecting some vertex to a graph and thus must be included in the MST.

- (b) Any edge  $e$  with weight  $w_{min}$ , must be part of some MST.

**Solution:** True. In some ordering of the edges by increasing weight,  $e$  will be the first, thus included in the tree constructed by the Kruskal's algorithm. Any tree constructed by the Kruskal's algorithm is an MST.

- (c) If  $G$  has a cycle and there is unique edge  $e$  which has the minimum weight on this cycle, then  $e$  must be part of every MST.

**Solution:** False. Consider the graph of a tetrahedron where the three edges of the base triangle have weights 2, 3, 4 and the three edges connecting the peak to the base all have weight 1. Then the MST is composed of those latter three edges and does not include the edge with weight 2 in the bottom cycle.

- (d) If the edge  $e$  is not part of any MST of  $G$ , then it must be the maximum weight edge on some cycle in  $G$ .

**Solution:** True. Take any MST  $T$ . Since  $e$  is not part of it,  $e$  must complete some cycle in  $T$ .  $e$  must be the heaviest edge on that cycle. Otherwise a smaller weight tree could be constructed by swapping the heavier edge on the cycle with  $e$  and thus  $T$  cannot be MST.

- (e) Suppose the edge weights are nonnegative. Then the shortest path between two vertices must be part of some MST.

**Solution:** False. Consider a simple triangle with sides 1, 1, 1.5. The MST is composed of the two lighter edges while the longer edge is the shortest path between two vertices.

**Problem 6. Traveling with the salesman.** [20 points] In the **traveling-salesman problem**, a salesman must visit  $n$  cities. Modeling the problem as a complete graph on  $n$  vertices, we can say that the salesman wishes to make a **tour** or a hamiltonian cycle, visiting each city exactly once and finishing at the city he starts from. The salesman incurs a nonnegative integer cost  $c(i, j)$  to travel from city  $i$  to city  $j$ , and the salesman wishes to make a tour whose total cost is minimum, where the total cost is the sum of the individual costs along the edges of the tour.

- (a) Formulate the traveling salesman problem as a language.

$$\text{TSP} =$$

**Solution:**

$$\text{TSP} = \left\{ \langle n, c, k \rangle \mid \begin{array}{l} \text{A complete graph of } n \text{ vertices with non-negative edge} \\ \text{weights } c \text{ has a hamiltonian cycle of cost at most } k. \end{array} \right\}$$

- (b) Prove that  $\text{TSP} \in \text{NP}$ .

**Solution:** An instance  $\langle n, c, k \rangle$  can be verified to be in TSP given a tour (a permutation of indices) as the certificate. The tour can be easily checked in poly-time to traverse every node exactly once and have a satisfying total cost.

A **hamiltonian cycle** in a graph is a cycle that visits every vertex exactly once. Define the language  $\text{HAM-CYCLE} = \{\langle G \rangle : \text{there is a hamiltonian cycle in } G\}$ .

- (c) Assuming that HAM-CYCLE is complete for the class NP, prove that TSP is NP-Complete.

**Solution:** To prove that TSP is NP-Hard, we show the reduction  $\text{HAM-CYCLE} \leq_p \text{TSP}$ . Given a graph  $G = (V, E)$  we define edge costs (in the complete graph)  $c(u, v) = 0$  if  $(u, v) \in E$  and  $c(u, v) = 1$  otherwise. If  $G$  has a hamiltonian cycle, then that cycle is also a tour of cost 0 in the complete graph with costs  $c$ , so  $\langle n, c, 0 \rangle \in \text{TSP}$ . For the converse, if there is a tour in the complete graph with cost 0, then it must traverse edges that are present in  $G$ , so this tour is also a hamiltonian cycle in  $G$ .

TSP is NP-Complete because it is both in NP and NP-Hard.

## SCRATCH PAPER

## Practice Final Exam

- Do not open this quiz booklet until you are directed to do so. Read all the instructions first.
- The quiz contains 6 problems, several with multiple parts. You have 180 minutes to earn 120 points.
- This quiz booklet contains 15 pages, including this one, and a sheet of scratch paper which can be detached.
- This quiz is closed book. You may use two double sided Letter ( $8\frac{1}{2}'' \times 11''$ ) or A4 crib sheets. No calculators or programmable devices are permitted. Cell phones must be put away.
- Write your solutions in the space provided. If you run out of space, continue your answer on the back of the same sheet and make a notation on the front of the sheet.
- Do not waste time deriving facts that we have studied. It is sufficient to cite known results.
- Do not spend too much time on any one problem. Generally, a problem's point value is an indication of how many minutes to spend on it.
- 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. Please be neat.
- Good luck!

Problem	Title	Points	Parts	Grade	Initials
0	Name	1	15		
1	True or False	44	11		
2	P, NP & Friends	10	1		
3	Taming MAX-CUT	10	3		
4	Spy Games	10	2		
5	Lots of Spanning trees	25	5		
6	Traveling with the salesman	20	3		
Total		120			

Name: \_\_\_\_\_

**Problem 0. Name.** [1 point] Write your name on every page of this exam booklet! Don't forget the cover.

**Problem 1. True or False.** [44 points] (11 parts)

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

- (a) **T F** [4 points] If problem  $A$  can be reduced to 3SAT via a deterministic polynomial-time reduction, and  $A \in \text{NP}$ , then  $A$  is NP-complete.

**Solution:** False. We need to reduce in the other direction (reduce an NP-hard problem to  $A$ ).

- (b) **T F** [4 points] Let  $G = (V, E)$  be a flow network, i.e., a weighted directed graph with a distinguished source vertex  $s$ , a sink vertex  $t$ , and non-negative capacity  $c(u, v)$  for every edge  $(u, v)$  in  $E$ . Suppose you find an  $s$ - $t$  cut  $C$  which has edges  $e_1, e_2, \dots, e_k$  and a capacity  $f$ . Suppose the value of the maximum  $s$ - $t$  flow in  $G$  is  $f$ .

Now let  $H$  be the flow network obtained by adding 1 to the capacity of each edge in  $C$ . Then the value of the maximum  $s$ - $t$  flow in  $H$  is  $f + k$ .

**Solution:** False. There could be multiple min-cuts. Consider the graph  $s$ - $v$ - $t$  where the edges have capacity 1; either edge in itself is a min-cut, but adding capacity to that edge alone does not increase the max flow.

- (c) **T F** [4 points] Let  $A$  and  $B$  be optimization problems where it is known that  $A$  reduces to  $B$  in polynomial time. Additionally, it is known that there exists a polynomial-time 2-approximation for  $B$ . Then there must exist a polynomial-time 2-approximation for  $A$ .

**Solution:** False. approximation factor is not (necessarily) carried over in poly-time reduction. See e.g. set cover vs. vertex cover.

- (d) **T F** [4 points] There exists a polynomial-time 2-approximation algorithm for the general Traveling Salesman Problem.

**Solution:** False, assuming  $P \neq NP$ . There is an approximation algorithm in the special case where the graph obeys the triangle inequality, but we don't know of one in general.

- (e) **T F** [4 points] If we use a max-queue instead of a min-queue in Kruskal's MST algorithm, it will return the spanning tree of maximum total cost (instead of returning the spanning tree of minimum total cost). (Assume the input is a weighted connected undirected graph.)

**Solution:** True. The proof is essentially the same as for the usual Kruskal's algorithm. Alternatively, this is equivalent to negating all the edge weights and running Kruskal's algorithm.

- (f) **T F** [4 points] A randomized algorithm for a decision problem with one-sided-error and correctness probability  $1/3$  (that is, if the answer is YES, it will always output YES, while if the answer is NO, it will output NO with probability  $1/3$ ) can always be amplified to a correctness probability of 99%.

**Solution:** True. Since the error is one-sided, it in fact suffices for the correctness probability to be any constant  $> 0$ . We can then repeat it, say,  $k$  times, and output NO if we ever see a NO, and YES otherwise. Then, if the correct answer is YES, all  $k$  repetitions of our algorithm will output YES, so our final answer is also YES, and if the correct answer is NO, each of our  $k$  repetitions has a  $1/3$  chance of returning NO, in which case our final answer is, correctly, NO, with probability  $1 - (2/3)^k$ , so  $k = \log_{3/2} 100$  repetitions suffice.

- (g) **T F** [4 points] Suppose that a randomized algorithm  $A$  has expected running time  $\Theta(n^2)$  on any input of size  $n$ . Then it is possible for some execution of  $A$  to take  $\Omega(3^n)$  time.

**Solution:** True. Imagine a scenario where the runtime of  $A$  is  $\Theta(3^n)$  with probability  $n^2 3^{-n}$ , and  $\Theta(n^2)$  otherwise. It is apparent that the expected run time of  $A$  is  $\Theta(n^2)$ . But, with non-zero probability some execution may take  $\Omega(3^n)$  time .

- (h) **T F** [4 points] Building a heap on  $n$  elements takes  $\Theta(n \lg n)$  time.

**Solution:** False. It can be done in  $O(n)$  time.

- (i) **T F** [4 points] We can evaluate a polynomial of degree-bound  $n$  at any set of  $n$  points in  $O(n \lg n)$  time.

**Solution:** False. Evaluating a polynomial of degree-bound  $n$  at any (arbitrary) set of  $n$  points takes  $O(n \lg^2 n)$  time. For details, one may refer to problem 2 of pset 8.

- (j) **T F** [4 points] Suppose that you have two deterministic online algorithms,  $A_1$  and  $A_2$ , with a competitive ratios  $c_1$  and  $c_2$  respectively. Consider the randomized algorithm  $A^*$  that flips a fair coin once at the beginning; if the coin comes up heads, it runs  $A_1$  from then on; if the coin comes up tails, it runs  $A_2$  from then on. Then the expected competitive ratio of  $A^*$  is at least  $\min\{c_1, c_2\}$ .

**Solution:** False. Suppose the problem is to guess which of two cups holds the bean. Algorithm  $A_1$  checks left cup then right cup and has competitive ratio  $c_1 = 2$ . Algorithm  $A_2$  checks right cup then left cup and has competitive ratio  $c_2 = 2$ . The randomized algorithm has competitive ratio  $c = 0.5 \cdot 2 + 0.5 \cdot 1 = 1.5 < \min\{c_1, c_2\}$

**Problem 2. Taming Max-Cut** [10 points] A CUT, in a graph  $G = (V, E)$ , is a partition of  $V$  into two non-intersecting sets  $A, B$ . An edge is said to be in the cut if one of its end points is in  $A$  and the other is in  $B$ . In the MAX-CUT problem, the objective is to maximize the number of edges in the cut. We intend to design an approximation scheme for MAX-CUT. Consider the following scheme. Every vertex  $v \in V$  is assigned to  $A, B$  uniformly at random.

- (a) What is the probability that  $e \in E$  is in the cut?

**Solution:** With probability  $1/2$ , one vertex of  $e$  is assigned to a set different from the other. Thus, the probability that  $e$  is in the cut is  $1/2$ .

- (b) What is the expected number of edges in the cut?

**Solution:** Use an indicator variable for every edge with probability of success (being in the cut) being  $1/2$ . Since, the variables are identical and independent, each experiment is a Bernoulli experiment. The expected number of successes among the  $|E|$  Bernoulli trials is  $|E|/2$ . Thus, the expected number of edges in the cut is  $|E|/2$ .

- (c) Conclude that the randomized scheme presented above is a 2-approximation to the MAX-CUT.

**Solution:** The maximum number of edges in the cut is  $|E|$ . The randomized scheme has no more than 2 times worse than the optimum in expectation.

**Problem 3. Lots of Spanning Trees.** (5 parts) [25 points] Let  $G = (V, E)$  be a connected undirected graph with edge-weight function  $w : E \rightarrow \mathbb{R}$ . Let  $w_{\min}$  and  $w_{\max}$  denote the minimum and maximum weights, respectively, of the edges in the graph. Do not assume that the edge weights in  $G$  are distinct or nonnegative. The following statements may or may not be correct. In each case, either prove the statement is correct or give a counterexample if it is incorrect.

- (a) If the graph  $G$  has more than  $|V| - 1$  edges and there is a unique edge having the largest weight  $w_{\max}$ , then this edge cannot be part of any minimum spanning tree.

**Solution:** False. This heavy edge could be the only edge connecting some vertex to a graph and thus must be included in the MST.

- (b) Any edge  $e$  with weight  $w_{min}$ , must be part of some MST.

**Solution:** True. In some ordering of the edges by increasing weight,  $e$  will be the first, thus included in the tree constructed by the Kruskal's algorithm. Any tree constructed by the Kruskal's algorithm is an MST.

- (c) If  $G$  has a cycle and there is unique edge  $e$  which has the minimum weight on this cycle, then  $e$  must be part of every MST.

**Solution:** False. Consider the graph of a tetrahedron where the three edges of the base triangle have weights 2, 3, 4 and the three edges connecting the peak to the base all have weight 1. Then the MST is composed of those latter three edges and does not include the edge with weight 2 in the bottom cycle.

- (d) If the edge  $e$  is not part of any MST of  $G$ , then it must be the maximum weight edge on some cycle in  $G$ .

**Solution:** True. Take any MST  $T$ . Since  $e$  is not part of it,  $e$  must complete some cycle in  $T$ .  $e$  must be the heaviest edge on that cycle. Otherwise a smaller weight tree could be constructed by swapping the heavier edge on the cycle with  $e$  and thus  $T$  cannot be MST.

- (e) Suppose the edge weights are nonnegative. Then the shortest path between two vertices must be part of some MST.

**Solution:** False. Consider a simple triangle with sides 1, 1, 1.5. The MST is composed of the two lighter edges while the longer edge is the shortest path between two vertices.

**Problem 4. Traveling with the salesman.** [20 points] In the **traveling-salesman problem**, a salesman must visit  $n$  cities. Modeling the problem as a complete graph on  $n$  vertices, we can say that the salesman wishes to make a **tour** or a hamiltonian cycle, visiting each city exactly once and finishing at the city he starts from. The salesman incurs a nonnegative integer cost  $c(i, j)$  to travel from city  $i$  to city  $j$ , and the salesman wishes to make a tour whose total cost is minimum, where the total cost is the sum of the individual costs along the edges of the tour.

- (a) Formulate the traveling salesman problem as a language.

$$\text{TSP} =$$

**Solution:**

$$\text{TSP} = \left\{ \langle n, c, k \rangle \mid \begin{array}{l} \text{A complete graph of } n \text{ vertices with non-negative edge} \\ \text{weights } c \text{ has a hamiltonian cycle of cost at most } k. \end{array} \right\}$$

- (b) Prove that  $\text{TSP} \in \text{NP}$ .

**Solution:** An instance  $\langle n, c, k \rangle$  can be verified to be in TSP given a tour (a permutation of indices) as the certificate. The tour can be easily checked in poly-time to traverse every node exactly once and have a satisfying total cost.

A **hamiltonian cycle** in a graph is a cycle that visits every vertex exactly once. Define the language HAM-CYCLE = { $\langle G \rangle$ : there is a hamiltonian cycle in  $G$ }.

- (c) Assuming that HAM-CYCLE is complete for the class NP, prove that TSP is NP-Complete.

**Solution:** To prove that TSP is NP-Hard, we show the reduction HAM-CYCLE  $\leq_p$  TSP. Given a graph  $G = (V, E)$  we define edge costs (in the complete graph)  $c(u, v) = 0$  if  $(u, v) \in E$  and  $c(u, v) = 1$  otherwise. If  $G$  has a hamiltonian cycle, then that cycle is also a tour of cost 0 in the complete graph with costs  $c$ , so  $\langle n, c, 0 \rangle \in$  TSP. For the converse, if there is a tour in the complete graph with cost 0, then it must traverse edges that are present in  $G$ , so this tour is also a hamiltonian cycle in  $G$ .

TSP is NP-Complete because it is both in NP and NP-Hard.

## SCRATCH PAPER

### Practice Problem 1. Coat Check

Suppose that  $n$  women check their coats at a concert. However, at the end of the night, the attendant has lost the claim checks and doesn't know which coat belongs to whom. All of the women came dressed in black coats that were nearly identical, but of different sizes. The attendant can have a woman try a coat, and find out whether the coat fits (meaning it belongs to that woman), or the coat is too big, or the coat is too small. However, the attendant cannot compare the sizes of two coats directly, or compare the sizes of two women directly. Describe how the attendant can determine which coat belongs to each woman in expected  $O(n \log n)$  time. Give a brief analysis of the running time of your algorithm.

**Solution:** We solve this using a modified version of quicksort:

1. Choose a random woman (whom we will call the “pivot woman”) and have her try on all the coats. Separate the coats into three groups: the coats that are too small for her, the coats that are too big for her, and her own coat, which we will call the “pivot coat.”
2. Have all the remaining women try on the pivot coat. Separate them into two groups: those who are too small for the pivot coat and those who are too big for the pivot coat.
3. Recursively solve the two remaining subproblems, matching the women and coats who are smaller than the matched pair and those who are larger than the matched pair.

This requires exactly twice as many operations as a standard randomized quicksort, so this algorithm still runs in expected  $O(n \log n)$  time.

### Practice Problem 2. The Power of Quicksort

To get concrete evidence of the power of randomized quicksort, you want to create discrete probability distributions of how many comparisons are needed to perform a randomized quicksort on  $n$  elements. Specifically, if we let  $X_n$  be a random variable denoting how many comparisons are needed, you want to determine  $P(X_n = x)$  for all integers  $x \geq 0$ .

For example, the distribution for  $X_3$  is

$$X_3 = \begin{cases} 2 & \text{with probability } \frac{1}{3} \\ 3 & \text{with probability } \frac{2}{3} \end{cases}$$

because there is a  $\frac{1}{3}$  probability we will only need 2 comparisons (if the chosen pivot is the middle element) and a  $\frac{2}{3}$  probability that we will need 3 comparisons (if the chosen pivot is the smallest or largest element).

The goal for this problem is to come up with an efficient algorithm that computes the distribution for  $X_n$ .

- (a) Given two random variables  $X$  and  $Y$  whose domains are the non-negative integers such that  $0 \leq X, Y \leq m$  for a known parameter  $m$  and their probability distributions, give an efficient algorithm that computes the probability distribution for  $X + Y$ . State the running time of this algorithm in terms of  $m$ .

**Solution:** Note that

$$P(X + Y = a) = \sum_{i=-\infty}^{\infty} P(X = i) \cdot P(Y = a - i).$$

Therefore we can solve this by convolving two lists of length  $m + 1$  where the  $i$ th element in each list is the probability that the associated random variable equals  $i$ . This can be done in  $O(m \log m)$  time using an FFT.

- (b) Give a polynomial-time dynamic programming algorithm that computes the distribution for  $X_n$ .

*Hint: Start by expressing  $X_n$  in terms of  $X_0, X_1, \dots, X_{n-1}$ .*

**Solution:** Going from the hint, the recurrence is:

$$X_n = (n-1) + \begin{cases} X_0 + X_{n-1} & \text{with probability } 1/n \\ X_1 + X_{n-2} & \text{with probability } 1/n \\ X_2 + X_{n-3} & \text{with probability } 1/n \\ \dots \\ X_{n-1} + X_0 & \text{with probability } 1/n \end{cases}$$

We can compute the distributions for each of the  $n$  cases using dynamic programming using the base case that  $X_0 = 0$  with probability 1. Since randomized quicksort runs in  $\Theta(n^2)$  in the worst case, computing each distribution takes  $O(n^2 \log n^2) = O(n^2 \log n)$  time. Each subproblem therefore requires  $O(n^3 \log n)$  time.

Since there are  $\Theta(n)$  subproblems in total, the final running time is  $O(n^4 \log n)$ .

### Practice Problem 3. The Sensational 6046

The Sensational 6046 is a group of superheroes in charge of fighting off an alien invasion of Earth. Their most powerful ability involves linking their bodies together to form various weapons (including swords, loganberries, and Turing machines). In the heat of battle, they must be able to form these weapons as quickly as possible. Thanks to your expertise in algorithms, you feel that you can help them.

There are  $n$  heroes numbered 1 through  $n$ . Each hero starts at some location in 3D space and, to form a particular weapon, they need to move to new points, which are also numbered from 1 through  $n$ . However, in order to maintain the structural integrity of the weapon, a hero must arrive at point  $j$  at or before time  $t \cdot j$  for some fixed constant  $t$ . You are given an  $n \times n$  matrix  $\mathbf{T}$  where  $T_{ij}$  denotes the amount of time it takes for hero  $i$  to move to point  $j$ .

The problem can be put in mathematical terms as follows:

Let  $\sigma_1, \dots, \sigma_n$  be a permutation of  $1, \dots, n$  and  $t$  a positive constant. This permutation is said to be ***t*-valid** if  $T_{i\sigma_i} \leq t \cdot \sigma_i$  for  $i = 1, \dots, n$ .

- (a) Give an efficient algorithm that, given  $\mathbf{T}$  and  $t$ , finds a  $t$ -valid permutation if one exists, or reports that one doesn't exist.

**Solution:** This can be formulated as a matching problem where we attempt to match heroes to destination points, which can in turn be formulated as a max-flow problem. Let  $G = (V, E)$  be our flow network where  $V = \{s, t, h_1, \dots, h_n, d_1, \dots, d_n\}$ , where  $h_i$  corresponds to hero  $i$  and  $d_j$  corresponds to destination point  $j$ . Create edges from  $s$  to each  $h_i$  and from each  $d_j$  to  $t$ , all with capacity 1. For each  $i, j$ , create an edge with capacity 1 from  $h_i$  to  $d_j$  if and only if  $T_{ij} \leq t \cdot j$ . Run Ford-Fulkerson on this flow network. If the flow has size  $n$ , the  $h_i \rightarrow d_j$  edges that are at capacity correspond to a  $t$ -valid permutation  $\sigma$  with  $\sigma_i = j$ . Otherwise no  $t$ -valid permutation exists.

This algorithm runs in  $O(n^3)$  time since the maximum flow can have size at most  $n$ .

The commander is mainly interested in knowing how quickly a weapon can be formed, i.e., the smallest value  $t_0$  such that there exists a  $t_0$ -valid permutation.

- (b) We will call  $t_c$  a ***critical value*** if there exist values  $i, j$  such that  $T_{ij} = t_c \cdot j$ . Show that  $t_0$  must be a critical value.

**Solution:** We prove this by contradiction. First assume that  $t_0$  is smaller than any critical value. Then by definition we have  $t_0 \cdot j < T_{ij}$  for all  $i, j$ , which implies that there can never be a  $t_0$ -valid permutation. Contradiction.

Now assume that  $t_0$  is not a critical value. Let  $t_c^*$  be the largest critical value that is smaller than  $t_0$ . Observe that if  $T_{ij} \leq t_c^* \cdot j$ , then  $T_{ij} \leq t_0 \cdot j$  as well because otherwise there would have to be a  $t'$  in between such that  $T_{ij} = t' \cdot j$ , which contradicts the definition of  $t_c^*$ . The converse also holds by virtue of the fact that  $t_c^* < t_0$ . Therefore if  $\sigma$  is  $t_0$ -valid, then  $\sigma$  is  $t_c^*$ -valid as well. But since  $t_c^* < t_0$ , we've contradicted the definition of  $t_0$ . Therefore  $t_0$  must be a critical value.

- (c) Give an efficient algorithm that finds  $t_0$ .

**Solution:** There are at most  $n^2$  distinct critical values, one for each pair of values for  $i$  and  $j$ . Therefore we may sort the list of critical values (which takes  $n^2 \log n$  time) and perform binary search to find the smallest critical value  $t_c$  such that there exists a  $t_c$ -valid permutation. Each query takes  $O(n^3)$  time, and since there are  $O(n^2)$  elements, we only need to perform  $O(\log n^2) = O(\log n)$  queries. The total running time of this algorithm is  $O(n^3 \log n)$ .

With a bit more cleverness, you can bring this back down to  $O(n^3)$ . Instead of starting with a brand new flow every time, modify the existing flow network by removing edges if a  $t_0$ -valid permutation was found or by adding edges if one was not found. This prevents us from performing the same computations many times.

## Extra Practice Problems for Quiz 1 – Solutions

This set of problems is to help you practice for Quiz 1 and is not to be turned in.

### Problem 2-1. Lower Bounds in the Comparison Model

Do problems 8.1-1 and 8.1-3 from CLRS.

**Solution:** **Problem 8.1-1.** First to clarify, we distinguish 2 types of leaves: *reachable* and *unreachable*. Reachable are those for which there exists at least one permutation for which the algorithm reaches the leaf. Otherwise, the leaf is unreachable. For unreachable leaves, the depth could be as small as constant. But we are concerned with reachable leaves.

The answer is  $\Omega(n)$ . We will prove in fact a lower bound of  $(n - 1)/2$ . Intuitively, once we decide on an answer (a permutation), we must have “looked” at all elements, but in one comparison we can look only at 2 elements.

More formally, consider any leaf that is at depth  $d < (n - 1)/2$ . The path from the root to the leaf contains  $d$  comparisons, and thus there are at most  $2d < n - 1$  elements involved in these comparisons. Thus there are two elements that have not been involved in any comparison. Since the leaf is reachable, there is at least one permutation for which the algorithm reaches the leaf. But then there must be at least two permutations reaching to this leaf (since we can, for example, exchange the order two uncompared elements and obtain a different permutation that is consistent with all the comparisons that have been made). But this is impossible in a correct algorithm.

**Problem 8.1-3.** In all three cases the answer is still  $\Omega(n \log n)$ .

In the class, we proved that any decision tree for sorting must be able to generate all possible permutations, and thus the worst-case lower bound for sorting is at least log of the number of possible permutations, which, in general, are  $n!$ .

Consider a decision tree that works for at least a half of the permutations. Then we can ignore all the other permutations (and the corresponding part of the decision tree). Then we obtain a decision tree that has  $n!/2$  leaves. The new lower bound will be  $\log(\# \text{leaves})$ . Now, if the number of possible permutations is  $n!/2$ , then the lower bound is  $\log(n!/2) = \log(n!) - 1 = \Omega(n \log n)$ . Also, if the number of possible permutations is  $n!/n$ , then the lower bound is  $\log(n!/n) = \log(n!) - \log n = \Omega(n \log n)$ . Finally, if the number of possible permutations is  $n!/2^n$ , then the lower bound is  $\log(n!/2^n) = \log n! - n = \Omega(n \log n)$ .

### Problem 2-2. Bounded Integers

Do problem 8.2-4 from CLRS.

**Solution:** In the preprocessing stage, we compute an array  $A_0, \dots, A_k$  such that  $A_i$  is equal to the number of the integers at less than or equal to  $i$ . Once we do such a preprocessing, answering the

query is easy: to find out the number of integers in the range  $[a, b]$ , compute  $A_b - A_{a-1}$  (assume that  $A_{-1} = 0$ ).

The preprocessing is done as follows. First compute the array  $C_0, \dots, C_k$ , where  $C_i$  is the number of integers equal to  $i$  (as in counting sort). Then, scan through  $C$  from 0 to  $k$  to compute  $A_i$ :  $A_i = A_{i-1} + C_i$ .

### Problem 2-3. Stable Sorting

Do problem 8.3-2 from CLRS.

**Solution:** INSERTION-SORT is stable since the **while** condition is  $A[i] > key$ . Using  $\geq$  instead of  $>$  would result in a correct but unstable sort.

The stability of MERGE-SORT depends on how we write the procedure MERGE. The pseudo-code for this procedure is not given in the text. Remember how MERGE-SORT works: it divides the array into two sub-arrays, the left (or front) and the right (or back) halves, which it recursively sorts and then merges by comparing the head of the two (sorted) sub-arrays and putting in place the smaller one. If the two elements are equal, MERGE-SORT will be stable if the element of the left sub-array is chosen. It would be correct but unstable if the element of the right sub-array is chosen.

HEAP-SORT is not stable. An example is as follows. Imagine the array is already a heap, and there are two equal elements,  $a$  and  $b$ , but  $a$  is in the left subtree (of the root) and  $b$  is in the right subtree. Then, during the sorting, just before the moment when either of the two become the root, both elements are children of the root, and the algorithm chooses the right element,  $b$ . But it's easy to arrange that  $a$  is before  $b$  in the original array.

QUICK-SORT is not stable either (at least the versions from the book). For example, let's consider the deterministic version from Chapter 7.1 from CLRS. The problem is in the PARTITION algorithm. Specifically, the 6th step breaks the stability. For example, imagine we partition using element  $x = 3$ , and we have the array 1, 1, 1, 5, 5, 2. Then the two 5's will change their positions.

To obtain a stable sorting, take any sorting algorithm and its input array  $A$ . We will augment each array element  $A[i]$  by storing with it the value of its initial index in the array,  $i$ . Let's denote this extra field as  $Index[i]$ . Whenever the algorithm performs a comparison on two equal array elements  $A[i] = A[j]$ , we will have it compare instead the values of  $Index[i]$  and  $Index[j]$ . Therefore, if two elements have equal values, the one which started out closer to the beginning of the array will be deemed “less” than the other element. This approach requires  $\Theta(n)$  extra storage space and  $\Theta(n)$  extra running time, but since any sorting algorithm must spend  $\Omega(n)$  time looking at all of its input elements, the extra running time is absorbed and has no impact on the overall asymptotic running time of the algorithm.

### Problem 2-4. Priority Queue Implementation

In this problem, we will implement some of the other operations on heaps: INSERT, DELETE, and INCREASE. (Together with the operations from the class, they will make up a priority queue implementation.)

- (a) Implement the procedure  $\text{INSERT}(Q, x)$  that inserts an element  $x$  into the heap  $Q$ .

**Solution:** See CLRS, Section 6.5.

- (b) Implement the procedure  $\text{DELETE}(Q, i)$  that deletes the element at position  $i$  from the heap  $Q$ . (Note that the procedure takes as a parameter the pointer  $i$  to the deleted element; otherwise, it would take  $\Omega(n)$  time just to find the element in the heap.)

**Solution:** One possible solution is to set the element to  $-\infty$  and run  $\text{MAX-HEAPIFY}(A, i)$ . Then we just need to set the size of the array smaller by one (thus ignoring the  $-\infty$ ).

- (c) Implement the procedure  $\text{INCREASE}(Q, i, k)$  that increases the value of the element at position  $i$  from the heap  $Q$  to a new value,  $k$ , which is assumed to have a higher value.

**Solution:** See CLRS, Section 6.5.

---

## Practice Quiz 1

- The real Quiz 1 will be held on Thursday, October 15, in lecture.
- There will be a quiz review on Friday, October 9, during recitation.
- The quiz will be closed book. You may use one double sided Letter ( $8\frac{1}{2}'' \times 11''$ ) or A4 crib sheet. No calculators or programmable devices are permitted.
- Write your solutions in the space provided. Extra scratch paper may be provided if you need more room, although your answer should fit in the given space.
- Do not waste time re-deriving facts that we have studied. It is sufficient to cite known results.
- Do not spend too much time on any one problem. Generally, a problem's point value is an indication of how much time to spend on it.
- 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.
- Good luck!

**Problem 1. Recurrences [15 points] (4 parts)**

Solve the following recurrences by giving tight  $\Theta$ -notation bounds. You do not need to justify your answers, but any justification that you provide will help when assigning partial credit. As usual, assume that for  $n \leq 10$ ,  $T(n) = O(1)$ .

(a) [2 points]  $T(n) = 3T(n/3) + 0.5n \lg(n).$

**Solution:** Using case 2 of the Master's Method gives us  $T(n) = \Theta(n \lg^2 n).$

(b) [4 points]  $T(n) = 9T(\sqrt[3]{n}) + \Theta(\log(n)).$

**Solution:** Let  $n = 2^m$ . Then the recurrence becomes  $T(2^m) = 9T(2^{m/3}) + \Theta(m)$ . Setting  $S(m) = T(2^m)$  gives us  $S(m) = 9S(m/3) + \Theta(m)$ . Using case 1 of the Master Method gives us  $S(m) = \Theta(m^2)$  or  $T(n) = \Theta(\log^2 n)$

(c) [4 points]  $T(n) = T(2n/7) + T(5n/7) + \Theta(n).$

**Solution:** The Master Theorem doesn't apply here. Draw recursion tree. At each level, do  $\Theta(n)$  work. Number of levels is  $\log_{7/5} n = \Theta(\lg n)$ , so guess  $T(n) = \Theta(n \lg n)$  and use the substitution method to verify guess.

(d) [5 points] Define  $T(n)$  by the recursion  $T(n) = 9(T(\lfloor n/3 \rfloor) - 1) + n^3 + 2n$  for  $n \geq 1$ , with base case  $T(0) = 0$ . Prove  $T(n) \leq 3n^3/2$ .

**Solution:** Inductive hypothesis:  $T(n) \leq 3n^3/2 - n$ .

Base case:  $T(0) = 0 \leq 30^3/2 - 0$ .

Inductive step:

$$\begin{aligned} T(n) &= 9(T(\lfloor n/3 \rfloor) - 1) + n^3 + 2n \\ &\leq 9(3\lfloor n/3 \rfloor^3/2 - \lfloor n/3 \rfloor - 1) + n^3 + 2n \\ &\leq 9(3(n/3)^3/2 - n/3) + n^3 + 2n \\ &= 3n^3/2 - n. \end{aligned}$$

**Problem 2. True or False, and Justify [13 points] (6 parts)**

Circle **T** or **F** for each of the following statements, and briefly explain why. Your justification is worth more points than your true-or-false designation.

- (a) **T F** [2 points] To achieve asymptotically optimal performance, a skip list must use promotion probability  $p = 0.5$ .

**Solution:** False. Any promotion probability between 0 and 1 achieves the same asymptotic performance.

- (b) **T F** [2 points] Universal hashing requires that you know what elements you'll hash in advance.

**Solution:** False. Perfect hashing requires knowing the elements in advance. Universal hashing does not.

- (c) **T F** [2 points] In a B-tree, the maximum number of children of an internal non-root node is at most twice the minimum of number of children.

**Solution:** True. For a B-tree with parameter  $t$ , there are at least  $t$  and at most  $2t$  children.

- (d) **T F** [2 points] A rotate operation on balanced tree always increases the depth of at least one node and decreases the depth of at least one node.

**Solution:** TRUE. Every rotate operation demotes the root of a subtree and promotes a new node to that position. Promotion decreases a node's depth. See CLRS 13.2 or Lecture 7 for a description/illustration of rotation.

- (e) **T F** [3 points] In a B-tree of minimum parameter  $t$ , every node contains at least  $t - 1$  elements.

**Solution:** FALSE, Normally nodes in a B-tree of parameter  $t$  must have between  $t - 1$  and  $2t - 1$  elements, but the root node is exempted from this rule to accommodate trees with fewer than  $t - 1$  elements. Consider a B-tree of parameter 3 that contains one element: the root node contains 1 element, but here  $t - 1 = 2$ .

**Problem 3. Short Answer [13 points] (4 parts)**

Give *brief*, but complete, answers to the following questions.

- (a) [2 points] What is the expected difference between the depth of the deepest leaf and the depth of the least deep leaf in a 2-3-4 tree containing  $N$  elements?

**Solution:** Zero. All leaves are at the same level.

- (b) [3 points] Show how to find a divisor  $d$  of  $N$  such that  $d$  is not 1 or  $N$ , given  $x, y$  such that  $x^2 = y^2 \pmod{N}$  and  $x \neq y \pmod{N}, x \neq -y \pmod{N}$ .

**Solution:** Compute  $\gcd(x - y, N)$  or compute  $\gcd(x + y, N)$

- (c) [4 points] Let  $\mathcal{H}$  be a universal hash family mapping  $[1 \dots N]$  to  $[1 \dots M]$ . Let  $X_{ijh}$  be the indicator variable for a collision between  $i$  and  $j$  under the hash function  $h$ ,  $i \neq j$  and  $h \in \mathcal{H}$ . What is  $E(X_{ijh})$ , where the expectation is taken over  $i$ ,  $j$ , and  $h$ ?

**Solution:** By the definition of a universal hash family, the probability of a collision is  $1/M$ , regardless of  $i$  and  $j$ . So the expected value of the indicator variable is  $1/M$ .

- (d) [4 points] Consider a balanced binary tree of  $n$  elements in which each node has an integer value. The weight of a path is the sum of the values of the nodes visited by the path. Give an optimal algorithm that computes the maximum possible weight of a path in the binary tree, starting at the root. What is the running time of your algorithm?

**Solution:** If the tree consists of a single node (ie, we are at a leaf), then the answer is simply the weight of that node. Otherwise, we recurse using

$$\text{MAXPATH}(root) = w(root) + \max \left\{ \begin{array}{l} \text{MAXPATH}(root \rightarrow \text{left}), \\ \text{MAXPATH}(root \rightarrow \text{right}) \end{array} \right\}$$

This algorithm accesses each node exactly once, so the runtime is  $\Theta(n)$ .

**Problem 4. Slightly-Longer Short Answer [29 points] (5 parts)**

Give *brief*, but complete, answers to the following questions.

- (a) [5 points] A sequence of  $n$  operations is performed, so that the  $i^{th}$  operation costs  $\lg(i)$  if  $i$  is an exact power of 2, and 1 otherwise. That is the amortized cost per operation?

**Solution:** True. Let  $c(i)$  be the cost of the  $i^{th}$  operation

$$c(i) = \begin{cases} \lg i & \text{if } i = 2^k, k \text{ integer} \\ 1 & \text{otherwise} \end{cases}$$

For any  $n$ , the total cost of  $n$  operations is

$$\begin{aligned} \sum_{i=1}^n c(i) &= n - \lfloor \lg n \rfloor + \sum_{i=1}^{\lfloor \lg n \rfloor} i \\ &= n + \Theta(\lg^2(n)) = \Theta(n) \end{aligned}$$

Therefore, the amortized cost per operation is  $\Theta(1)$ .

- (b) [6 points] Define set  $S = \{A \in Z_N^* \mid A = x^{(N-1)/2} \pmod{N}\}$  for a prime  $N$ . Is  $S$  a sub-group of  $Z_N^*$ ? If so, what can you say about the size of set  $S$ ?

**Solution:**  $S$  is a sub-group of  $Z_N^*$ :

- identity:  $1 \in S$
- closure: given  $A = x^{(N-1)/2} \pmod{N}$  and  $B = y^{(N-1)/2} \pmod{N}$ ,  $AB = (xy)^{(N-1)/2} \pmod{N} \in S$
- inverse: given  $A = x^{(N-1)/2} \pmod{N}$ , since  $x \neq 0$  and  $N$  is a prime, it must be that  $x \in Z_N^*$ . Thus,  $x$  must have an inverse, so  $A^{-1} = (x^{-1})^{(N-1)/2} \pmod{N}$ , which is also in  $S$ .

By Lagrange's Theorem, we know that  $|S|$  divides  $|Z_N^*|$ , but in this case, we can say something stronger:  $A = x^{(N-1)/2} \pmod{N}$ , so  $A^2 = x^{(N-1)} = 1 \pmod{N}$  (by Fermat's Little Theorem), which has only two solutions (1 and  $N-1$  by Modular Sqrt Theorem), so  $|S| = 2$ .

- (c) [6 points] Given two sets  $A$  and  $B$  of  $n$  integers, give an efficient deterministic algorithm to find  $A \cap B$  and analyze its runtime. Can you do better with randomization? Explain.

**Solution:** For the deterministic algorithm, sort each list. Then, iterate through the elements looking for elements common to both arrays. This takes  $O(n \lg n)$  time. Using randomization, hash the elements of  $A$ . Then iterate through  $B$ , looking up elements in the hash of  $A$ . The expected running time is  $O(n)$ .

- (d) [6 points]

Consider an array  $A$  of  $n$  integers. Find all elements occurring at least  $n/3$  times.

**Solution:** Replace the  $i$ th element with a pair  $(A[i], i)$  to make them all distinct. Comparison between pairs is done by comparing the first elements and breaking ties by the second elements.

Use the select algorithm to find the elements of ranks  $n/3$ ,  $2n/3$  and  $n$ . If an element occurs at least  $n/3$  times, it must be one of those three elements. Check all three to see if any of them occurs at least  $n/3$  times. The running time is  $O(n)$ .

(e) [6 points]

Consider a sorted array  $A$  of size  $n$ , containing distinct integers. Give an  $O(\lg n)$  algorithm to find an index  $i$  such that  $A[i] = i$  (or *none*, if no such index exists). Does your algorithm still work if  $A$  contains repeat elements? Explain why or why not.

**Solution:** Consider  $A[n/2]$ . If  $A[n/2] = n/2$ , then we're done. Otherwise, if  $A[n/2] > n/2$ , recurse on  $A[1 \dots n/2 - 1]$ . If  $A[n/2] < n/2$ , recurse on  $A[n/2 + 1 \dots n]$ . The runtime is  $O(\lg n)$ .

If there are repeat elements, then we can no longer ensure that the answer is on one side of the median, since it may be true that  $A[1] = 1$  and  $A[n] = n$ , for any value of the median between 2 and  $n - 1$ .

**Problem 5. Searching in multiple lists [8 points]**

Consider two disjoint sorted arrays  $A[1 \dots m]$  and  $B[1 \dots n]$ . Find an  $O(\log k)$  time algorithm for computing the  $k$ -th smallest element in the union of the two arrays.

**Solution:** Consider  $A[k/2]$  and  $B[k/2]$ . Without loss of generality, assume  $A[k/2] < B[k/2]$ . Then  $A[k/2]$  is greater than at most  $k$  elements. Furthermore the elements  $A[1 \dots k/2 - 1]$  are all less than the  $k$ -th element, so we can eliminate them. Similarly,  $B[k/2]$  is greater than at least  $k$  elements, so the elements  $B[k/2 + 1 \dots n]$  are all larger than the  $k$ -th element. We can therefore eliminate them too. We are therefore left with two subarrays, and we now want to find the  $k/2$ -th element (since we eliminated  $k/2$  elements that were guaranteed to be less than the  $k$ -th element). This divide-and-conquer algorithm follows the recursion  $T(k) = T(k/2) + 1$ , which is  $O(\log k)$ .

**Problem 6. The Eccentric Landlord** [8 points] (2 parts)

Your construction firm is hired to build an apartment building for an eccentric landlord. He wants his building to be a square of size  $M \times M$ , containing  $M^2$  identical square apartments.

The landlord will add one tenant a day. When he can't fit a new tenant, he will tear down two sides of the building and have new walls built, expanding it an  $(M + 1) \times (M + 1)$  building.

It costs your firm \$1 to build one apartment's exterior wall; your other costs (demolishing exterior walls, building interior walls, etc) are negligible. Your costs will be:

**Day 1:** \$4 (build four walls)

**Day 2:** \$6 (expand to 2x2)

**Day 3:** \$0 (tenant moves into empty unit)

**Day 4:** \$0 (tenant moves into empty unit)

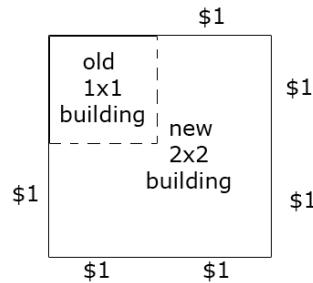
**Day 5:** \$8 (expand to 3x3)

**Day 6:** \$0 (tenant moves into empty unit)

:

The costs incurred on day 2 are shown below.

Day 2: new walls cost \$6



- (a)** [4 points] What will your asymptotic aggregate cost be for this project? Give your answer as a function of the number of days elapsed.

**Solution:** We build between 1 and 2 new walls for each tenant, and tenants arrive at a rate of 1/day, so the cost per day is  $O(1)$ . The aggregate cost is therefore  $O(n)$ .

- (b) [4 points] You convince the landlord to expand his building in bigger steps: Whenever he can't fit a new tenant, he will double the building side length (instead of increasing it by one unit). Repeat your analysis from part (a) for this new condition.

**Solution:** We must spend  $O(\sqrt{n})$  dollars to fit  $O(n)$  tenants, since the cost is dominated by the most current addition.

**Problem 7. Chemical testing [15 points]**

A chemistry lab is given  $n$  samples, with the goal of determining which of the samples contain traces of a foreign substance. It is assumed that only few (say, at most  $t$ ) samples test positive. The tests are very sensitive, and can detect even the slightest trace of the substance in a sample. However, each test is very expensive. Because of that, the lab decided to test "sample pools" instead. Each pool contains a mixture of some of the samples (each sample can participate in several pools). A test of a pool returns positive if any of the samples contributing to the pool contains a trace of the substance.

Design a testing method that correctly determines the positive samples using only  $O(t \log n)$  tests. The method can be **adaptive**, i.e., the choice of the next test can depend on the outcomes of the previous tests.

**Solution:** There exist several related algorithms that solve this problem. The simplest one proceeds as follows: we divide the samples into  $2t$  groups of size  $\frac{n}{2t}$  each. We pool and test each group. Since at most  $t$  groups are positive, we can label at least  $n/2$  samples as negative. Then we recurse on the remaining  $n/2$  samples. It is easy to see that the number of recursion levels is  $O(\log n)$ . Since  $2t$  tests are performed at each level, the total number of tests is at most  $O(t \log n)$ .

A different algorithm divide the samples into two groups of size  $n/2$ . Both groups are tested, and the algorithm recurses on group(s) that test positive. As before, the recursion tree has depth  $\log n$ , since we divide the group size by 2 at each level. Moreover, the recursion tree contains at most  $t$  leaves. Therefore the total number of tree nodes (and therefore tests) is  $O(t \log n)$ .

## Practice Quiz 1 Solutions

### Problem -1. Recurrences

Solve the following recurrences by giving tight  $\Theta$ -notation bounds. You do not need to justify your answers, but any justification that you provide will help when assigning partial credit.

(a)  $T(n) = T(n/3) + T(n/6) + \Theta(n^{\sqrt{\lg n}})$

**Solution:** Master method does not apply directly, but we have  $T(n) \leq S(n) = 2T(n/3) + \Theta(n^{\sqrt{\lg n}})$ . Now apply case 3 of master method to get  $T(n) \leq S(n) = \Theta(n^{\sqrt{\lg n}})$ . Therefore, we have  $T(n) = O(n^{\sqrt{\lg n}})$ . Lower bound is obvious.

(b)  $T(n) = T(n/2) + T(\sqrt{n}) + n$

**Solution:** Master method does not apply directly. But  $\sqrt{n}$  is much smaller than  $n/2$ , therefore ignore the lower order term and guess that the answer is  $T(n) = \Theta(n)$ . Check by substitution.

(c)  $T(n) = 3T(n/5) + \lg^2 n$

**Solution:** By Case 1 of the Master Method, we have  $T(n) = \Theta(n^{\log_5(3)})$ .

(d)  $T(n) = 2T(n/3) + n \lg n$

**Solution:** By Case 3 of the Master Method, we have  $T(n) = \Theta(n \lg n)$ .

(e)  $T(n) = T(n/5) + \lg^2 n$

**Solution:** By Case 2 of the Master Method, we have  $T(n) = \Theta(\lg^3 n)$ .

(f)  $T(n) = 8T(n/2) + n^3$

**Solution:** By Case 2 of the Master Method, we have  $T(n) = \Theta(n^3 \log n)$ .

(g)  $T(n) = 7T(n/2) + n^3$

**Solution:** By Case 3 of the Master Method, we have  $T(n) = \Theta(n^3)$ .

(h)  $T(n) = T(n - 2) + \lg n$

**Solution:**  $T(n) = \Theta(n \log n)$ . This is  $\sum_{i=1}^{n/2} \lg 2i \geq \sum_{i=1}^{n/2} \lg i \geq (n/4)(\lg n/4) = \Omega(n \lg n)$ . For the upper bound, note that  $T(n) \leq S(n)$ , where  $S(n) = S(n-1) + \lg n$ , which is clearly  $O(n \lg n)$ .

### Problem -2. True or False

Circle **T** or **F** for each of the following statements, and briefly explain why. The better your argument, the higher your grade, but be brief. No points will be given even for a correct solution if no justification is presented.

(a) **T F** For all asymptotically positive  $f(n)$ ,  $f(n) + o(f(n)) = \Theta(f(n))$ .

**Solution: True.** Clearly,  $f(n) + o(f(n))$  is  $\Omega(f(n))$ . Let  $g(n) \in o(f(n))$ . For any  $c > 0$ ,  $g(n) \leq c(f(n))$  for all  $n \geq n_0$  for some  $n_0$ . Hence,  $g(n) = O(f(n))$ , whence  $f(n) + o(f(n)) = O(f(n))$ . Thus,  $f(n) + o(f(n)) = \Theta(f(n))$ .

(b) **T F** The worst-case running time and expected running time are equal to within constant factors for any randomized algorithm.

**Solution: False.** Randomized quicksort has worst-case running time of  $\Theta(n^2)$  and expected running time of  $\Theta(n \lg n)$ .

(b) **T F** The collection  $\mathcal{H} = \{h_1, h_2, h_3\}$  of hash functions is universal, where the three hash functions map the universe  $\{A, B, C, D\}$  of keys into the range  $\{0, 1, 2\}$  according to the following table:

$x$	$h_1(x)$	$h_2(x)$	$h_3(x)$
$A$	1	0	2
$B$	0	1	2
$C$	0	0	0
$D$	1	1	0

**Solution: True.** A hash family  $\mathcal{H}$  that maps a universe of keys  $U$  into  $m$  slots is *universal* if for each pair of distinct keys  $x, y \in U$ , the number of hash functions  $h \in \mathcal{H}$  for which  $h(x) = h(y)$  is exactly  $|\mathcal{H}|/m$ . In this problem,  $|\mathcal{H}| = 3$  and  $m = 3$ . Therefore, for any pair of the four distinct keys, exactly 1 hash function should make them collide. By consulting the table above, we have:

$h(A) = h(B)$	only for $h_3$	mapping into slot 2
$h(A) = h(C)$	only for $h_2$	mapping into slot 0
$h(A) = h(D)$	only for $h_1$	mapping into slot 1
$h(B) = h(C)$	only for $h_1$	mapping into slot 0
$h(B) = h(D)$	only for $h_2$	mapping into slot 1
$h(C) = h(D)$	only for $h_3$	mapping into slot 0

### Problem -3. Short Answers

Give *brief*, but complete, answers to the following questions.

- (a) Argue that any comparison based sorting algorithm can be made to be stable, without affecting the running time by more than a constant factor.

**Solution:** To make a comparison based sorting algorithm stable, we just tag all elements with their original positions in the array. Now, if  $A[i] = A[j]$ , then we compare  $i$  and  $j$ , to decide the position of the elements. This increases the running time at a factor of 2 (at most).

- (b) Argue that you cannot have a Priority Queue in the comparison model with both the following properties.

- EXTRACT-MIN runs in  $\Theta(1)$  time.
- BUILD-HEAP runs in  $\Theta(n)$  time.

### Solution:

If such priority queues existed, then we could sort by running BUILD-HEAP ( $\Theta(n)$ ) and then extracting the minimum  $n$  times ( $n.\Theta(1) = \Theta(n)$ ). This algorithm would sort  $\Theta(n)$  time in the comparison model, which violates the  $\Theta(n \log n)$  lower bound for comparison based sorting.

- (c) Given a heap in an array  $A[1 \dots n]$  with  $A[1]$  as the maximum key (the heap is a max heap), give pseudo-code to implement the following routine, while maintaining the max heap property.

$\text{DECREASE-KEY}(i, \delta)$  – Decrease the value of the key currently at  $A[i]$  by  $\delta$ . Assume  $\delta \geq 0$ .

**Solution:**

```
DECREASE-KEY( $i, \delta$ )
   $A[i] \leftarrow A[i] - \delta$ 
  MAX-HEAPIFY( $A, i$ )
```

- (d) Given a sorted array  $A$  of  $n$  *distinct* integers, some of which may be negative, give an algorithm to find an index  $i$  such that  $1 \leq i \leq n$  and  $A[i] = i$  provided such an index exists. If there are many such indices, the algorithm can return any one of them.

**Solution:**

The key observation is that if  $A[j] > j$  and  $A[i] = i$ , then  $i < j$ . Similarly if  $A[j] < j$  and  $A[i] = i$ , then  $i > j$ . So if we look at the middle element of the array, then half of the array can be eliminated. The algorithm below (INDEX-SEARCH) is similar to binary search and runs in  $\Theta(\log n)$  time. It returns -1 if there is no answer.

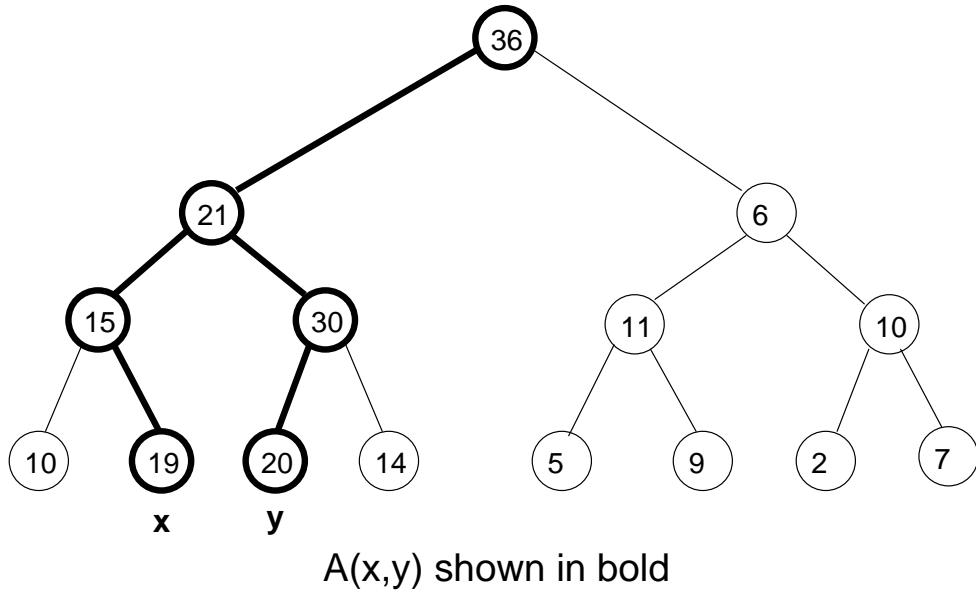
```
INDEX-SEARCH( $A, b, e$ )
  if ( $e > b$ )
    return -1
   $m = \lceil \frac{e+b}{2} \rceil$ 
  if  $A[m] = m$ 
    then return  $m$ 
  if  $A[m] > m$ 
    then return INDEX-SEARCH( $A, b, m$ )
  else return INDEX-SEARCH( $A, m, e$ )
```

**Problem -4.** Suppose you are given a complete binary tree of height  $h$  with  $n = 2^h$  leaves, where each node and each leaf of this tree has an associated “value”  $v$  (an arbitrary real number).

If  $x$  is a leaf, we denote by  $A(x)$  the set of ancestors of  $x$  (including  $x$  as one of its own ancestors). That is,  $A(x)$  consists of  $x$ ,  $x$ ’s parent, grandparent, etc. up to the root of the tree.

Similarly, if  $x$  and  $y$  are distinct leaves we denote by  $A(x, y)$  the ancestors of *either*  $x$  or  $y$ . That is,

$$A(x, y) = A(x) \cup A(y) .$$



Define the function  $f(x, y)$  to be the sum of the values of the nodes in  $A(x, y)$ .

Give an algorithm (pseudo-code not necessary) that efficiently finds two leaves  $x_0$  and  $y_0$  such that  $f(x_0, y_0)$  is as large as possible. What is the running time of your algorithm?

### Solution:

There are several different styles of solution to this problem. Since we studied divide-and-conquer algorithms in class, we just give a divide-and-conquer solution here. There were also several different quality algorithms, running in  $O(n)$ ,  $O(n \lg n)$ , and  $O(n^2 \lg n)$ . These were worth up to 11, 9, and 4 points, respectively. A correct analysis is worth up to 4 points.

First, let us look at an  $O(n \lg n)$  solution then show how to make it  $O(n)$ . For simplicity, the solution given here just finds the maximum value, but it is not any harder to return the leaves giving this value as well.

We define a recursive function  $\text{MAX1}(z)$  to return the maximum value of  $f(x)$ —the sum of the ancestors of a single node—over all leaves  $x$  in  $z$ 's subtree. Similarly, we define  $\text{MAX2}(z)$  to be a

function returning the maximum value of  $f(x, y)$  over all pairs of leaves  $x, y$  in  $z$ 's subtree. Calling MAX2 on the root will return the answer to the problem.

First, let us implement MAX1( $z$ ). The maximum path can either be in  $z$ 's left subtree or  $z$ 's right subtree, so we end up with a straightforward divide and conquer algorithm given as:

```
MAX1( $z$ )
1 return ( $value(z) + \max\{\text{MAX1}(\text{left}[z]), \text{MAX1}(\text{right}[z])\}$ )
```

For MAX2( $z$ ), we note that there are three possible types of solutions: the two leaves are in  $z$ 's left subtree, the two leaves are in  $z$ 's right subtree, or one leaf is in each subtree. We have the following pseudocode:

```
MAX2( $z$ )
1 return ( $value(z) + \max\{\text{MAX2}(\text{left}[z]), \text{MAX2}(\text{right}[z]), \text{MAX1}(\text{left}[z]) + \text{MAX1}(\text{right}[z])\}$ )
```

### Analysis:

For MAX1, we have the following recurrence

$$\begin{aligned} T_1(n) &= 2T_1\left(\frac{n-1}{2}\right) + \Theta(1) \\ &= \Theta(n) \end{aligned} \tag{1}$$

by applying the Master Method.

For MAX2, we have

$$\begin{aligned} T_2(n) &= 2T_2\left(\frac{n-1}{2}\right) + 2T_1\left(\frac{n-1}{2}\right) + \Theta(1) \\ &= 2T_2\left(\frac{n-1}{2}\right) + \Theta(n) \\ &= \Theta(n \lg n) \end{aligned} \tag{2}$$

by case 2 of the Master Method.

To get an  $O(n)$  solution, we just define a single function, MAXBOTH, that returns a pair—the answer to MAX1 and the answer to MAX2. With this simple change, the recurrence is the same as MAX1

### Problem -5. Sorting small multisets

For this problem  $A$  is an array of length  $n$  objects that has at most  $k$  distinct keys in it, where  $k < \sqrt{n}$ . Our goal is to sort this array in time faster than  $\Omega(n \log n)$ . We will do so in two phases. In the first phase, we will compute a *sorted* array  $B$  that contains the  $k$  *distinct* keys occurring in  $A$ . In the second phase we will sort the array  $A$  using the array  $B$  to help us.

Note that  $k$  might be very small, like a constant, and your running time should depend on  $k$  as well as  $n$ . The  $n$  objects have satellite data in addition to the keys.

**Example:** Let  $A = \left[5, 10^{10}, \pi, \frac{128}{279}, 10^{10}, \pi, 5, 10^{10}, \pi, \frac{128}{279}\right]$ . Then  $n = 10$  and  $k = 4$ .

In the first phase we compute  $B = \left[\frac{128}{279}, \pi, 5, 10^{10}\right]$ .

The output after the second phase should be  $\left[\frac{128}{279}, \frac{128}{279}, \pi, \pi, \pi, 5, 5, 10^{10}, 10^{10}, 10^{10}\right]$ .

Your goal is to design and analyse efficient algorithms and analyses for the two phases. Remember, the more efficient your solutions, the better your grade!

- (a) Design an algorithm for the first phase, that is computing the sorted array  $B$  of length  $k$  containing the  $k$  distinct keys. The value of  $k$  is not provided as input to the algorithm.

### Solution:

The algorithm adds (non-duplicate) elements to array  $B$  while maintaining  $B$  sorted at every intermediate stage. For  $i = 1, 2, \dots, n$ , element  $A[i]$  is binary searched in array  $B$ . If  $A[i]$  occurs in  $B$ , then it need not be inserted. Otherwise, binary search also provides the location where  $A[i]$  should be inserted into array  $B$  to maintain  $B$  in sorted order. All elements in  $B$  to the right of this position are shifted by one place to make place for  $A[i]$ .

- (b) Analyse your algorithm for part (a).

### Solution:

Binary search in array  $B$  for each element of array  $A$  takes  $O(\lg k)$  time since size of  $B$  is at most  $k$ . This takes a total of  $O(n \lg k)$  time. Also, a new element is inserted into array  $B$  exactly  $k$  times, and the total time over all such insertions is  $O(1 + 2 + \dots + k) = O(k^2)$ . Thus, the total time for the algorithm is  $O(n \lg k + k^2) = O(n \lg k)$  since  $k < \sqrt{n}$ .

- (c) Design an algorithm for the second phase, that is, sorting the given array  $A$ , using the array  $B$  that you created in part (a). Note that since the objects have satellite data, it is not sufficient to count the number of elements with a given key and duplicate them.

*Hint: Adapt Counting Sort.*

### Solution:

Build the array  $C$  as in counting sort, with  $C[i]$  containing the number of elements in  $A$  that have values less than or equal to  $B[i]$ . Counting sort will not work as is since

$A[i]$  is necessarily an integer. Or, it may be some integer of very large value (there is no restriction on our input range). Therefore  $A[i]$  is an invalid index into our array  $C$ . What we would like to do is assign an integral “label” for the value  $A[i]$ . The label we choose is the index of the value  $A[i]$  in the array  $B$  calculated in the last part of the problem.

How do we find this index? We could search through  $B$  from beginning to end, looking for the value  $A[i]$ , then returning the index of  $B$  that contains  $A[i]$ . This would take  $O(k)$  time. But, since  $B$  is already sorted, we can use BINARY-SEARCH to speed this up to  $O(\log k)$ . Let BINARY-SEARCH( $S, x$ ) be a procedure that takes a sorted array  $S$  and an item  $x$  within the array, and returns  $i$  such that  $S[i] = x$ . The modified version of COUNTING SORT is included below, with modified lines in bold:

```

COUNTING-SORT( $A$ )
/* Uses Arrays  $C[1..k]$ ,  $D[1..k]$ , and  $A\text{-out}[1..n]$  */
For  $i = 1$  to  $k$  do  $C[i] \leftarrow 0$ ; /* Initialize */
For  $i = 1$  to  $n$  do /* Count number of elements */
    Location  $\leftarrow$  BINARY-SEARCH( $B, A[i]$ );
     $C[\text{Location}] \leftarrow C[\text{Location}] + 1$ ;
 $D[1] \leftarrow C[1]$ ;
For  $j = 2$  to  $k$  do /* Build cumulative counts */
     $D[j] \leftarrow D[j - 1] + C[j]$ ;
For  $i = n$  downto  $1$  do /* Construct Sorted List A-Out */
    Location  $\leftarrow$  BINARY-SEARCH( $B, A[i]$ );
    Out-Location  $\leftarrow D[\text{Location}]$ ;
     $D[\text{Location}] \leftarrow D[\text{Location}] - 1$ ;
     $A\text{-out}[\text{Out-Location}] \leftarrow A[i]$ ;
Output( $A\text{-out}$ );

```

- (d) Analyse your algorithm for part (c).

### Solution:

The running time of the modification to COUNTING-SORT we described can be broken down as follows:

- First Loop:  $O(k)$ .
- Second Loop:  $O(n)$  iterations, each iteration performing a BINARY-SEARCH on an array of size  $k$ . Total Work:  $O(n \log k)$ .
- Third Loop:  $O(k)$ .
- Fourth Loop:  $O(n)$  iterations, each iteration performing a BINARY-SEARCH on an array of size  $k$ . Total Work:  $O(n \log k)$ .

The running time is dominated by the second and fourth loops, so the total running time is  $O(n \log k)$ .

---

## Practice for Final Exam (Solutions)

- Do not open this quiz booklet until you are directed to do so.
- This final ends at 12:00 P.M.
- This exam is closed book. You may use three sides of handwritten  $8\frac{1}{2}'' \times 11''$  crib sheets.
- 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!

Name: \_\_\_\_\_

Please circle your TA's name and recitation:

Nitin Brian Mihai Yoav

10am 11am 12pm 1pm 2pm

**Problem 1.** Short questions

In the following questions, fill out the blank boxes. In case more than one answer is correct, you should provide the **best known** correct answer. E.g., for a question “Sorting of  $n$  elements in the comparison-based model can be done in  time”, the best correct answer is  $O(n \log n)$ . No partial credit will be given for correct but suboptimal answers. However, you **do not** have to provide any justification.

- (a) Consider a modification to QUICKSORT, such that each time PARTITION is called, the median of the partitioned array is found (using the SELECT algorithm) and used as a pivot.

The worst-case running time of this algorithm is:

**Answer:**  $\Theta(n \log n)$ . Reason: the median can be found in  $O(n)$  time, so we have the running time recurrence  $T(n) = 2T(n/2) + O(n)$ .

- (b) If a data structure supports an operation `foo` such that a sequence of  $n$  `foo`'s takes  $\Theta(n \log n)$  time to perform in the worst case, then the amortized time of a `foo` operation is  $\Theta\left(\frac{\Theta(n \log n)}{n}\right)$ , while the actual time of a single `foo` operation could be as high as  $\Theta\left(\frac{\Theta(n \log n)}{n}\right)$ .

**Answer:** Amortized time  $\Theta(\log n)$ , worst-case time  $\Theta(n \log n)$ .

- (c) Does there exist a polynomial time algorithm that finds the value of an  $s-t$  minimum cut in a directed graph?  (Yes or No)

**Answer:** Yes. Reason: The value of an  $s-t$  minimum cut is equal to the value of the  $s-t$  maximum flow, and there are many algorithms for computing  $s-t$  maximum flows in polynomial time (e.g. the Edmonds-Karp algorithm).

**Problem 2.** True or False?

For each of the questions below, circle either **T** (for True) or **F** (for False). **No explanations are needed.** Incorrect answers or unanswered questions are worth zero points.

- T F** By the master theorem, the solution to the recurrence  $T(n) = 3T(n/3) + \log n$  is  $T(n) = \Theta(n \log n)$ .

**Answer:** False

- T F** Computing the median of  $n$  elements takes  $\Omega(n \log n)$  time for any algorithm working in the comparison-based model.

**Answer:** False

- T F** Every binary search tree on  $n$  nodes has height  $O(\log n)$ .

**Answer:** False

- T F** Given a graph  $G = (V, E)$  with cost on edges and a set  $S \subseteq V$ , let  $(u, v)$  be an edge such that  $(u, v)$  is the minimum cost edge between any vertex in  $S$  and any vertex in  $V - S$ . Then, the minimum spanning tree of  $G$  must include the edge  $(u, v)$ . (You may assume the costs on all edges are distinct, if needed.)

**Answer:** True

- T F** Let  $T$  be a minimum spanning tree of  $G$ . Then, for any pair of vertices  $s$  and  $t$ , the shortest path from  $s$  to  $t$  in  $G$  is the path from  $s$  to  $t$  in  $T$ .

**Answer:** False

- T F** If a problem  $L_1$  is polynomial time reducible to a problem  $L_2$  and  $L_2$  has a polynomial time algorithm, then  $L_1$  has a polynomial time algorithm.

**Answer:** True

**Problem 3.** True or False and Justify.

For each of the questions below, circle either **T** (for True) or **F** (for False). Then give a brief **justification** for your answer. Your answers will be evaluated based on the justification, and **not** the **T/F** marking alone.

- T F** There exists a data structure to maintain a dynamic set with operations  $\text{Insert}(x,S)$ ,  $\text{Delete}(x,S)$ , and  $\text{Member?}(x,S)$  that has an expected running time of  $O(1)$  per operation.

**Answer:** True. Use a hash table.

- T F** The total amortized cost of a sequence of  $n$  operations (i.e., the sum over all operations, of the amortized cost per operation) gives a lower bound on the total actual cost of the sequence.

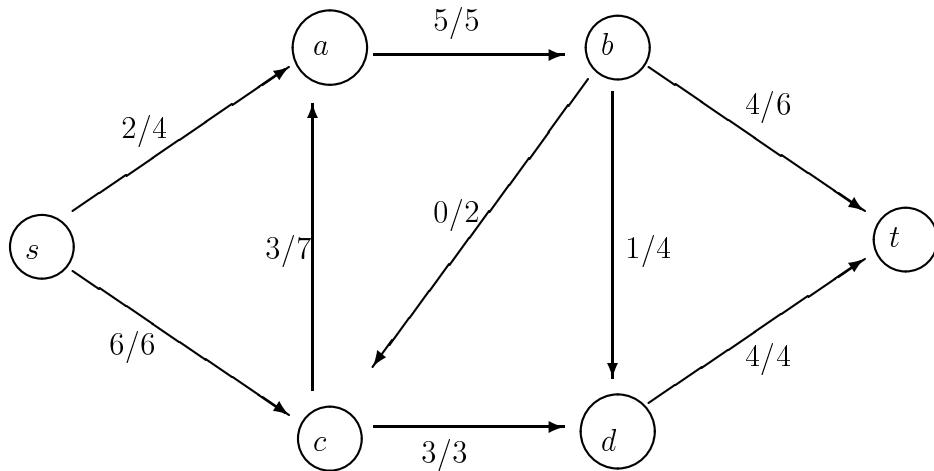
**Answer:** False. This only gives an upper bound on the actual cost.

- T F** Let  $G = (V, E)$  be a weighted graph and let  $M$  be a minimum spanning tree of  $G$ . The path in  $M$  between any pair of vertices  $v_1$  and  $v_2$  must be a shortest path in  $G$ .

**Answer:** False. Consider the graph in which  $V = \{a, b, c\}$  and the edges are  $(a, b), (b, c), (c, a)$ . The weight of the edges are 3, 3 and 4 respectively. The MST is clearly  $(a, b), (b, c)$ . However the shortest path between  $a$  and  $c$  is of cost 4 not 6 as seen from the MST.

- T F** The figure below describes a flow assignment in a flow network. The notation  $a/b$  describes  $a$  units of flow in an edge of capacity  $b$ .

True or False: The following flow is a maximal flow.



**Answer:** True. The flow pushes 8 units and the cut  $\{s, a, c\}$  vs.  $\{b, d, t\}$  has capacity 8. The cut must be maximum by the Max-Flow Min-cut theorem.

**Problem 4.** Typesetting

Suppose we would like to neatly typeset a text. The input is a sequence of  $n$  words of lengths  $l_1, l_2, \dots, l_n$  (measured in the number of fixed-size characters they take). Each line can hold at most  $P$  characters, the text is left-aligned, and words cannot be split between lines. If a line contains words from  $i$  to  $j$  (inclusive) then the number of spaces at the end of the line is  $s = P - \sum_{k=i}^j l_k - (j - i)$ . We would like to typeset the text so as to avoid large white spaces at the end of lines; formally, we would like to minimize the sum over all lines of the square of the number of white spaces at the end of the line. Give an efficient algorithm for this problem. What is its running time?

**Answer:**

We solve the problem with dynamic programming. Let  $A[j]$  denote the optimal ‘‘cost’’ (that is, the sum of the square of number of trailing white space characters over all lines) one may achieve by typesetting only the words  $1\dots j$  (ignoring the remaining words). We can express  $A[j]$  recursively as follows:

$$A[j] = \min_{i < j: T[j] - T[i] \leq P} A[i] + (P - (T[j] - T[i]))^2,$$

where  $T[j] = \sum_{i=1}^j l_i$ . A table of the values  $T[1\dots n]$  can be initially computed in  $O(n)$  time. The equation above says the following: in order to optimally typeset words  $1\dots j$ , we must first optimally typeset words  $1\dots i$  for some  $i < j$ , and then place the remaining words  $i+1\dots j$  on the final line.

Using dynamic programming, we compute each value  $A[j]$  in sequence for all  $j = 1\dots n$ . Each value is requires  $O(n)$  time to compute, for a total running time of  $O(n^2)$ . After termination,  $A[n]$  will contain the value of the optimal solution; we can reconstruct the solution itself (that is, the locations where we should insert line breaks) by maintaining ‘‘backpointers’’ as is usually done with dynamic programming.

**Problem 5.** Princess of Algorithmia (2 parts)

Political upheaval in Algorithmia has forced Princess Michelle X. Goewomans to vacate her royal palace; she plans to relocate to a farm in Nebraska. The princess wants to move all of her possessions from the palace in Algorithmia to the Nebraskan farm using as few trips as possible in her Ferrari. For simplicity, let us assume that the princess's Ferrari has size 1, and that her  $n$  possessions  $x_1, x_2, \dots, x_n$  have real number sizes between 0 and 1. The problem is to divide the princess's possessions into as few carloads as possible, so that all her possessions will be transported from Algorithmia to Nebraska and so that her Ferrari will never be overpacked. It turns out this problem is NP-hard.

Consider the following **first-fit** approximation algorithm. Put item  $x_1$  in the first carload. Then, for  $i = 2, 3, \dots, n$ , put  $x_i$  in the first carload that has room for it (or start a new carload if necessary). For example, if  $x_1 = 0.2$ ,  $x_2 = 0.4$ ,  $x_3 = 0.6$ , and  $x_4 = 0.3$ , the first-fit algorithm would place  $x_1$  and  $x_2$  in the first carload,  $x_3$  in the second carload, and then  $x_4$  in the first carload (where there is still enough room). Note that all decisions are made offline; we divide the princess's  $n$  possessions into carloads before any trips have actually been made.

- (a) The princess's charming husband, Craig, has asserted, "The first-fit algorithm will always minimize the total number of car trips needed." Give a counterexample to Craig's claim.

**Answer:**

Let  $n = 4$  and let  $x_1 = 0.3$ ,  $x_2 = 0.8$ ,  $x_3 = 0.2$ , and  $x_4 = 0.7$ . The optimal number of car trips needed is 2, while the first-fit algorithm produces 3 car trips.

- (b) Prove that the first-fit algorithm has a ratio bound of 2. (*Hint:* How many carloads can be less than half full?)

**Answer:**

Consider a carload  $u$  of size  $p$  which is less than half full, i.e.,  $p < 0.5$ . Let  $x_i$  be the first item to go into the next carload  $v$ . Then, due to the nature of the first-fit algorithm, it follows that  $p+x_i > 1$  and  $x_i > 0.5$ . Hence, carload  $v$  is more than half full. For calculation purposes, we can transfer a portion of the load of item  $x_i$  from carload  $v$  to carload  $u$  to make carload  $u$  exactly half full while keeping carload  $v$  at least half full. This is because  $0.5-p < x_i - 0.5$ . After we perform this operation, every carload is at least half full. Thus, if  $h$  is the number of carloads produced by the first-fit algorithm, then  $\sum_{i=1}^n x_i \geq 0.5h$ . Note that the optimal number of carloads  $OPT$  is at least  $\sum_{i=1}^n x_i$ . Hence,  $h/OPT \leq 2$ , which proves that the first-fit algorithm has a ratio bound of 2.

**Problem 6. Division of Power in King Arthur's Court** (2 parts)

A large number of problems that we now refer to as optimization problems, actually date back to King Arthur's time. King Arthur's court had many knights. They were the source of many of his problems, some computational and others not. King Arthur often sought the advice of his powerful wizard — Merlin — to solve these computational problems. It is rumored that the mighty Merlin underwent a voyage in a time machine, took 6.046 in the Fall of 2000 and used these skills to solve many of the challenges posed to him.

In the two parts of this question, we'll see some of the problems posed to him.

**(a) Ruling the land:**

King Arthur's court had  $n$  knights. He ruled over  $m$  counties. Each knight  $i$  had a quota  $q_i$  of the number of counties he could oversee. Each county  $j$ , in turn, produced a set  $S_j$  of the knights that it would be willing to be overseen by. The King sets upon Merlin the task of computing an assignment of counties to the knights so that no knight would exceed his quota, while every county  $j$  is overseen by a knight from its set  $S_j$ .

- (i) Show how Merlin can employ the Max-Flow algorithm to compute the assignments. Describe the running time of your algorithm. (You may express your running time using function  $F(v, e)$ , where  $F(v, e)$  denotes the running time of the Max-Flow algorithm on a network with  $v$  vertices and  $e$  edges.)

**Answer:** We make a graph with  $n+m+2$  vertices,  $n$  vertices  $k_1, \dots, k_n$  corresponding to the knights,  $m$  vertices  $c_1, \dots, c_m$  corresponding to the counties, and two special vertices  $s$  and  $t$ .

We put an edge from  $s$  to  $k_i$  with capacity  $q_i$ . We put an edge from  $k_i$  to  $c_j$  with capacity 1 if county  $j$  is willing to be ruled by knight  $i$ . We put an edge of capacity 1 from  $c_j$  to  $t$ .

We now find a maximum flow in this graph. If the flow has value  $m$ , then there is a way to assign knights to all counties, as argued next. Since this flow is integral, it will pick one incoming edge for each county  $c_j$  to have flow of 1. If this edge comes from knight  $k_i$ , then county  $j$  is ruled by knight  $i$ .

The running time of this algorithm is  $F(n + m + 2, \sum_j |S_j|)$ .

- (ii) Merlin runs his algorithm (from Part (i)) but the algorithm does not produce an assignment that fits the requirements. King Arthur demands an explanation. Merlin explains his algorithm to King Arthur, and King Arthur is convinced that the algorithm is correct. But he does not believe Merlin has executed this algorithm correctly on the given instance of the problem. He wants Merlin to convince him that no assignment is possible in this particular case. Based on your understanding on Max-Flow, what proof would you suggest? (Note that your proof strategy should work for every instance of the problem for which a satisfactory assignment does not exist.)

**Answer:** If there is no flow of size  $m$ , then there must a cut in this graph of capacity less than  $m$ . Looking at the structure of the cut, we note that this gives a set  $T$  of counties such that if the  $U$  is the union of the sets  $S_j$  for  $j \in T$ , then the sum  $\sum_{i \in U} q_i$  is less than  $|T|$ . (I.e., there is a subset of counties such that the quotas of the knights that they are willing to be ruled by is smaller than the number of counties in the subset.) Merlin can present the sets  $T$  and  $U$  to Arthur explaining why  $T$  can not be assigned knights.

(b) **Conflict resolution:** At an annual meeting of the knights, King Arthur notes that several of knights have started to develop conflicts. Sending his undercover agents, he has managed to find out a list of all possible duelling pairs. He now wishes to exile a small set of knights away so that no duelling pair remains at the annual meeting. Merlin is now set with this task of finding out which knights to send away. Being a perfectionist, Merlin wishes to find the smallest set he could send into exile. However, after much thought, he is unable to find the optimal solution using any efficient algorithm.

- (i) Explain why? (I.e., what problem is Merlin trying to solve and why is he unable to do it.)

**Answer:** The set of knights to be exiled form a vertex cover in the incompatibility graph. Finding the smallest such cover is NP-complete and this is why Merlin is unable to solve the problem efficiently.

- (ii) How could Merlin weaken his goals to find a reasonable solution?

**Answer:** He can use a 2-approximation algorithm for Vertex Cover and thus exile a subset of knights of size at most twice the optimum number.

## Practice Quiz 1

- Do not open this quiz booklet until you are directed to do so. Read all the instructions first.
- The quiz contains multi-part problems. You have 60 minutes to earn 60 points.
- This quiz booklet contains **8 double-sided** pages, including this one and a double-sided sheet of scratch paper; there should be 12 (numbered) pages of problems.
- This quiz is closed book. You may use one double sided Letter ( $8\frac{1}{2}'' \times 11''$ ) or A4 crib sheet. No calculators or programmable devices are permitted. Cell phones must be put away.
- Write your solutions in the space provided. Extra scratch paper may be provided if you need more room, although your answer should fit in the given space.
- Do not waste time re-deriving facts that we have studied. It is sufficient to cite known results.
- Do not spend too much time on any one problem. Generally, a problem's point value is an indication of how much time to spend on it.
- 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.
- Good luck!

Problem	Points	Grade	Initials
1	8		
2	10		
3	10		
4	10		
5	10		
6	12		
Total			

Name: \_\_\_\_\_  
Circle your recitation:

R01	R02	R03	R04	R05	R06
F10	F11	F12	F1	F2	F3
Michael	Michael	Prasant	Prasant	Szymon	Szymon

## Probability Toolkit

1. **Markov Inequality:** For any non-negative random variable  $X$ ,  $\Pr \{X \geq \lambda\} \leq E[X] / \lambda$ .
2. **Chernoff Bounds:** Let  $X_1, X_2, X_3 \dots X_n$  be independent Bernoulli trials such that, for  $1 \leq i \leq n$ ,  $\Pr \{X_i = 1\} = p_i$ , where  $0 < p_i < 1$ . Then, for  $X = \sum_{i=1}^n p_i$ ,  $\mu = E[X] = \sum_{i=1}^n p_i$ ,

$$\Pr \{X < (1 - \epsilon)\mu\} < \begin{cases} e^{-\mu\epsilon^2/2} & \text{for } 0 < \epsilon \leq 1 \\ 2^{-(1+\epsilon)\mu} & \text{for } \epsilon > 2e - 1 \end{cases}$$

**Problem 1. Recurrences** Solve the following recurrences by giving tight  $\Theta$ -notation bounds. As usual, assume that  $T(n) = O(1)$  for  $n \leq 2$ . [8 points] (2 parts)

(a) [4 points]  $T(n) = 9T(\sqrt[3]{n}) + \Theta(\log(n)).$

**Solution:** Let  $n = 2^m$ . Then the recurrence becomes  $T(2^m) = 9T(2^{m/3}) + \Theta(m)$ . Setting  $S(m) = T(2^m)$  gives us  $S(m) = 9S(m/3) + \Theta(m)$ . Using case 1 of the Master's Method gives us  $S(m) = \Theta(m^2)$  or  $T(n) = \Theta(\log^2 n)$

(b) [4 points]  $T(n) = 7T(n/2) + \Theta(n).$

**Solution:** Master's theorem.  $T(n) = \Theta(n^{\lg 7})$ .

**Problem 2. True or False, and Justify** [10 points] (5 parts)

Circle **T** or **F** for each of the following statements, and briefly explain why. Your justification is worth more points than your true-or-false designation.

- (a) **T F** [2 points] Michael is working on his thesis and is stuck on the following problem: He has pairs of DNA sequences and a priority (the distribution of priorities is uniform over  $[0, 1]$ ) associated with each DNA sequence. He would like to insert, delete pairs dynamically and search through  $N$  DNA sequences in  $O(\log N)$  expected time and at the same time retrieve the DNA sequence with the maximum priority in  $O(1)$  time in the worst case. He hears about TREAPS from a student enrolled in 6.046 and claims he has completed his thesis. Is he right?

**Solution:** True. Treaps serve the same purpose. Since the distribution of priorities is uniform, the expected tree height is  $O(\log N)$  which gives us the desired expected time for search.

- (b) **T F** [2 points] A static set of  $n$  elements can be stored in a hash table that uses  $O(n)$  space and supports look up in  $O(1)$  time in the worst case.

**Solution:** True. Perfect Hashing.

- (c) **T F** [2 points] A Monte Carlo algorithm always runs in deterministic time.

**Solution:** True. A Monte Carlo algorithm always runs in deterministic time. Its output, however, may not be always correct.

- (d) **T F** [2 points] Suppose we have computed a minimum spanning tree (MST) and its total weight for some graph  $G = (V, E)$ . If we make a new graph  $G'$  by adding 1 to the weight of every edge in  $G$ , we will need to spend  $\Omega(|E|)$  time to compute an MST and its total weight for the new graph  $G'$ .

**Solution:** False. If  $T$  is an MST for  $G$  with weight  $w$ , then it is also an MST for  $G'$  with weight  $w + |V| - 1$ .

- (e) **T F** [2 points] A data structure  $D$  allows insertions, deletions and search in  $O(1)$  amortized time. Imagine the state of  $D$  after  $n$  insertions and  $m$ ,  $m \gg n$ , searches. Then,  $D$  cannot take  $\Omega(n^2)$  time for any deletion.

**Solution:** False. An amortized bound does not guarantee worst case time bounds on the execution of any single operation. Here, the amortized bound only guarantees that the next deletion cannot take more than  $O(n + m)$  time, which is the worst-case bound for the whole sequence.

**Problem 3. Boosting the probability** (2 parts) [10 points]

- (a) [5 points] Consider a Randomized algorithm A which is always correct when it outputs YES while it may tag a YES instance as a NO with probability  $1/3$ . It has a runtime of  $O(n \log n)$  for an input instance of size  $n$ . Can you amplify the probability of success to  $1 - O(1/n^2)$ ? If yes, give an upper bound on the running time required to achieve it. Otherwise, give a justification as to why it is not possible to amplify probability of success.

**Solution:** Yes. The answer follows the construction given in pset problem 3-1(d). Let  $A'$  be an algorithm which runs algorithm A  $k$  independent times.  $A'$  returns YES if any instance of A returns YES and returns NO otherwise. Thus,  $A'$  is always correct when it returns YES while it may tag a YES instance as a NO with probability  $(1/3)^k$ . The probability of success of  $A'$  is  $1 - (1/3)^k$ . If  $k = 2 \log_3 n$ , then the probability of success is  $1 - 1/n^2 = 1 - O(1/n^2)$ .  $A'$  runs in  $O(n \log^2 n)$  time.

- (b) [5 points]**  $\Pi$  is a Randomized algorithm that has an error probability of  $1/4$ . That is, it is a two sided error algorithm and hence outputs both false positives and false negatives. Can you design a new algorithm  $\Pi'$  which has the same functionality as  $\Pi$  but an error probability of  $2^{-c}$ , where  $c > 3$  is a constant? If so, how are the run times of  $\Pi$  and  $\Pi'$  related? Otherwise, give a justification as to why one cannot construct  $\Pi'$ .

**Solution:** It is possible to construct  $\Pi'$ . To achieve the necessary bounds, we run  $\Pi$   $t$  times and then take a majority, where  $t$  is a parameter that comes out from the analysis.

If we execute  $\Pi$   $t$  times, the expected number of correct answers is  $3t/4$ . However, we cannot be sure about the deviations from the expectation – which can be huge – so we need something more stronger. Specifically, we need to show that the deviation from the mean is also small. In particular, we will need to show that the probability we see less than  $t/2$  successes is small (in our case  $2^{-c}$ ).

Let  $X$  denote the number of successes. Thus, we would like to bound the  $\Pr\{X < t/2\}$ . Since, each of the  $t$  executions are independent Bernoulli random variables we can use Chernoff bounds over the range  $\epsilon \in (0, 1)$ .

$$\begin{aligned}\Pr\{X < (1 - \epsilon)\mu\} &< e^{-\mu\epsilon^2/2} \\ \Pr\{X < (1 - \epsilon)3t/4\} &< e^{-3t\epsilon^2/8}\end{aligned}$$

Set  $\epsilon = 1/3$  to bound

$$\Pr\{X < t/2\} < e^{-t/24}$$

We request that

$$e^{-t/24} < 2^{-c}$$

which is true when

$$t > 24c \ln 2$$

Thus, algorithm  $\Pi'$  executes  $\Pi$   $24c \ln 2$  times and then takes the majority answer. The resulting algorithm has an error probability smaller than  $2^{-c}$  and running time  $24c \ln 2$  times longer than the original algorithm.

*Remark on Chernoff Bounds:* It is important to distinguish the cases when you need Chernoff and the cases which do not need Chernoff like part (a). Chernoff bounds are extremely useful if you have independent Bernoulli random variables and your interest is to bound the deviation from expectation.

**Problem 4. Binary Counting** (5 parts) [10 points] Consider the problem of implementing a  $k$ -bit binary counter that counts upward from 0. We use an array  $A[0 \dots k - 1]$  of bits, where  $A.length = k$ , as the counter. The least significant bit is stored in  $A[0]$ . So,  $x = \sum_{i=0}^{k-1} A[i].2^i$ . Initially,  $x = 0$ , and thus  $A[i] = 0$  for  $i = 0, 1 \dots n - 1$ . To add 1(modulo $2^k$ ) to the value in the counter, we use the following procedure.

INCREMENT(A):

```
1  i = 0
2  while i < A.length and A[i] == 1
3      A[i] = 0
4      i = i + 1
5  if i < A.length
6      A[i] = 1
```

- (a) [2 points] Argue that  $A[1]$  is flipped only every time the counter is incremented. Extend the above argument to show that the bit  $A[i]$  is flipped every  $2^i$ -th time for  $i \geq 0$ .

**Solution:** It is easy to see that the first bit  $A[0]$  is flipped every time the counter is incremented.  $A[1]$  is flipped only when  $A[0]=1$ , which is every other increment operation. In general for  $i \geq 1$ ,  $A[i]$  is flipped only when  $A[j]=1$  for all  $0 \leq j < i$ . This only happens once out of every  $2^i$  possibilities for  $A[0], \dots, A[i - 1]$ .

- (b)** [2 points] Use an aggregate analysis to conclude that the total work performed for  $n$  INCREMENT operations is  $O(n)$  in the worst case.

**Solution:** The total cost is equal to the number of times a bit is flipped. From part (a) we know that  $A[i]$  will flip a total of  $\lfloor n/2^i \rfloor$ .

$$\begin{aligned}
\sum_0^n c_i &= \sum_0^k \text{Number of times } A[i] \text{ is flipped} \\
&= \sum_0^k \lfloor n/2^i \rfloor \\
&\leq \sum_0^k n/2^i \\
&= n \sum_0^k 1/2^i \\
&\leq 2n \\
&= O(n)
\end{aligned}$$

- (c)** [2 points] Show that if a DECREMENT operation were included in the  $k$ -bit counter,  $n$  operations could cost as much as  $\Theta(nk)$  time.

**Solution:** Consider a sequence of operation that begins with DECREMENT and then alternates between INCREMENT and DECREMENT. Then, the counter alternates between all 0's and all 1's. Each operation changes every bit and costs  $\Theta(k)$  time. The total cost is  $\Theta(nk)$ .

- (d) [2 points] Use a potential function argument, with the number of 1s in the counter after the operation as the potential function, to prove that each INCREMENT operation cost  $O(1)$  amortized time.

**Solution:** Let  $A_i$  be the state of the counter after the  $i$ -th operation,  $\Phi(A_i)$  be the number of 1's in  $A_i$ , and  $\hat{c}_i$  be the amortized cost of the  $i$ -th operation. The total amortized cost is

$$\sum_0^n \hat{c}_i = \sum_0^n c_i + \Phi(A_n) - \Phi(A_0)$$

Since  $\Phi(A_0) = 0$  and  $\Phi(A_i) \geq 0$ , the total amortized cost is an upper bound on the actual total cost. If  $A_{i-1}$  is filled with all 1's, then  $\Phi(A_i) = 0$ ,  $\Phi(A_{i-1}) = k$ , and  $c_i = k$ . In this case,  $\hat{c}_i = 0$ . Otherwise if  $A_{i-1}$  is not filled with 1's, then operation  $i$  will flip one bit from 0 to 1 and flip  $c_i - 1$  bits from 1 to 0. In this case,  $\Phi(A_i) - \Phi(A_{i-1}) = c_i - 2$  and  $\hat{c}_i = 2$ . The amortized cost is always  $O(1)$ .

- (e) [2 points] Suppose that a counter begins at a number with  $b$  1s in its binary representation, rather than at 0. Show that the cost of performing  $n$  INCREMENT operations is  $O(n)$  if  $n = \Omega(b)$ . (Do not assume that  $b$  is constant.)

**Solution:** If we use the same potential function, then  $\Phi(A_0) = b$ , and the potential difference  $\Phi(A_i) - \Phi(A_0) \geq -b$  may now be negative. However even though  $\sum \hat{c}_i$  may no longer be an upper bound on the total actual cost, we have a new upper bound:

$$\sum c_i \leq \sum \hat{c}_i + \Phi(A_0) = O(n + b)$$

If  $n = \Omega(b)$ , then the total cost is still  $O(n)$ .

**Problem 5.** (2 parts) [10 points] Suppose that you are given an array  $A$  of  $n$  bits that is either of type 1: contains half zeros and half ones in some arbitrary order or of type 2: contains  $2n/3$  zeros and  $n/3$  ones in some arbitrary order. You are given either a type 1 or a type 2 array with equal probability. Your goal is to determine whether  $A$  is type 1 or type 2.

- (a) [5 points] Give an exact lower bound in terms of  $n$  (not using asymptotic notation) on the worst-case running time of any deterministic algorithm that solves this problem.

**Solution:** Any correct deterministic algorithm must look at exactly  $5n/6 + 1$  entries in the worst case. Otherwise, the adversary can show it  $n/3$  ones, and  $n/2$  zeros ( $5/6n$  entries). The remaining  $n/6$  elements will either be all ones (type 1) or all zeros (type 2) and the algorithm must make another test.

- (b) [5 points] Consider the following randomized strategy: Choose uniformly at random an element from the given array. If the element is 0, it outputs “type 2” else it outputs “type 1”. Show that this algorithm makes an error with probability at most  $1/2$ .

**Solution:** The algorithm makes an error if it picks the wrong type.

$$\begin{aligned}\Pr(\text{error}) &= \Pr(\text{output type} \neq \text{actual type}) = \\ &= \Pr(\text{drawn 0} \wedge \text{type 1}) + \Pr(\text{drawn 1} \wedge \text{type 2}) = \\ &= \Pr(\text{drawn 0}|\text{type 1}) \Pr(\text{type 1}) + \Pr(\text{drawn 1}|\text{type 2}) \Pr(\text{type 2}) = \\ &= \frac{1}{2} \cdot \frac{1}{2} + \frac{1}{3} \cdot \frac{1}{2} = \frac{5}{12}\end{aligned}$$

**Problem 6. Universal Hashing** (4 parts) [12 points]

Recall that a collection  $\mathcal{H}$  of hash function from a universe  $\mathcal{U}$  to a range  $R$  is called **universal** if for all  $x \neq y$  in  $\mathcal{U}$  we have

$$\Pr_{h \in \mathcal{H}} [ h(x) = h(y) ] = \frac{1}{|R|}$$

We want to implement universal hashing from  $\mathcal{U} = \{0, 1\}^p$  to  $R = \{0, 1\}^q$  (where  $p > q$ ).

For any  $q \times p$  boolean matrix  $A$  and any  $q$ -bit vector  $b$  we define the function  $h_{A,b} : \{0, 1\}^p \rightarrow \{0, 1\}^q$  as  $h_{A,b}(x) = Ax + b$ , where by this we mean the usual matrix-vector multiplication and the usual vector addition, except that all the operations are done modulo 2. For example, if  $q = 2, p = 3$  and

$$A = \begin{pmatrix} 0 & 1 & 1 \\ 1 & 0 & 1 \end{pmatrix} \quad b = \begin{pmatrix} 1 \\ 0 \end{pmatrix} \quad x = \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix},$$

we have

$$h_{A,b}(x) = \begin{pmatrix} 0 & 1 & 1 \\ 1 & 0 & 1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} + \begin{pmatrix} 1 \\ 0 \end{pmatrix} = \begin{pmatrix} x_2 + x_3 + 1 \mod 2 \\ x_1 + x_3 \mod 2 \end{pmatrix}$$

Let us establish that the hash family  $\mathcal{H}$  is indeed universal for the specified range.

Notice that for any function  $h_{A,b}$  we have  $h(x) = h(y)$  if and only if  $A(x - y) = \bar{0}$ . We want to show that for any non-zero vector  $z (= x - y) \in \{0, 1\}^p$ , if we choose  $A$  at random from  $\{0, 1\}^{q \times p}$  then the probability of getting  $Az = \bar{0}$  is exactly  $1/2^q$ .

So let  $z$  be any non-zero vector in  $\{0, 1\}^p$ , and assume w.l.o.g. that the first coordinate of  $z$  is non-zero (the same argument holds if we assume that any other coordinate of  $z$  is non-zero). In other words, we assume that  $z_1 = 1$ .

Denote  $A = \{a_{ij}\}$  and consider any choice of all the elements in  $A$  except for the first column. That is, we assume that we have already chosen all the elements

$$\begin{array}{cccc} a_{12} & a_{13} & \cdots & a_{1p} \\ & \vdots & & \\ a_{q2} & a_{q3} & \cdots & a_{qp} \end{array}$$

and the only free variables left are  $a_{11}, a_{21}, \dots, a_{q1}$ .

- (a) [3 points] Argue that in order to satisfy  $Az = \bar{0}$ , these  $a_{i1}$ 's need to satisfy  $a_{i1}z_1 + a_{i2}z_2 + \cdots + a_{ip}z_p = 0 \pmod{2}$  for all  $i$ .

**Solution:** Follows from matrix multiplication.

- (b) [3 points] Conclude that for  $z_1 = 1$ , the only choice which will satisfy  $Az = \bar{0}$  is

$$\begin{aligned} a_{11} &= -(a_{12}z_2 + \cdots + a_{1n}z_p) \pmod{2} \\ &\vdots \\ a_{q1} &= -(a_{q2}z_2 + \cdots + a_{qn}z_p) \pmod{2} \end{aligned}$$

**Solution:** Substitute  $z_1 = 1$  and solve for  $a_{i1}$ , for  $1 \leq i \leq q$ , from the equations in part (a).

- (c) [3 points] Show that the probability of hitting the only value satisfying  $Az = \bar{0}$  is  $1/2^q$  and conclude that  $\mathcal{H}$  is an universal hash family from  $\mathcal{U}$  to  $\mathcal{R}$ .

**Solution:** There are  $2^q$  possibilities for  $a_{11}, \dots, a_{q1}$  that are chosen uniformly at random. Only the choice described in part (b) satisfies  $Az = \bar{0}$ , and it is chosen with probability  $1/2^q = 1/|R|$ . It follows from the definition that  $\mathcal{H}$  is a universal hash family.

- (d) [3 points] Let  $S \subseteq \mathcal{U}$  be the set we would like to hash. Let  $n = |S|$  and  $m = 2^q$ . Prove that if we choose  $h_{A,b}$  from  $\mathcal{H}$  uniformly at random, the expected number of pairs  $(x, y) \in S \times S$  with  $x \neq y$  and  $h_{A,b}(x) = h_{A,b}(y)$  is  $O(\frac{n^2}{m})$ .

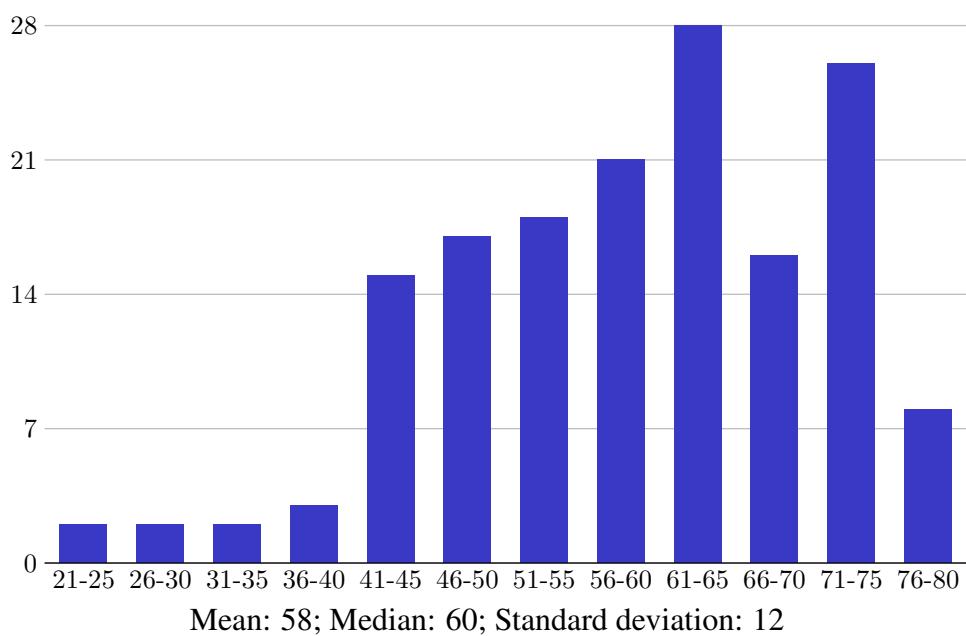
**Solution:** For any 2 distinct elements  $x, y \in S$ , let  $I_{xy}$  be the indicator variable for the event that  $h_{A,b}(x) = h_{A,b}(y)$ . Since  $\mathcal{H}$  is universal,  $E[I_{xy}] = \Pr\{h_{A,b}(x) = h_{A,b}(y)\} = \frac{1}{m}$ . The expected number of pairs  $(x, y) \in S \times S$  with  $x \neq y$  and  $h_{A,b}(x) = h_{A,b}(y)$  is therefore:

$$E\left[\sum_{x \neq y} I_{xy}\right] = \sum_{x \neq y} E[I_{xy}] = \frac{n(n-1)}{m} = O\left(\frac{n^2}{m}\right).$$

## SCRATCH PAPER

## SCRATCH PAPER

## Quiz 1 Solutions



**Problem 0. Name.** [1 point] Write your name on every page of this exam booklet! Don't forget the cover.

**Problem 1. True or False.** [24 points] (8 parts)

Circle **T** or **F** for each of the following statements to indicate whether the statement is true or false, and briefly explain why. Your justification is worth more points than your true-or-false designation.

- (a) **T F** The solution to the recurrence  $T(n) = 2^n T(n - 1)$  is  $T(n) = \Theta((\sqrt{2})^{n^2+n})$ .  
 (Assume  $T(n) = 1$  for  $n$  smaller than some constant  $c$ ).

**Solution:** [3 points] True. Let  $T(0) = 1$ .

$$\text{Then } T(n) = 2^n \cdot 2^{n-1} \cdot 2^{n-2} \dots 2^1 = 2^{(n+(n-1)+(n-2)+\dots+1)}.$$

$$\text{Because } \sum_{i=1}^n i = n(n+1)/2, \text{ we therefore have } T(n) = 2^{(n^2+n)/2} = (\sqrt{2})^{n^2+n}.$$

Some students also correctly solved the problem by using the substitution method.  
 Some students made the mistake of multiplying the exponents instead of adding them. Also, it has to be noted that  $2^{n^2/2} \neq \Theta(2^{n^2})$ .

- (b) **T F** The solution to the recurrence  $T(n) = T(n/6) + T(7n/9) + O(n)$  is  $O(n)$ .  
 (Assume  $T(n) = 1$  for  $n$  smaller than some constant  $c$ ).

**Solution:** [3 points] True. Using the substitution method:

$$\begin{aligned} T(n) &\leq cn/6 + 7cn/9 + an \\ &\leq 17cn/18 + an \\ &\leq cn - (cn/18 - an) \end{aligned}$$

This holds if  $c/18 - a \geq 0$ , so it holds for any constant  $c$  such that  $c \geq 18a$ .

Full credit was also given for solutions that uses a recursion tree, noting that the total work at level  $i$  is  $(17/18)^i n$ , which converges to  $O(n)$ .

- (c) **T F** In a simple, undirected, connected, weighted graph with at least three vertices and unique edge weights, the heaviest edge in the graph is in no minimum spanning tree.

**Solution:** [3 points] False. If the heaviest edge in the graph is the only edge connecting some vertex to the rest of the graph, then it must be in every minimum spanning tree.

- (d) **T F** The weighted task scheduling problem with weights in the set  $\{1, 2\}$  can be solved optimally by the same greedy algorithm used for the unweighted case.

**Solution:** [3 points] False. The algorithm will fail given the set of tasks (given in the form  $((s_i, f_i), w_i)$ ):  
 $\{((0, 1), 1), ((0, 2), 2)\}$ .

- (e) **T F** Two polynomials  $p, q$  of degree at most  $n - 1$  are given by their coefficients, and a number  $x$  is given. Then one can compute the multiplication  $p(x) \cdot q(x)$  in time  $O(\log n)$ .

**Solution:** [3 points] False. We need at least  $\Theta(n)$  time to evaluate each polynomial on  $x$  and to multiply the results. Some students argued incorrectly that it must take  $O(n \log n)$  using FFT, but FFT overkills because it computes all coefficients, not just one.

- (f) **T F** Suppose we are given an array  $A$  of  $n$  distinct elements, and we want to find  $n/2$  elements in the array whose median is also the median of  $A$ . Any algorithm that does this must take  $\Omega(n \log n)$  time.

**Solution:** [3 points] False. It's possible to do this in linear time using SELECT: first find the median of  $A$  in  $\Theta(n)$  time, and then partition  $A$  around its median. Then we can take  $n/4$  elements from either side to get a total of  $n/2$  elements in  $A$  whose median is also the median of  $A$ .

- (g) **T F** There is a density  $0 < \rho < 1$  such that the asymptotic running time of the Floyd-Warshall algorithm on graphs  $G = (V, E)$  where  $|E| = \rho |V|^2$  is better than that of Johnson's algorithm.

**Solution:** [3 points] False. The asymptotic running time of Floyd-Warshall is  $O(V^3)$ , which is at best the same asymptotic running time of Johnson's (which runs in  $O(VE + V \log V)$  time), since  $E = O(V^2)$

- (h) **T F** Consider the all pairs shortest paths problem where there are also weights on the vertices, and the weight of a path is the sum of the weights on the edges and vertices on the path. Then, the following algorithm finds the weights of the shortest paths between all pairs in the graph:

APSP-WITH-WEIGHTED-VERTICES( $G, w$ ):

- 1 **for**  $(u, v) \in E$
- 2     Set  $w'(u, v) = (w(u) + w(v))/2 + w(u, v)$
- 3     Run Johnson's algorithm on  $G, w'$  to compute the distances  $\delta'(u, v)$  for all  $u, v \in V$ .
- 4 **for**  $u, v \in V$
- 5     Set  $d_{uv} = \delta'(u, v) + \frac{1}{2}(w(u) + w(v))$

**Solution:** [3 points] True. Any shortest path from  $u$  to  $v$  in the original graph is still a shortest path in the new graph. For some path  $\{v_0, v_1, \dots, v_k\}$ , we have:

$$\begin{aligned}
 w'(v_0 \rightsquigarrow v_k) &= \sum_{i=0}^{k-1} ((w(v_i) + w(v_{i+1}))/2 + w(v_i, v_{i+1})) \\
 &= \sum_{i=0}^{k-1} w(v_i, v_{i+1}) + \frac{1}{2} \left( \sum_{i=0}^{k-1} w(v_i) + \sum_{i=1}^k w(v_i) \right) \\
 &= \sum_{i=0}^{k-1} w(v_i, v_{i+1}) + \sum_{i=0}^k w(v_i) - \frac{1}{2}(w(v_0) + w(v_k)) \\
 &= w(v_0 \rightsquigarrow v_k) - \frac{1}{2}(w(v_0) + w(v_k))
 \end{aligned}$$

Therefore, the order of all paths from  $v_0$  to  $v_k$  remains unchanged so Johnson's algorithm in line 3. finds the correct path, and the adjustment in line 5 finds the correct length  $d_{v_0v_k}$ .

**Problem 2. Translation** [25 points] (5 parts)

You have been hired to manage the translation process for some documentation. Unfortunately, different sections of the documentation were written in different languages:  $n$  languages in total. Your boss wants the entire documentation to be available in all  $n$  languages.

There are  $m$  different translators for hire. Some of those translators are volunteers that do not get any money for their services. Each translator knows *exactly* two different languages and can translate back and forth between them. Each translator has a non-negative hiring cost (some may work for free). Unfortunately, your budget is too small to hire one translator for each pair of languages. Instead, you must rely on chains of translators: an English-Spanish translator and a Spanish-French translator, working together, can translate between English and French. Your goal is to find a minimum-cost set of translators that will let you translate between every pair of languages.

We may formulate this problem as a connected undirected graph  $G = (V, E)$  with non-negative (i.e., zero or positive) edge weights  $w$ . The vertices  $V$  are the languages for which you wish to generate translations. The edges  $E$  are the translators. The edge weight  $w(e)$  for a translator  $e$  gives the cost for hiring the translator  $w(e)$ . A subset  $S \subseteq E$  of translators can be used to translate between  $a, b \in V$  if and only if the subgraph  $G_S = (V, S)$  contains a path between  $a$  and  $b$ . The set  $S \subseteq E$  is a translation network if and only if  $S$  can be used to translate between all pairs  $a, b \in V$ .

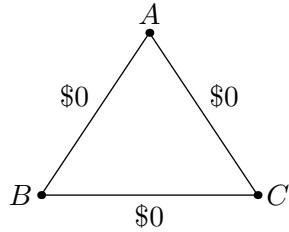
- (a) Prove that each minimum spanning tree of  $G$  is also a minimum-cost translation network.

**Solution:** [5 points] Let  $T$  be some minimum spanning tree of  $G$ . Because  $T$  is a spanning tree, there is a path in  $T$  between any pair of vertices. Hence,  $T$  is a translation network.

For the sake of contradiction, suppose that  $T$  is not a minimum-cost translation network. Then there must be some translation network  $S$  with total cost strictly smaller than  $T$ . Because  $S$  connects every pair of vertices, it must have some spanning tree  $T^*$  as a subgraph. Because all edge weights are nonnegative, we have  $w(T^*) \leq w(S) < w(T)$ . Hence,  $T$  is not the minimum spanning tree. This contradiction means that all spanning trees are also minimum-cost translation networks.

- (b) Give an example of a minimum-cost translation network that is not a minimum spanning tree of  $G$ .

**Solution:** [5 points] If the graph of translators contains a cycle of translators all willing to work for \$0, then it is possible to hire all of the translators in the cycle without increasing the overall cost of the translation network. The smallest example of this is the following:



All three MSTs of the graph have total cost \$0, so any translation network of cost \$0 has minimum cost. Hence, we can take all translators in the cycle to get a minimum-cost translation network that is not an MST.

- (c) Give an efficient algorithm that takes  $G$  as input, and outputs a minimum-cost translation network of  $G$ . State the runtime of your algorithm in terms of the number of languages  $n$  and the number of potential translators  $m$ .

**Solution:** [5 points] We saw in part (a) that every minimum spanning tree is a minimum-cost translation network of  $G$ . Hence, to find a minimum-cost translation network, it is sufficient to find a minimum spanning tree of  $G$ . We may do so using Kruskal's algorithm, for a runtime of  $\Theta(m \log n)$ , or using Prim's algorithm, for a runtime of  $\Theta(m + n \log n)$ .

Your bosses have decided that the previous approach to translation doesn't work. When attempting to translate between Spanish and Portuguese — two relatively similar languages — it degrades the translation quality to translate from Spanish to Tagalog to Mandarin to Portuguese. There are certain clusters of languages that are more closely related than others. When translating between two languages that lie within the same cluster, such as Spanish and Portuguese, the translation is of high quality when the sequence of languages used to translate between them is completely contained within the cluster.

More formally, the language set  $V$  can be divided into disjoint clusters  $C_1, \dots, C_k$ . Each cluster  $C_i$  contains languages that are fairly similar; each language is contained in exactly one cluster. Your bosses have decided that a translation between  $a, b \in C_i$  is high-quality if and only if all of the languages used on the path from  $a$  to  $b$  are also in  $C_i$ . The translator set  $S$  is a high-quality translation network if and only if it is a translation network, and for any language cluster  $C_i$  and any languages  $a, b \in C_i$ ,  $S$  can be used for a high-quality translation between  $a$  and  $b$ .

- (d) Suppose that  $S$  is a minimum-cost high-quality translation network. Let  $S_i = S \cap (C_i \times C_i)$  be the part of the network  $S$  that lies within the cluster  $C_i$ . Show that  $S_i$  is a minimum-cost translation network for the cluster  $C_i$ .

**Solution:** [5 points] Let  $S_i^*$  be a minimum-cost translation network for  $C_i$ . Because  $S$  is a high-quality translation network,  $S_i$  must contain a path between every pair of nodes in  $C_i$ , so  $S_i$  is a translation network for  $C_i$ . For the sake of contradiction, assume that  $S_i$  is not minimum-cost. Then  $w(S_i) > w(S_i^*)$ .

Consider the translation network  $S^* = (S - S_i) \cup S_i^*$ . Then  $w(S^*) = w(S) - w(S_i) + w(S_i^*) < w(S)$ . Because  $S_i^*$  is a translation network of  $C_i$ , replacing  $S_i$  with  $S_i^*$  will not disconnect any pair of vertices in the graph. Furthermore, any pair of vertices connected by a path in  $S$  that lay inside a particular cluster will be connected by a path in  $S^*$  that lies within the same cluster. Hence,  $S^*$  is a high-quality translation network with cost strictly less than  $S$ . This contradicts the definition of  $S$ , so  $S_i$  must be a minimum-cost translation network.

- (e) Give an efficient algorithm for computing a minimum-cost high-quality translation network. Analyze the runtime of your algorithm in terms of the number of languages  $n$  and the number of translators  $m$ .

**Solution:** [5 points] The idea behind this algorithm is to first compute one MST for each individual cluster, and then to compute a global MST using the remaining edges. More specifically, we do the following:

1. For each edge  $(u, v)$ , if there is some cluster  $C_i$  such that  $u, v \in C_i$ , then add  $(u, v)$  to the set  $E_i$ . Otherwise, add  $(u, v)$  to the set  $E_{global}$ .

2. For each cluster  $C_i$ , run Kruskal's algorithm on the graph  $(C_i, E_i)$  to get a minimum-cost translation network  $T_i$  for the cluster. Take the union of these minimum to get a forest  $T$ .
3. Construct an empty graph  $G_{global}$  on nodes  $\{1, \dots, k\}$ . For each edge  $(u, v)$  in  $E_{global}$ , where  $u \in C_i$  and  $v \in C_j$ , check whether the edge  $(i, j)$  is in the graph  $G_{global}$ . If so, set  $w(i, j) = \min\{w(i, j), w(u, v)\}$ . Otherwise, add the edge  $(i, j)$  to the graph  $G_{global}$ . In either case, keep a mapping  $source(i, j) = (u^*, v^*)$  such that  $w(i, j) = w(u^*, v^*)$ .
4. Run Kruskal's algorithm on the graph  $G_{global}$  to get  $T_{global}$ .
5. For each edge  $(i, j) \in T_{global}$  add the edge  $source(i, j)$  to  $T$ .

We begin by examining the runtime of this algorithm. The first step requires us to be able to efficiently discover the cluster  $C_i$  that contains each vertex, which can be precomputed in time  $\Theta(m)$  and stored with the vertices for efficient lookup. So this filtering step requires  $\Theta(1)$  lookup per edge, for a total of  $\Theta(m)$  time.

The second step is more complex. We run Kruskal's algorithm on each individual cluster. So for the cluster  $C_i$ , the runtime is  $\Theta(|E_i| \lg |C_i|)$ . The total runtime here is:

$$\sum_{i=1}^k a \cdot |E_i| \lg |C_i| \leq \sum_{i=1}^k a \cdot |E_i| \lg n = (a \lg n) \sum_{i=1}^k |E_i| \leq am \lg n$$

Hence, the total runtime for this step is  $\Theta(m \lg n)$ .

The third step also requires care. We can store the graph to allow us to efficiently lookup  $w(\cdot, \cdot)$  and to tell whether an edge  $(i, j)$  has been added to the graph. We can also store  $source(\cdot, \cdot)$  to allow for  $\Theta(1)$  lookups. So the runtime here is bounded by  $\Theta(m)$ . The fourth step is Kruskal's again, only once, on a graph with  $\leq n$  vertices and  $\leq m$  edges, so the total runtime is  $\Theta(m \lg n)$ . The final step involves a lookup for each edge  $(i, j) \in T$ , for a total runtime of  $\Theta(k) \leq \Theta(n)$ . Hence, the runtime is dominated by the two steps involving Kruskal. So the total worst-case runtime is  $\Theta(m \lg n)$ .

Next we consider the correctness of this algorithm. Suppose that the result of this process is not the minimum-cost high-quality translator network. Then there must be a high-quality translator network  $S$  that has strictly smaller cost. Because  $S$  is a minimum-cost high-quality translator network, we know that the portion of  $S$  contained in the cluster  $C_i$  is a minimum-cost translator network for  $C_i$ , which has the same cost as the minimum spanning tree for that cluster computed in step 2 of the algorithm. So the total weight of all inter-cluster edges in  $S$  must be strictly less than the total weight of all inter-cluster edges in  $T$ . But the set of all inter-cluster edges in  $T$  formed a minimum spanning tree on the cluster, so any strictly smaller set of edges cannot span the set of all clusters. So  $S$  cannot be a translation network. This contradicts our assumption, and so  $T$  must be a minimum-cost high-quality translation network.

**Problem 3. All Pairs Shortest Red/Blue Paths.** [18 points] (3 parts)

You are given a directed graph  $G = (V, E)$  with edge weights  $w : E \rightarrow \mathbb{R}$ . In addition, each edge of the graph is either red or blue. The shortest red/blue path from vertex  $i \in V$  to vertex  $j \in V$  is defined as the shortest path from  $i$  to  $j$  among those paths that go through *exactly* one red edge (if there are no such paths, the length of the shortest red/blue path is  $\infty$ ).

We can represent this graph with two  $n \times n$  matrices of edge weights,  $W_r$  and  $W_b$ , where  $W_r$  contains the weights of all red edges, and  $W_b$  contains the weights of all blue edges.

- (a) Given the Floyd-Warshall algorithm below, how would you modify the algorithm to obtain the lengths of the shortest paths that only go through blue edges?

FLOYD-WARSHALL( $W$ ):

```

1   $n = W.\text{rows}$ 
2   $D^{(0)} = W$ 
3  for  $k = 1$  to  $n$ 
4    let  $D^{(k)} = (d_{ij}^{(k)})$  be a new  $n \times n$  matrix
5    for  $i = 1$  to  $n$ 
6      for  $j = 1$  to  $n$ 
7         $d_{ij}^{(k)} = \min(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)})$ 
8  return  $D^{(n)}$ 
```

**Solution:** [6 points] In order to find shortest paths going through only blue edges, it suffices to ignore the red edges and run Floyd-Warshall on only the blue edges of the graph. We make the following changes:

- Replace each occurrence of  $W$  with  $W_b$  in lines 1 and 2.
- Replace the matrices  $D^{(k)}$  with blue versions  $D_b^{(k)}$  in lines 2, 4, and 8.
- Replace the matrix elements  $d_{ij}^{(k)}$  with blue versions  $d_{b,ij}^{(k)}$  in lines 4 and 7.

While the second and third changes above are unnecessary for this part, they lay the groundwork for future parts below.

- (b) How would you modify your algorithm from part (a) to keep track not only of shortest paths with only blue edges, but also those with exactly one red edge, and to output the lengths of the shortest red/blue paths for all pairs of vertices in this graph?

**Solution:** [6 points] Add a new set of matrices  $D_r^{(k)}$  that give lengths of shortest paths with exactly one red edge and intermediate vertices up to  $k$ . The resulting pseudocode is as follows:

RED-BLUE-FLOYD-WARSHALL( $W_r, W_b$ ):

```

1    $n = W_r.rows$ 
2    $D_b^{(0)} = W_b$ 
3    $D_r^{(0)} = W_r$ 
4   for  $k = 1$  to  $n$ 
5       let  $D_b^{(k)} = (d_{b,ij}^{(k)})$ ,  $D_r^{(k)} = (d_{r,ij}^{(k)})$  be new  $n \times n$  matrices
6       for  $i = 1$  to  $n$ 
7           for  $j = 1$  to  $n$ 
8                $d_{b,ij}^{(k)} = \min(d_{b,ij}^{(k-1)}, d_{b,ik}^{(k-1)} + d_{b,kj}^{(k-1)})$ 
9                $d_{r,ij}^{(k)} = \min(d_{r,ij}^{(k-1)}, d_{r,ik}^{(k-1)} + d_{b,kj}^{(k-1)}, d_{b,ik}^{(k-1)} + d_{r,kj}^{(k-1)})$ 
10  return  $D_r^{(n)}$ 
```

(c) Prove the correctness of your algorithm using a loop invariant.

**Solution:** [6 points] The procedure for keeping track of paths that go through only blue edges is exactly equivalent to running Floyd-Warshall on the subgraph that contains only the blue edges of  $G$ , which is sufficient to show the correctness of  $D_b^{(k)}$  for all  $k$ .

Loop invariant: at the end of every iteration of the for loop from  $k = 1$  to  $n$ , we have both the length of the shortest path for every pair  $(i, j)$  going through only blue edges and the length of the shortest path for every pair  $(i, j)$  going through exactly one red edge, in each using intermediate vertices only up to  $k$ .

Initialization: At initialization  $k = 0$ , there are no intermediate vertices. The only blue paths are blue edges, and the only paths with exactly one red edge (red/blue paths) are red edges, by definition.

Maintenance: Each iteration gets the shortest blue-edges-only path going from  $i$  to  $j$  using intermediate vertices up through  $k$  accurately, due to the correctness of the Floyd-Warshall algorithm.

For the paths including exactly one red edge, for a given pair  $(i, j)$ , there are two cases: either the shortest red/blue path from  $i$  to  $j$  using intermediate vertices through  $k$  does not go through  $k$ , or it does. If it does not go through  $k$ , then the length of this path is equal to  $d_{r,ij}^{(k-1)}$ . If it does go through  $k$ , then the red edge on this path is either between  $i$  and  $k$  or between  $k$  and  $j$ . Because of the optimal substructure of shortest paths, we can therefore break this case down into two subcases: the shortest path length is equal to  $\min(d_{r,ik}^{(k-1)} + d_{b,kj}^{(k-1)}, d_{b,ik}^{(k-1)} + d_{r,kj}^{(k-1)})$ . Line 9 in the algorithm above finds the minimum path length of these three different possibilities, so this iteration must find the shortest path length from  $i$  to  $j$  going through exactly one red edge, using only intermediate vertices through  $k$ .

Termination: After  $n$  passes through the loop, all paths with intermediate vertices up to  $n$  have been included, and as there are only  $n$  vertices, shortest paths using all vertices as intermediates will be discovered.

**Problem 4. Telephone Psetting.** [12 points] (3 parts)

Upon realizing that it was 8:30 PM on Wednesday and he had not yet started his 6.046 pset, Ben Bitdiddle found  $n - 1$  other students (for a total of  $n$  students) in the same situation, and they decided to do the pset, which coincidentally had  $n$  problems in total, together.

Their brilliant plan for finishing the pset in time was to sit in a circle and assign one problem to each student, so that for  $i = 0, 1, \dots, n-1$ , student  $i$  did problem number  $i$ , and wrote up a solution for problem  $i$  meriting  $p(i)$  points. Then, they each copied the solutions to all the other problems from the student next to them, so that student  $i$  was copying from student  $i - 1$  (and student 0 was copying from student  $n - 1$ ).

Unfortunately, they were in such a hurry that the copying chain degraded the quality of the solutions: by the time student  $i$ 's solution to problem  $i$  reached student  $j$ , where  $d = j - i \pmod{n}$ ,  $d \in \{0, 1, \dots, n - 1\}$ , the solution was only worth  $\frac{1}{d+1}p(i)$  points.

- (a) Write a formula that describes the total pset score  $S(x)$  for student  $x$ , where the total pset score is the sum of the scores that student  $x$  got on each of the  $n$  problems.

**Solution:** [3 points]  $S(x) = \sum_{i=0}^{n-1} \frac{1}{((x-i) \pmod{n}) + 1} \cdot p(i)$

Alternatively, it is also correct to give the equivalent sum:

$$S(x) = \sum_{i=0}^{n-1} \frac{1}{i+1} \cdot p((x-i) \pmod{n})$$

- (b) Describe a simple  $O(n^2)$  algorithm to calculate the pset scores of all the students.

**Solution:** [3 points] Use the formula given in part (a) to calculate each student's score. Because calculating the score requires summing  $n$  numbers, it takes  $O(n)$  time to calculate a single score, and therefore  $O(n^2)$  time to calculate all the scores.

- (c) Describe a  $O(n \log n)$  algorithm to calculate the pset scores of all the students.

**Solution:** [6 points] The formula in the solution to part (a) describes a convolution: letting  $g(y) = \frac{1}{y+1}$ , the formula in part (a) can be written as  $S : \mathbb{Z}_n \rightarrow \mathbb{R}$ , where  $S(x) = \sum_{i \in \mathbb{Z}_n} p(i)g(x-i) = (p \otimes g)(x)$ . The algorithm is as follows:

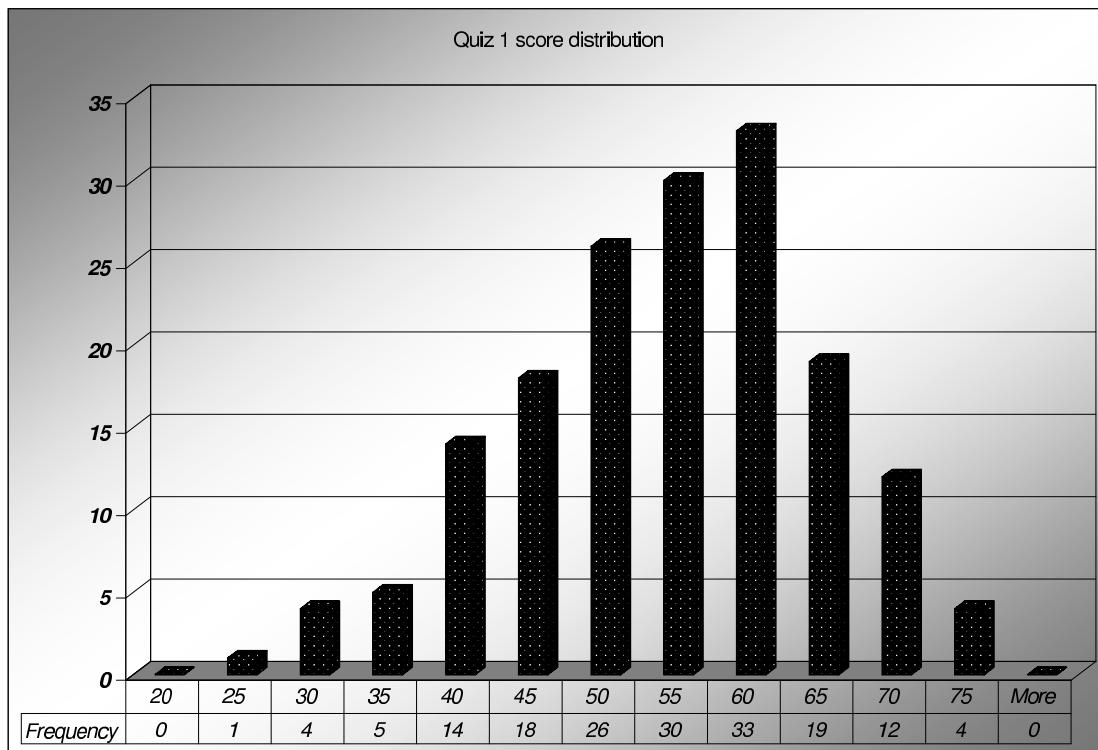
1. Apply FFT on  $p$  and  $g$  to get  $\hat{p}$  and  $\hat{g}$  (time  $\Theta(n \log n)$ ).
2. Compute the transformed convolution  $\hat{S} = \hat{p} \cdot \hat{g}$  (time  $\Theta(n)$ ).
3. Apply the inverse FFT to get back the convolution  $S$  (time  $\Theta(n \log n)$ ).

Alternatively, define two polynomials:

$P_1(x) = \sum_{i=0}^{2n-1} p(i \bmod n)x^i$  and  $P_2(x) = \sum_{i=0}^{n-1} \frac{1}{i+1}x^i$ . Then student  $j$ 's pset score is equal to the coefficient of  $x^{n+j}$  in the product of the polynomials  $P_1$  and  $P_2$ . Because we are multiplying two polynomials of degree at most  $2n - 1$ , we can apply the polynomial multiplication method seen in class, which is outlined above: apply the FFT to get the evaluations of  $P_1$  and  $P_2$  on the roots of unity of order  $2n$ , pointwise multiply, and finally apply the inverse FFT to get the coefficients of the product.

## SCRATCH PAPER

## Quiz 1 Solution



**Figure 1:** grade distribution

Problem	Points	Grade	Initials
1	16		
2	19		
3	45		
Total	80		

Name: \_\_\_\_\_

**MIT students:** Circle the name of your recitation instructor:

Bob

Chong

George

Jennifer

Rachel

**Problem 1. Match-up** [16 points]

Fill in the following table by specifying, for each algorithm, the letter of the recurrence for its running time  $T(n)$  and the numeral of the solution to that recurrence. Points will be deducted for wrong answers, so do not guess unless you are reasonably sure.

Algorithm	Recurrence	Solution
Merge sort (worst case)	d	6
Binary search (worst case)	b	2
Randomized quicksort (expected)	g	6
Strassen's algorithm (worst case)	a	3
Selection (worst case)	c	1
Randomized selection (expected)	e	1

Letter	Recurrence
a	$T(n) = 7 T(n/2) + \Theta(n^2)$
b	$T(n) = T(n/2) + \Theta(1)$
c	$T(n) = T(\lceil n/5 \rceil) + T(7n/10 + 6) + \Theta(n)$
d	$T(n) = 2 T(n/2) + \Theta(n)$
e	$T(n) = \frac{2}{n} \sum_{k=\lfloor n/2 \rfloor}^n T(k) + \Theta(n)$
f	$T(n) = 2 T(n/2) + \Theta(1)$
g	$T(n) = \frac{2}{n} \sum_{k=0}^{n-1} T(k) + \Theta(n)$
h	$T(n) = T(n - 1) + \Theta(n)$

Numeral	Solution
1	$T(n) = \Theta(n)$
2	$T(n) = \Theta(\lg n)$
3	$T(n) = \Theta(n^{\lg 7})$
4	$T(n) = \Theta(1)$
5	$T(n) = \Theta(n^2)$
6	$T(n) = \Theta(n \lg n)$

**Problem 2. Merging several sorted lists [19 points]**

Professor Fiorina uses the following algorithm for merging  $k$  sorted lists, each having  $n/k$  elements. She takes the first list and merges it with the second list using a linear-time algorithm for merging two sorted lists, such as the merging algorithm used in merge sort. Then, she merges the resulting list of  $2n/k$  elements with the third list, merges the list of  $3n/k$  elements that results with the fourth list, and so forth, until she ends up with a single sorted list of all  $n$  elements.

- (a) Analyze the worst-case running time of the professor's algorithm in terms of  $n$  and  $k$ .

**Solution:** Merging the first two lists, each of  $n/k$  elements, takes  $2n/k$  time. Merging the resulting  $2n/k$  elements with the third list of  $n/k$  elements takes  $3n/k$  time, and so on. Thus for a total of  $k$  lists, we have:

$$\begin{aligned}
 Time &= \frac{2n}{k} + \frac{3n}{k} + \cdots + \frac{kn}{k} \\
 &= \sum_{i=2}^k \frac{in}{k} \\
 &= \frac{n}{k} \sum_{i=2}^k i \\
 &= \frac{n}{k} \frac{(k+2)(k-1)}{2} \\
 &= \theta(nk)
 \end{aligned}$$

- (b) Briefly describe an algorithm for merging  $k$  sorted lists, each of length  $n/k$ , whose worst-case running time is  $O(n \lg k)$ . Briefly justify the running time of your algorithm. (If you cannot achieve  $O(n \lg k)$ , do the best you can for partial credit.)

**Solution:** There are several possible answers (only one is required). One method is to repeatedly pair up the lists, and merge each pair. This method can also be seen as a tail component of the execution merge sort, where the analysis is clear. Finally, a conceptually simple method is to store a min priority queue of the minimum elements of each of the  $k$  lists. At each step, we output the extracted minimum of the priority queue, and using satellite data determine from which of the  $k$  lists it came, and insert the next element from that list into the priority queue.

(c) Argue that there are

$$\frac{n!}{((n/k)!)^k}$$

different ways to interleave  $k$  lists, each of length  $n/k$ , into a single list of length  $n$ .

**Solution:** For the first list,

$$\text{number of ways of choosing } \frac{n}{k} \text{ elements from } n \text{ elements} = \frac{n!}{(n - \frac{n}{k})!(\frac{n}{k})!}$$

Note that of the  $(\frac{n}{k})!$  possible permutation of the elements in the list, only 1 permutation is valid because the elements in the list is already sorted. Hence the  $(\frac{n}{k})!$  term in the denominator.

For the second list,

$$\text{number of ways of choosing } \frac{n}{k} \text{ elements from the remaining } n - \frac{n}{k} \text{ elements} = \frac{(n - \frac{n}{k})!}{(n - \frac{2n}{k})!(\frac{n}{k})!}$$

And so on. Therefore, total number of ways of forming the  $k$  lists, each of size  $n/k$ , from  $n$  elements is:

$$\begin{aligned} &= \frac{n!}{(n - \frac{n}{k})!(\frac{n}{k})!} \frac{(n - \frac{n}{k})!}{(n - \frac{2n}{k})!(\frac{n}{k})!} \cdots \frac{(n - \frac{(k-2)n}{k})!}{(n - \frac{(k-1)n}{k})!(\frac{n}{k})!} \frac{(n - \frac{(k-1)n}{k})!}{(n - \frac{kn}{k})!(\frac{n}{k})!} \\ &= \frac{n!}{((n/k)!)^k} \end{aligned}$$

This is exactly the number of different ways of interleaving the  $k$  list, into the single list of  $n$  elements

- (d) Prove a lower bound of  $\Omega(n \lg k)$  on the worst-case running time of any comparison algorithm for merging  $k$  sorted lists each of length  $n/k$ . (*Hint:* Use the fact that  $(n/e)^n \leq n! \leq n^n$ .)

**Solution:** Consider the decision tree used to make the comparisons. The worst-case running time corresponds to the height of the tree:

$$\begin{aligned}
Time &= \lg \frac{n!}{((n/k)!)^k} \\
&= \lg n! - \lg((\frac{n}{k})!)^k \\
&\geq \lg n! - \lg((\frac{n}{k})^{\frac{n}{k}})^k \\
&= \lg n! - n \lg \frac{n}{k} \\
&= \lg n! - n \lg n + n \lg k \\
&\geq \lg(\frac{n}{e})^n - n \lg n + n \lg k \\
&= n \lg n - n \lg e - n \lg n + n \lg k \\
&= n \lg k - n \lg e \\
&= \Omega(n \lg k)
\end{aligned}$$

**Problem 3. True or False, and Justify** [45 points]

Circle **T** or **F** for each of the following statements, and briefly explain why. The more content you provide in your justification, the higher your grade, but be brief. Your justification is worth more points than your true-or-false designation.

- (a) **T F** A constant  $n_0 \geq 1$  exists such that for any  $n \geq n_0$ , there is an array of  $n$  elements such that insertion sort runs faster than merge sort on that input.

**Solution:** true.

If the  $n$  elements are already in sorted order, the running time for insertion sort is  $O(n)$ , whereas that of merge sort is  $O(n \lg n)$ .

- (b) **T F** Consider the algorithm from the textbook for building a max-heap:

```
BUILD-MAX-HEAP( $A$ )
1  $heap\text{-size}[A] \leftarrow length[A]$ 
2 for  $i \leftarrow \lfloor length[A]/2 \rfloor$  downto 1
3   do MAX-HEAPIFY( $A, i$ )
```

On an array of  $n$  elements, this code runs in  $\Theta(n \lg n)$  time in the worst case, because there are  $\Theta(n)$  calls to MAX-HEAPIFY, and the worst-case time for any call is  $\Theta(\lg n)$ .

**Solution:** false.

$n \lg n$  is not a tight bound. In fact, Build-Max-Heap runs in  $\Theta(n)$  time, as proven during recitation.

- (c) **T F** Given a number  $a$  and a positive integer  $n$ , the value  $a^{2^n} = a^{(2^n)}$  can be computed in  $O(n \lg n)$  time by repeated squaring. (Assume that multiplying two numbers takes constant time.)

**Solution:** true.

$$a^{2^n} = a^{2^{n-1}} a^{2^{n-1}}$$

Letting  $T(n) = a^{2^n}$ , we have:

$$\begin{aligned} T(n) &= 2T(n-1) + \Theta(1) \\ &= \Theta(n) \\ &= O(n \lg n) \end{aligned}$$

Note: it is important to understand that  $\Theta(n) = O(n \lg n)$ .

- (d) **T F** An adversary can force randomized quicksort to run in  $\Omega(n^2)$  time by providing as input an already sorted or reverse-sorted array of size  $n$ .

**Solution:** false.

The running time is not dependent on the input that the adversary provides. This is because randomized quicksort will randomly choose a pivot to partition, independent of the input.

- (e) **T F** For any integer-valued random variable  $X \geq 0$ , we have

$$\Pr\{X = 0\} \geq 1 - \mathbb{E}[X].$$

**Solution:** true

$$\begin{aligned} \Pr\{X \geq t\} &\leq \frac{\mathbb{E}[x]}{t} \\ \Pr\{X \geq 1\} &\leq \mathbb{E}[x] \end{aligned}$$

$$\begin{aligned} \Pr\{X = 0\} &= 1 - \Pr\{X \geq 1\} \\ &\geq 1 - \mathbb{E}[X] \end{aligned}$$

- (f) T F Bucket sort is a suitable auxiliary sort for radix sort.

**Solution:** true.

An auxilliary sort for radix sort must be stable. Bucket sort is stable and hence suitable for use as an auxilliary sort for radix sort.

- (g) T F Consider two implementations of a hash table with  $m$  slots storing  $n$  keys, where  $n < m$ . Let  $T_c(m, n)$  be the expected time for an unsuccessful search in the table if collisions are resolved by chaining, using the assumption of simple uniform hashing. Let  $T_o(m, n)$  be the expected time for an unsuccessful search in the table if collisions are resolved by open addressing, using the assumption of uniform hashing. Then, we have  $T_c(m, n) = \Theta(T_o(m, n))$ .

**Solution:** false.

$$\begin{aligned}T_c(m, n) &= \Theta\left(1 + \frac{n}{m}\right) \\T_o(m, n) &= \Theta\left(\frac{1}{1 - \frac{n}{m}}\right)\end{aligned}$$

In the worst case, in the chaining version, we need to search through all the values in one particular slot. On the other hand, in the open addressing version, we need to search through every slot. Hence  $T_c(m, n) \neq \Theta(T_o(m, n))$ .

- (h) T F** Let  $\mathcal{H}$  be a class of universal hash functions for a hash table of size  $m = n^3$ . Then, if we use a random  $h \in \mathcal{H}$  to hash  $n$  keys into the table, the expected number of collisions is at most  $1/n$ .

**Solution:** true.

Number of ways to choose 2 keys out of  $n$  keys is  $\frac{n(n-1)}{2}$   
Therefore, the expected number of collisions,

$$\begin{aligned} E[\text{number of collisions}] &= \frac{n(n-1)}{2} \frac{1}{m} \\ &= \frac{n(n-1)}{2} \frac{1}{n^3} \\ &\leq \frac{1}{n} \end{aligned}$$

- (i) T F** Given a set  $\{(x_1, y_1), (x_2, y_2), \dots, (x_n, y_n)\}$  of points in the plane, the  $\sqrt{n}$  points closest to the origin  $(0, 0)$  (using normal Euclidean distance) can be found in  $O(n)$  time in the worst case.

**Solution:** true.

Finding the distance of every point from the origin takes  $O(n)$ . We can then use Select to get the  $\sqrt{n}^{th}$  point. That again takes  $O(n)$ . Finally, partitioning around the  $\sqrt{n}^{th}$  point requires  $O(n)$ . Each of the steps takes  $O(n)$ . Therefore, total time is  $O(n)$

## Practice Quiz 1 Solutions

### Problem 1. Recurrences

Solve the following recurrences by giving tight  $\Theta$ -notation bounds.

(a)  $T(n) = 3T(n/5) + \lg^2 n$

**Solution:** By Case 1 of the Master Method, we have  $T(n) = \Theta(n^{\log_5(3)})$ .

(b)  $T(n) = 2T(n/3) + n \lg n$

**Solution:** By Case 3 of the Master Method, we have  $T(n) = \Theta(n \lg n)$ .

(c)  $T(n) = T(n/5) + \lg^2 n$

**Solution:** By Case 2 of the Master Method, we have  $T(n) = \Theta(\lg^3 n)$ .

(d)  $T(n) = 8T(n/2) + n^3$

**Solution:** By Case 2 of the Master Method, we have  $T(n) = \Theta(n^2 \log n)$ .

(e)  $T(n) = 7T(n/2) + n^3$

**Solution:** By Case 3 of the Master Method, we have  $T(n) = \Theta(n^3)$ .

(f)  $T(n) = T(n - 2) + \lg n$

**Solution:**  $T(n) = \Theta(n \log n)$ . This is  $\sum_{i=1}^{n/2} \lg 2i \geq \sum_{i=1}^{n/2} \lg i \geq (n/4)(\lg n/4) = \Omega(n \lg n)$ . For the upper bound, note that  $T(n) \leq S(n)$ , where  $S(n) = S(n-1) + \lg n$ , which is clearly  $O(n \lg n)$ .

**Problem 2. True or False, and Justify**

Circle **T** or **F** for each of the following statements, and briefly explain why. The better your argument, the higher your grade, but be brief. Your justification is worth more points than your true-or-false designation.

- (a) **T F** If  $f(n)$  does not belong to the set  $o(g(n))$ , then  $f(n) = \Omega(g(n))$ .

**Solution:** False. For example  $f(n) = n^{\sin(n)}$  and  $g(n) = n^{\cos(n)}$ .

- (b) **T F** A set of  $n$  integers in the range  $\{1, 2, \dots, n\}$  can be sorted by RADIX-SORT in  $O(n)$  time by running COUNTING-SORT on each bit of the binary representation.

**Solution:** False. This results in  $\Theta(\log n)$  iterations of counting sort, and thus an overall running time of  $\Theta(n \log n)$ .

- (c) **T F** An adversary can construct an input of size  $n$  to force RANDOMIZED-MEDIAN to run in  $\Omega(n^2)$  time.

**Solution:** False. The *expected* running time of RANDOMIZED MEDIAN is  $\Theta(n)$ . This applies to *any* input.

**Problem 3. Short Solution**

Give *brief*, but complete, solutions to the following questions.

- (a) Consider any priority queue (supporting INSERT and EXTRACT-MAX operations) in the comparison model. Explain why there must exist a sequence of  $n$  operations such that at least one operation in the sequence requires  $\Omega(\lg n)$  time to execute.

**Solution:** Take an array  $A$  of  $n/2$  elements. Insert the  $n/2$  elements into the priority queue, and then extract-max  $n/2$  times. This sorts the array à la Heapsort. If none of the operations in this sequence require  $\Omega(\lg n)$  time, then these  $n$  operations take  $o(n \lg n)$  time, which is impossible in the comparison model, since we've shown an  $\Omega(n/2 \lg(n/2)) = \Omega(n \lg n)$  lower bound for any sorting algorithm.

- (b) Suppose that an array  $A$  has the property that only adjacent elements might be out of order—i.e., if  $i < j$  and  $A[i] > A[j]$ , then  $j = i + 1$ . Which of INSERTION-SORT or MERGE-SORT is a better algorithm to sort the elements of  $A$ ? Justify your choice.

**Solution:** INSERTION-SORT is a better choice: since its running time depends on the number of inversions in the array. We may swap each element with the element immediately to its left, but that's all that we do in INSERTION-SORT. So INSERTION-SORT is  $O(n)$  in this case, while MERGE-SORT remains  $\Theta(n \log n)$ .

- (c) Recall from lecture that for  $n$ -by- $n$  matrices  $A$ ,  $B$ , and  $C$ , we can probabilistically test whether  $AB = C$  by choosing a random 0-1 vector  $x$  and checking whether  $ABx = Cx$ . In lecture we showed a lower bound of  $p \geq 1/2$  on the probability  $p$  that this test is successful as follows. For a matrix  $M$ , let  $M_{ij}$  denote the  $ij$ th element of  $M$ . We argued that if  $(AB)_{ij} \neq C_{ij}$ , then at most one of the two values for  $x_i$  can make the equation

$$\sum_{k=1}^n (AB)_{kj} x_k = \sum_{k=1}^n C_{kj} x_k$$

hold.

Suppose that instead we choose a random vector  $x$  where each element of  $x$  is chosen independently at random from the set  $\{0, 1, 2, 3\}$  instead of from  $\{0, 1\}$ , and perform the same test. What lower bound on  $p$  can be deduced in this case?

**Solution:** The probability of success was  $1/2$  if  $AB \neq C$ , since at most one of the two values for  $x_i$  can make  $(AB)^{(j)}x = C^{(j)}x$  if  $(AB)_{i,j} \neq C_{i,j}$ . So, similarly, at most one of the four values for  $y_i$  can do the same, so the same analysis shows that the probability of success is at least  $3/4$ . In either case, if  $AB = C$ , then we will always produce the correct solution.

**Problem 4. Anagram pattern matching (22 points)**

Assume you are given a *text* array  $T[1 \dots n]$  containing letters from the standard Latin alphabet. In other words,  $T[i] \in \{a, b, \dots, z\}$  for  $i = 1 \dots n$ . In addition, you are given a *pattern* array  $P[1 \dots m]$ ,  $m < n$ , which also contains letters from the Latin alphabet. For any sub-array  $T_i^m = T[i \dots i + m - 1]$  of  $T$ , we say that  $T_i^m$  is an *anagram* of  $P$  if there is a way of permuting symbols in  $T_i^m$  so that the resulting array is equal to  $P$ .

- (a) Give an algorithm that, given an index  $i$  (and  $m$ ), determines whether  $T_i^m$  is an *anagram* of  $P$ . Try to give an algorithm that is as efficient as possible. However, partial credit will also be given for less efficient solutions.

**Solution:** The two  $m$ -element sequences  $P$  and  $T_i^m$  will be anagrams of each-other if and only if their sorted orderings are equal. Based on this observation, our algorithm is to sort both  $P$  and  $T_i^m$ , and to compare the resulting sorted orderings. Since the elements from these sequences come from a constant-size alphabet, we can use counting sort to sort both of them in  $\Theta(m)$  time. Afterwards, it takes  $\Theta(m)$  time to compare the sorted orderings. The total running time is therefore  $\Theta(m)$ .

- (b) Design an algorithm, which given  $T$  and  $P$  as an input, reports all  $i$ 's such that  $T_i^m$  is an anagram of  $P$ . Ideally, your algorithm should run in  $O(n + m)$  time. However, partial credit will also be given for less efficient solutions.

**Solution:** Start with  $i = 1$ . Let us count the number of occurrences of each letter in  $P$  and in  $T_i^m$ , just as is done in the first stage of the counting sort algorithm. Specifically, we will scan through the elements of  $P$  and construct an array  $C[a \dots z]$  of counts, such that  $C[l]$  gives the number of times the letter  $l$  appears in the array  $P$ . Similarly, we create an array  $D[a \dots z]$  which contains the number of occurrences of each letter in  $T_i^m$ . Construction of the arrays  $C$  and  $D$  will consume  $\Theta(m)$  time, as it requires a single linear scan over  $P$  and  $T_i^m$ . Note that we can compare the arrays  $C$  and  $D$  in  $\Theta(1)$  time, as the arrays have constant size. If  $C = D$ , then  $P$  will be an anagram of  $T_i^m$ .

Now let us loop over all values of  $i$ . For each value of  $i$ , we check, in  $\Theta(1)$  time, if  $C = D$ , and if so, we output that value of  $i$  since  $T_i^m$  will be an anagram of  $P$ . When we move from  $i$  to  $i+1$ , we will update the array  $D$  by decrementing  $D[T[i]]$  (since the letter  $T[i]$  will no longer be present in  $T_{i+1}^m$ ) and by incrementing  $D[T[i+m]]$  (since  $T[i+m]$  will now be present in  $T_{i+1}^m$ ). In total, we spend  $\Theta(1)$  time per iteration of our loop over  $i$ , for a total running time of  $\Theta(n)$  plus the initial cost  $\Theta(m)$  of constructing  $C$  and  $D$  for  $i = 1$ . Therefore, total running time is  $\Theta(m + n)$ .

**Problem 5. Mean 6.042 instructors**

There are two types of professors who have taught 6.042: ***nice*** professors and ***mean*** professors. The nice professors assign A's to all of their students, and the mean professors assign A's to **exactly** 75% of their students and B's to the remaining 25% of the students. For example, for  $n = 8$ , the arrays [A A A B A B A A] and [A B B A A A A] represent grades assigned by mean professors, and the array [A A A A A A A A] represents grades assigned by a nice professor. Given an array  $G[1 \dots n]$  of grades from 6.042, we wish to decide whether the professor who assigned the grades was nice or mean.

Give an efficient **randomized** algorithm to decide whether a given array  $G$  represents grades assigned by a mean or nice professor. Your algorithm should be correct with probability at least 51%.

**Solution:** Here's the algorithm:

1. Repeat the following  $t$  times:

- (a) choose a random index  $i \in \{1, \dots, n\}$ .
- (b) if  $G[i] = B$ , then return "mean"

2. Return "nice".

First, note that with probability one, a nice professor will be noted as such, since there are no  $B$ 's in  $G$  in this case.

The probability that a mean professor makes it through an iteration of the loop is  $3/4$ , since  $1/4$  of the entries of array are  $B$ 's. The probability that the mean professor makes it through  $t$  iterations is  $(3/4)^t$ . If we take  $t$  to be 3, then  $(3/4)^3 < 0.49$ . Thus the probability of giving a correct solution is at least 51%.

The running time is obviously  $O(1)$ , since we do a constant number of iterations of constant-time loop.

**Comment:** The above arguments show that

- If a professor is nice, the probability of an incorrect solution is 0
- If a professor is mean, the probability of an incorrect solution is  $\leq 49\%$

However, in several cases, people were making "reverse" statements, e.g.:

"If we get an A, then the probability that the professor is mean is ..."

Although intuitive, a closer inspection reveals that such a statement does not have any real meaning. This is because there is *no* probability distribution determining if the professor is mean or not. In case of randomized algorithms, the input is *not* random. Instead, for *any* input, the algorithm should report the correct solution with at least certain probability.

## Practice Quiz 1

- Do not open this quiz booklet until you are directed to do so. Read all the instructions first.
- When the quiz begins, write your name on every page of this quiz booklet.
- The quiz contains 5 multi-part problems. You have 80 minutes to earn 80 points.
- This quiz booklet contains 10 pages, including this one. Two extra sheets of scratch paper are attached. Please detach them before turning in your quiz.
- This quiz is closed book. You may use one handwritten A4 or  $8\frac{1}{2}'' \times 11''$  crib sheet. No calculators or programmable devices are permitted.
- 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, since the pages may be separated for grading.
- Do not waste time and paper rederiving facts that we have studied. It is sufficient to cite known results.
- 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.
- 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.
- Good luck!

Problem	Points	Grade	Initials
1	12		
2	12		
3	30		
4	13		
5	13		
Total	80		

Name: Solutions \_\_\_\_\_

**Problem 1. Recurrences [12 points]**

Solve the following recurrences. Give tight, i.e.  $\Theta(\cdot)$ , bounds.

(a)  $T_1(n) = 5 T_1(n/2) + \sqrt{n}$

**Solution:** We use the Master Theorem: note that  $\log_2 5 > 1/2 + \epsilon$ , so the solution is  $T_1(n) = \Theta(n^{\log_2 5})$ .

(b)  $T_2(n) = 64 T_2(n/4) + 8^{\lg n}$

**Solution:** First, notice that  $8^{\lg n} = n^{\lg 8}$  (this can be seen by taking  $\lg$  of both sides). Since  $\log_4 64 = \lg 8 = 3$ , we use case (ii) of the Master Theorem:  $T_2(n) = \Theta(n^3 \log n)$ .

(c) Set up the recurrence for Strassen's matrix multiplication algorithm and solve it.

**Solution:** The recurrence is:  $T(n) = 7T(n/2) + c \cdot n$ .

We can apply the Master Method. We calculate  $n^{\log_b a} = n^{\log_2 7}$ . Since  $f(n) = c \cdot n$  and  $f(n) = O(n^{\log_2 7})$ , we can use Case 1. Thus,  $T(n) = \Theta(n^{\log_2 7})$ .

**Problem 2. Short Answer [12 points]**

Give *brief*, but complete, answers to the following questions.

- (a) Briefly describe the difference between a *deterministic* and a *randomized* algorithm, and name two examples of algorithms that are not deterministic.

**Solution:** On identical inputs, a deterministic algorithm always performs exactly the same computations and returns the same output. A randomized algorithm is one which “flips coins,” i.e. one which makes random choices that may cause it to perform different computations, even on the same input. RANDOMIZED-QUICKSORT and RANDOMIZED-SELECT are two examples of non-deterministic algorithms.

- (b) Describe the difference between *average-case* and *worst-case* analysis of deterministic algorithms, and give an example of a deterministic algorithm whose average-case running time is different from its worst-case running time.

**Solution:** An average-case analysis assumes some distribution over the inputs (e.g., uniform), and computes the expected (average) running time of an algorithm subject to that distribution. A worst-case analysis considers those inputs which force an algorithm to run for the longest amount of time, and computes the running time under those inputs. QUICKSORT has a worst-case running time of  $\Theta(n^2)$  (on an already-sorted or reverse-sorted array), but has an average-case running time of  $\Theta(n \log n)$  (assuming all input permutations are equally likely).

- (c) If you can multiply 4-by-4 matrices using 48 scalar multiplications, can you multiply  $n \times n$  matrices asymptotically faster than Strassen’s algorithm (which runs in  $O(n^{\lg 7})$  time)? Explain your answer.

**Solution:** Our algorithm breaks an  $n \times n$  matrix into a 4-block by 4-block matrix (where each block is  $n/4 \times n/4$ ). It then multiplies the appropriate blocks using 48 recursive calls (corresponding to the scalar multiplications) and combines their products. Our new algorithm’s running time is  $T(n) = 48T(n/4) + \Theta(n^2)$ , which is  $O(n^{\log_4 48})$  by the Master Theorem. For comparison with Strassen’s algorithm, note that  $\log_2 7 = \log_{2^2} 7^2 = \log_4 49$ . Therefore our algorithm is asymptotically better.

**Problem 3. True or False, and Justify [30 points]**

Circle **T** or **F** for each of the following statements, and briefly explain why. The better your argument, the higher your grade, but be brief. Your justification is worth more points than your true-or-false designation.

- (a) **T F** Every comparison-based sort uses at most  $O(n \log n)$  comparisons in the worst case.

**Solution:** False. INSERTION-SORT, for example, uses  $\Theta(n^2) \neq O(n \log n)$  comparisons in the worst-case (a reverse-sorted array). The statement would be true if it read “...at least  $\Omega(n \log n)$  comparisons in the worst case.”

- (b) **T F** RADIX-SORT is stable if its auxiliary sorting routine is stable.

**Solution:** True. If two numbers are equal, then they have the same digits. Each intermediate sort is stable, so the two equal numbers never change relative positions.

- (c) **T F** It is possible to compute the smallest  $\sqrt{n}$  elements of an  $n$ -element array, in sorted order, in  $O(n)$  time.

**Solution:** True. We can SELECT the  $\sqrt{n}$ th smallest element and partition around it, then sort those  $\sqrt{n}$  elements in  $O(n)$  time. Alternately, we can build a min-heap in  $O(n)$  time and call EXTRACT-MIN  $\sqrt{n}$  times, for a total runtime of  $O(n + \sqrt{n} \log n) = O(n)$ .

Some incorrect solutions amounted to: “we must do  $\sqrt{n}$  order statistic queries, each of which take  $O(n)$  time, for a total running time of  $O(n\sqrt{n})$ .” However, this argument does not preclude us from coming up with a more clever algorithm (like the one above) that is more efficient. In fact, a similar argument would “prove” that sorting must take  $\Omega(n^2)$  time (despite the existence of MERGESORT etc.), because we must do  $n$  order statistic queries!

- (d) **T F** Consider hashing the universe  $U = \{0, \dots, 2^r - 1\}$ ,  $r > 2$ , into the hash table  $\{0, 1\}$ . Consider the family of hash functions  $\mathcal{H} = \{h_1, \dots, h_r\}$ , where  $h_i(x)$  is the  $i$ th bit of the binary representation of  $x$ . Then  $\mathcal{H}$  is universal.

**Solution:** False. Take  $x = 0, y = 1$ . Then all of  $x$ 's binary digits are the same as  $y$ 's, except for the least significant one. Thus  $\Pr_{h \in \mathcal{H}}[h(x) = h(y)] = (r-1)/r > 1/2$ . If  $\mathcal{H}$  were universal, the probability would be at most  $1/2$ .

Some incorrect solutions said that the probability of a collision, taken over random choices of  $h$ ,  $x$ , and  $y$  is  $1/2$ . This is true, but the universality condition demands something stronger: it says that for *every* fixed (distinct) pair  $x, y$ , their probability of collision over the random choice of  $h$  must be at most  $1/2$ .

- (e) **T F** RANDOMIZED-SELECT can be forced to run in  $\Omega(n \log n)$  time by choosing a bad input array.

**Solution:** False. RANDOMIZED-SELECT runs in expected  $O(n)$  time; the only way it can take longer is if its random choices of pivots are unlucky. The input array cannot force these unlucky choices.

- (f) **T F** For every two functions  $f(n)$  and  $g(n)$ , either  $f(n) = O(g(n))$  or  $g(n) = O(f(n))$ .

**Solution:** False. Let  $f(n) = \sin n$  and  $g(n) = \cos n$ ; then neither case holds. Another example is  $f(n) = \sqrt{n}$  and  $g(n) = n^{\sin n}$ . Finally, one could let  $f(n)$  and  $g(n)$  be any strictly-negative functions; by a technical condition of the definition,  $f(n)$  must be at least 0 to be  $O(g(n))$ .

Many incorrect answers argued that one of the statements  $f(n) \leq cg(n)$ ,  $f(n) = cg(n)$ , or  $g(n) \leq cf(n)$  must be true. This is correct for any *particular* value of  $n$ , but it doesn't mean that the *same* statement is true for *all* sufficiently large values of  $n$ , which is the condition needed in the definition of big- $O$ .

- (g) **T F** Let  $H$  be a universal hash family mapping keys into a table of size  $m = n^2$ . Then, if we use random  $h \in H$  to hash  $n$  keys into the table, the expected number of collisions is at most  $1/n$ .

**Solution:** False. The probability that any two elements hash to the same value is at most  $1/n^2$ . We want to calculate  $E[X] = E[\sum_{i < j} X_{ij}]$  where the indicator random variable  $X_{ij}$  is a 1 if there is a collision between elements  $i$  and  $j$  and 0 otherwise. Thus,  $X_{ij}$  is 1 with probability at most  $1/n^2$  and 0 otherwise.  $E[X_{ij}] \leq 1/n^2$ . So, we have  $E[X] \leq \sum_{i < j} X_{ij} = \sum_{i < j} 1/n^2 = 1/n^2 \cdot \binom{n}{2} < 1/2$ .

- (h) **T F** The following array  $A$  is a max-heap:

30 25 7 18 24 8 4 9 12 22 5

**Solution:** False. See that  $7 = A[3] < A[2 \cdot 3] = 8$ , which is a violation of the max-heap property.

- (i) **T F** Suppose we use HEAPSORT instead of INSERTION-SORT as a subroutine of BUCKET-SORT to sort  $n$  elements. Then BUCKET-SORT still runs in average-case linear time, but its worst-case running time is now  $O(n \log n)$ .

**Solution:** True. Even if all the elements land in the same bucket (the worst-case input), HEAPSORT sorts them in  $O(n \log n)$  time.

- (j) **T F** If memory is limited, one would prefer to sort using HEAPSORT instead of MERGESORT.

**Solution:** True. MERGESORT is not in-place, which means it requires an auxiliary array as big as the input. HEAPSORT is in-place, which means it only uses  $O(1)$  auxiliary space.

**Problem 4.** Mode finding [16 points]

Assume that you are given an array  $A[1 \dots n]$  of distinct numbers. You are told that the sequence of numbers in the array is *unimodal*, i.e., there is an index  $i$  such that the sequence  $A[1 \dots i]$  is increasing (i.e.  $A[j] < A[j + 1]$  for  $1 \leq j < i - 1$ ) and the sequence  $A[i \dots n]$  is decreasing. The index  $i$  is called the *mode* of  $A$ .

Show that the mode of  $A$  can be found in  $O(\log n)$  time.

**Solution:** The most common solution was to modify binary search to look at the *pair* of successive middle elements of the array.

If the array is only a single element, that element is the mode. If the successive elements are increasing, we know that our elements are in the increasing portion of the array. Thus, the mode will be found to the right of the pair, and we recurse on that half of the array. Otherwise, we know the pair is in the decreasing portion of the array, and the mode will be found to the left. (Assuming indices increase from left to right.)

```

FIND_MODE( $A$ )
1 if ( $\text{length}(A) = 1$ ) then return 1
2  $mid \leftarrow \lfloor \text{length}(A)/2 \rfloor$ 
3 if  $A[mid] < A[mid + 1]$ 
4   then return FIND_MODE( $A[1 \dots mid]$ )
5   else return  $mid + \text{FIND\_MODE}(A[mid + 1 \dots \text{length}(A)])$ 
```

To compute the running time we note FIND\_MODE employs a divide and conquer strategy. The divide step requires constant work to compute the middle index of the array. The combine step require constant work to compute the return value. The conquer step recurses on half the array, yielding the recurrence  $T(n) = T(n/2) + \Theta(1)$ , which implies  $T(n) = \Theta(\log n)$ , as needed.

**Problem 5. Assigning Grades [13 points]**

It is the not-too-distant-future, and you are a computer science professor at a prestigious north-eastern technical institute. After teaching your course, “6.66: Algorithms from Hell,” you have to assign a letter grade to each student based on his or her unique total score. (Scores can only be compared to each other.) You are grading on a curve, and there are a total of  $k$  different grades possible. You want to rearrange the students into  $k$  equal-sized groups, such that everybody in the top group has a higher score than everybody in the second group, etc. However, you don’t care how the students are ordered within each group (because they will all receive the same grade).

- (a) Describe and analyze a simple algorithm that takes an unsorted  $n$ -element array  $A$  of scores and an integer  $k$ , and divides  $A$  into  $k$  equal-sized groups, as described above. Your algorithm should run in time  $O(nk)$ . (If you find a faster algorithm, see part (c).) You may assume that  $n$  is divisible by  $k$ . *Note:*  $k$  is an input to the algorithm, not a fixed constant.

**Solution:** [5 points] Our algorithm first uses SELECT to find the  $n/k$ th order statistic, then partitions around it. At this point, the first  $n/k$  elements of the array form the bottom group. Then it uses SELECT to find the  $n/2k$ th order statistic of the remainder of the array, and partitions around it, etc., until all the groups have been separated. Each SELECT and PARTITION requires linear time in the number of remaining elements, which is at most  $n$ , so the running time is  $O(nk)$ .

- (b) In the case that  $k = n$ , prove that any algorithm to solve this problem must run in time  $\Omega(n \log k)$  in the worst case. Recall that we are only considering comparison-based algorithms, i.e., algorithms that only compare scores to each other as a way of finding information about the input. *Hint:* There is a very short proof.

**Solution:** [3 points] Any algorithm for this problem can fully sort an array of  $n$  elements if we provide it with an input where  $k = n$ . Since sorting requires  $\Omega(n \lg n)$  comparisons in the worst case, the algorithm must run in time  $\Omega(n \lg n) = \Omega(n \lg k)$  in the worst case.

- (c) Now describe and analyze an algorithm for this problem that runs in time  $O(n \log k)$ . You may also assume that  $k$  is a power of 2, in addition to assuming that  $n$  is divisible by  $k$ .

**Solution:** [5 points] We use a recursive algorithm GROUP, which takes an array and a value  $k$ , and works as follows: if  $k = 1$ , return. Otherwise, SELECT and PARTITION around the median of the array. Then call GROUP on the lower half of the array with  $k/2$ , and again on the top half with  $k/2$ .

To see that this works, note that after partitioning, all grades in the upper half of the array are greater than those in the lower half. By induction, the two recursive calls divide each half into  $k/2$  groups, for a total of  $k$  groups. Finally, note that the base case satisfies the problem statement.

We now analyze the running time: the recurrence describing the algorithm's running time is  $T(n, k) = 2T(n/2, k/2) + \Theta(n)$  because SELECT and PARTITION are linear-time. The base case of the recurrence is  $T(n, 1) = \Theta(1)$  for any  $n$ . Therefore the recurrence tree does  $\Theta(n)$  work at each level, and has  $\lg k$  levels, for a total running time of  $\Theta(n \log k)$ .

Some students correctly observed that this solution is essentially an “early quitting” QUICKSORT, where the pivot is always chosen to be the median, and the algorithm terminates once the recursion depth reaches  $\lg k$ .

**SCRATCH PAPER** — Please detach this page before handing in your quiz.

**SCRATCH PAPER** — Please detach this page before handing in your quiz.

## Quiz 1

- Do not open this quiz booklet until you are directed to do so. Read all the instructions first.
- When the quiz begins, write your name on every page of this quiz booklet.
- The quiz contains five multi-part problems. You have 120 minutes to earn 120 points.
- This quiz booklet contains **14** pages, including this one. An extra sheet of scratch paper is attached. Please detach it before turning in your quiz.
- This quiz is closed book. You may use one handwritten A4 or  $8\frac{1}{2}'' \times 11''$  crib sheet. No calculators or programmable devices are permitted.
- Write your solutions in the space provided. Extra scratch paper may be provided if you need more room, although your answer should fit in the given space.
- Do not waste time and paper re-deriving facts that we have studied. It is sufficient to cite known results.
- 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. Generally, a problem's point value is an indication of how much time to spend on it.
- 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.
- Good luck!

Problem	Points	Grade	Initials
1	20		
2	15		
3	30		
4	25		
5	30		
Total	120		

Name: Solutions \_\_\_\_\_  
Circle your recitation:

Brian 11

Brian 12

Jen 12

Jen 1

Brian 2

**Problem 1. Recurrences** [20 points] (5 parts)

For each of the following algorithms in parts (a)-(c), provide a recurrence relation and give an asymptotically tight ( $\Theta(\cdot)$ ) bound. For parts (d)-(e), just provide an asymptotically tight bound. Justify your answer by naming the particular case of the Master Method, by iterating the recurrences, using the substitution method, or using Akra-Bazzi.

**Example:** [0 points] BINARY SEARCH

Recurrence:  $T(n) = T(n/2) + c$

Solution by iteration:

$$T(n) = T(n/4) + c + c = \sum_{i=0}^{\log n} c = c \log n = \Theta(\log n)$$

## (a) [4 points] MERGE SORT

**Solution:** Recurrence:  $T(n) = 2T(n/2) + \Theta(n)$ .

$T(n) = \Theta(n \log n)$  by part 2 of the Master Method.

## (b) [4 points] KARATSUBA INTEGER MULTIPLICATION

(The Divide and Conquer algorithm from Lecture 2 and Problem Set 1-4.)

**Solution:** Recurrence:  $T(n) = 3T(n/2) + \Theta(n)$

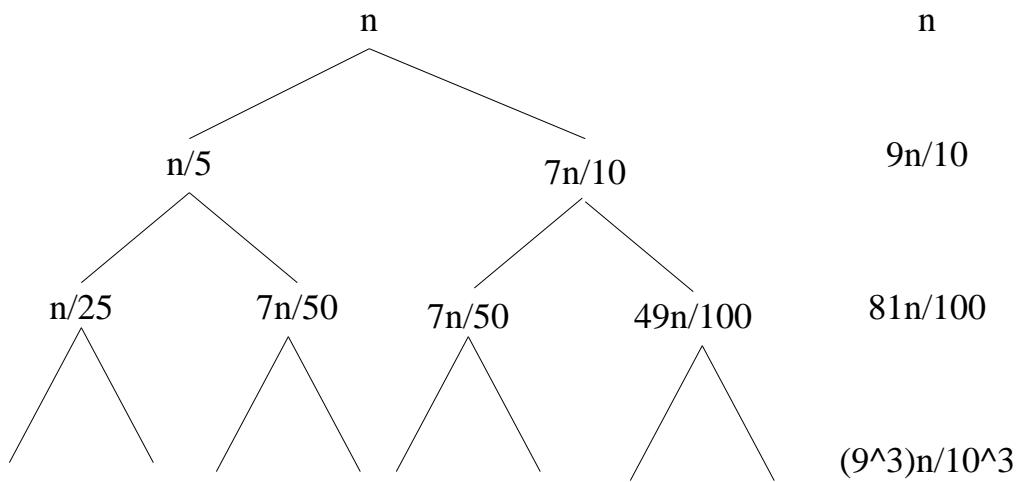
$T(n) = \Theta(n^{\log_2 3})$  by part 1 of the Master Method.

- (c) [4 points] DETERMINISTIC SELECT (Code provided on the last page of this exam.)

**Solution:** Recurrence:  $T(n) \leq T(n/5) + T(7n/10) + \Theta(n)$

Using a recursion tree method, depicted in Figure 1, we deduce that:

$$T(n) = \left( \sum_{i=0}^{\log_5 n} \left(\frac{9}{10}\right)^i n \right) = \Theta(n)$$



**Figure 1:** Recursion tree for Deterministic Select

(d) [4 points]  $T(n) = 7T(n/2) + n^3$ .

**Solution:**

$T(n) = \Theta(n^3)$  by part 3 of the Master Method.

(e) [4 points]  $T(n) = 2T(n/4) + 5\sqrt{n}$ .

**Solution:**  $T(n) = \Theta(\sqrt{n} \log n)$  by part 2 of the Master Method.

**Problem 2. True or False, and Justify** [15 points] (5 parts)

Circle **T** or **F** for each of the following statements, and briefly explain why. The better your argument, the higher your grade, but be brief. Your justification is worth more points than your true-or-false designation.

- (a) **T F** [3 points] RADIX SORT works correctly if we sort each individual digit with HEAPSORT instead of COUNTING SORT.

**Solution:** False. HEAPSORT is not stable. If the first  $i - 1$  digits are sorted in RADIX SORT, sorting the  $i$ th digit using stable COUNTING SORT won't change the order of two items with the same  $i$ th digit. This cannot be guaranteed with HEAPSORT

- (b) **T F** [3 points] The following array  $A$  is a Min-Heap:

1 11 3 12 100 5 10 13 50

**Solution:** True.  $\forall i (A[i] \leq A[2i]) \wedge (A[i] \leq A[2i + 1])$

- (c) **T F** [3 points] RADIX-SORT can be used to sort  $n$  elements with integer keys in the range from 1 to  $n^{f(n)}$  in  $O(nf(n))$  time.

**Solution:** True. By using base  $n$  representations, each COUNTING-SORT takes  $O(n)$  time, and there are a total of  $\log_n n^{f(n)} = f(n)$  digits, for a runtime of  $O(nf(n))$  time.

- (d) **T F** [3 points] There is an  $\Omega(n \log n)$  lower bound to build a Min-Heap of size  $\Theta(n)$  in the comparison model.

**Solution:** False. Using the BUILD HEAP procedure from CLRS, we can build a new heap in  $\Theta(n)$  time.

- (e) **T F** [3 points] If we only assume that all buckets have the same size, BUCKET SORT runs in  $O(n)$ -time on average independent of the input distribution.

**Solution:** False. By picking an input distribution that overloaded a single bucket, you could force BUCKET SORT to run in the worst-case time of it's subroutine sorting algorithm ( $O(n^2)$  for INSERTION SORT).

**Problem 3. Short Answer** [30 points] (3 parts)

Give *brief*, but complete, answers to the following questions.

- (a) [5 points] Describe the difference between *average-case* and *worst-case* analysis of algorithms, and give an example of an algorithm whose average-case running time is different from its worst-case running time.

**Solution:** An average-case analysis assumes some distribution over the inputs (e.g., uniform), and computes the expected (average) running time of an algorithm subject to that distribution. A worst-case analysis considers those inputs which force an algorithm to run for the longest amount of time, and computes the running time under those inputs. QUICKSORT has a worst-case running time of  $\Theta(n^2)$  (on an already-sorted or reverse-sorted array), but has an average-case running time of  $\Theta(n \log n)$  (assuming all input permutations are equally likely).

(b) [10 points] Suppose you are given an array  $A$  of size  $n$  that either contains all zeros or  $2n/3$  zeros and  $n/3$  ones in some arbitrary order. Your problem is to determine whether  $A$  contains any ones.

1. Give an exact lower bound in terms of  $n$  (not using asymptotic notation) on the worst-case running time of any deterministic algorithm that solves this problem.
2. Give a randomized algorithm that runs in  $O(1)$ -time and gives the right answer with probability at least  $1/3$ .
3. Give a randomized algorithm that runs in  $O(1)$ -time and gives the right answer with probability at least  $5/9$ .

**Solution:**

1. Any correct deterministic algorithm must look at  $2n/3 + 1$  entries, because if it didn't, it could see all zeros even when there was a one somewhere.
2. Pick a random location. If it is a 1, output "has ones". Otherwise, output "doesn't have ones". Clearly the algorithm runs in  $O(1)$  time. If the array is all zeros, it is always correct.

However if the array contains  $1/3$  ones, the algorithm may make a mistake. Since there are  $n/3$  ones, you will select a 1 value at random and correctly output "has ones" with probability  $1/3$ .

3. Pick two random locations. The correctness and  $O(1)$  runtime are identical as above. Again, if the array is all zeros, the algorithm is always correct.

A mistake arises if the array contains ones, but the algorithm picks two zeros. Picking a single zero occurs with probability  $2/3$ , and picking two zeros independently occurs with probability  $(2/3)^2 = 4/9$ . Therefore, the probability that the algorithm is correct is  $1 - (4/9) = 5/9$ .

- (c) [15 points] A *k-multiset* is a set of  $n$  elements in which  $k$  distinct elements each appear exactly  $n/k$  times. For example,  $\{1, 1, 2, 2, 3, 3, 4, 4, 5, 5\}$  is a sorted 5-multiset of size  $n = 10$ . Give an  $\Theta(n \log k)$ -time algorithm to sort a *k-multiset* of size  $n$ . For convenience, you may assume that  $n = 2^i$  and  $k = 2^j$  for some  $i \geq j$ .

**Solution:** On the *k-multiset*  $S$ , do the following:

1. Use DETERMINISTIC SELECT to find the median  $s_m$  of  $S$ .
2. Partition  $S$  around  $s_m$ .
3. Since there are  $n/k$  values equal to  $s_m$ , the partition may not split the set exactly in half. Extract all values equal to  $s_m$  from both sides of the partition and add them to the “lighter” of the two sublists.
4. Recurse on each subproblem if it has size greater than  $n/k$ .

Each level of recursion will split the set  $S$  into two subproblems of size  $S/2$  in  $O(n)$  time. This algorithm will halt with subproblems of size  $n/k$ . Since  $n/k = n/2^j$ , this algorithm will recurse exactly  $j = \log k$  levels before halting on sublist leaves of size exactly  $n/k$ .

Consider the “least” leaf produced by our partitions. It will contain the  $n/k$  least elements of the array, which all happen to be the same value. Similarly, the next “least” leaf will contain the next  $n/k$  least elements, which are also all the same value. Inductively, we can see that each  $n/k$ -sized leaf is homogeneous. After  $j = \log k$  levels of recursion, the algorithm can just output its leafs in order.

$O(n)$  work is done on each level of recursion and there are  $\log k$  levels of recursion, so this algorithm takes  $O(n \log k)$  total time.

**Problem 4. Gaagle's Cappuccino Machine** [25 points] (1 parts)

You've been hired by the high-tech start-up Gaagle to find an ideal location for their new cappuccino machine. Gaagle's offices are arranged at various points along a single long corridor. Gaagle wishes to minimize the total distance walked by all their employees. You may assume the following:

- Each employee visits the cappuccino machine exactly the same number of times each day.
- Gaagle has an odd number of employees.

Describe where you should place the cappuccino machine and prove that it is optimal.

**Solution:**

Suppose Gaagle's employees occupy offices  $(p_1, \dots, p_n)$ . Considering a particular placement  $x$ , let's consider the sets  $L = \{p_i | p_i < x\}$  and  $G = \{p_i | p_i > x\}$  would result in the employees having to walk:

$$\sum_{p_i \in L} (x - p_i) + \sum_{p_i \in G} (p_i - x)$$

Suppose  $x$  is the median office location  $p_m$ . For convenience let  $d = p_{m+1} - p_m$ . If we changed  $x$  to be  $p_{m+1}$ , it would result in each of the  $(n-1)/2$  person in  $L$  and the person at  $p_m$  having to walk an additional  $d$  distance. Each of the  $(n-1)/2$  people in  $G$  would have to walk  $d$  less distance. Thus, moving the machine from the median results in a net gain of  $d$  distance walked.

The same argument holds if we move  $x$  to  $p_{m-1}$ . Therefore, since a move in either direction necessarily increases the total distance walked, placing the cappuccino machine at the median is optimal.

**Problem 5. Perfect Powers [30 points] (4 parts)**

A perfect power is an integer  $X$ , for which there exist integers  $B \geq 2$  and  $e \geq 2$  such that  $X = B^e$ . In this problem, you will come up with a fast algorithm that on input  $X$ , outputs  $B$  and  $e$  such that  $X = B^e$ . If no such values exist, your algorithm will output **null**.

If you are unable to answer one part of the problem, you may assume its results for the other parts. Take care not to confuse the input *lengths* with the input *values*.

- (a) [3 points] Let  $X$  be an  $n$ -bit integer. If  $X = B^e$ , prove that ( $e \leq n$ ).

**Solution:** First, since  $X$  is a  $n$ -bit integer ( $n \geq \log X = e \log B$ ). Since ( $B \geq 2$ ), then ( $e \leq e \log B \leq n$ ).

- (b) [5 points] Give an  $O(n^2 \log n)$ -time algorithm that on inputs  $B$  and  $e$ , computes  $B^e$ . Assume that  $B^e \leq 2^n$  and that multiplying an  $n$ -bit integer by an  $m$ -bit integer takes  $O(nm)$  time.

**Solution:**

`EXPONENT( $B, e$ ):`

```

1   $k \leftarrow \log e$    $\triangleright e$  is a  $k$ -bit integer:  $e = 1e_{k-1} \dots e_0$ 
2   $X \leftarrow B$    $\triangleright X = B^{e_k}$  and  $e_k = 1$ 
3  for  $i \leftarrow k - 1$  downto 1:
4     $X \leftarrow 2 * X$    $\triangleright X = B^{b_k \dots b_{i+1} 0}$ .
5    if ( $e_i = 1$ )  $X \leftarrow X + B$    $\triangleright X = B^{b_k \dots b_i}$ 
6  return  $X$ 
```

This code performs  $k$  multiplications, where ( $k = \log e \leq \log n$ ). Because ( $B^e \leq 2^n$ ), each multiplication is of two integers less than  $n$  bits long. Each multiplication thus takes  $O(n^2)$  time, so the overall algorithm takes  $O(n^2 \log n)$  time.

- (c) [12 points] On inputs  $X$  and  $e$ , where  $X$  is an  $n$ -bit integer and  $e \leq n$ , give an  $O(n^3 \log n)$ -time algorithm to find  $B$  such that  $X = B^e$ . That is, find the  $e^{th}$  integer root of  $X$ ,  $B = \sqrt[e]{X}$ . If no such  $B$  exists, return **null**.

**Solution:**

Initial Values for  $L$  and  $H$ :

$$L \leftarrow 2$$

$$H \leftarrow \sqrt{X}$$

ROOT( $X, e, L, H$ ):

$$M \leftarrow (L + H)/2$$

$$Y \leftarrow \text{EXPONENT}(M, e)$$

**if** ( $Y = X$ ) **return**  $M$

**elseif** ( $L = H$ ) **return** **null**

**elseif** ( $Y > X$ ) ROOT( $X, e, L, M$ )

**elseif** ( $Y < X$ ) ROOT( $X, e, M, H$ )

If  $B$  exists, it must be in the range  $[2, \sqrt{X}]$ . Suppose such a  $B$  exists. We can BINARY SEARCH for  $B$  by selecting a midpoint  $M$  of the range  $[L, H]$  and computing  $M^e$  using our  $O(n^2 \log n)$ -time algorithm from part (b). If  $(M^e < X)$ , then clearly  $B \in [M, H]$ . Similarly, if  $(M^e > X)$ , then  $B \in [L, M]$ . If  $B$  does not exist, then the binary search will halt and return **null** after  $(L = H)$ .

This BINARY SEARCH is over a range with size at most  $(\sqrt{X} \leq X)$ . It will perform at most ( $\log X = n$ ) queries, which each take  $O(n^2 \log n)$  time, for a total runtime of  $O(n^3 \log n)$ .

- (d) [10 points] Given a  $n$ -bit integer  $X$ , give an  $O(n^4 \log n)$ -time algorithm to determine whether  $X$  is a perfect power. If  $X$  is not a perfect power, return **null**.

**Solution:**

PERFECT POWER SEARCH( $X$ ):

```
 $n \leftarrow \log X$ 
for  $e := 2$  to  $n$ :
    BINARY-SEARCH for a  $B \in [2, \sqrt{X}]$  such that ( $X = B^e$ ).
    If successful, return  $(B, e)$ 
return null.
```

We can exhaustively test every one of  $n$  possible  $e$  values using our method in part (c). Each call takes  $O(n^3 \log n)$  time, so the entire search will take  $O(n^4 \log n)$  total time.

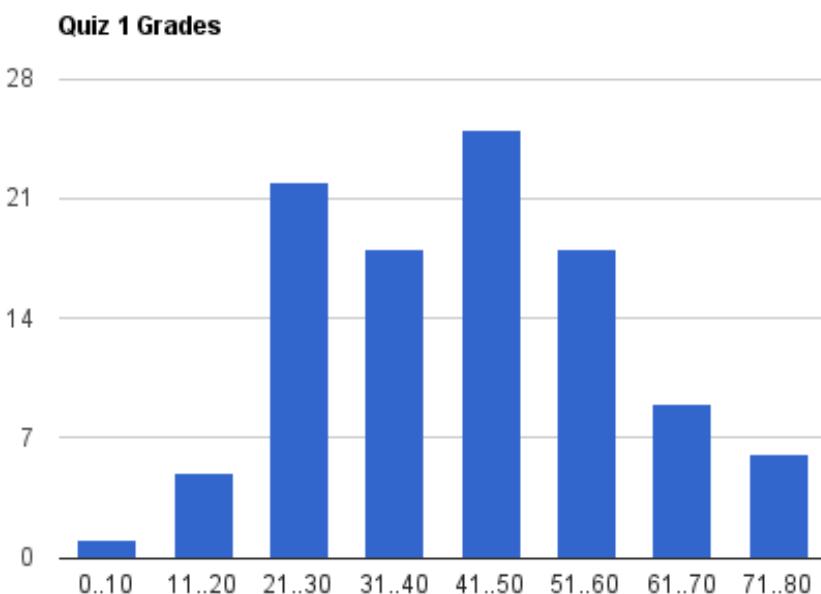
## SCRATCH PAPER

**DETERMINISTICSELECT**( $A, n, i$ )

- 1 Divide the elements of the input array  $A$  into groups of 5 elements.
- 2 Find median of each group of 5 elements and put them in array  $B$ .
- 3 Call **DETERMINISTICSELECT**( $B, n/5, n/10$ ) to find median of the medians,  $x$ .
- 4 Partition the input array around  $x$  into  $A_1$  containing  $k$  elements  $\leq x$   
and  $A_2$  containing  $n - k - 1$  elements  $\geq x$ .
- 5 If  $i = k + 1$ , then return  $x$ .
- 6 Else if  $i \leq k$ , **DETERMINISTICSELECT**( $A_1, k, i$ ).
- 7 Else if  $i > k$ , **DETERMINISTICSELECT**( $A_2, n - k - 1, i - (k + 1)$ ).

---

## Quiz 1



**Problem 0. Name.** [1 point] Write your name on every page of this exam booklet! Don't forget the cover.

### Possibly useful facts for elsewhere in the quiz

1. **Markov Inequality:** For any nonnegative random variable  $X$ , we have

$$\Pr\{X \geq \lambda\} \leq \mathbb{E}[X] / \lambda.$$

2. **Chernoff Bounds:** Let  $X_1, \dots, X_n$  be  $n$  independent Boolean random variables. Suppose that for  $i = 1, 2, \dots, n$ , we have  $\Pr\{X_i = 1\} = \delta_i$  for  $0 \leq \delta_i \leq 1$ . Let  $X = \sum_{i=1}^n X_i$  and  $\delta = (1/n) \cdot \sum_{i=1}^n \delta_i$ . Then, for any  $\gamma \leq \delta \leq 1$ ,

$$\Pr\{X \geq \gamma n\} \leq e^{-2(\gamma-\delta)^2 n}.$$

Note that this is a generalization of the Chernoff bound we saw in class to the case of not necessarily identically distributed random variables.

3. **Harmonic series:**

$$\sum_{i=1}^n \frac{1}{i} = \ln n + O(1).$$

**Problem 1. True or False.** [21 points] (7 parts)

Circle **T** or **F** for each of the following statements to indicate whether the statement is true or false, respectively. You need not justify your answers, but since wrong answers will be penalized, do not guess unless you are reasonably sure.

- (a) **T F** The recurrence  $T(n) = 2T(\sqrt{n}) + \lg n$  has solution  $T(n) = \Theta(\lg^2 n)$ .

**Solution:** False: Substitute  $m = \lg n$ . Then,  $T(m) = 2T(m/2) + m = \Theta(m \lg m)$ . So,  $T(n) = \Theta(\lg n \lg \lg n)$ .

- (b) **T F** The numbers  $1, 2, \dots, 10$  can be placed into a tree data structure such that the tree satisfies both the min-heap property and the binary-search-tree property at the same time.

**Solution:** True: Consider a tree that has only its rightmost branch, containing all the elements in monotonically increasing order.

- (c) **T F** The following collection  $\mathcal{H} = \{h_1, h_2, h_3\}$  of hash functions is universal, where each hash function maps the universe  $U = \{A, B, C, D\}$  of keys into the range  $\{0, 1, 2\}$  according to the following table:

$x$	$h_1(x)$	$h_2(x)$	$h_3(x)$
$A$	1	0	2
$B$	0	1	2
$C$	0	0	0
$D$	1	1	0

**Solution:** True: By verifying that any two rows do no have more than one element in common.

- (d) **T F** Consider a sequence of  $n$  INSERT operations,  $2n$  DECREASE-KEY operations, and  $\sqrt{n}$  EXTRACT-MIN operations on an initially empty Fibonacci heap. The total running time of all these operations is  $\Theta(n \lg n)$  in the worst case.

**Solution:** False: The time by this sequence of operations is  $O(n + 2n + \sqrt{n} \lg n) = O(n)$ .

- (e) **T F** In the analysis of the disjoint-set data structure presented during lecture, once a node other than the root or a child of the root is block-charged, it will never again be path-charged.

**Solution:** True: Follows from the definitions of block-charges and path-charges

.

- (f) **T F** A van-Emde-Boas data structure can support FIND-MIN, FIND-MAX, SUCCESSION, and PREDECESSOR operations over the set  $\{1, 2, \dots, 2^{\sqrt{n}}\}$  in  $O(\log n)$  time.

**Solution:** True: vEB trees can perform the above operations in  $O(\lg \lg(2^{\sqrt{n}}))$  or  $O(\log n)$  time.

- (g) **T F** Let  $G = (V, E)$  be a connected undirected graph with edge-weight function  $w : E \rightarrow \{1, 2, \dots, 10|E|\}$ . Then a minimum spanning tree of  $G$  can be constructed in  $O(E \lg \lg V)$  time.

**Solution:** True: Use vEB data structure as heap.

**Problem 2. S3L3CT.** [12 points]

Professor John Von Meanmann is implementing an algorithm to find the  $i$ th smallest of a set  $S$  of  $n$  distinct elements. The professor improvises on the traditional worst-case linear-time algorithm and organizes elements into groups of 3 instead of groups of 5, resulting in the following algorithm, called S3L3CT:

1. If  $n = 1$ , return the only element in the array.
2. Divide the  $n$  elements of the input array into  $\lfloor n/3 \rfloor$  groups of 3 elements each, with 0 to 2 elements left over.
3. Find the median of each of the  $\lceil n/3 \rceil$  groups by rote.
4. Use S3L3CT recursively to find the median  $x$  of the  $\lceil n/3 \rceil$  medians found in Step 3.
5. Partition the input array around  $x$ . Let  $k = \text{RANK}(x)$ .
6. If  $i = k$ , then return  $x$ . Otherwise, use S3L3CT recursively to find the  $i$ th smallest element on the low side if  $i < k$ , or the  $(i - k)$ th smallest element on the high side if  $i > k$ .

State the recurrence for the running time  $T(n)$  of S3L3CT running on an input of  $n$  elements, and provide a tight asymptotic upper bound on its solution in terms of  $n$ . In order to simplify the math, assume that any set on which S3L3CT operates contains a multiple of 3 elements.

**Solution:**  $x$  is at least as large as  $n/3$  elements (for half of the triplets, 2 out of 3 elements) and at least as small as  $n/3$  elements (for the other half of the triplets, 2 out of 3 elements). So the recursion in Step 5 is, in the worst case, on an array of size  $2n/3$ . The recursion in Step 3 is always on  $n/3$  elements. So we get the recurrence  $T(n) = T(2n/3) + T(n/3) + \Theta(n)$ .

We now use a recursion tree to estimate a solution to the above recurrence. The depth of the tree is dominated by the term  $T(2n/3)$ . Thus, a good estimate for the depth of the recursion tree is  $O(\log_{3/2} n)$ , which is  $O(\lg n)$ . And from the recurrence, it is clear that we do  $O(n)$  at each level. Thus, an initial guess would be that the recurrence is bounded by  $O(n \lg n)$ . We shall now use the substitution method to verify our guess. Make an inductive hypothesis that  $T(m) \leq dm \lg m$ , for all  $m \in [1, n]$ . We will, now, prove the hypothesis for  $n$ .

$$\begin{aligned}
T(n) &= d2n/3 \lg 2n/3 + dn/3 \lg n/3 + \Theta(n) \\
&= 2dn/3(\lg 2 + \lg n - \lg 3) + nd/3(\lg n - \lg 3) + \Theta(n) \\
&= dn/3(3 \lg n + 2 \lg 2 - \lg 3) + \Theta(n) \\
&= dn \lg n + dn/3(2 - 2 \lg 3) + \Theta(n) \\
&\leq dn \lg n + 2dn/3(1 - \lg 3) + kn \quad (\text{from the definition of } \Theta(n)) \\
&= dn \lg n + n(2d/3(1 - \lg 3) + k) \\
&= dn \lg n - n(2d/3(\lg 3 - 1) - k) \\
&< dn \lg n, \text{ for all } d > 3k/2(\lg 3 - 1)
\end{aligned}$$

Thus, we have established that there is an absolute constant  $d$  such that for sufficiently large  $n$ ,  $T(n) < dn \lg n$ . Thus,  $T(n) = O(n \lg n)$ .

**Problem 3. SWAT Team.** [12 points]

Two swatsmen with fly swatters are located at arbitrary positions along a long corridor with many leaky windows. One at a time, houseflies appear at various locations along the corridor, and a swatsman goes to the location of the fly and swats it dead. The cost of a given strategy is the total distance traveled by the swatsmen. Argue that the greedy strategy of the closest swatsman going to the location of the fly is not  $\alpha$ -competitive for any finite  $\alpha$ .

(*Hint:* Consider the case that flies only appear at three locations  $A$ ,  $B$ , and  $C$ , where  $B$  falls between  $A$  and  $C$  and the distance from  $A$  to  $B$  is much smaller than the distance from  $B$  to  $C$ , as shown below:



Consider the sequence  $\langle C, A, B, A, B, A, B, A, B, \dots \rangle$  of fly arrivals.)

**Solution:** Suppose on way of contradiction that there are  $\alpha$  and  $\beta$  such that  $\text{Greedy} \leq \alpha \cdot \text{OPT} + \beta$ . Let  $L$  be the distance between  $A$  and  $C$ , and  $\epsilon L$  be the distance between  $A$  and  $B$ , where  $0 < \epsilon < 1/2$ . Let  $T$  be the length of the sequence of flight arrivals. Take  $T > (3\alpha/\epsilon) + (\beta/\epsilon L)$ .

On the sequence of fly arrivals defined above:

- OPT sends one swatsman to  $C$  for the first fly, and then, for the rest of the flies, send one swatsman to  $A$  and one swatsman to  $B$ . The total cost is at most  $3L$ .
- The greedy algorithm, after sending one swatsman to  $C$  and the other to  $A$ , lets the second swatsman take care of all flies in  $B$  and  $A$ . The cost is at least  $\epsilon LT$ .

We chose the parameters so  $\epsilon LT > \alpha \cdot 3L + \beta$ , which contradicts the assumption.

**Problem 4. FIFO = 2 × LIFO.** [12 points]

A FIFO queue  $Q$  supporting the operations ENQUEUE and DEQUEUE can be implemented using two stacks  $S_1$  and  $S_2$ , each of which supports the operations PUSH, POP, and a test whether the stack is empty.

ENQUEUE( $Q, x$ ):

1 PUSH( $S_1, x$ )

DEQUEUE( $Q$ ):

```

1 if  $S_1 = \emptyset$  and  $S_2 = \emptyset$ 
2   error "queue underflow"
3 if  $S_2 = \emptyset$ 
4   while  $S_1 \neq \emptyset$ 
5      $x = \text{POP}(S_1)$ 
6     PUSH( $S_2, x$ )
7 return POP( $S_2$ )

```

Define a potential function  $\Phi(Q) = c |S_1|$  for an appropriate constant  $c > 0$ , where  $|S_1|$  is the number of items in  $S_1$ . Argue using a potential-function argument that each ENQUEUE and DEQUEUE operation takes  $O(1)$  amortized time.

**Solution:**

Take the potential function  $\Phi$  to be  $3 |S_1|$ . Clearly,  $\Phi$  is always non-negative and initially,  $\Phi_0 = 0$ . The amortized cost of each operation is its true cost plus the change in the potential. Let us evaluate it for each of the operations separately:

- ENQUEUE: The change in potential is  $+3$ . The true cost is  $1$ . Thus, the amortized cost is at most  $4$ .
- DEQUEUE: There are two cases:
  - If  $S_2 \neq \emptyset$ , the change in potential is  $0$ , and the true cost is  $3$  (the operations in steps 1,3,7). Thus, the amortized cost is  $3$ .
  - If  $S_2 = \emptyset$ , the change in potential is  $-3 |S_1|$ , and the true cost is at most  $3 |S_1| + 4$ . Thus, the amortized cost is at most  $4$ .

In all cases, the amortized cost is at most  $4$ .

**Problem 5. Big Edges.** [10 points]

Let  $G = (V, E)$  be a connected undirected graph with distinct edge weights  $w : E \rightarrow \mathbb{R}$ , and let  $c$  be a cycle in  $G$ . Consider the edge  $e$  on  $c$  with the largest weight, that is,  $w(e) \geq w(e')$  for all  $e' \in c$ . Prove that  $e$  does not belong to the minimum spanning tree of  $G$ .

**Solution:** Suppose for the sake of contradiction that  $e = \{u, v\}$  is in a minimum spanning tree  $T$  of  $G$ . If we remove  $e$  from  $T$ , we divide it into two trees. This corresponds to a cut  $(C, V - C)$  in  $G$  such that  $C$  is spanned by one of the trees and  $V - C$  is spanned by the other. Since  $e$  is in a cycle and it crosses this cut, there must exist another edge in the cycle,  $e'$ , that crosses this cut. (Any cycle must cross a cut an even number of times. To see why, follow the edges of the cycle.) Since edge  $e'$  crosses the cut, adding it in will connect the two trees thus forming a spanning tree:  $T - \{e\} \cup \{e'\}$ .

Since all edge weights are distinct and  $e$  is the edge with the largest weight on the cycle, necessarily  $w(e') < w(e)$ . The weight of  $T'$  is therefore strictly lower than the weight of  $T$ , and therefore  $T$  cannot be a minimum spanning tree.

**Problem 6. Minimum Madness.** [12 points] (4 parts)

Consider the following program to find the minimum value in an array  $A$  of  $n$  distinct elements.

$\text{MINIMUM}(A, n)$ :

```

1   min = ∞ // Set min to be a large value.
2   for i = 1 to n
3       if min > A[i]
4           min = A[i]
```

Assume that  $A$ 's elements are randomly permuted before invoking  $\text{MINIMUM}(A, n)$  and that all permutations are equally likely. Let  $X_i$  be the indicator random variable associated with the event that the variable  $min$  is changed in line 4 during the  $i$ th iteration of the **for** loop, and let  $Y = \sum_{i=1}^n X_i$  be the random variable denoting the total number of times  $min$  is so updated.

- (a) Argue that the probability that  $A[i]$  is smaller than all the elements in  $A[1..i - 1]$  is  $1/i$ .

**Solution:** Notice that the probability that  $A[i]$  is smaller than all the elements  $A[1..i - 1]$  is exactly equal to the probability that we find the minimum element of  $A[1..i]$  at position  $i$  (here we use that all the elements are distinct). Since  $A$  is randomly permuted and every permutation is equally likely, each one of positions  $1..i$  is equally likely to hold the minimum element, and the probability it lands at position  $i$  is exactly  $1/i$ .

- (b) Show that  $E[X_i] = 1/i$ .

**Solution:**

$$\begin{aligned} E[X_i] &= 0 \cdot \Pr\{X_i = 0\} + 1 \cdot \Pr\{X_i = 1\} \\ &= \Pr\{X_i = 1\} \\ &= 1/i \end{aligned}$$

(c) Prove that  $E[Y] = \Theta(\lg n)$ .

**Solution:** By linearity of expectation and the sum of the harmonic series,

$$\begin{aligned} E[Y] &= E\left[\sum_{i=1}^n X_i\right] \\ &= \sum_{i=1}^n E[X_i] \\ &= \sum_{i=1}^n 1/i \\ &= \ln n + O(1) \\ &= \lg n \cdot \lg e + O(1) \\ &= \Theta(\lg n) \end{aligned}$$

(d) Prove that for sufficiently large  $n$ , it holds that  $\Pr\{Y \geq 5E[Y]\} \leq 1/n^4$ .

**Solution:** We use the following *multiplicative* version of Chernoff bound: For independent Boolean random variables  $X_1, \dots, X_n$ , let  $X = \sum_{i=1}^n X_i$ , and  $\mu = E[X]$ . Then, for any  $0 < \delta \leq 2e - 1$ ,

$$\Pr\{X > (1 + \delta)\mu\} < e^{-\mu\delta^2/4}.$$

For our problem, since  $\mu > \ln n$ , and so

$$\Pr\{Y > 5E[Y]\} < e^{-4\ln n} = \frac{1}{n^4}.$$

The statement of the Chernoff bound given in the beginning of the quiz is not strong enough to prove the required bound. We gave full credit to anyone who used it correctly.

## Quiz 1

- Do not open this quiz booklet until you are directed to do so. Read all the instructions first.
- When the quiz begins, write your name on every page of this quiz booklet.
- The quiz contains five multi-part problems. You have 80 minutes to earn 80 points.
- This quiz booklet contains **11** pages, including this one and an extra sheet of scratch paper, which is included for your convenience.
- This quiz is closed book. You may use one handwritten Letter ( $8\frac{1}{2}'' \times 11''$ ) or A4 crib sheet. No calculators or programmable devices are permitted.
- Write your solutions in the space provided. Extra scratch paper may be provided if you need more room, although your answer should fit in the given space.
- Do not waste time and paper re-deriving facts that we have studied. It is sufficient to cite known results.
- 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. Generally, a problem's point value is an indication of how much time to spend on it.
- 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.
- Good luck!

Problem	Points	Grade	Initials
1	15		
2	20		
3	15		
4	20		
5	10		
Total	80		

Name: \_\_\_\_\_

Circle your recitation:

Minji 10AM      Jared 11AM      Minji 12PM      Jared 1PM(in 36-112)  
Jose 1PM(in 36-144)    Jose 2PM(in 36-144)    Jinwoo 2PM(in 36-112)    Jinwoo 3PM

**Problem 1. Recurrences [15 points] (3 parts)**

For each of the following recurrences, give an asymptotically tight ( $\Theta(\cdot)$ ) bound. Justify your answer by naming the particular case of the Master's Method, by iterating the recurrences, or by using the substitution method. As usual, assume that for  $n \leq 10$ ,  $T(n) = O(1)$ .

**Example:** [0 points] BINARY SEARCH

Recurrence:  $T(n) = T(n/2) + c$

Solution by iteration:

$$T(n) = T(n/4) + c + c = \sum_{i=0}^{\log n} c = c \log n = \Theta(\log n)$$

(a) [5 points]  $T(n) = 8T(n/2) + \Theta(n).$

**Solution:**  $T(n) = \Theta(n^3)$  by part 1 of the Master's Method.

(b) [5 points]  $T(n) = 9T(n/9) + \Theta(n\sqrt{n}).$

**Solution:**  $T(n) = \Theta(n\sqrt{n})$  by part 3 of the Master's Method.

- (c) [5 points]  $T(n) = T(\sqrt{n}) + \log n$ . (It is fine to assume that  $n$  is of the form  $2^{2^k}$  in order to avoid floor and ceiling notation.)

**Solution:** Let  $n = 2^k$ . Then the recurrence becomes  $T(2^k) = T(2^{k/2}) + k$ , or, if we set  $L(k) = T(2^k)$ ,  $L(k) = L(k/2) + k$ . This solves to  $L(k) = \Theta(k)$  by part 3 of the Master's Method. Thus,  $T(n) = \Theta(k) = \Theta(\log n)$ .

Alternative solution is note that  $T(n) \geq \log n$  and, for the upper bound, iterate the recurrence:

$$T(n) = T(\sqrt{n}) + \log n = T(n^{1/4}) + \log \sqrt{n} + \log n \leq \sum_{i=0}^{\infty} \log n^{1/2^i} = \log n \sum_{i=0}^{\infty} \frac{1}{2^i} \leq 2 \log n.$$

Many students made mistakes on this problem, but partial credit (2-3 points) was given where solutions were on the right track, but made a math mistake or used the wrong case of the Master Method (e.g. finding  $L(k) = \Theta(k \log k)$  instead of  $L(k) = \Theta(k)$ ).

**Problem 2. True or False, and Justify [20 points] (4 parts)**

Circle T or F for each of the following statements, and briefly explain why. The better your argument, the higher your grade, but be brief. Your justification is worth more points than your true-or-false designation.

- (a) **T F** [5 points]  $f(n) = \Theta(g(n))$  is equivalent to  $g(n) = \Theta(f(n))$ .

**Solution:** True.  $f(n) = \Theta(g(n))$  means there exists some  $n_0$  and constants  $c_1, c_2 > 0$  such that for  $n \geq n_0$  we have  $c_1g(n) \leq f(n) \leq c_2g(n)$ . But this is equivalent to  $\frac{1}{c_2}f(n) \leq g(n) \leq \frac{1}{c_1}f(n)$ , or  $g(n) = \Theta(f(n))$ .

Many people simple argued that if  $f(n) = \Theta(g(n))$  then  $f(n)$  “grows asymptotically as fast as”  $g(n)$ , which implies that  $g(n)$  also grows as fast as  $f(n)$ . This argument is clearly not enough since we expected that you use the definitions, so no points were given for that justification.

Some people also assumed the fact that  $f(n) = O(g(n))$  implies that  $g(n) = \Omega(f(n))$  to conclude (which, although it is true, needed a line of justification and so one or two points were taken off in those cases).

- (b) **T F** [5 points] There is a deterministic algorithm to sort  $n$  numbers in the comparison model running in time  $\log F_n$ , where  $F_n$  is the  $n$ th Fibonacci number. (Recall that the Fibonacci numbers are defined as follows:  $F_0 = 0, F_1 = 1$  and for  $i > 1$ ,  $F_i = F_{i-1} + F_{i-2}$ .)

**Solution:** False. Remember that  $F_n \leq c^n$ , where  $c$  is some constant. Then,  $\log F_n = n \log c = O(n)$ . In the comparison model, we know that sorting takes  $\Omega(n \log n)$  time.

Some people also justified this part by saying that it is impossible to sort 2 numbers using  $0 = \log 1 = \log F_2$  comparisons in the comparison model. Although the argument is correct we were expecting an asymptotic argument using the lower bound for sorting in the comparison model. Also, a common mistake was to confuse the magnitude of  $F_n$  with the time needed to compute the  $n$ -th Fibonacci number, which is exponentially smaller.

- (c) T F [5 points] It is impossible to build a heap on  $n$  elements in time  $o(n \log n)$ .

**Solution:** False. We can build a heap in  $O(n)$  time.

Many of you thought that building a heap is equivalent to sorting an array. Remember, heap is an array that satisfies the “Heap Property”, which is weaker than the property satisfied by a sorted array. For example, given elements  $[5, 7, 9]$ , a sorted (non-increasing) array can only take the form  $[9, 7, 5]$  while a heap can be either  $[9, 7, 5]$  or  $[9, 5, 7]$ . This distinction between sorted array and heap is important.

One possible reason why many of you were confused between building a heap and sorting an array might be because we talked about “Heapsort” in class. Heapsort is a sorting algorithm that uses heap as a nice datastructure to sequentially extract the maximum element from the input. So, heapsort will give you a sorted array, but heap is not a sorted array.

- (d) T F [5 points] RADIX SORT does not work correctly if we sort each individual digit with a stable INSERTION SORT instead of COUNTING SORT.

**Solution:** False, RADIX SORT is always correct with any stable sort. However it is less efficient. A step of RADIX SORT requires just that the respective individual digits are sorted, and any sorting algorithm works instead of COUNTING SORT. However, you probably don’t want to use INSERTIONSORT instead of COUNTING SORT for sorting digits. INSERTIONSORT will take  $\Theta(n^2)$  time, while COUNTING SORT will take only  $\Theta(n)$  time.

**Problem 3. Plural Elements [15 points] (2 parts)**

You are given an array of  $n$  integers. For each of the questions below, remember to give a brief, clear explanation of why the algorithm works.

- (a) [5 points] Suppose there exists an integer that appears more than  $n/2$  times in the array. Give a linear time deterministic algorithm to find such an integer.

**Solution:** Find the median of the array – it will be the majority element. Since, if the median wouldn't be the majority element, then the majority element would be smaller or larger than the median, which is impossible (e.g., there are only  $n/2$  elements smaller than the median).

Many people thought an algorithm using the COUNTING idea, which was used in COUNTING SORT. But, it need an assumption for the range of elements, otherwise it is impossible to implement and it may not run in  $O(n)$  time. Also, some people designed an algorithm essentially using the Sorting algorithm, which runs in  $\Theta(n \log n)$  time. Both answers got approximately half of full points.

- (b) [10 points] Now suppose there exists an integer that appears more than  $n/k$  times, where  $k > 2$  is a constant (suppose  $n$  is divisible by  $k$ ). Give a linear time deterministic algorithm to find all such integers.

**Solution:** Let's call a common element an integer that appears more than  $n/k$  times. Let  $a_1, a_2, \dots, a_k$  be the elements with ranks respectively  $n/k, 2n/k, \dots, (k-1)n/k, n$ . Then we check each of the elements  $a_1, a_2, \dots, a_k$  whether it is a common element (just a linear scan per element).

The correctness of the algorithm follows from the claim that any common element must be among  $a_1, a_2, \dots, a_k$ . To see that, consider the sorted array. Any common element is a contiguous block of size  $> n/k$ . Since our “probes”  $a_1, a_2, \dots, a_k$  are at distance  $n/k$ , they must strand the block of a common element.

**Problem 4. Min and Max Revisited** [20 points] (3 parts) In this problem we will investigate a divide and conquer approach for *simultaneously* finding the minimum and maximum of an array of  $n$  integers. in the *comparison* model (i.e., your algorithm is only allowed to compare elements, but not add/subtract/index with them).

Assume for simplicity that  $n$  is a power of 2.

For each of the questions below, remember to give a brief and clear justification.

- (a) [2 points] Suppose you compute the maximum separately and the minimum separately, and output both. How many comparisons does this require?

**Solution:** Computing the maximum takes  $n - 1$  comparisons and the minimum takes another  $n - 1$  comparisons. Total number of comparisons is  $2n - 2$ .

(Full points were given for just saying  $\Theta(n)$ . A few students thought we were asking for lower bounds and gave lower bounds of  $\Omega(n)$  or  $\Omega(\log n)$ . Both answer got full points as well.)

- (b) [8 points] For the rest of this problem, we would like to reduce the leading constant factor in the number of comparisons. We want a result that is not necessarily better asymptotically, but in terms of the exact number of comparisons.

Let's consider an idea for designing a divide and conquer solution for simultaneously computing the minimum and maximum element. The plan is to break the problem into two sublists, compute the min and max of each sublist and then combine the results. An outline of such an algorithm is given below. In the following page, you will be given the opportunity to fill in some details:

**MINMAX**( $A, n$ );

```
If  $n = 2$  then  $\langle \dots \rangle$  fill in base case  
else Let  $A1$  = left half and  $A2$  = right half  
       $(a, b) = \text{MINMAX}(A1, n/2)$   
       $(c, d) = \text{MINMAX}(A2, n/2)$   
 $\langle \dots \rangle$  fill in combine stage
```

- (i) Fill in the details for the case  $n = 2$ .

**Solution:** If  $A[1] < A[2]$  return( $A[1], A[2]$ ), else return( $A[2], A[1]$ );  
A few solutions said “Return  $\min(A[1], A[2]), \max(A[1], A[2])$ ”. Since the goal is to minimize actual number of comparisons this is wasteful. Usually (depending on clarifications in Part (c)), two to four points were taken off for this error.

- (ii) Fill in the details for the combine step here.

**Solution:** If  $a < c$  let  $a' = a$ , else  $a' = c$ . If  $b < d$  then  $b' = d$  else  $b' = b$ .  
Return( $a', b'$ )

(c) [10 points]

Let  $T(n) = c_1n - c_2$  be the number of comparisons used by the algorithm. Find an exact expression for  $T(n)$  (i.e., the best values of the constants  $c_1$  and  $c_2$ ). (You may do this by writing out and solving the recurrence for  $T(n)$  using the substitution method. ) Is this really better than the naive algorithm?

**Solution:** The recurrence for the number of comparisons is  $T(n) = 2T(n/2) + 2$  with the base case  $T(2) = 1$ .

We guess that the solution is of the form  $T(n) = c_1n - c_2$ . Then we have  $c_1n - c_2 = 2(c_1n/2 - c_2) + 2$ , giving  $c_2 = 2$ . Now plugging in  $T(2) = 2c_1 - 2 = 1$ , we get  $c_1 = 3/2$ .

Thus the number of comparisons is  $(3/2)n - 2$ , which has a better leading constant than the naive solution.

Alternate solutions expanded the recurrence as  $2+4+\dots+n/2+n/2T(2) = 3/2n-2$ , using  $T(2) = 1$ . Such a solution also received full credit.

Common mistakes in this part were to ignore the base case, or to say  $T(n) = 2T(n/2) + \Theta(1)$ .

**Problem 5. Matrix Multiplication [10 points] (2 parts)**

In this problem we are given three matrices and the question is to compute their product. For each of the two cases below, report the fastest algorithm you know to compute the product of the given three matrices. You need to prove correctness of the algorithm and argue its running time.

- (a) [5 points] Given matrices  $A, B, C$ , each of dimension  $n \times n$ , give a fast algorithm to compute  $A \cdot B \cdot C$ ? What is its asymptotic running time? (You may use algorithms mentioned in lectures without describing them.)

**Solution:** The solution depends on what's the fastest way to compute the product of just two matrices. Suppose it takes  $T$  time to compute the product of two  $n \times n$  matrices. Then, computing  $A \cdot B \cdot C$  takes  $2T$  time – just compute  $A \cdot B$  first, and then multiply the result, also an  $n \times n$  matrix, by  $C$ .

What is the best known  $T$ ? Strassen's algorithm gives a  $T = O(n^{\log_2 7})$  time algorithm for computing the product of two matrices. The currently best known algorithm has  $T = O(n^{2.376})$ , and is due to Coppersmith and Winograd. We gave full points for the answer  $O(n^{\log_2 7})$ , and four points for knowing there is a faster algorithm than the straightforward  $O(n^3)$  algorithm, but not remembering the runtime.

Several solutions forgot to mention the runtime, two to three points were taken off for this type of error.

- (b) [5 points] Now, suppose  $C$  is of dimension  $n \times 1$  (i.e.,  $C$  is a vertical vector), and  $A$  and  $B$  are still of dimension  $n \times n$ . Give a fast algorithm for computing  $A \cdot B \cdot C$ ? What is its asymptotic running time?

**Solution:** Here the trick is to perform the computations in the right order, using the fact that multiplying an  $n \times n$  matrix by an  $n \times 1$  vector can be done in  $O(n^2)$  time with the straight-forward algorithm. First compute  $B \cdot C$ , which takes  $O(n^2)$  time. Then compute  $A \cdot (B \cdot C)$ , which is also a multiplication of a  $n \times n$  matrix by a  $n \times 1$  vector, and takes  $O(n^2)$  times. So, the total running time is  $O(n^2)$ .

Some people thought that matrix multiplication is commutative, please check to convince yourself that it is not. Three to four points were given for the right algorithm, but wrong runtime analysis.

## SCRATCH PAPER

## Quiz 1

- Do not open this quiz booklet until you are directed to do so. Read all the instructions first.
- The quiz contains six multi-part problems. You have 80 minutes to earn 80 points.
- This quiz booklet contains 12 pages, including this one and an extra sheet of scratch paper, which is included for your convenience.
- This quiz is closed book. You may use one handwritten Letter ( $8\frac{1}{2}'' \times 11''$ ) or A4 crib sheet. No calculators or programmable devices are permitted. **TURN IN your crib sheet with this quiz if you used one.**
- Write your solutions in the space provided. Extra scratch paper may be provided if you need more room, although your answer should fit in the given space.
- Do not waste time and paper re-deriving facts that we have studied. It is sufficient to cite known results.
- 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. Generally, a problem's point value is an indication of how much time to spend on it.
- 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.
- Good luck!

Problem	Points	Grade	Initials
1	3		
2	9		
3	18		
4	16		
5	20		
6	14		
Total	80		

Name: \_\_\_\_\_

Circle your recitation instructor:

Huy Nguyen (r03 at 12, r07 at 11)

Chris Wilkens (r01 at 10, r02 at 11)

Robert Bryant (r05 at 2, r08 at 12)

Katherine Lai (r09 at 1, r11 at 2)

**Problem 1.** [3 points] Write your name on every page!

**Problem 2. Recurrences** [9 points] (4 parts)

Solve the following recurrences by giving tight  $\Theta$ -notation bounds. You do not need to justify your answers, but any justification that you provide will help when assigning partial credit. As usual, assume that for  $n \leq 10$ ,  $T(n) = O(1)$ .

(a) [2 points]  $T(n) = 7T(n/3) + n^2 \log n.$

**Solution:**  $T(n) = \Theta(n^2 \log n)$  by case 3 of Master's Method.

(b) [2 points]  $T(n) = 9T(n/3) + n^2 \log n.$

**Solution:**  $T(n) = \Theta(n^2 \log^2 n)$  by generalized case 2 of Master's Method.

(c) [2 points]  $T(n) = 5T(n/2) + n \log n$

**Solution:**  $T(n) = \Theta(n^{\log_2 5})$  by case 1 of Master's Method.

(d) [3 points]  $T(n) = 4T(\sqrt{n}/3) + \log^2 n.$

**Solution:** Let  $n = 2^m$ . Then the recurrence becomes  $T(2^m) = 4T(2^{m/2}/3) + m^2$ . Setting  $S(m) = T(2^m)$  gives us  $S(m) = 4S(m/2 - \log 3) + m^2$ . Dropping lower order terms and using case 2 of the Master's Method gives us  $S(m) = \Theta(m^2 \log m)$ , or  $T(n) = \Theta(\log^2 n \log \log n)$ .

**Problem 3. True or False, and Justify [18 points] (6 parts)**

Circle **T** or **F** for each of the following statements, and briefly explain why. The better your argument, the higher your grade, but be brief. Your justification is worth more points than your true-or-false designation.

- (a) **T F** [3 points]  $f(n) = O(g(n))$  and  $g(n) = \omega(h(n))$  implies that  $f(n) = O(h(n))$

**Solution:** False. If  $f(n) = n^2$ ,  $g(n) = n^3$ , and  $h(n) = n$ , the conditions are true, but  $f(n) \neq O(h(n))$ .

- (b) **T F** [3 points] If we can only perform pairwise comparisons, we *cannot* sort an array of numbers in  $O(n)$  worst-case time.

**Solution:** True. Sorting requires  $\Omega(n \log n)$  comparisons in the worst case.

- (c) **T F** [3 points] Counting sort will always sort an array of  $n$  integers from  $\{1, 2, \dots, l\}$  in  $O(n)$  time.

**Solution:** False. Counting sort runs in  $\Theta(n + l)$  time, which is not  $O(n)$  for  $l = \omega(n)$ .

- (d) **T F** [3 points] Using a comparison-based algorithm, we can either sort  $n$  items in  $O(n \lg n)$  time or sort them in place. However, there is no sorting algorithm with *both* aforementioned properties.

**Solution:** False. For example, HEAPSORT does this.

- (e) **T F** [3 points] Irrespective of running time, RADIX-SORT will still work correctly if we use INSERTION-SORT as its intermediate sort.

**Solution:** True. As presented in class, COUNTING-SORT is a stable sort. Thus, it can be used as an intermediate sort for RADIX-SORT.

- (f) **T F** [3 points] Consider a procedure for constructing a max-heap out of an array  $A$ , which first sets heap-size of  $A$  to 0, and then calls MAX-HEAP-INSERT on each element of  $A$  from left to right. This yields the same runtime as the standard BUILD-HEAP algorithm.

**Solution:** False. BUILD-HEAP ends up being  $\Theta(n)$  because with every element of  $A$  processed (from  $n/2$  to 1), the cost of processing is  $O(h)$  of the heap rooted at that element. The number of nodes for which  $h$  is large is small (the higher you go up the heap, the fewer nodes there are). In the case of the iterative MAX-HEAP-INSERT approach, the cost of each insertion is again  $O(h)$ , but this time it is  $h$  of the single heap being built. The number of nodes inserted into the heap when it has large  $h$  is significant. Just consider the bottom row of “leaves”:  $n/2$  insertions, each at cost  $\Theta(\lg n)$ . The runtime for this algorithm is therefore  $O(n \lg n)$ .

**Problem 4. Short Answers [16 points] (4 parts)**

Give *brief*, but complete, answers to the following questions.

- (a) [4 points] Recall the divide-and-conquer algorithm for multiplying two degree- $n$  polynomials  $a$  and  $b$  (with coefficients  $a_i$  and  $b_i$ ,  $i = 0, 1, \dots, n$ ). If we are given the further restriction that  $b_i = 0$  for  $1 \leq i \leq n/2$ , do we see an improvement in runtime? Justify your answer.

**Solution:** We do not see any improvement. In our solution for polynomial multiplication, we first break down  $a$  and  $b$  into  $a = p + x^{n/2}q$  and  $b = s + x^{n/2}t$ . We then recursively define our multiplication as  $a * b = ps + x^{n/2}((p+q)*(s+t) - ps - qt) + x^nqt$ , which allows us to only perform 3 recursive multiplications, yielding a runtime of  $T(n) = \Theta(n^{1.58})$ . While our further restriction defines  $s = 0$  for our first recursive iteration, yielding  $a * b = x^{n/2}((p+q)*(t) - qt) + x^nqt$ , which requires only two recursive multiplications, this restriction on  $s$  will not hold for further recursions.

An alternative valid solution is to state that the original problem with input restriction is identical to calling the polynomial multiplication algorithm on size  $n$  and size  $n/2$  polynomials without input restriction (or is greater than or equal to calling the algorithm on two size  $n/2$  polynomials without restriction). This obviously has the same *asymptotic* running time as multiplying two size  $n$  polynomials.

- (b) [4 points] Argue that you cannot build a binary search tree in the comparison model in  $o(n \lg n)$  time.

**Solution:** An in-order tree walk requires  $O(n)$  time and visits the nodes of a BST in sorted order. As such, we could sort in  $o(n \lg n)$  time by building a BST in  $o(n \lg n)$  time and then doing an in-order tree walk in  $O(n)$  time.

- (c) [4 points] In spite of having to maintain extra information for red-black trees as compared to standard binary search trees, they are very useful. Why?

**Solution:** The properties of red-black trees ensure that a tree is roughly balanced and, therefore, that its height is  $O(\lg n)$ . This ensures  $O(\lg n)$  running times for the basic tree operations.

- (d) [4 points] Ben claims that he can put the numbers  $1, \dots, 10$  in a tree such that it is both a valid heap and binary search tree at the same time. Is he right?

**Solution:** No. Assume we are using a min-heap. If the tree is a BST, then all items in the left subtree of the root are smaller than the root. On the other hand, since it is a heap, the root must be smaller than its children. This implies that the left subtree must be empty, which cannot be true in a nearly-complete binary tree (which heaps are) on 10 nodes. An analogous argument holds for a max-heap on the right subtree.

**Problem 5. Slightly Longer Short Answer Questions [20 points] (4 parts)**

- (a) [5 points] The running time of QUICKSORT can be improved in practice by taking advantage of the fast running time of insertion sort when its input is “nearly” sorted (i.e., has few inversions). When QUICKSORT is called on a subarray with fewer than  $k$  elements, let it simply return without sorting the subarray. After the top-level call to QUICKSORT returns, run INSERTION-SORT on the entire array to finish the sorting process. Analyze the expected running time. Is it asymptotically better than the original QUICKSORT?

**Solution:** We break the running time into the sum of two parts: the cost of the early-quitting QUICKSORT and the cost of the INSERTION-SORT. The early-quitting QUICKSORT is exactly like a normal QUICKSORT, except that it only recurses  $\Theta(\lg n - \lg k) = \Theta(\lg \frac{n}{k})$  times. This yields an overall runtime of  $O(n \lg \frac{n}{k})$ .

As proven in problem set 2, the runtime of INSERTION-SORT is asymptotically equal to the number of inversions in the input array. Because of the preliminary QUICKSORT, we are guaranteed to only have inversions within any of the  $n/k$  groups of  $k$  elements. This yields  $\frac{n}{k} * \binom{k}{2}$  possible inversions, for a total runtime of  $\frac{n}{k} * \Theta(k^2) = O(nk)$ . Our total runtime is thus  $O(nk + n \lg(n/k))$ .

- (b) [5 points] Design a data structure called DISTINCT which maintains a *multi-set* of integers in the range  $\{1, 2, \dots, n^3\}$  under insertions and deletions. A multi-set is a set of elements where duplicate keys are allowed (e.g.,  $\{1, 2, 1, 3\}$  is a multi-set). At any time, the data structure should provide the current number of *distinct* elements in the multi-set. For example, the multi-set  $\{1, 2, 1, 3\}$  contains 3 distinct elements.

Design a randomized data structure that supports the above operations in  $O(1)$  expected time. You can assume that at any time the total number of distinct elements in the set is at most  $n$ . Moreover, you can assume that you are given an  $O(n)$ -size empty block of memory at the beginning.

**Solution:** We maintain a hash table of size  $n$  to stored integers in the multiset along with their duplication counts. For each insertion, we lookup the inserted integer in the hash table. If the number is already there, we increase to its duplication count by 1. If it is not, we add that it to the table with duplication count 1. For each deletion, we look up the deleted number in the table, decrease its duplication count by 1 and remove it from the table if the duplication count is 0. We also need to maintain a distinct count for answer queries. This count is increased by 1 whenever new number is added into the table and decreased by 1 whenever a number is removed from the table.

By choosing a hash function  $h$  randomly from a universal hashing family, the expected time for each table lookup is  $O(1)$ , and therefore,  $O(1)$  for each operation.

- (c) [5 points] In order to survive in a long, cold winter, Ug stores  $n$  pieces of meat of sizes  $1, 2, \dots, n$  respectively in his cave. Every day, he takes a piece at random from the storage. If the piece has even size, he eats half of it and puts the other half back. If the piece has odd size, he eats it all. Argue that with this strategy, Ug can live off of this meat for only  $O(n)$  winter days.

**Solution:**

In the original  $n$  pieces of meat,  $\frac{n}{2}$  of them (with odd sizes) are eaten once,  $\frac{n}{4}$  pieces are eaten twice,  $\frac{n}{8}$  pieces are eaten 3 times, etc. Therefore, the total number of times Ug can eat is:

$$\begin{aligned} & \left\lceil \frac{n}{2} \right\rceil + 2 \left\lceil \frac{n}{4} \right\rceil + 3 \left\lceil \frac{n}{8} \right\rceil + \dots + L \left\lceil \frac{n}{2^L} \right\rceil \\ &= \sum_{k=1}^L \sum_{i=k}^L \frac{n}{2^i} \\ &= \sum_{k=1}^L O\left(\frac{n}{2^{k-1}}\right) \\ &= O(n) \end{aligned}$$

- (d) [5 points] Ug's daughter has started to write a daily diary. At the beginning, she was very diligent about writing down what happened that day. As time goes on, however, she gets lazier and writes in it less often. In fact, on the  $i$ -th day since she has started her diary, she will write an entry with probability of only  $1/i$ . After  $t$  days have passed since she started writing, what is the expected number of entries in her diary? Use indicator random variables.

**Solution:** Define the indicator random variable  $X_i$  that is 1 if Ug's daughter writes in her diary on the  $i$ -th day and 0 otherwise.  $\Pr[X_i = 1] = 1/i$ , so  $E[X_i] = 1/i$ . By linearity of expectation and the harmonic series, we should get the following unique timestamps:

$$\begin{aligned} E \left[ \sum_{i=1}^t X_i \right] &= \sum_{i=1}^t E[X_i] \\ &= \sum_{i=1}^t \frac{1}{i} \\ &= O(\log t) \end{aligned}$$

**Problem 6. Ug's Binary Trees [14 points] (2 parts)**

After his success in sorting, Ug has gotten into the data storage business. In particular, he is maintaining binary search trees on the behalf of the government. The binary trees support the usual operations: insertions, deletions and successor. Unfortunately, the government also needs to keep records, and therefore they need to know both the current state of the tree *and* all prior states of the tree. Specifically, they need to be able to use a *generalized* successor operation, which allows to specify both the value *and* the time at which we want the successor.

Ug's first thought is to copy the tree every time the government modifies the tree (via insertions or deletions). Unfortunately, this is expensive.

He soon has another idea: each time the tree is modified, very few pointers actually change. Instead of copying the tree, he decides to store changes to each pointer in individual HISTORY structures (one for each pointer). Each time a pointer in the tree is modified, the HISTORY for that pointer is updated. Each entry in the HISTORY contains a timestamp  $t$  (counting from 1, 2, 3, ... and so on) and a copy of the pointer at the specified time. Then, when the government makes a generalized successor query, Ug makes sure to follow the version of the pointer at the time specified by the government.

For example, consider inserting a node  $u$  into a tree at time  $t$ . We create one new pointer from  $u$ 's parent to  $u$ . The HISTORY for this pointer should say that it did not exist right before time  $t$  and that it pointed to node  $u$  starting at time  $t$ . Other operations are similar.

When a generalized successor call is made with argument  $t'$ , Ug looks at each pointer he needs to follow, finds the version at time  $t'$ , and follows that pointer.

Now, Ug must design the HISTORY structure, and needs your help.

- (a) [7 points] Ug decides to use a linked list for the HISTORY structure. If the current time is  $t$ , how much time would the generalized successor queries take? What about the usual successor queries and updates? You can assume that the height of the tree is at most  $h$ .

**Solution:** If Ug uses linked lists, it will potentially take  $O(t)$  time at each node to find out which version of the pointer to use. Since Ug has to follow  $O(h)$  pointers, this might take  $O(th)$  time per special query. Normal queries and updates will still take  $O(h)$  time because for any pointer, Ug just has to either consult the last item in the linked list for  $O(1)$  time or add some new change to the end of a linked list, also in  $O(1)$  time per pointer.

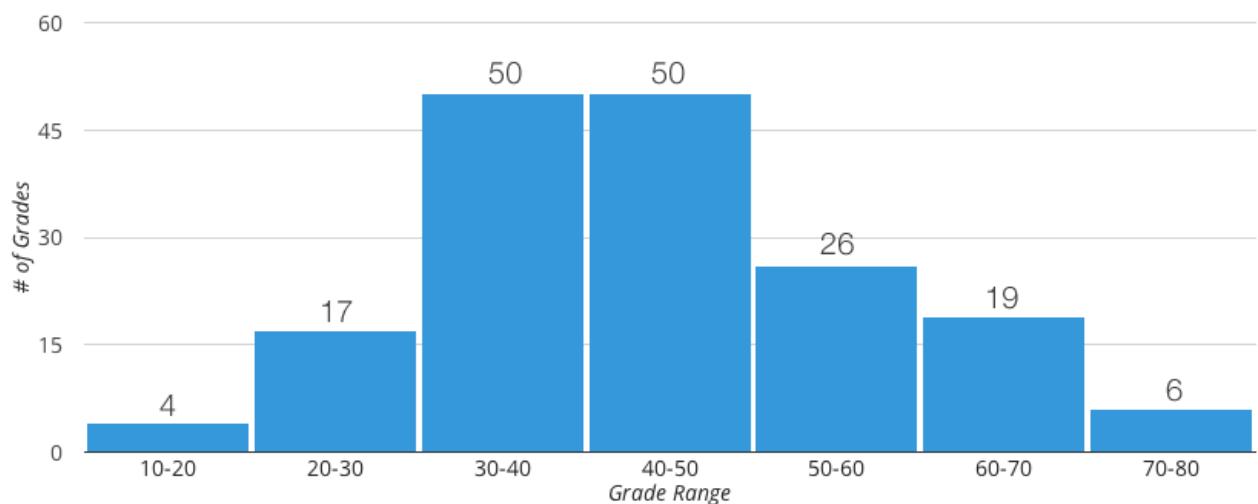
- (b)** [7 points] Some government employees are very nostalgic and make lots of queries to remember what the data structure was like in the past. Come up with another scheme for recording changes to the data structure such that it takes  $O(h \log t)$  time for both the generalized successor queries and the usual updates, where  $h$  is the maximum height of the tree.

**Solution:** Keep a balanced binary search tree (like a red-black tree) at each pointer so that you can search for the right version in  $O(\log t)$  per pointer.

## SCRATCH PAPER

## SCRATCH PAPER

## Quiz 1 Solutions



- Do not open this quiz booklet until you are directed to do so. Read all the instructions first.
- The quiz contains **5** problems. You have 80 minutes to earn **81** points.
- This quiz booklet contains **12** pages, including this one, and a sheet of scratch paper.
- This quiz is closed book. You may use one double-sided letter ( $8\frac{1}{2}'' \times 11''$ ) or A4 crib sheet. No calculators or programmable devices are permitted. Cell phones must be put away.
- Write your solutions in the space provided. If you run out of space, continue your answer on the back of the same sheet and make a notation on the front of the sheet.
- Do not waste time deriving facts that we have studied. Just cite results from class.
- When we ask you to “give an algorithm” in this quiz, describe your algorithm in English or pseudocode, and provide a short argument for correctness and running time. You do not need to provide a diagram or example unless it helps make your explanation clearer.
- Do not spend too much time on any one problem. Generally, a problem’s point value is an indication of how many minutes to spend on it.
- 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. Please be neat.
- Good luck!

Problem	Title	Points	Parts	Grade	Initials
0	Name	1	1		
1	True or False	20	10		
2	Short Answers	35	7		
3	Sequoia MST	10	1		
4	Histograms	10	2		
5	Bowtie	5	1		
Total		81			

Name: \_\_\_\_\_

F10 R01 Nathan	F11 R02 Nathan	F11 R07 Adam	F12 R03 Bonny	F12 R08 Adam	F1 R04 Bonny	F1 R09 Casey	F2 R05 Hayden	F2 R10 Christina	F3 R06 Hayden
----------------------	----------------------	--------------------	---------------------	--------------------	--------------------	--------------------	---------------------	------------------------	---------------------

**Problem 0. Name.** [1 points] Write your name on every page of this exam booklet! Don't forget the cover.

**Problem 1. True or False.** [20 points] (10 parts)

Circle **T** or **F** for each of the following statements to indicate whether the statement is true or false and briefly explain why.

- (a) **T F** For every possible input and every sequence of random coin tosses, a Las Vegas algorithm always runs in polynomial time. However, with some nonzero probability over the coin tosses, a Las Vegas algorithm may return an incorrect output.

**Solution:** [2 points] False. Las Vegas algorithms always return correct results, but their runtime is not bounded for every sequence of coin tosses.

- (b) **T F** Let  $A$  be a polynomial-time Monte Carlo algorithm such that

- If the correct answer is TRUE,  $A$  always outputs TRUE.
- If the correct answer is FALSE,  $A$  outputs FALSE with probability  $\frac{1}{10}$  and TRUE with probability  $\frac{9}{10}$ .

We say that  $A$  has a *one-sided error* of  $\frac{9}{10}$ .

It is always possible to construct a polynomial-time algorithm that solves the same problem as  $A$ , but has a one-sided error of at most  $\frac{1}{10}$ .

**Solution:** [2 points] True. This is possible via amplification. In particular, we run  $A$   $k$  times, for some constant  $k \geq 22$ . The probability that the amplified algorithm returns TRUE all  $k$  times, even if the correct answer is FALSE, is  $\leq (\frac{9}{10})^{22} < \frac{1}{10}$ . This is its one-sided error.

- (c) **T F** An adversary can construct an input of length  $n$  to force RANDOMIZED-QUICKSORT to run always in  $\Omega(n^2)$  time.

**Solution:** [2 points] False. The *expected* running time of RANDOMIZED-QUICKSORT is  $\Theta(n \log n)$ . This applies to *any* input.

- (d) **T F** Consider the following Monte Carlo algorithm for primality testing:

**Algorithm** FERMAT-TEST. On input  $N > 2$ :

1. Repeat the following twice:
  - (a) Pick a random integer  $A \in \mathbb{Z}_N^*$ .
  - (b) If  $A^{N-1} \neq 1 \pmod{N}$ , then return “composite”.
2. If the algorithm did not return “composite” in step 1, return “probably prime”.

The algorithm has a one-sided error of at most  $\frac{1}{2}$  on every input  $N$ .

**Solution:** [2 points] False. If  $N$  is a Carmichael number, the algorithm always returns “probably prime”. Therefore, the error probability for the case that  $N$  is a Carmichael number is 1.

- (e) **T F** Consider a graph  $G = (V, E)$  with  $n = |V|$  and  $m = |E|$ . Suppose someone discovers an  $O(nm/\log n + n^2 \log n)$  time algorithm for all-pairs shortest paths with non-negative edge weights. Then, we can directly use this in Johnson’s algorithm — in place of calling Dijkstra’s algorithm  $n$  times — to obtain a general all-pairs shortest paths algorithm that runs in  $O(nm/\log n + n^2 \log n)$  time.

**Solution:** [2 points] False. To use Johnson’s algorithm, we still need to run the Bellman-Ford algorithm, which would add an  $O(nm)$  term to the running time.

- (f) **T F** Consider the following algorithm for finding the maximum spanning tree in a connected graph with distinct edge weights:

**Algorithm** FIND-MAX-SPANNING-TREE. On input graph:

1. Greedily select the largest-weight edge that connects two unconnected components and add it to the forest.
2. Repeat step (1) until obtaining a tree.

This algorithm gives the correct maximum spanning tree.

**Solution:** [2 points] True. The algorithm is effectively Kruskal’s algorithm in reverse. Similarly, it is equivalent to making all edge weights negative and running Kruskal’s algorithm on the new graph.

- (g) **T F** Consider the max-flow problem in graphs whose capacities are real numbers in the range  $[0, 1]$ . For such inputs, the Ford-Fulkerson algorithm always terminates in a time that is polynomial in the number of vertices and edges.

**Solution:** [2 points] False. Although the capacities are bounded, they are not integral.

- (h) **T F** If a decision problem is in NP, then the best algorithm that solves it must run in at least exponential time.

**Solution:** [2 points] False. Any problem in P is also in NP. Thus, there are decision problems in NP that can be solved in polynomial time.

*Grading note:* Many students correctly answered False, but justified this by saying it is False because we do not know whether P = NP; these answers received partial credit.

- (i) **T F** The 6.046 staff does not know of a way to solve the traveling salesman problem in polynomial time.

**Solution:** [2 points] True. Such an algorithm would show that P = NP.

- (j) **T F** For a fixed constant  $k > 1$ , there exists a polynomial time algorithm to detect if there is a clique of size  $k$  in a graph.

**Solution:** [2 points] True. This can be done in  $O(n^k)$  time by iterating over  $O(n^k)$  subgraphs of the input graph, and checking whether the  $\binom{k}{2}$  edges of each subgraph form a clique.  $\binom{k}{2}$  is a constant, and thus does not appear in the asymptotic runtime. Note that this is polynomial in  $n$ , the input size; it is not exponential in  $k$  because  $k$  is a fixed constant.

*Grading note:* Most students answered False and justified this by saying CLIQUE is an NP-complete problem. However, when we fix the input  $k$  to be a constant, it is no longer NP-complete.

**Problem 2. Short Answers.** [35 points] (7 parts)

Please answer the following questions.

- (a) You are given a list of distinct integers  $x_1, x_2, \dots, x_n$ . Define an *inversion* as a pair of indices  $(i, j)$ , with  $i < j$ , such that  $x_i > x_j$ . In lecture you saw how to count the number of inversions in the list in  $\Theta(n \log n)$  time. Define a *super-inversion* as a pair of indices  $(i, j)$ , with  $i < j$ , such that  $x_i > x_j^2$ . Explain how you can modify the algorithm from lecture to count the number of super-inversions in the list without altering its runtime.

**Solution:** [5 points] We assume that the input list consists of distinct positive integers. Recall that this problem was solved by a divide-and-conquer algorithm that “piggy-backed” on merge sort. We split the list into two halves  $L$  and  $R$  and recursively ran the algorithm on each half. This “divide” and “conquer” step stays the same. However, the “combine” step of the algorithm must change. In particular, we need to perform two merges. Let  $R'$  consist of the squares of all the integers in  $R$ . We first merge  $L$  and  $R'$  in order to count the number of super-inversions, in the same way that the original algorithm merged  $L$  and  $R$  to count the number of inversions. The total number of super-inversions at the call is the sum of the number of super-inversions in  $L$ , the number in  $R$ , and the number counted when merging  $L$  with  $R'$ . Furthermore, we must merge  $L$  and  $R$  to obtain a sorted list of all the elements, which must be returned. Like in the original algorithm, in addition to returning the sorted list at each call, we also return the number of super-inversions.

*Grading note:* In this solution we assume that the input list were distinct positive integers. However, because this was not given in the problem, any student who attempted to solve the problem with negative integers or noted its difficulty received full credit.

- (b) You are given  $x$ ,  $y$ , and  $N$  such that  $x^2 \equiv y^2 \pmod{N}$ ,  $x \not\equiv y \pmod{N}$ , and  $x \not\equiv -y \pmod{N}$ . In a time that is polynomial in the size of the input, show how to find a divisor  $d$  of  $N$  such that  $d$  is not 1 or  $N$ .

**Solution:** [5 points] By the second and third statements,  $N$  does not divide  $x - y$  and  $N$  does not divide  $x + y$ . However, by the first statement,  $N$  divides  $x^2 - y^2 = (x - y)(x + y)$ . Thus,  $N$  has common factors with  $x - y$  and  $x + y$  that are not 1 or  $N$ . So we can compute  $\gcd(x - y, N)$  or  $\gcd(x + y, N)$ .

- (c) A *multiset* is a generalization of a set, where order is still ignored but elements may appear more than once. Let  $X$  and  $Y$  be multisets of size  $N$  made up of integers in the range  $[0, 8N]$ . Consider the multiset  $\{x - y \mid x \in X \text{ and } y \in Y\}$ . Give an efficient algorithm that finds the distinct elements in this multiset and the number of times each occurs.

**Solution:** [5 points] This problem is a small modification of **Homework Problem 1-4**. The approach we take is based on the equation  $x - y = [x + (8N - y)] - 8N$ . In particular, we will first compute the values  $x + (8N - y)$  and then subtract  $8N$  from those values. First, let  $Y' = \{8N - y \mid y \in Y\}$ . Notice that all of the values in  $Y'$  are nonnegative. Therefore, we can encode the values of  $X$  and  $Y'$  as polynomials with nonnegative exponents. In particular, let

$$A(x) = a_0 + a_1x + a_2x^2 + \cdots + a_{8N}x^{8N} \text{ and } B(x) = b_0 + b_1x + b_2x^2 + \cdots + b_{8N}x^{8N}.$$

where  $a_i$  is the number of occurrences of the value  $i$  in  $X$  and  $b_j$  is the number of occurrences of the value  $j$  in  $Y'$ . Let

$$C(x) = A(x) \cdot B(x) = c_0 + c_1x + c_2x^2 + \cdots + c_{16N}x^{16N}.$$

Since  $A(x)$  and  $B(x)$  are polynomials of degree  $\Theta(N)$ , we can compute the product  $C(x)$  in  $\Theta(N \log N)$  time with a fast Fourier transform. Note that  $c_k = \sum_{i=0}^k a_i b_{k-i}$ .

Therefore, the product polynomial encodes the values in the multiset  $\{x + y' \mid x \in X \text{ and } y' \in Y'\}$ , with each  $c_k$  giving the number of times the value  $k$  is in the multiset. This multiset is equivalent to  $\{x + (8N - y) \mid x \in X \text{ and } y \in Y\}$ . Now, we subtract  $8N$  from all the resulting values to determine the values in  $\{x - y \mid x \in X \text{ and } y \in Y\}$ .

*Grading note:* Simply negating the values of  $Y$  and encoding this as a polynomial would require multiplying a polynomial by one with negative exponents. The FFT as described in lecture requires that exponents be nonnegative. Also, some students gave a correct  $\Theta(n^2)$  algorithm; those who did so received partial credit.

- (d) Let set  $S = \{A \in Z_N^* \mid A = x^{(N-1)/2} \pmod{N}, x \in \{0, \dots, N-1\}\}$  for a prime  $N$ . Show that  $S$  is a sub-group of  $Z_N^*$ . What is the size of set  $S$ ?

**Solution:** [5 points] Because  $A = x^{(N-1)/2} \pmod{N}$ , it follows that  $A^2 = x^{(N-1)}$   $\pmod{N}$ . By Fermat's Little Theorem,  $x^{(N-1)} = 1 \pmod{N}$ , so  $A^2 = 1 \pmod{N}$ . By the Modular Square Roots Theorem, this has two solutions:  $A = 1 \pmod{N}$  and  $A = -1 \pmod{N}$ . Therefore,  $S = \{1, -1\}$ . (Equivalently,  $S = \{1, N-1\}$ .) So the size of  $S$  is 2. It is now easy to show that  $S$  is a subgroup under multiplication mod  $N$  by showing it satisfies the group axioms. Namely,

- **identity:**  $1 \in S$ .

- **closure:**  $(1)(1) \in S$ ,  $(1)(-1) \in S$ , and  $(-1)(-1) \in S$ .
- **inverse:** 1 has the inverse 1 because  $(1)(1) = 1 \pmod{N}$  and  $-1$  has the inverse  $-1$  because  $(-1)(-1) = 1 \pmod{N}$ .

- (e) Suppose that you are given an array  $A$  of  $n$  bits that either contains all zeros or contains  $3n/4$  zeros and  $n/4$  ones in some arbitrary order. Your goal is to determine whether  $A$  contains any ones.
1. Give an exact lower bound in terms of  $n$  (**not** using asymptotic notation) on the worst-case running time of any deterministic algorithm that solves this problem.
  2. Give a randomized algorithm that runs in  $O(1)$  time and gives the right answer with probability at least  $1/4$ .
  3. Give a randomized algorithm that runs in  $O(1)$  time and gives the right answer with probability at least  $37/64$ .

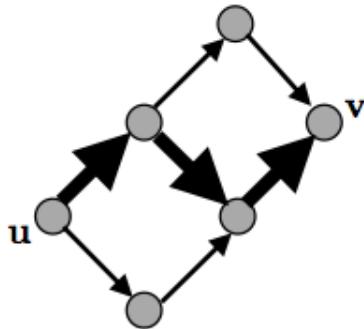
**Solution:** [5 points]

1. Any correct deterministic algorithm must look at  $3n/4 + 1$  entries in the worst-case. If it looks at any fewer, it may come across all zeros even if there exists a one somewhere.
2. Pick a random location. If it is a 1, output “has ones”. Otherwise, output “doesn’t have ones”. Clearly the algorithm runs in  $O(1)$  time. If the array is all zeros, it is always correct.  
However, if the array contains  $n/4$  ones, the algorithm may make a mistake. Since there are  $n/4$  ones, it will select a 1 value at random and correctly output “has ones” with probability  $1/4$ .
3. Pick three random locations. If any of the three is a 1, output “has ones”. Otherwise, output “doesn’t have ones”. Like above, the algorithm runs in  $O(1)$  time. Again, if the array is all zeros, the algorithm is always correct.  
A mistake arises if the array contains ones, but the algorithm picks three zeros. Picking a single zero occurs with probability  $3/4$ , so picking three zeros independently occurs with probability  $(3/4)^3 = 27/64$ . Therefore, the probability that the algorithm is correct is  $1 - 27/64 = 37/64$ .

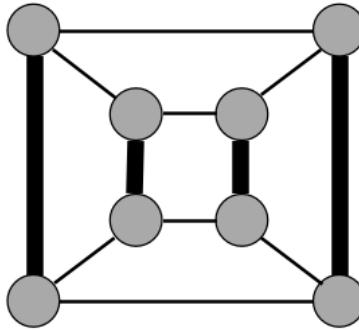
- (f) In the disjoint paths problem, you are given a directed, unweighted graph  $G = (V, E)$ , two vertices  $u \in V$  and  $v \in V$ , and an integer  $k$ . The problem is to decide whether there are  $k$  paths from  $u$  to  $v$  whose edges do not overlap. Give a polynomial-time algorithm to solve this problem.

**Solution:** [5 points] We can solve this by transforming the problem into a network-flow problem. In particular, add an edge from the source to  $u$  with capacity  $k$ , and add an edge from  $v$  to the sink with capacity  $k$ . Give each edge in  $G$  a unit capacity. A flow in this network must travel through non-overlapping edges in paths from  $u$  to  $v$ . Thus, we can run a max-flow algorithm on the network and report that there are  $k$  paths from  $u$  to  $v$  if and only if the max flow is at least  $k$ . Using the Edmonds-Karp algorithm, the runtime is  $O(VE^2)$ .

*Grading note:* One of the most common mistakes was searching the graph, often with BFS or DFS, to find paths from  $u$  to  $v$ . The algorithm would then remove edges along a path that it finds, and iterate again looking for another path. It would return TRUE if  $k$  paths are found. However, there are graphs on which this algorithm works incorrectly. Consider the below counterexample:



There are clearly 2 paths from  $u$  to  $v$  whose edges do not overlap, so the problem is TRUE for  $k = 2$ . However, a search of the graph may first find the bolded path and then remove its edges. On the next iteration, the algorithm would not find a path from  $u$  to  $v$ , and would therefore incorrectly report that the answer is FALSE for  $k = 2$ .



- (g) Recall that a *matching* of a graph  $G = (V, E)$  is a subset of the edges such that no two edges are incident on the same vertex. In a *perfect matching*, every vertex in the graph is incident to exactly one edge in the matching. For example, in the below graph the bolded edges form a perfect matching:

Given an oracle that decides whether a graph contains a perfect matching, show how you can find a perfect matching in a graph (or determine that one doesn't exist) in  $O(|E|)$  calls to the oracle.

**Solution:** [5 points] We can construct a Cook-reduction of the perfect matching search problem to the decision problem as follows:

1. If the oracle decides there is no perfect matching, return NONE.
2. For every edge  $e_1, e_2, \dots, e_n$  of the graph in an arbitrary order:
  - (a) Modify the graph by removing edge  $e_i$ .
  - (b) If the oracle decides there is still a perfect matching, then the perfect matching must not involve  $e_i$ . Therefore, permanently remove edge  $e_i$ .
  - (c) If the oracle decides there is not a perfect matching, then  $e_i$  must be in the perfect matching. So add  $e_i$  back to the graph.
3. The edges that remain in the graph form the perfect matching.

There are at most  $|E| + 1$  calls to the oracle, which is  $O(|E|)$ .

*Grading note:* The most common mistake was forgetting to include the case where there is no perfect matching in the original graph. Before iterating over edges, it is necessary to call the decision oracle and return NONE if there is no perfect matching. Another frequent mistake was casting this as a bipartite matching problem; you cannot assume that the given graph is bipartite.

**Problem 3. Sequoia MST [10 points]**

Sequoia MST Inc. aims to develop a very efficient algorithm for computing huge minimum spanning trees. Specifically, the input graphs  $G = (V, E)$  are so huge that the algorithm cannot even store the whole graph in memory. Instead, the graphs are stored on a read-only disk, and the algorithm can make only a constant number of passes over the whole disk. Moreover, the algorithm can only store at most  $O(V)$  units of data (vertices, edges, pointers, counters, etc.) in the memory, so it cannot simply transfer the edges from disk to memory — there isn't enough space. Fortunately, the marketing department observed that the customer graphs are very flat: all edge weights are in the set  $\{1, 2, 3, 4, 5\}$ .

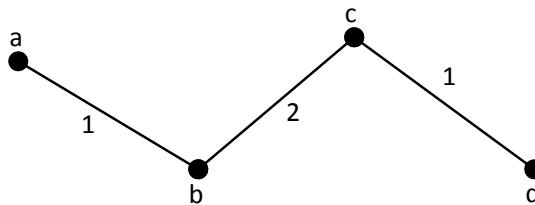
Give an algorithm that computes the MST for such graphs, using only  $O(V)$  space and  $O(1)$  read-only passes over the data.

**Solution:** We essentially adapt Kruskal's algorithm and run only for 5 iterations. Perform a first pass over the dataset to obtain  $|V|$ . Next, set up a disjoint sets data structure with one set for each vertex. Perform a second pass over the dataset and unite any disjoint sets connected by a edge of weight 1. Repeat this process 4 more times for the weights 2, 3, 4, and 5. Since edges were added in order of increasing weight in a manner identical to that of Kruskal's algorithm, this algorithm performs yields an identical spanning tree to Kruskal's which we know to be a correct algorithm for finding the MST. Therefore, our algorithm is correct and uses  $O(V)$  space and  $O(1)$  passes over the data.

*Grading note:* We can only access data from the disk, but we cannot “modify” the graph without loading it into memory. Thus, we cannot “remove” nodes or edges from the graph, unless you specify how you store the graph in memory with only  $O(V)$  space.

Prim's algorithm builds the tree beginning with a *single* vertex. This could use up to  $O(V)$  passes over the data, since every time we add an edge, we need to make a pass over the data to update the boundary (i.e. performing decrease-key for neighbors). Thus Prim's doesn't work.

In running Kruskal's algorithm, we are growing a forest from multiple points. Thus we are concerned about connectivity rather than “discovered” vertices. A common mistake was to include an edge if either of the two endpoints were not yet “discovered” rather than connected (in the forest). This could result in a solution that is not a connected tree. For example in the following graph, such an algorithm would “miss out” on the edge (b,c).



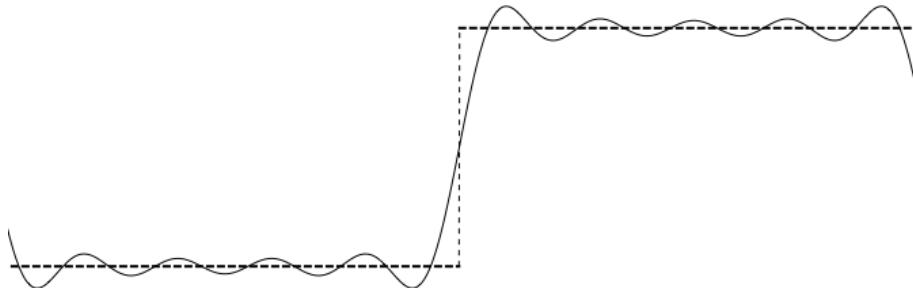
**Problem 4. Histograms [10 points] (2 parts)**

A sequence of numbers  $h[j]$ , for  $j = 1, \dots, n$ , is a  $k$ -histogram if it can be represented by  $k$  disjoint intervals  $I_1, \dots, I_k$  and  $k$  values  $v_1, \dots, v_k$  such that for any  $j = 1, \dots, n$ , if  $j \in I_i$ , then  $h[j] = v_i$ . Histograms that approximate numerical sequences are often easier to display and comprehend than the original sequences.

Given a sequence  $x[j]$  and positive integer  $k$ , we say that a  $k$ -histogram  $h[j]$  provides the optimal solution if it minimizes the square error

$$\sum_{j=1}^n (x[j] - h[j])^2.$$

For example, the graphic below shows a sequence (solid) and an optimal 2-histogram (dashed).



In this problem you will design a polynomial-time algorithm that finds the optimal  $k$ -histogram.

- (a) Assume  $k = 1$ . Show that an interval  $I_1 = \{1 \dots n\}$  with value  $v_1 = \frac{\sum_j x[j]}{n}$  provides the optimal solution.

**Solution:** For  $k = 1$ ,  $I_1$  must contain all the points, so  $I_1 = \{1 \dots n\}$ . We wish to choose  $v_1$  so that  $\sum_{j=1}^n (x[j] - v_1)^2$  is minimized. We set

$$\frac{d}{dv_1} \left( \sum_{j=1}^n (x[j] - v_1)^2 \right) = 0.$$

Simplifying, we obtain

$$\frac{d}{dv_1} \left( \sum_{j=1}^n x[j]^2 - 2v_1 \sum_{j=1}^n x[j] + v_1^2 \right) = -2 \sum_{j=1}^n x[j] + 2nv_1 = 0.$$

Solving for  $v_1$ , we have

$$v_1 = \frac{\sum_j x[j]}{n}.$$

Since the error is quadratic and convex in  $v_1$  and  $\frac{\sum_j x[j]}{n}$  is a local extremum,  $\frac{\sum_j x[j]}{n}$  yields the globally minimum error.

*Grading note:* This part was worth 3 points. In order to receive full credit here, it was necessary to take the derivative and actually show that  $v_1$  gives the optimal solution. 1 point was awarded for answers along the lines of “it must be minimal because  $v_1$  is the average”.

- (b) Assume  $k > 1$ . Show how to find the optimal  $k$ -histogram using dynamic programming.

**Solution:** For any given interval, the optimal value assigned to that interval is the mean of the subsequence that lies within that interval. We denote by  $H(l, m)$  the optimal histogram of  $l$  intervals on the first  $m$  points. We will choose  $v_i = \text{mean}(I_i)$  for all  $i$ , as any other choice of  $v_i$  is suboptimal for that interval and therefore for the entire histogram. We construct our recurrence for  $H(l, m)$ . Our base case from part a) is

$$\begin{aligned} H(1, m) &= \{I_1 = [1, m] \forall m \in \{1, \dots, n\}\} \\ E(1, m) &= \text{error}(H(1, m)). \end{aligned}$$

Next, we initialize all other values to

$$E(l, m) = \infty \quad \forall m \in \{1, \dots, l\}.$$

Then,

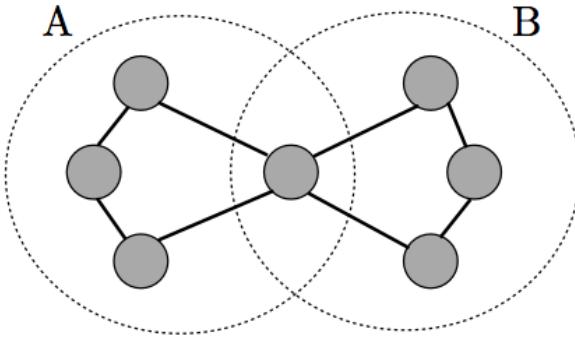
$$\begin{aligned} c^* &= \arg \min_{c \in [1, m]} \{E(l - 1, c - 1) + \text{error}([c, m])\} \\ H(l, m) &= \{H(l - 1, c^* - 1); I_l = [c^*, n]\} \\ E(l, m) &= \text{error}(H(l, m)). \end{aligned}$$

We proceed with bottom-up DP described above to compute  $H(k, n)$ . There are  $kn$  values of  $H(l, m)$  to compute, each of which takes  $O(n)$  time. So the algorithm takes  $O(kn^2)$  time.

*Grading note:* This part was worth 7 points. Aside from most students writing an incorrect recurrence, most students failed to state the base case (worth 2 points) or list the runtime (worth 1 point). Another common mistake included giving greedy algorithms for either merging histograms with  $n$  initial bars or splitting a single histogram greedily at each step. A counterexample for this is the input  $x = [2, 4, 6, 8, 10, 12]$ . For merging histograms,  $k = 2$  gives a counterexample. For splitting a single histogram greedily,  $k = 3$  gives a counterexample.

**Problem 5. Bowtie [5 points]**

Let  $G = (V, E)$  be a graph. Assume that  $|V|$  is odd. We say that  $G$  contains a *bowtie* if we can construct two subsets of  $V$ , called  $A$  and  $B$ , each of size  $\lceil \frac{|V|}{2} \rceil$ , such that (i) their union is the set of all vertices, (ii) their intersection is exactly one vertex, and (iii) each contains a cycle of size  $\lceil \frac{|V|}{2} \rceil$ . For example, the graph below contains a bowtie:



Show that the problem of whether  $G$  contains a bowtie is NP-complete.

**Solution:** First, we show that bowtie is in NP. Let  $G$  be an input that contains a bowtie, and let  $y$  be two lists of the vertices contained in  $A$  and  $B$  respectively, with each list ordered so that a vertex comes in between two of its neighbors in the cycle. Each list of vertices can be verified to be a cycle in linear time, and the unions and intersections of the two lists can be computed in quadratic time. Thus, bowtie is in NP.

Next, we show that bowtie is in NP-Hard by reduction from Hamiltonian Cycle. Let  $G$  be an input to Hamiltonian Cycle, and let  $G'$  be constructed from  $G$  as follows. Construct two copies of  $G$  which we call  $G_1$  and  $G_2$ , remove corresponding vertices  $v_1$  and  $v_2$  from  $G_1$  and  $G_2$  respectively, and draw edges from a new vertex  $v$  to the neighbors of both  $v_1$  and  $v_2$ . If  $G$  has a Hamiltonian cycle, then  $G'$  has a bowtie since conditions i), ii), and iii) are satisfied by construction. Note that  $G_1$  and  $G_2$  are the only connected subgraphs of  $G'$  that are of size  $\lceil \frac{|V|}{2} \rceil$  and whose intersection contains exactly one vertex. Thus if  $G'$  has a bowtie, then  $G_1$  and  $G_2$  must contain cycles of size  $\lceil \frac{|V|}{2} \rceil$ , and since this cycle is equal to the size of  $G$ , it must be a Hamiltonian cycle in  $G$ .

## SCRATCH PAPER

## **Quiz 1 Solutions**

**Problem 0. Name.** [1 point] Write your name on every page of this exam booklet! Don't forget the cover.

**Problem 1. True or False.** [15 points] (5 parts)

Circle **T** or **F** for each of the following statements to indicate whether the statement is true or false, and briefly explain why.

- (a) **T F** Strassen's algorithm for multiplying two  $n$ -by- $n$  matrices reduces the problem to 7 multiplications and 18 additions of  $n/2$ -by- $n/2$  matrices. If we devised an algorithm for multiplying two  $n$ -by- $n$  matrices by reducing it to 5 multiplications and 125 additions of  $n/2$ -by- $n/2$  matrices, its running time would be  $\Theta(n^2)$ .

**Solution:** [3 points] False. The runtime would be  $\Theta(n^{\log_2 5})$ . Note: Some people wrote that there is no known  $\Theta(n^2)$  algorithm for this problem, which is true. However, the problem stipulates a hypothetical, so the fact that no  $\Theta(n^2)$  algorithm is yet known does not address this question.

- (b) **T F** The deterministic algorithm for order statistics (selecting the  $k$ th smallest item) shown in lecture would have worst case running time  $\Omega(n \log n)$  if the initial step partitioned the  $n$  items into groups of 7 rather than 5.

**Solution:** [3 points] False. As mentioned in class and seen in Exercise 4 in Pset 1, the running time remains  $\Theta(n)$ .

- (c) **T F** Suppose that we use the FFT to multiply two polynomials. The forward and inverse transforms take asymptotically the same amount of time.

**Solution:** [3 points] True. The matrix for the inverse discrete Fourier transform was just a rearrangement of the rows of the Fourier transform matrix. This means that applying the inverse transform can be done by rearranging the vector and then applying the usual Fourier transform.

- (d) **T F** Let  $G = (V, E)$  be an undirected graph with four vertices  $V = \{0, 1, 2, 3\}$  and four edges  $E = \{\{0, 1\}, \{1, 2\}, \{2, 3\}, \{3, 0\}\}$  (i.e., there are no self-loops). Start a random walk at vertex 0 with probability  $p$  and at vertex 1 with probability  $1 - p$ . In other words, the initial distribution  $x_0$  is  $(p, 1 - p, 0, 0)$ . Then  $x_t$ , the distribution after  $t$  steps, does not converge as  $t \rightarrow \infty$  for any choice of  $p \in [0, 1]$ .

**Solution:** [3 points] False. It converges in two steps for  $p = 0.5$ .

- (e) **T F** Consider the problem of sorting  $n$  elements by inserting them into an initially empty binary search tree and then performing an inorder traversal of that tree. The worst-case running time of this sorting algorithm will be the same asymptotically if we use AVL trees or splay trees.

**Solution:** [3 points] True. For both AVL and splay trees, the *total* time for the  $n$  insertions is  $\Theta(n \log n)$ . The inorder traversal takes  $\Theta(n)$  time regardless of the type of tree. Note that for splay trees, the amortized cost of a single operation is  $\Theta(\log n)$ , but the total cost of  $n$  operations is worst case  $\Theta(n \log n)$ . This is a fundamental property of amortization! *Note:* Some people parsed this problem in a way that we had not expected. They assumed that there are actually three types of trees being considered here: Unbalanced binary search trees, AVL trees, and splay trees. If we discerned that you gave the right answer for this interpretation, we gave you full credit.

**Problem 2. Testing Array Equality** [10 points] (2 parts) You are given two arrays  $A$  and  $B$ , each of length  $n$ . Assume that you can access the  $i$ th element of an array in  $O(1)$  time and can determine whether two elements are equal in  $O(1)$  time. The arrays are said to *differ* at exactly  $k$  indices if there exist  $k$  distinct indices  $i$  such that at each such  $i$ ,  $A[i] \neq B[i]$ , and  $n - k$  distinct indices  $j$  such that at each such  $j$ ,  $A[j] = B[j]$ . Suppose that  $A$  and  $B$  are either equal (they differ at 0 indices) or they differ at exactly  $n/3$  indices. You wish to determine whether  $A$  equals  $B$ . Assume that  $n/3$  is an integer.

- (a) Give an **exact** lower bound in terms of  $n$  (i.e., without asymptotic notation) on the number of elements that any **deterministic** algorithm for this problem needs to read from both arrays in the worst-case.

**Solution:** Any deterministic algorithm must compare  $2n/3 + 1$  indices in the worst-case. If it looks at any fewer, it may encounter only indices whose elements are equal even if there are differences elsewhere. Since each comparison requires a read from each of the two arrays, there are  $4n/3 + 2$  reads.

*Comments:* Many students compared only  $2n/3$  indices. However, when  $A$  and  $B$  are not equal there are  $2n/3$  indices  $i$  such that at each  $i$ ,  $A[i] = B[i]$ . Thus, if one only compares  $2n/3$  indices, it is possible to conclude that  $A$  and  $B$  are equal when in fact they are not.

- (b) Give a Monte Carlo randomized algorithm that runs in  $O(1)$  time and gives the right answer with probability at least  $5/9$ . Argue that your algorithm meets this error bound.

**Solution:** Pick two random indices  $i$  and  $j$ , and compare  $A[i]$  with  $B[i]$  and  $A[j]$  with  $B[j]$ . If either pair differs, output that the arrays are not equal. Otherwise, output that they are equal. The algorithm clearly runs in  $O(1)$  time. If the two arrays are equal, it is always correct. If the arrays differ at  $n/3$  indices, the algorithm makes an error when the two indices it picks are among the  $2n/3$  whose elements are equal. This occurs with probability  $(2/3)^2 = 4/9$ . Therefore, the probability that the algorithm is correct in this case is  $1 - 4/9 = 5/9$ .

*Comments:* Many students gave an algorithm that compares just one randomly selected index  $i$  and returns “equal” iff  $A[i] = B[i]$ . When  $A = B$  this algorithm is correct. However, when  $A \neq B$ , this algorithm is only correct with probability  $1/3$ , and thus one cannot make a valid argument that the algorithm is correct overall with probability  $5/9$ . It is necessary to “amplify” the algorithm to achieve the error bound.

**Problem 3. Van Emde Boas Space** [10 points] (1 parts) The goal of this problem is to determine the space required for a van Emde Boas tree (vEB). Suppose that in a given implementation of vEBs, the space required for a vEB with universe size  $U = 2^{2^k}$  with  $k \geq 1$  is the space required for the summary vEB and the cluster vEBs, plus  $c\sqrt{U}$  overhead. Further, suppose that in this implementation, a vEB with universe size  $U = 2$  values takes  $d$  space. Using induction, prove that for any integer  $k \geq 0$ , such a Van Emde Boas tree with universe size  $U = 2^{2^k}$  uses at most  $(c + d)U - (2c + d)$  space.

**Solution:** In the base case,  $k = 0$  and  $U = 2$ . Thus, by assumption, the vEB takes  $d$  space, which equals  $2(c+d) - (2c+d)$ . Now assume by induction that a vEB with universe size  $2^{2^k}$  uses at most  $(c+d)U - (2c+d)$  space. The space required by a vEB with  $U = 2^{2^{k+1}}$  is the space for the summary vEB, the cluster vEBs, and  $c\sqrt{U}$  overhead. The summary and cluster vEBs all have universe size  $\sqrt{U} = 2^{2^k}$ , so by the induction assumption they each take  $(c+d)\sqrt{U} - (2c+d)$  space. There are  $\sqrt{U}$  cluster vEBs and one summary vEB, so the total space requirement is  $(\sqrt{U}+1)((c+d)\sqrt{U} - (2c+d))$  plus  $c\sqrt{U}$  overhead, which equals  $(c+d)U + (c+d)\sqrt{U} - (2c+d)\sqrt{U} - (2c+d) + c\sqrt{U} = (c+d)U - (2c+d)$ . Thus, by induction,  $(c+d)U - (2c+d)$  space is used for all  $U = 2^{2^k}$ .

Common mistakes included forgetting about the summary vEB or assuming that the summary vEB takes  $\sqrt{U}$  space instead of  $(c+d)\sqrt{U} - (2c+d)$  space. Some students forgot to add the overhead. Still others considered only one cluster vEB instead of  $\sqrt{U}$ . Students that got all these details right but messed up the induction or made an arithmetic mistake got most of the points on the problem.

**Problem 4. Non-universal Hashing** [7 points] (1 parts) Let  $m$  be an integer greater than 1 and let  $[m] = \{0, 1, \dots, 2^m - 1\}$ . Let  $U = \{0, 1, 2, \dots, 2^m - 1\}$  be the universe of keys. Let  $\mathcal{H}$  be the collection of hash functions  $\{h_{a,b}\}_{(a,b) \in [m] \times [m]}$ , where  $h_{a,b}(x) \triangleq ax + b \pmod{2^m}$ . Let  $h$  be drawn uniformly at random from  $\mathcal{H}$ . Find a pair  $x, y \in U, x \neq y$ , such that  $Pr_{h \in \mathcal{H}}(h(x) = h(y)) \geq 1/2$ . This shows that when  $m > 1$ , this hash family is not universal.

**Solution:** Pick  $x = 0, y = 2^{m-1}$ . If  $a$  is an even number, let  $a = 2k$  where  $k$  is an integer. Then

$$ax + b = b \equiv 2^m \times k + b = ay + b \pmod{2^m}.$$

The probability that  $a$  is an even number is 0.5. Therefore,  $Pr_{h \in \mathcal{H}}(h(x) = h(y)) \geq 1/2$  for  $x = 0, y = 2^{m-1}$ .

**Problem 5. Using FFT for Exponentiation** [10 points] (1 parts) Given the coefficients of a degree  $n$  polynomial  $p(x)$ , how would you use FFT to find the coefficients of  $(p(x))^k$  using  $O(kn \log kn)$  arithmetic operations? (By arithmetic operations, we mean addition, subtraction, multiplication, or division of real or complex numbers.)

**Solution:** Note that the degree of  $p(x)^k$  will be  $kn$ . We will therefore start by applying the discrete Fourier transform using a primitive  $(kn + 1)$ th root of unity  $\omega$ . This takes  $O(kn \log kn)$  time, and results in the values of  $p(\omega^i)$  for  $i = 0, 1, \dots, kn$ .

Next, we raise each of these values to the  $k$ -th power. We can use repeated squaring to do this: to calculate  $c^k$ , first calculate  $c^{2^i}$  for  $i = 0, 1, \dots, \lfloor \log_2 k \rfloor$  by repeatedly squaring  $c$ , and then multiply together the values of  $c^{2^i}$  for  $i$  corresponding to the ones occurring in the binary expansion of  $k$ . This takes  $O(\log k)$  arithmetic operations per value, for a total of  $O(kn \log k)$  for all  $kn + 1$  values.

Finally, we use the inverse Fourier transform to get a polynomial of degree at most  $kn$  whose values on the roots of unity match  $p(\omega^i)^k$  for all  $i = 0, 1, \dots, kn$ . This polynomial must equal  $p(x)^k$ , since a degree  $kn$  polynomial is uniquely determined by its values on  $kn + 1$  points. The inverse Fourier transform again takes  $O(kn \log kn)$  arithmetic operations, so the total number of operations required for all the steps is  $O(kn \log kn)$ .

Comments: Most students are able to solve this problem. A common mistake is that we need to evaluate the polynomial of degree  $kn$  at  $(kn+1)$  roots of unity, instead of at  $kn$  roots of unity, in order to determine its  $(kn+1)$  coefficients. In the lecture, our polynomial has a degree of  $(n-1)$ , and hence we evaluated at  $n$  points. Some students used repeated squaring to solve this problem. This is a decent alternative method, but it is important to note that we cannot treat  $k$  as  $O(1)$  and claim  $O(k \log(k) n \log(kn)) = O(kn \log(kn))$ .

**Problem 6. Flight Scheduling** [10 points] (1 parts) You're trying to take the maximum number of round trip flights from Boston (BOS) to Raleigh (RDW) and back in a single month. You are given a list of  $n$  flights represented by  $(d_i, a_i, x_i)$  where  $d_i$  is the departure date and time,  $a_i$  is the arrival date and time, and  $x_i = 0$  if the flight is BOS→RDW and  $x_i = 1$  if the flight is RDW→BOS.

Devise an efficient polynomial time algorithm which, given these flights, returns an itinerary (list of flights) that starts and ends in Boston, and maximizes the total number of round trip flights and prove that your algorithm is correct.

**Solution:** [5 points] The algorithm is similar to interval scheduling: Make two sorted lists  $B$  and  $R$ .  $B$  will contain the flights departing BOS, sorted by arrival time, and  $R$  will contain the flights departing RDW, sorted by arrival time. Start at the beginning of  $B$  and take the first flight. Now in  $R$ , skip over all flights which have already departed. Take the first flight remaining. Continue in this fashion, alternating between  $B$  and  $R$ , taking the next available flight (earliest arrival time) at each location.

Sorting the flights takes  $O(n \log n)$  time. The remainder of the algorithm simply goes through both arrays in order, and thus takes  $O(n)$  time. Therefore, the total running time is  $O(n \log n)$ .

To prove correctness, let  $OPT$  be the optimal itinerary. Let  $A$  be the itinerary our algorithm finds. We will prove  $|OPT| = |A|$  by strong induction on  $n$ . Assume the claim is true for all  $k < n$ . Now consider our algorithm on  $n$ . Let the first flight in  $A$  takes be  $f_A = (d_A, a_A, 0)$  and let the first flight in  $OPT$  be  $f_{OPT} = (d_{OPT}, a_{OPT}, 0)$ . It is clear that  $a_A \leq a_{OPT}$ , thus  $a_A$  is compatible with all remaining flights in  $OPT$ . By induction and symmetry (switch BOS and RDW),  $|OPT - f_{OPT}| = |A - f_A|$ . Therefore,  $|OPT| = |A|$ , as desired.

Comments: Many students solved this problem correctly. Some recurrent mistakes were: (1) failing to recognize that Greedy applies and trying an  $O(n^2)$  DP solution. (2) failing to sort, and taking  $O(n)$  time to search through the list for the earliest-ending compatible interval, resulting in  $O(n^2)$  time. (3) even when implementing the sorting and linear scanning, failing to recognize (in the runtime analysis part), that the traversal is  $O(n)$  over \*all\* iterations (given the single traversal of each element in the list), as they are all traversed once, even though it's possibly  $\Theta(n)$  at each iteration. (4) Some students sorted one list by arrival and the other by departure, when both lists should have been sorted by arrival. (5) Some students forgot to include a runtime analysis, which was requested at the beginning of the exam for all problems and needed to be included. You should always include a runtime analysis when presenting an algorithm. (6) Some students tried to first construct all pairs of flights, but any such algorithm is  $O(n^2)$  and typically  $O(n^2 \log n)$  if the pairs are sorted.

**Problem 7. Quasi-Sorting** [10 points] (1 parts)

You are given  $n$  distinct numbers and an integer  $k > 1$  with  $n$  divisible by  $k$ . Using only comparisons, you want to partition the numbers into  $k$  equally sized disjoint sets  $S_1, S_2, \dots, S_k$  such that for any  $i < j$  and any  $s_i \in S_i, s_j \in S_j$  you are guaranteed  $s_i < s_j$ .

Devise an algorithm running in  $\Theta(n \log k)$  time that partitions the numbers as desired. You may assume  $k$  is a power of 2, if it helps make your runtime analysis easier.

**Solution:** [10 points] The algorithm employs divide and conquer on  $n$  and  $k$ . Find the median and partition the numbers around it in  $\Theta(n)$  time. Now we just need to partition each side of  $n/2$  elements into  $k/2$  sets, recursively. The runtime recurrence is  $T(n, k) = 2T(n/2, k/2) + \Theta(n)$ . Note that this recursion has  $\log k$  levels, each of which is  $\Theta(n)$  work. Thus it solves to  $\Theta(n \log k)$ .

Some students wrote a solution concerning partitioning the numbers into  $k$  sets arbitrarily, sorting (or otherwise partitioning) these sets, and then somehow massaging the sets to form the desired  $S_1, \dots, S_k$ . Unfortunately, any way you do this, it either yields incorrect sets  $S_1, \dots, S_k$ , or is equivalent to sorting all the numbers, which takes  $\Theta(n \log n)$  time.

**Problem 8. Filtered Queues [10 points] (1 parts)** A filtered queue is a special kind of queue that supports the following operations:

- ENQUEUE( $x$ ): Enqueue the element  $x$
- DEQUEUE(): Dequeue the head element
- FILTER(): For every element in the queue, dequeue it and then immediately enqueue it, except that every 5th element that is dequeued is removed (not enqueue again).

Using an amortization method of your choice, show that any sequence of  $n$  of these operations has a total cost of  $O(n)$ . That is, the amortized cost per operation is constant.

**Solution:** [10 points] We can use an accounting scheme in which each enqueue operation pays 6 rubles, one for the actual enqueueing cost and 5 for the cost of a future filter operation for itself and the next the four elements after it. A dequeue pays 1 for its actual work. A filter pays 4 rubles for the cost of filtering the first four elements in the queue. Every element in the queue has a 5 rubles that were prepaid by an enqueue. Therefore, when we filter, the first four elements are paid for by the filter operation itself and every fifth element is removed and its rubles are used to pay for the cost of dequeuing and re-enqueueing the next four elements.

A common mistake was to claim that an element can survive at most 5 filters and to use this in the accounting scheme - this claim is incorrect.

Alternatively, we can use the potential method with

$$\Phi_i = 5 \cdot \#(\text{elements at time } i).$$

Note that this potential function is initially 0 and is non-negative at the end, as required. The enqueue and dequeue operations have constant amortized cost for this potential function since their actual cost is 1 and they change the size of the queue by plus or minus 1, thus contributing a change in the potential that is plus or minus 5. Suppose that the filtered queue has  $k$  elements at time  $i - 1$ . A filter has actual cost  $c_i = k$  and decreases the size of the data structure by  $k/5$ , so the filter has amortized cost:

$$\hat{c}_i = c_i + \Phi_i - \Phi_{i-1} = k + 5 \cdot (4k/5) - 5k = O(1),$$

as desired.

**Problem 9. Binary Counters with Reset** [10 points] (1 parts) In class, we examined a binary counter that starts with value 0. We showed that a sequence of  $n$  consecutive INCREMENT operations takes a total of  $O(n)$  time. Now, we'd like to add a second operation called RESET that resets the counter to have value 0. To facilitate this, we'll maintain a pointer to the most significant bit in the counter.

Define an appropriate potential function and use it to show that the amortized cost of operations INCREMENT and RESET is  $O(1)$  (assuming that the counter is initially 0).

**Solution:** Let  $s$  denote the index of the most significant bit in the counter (where the least significant bit has index 0) and let  $\ell$  denote the number of 1's in the counter. Define  $\Phi = s + \ell$ . Note that this potential function is initially 0 and is non-negative at the end, as required. An INCREMENT operation takes time  $k + 1$  where  $k$  denotes the number of consecutive 1's beginning in the least significant bit. Then, the amortized cost of an INCREMENT is  $k + \Delta\Phi$ . The potential function decreases by  $k$  (because those  $k$  1's are changed to 0's), increases by 1 due to a 0 flipped to a 1, and possibly increases by 1 more due to moving the most significant bit one position to the left. In total, this results in an amortized cost of 2. A RESET has actual cost  $s$  but reduces the potential by  $(s + \ell)$ , resulting in amortized cost that is at most 0.

*Note:* Common errors were to use just the number of 1's or just the index of the most significant bit as the potential function. In the former case, the analysis works fine for an INCREMENT. However, resetting is a problem in this case because the actual cost of resetting is proportional to the index of the most significant bit, since we have to examine each bit up to that point to check if it is a 0 or 1 and flip it if it's a 1. But, the potential might only drop by 1 in this case (if the only 1 in the counter is the one at the most significant bit). Using the index of the most significant bit works fine for the RESET operation but not for an INCREMENT. To see this, note that an increment could incur some large cost  $k$  if there are  $k - 1$  consecutive 1's in the low order bits. But, the position of the most significant bit might not change at all as result of the INCREMENT, meaning that our large actual cost is not offset by a drop in potential.

**Problem 10. Set Balancing** [7 points] (3 parts) Consider an  $n \times n$  matrix  $M$ , with entries in  $\{-1, +1\}$ . We want to find a column vector  $v$  with entries in  $\{-1, +1\}$  that minimizes the maximum entry of the product  $Mv$ .

We claim that the following simple randomized algorithm gives a vector  $v$ , such that the maximum entry of  $Mv$  is bounded by  $c \cdot \sqrt{n \log n}$  for some constant  $c > 0$  with high probability: assign  $-1$  or  $+1$  to each entry of  $v$  independently at random with equal probability  $1/2$ . We will prove this in three steps.

- (a) Let  $X_1, \dots, X_n$  be a collection of independent random variables taking values in  $\{-1, +1\}$  with  $\Pr[X_i = +1] = \Pr[X_i = -1] = 1/2$ . Prove that, for any  $\delta > 0$ ,  $\Pr[\sum_{i=1}^n X_i \geq \delta] \leq e^{-\delta^2/(6n)}$ . In your proof, you may find it helpful to apply the standard Chernoff bound, which states that for  $Y_1, \dots, Y_n$  a collection of independent random variables taking values in  $\{0, 1\}$  and for  $\beta \in (0, 1)$ , we have

$$\Pr \left[ \sum_{i=1}^n Y_i \geq (1 + \beta)\mathbb{E} \left[ \sum_{i=1}^n Y_i \right] \right] \leq e^{-\beta^2 \mathbb{E}[\sum_i Y_i]/3}.$$

**Solution:** [3 points] If  $\delta > n$ , the inequality is trivial, so we need only consider  $\delta$  between 0 and  $n$  exclusive.

Let  $Y_i = (1 + X_i)/2$ . The  $Y_i$  are independent, binary-valued random variables with  $\Pr[Y_i = 0] = \Pr[Y_i = 1] = 1/2$  (so  $\mathbb{E}[\sum Y_i] = n/2$ ). We can express  $X_i$  in terms of  $Y_i$ :  $X_i = 2Y_i - 1$ . Substituting, the inequality that seek to prove is

$$\Pr \left[ \sum_i (2Y_i - 1) \geq \delta \right] \leq e^{-\delta^2/6n}.$$

Rearranging, we need to prove

$$\Pr \left[ \sum_i Y_i \geq \frac{\delta}{2} + \frac{n}{2} \right] \leq e^{-\delta^2/6n}.$$

The upper-tail Chernoff bound for the  $Y_i$  states that, for any  $\beta$  between 0 and 1 exclusive, we have

$$\Pr \left[ \sum_i Y_i \geq (1 + \beta)\mathbb{E} \left[ \sum_i Y_i \right] \right] \leq e^{-\beta^2 \mathbb{E}[\sum_i Y_i]/3n}.$$

Substituting  $\beta = \delta/n$  and  $\mathbb{E}[\sum Y_i] = n/2$ , we have

$$\Pr \left[ \sum_i Y_i \geq (1 + \delta/n) \cdot (n/2) \right] \leq e^{-(\delta/n)^2 \cdot n/6}.$$

Simplifying, we have

$$\Pr \left[ \sum_i Y_i \geq \frac{\delta}{2} + \frac{n}{2} \right] \leq e^{-\delta^2/6n},$$

which is what we wanted to prove.

*Remark:* In fact, we have  $\Pr [\sum_{i=0}^n X_i \geq \delta] \leq e^{-\delta^2/2n}$ .

The most common mistakes were (1) not defining the variables  $Y_i$  properly so that  $Y_i \in \{0, 1\}$  and (2) arithmetic mistakes that yield qualitatively incorrect tail bounds (e.g. bounds that do not get worse as  $\delta \rightarrow 0$ ).

- (b) Consider a single entry  $E_i$  of the product  $Mv$ . Using the previous result (even if you did not prove it), show that  $E_i < c \cdot \sqrt{n \log n}$  with probability at least  $1 - 1/n^2$ , for some constant  $c$ . **Solution:** [4 points] The entry  $E_i$  of  $Mv$  may be expressed as a

sum:

$$E_i = \sum_{j=1}^n M_{i,j} v_j.$$

Let  $X_j = M_{i,j} v_j$ . Because the  $v_j$  are independent and the  $M_{i,j}$  are effectively constant, the  $X_j$  are independent. Because  $M_{i,j} \in \{-1, +1\}$  and  $v_j \in \{-1, +1\}$ ,  $X_j \in \{-1, +1\}$ . We can now apply the previous result with  $\delta = c \cdot \sqrt{n \log n}$  and simplify:

$$\begin{aligned} \Pr \left[ \sum_{i=1}^n X_i \geq c \cdot \sqrt{n \log n} \right] &\leq e^{-(c \cdot \sqrt{n \log n})^2/6n} \\ &= e^{-c^2 n \log n / 6n} = n^{-c^2/6}. \end{aligned}$$

Taking  $c = \sqrt{2 \cdot 6}$  yields

$$\Pr \left[ \sum_{i=1}^n X_i \geq c \cdot \sqrt{n \log n} \right] \leq 1/n^2,$$

or equivalently

$$\Pr[E_i < c \cdot \sqrt{n \log n}] \geq 1 - 1/n^2,$$

as desired.

The most common mistakes were arithmetic mistakes that lead to substantially stronger tail bounds than the solution asks for, such as tail bounds of the form  $e^{-n^{\Omega(1)}}$  (these stronger bounds are false).

- (c) Conclude that the maximum entry of  $Mv$  is bounded by  $c \cdot \sqrt{n \log n}$  for some constant  $c > 0$  with probability at least  $1 - 1/n$  (again, you can use the result in (b) even if you did not prove it). **Solution:** [3 points] Let  $O_i$  be the event that  $E_i \geq c \cdot \sqrt{n \log n}$  for the constant  $c$  derived in the previous part. We have  $\Pr[O_i] < 1/n^2$ , so by the union bound

$$\Pr \left[ \bigcup_i O_i \right] < n \cdot 1/n^2 = 1/n.$$

That is, with high probability  $1 - 1/n$ , no event  $O_i$  holds:  $E_i < c \cdot \sqrt{n \log n}$  for every  $i$ . Equivalently, the maximum entry of  $Mv$  is bounded above by  $c \cdot \sqrt{n \log n} = O(\sqrt{n \log n})$  with high probability.

The most common mistake was in assuming that the events  $O_i$  are independent.

## SCRATCH PAPER

## Quiz 1 Solutions

- Do not open this quiz booklet until you are directed to do so. Read all the instructions on this page.
- When the quiz begins, write your name on every page of this quiz booklet.
- This quiz contains 4 problems, some with multiple parts. You have 80 minutes to earn 80 points.
- This quiz booklet contains 13 pages, including this one. Two extra sheets of scratch paper are attached. Please detach them before turning in your quiz at the end of the examination period.
- This quiz is closed book. You may use one handwritten A4 or  $8\frac{1}{2}'' \times 11''$  crib sheet. No calculators or programmable devices are permitted.
- 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, since the pages may be separated for grading.
- Do not waste time and paper rederiving facts that we have studied. It is sufficient to cite known results.
- 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.
- 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.
- Good luck!

Problem	Parts	Points	Grade	Grader
1	4	12		
2	1	7		
3	11	44		
4	3	17		
Total		80		

Name: \_\_\_\_\_

**MIT students:** Circle the name of your recitation instructor:

George

Jim

Kunal

Marten

Seth

**Problem 1. Asymptotic Running Times [12 points] (4 parts)**

For each algorithm listed below,

- give a recurrence that describes its worst-case running time, and
- give its worst-case running time using  $\Theta$ -notation.

You need not justify your answers.

**(a) Binary search**

**Solution:**  $T(n) = T(n/2) + \Theta(1) = \Theta(\lg n)$

**(b) Insertion sort**

**Solution:**  $T(n) = T(n - 1) + \Theta(n) = \Theta(n^2)$

(c) Strassen's algorithm

**Solution:**  $T(n) = 7T(n/2) + \Theta(n^2) = \Theta(n^{\lg 7})$

(d) Merge sort

**Solution:**  $T(n) = 2T(n/2) + \Theta(n) = \Theta(n \lg n)$

**Problem 2. Substitution Method [7 points]**

Consider the recurrence

$$\begin{aligned} T(n) &= T(n/2) + T(n/4) + n, \\ T(m) &= 1 \quad \text{for } m \leq 5. \end{aligned}$$

Use the substitution method to give a tight upper bound on the solution to the recurrence using  $\mathcal{O}$ -notation.

**Solution:** We guess  $T(n) = O(n)$ , which leads to the induction hypothesis  $T(m) \leq cm$  for all  $m < n$ . For  $c \geq 1$ , we have the base cases  $T(n) = 1 \leq cn$  for  $n \leq 5$ . The induction hypothesis yields

$$T(n) = T(n/2) + T(n/4) + n \leq cn/2 + cn/4 + n = (3c/4 + 1)n.$$

If we choose  $c = 4$ , then  $T(n) \leq (3 + 1)n = 4n = cn$ . By induction on  $n$ ,  $T(n) \leq cn$  for  $c \geq 4$  and all  $n \geq 1$ .

**Problem 3. True or False, and Justify [44 points] (11 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. The more content you provide in your justification, the higher your grade, but be brief. Your justification is worth more points than your true-or-false designation.

**T F** The solution to the recurrence  $T(n) = 3T(n/3) + O(\lg n)$  is  $T(n) = \Theta(n \lg n)$ .

**Solution: False.** Case 3 of the master theorem applies:  $f(n) = O(n^{\log_3 3}) = O(n)$  for  $f(n) = O(\lg n)$ , hence,  $T(n) = O(n)$ .

**T F** Let  $F_k$  denote the  $k$ th Fibonacci number. Then, the  $n^2$ th Fibonacci number  $F_{n^2}$  can be computed in  $O(\lg n)$  time.

**Solution: True.** The  $n^2$ th Fibonacci number can be computed in  $O(\lg n^2) = O(\lg n)$  time by using square and multiply method with matrix

$$\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}.$$

**T F** Suppose that an array contains  $n$  numbers, each of which is  $-1$ ,  $0$ , or  $1$ . Then, the array can be sorted in  $O(n)$  time in the worst case.

**Solution:** **True.** We may use counting sort. We first add 1 to each of the elements in the input array such that the precondition of counting sort is satisfied. After running counting sort, we subtract 1 from each of the elements in the sorted output array.

A solution based on partitioning is as follows. Let  $A[1 \dots n]$  be the input array. We define the invariant

- $A[1 \dots i]$  contains only  $-1$ ,
- $A[i + 1 \dots j]$  contains only  $0$ , and
- $A[h \dots n]$  contains only  $+1$ .

Initially,  $i = 0$ ,  $j = 0$ , and  $h = n + 1$ . If  $h = j + 1$ , then we are done; the array is sorted. In the loop we examine  $A[j + 1]$ . If  $A[j + 1] = -1$ , then we exchange  $A[j + 1]$  and  $A[i + 1]$  and we increase both  $i$  and  $j$  with 1 (as in partition in quicksort). If  $A[j + 1] = 0$ , then we increase  $j$  with 1. Finally, if  $A[j + 1] = +1$ , then we exchange  $A[j + 1]$  and  $A[h - 1]$  and we decrease  $h$  by 1.

**T F** An adversary can provide randomized quicksort with an input array of length  $n$  that forces the algorithm to run in  $\omega(n \lg n)$  time on that input.

**Solution:** **False.** As we saw in lecture, for *any* input, the expected running time of quicksort is  $O(n \lg n)$ , where the expectation is taken over the random choices made by quicksort, independent of the choice of the input.

**T F** The array

20 15 18 7 9 5 12 3 6 2

forms a max-heap.

**Solution:** **True.**

- $A[1] = 20$  has children  $A[2] = 15 \leq 20$  and  $A[3] = 18 \leq 20$ .
- $A[2] = 15$  has children  $A[4] = 7 \leq 15$  and  $A[5] = 9 \leq 15$ .
- $A[3] = 18$  has children  $A[6] = 5 \leq 18$  and  $A[7] = 12 \leq 18$ .
- $A[4] = 7$  has children  $A[8] = 3 \leq 7$  and  $A[9] = 6 \leq 7$ .
- $A[5] = 9$  has child  $A[10] = 2$ .
- $A[6], \dots, A[10]$  have no children.

**T F** Heapsort can be used as the auxiliary sorting routine in radix sort, because it operates in place.

**Solution:** **False.** The auxiliary sorting routine in radix sort needs to be stable, meaning that numbers with the same value appear in the output array in the same order as they do appear in the input array. Heapsort is not stable. It does operate in place, meaning that only a constant number of elements of the input array are ever stored outside the array.

- T F** There exists a comparison sort of 5 numbers that uses at most 6 comparisons in the worst case.

**Solution:** **False.** The number of leaves of a decision tree which sorts 5 numbers is  $5!$  and the height of the tree is at least  $\lg(5!)$ . Since  $5! = 120$ ,  $2^6 = 64$ , and  $2^7 = 128$ , we have  $6 < \lg(5!) < 7$ . Thus at least 7 comparisons are required.

- T F** Suppose that a hash table with collisions resolved by chaining contains  $n$  items and has a load factor of  $\alpha = 1/\lg n$ . Assuming simple uniform hashing, the expected time to search for an item in the table is  $O(1/\lg n)$ .

**Solution:** **False.** The expected time to search for an item in the table is  $O(1 + \alpha) = O(1 + 1/\lg n) = O(1)$ . At least a constant running time  $O(1)$  is needed to search for an item; subconstant running time  $O(1/\lg n)$  is not possible.

**T F** Let  $X$  be an indicator random variable such that  $E[X] = 1/2$ . Then, we have  $E[\sqrt{X}] = 1/\sqrt{2}$ .

**Solution:** **False.** Since  $X$  is an indicator random variable,  $X = 0$  or  $X = 1$ . For both possible values  $\sqrt{X} = X$ , which implies that  $E[\sqrt{X}] = E[X] = 1/2$ .

**T F** Suppose that a hash table of  $m$  slots contains a single element with key  $k$  and the rest of the slots are empty. Suppose further that we search  $r$  times in the table for various other keys not equal to  $k$ . Assuming simple uniform hashing, the probability is  $r/m$  that one of the  $r$  searches probes the slot containing the single element stored in the table.

**Solution:** **False.** The probability  $p$  that one of the  $r$  searches collides with the single element stored in the table is equal to 1 minus the probability that none of the  $r$  searches collides with the single element stored in the table. That is,  $p = 1 - (1 - 1/m)^r$ .

**T F** Let  $S$  be a set of  $n$  integers. One can create a data structure for  $S$  so that determining whether an integer  $x$  belongs to  $S$  can be performed in  $O(1)$  time in the worst case.

**Solution:** **True.** Perfect hashing.

**Problem 4. Close Numbers** [17 points] (3 parts)

Consider a set  $S$  of  $n \geq 2$  distinct numbers. For simplicity, assume that  $n = 2^k + 1$  for some  $k \geq 0$ . Call a pair of distinct numbers  $x, y \in S$  **close** in  $S$  if

$$|x - y| \leq \frac{1}{n-1} \left( \max_{z \in S} z - \min_{z \in S} z \right),$$

that is, if the distance between  $x$  and  $y$  is at most the average distance between consecutive numbers in the sorted order.

- (a) Explain briefly why every set  $S$  of  $n \geq 2$  distinct numbers contains a close pair of numbers.

**Solution:** Without loss of generality, assume  $S = \{z_1, z_2, \dots, z_n\}$ , with  $z_i \leq z_{i+1}$ . The average distance between two consecutive numbers  $z_i$  and  $z_{i+1}$  is

$$\frac{1}{n-1} \sum_{i=1}^{n-1} (z_{i+1} - z_i) = \frac{1}{n-1} (z_n - z_1).$$

There exists at least one pair of consecutive numbers  $x$  and  $y$  whose distance between them is less than or equal to the average. The result then follows from the definition of the *close* pair.

- (b) Suppose that we partition  $S$  around a pivot element  $p \in S$ , organizing the result into two subsets of  $S$ :  $S_1 = \{x \in S \mid x \leq p\}$  and  $S_2 = \{x \in S \mid x \geq p\}$ . Prove that either
1. every pair  $x, y \in S_1$  of numbers that is close in  $S_1$  is also close in  $S$ , or
  2. every pair  $x, y \in S_2$  of numbers that is close in  $S_2$  is also close in  $S$ .

Show how to determine, in  $O(n)$  time, a value  $k \in \{1, 2\}$  such that every pair  $x, y \in S_k$  of numbers that is close in  $S_k$  is also close in  $S$ .

**Solution:** Without loss of generality, assume that the elements in  $S_i$  are in sorted order. For  $k = 1, 2$ , let  $a_k$  be the average distance between two consecutive numbers in  $S_k$ , and let  $n_k$  the number of elements in  $S_k$ . Using the result from Part (a), we have

$$a_1 = \frac{1}{n_1 - 1} \left( \max_{z \in S_1} z - \min_{z \in S_1} z \right) = \frac{1}{n_1 - 1} \left( p - \min_{z \in S} z \right),$$

and

$$a_2 = \frac{1}{n_2 - 1} \left( \max_{z \in S_2} z - \min_{z \in S_2} z \right) = \frac{1}{n_2 - 1} \left( \max_{z \in S} z - p \right).$$

The average distance  $a$  between two consecutive numbers in  $S$  in sorted order is then given by

$$\begin{aligned} a &= \frac{1}{n - 1} \left( \max_{z \in S} z - \min_{z \in S} z \right) \\ &= \frac{1}{n - 1} \left( p - \min_{z \in S} z \right) + \frac{1}{n - 1} \left( \max_{z \in S} z - p \right) \\ &= \frac{n_1 - 1}{n - 1} a_1 + \frac{n_2 - 1}{n - 1} a_2. \end{aligned}$$

Note that  $n_1 + n_2 = n + 1$ , because  $p$  is included in both  $S_1$  and  $S_2$ . So,  $a$  is a weighted average of  $a_1$  and  $a_2$ :

$$a = (1 - \alpha)a_1 + \alpha a_2,$$

where  $\alpha = (n_2 - 1)/(n - 1)$ .

Suppose that  $a_1 \leq a_2$ . If  $x$  and  $y$  are a close pair in  $S_1$ , then

$$|x - y| \leq a_1 = (1 - \alpha)a_1 + \alpha a_1 \leq (1 - \alpha)a_1 + \alpha a_2 = a.$$

This implies that every close pair in  $S_1$  is also a close pair in  $S$ . Similarly, if  $a_2 \leq a_1$ , then every close pair in  $S_2$  is a close pair in  $S$ .

The average distance  $a_k$  can be computed in  $O(n)$  time, by searching for the minimum or the maximum number in  $S_k$ . Therefore, the subset  $S_k$  with the specified property can be computed in  $O(n)$  time.

- (c) Describe an  $O(n)$ -time algorithm to find a close pair of numbers in  $S$ . Explain briefly why your algorithm is correct, and analyze its running time. (*Hint:* Use divide and conquer.)

**Solution:** The idea is to partition  $S$  recursively until we find a close pair.

1. Determine the median of  $S$  and use it to partition  $S$  into  $S_1$  and  $S_2$ .
2. Use the result from Part (b) to determine the set  $S_k$  that contains a close pair of  $S$ .
3. Recurse on  $S_k$  until  $S_k$  contains 2 elements.

Since each recursive step reduces the cardinality of the set by roughly a half, the recursion is guaranteed to terminate. After each recursive step, the remaining set contains a close pair of  $S$ .

Step 1 takes  $O(n)$  time in the worst case, if we use the deterministic median-finding algorithm. Step 2 takes  $O(n)$  time based on the result from Part (b). Therefore, the running time of the algorithm is given by the following recurrence:

$$T(n) = T(n/2) + O(n),$$

with the solution  $T(n) = O(n)$  according to the master theorem.

**SCRATCH PAPER** — Please detach this page before handing in your exam.

**SCRATCH PAPER** — Please detach this page before handing in your exam.

## Quiz 1

- Do not open this quiz booklet until you are directed to do so. Read all the instructions first.
- **For this quiz, you *need not* provide rigorous proofs of correctness. Instead, give informal arguments for why you believe your algorithms are correct. Pseudocode is only required when explicitly indicated, but you may include it if it clarifies your answers.**
- When the quiz begins, write your name on every page of this quiz booklet.
- The quiz contains 5 multi-part problems. You have 80 minutes to earn 80 points.
- This quiz booklet contains 11 pages, including this one. Two extra sheets of scratch paper are attached. Please detach them before turning in your quiz.
- This quiz is closed book. You may use one handwritten A4 or  $8\frac{1}{2}'' \times 11''$  crib sheet. No calculators or programmable devices are permitted.
- 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, since the pages may be separated for grading.
- Do not waste time and paper rederiving facts that we have studied. It is sufficient to cite known results.
- 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.
- 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.
- Good luck!

Problem	Points	Grade	Initials
1	12		
2	12		
3	30		
4	13		
5	13		
Total	80		

Name: Solutions \_\_\_\_\_

Circle the name of your recitation instructor:

Moses

Jen

Steve

**Problem 1. Recurrences [12 points]**

Solve the following recurrences. Give tight, i.e.  $\Theta(\cdot)$ , bounds.

(a)  $T_1(n) = 5 T_1(n/2) + \sqrt{n}$

**Solution:** We use the Master Theorem: note that  $\log_2 5 > 1/2 + \epsilon$ , so the solution is  $T_1(n) = \Theta(n^{\log_2 5})$ .

(b)  $T_2(n) = 64 T_2(n/4) + 8^{\lg n}$

**Solution:** First, notice that  $8^{\lg n} = n^{\lg 8}$  (this can be seen by taking  $\lg$  of both sides). Since  $\log_4 64 = \lg 8 = 3$ , we use case (ii) of the Master Theorem:  $T_2(n) = \Theta(n^3 \log n)$ .

(c)  $T_3(n) = 2 T_3(3n/8) + T_3(n/4) + 6n$

**Solution:** We solve this recurrence by Akra-Bazzi, with  $p = 1$ . Then we get

$$T_3(n) = \Theta\left(n \left(1 + \int_1^n \frac{6x}{x^2} dx\right)\right) = \Theta(n(1 + 6 \ln n)) = \Theta(n \log n).$$

**Problem 2. Short Answer [12 points]**

Give *brief*, but complete, answers to the following questions.

- (a) Briefly describe the difference between a *deterministic* and a *randomized* algorithm, and name two examples of algorithms that are not deterministic.

**Solution:** On identical inputs, a deterministic algorithm always performs exactly the same computations and returns the same output. A randomized algorithm is one which “flips coins,” i.e. one which makes random choices that may cause it to perform different computations, even on the same input. RANDOMIZED-QUICKSORT and RANDOMIZED-SELECT are two examples of non-deterministic algorithms.

- (b) Describe the difference between *average-case* and *worst-case* analysis of deterministic algorithms, and give an example of a deterministic algorithm whose average-case running time is different from its worst-case running time.

**Solution:** An average-case analysis assumes some distribution over the inputs (e.g., uniform), and computes the expected (average) running time of an algorithm subject to that distribution. A worst-case analysis considers those inputs which force an algorithm to run for the longest amount of time, and computes the running time under those inputs. QUICKSORT has a worst-case running time of  $\Theta(n^2)$  (on an already-sorted or reverse-sorted array), but has an average-case running time of  $\Theta(n \log n)$  (assuming all input permutations are equally likely).

- (c) If you can multiply 4-by-4 matrices using 48 scalar multiplications, can you multiply  $n \times n$  matrices asymptotically faster than Strassen’s algorithm (which runs in  $O(n^{\lg 7})$  time)? Explain your answer.

**Solution:** Our algorithm breaks an  $n \times n$  matrix into a 4-block by 4-block matrix (where each block is  $n/4 \times n/4$ ). It then multiplies the appropriate blocks using 48 recursive calls (corresponding to the scalar multiplications) and combines their products. Our new algorithm’s running time is  $T(n) = 48T(n/4) + \Theta(n^2)$ , which is  $O(n^{\log_4 48})$  by the Master Theorem. For comparison with Strassen’s algorithm, note that  $\log_2 7 = \log_{2^2} 7^2 = \log_4 49$ . Therefore our algorithm is asymptotically better.

**Problem 3. True or False, and Justify [30 points]**

Circle **T** or **F** for each of the following statements, and briefly explain why. The better your argument, the higher your grade, but be brief. Your justification is worth more points than your true-or-false designation.

- (a) **T F** Every comparison-based sort uses at most  $O(n \log n)$  comparisons in the worst case.

**Solution:** False. INSERTION-SORT, for example, uses  $\Theta(n^2) \neq O(n \log n)$  comparisons in the worst-case (a reverse-sorted array). The statement would be true if it read "...at least  $\Omega(n \log n)$  comparisons in the worst case."

- (b) **T F** RADIX-SORT is stable if its auxiliary sorting routine is stable.

**Solution:** True. If two numbers are equal, then they have the same digits. Each intermediate sort is stable, so the two equal numbers never change relative positions.

- (c) **T F** It is possible to compute the smallest  $\sqrt{n}$  elements of an  $n$ -element array, in sorted order, in  $O(n)$  time.

**Solution:** True. We can SELECT the  $\sqrt{n}$ th smallest element and partition around it, then sort those  $\sqrt{n}$  elements in  $O(n)$  time. Alternately, we can build a min-heap in  $O(n)$  time and call EXTRACT-MIN  $\sqrt{n}$  times, for a total runtime of  $O(n + \sqrt{n} \log n) = O(n)$ .

Some incorrect solutions amounted to: "we must do  $\sqrt{n}$  order statistic queries, each of which take  $O(n)$  time, for a total running time of  $O(n\sqrt{n})$ ." However, this argument does not preclude us from coming up with a more clever algorithm (like the one above) that is more efficient. In fact, a similar argument would "prove" that sorting must take  $\Omega(n^2)$  time (despite the existence of MERGESORT etc.), because we must do  $n$  order statistic queries!

- (d) **T F** Consider hashing the universe  $U = \{0, \dots, 2^r - 1\}$ ,  $r > 2$ , into the hash table  $\{0, 1\}$ . Consider the family of hash functions  $\mathcal{H} = \{h_1, \dots, h_r\}$ , where  $h_i(x)$  is the  $i$ th bit of the binary representation of  $x$ . Then  $\mathcal{H}$  is universal.

**Solution:** False. Take  $x = 0, y = 1$ . Then all of  $x$ 's binary digits are the same as  $y$ 's, except for the least significant one. Thus  $\Pr_{h \in \mathcal{H}}[h(x) = h(y)] = (r-1)/r > 1/2$ . If  $\mathcal{H}$  were universal, the probability would be at most  $1/2$ .

Some incorrect solutions said that the probability of a collision, taken over random choices of  $h$ ,  $x$ , and  $y$  is  $1/2$ . This is true, but the universality condition demands something stronger: it says that for *every* fixed (distinct) pair  $x, y$ , their probability of collision over the random choice of  $h$  must be at most  $1/2$ .

- (e) **T F** RANDOMIZED-SELECT can be forced to run in  $\Omega(n \log n)$  time by choosing a bad input array.

**Solution:** False. RANDOMIZED-SELECT runs in expected  $O(n)$  time; the only way it can take longer is if its random choices of pivots are unlucky. The input array cannot force these unlucky choices.

- (f) **T F** For every two functions  $f(n)$  and  $g(n)$ , either  $f(n) = O(g(n))$  or  $g(n) = O(f(n))$ .

**Solution:** False. Let  $f(n) = \sin n$  and  $g(n) = \cos n$ ; then neither case holds. Another example is  $f(n) = \sqrt{n}$  and  $g(n) = n^{\sin n}$ . Finally, one could let  $f(n)$  and  $g(n)$  be any strictly-negative functions; by a technical condition of the definition,  $f(n)$  must be at least 0 to be  $O(g(n))$ .

Many incorrect answers argued that one of the statements  $f(n) \leq cg(n)$ ,  $f(n) = cg(n)$ , or  $g(n) \leq cf(n)$  must be true. This is correct for any *particular* value of  $n$ , but it doesn't mean that the *same* statement is true for *all* sufficiently large values of  $n$ , which is the condition needed in the definition of big- $O$ .

- (g) **T F** Suppose that we have a hash table with  $2n$  slots, with collisions resolved by chaining, and suppose that  $n/2$  keys are inserted into the table. Each key is equally likely to be hashed into each slot (*simple uniform hashing*). Then the expected number of keys for each slot is  $1/4$ .

**Solution:** True. Define  $X_j$  (for  $j = 1, \dots, n/2$ ) to be the indicator which is 1 if element  $j$  hashes to slot  $i$ , and 0 otherwise. Then  $E[X_j] = \Pr[X_j = 1] = 1/2n$ . Then the expected number of elements in slot  $i$  is  $E[\sum_{j=1}^{n/2} X_j] = \sum_{j=1}^{n/2} E[X_j] = n/4n = 1/4$  by linearity of expectation.

- (h) **T F** The following array  $A$  is a max-heap:

30 25 7 18 24 8 4 9 12 22 5

**Solution:** False. See that  $7 = A[3] < A[2 \cdot 3] = 8$ , which is a violation of the max-heap property.

- (i) **T F** Suppose we use HEAPSORT instead of INSERTION-SORT as a subroutine of BUCKET-SORT to sort  $n$  elements. Then BUCKET-SORT still runs in average-case linear time, but its worst-case running time is now  $O(n \log n)$ .

**Solution:** True. Even if all the elements land in the same bucket (the worst-case input), HEAPSORT sorts them in  $O(n \log n)$  time.

- (j) **T F** If memory is limited, one would prefer to sort using HEAPSORT instead of MERGESORT.

**Solution:** True. MERGESORT is not in-place, which means it requires an auxiliary array as big as the input. HEAPSORT is in-place, which means it only uses  $O(1)$  auxiliary space.

**Problem 4. Perfect Powers [13 points]**

You have just discovered a breakthrough result in number theory that will quickly become world-famous, but there is one step left for you to complete. You need to find a fast algorithm to tell whether a number  $X$  is a perfect power. That is, you want a fast algorithm which, on an input integer  $X$  that is  $n$  bits long, finds whether there exist integers  $B \geq 2$  and  $e \geq 2$  such that  $X = B^e$ . If so, your algorithm should output the values of  $B$  and  $e$ .

- (a) If  $X = B^e$  is a perfect power, how large can  $e$  be? Express your answer as a function of  $n$ .

**Solution:** (First, some trivia: the scenario from this problem is entirely real! The ground-breaking deterministic primality-testing algorithm by Agrawal et al, discovered with great fanfare this past summer, performs the perfect-power test as one of its first steps.)

[3 points] Note that  $n \geq \lg X = e \lg B$ , and since  $B \geq 2$ ,  $e \leq n$ .

A common error was to write only  $e \leq \frac{n}{\lg B}$ , which is insufficient because we asked for  $e$  as a function of  $n$ , and also because the value of  $B$  (if any) is not known in advance.

- (b) Suppose that your computer had an  $O(1)$ -time operation  $\text{ROOT}(M, r)$  that returns the  $r$ th root of an integer  $M$ , if that root is an integer (and returns  $\perp$  otherwise). Give an algorithm to solve the perfect-power problem and analyze its running time.

**Solution:** [5 points] We simply test all values of  $e$  up to  $n$ : for  $e = 1, \dots, n$ , if  $\text{ROOT}(X, e) = B$  where  $B$  is an integer, then return the pair  $(B, e)$ . If no such root is an integer, return  $\perp$ . The running time of this procedure is  $O(n)$ .

Some solutions looped over all possible values of both  $B$  and  $e$ , and only returned if  $B = \text{ROOT}(X, e)$ . This solution is wasteful, because  $\text{ROOT}$  returns the proper base, so there is no need to guess it in advance. This solution is also too inefficient, because  $B$  could be as large as  $\sqrt{X} \approx 2^{n/2}$ , so an exponential number of iterations would be performed.

- (c) In reality, there is no such  $O(1)$ -time ROOT procedure. Still it is possible to solve the perfect power problem in  $O(n^2 \log n)$  by using a suitable algorithm for ROOT. Describe such an algorithm and analyze its running time. You may assume that multiplying two integers takes  $O(1)$  time (no matter how large they are).

**Solution:** [5 points] We will implement a ROOT procedure that runs in time  $O(n \log n)$  when used in the above algorithm, so that we solve the perfect power problem in  $O(n^2 \log n)$  time. On input  $(M, r)$ , our ROOT procedure performs a binary search for the  $r$ th root of  $M$  in the range  $[2 \dots M]$ . When testing the midpoint  $m$  of the range, if  $m^r < M$  then we recursively search the upper half of the range; if  $m^r > M$  then we search the lower half; if  $m^r = M$  then we return  $m$  as the  $r$ th root of  $M$ .

In our usage of ROOT from the previous part,  $M = X \leq 2^n$  and  $r \leq n$ . Computing each  $m^r$  takes  $O(\log r) = O(\log n)$  multiplications by the repeated-squaring technique from class, and the binary search does  $O(\log M) = O(n)$  iterations, each with one exponentiation, for the claimed runtime of  $O(n \log n)$ .

Many solutions did a brute-force search for  $B$ , but this is too inefficient. In the worst case,  $r = 2$ , so all values of  $B$  from 1 to  $\sqrt{M} = \sqrt{X} = 2^{n/2}$  would have to be checked, which is an exponential number of tests. All exponential-time algorithms received no points for this part.

Other incorrect solutions made mathematical errors: assuming that the  $r$ th root of  $M$  must be smaller than  $\log_r M$ , or returning  $\log_r M$  by repeatedly dividing  $M$  by  $r$  or multiplying  $r$  by itself (this treats  $r$  as the base, instead of its proper role as the exponent).

**Problem 5. Assigning Grades [13 points]**

It is the not-too-distant-future, and you are a computer science professor at a prestigious north-eastern technical institute. After teaching your course, “6.66: Algorithms from Hell,” you have to assign a letter grade to each student based on his or her unique total score. (Scores can only be compared to each other.) You are grading on a curve, and there are a total of  $k$  different grades possible. You want to rearrange the students into  $k$  equal-sized groups, such that everybody in the top group has a higher score than everybody in the second group, etc. However, you don’t care how the students are ordered within each group (because they will all receive the same grade).

- (a) Describe and analyze a simple algorithm that takes an unsorted  $n$ -element array  $A$  of scores and an integer  $k$ , and divides  $A$  into  $k$  equal-sized groups, as described above. Your algorithm should run in time  $O(nk)$ . (If you find a faster algorithm, see part (c).) You may assume that  $n$  is divisible by  $k$ . *Note:*  $k$  is an input to the algorithm, not a fixed constant.

**Solution:** [5 points] Our algorithm first uses SELECT to find the  $n/k$ th order statistic, then partitions around it. At this point, the first  $n/k$  elements of the array form the bottom group. Then it uses SELECT to find the  $n/2k$ th order statistic of the remainder of the array, and partitions around it, etc., until all the groups have been separated. Each SELECT and PARTITION requires linear time in the number of remaining elements, which is at most  $n$ , so the running time is  $O(nk)$ .

- (b) In the case that  $k = n$ , prove that any algorithm to solve this problem must run in time  $\Omega(n \log k)$  in the worst case. Recall that we are only considering comparison-based algorithms, i.e., algorithms that only compare scores to each other as a way of finding information about the input. *Hint:* There is a very short proof.

**Solution:** [3 points] Any algorithm for this problem can fully sort an array of  $n$  elements if we provide it with an input where  $k = n$ . Since sorting requires  $\Omega(n \lg n)$  comparisons in the worst case, the algorithm must run in time  $\Omega(n \lg n) = \Omega(n \lg k)$  in the worst case.

- (c) Now describe and analyze an algorithm for this problem that runs in time  $O(n \log k)$ . You may also assume that  $k$  is a power of 2, in addition to assuming that  $n$  is divisible by  $k$ .

**Solution:** [5 points] We use a recursive algorithm GROUP, which takes an array and a value  $k$ , and works as follows: if  $k = 1$ , return. Otherwise, SELECT and PARTITION around the median of the array. Then call GROUP on the lower half of the array with  $k/2$ , and again on the top half with  $k/2$ .

To see that this works, note that after partitioning, all grades in the upper half of the array are greater than those in the lower half. By induction, the two recursive calls divide each half into  $k/2$  groups, for a total of  $k$  groups. Finally, note that the base case satisfies the problem statement.

We now analyze the running time: the recurrence describing the algorithm's running time is  $T(n, k) = 2T(n/2, k/2) + \Theta(n)$  because SELECT and PARTITION are linear-time. The base case of the recurrence is  $T(n, 1) = \Theta(1)$  for any  $n$ . Therefore the recurrence tree does  $\Theta(n)$  work at each level, and has  $\lg k$  levels, for a total running time of  $\Theta(n \log k)$ .

Some students correctly observed that this solution is essentially an “early quitting” QUICKSORT, where the pivot is always chosen to be the median, and the algorithm terminates once the recursion depth reaches  $\lg k$ .

**SCRATCH PAPER** — Please detach this page before handing in your quiz.

**SCRATCH PAPER** — Please detach this page before handing in your quiz.

## Quiz 1 Solutions

- Do not open this quiz booklet until you are directed to do so. Read all the instructions first.
- When the quiz begins, write your name on every page of this quiz booklet.
- The quiz contains five multi-part problems. You have 80 minutes to earn 80 points.
- This quiz booklet contains 13 pages, including this one. An extra sheet of scratch paper is attached. Please detach it before turning in your quiz.
- This quiz is closed book. You may use one handwritten A4 or  $8\frac{1}{2}'' \times 11''$  crib sheet. No calculators or programmable devices are permitted.
- 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, since the pages may be separated for grading.
- Do not waste time and paper rederiving facts that we have studied. It is sufficient to cite known results.
- 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.
- 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.
- Good luck!

Problem	Points	Grade	Initials
1	11		
2	19		
3	10		
4	20		
5	20		
Total	80		

Name: Solutions \_\_\_\_\_  
Circle your recitation letter and the name of your recitation instructor:

David      A      B      Steve      C      D      Hanson      E      F

**Problem -1. Recurrences [11 points]**

Solve the following recurrences. Give tight, i.e.  $\Theta(\cdot)$ , bounds.

(a)  $T(n) = 81T\left(\frac{n}{9}\right) + n^4 \lg n$  [2 points]

**Solution:**  $T(n) = \Theta(n^4 \lg n)$  by Case 3 of the Master Method. We have:  $n^{\log_b a} = n^2$  and  $f(n) = n^4 \lg n$ . Thus, if  $\epsilon = 1$ , then  $f(n) = \Omega(n^{2+\epsilon})$ . The regularity condition,  $a \cdot f(n/b) < c \cdot f(n)$  holds here for  $c = .99$ .

You received one point if you mentioned Case 3 of the Master Method and showed that it complied with the regularity condition and one point if you gave the correct answer.

(b)  $T(n) = 2T\left(\frac{n}{2}\right) + \frac{n}{\lg n}$  [2 points]

**Solution:** Note that we can not use the Master Method to solve this recurrence. If we expand this recurrence, we obtain:

$$\begin{aligned} T(n) &= \frac{n}{\lg n} + \frac{n}{\lg \frac{n}{2}} + \frac{n}{\lg \frac{n}{4}} \cdots \frac{n}{\lg \frac{n}{n}} = n \cdot \sum_{i=1}^{\lg n} \frac{1}{\lg \frac{n}{2^i}} = \\ &n \cdot \sum_{i=1}^{\lg n} \frac{1}{\lg n - i} = n \cdot \sum_{i=1}^{\lg n} \frac{1}{i} = \Theta(n \lg \lg n). \end{aligned}$$

- (c)  $T(n) = 4T(n/3) + n^{\log_3 4}$  [2 points]

**Solution:**  $T(n) = \Theta(n^{\log_3 4} \lg n)$  by Case 2 of the Master Method.

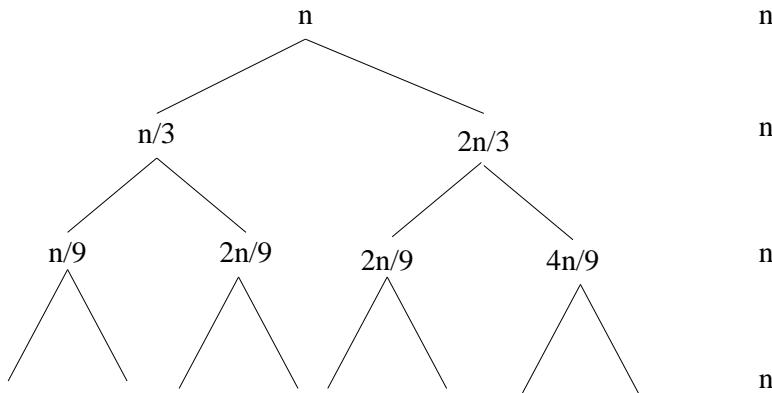
You received one point if you mentioned the correct case of the Master Method and one point for the correct answer. A common error was to not cite which case of the Master Method you used.

- (d) Write down and solve the recurrence for the running time of the DETERMINISTIC SELECT algorithm using groups of 3 (rather than 5) elements. The code is provided on the last page of the exam. [5 points]

**Solution:**  $T(n) = T(n/3) + T(2n/3) + cn$ . We solve this recurrence using a recursion tree, see Figure 1. The height of the tree is at least  $\log_3 n$  and is at most  $\log_{3/2} n$  and the sum of the costs in each level is  $n$ . Hence  $T(n) = \Theta(n \lg n)$ .

A correct recurrence received 3 points. One point was awarded for solving whichever recurrence you gave correctly and one point was awarded for justification. An incorrect recurrence received 1 point if it was almost correct.

A common error was omitting the  $T(2n/3)$  term in the recurrence.



**Figure 1:** Recursion tree

**Problem -2. True or False, and Justify [19 points]**

Circle **T** or **F** for each of the following statements, and briefly explain why. The better your argument, the higher your grade, but be brief. Your justification is worth more points than your true-or-false designation.

- (a) **T F** Given a set of  $n$  integers, the number of comparisons necessary to find the maximum element is  $n - 1$  and the number of comparisons necessary to find the minimum element is  $n - 1$ . Therefore the number of comparisons necessary to simultaneously find the smallest and largest elements is  $2n - 2$ . **[3 points]**

**Solution:** False. In recitation, we gave an algorithm that uses  $3n/2$  comparisons to simultaneously find the maximum and minimum elements.

A common error was to say, "You can just keep track of min and max at the same time", which does not say how many comparisons that would take or how it would work.

- (b) **T F** There are  $n$  people born between the year 0 A.D. and the year 2003 A.D. It is possible to sort all of them by birthdate in  $o(n \lg n)$  time.

**Solution:** True. We can represent a birthdate using  $c$  digits. This database can then be sorted in  $O(c \cdot n) = O(n)$  time using RADIX SORT.

Common errors included saying, "Use Mergesort" or "Use Radix Sort" (with no explanation).

- (c) **T F** There is some input for which RANDOMIZED QUICKSORT always runs in  $\Theta(n^2)$  time. [3 points]

**Solution:** False. The *expected* running time of RANDOMIZED QUICKSORT is  $\Theta(n \lg n)$ . This applies to *any* input.

A common error was to say, "RANDOMIZED QUICKSORT will take  $O(n^2)$  only if it's very unlucky", but you needed to say what it's expected runtime is.

- (d) **T F** The following array  $A$  is a Min Heap:

2    5    9    8    10    13    12    22    50

[3 points]

**Solution:** True.

- (e) **T F** If  $f(n) = \Omega(g(n))$  and  $g(n) = O(f(n))$  then  $f(n) = \Theta(g(n))$ . [3 points]

**Solution:** False. For example  $f(n) = n^2$  and  $g(n) = n$ .

A common error was to mix up  $f$  and  $g$  or  $O$  and  $\Omega$ .

- (f) **T F** Let  $k, i, j$  be integers, where  $k > 3$  and  $1 \leq i, j \leq k$ . Let  $h_{ij}^k$  be the hash function mapping a  $k$ -bit integer  $b_1 b_2 \dots b_k$  to the 2-bit value  $b_i b_j$ . For example,  $h_{31}^8(\mathbf{00101011}) = 10$ .

The set  $\{h_{ij}^k : 1 \leq i, j \leq k\}$  is a universal family of hash functions from  $k$ -bit integers into  $\{00, 01, 10, 11\}$ . [4 points]

**Solution:** False. Take  $x = 0, y = 1$ . Then all of  $x$ 's binary digits are the same as  $y$ 's, except for the least significant bit. Thus, if we choose one of the  $k$  hash functions at random,  $\Pr[h(x) = h(y)] = (k^2 - k)/k^2 = (k - 1)/k > 1/4$ . If this were a universal class of hash functions, then this probability should be at most  $1/4$ .

A common error was to show a single counter-example without explanation.

**Problem -3. Short Answer [10 points]**

Give *brief*, but complete, answers to the following questions.

- (a) Explain the differences between average-case running time analysis and expected running time analysis. For each type of running time analysis, name an algorithm we studied to which that analysis was applied.

**[5 points]**

**Solution:** The average-case running time does not provide a guarantee for the worst-case, i.e. it only applies to a specific input distribution, while the expected running time provides a guarantee (in expectation) for every input.

In the average-case running time, the probability is taken over the random choices over an input distribution. In the expected running time, the probability is taken over the random choices made by the algorithm.

The average-case running time of BUCKET SORT is  $\Theta(n)$  when the input is chosen at random from the uniform distribution. The expected running time of QUICK SORT is  $\Theta(n \lg n)$ .

- (b) Suppose we have a hash table with  $2n$  slots with collisions resolved by chaining, and suppose that  $n/8$  keys are inserted into the table. Assume each key is equally likely to be hashed into each slot (**simple uniform hashing**). What is the expected number of keys for each slot? Justify your answer. [5 points]

**Solution:** Define  $X_j$  (for  $j = 1, \dots, n$ ) to be the indicator which is 1 if element  $j$  hashes to slot  $i$  and 0 otherwise. Then  $E[X_j] = Pr[X_j = 1] = 1/(2n)$ . Then the expected number of elements in slot  $i$  is  $E[\sum_{j=1}^{n/8} X_j] = \sum_{j=1}^{n/8} E[X_j] = n/(8(2n)) = 1/16$  by linearity of expectation.

**Problem -4. Checking Properties of Sets [20 points]**

In this problem, more efficient algorithms will be given more credit. Let  $S$  be a finite set of  $n$  positive integers,  $S \subset \mathbb{Z}^+$ . You may assume that all basic arithmetic operations, i.e. addition, multiplication, and comparisons, can be done in unit time. In this problem, partial credit will be given for correct but inefficient algorithms.

- (a) Design an  $O(n \lg n)$  algorithm to verify that:

$$\forall T \subseteq S, \quad \sum_{t \in T} t \geq |T|^3.$$

In other words, if there is some subset  $T \subseteq S$  such that the sum of the elements in  $T$  is less than  $|T|^3$ , the your algorithm should output “no”. Otherwise, it should output “yes”. Argue (informally) that your algorithm is correct and analyze its running time. **[10 points]**

**Solution:** Sort the set  $S$  in time  $O(n \lg n)$  using  $O(n \lg n)$  sorting algorithm such as MERGE SORT or HEAP SORT. For each  $k$  between 1 and  $n$ , verify that  $(\sum_{i=0}^k s_i) \geq k^3$ . By maintaining a running sum, checking this sum for each value of  $k$  requires only one addition and one comparison operation. Thus, the total running time is  $O(n \lg n) + O(n) = O(n \lg n)$ .

Correctness: The key observation is that *every* set of  $k$  elements is at least  $k^3$  iff the sum of the smallest  $k$  elements is at least  $k^3$ . Thus, if we sort the elements, the sum of the first  $k$  elements, namely the  $k$  smallest elements, will be at least  $k^3$  iff every set of size  $k$  has sum at least  $k^3$ .

The following procedure CHECKALLSUMS takes as input an array  $A$  containing  $S$  and an integer  $n$  indicating the size of  $S$ .

```
CHECKALLSETS( $A, n$ )
1 Sort  $A$  using MERGESORT
2 sum  $\leftarrow 0$ 
3 from  $i \leftarrow 1$  to  $n$ 
4     sum  $\leftarrow$  sum +  $A[i]$ 
5     if sum <  $k^3$ 
6         return “no”
7 return “yes”
```

A common error was to use COUNTING SORT, which would not necessarily be efficient for an arbitrary set of  $n$  integers.

Many people got this problem backwards, i.e. they tried to find *some* set such that its sum was at least its size cubed. This was not heavily penalized.

- (b)** In addition to  $S$  and  $n$ , you are given an integer  $k$ ,  $1 \leq k \leq n$ . Design a more efficient (than in part **(a)**) algorithm to verify that:

$$\forall T \subseteq S, |T| = k, \sum_{t \in T} t \geq k^3.$$

In other words, if there is some subset  $T \subseteq S$  such that  $T$  contains exactly  $k$  elements and the sum of the elements in  $T$  is less than  $k^3$ , then your algorithm should output “no”. Otherwise, it should output “yes”. Argue (informally) that your algorithm is correct and analyze its running time. **[10 points]**

**Solution:** Find the  $k$ th smallest element using the linear time DETERMINISTICSELECT algorithm. Then find the  $k$  smallest elements using the PARTITION procedure. Check that the sum of these  $k$  smallest elements is greater than  $k^3$ . The total runtime is  $O(n) + O(n) + O(k) = O(n)$ .

CHECKSETSIZESK( $A, n, k$ )

- 1 Run SELECT( $A, n, k$ ) to find  $k^{th}$  smallest element
- 2 Run PARTITION( $A, n, k$ ) to put  $k$  smallest elements in  $A[1 \dots k]$
- 3 sum  $\leftarrow 0$
- 4 from  $i \leftarrow 1$  to  $k$
- 5     sum  $\leftarrow$  sum +  $A[i]$
- 6 if sum  $< k^3$
- 7     return “no”
- 8 return “yes”

Correctness: Again, as in part **(a)**, the key observation is that *every* set of  $k$  elements has sum at least  $k^3$  iff the sum of the  $k$  smallest elements is at least  $k^3$ .

**Problem -5. Finding the Missing Number [20 points]**

Suppose you are given an unsorted array  $A$  of all integers in the range 0 to  $n$  except for one integer, denoted the *missing number*. Assume  $n = 2^k - 1$ .

- (a) Design a  $O(n)$  Divide and Conquer algorithm to find the missing number. Partial credit will be given for non Divide and Conquer algorithms. Argue (informally) that your algorithm is correct and analyze its running time. **[12 points]**

**Solution:** We can use SELECT to find the median element and check to see if it is in the array. If it is not, then it is the missing number. Otherwise, we PARTITION the array around the median element  $x$  into elements  $\leq x$  and  $> x$ . If the first one has size less than  $x + 1$ , then we recurse on this subarray. Otherwise we recurse on the other subarray.

The procedure MISSINGINTEGER( $A, n, [i, j]$ ) takes as input an array  $A$  and a range  $[i, j]$  in which the missing number lies.

```
MISSINGINTEGER( $A, [i, j]$ )
1 Determine median element  $x$  in range  $i \dots j$ 
2 Check to see if  $x$  is in  $A$ 
3 PARTITION  $A$  into  $B$ , elements  $< x$ , and  $C$ , elements  $\geq x$ 
4 If SIZE( $B$ )  $< x + 1$ 
   5   MISSINGINTEGER( $B, [i, x]$ )
6 Else MISSINGINTEGER( $C, [x + 1, j]$ )
```

The running time is  $O(n)$  because the recurrence for this algorithm is  $T(n) = T(n/2) + n$ , which is  $O(n)$  by the Master Method.

Common errors included using a randomized, instead of deterministic, partitioning scheme and using COUNTING SORT and then stepping through the array to find adjacent pairs that differ by two, which is not a Divide and Conquer approach.

(b) Suppose the integers in  $A$  are stored as  $k$ -bit binary numbers, i.e. each bit is 0 or 1.

For example, if  $k = 2$  and the array  $A = [01, 00, 11]$ , then the missing number is 10.

Now the only operation to examine the integers is  $\text{BIT-LOOKUP}(i, j)$ , which returns the  $j$ th bit of number  $A[i]$  and costs unit time. Design an  $O(n)$  algorithm to find the missing number. Argue (informally) that your algorithm is correct and analyze its running time. [8 points]

**Solution:** We shall examine bit by bit starting from the least significant bit to the most significant bit. Make a count of the number of 1's and 0's in each bit position, we can find whether the missing number has a 0 or 1 at the bit position being examined. Having done this, we have reduced the problem space by half as we have to search only among the numbers with that bit in that position. We continue in this manner till we have exhausted all the bit-positions (ie.,  $k$  to 1).

The algorithm is as follows. For convenience, we have 3 sets  $S, S_0, S_1$  which are maintained as link-lists.  $S$  contains the indices of the elements in  $A$  which we are going to examine while  $S_0$  ( $S_1$ ) contains the indices of the elements in  $A$  which have 0 (1) in the bit position being examined.

```
MISSING-INTEGER( $A, n$ )
1   $k \leftarrow \lg n$        $\triangleright n = 2^k - 1$ 
2   $S \leftarrow \{1, 2, \dots, n\}$ 
3   $S_0 \leftarrow S_1 \leftarrow \{\}$      $\triangleright$  Initialized to Empty List
4   $count0 \leftarrow count1 \leftarrow 0$ 
5  for  $posn \leftarrow k$  downto 1 do
6    for each  $i \in S$  do
7       $bit \leftarrow \text{BIT-LOOKUP}(i, posn)$ 
8      if  $bit = 0$ 
9        then  $count0 \leftarrow count0 + 1$ 
10       Add  $i$  to  $S_0$ 
11      else  $count1 \leftarrow count1 + 1$ 
12       Add  $i$  to  $S_1$ 
13      if  $count0 > count1$ 
14        then  $missing[posn] \leftarrow 1$ 
15         $S \leftarrow S_1$ 
16        else  $missing[posn] \leftarrow 0$ 
17         $S \leftarrow S_0$ 
18       $S_0 \leftarrow S_1 \leftarrow \{\}$ 
19       $count0 \leftarrow count1 \leftarrow 0$ 
20  return  $missing$ 
```

It can be noted that the following invariant holds at the end of the loop in Step 5-19.

- The bits of the missing integer in the bit position ( $posn$  to  $k$ ) is given by  $missing[posn] \dots missing[k]$ .
- $S = \{i : j^{\text{th}} \text{ bit of } A[i] = missing[j] \text{ for } posn \leq j \leq k\}$

This loop invariant ensures the correctness of the algorithm.

Each loop iteration (Step 5-19) makes  $|S|$   $\text{BIT-LOOKUP}$  operations. And the size of  $S$  is halved in each iteration. Hence total number of  $\text{BIT-LOOKUP}$  operations is  $\sum_{i=0}^{k-1} \frac{n}{2^i}$ , which is  $O(n)$ .

The grading for this problem was (-6) points for an algorithm that runs in  $\Theta(nk)$ , like RADIX SORT (or any of a number of built-from-scratch radix-sort-like approaches). Another point was deducted

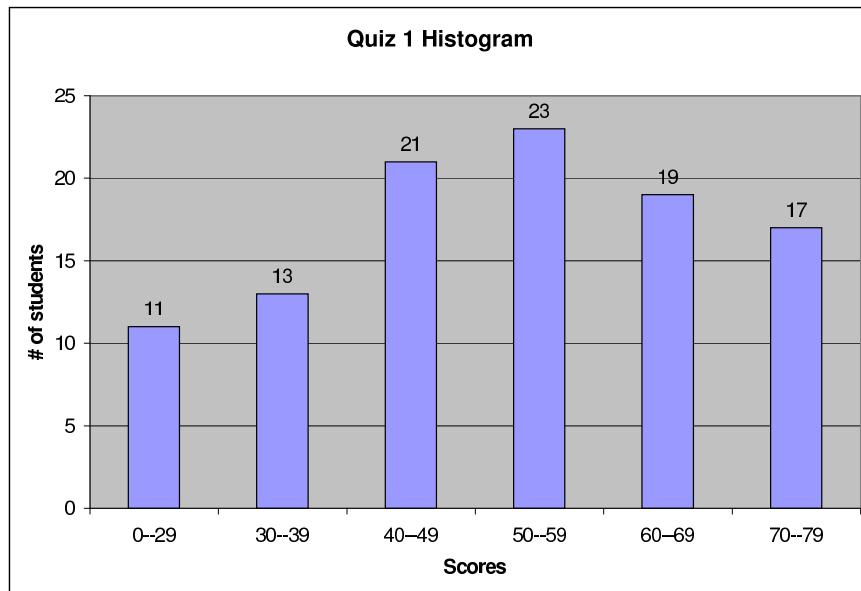
DETERMINISTICSELECT( $A, n, i$ )

- 1 Divide the elements of the input array  $A$  into groups of 3 elements.
- 2 Find median of each group of 3 elements and put them in array  $B$ .
- 3 Call DETERMINISTICSELECT( $B, n/3, n/6$ ) to find median of the medians,  $x$ .
- 4 Partition the input array around  $x$  into  $A_1$  containing  $k$  elements  $\leq x$   
and  $A_2$  containing  $n - k - 1$  elements  $\geq x$ .
- 5 If  $i = k + 1$ , then return  $x$ .
- 6 Else if  $i \leq k$ , DETERMINISTICSELECT( $A_1, k, i$ ).
- 7 Else if  $i > k$ , DETERMINISTICSELECT( $A_2, n - k - 1, i - (k + 1)$ ).

## SCRATCH PAPER

**SCRATCH PAPER** — Please detach this page before handing in your quiz.

## Quiz 1 Solutions



### Quiz 1 Statistics

Median	53
Mean	51.7
Std. Deviation	16.0
Max	79

Problem	Parts	Points	Grade	Grader
1	6	12		
2	1	13		
3	6	30		
4	3	25		
Total		80		

**Problem 1. Asymptotic Running Times [12 points] (6 parts)**

For each algorithm listed below, give its worst-case running time using  $\Theta$ -notation, as a function of the specified parameters. If the algorithm is randomized, give its expected running time. You need not justify your answers.

- (a) [2 points] EXTRACT-MIN in a Min-Heap of  $n$  elements.

**Solution:**  $T(n) = \Theta(\log n)$ .

- (b) [2 points] STRASSEN's algorithm to multiply  $n \times n$  matrices.

**Solution:**  $T(n) = \Theta(n^{\lg 7})$ .

- (c) [2 points] RADIX SORT on an  $n$ -element array of  $b$ -bit integers (with optimum choice of radix).

**Solution:**  $T(n) = \Theta(n + nb/(\lg n))$ .

- (d) [2 points] RANDOMIZED QUICKSORT on an  $n$ -element array.

**Solution:**  $T(n) = \Theta(n \lg n)$ .

- (e) [2 points] HASH-INSERT to insert an element into a chaining-based hash table with  $2n$  slots, a load factor of  $\alpha = 1/2$ , and a hash function chosen from a universal hash family.

**Solution:**  $T(n) = \Theta(1)$ .

- (f) [2 points] DETERMINISTIC SELECT to find the median of an  $n$ -element array.

**Solution:**  $T(n) = \Theta(n)$ .

**Problem 2. Recurrences** [13 points] Professor M. Onotono has managed to come up with a new variant of Mergesort. It splits an array with  $n$  elements into three parts, two of size  $n/4$  and one of size  $n/2$ . After sorting the subarrays, he can merge all three with just  $n$  comparisons. Thus the number of comparisons made by his algorithm, denoted  $T(n)$ , satisfies the recurrence:

$$T(n) \leq 2T(n/4) + T(n/2) + n,$$

with  $T(1) = T(2) = 1$ . Your task is to complete his analysis by proving a tight upper bound on  $T(n)$  using  $O$ -notation.

**Solution:** We guess that  $T(n) \leq cn \lg n$  for  $n \geq 2$ , and prove it by induction on  $n$ . By sloppiness, we assume that  $n$  is a power of 2. The base case  $n = 2$  says that  $T(2) = 1 \leq c2 \lg 2 = 2c$ , which holds provided  $c \geq 1/2$ . The base case  $n = 4$  says that  $T(4) = 2T(1) + T(2) + 4 = 2 + 1 + 4 = 7 \leq c4 \lg 4 = 8c$ , which holds provided  $c \geq 7/8$ . For the induction step, where  $n \geq 8$ , assume that  $T(n') \leq cn' \lg n'$  for all  $n' < n$  (in particular, for  $n' = n/2 \geq 4$  and  $n' = n/4 \geq 2$ ). Now consider  $T(n)$ :

$$\begin{aligned} T(n) &\leq 2c(n/4) \lg(n/4) + c(n/2) \lg(n/2) + n \\ &= c(n/2)(\lg n - 2) + c(n/2)(\lg n - 1) + n \\ &= cn \lg n - (3/2)cn + n \\ &= cn \lg n - ((3/2)c - 1)n \\ &\leq cn \lg n \end{aligned}$$

provided  $c \geq 2/3$ . Thus we can choose  $c = \max\{1/2, 7/8, 2/3\} = 7/8$ . Therefore,  $T(n) = O(n \lg n)$ .

**Problem 3. 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. The more content you provide in your justification, the higher your grade, but be brief. Your justification is worth more points than your true-or-false designation.

**T F** There exist functions  $f(n)$  and  $g(n)$  such that  $f(n) = O(g(n))$  and  $f(n) = \omega(g(n))$ .

**Solution: False.** If both are true, we find that there exist  $c, n_0$  such that for all  $n \geq n_0$  it is the case that  $f(n) \leq cg(n)$ . But we also have that for every  $c'$  there exists  $n_1$  such that for all  $n > n_1$ ,  $f(n) > c'g(n)$ . Applying this to  $c' = c$  and taking  $n = \max\{n_0, n_1\}$  we get  $f(n) \leq cg(n)$ , but  $f(n) > c'g(n) = cg(n)$  which clearly contradict each other.

**T F** Let  $f(n) = \sqrt{n}$  and  $g(n) = n \cdot (n \bmod 100)$ . Then  $f(n) = O(g(n))$ .

**Solution: False.** Assume otherwise. Then there should exist  $c, n_0$  such that for every  $n \geq n_0$  it is the case that  $f(n) \leq cg(n)$ . But this is clearly false if we pick  $n$  to be a large multiple of 100. In this case, we have  $g(n) = 0$ , while  $f(n) > 0$ .

**T F** If  $T(n) = 5T(n/4) + n$ , then  $T(n) = O(n \log^{5/4} n)$ .

**Solution:** **False.** By the Master method, we have  $T(n) = \Theta(n^{\log_4 5}) \notin O(n \log^{5/4} n)$ .

**T F** The probability that RANDOMIZED QUICKSORT takes  $\Omega(n^2)$  time to sort  $n$  inputs is at least  $1/(n!)$ .

**Solution:** **True.** If at each step the algorithm pivots on the element of highest rank, then Randomized Quicksort performs like deterministic Quicksort on a sorted array and takes  $\Omega(n^2)$  time. The probability of this happening is  $\prod_{k=1}^n (1/k) = 1/(n!)$ .

**T F** Consider an algorithm that takes an unsorted array of  $3n$  distinct integers  $A[1..3n]$  and produces two numbers  $x$  and  $y$  such that  $x < y$ ,  $n$  elements of  $A$  have value smaller than  $x$ ,  $n$  elements have value greater than  $y$ , and  $n$  elements have value between  $x$  and  $y$ . Any such algorithm working in the comparison model must take  $\Omega(n \log n)$  time.

**Solution: False.** This is an order-statistics problem: call  $\text{SELECT}(A, n + 1)$  to compute  $x$ , and call  $\text{SELECT}(A, 2n)$  to compute  $y$ . Each call takes  $O(n)$  time, for a total of  $O(n)$  time, which is  $o(n \log n)$ .

**T F** There exists a randomized algorithm that, in  $O(n)$  expected time, determines whether an array of  $n$  integers has repeated elements, i.e., whether there are two distinct indices  $i, j$  such that  $A[i] = A[j]$ .

**Solution: True.** Construct a universal hash function and build a hash table of size  $\Theta(n)$ , so that the load factor  $\alpha = \Theta(1)$ . For  $i = 1, 2, \dots, n$ , search for  $A[i]$  in the hash table. If that key is already present, report “yes”. Otherwise, insert  $A[i]$  into the hash table. Each search and possible insertion takes  $O(1 + \alpha) = O(1)$  time in expectation, so by linearity of expectation, the total running time is  $O(n)$  in expectation.

**Problem 4. Finding common elements with “=”** [25 points] (3 parts)

The Center for Disease Collection is trying to identify some common causes of the flu. To do so, they have blood samples from  $n$  infected cases, numbered  $1, 2, \dots, n$ . Each sample  $i$  contains traces of the virus  $A[i]$  responsible for the infection. The CDC can compare two samples  $i$  and  $j$  to tell whether the same virus caused the two infections (i.e., whether  $A[i] = A[j]$ ). However, they cannot perform any other type of comparison ( $<$ ,  $>$ ), nor can they hash viruses. The CDC would like to know whether there is a virus that is responsible for at least 10% of the cases, and if so, they would like a list of all such viruses. A test comparing two blood samples is time consuming and so they would like to minimize the number of tests performed.

We model this problem more precisely as follows. For a subarray  $A[p..q]$  and a real number  $\alpha$ ,  $0 < \alpha < 1$ , define an element  $x$  to be  $\alpha$ -common if  $A[i] = x$  for at least  $\alpha \cdot (q - p + 1)$  choices of the index  $i \in \{p, p + 1, \dots, q\}$ . Our goal is to find an algorithm that, given an array  $A[1..n]$  of  $n$  elements, reports all  $1/10$ -common elements in  $A[1..n]$ . The only operations that the algorithm is permitted to perform on values in  $A$  are equality comparisons, i.e., tests of the form “does  $A[i] = A[j]?$ ”. Our goal is to devise such an algorithm for finding  $1/10$ -common elements that runs in  $O(n \log n)$  time.

Throughout this problem, assume that  $n$  is a power of 2.

- (a) [3 points]** Give an upper bound on the number of distinct  $\alpha$ -common elements in  $A[p..q]$ . Write your answer as a function of  $\alpha$ ,  $p$ , and  $q$ .

**Solution:** In an array of size  $q - p + 1$ , the number of elements that can repeat  $\alpha(q - p + 1)$  times is at most  $1/\alpha$ . Thus, we can have at most  $1/\alpha$  distinct  $\alpha$ -common elements in  $A[p..q]$ .

- (b) [7 points] Prove that there is some  $\beta > 0$  such that any  $\alpha$ -common element in  $A[1..n]$  is  $\beta$ -common in *at least one* of the subarrays  $A[1..n/2]$  and  $A[n/2+1..n]$ . The larger the  $\beta$  the better.

**Solution:** We show that one can choose  $\beta = \alpha$ . Assume for contradiction that some element  $x$  is  $\alpha$ -common in  $A[1..n]$ , but not  $\alpha$ -common in either  $A[1..n/2]$  or  $A[n/2 + 1..n]$ . Let  $X$  be the number of occurrences of  $x$  in  $A[1..n]$ ,  $X_1$  be the number of occurrences of  $x$  in  $A[1..n/2]$ , and let  $X_2$  be the number of occurrences of  $x$  in  $A[n/2 + 1..n]$ .

Because  $x$  is  $\alpha$ -common in  $A[1..n]$ ,  $X \geq \alpha n$ . Because  $x$  is not  $\alpha$ -common in either  $A[1..n/2]$  or  $A[n/2 + 1..n]$ , we know that  $X_1 < \alpha n/2$  and  $X_2 < \alpha n/2$ . But  $X = X_1 + X_2 < \alpha n/2 + \alpha n/2 = \alpha n$ , leading to a contradiction. Therefore,  $x$  must  $\alpha$ -common in at least one of the subarrays  $A[1..n/2]$  and  $A[n/2 + 1..n]$ .

- (c) [15 points] Use the previous part to give a divide-and-conquer algorithm TOPTEN for finding the  $1/\alpha$ -common elements in  $A[p \dots q]$ . Describe the divide, conquer, and combine steps of your algorithm. Analyze the running time of your algorithm. (*Hint:* It should satisfy the same recurrence as MERGESORT.)

**Solution:**

**Algorithm:** Let  $\text{TOPTEN}(A, p, q, \alpha)$  be the function that returns the list of all  $\alpha$ -common elements in  $A[p \dots q]$ . We compute  $\text{TOPTEN}$  as follows:

*Base case:* If  $(q - p + 1) \leq 1/\alpha$ , then return a list  $Y$  of all the elements in  $A[p \dots q]$ .

*Divide and conquer step:* Divide  $A[p \dots q]$  in half, and recursively call  $\text{TOPTEN}$  on each half with the same parameter  $\alpha$ . Let  $Y_1 = \text{TOPTEN}(A, p, (p + q)/2, \alpha)$  and  $Y_2 = \text{TOPTEN}(A, (p + q)/2 + 1, q, \alpha)$ .

*Combine step:* Start with an empty list  $Y = \emptyset$  to store the answer. For each  $x$  in  $Y_1$ , scan through  $A[p \dots q]$  and count the occurrences of  $x$ . If  $x$  occurs more than  $\alpha(q - p + 1)$  times, then add  $x$  to  $Y$ . Repeat this loop for each  $x$  in  $Y_2$ . Return  $Y$  as the final answer.

**Correctness:** From (b), we know that any element  $x$  which is  $\alpha$ -common in  $A[p \dots q]$  must be  $\alpha$ -common in at least one of two subarrays we recurse on in the divide step. Thus, any candidate element  $x$  which might be  $\alpha$ -common must belong to one of  $Y_1$  and  $Y_2$ . In the base case, if  $n = (q - p + 1) < 1/\alpha$ , then every element in  $A[p \dots q]$  is  $\alpha$ -common.

**Runtime:** The divide step can be performed in constant time. In the combine step, for each  $x$ , the scan through  $A$  requires  $O(n)$  time. From (a), we know that there at most  $1/\alpha$  elements in  $Y_1$  and  $1/\alpha$  elements in  $Y_2$ . Thus, the runtime of the algorithm is given by the recurrence  $T(n) = 2T(n/2) + O(n/\alpha)$ , with base case  $T(n) = O(1)$  for  $n < 1/\alpha$ . For we have  $\alpha = 1/10$ , this recurrence is the same as for MERGESORT; thus, the runtime is  $O(n \lg n)$ .

## Quiz 1

- Do not open this quiz booklet until you are directed to do so. Read all the instructions first.
- The quiz contains **6** multi-part problems. You have 80 minutes to earn 80 points.
- This quiz booklet contains **16** pages, including this one and two extra sheets of scratch paper, which are included for your convenience.
- This quiz is closed book. You may use one double sided Letter ( $8\frac{1}{2}'' \times 11''$ ) or A4 crib sheet. No calculators or programmable devices are permitted.
- Write your solutions in the space provided. Extra scratch paper may be provided if you need more room, although your answer should fit in the given space.
- Do not waste time re-deriving facts that we have studied. It is sufficient to cite known results.
- Do not spend too much time on any one problem. Generally, a problem's point value is an indication of how much time to spend on it.
- 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.
- Good luck!

Problem	Points	Grade	Initials
1	1		
2	15		
3	27		
4	19		
5	9		
6	9		
Total	80		

Name: \_\_\_\_\_  
Circle your recitation time:

F10      F11      F12      F1      F2      F3

**Problem 1.** [1 points] Write your name on every page! Don't forget the cover.

**Problem 2. Recurrences** [15 points] (5 parts)

Solve the following recurrences by giving tight  $\Theta$ -notation bounds. You do not need to justify your answers, but any justification that you provide will help when assigning partial credit. As usual, assume that for  $n \leq 10$ ,  $T(n) = O(1)$ .

(a) [2 points]  $T(n) = 3T(n/5) + \lg^2 n$

**Solution:** Since  $\lg^2 n = O(n^{\log_3 5 - \epsilon})$ , this is the Case 1 of the Master Method. We have  $T(n) = \Theta(n^{\log_5 3})$ .

(b) [2 points]  $T(n) = 2T(n/3) + n \lg n$

**Solution:** Since  $n \lg n = \Omega(n^{\log_2 3 + \epsilon})$  and  $2\frac{n}{3} \lg \frac{n}{3} \leq \frac{2}{3}n \lg n$ , this is the Case 3 of the Master Method. We have  $T(n) = \Theta(n \lg n)$ .

- (c) [2 points]  $T(n) = T(n/5) + \lg^2 n$

**Solution:** Since  $\lg^2 = \Theta(n^{\log_1 5} \lg^2 n)$ , this is the Case 2 of the Master Method. We have  $T(n) = \Theta(\lg^3 n)$ .

- (d) [3 points]  $T(n) = T(n - 2) + \lg n$

**Solution:**  $T(n) = \Theta(n \log n)$ . This is  $\sum_{i=1}^{n/2} \lg 2i \geq \sum_{i=1}^{n/2} \lg i \geq (n/4)(\lg n/4) = \Omega(n \lg n)$ . For the upper bound, note that  $T(n) \leq S(n)$ , where  $S(n) = S(n-1) + \lg n$ , which is clearly  $O(n \lg n)$ .

(e) [6 points] An improvement to the multiplication method given in class involves splitting each  $n$ -bit number into *three* pieces of  $n/3$  bits each (i.e., write  $X$  as  $2^{2n/3}A + 2^{n/3}B + C$  and write  $Y$  as  $2^{2n/3}D + 2^{n/3}E + F$ ). A straightforward product would now involve 9 multiplications of  $n/3$ -bit numbers, but by cleverly rearranging terms, it is possible to reduce this to some number  $a$  multiplications of  $n/3$ -bit numbers, plus a constant number of additions and shifts.

- Set up and solve the recurrence for the improved running time algorithm in terms of  $n$  and  $a$ .
- What is the largest value of  $a$  such that this algorithm is asymptotically faster than the algorithms we learned in class, which ran in  $O(n^{1.58})$ ?

**Solution:** In order to beat the algorithm in class,

$$\begin{aligned}\log_3 a &< \log_2 3 \\ \lg a \lg 2 &< \lg^2 3 \\ a &\leq 5\end{aligned}$$

**Problem 3. True or False, and Justify [27 points] (9 parts)**

Circle **T** or **F** for each of the following statements, and briefly explain why. Your justification is worth more points than your true-or-false designation.

- (a) **T F** [3 points] The sum of the largest  $\lg n$  elements in an unsorted array of  $n$  distinct elements can be found in  $O(n)$  time.

**Solution:** True. We can use deterministic select algorithm to find the largest  $\lg n$  elements in  $O(n)$ . Compute sum of these number can be done in  $O(\lg n)$  time.

- (b) **T F** [3 points] A Monte Carlo algorithm for every possible input and every sequence of random coin tosses always returns a correct output. However with some nonzero probability over the coin tosses a Monte Carlo algorithm may not run in polynomial time.

**Solution:** False. Monte Carlo algorithms may return incorrect outputs with some probability.

- (c) **T F** [3 points] Let  $N$  be a positive integer and let  $a$  be an element in  $\mathbb{Z}_N^*$ . If  $a$  has at least three different square roots modulo  $N$ , then  $N$  is composite.

**Solution:** True. We can prove this claim by contradiction. Assume that  $N$  is a prime. Consider any integer  $a \in \mathbb{Z}_N^*$ , such that  $a$  has  $x$  as a square root modulo  $N$ . For any  $y \neq x$  and  $y \neq -x$ ,  $y^2 \neq x^2 = a$  modulo  $N$  (shown in **Homework 2-2**). Therefore,  $a$  has at most two different square roots modulo  $N$ . This contradicts to the fact that  $a$  has at least three different square roots modulo  $N$ . Therefore,  $N$  is composite.

- (d) **T F** [3 points] An adversary can construct an input of length  $n$  to force RANDOMIZED-QUICKSORT to run in  $\Omega(n^2)$  time.

**Solution:** False. The *expected* running time of RANDOMIZED QUICKSORT is  $\Theta(n \log n)$ . This applies to *any* input.

- (e) **T F** [3 points] Searching in a skip list takes  $\Theta(\log n)$  time with high probability, but could take  $\Omega(2^n)$  time with nonzero probability.

**Solution:** True. A skip list could be of any height or be a simple linked list, depending on its random choices.

- (f) **T F** [3 points] If an operation runs in  $O(f(n))$  amortized time, it also takes  $O(f(n))$  worst-case time (per operation).

**Solution:** False. Consider the **table doubling** example in the lecture. Although the amortized cost of each operation in this example is  $O(1)$  but the worst-case time is  $O(n)$ .

- (g) **T F** [3 points] Let  $k, i, j$  be integers, where  $k > 3$  and  $1 \leq i, j \leq k$ . Let  $h_{ij}^k$  be the hash function mapping a  $k$ -bit integer  $b_1 b_2 \dots b_k$  to the 2-bit value  $b_i b_j$ . For example,  $h_{31}^8(00\underline{1}01011) = 10$ .

The set  $\{h_{ij}^k : 1 \leq i, j \leq k\}$  is a universal family of hash functions from  $k$ -bit integers into  $\{00, 01, 10, 11\}$ .

**Solution:** False. Take  $x = 0, y = 1$ . Then all of  $x$ 's binary digits are the same as  $y$ 's, except for the least significant bit. Thus, if we choose one of the  $k$  hash functions at random,  $\Pr[h(x) = h(y)] = (k^2 - k)/k^2 = (k - 1)/k > 1/4$ . If this were a universal class of hash functions, then this probability should be at most  $1/4$ .

- (h) **T F** [3 points] The rightmost child subtree of the root of an  $n$ -node 2-3 tree contains  $\Omega(n)$  nodes.

**Solution:** False. A tree with all degree-3 nodes on one subtree and degree-2 nodes on the other will have depth  $h = \log_3 n$ . There will be  $2^{\log_3 n} = n^{\log_3 2} = o(n)$  nodes on the sparse subtree.

- (i) **T F** [3 points] The following Monte Carlo primality testing algorithm has error probability less than  $1/2$ .

**Algorithm** FERMAT-TEST. On input  $N > 2$ :

1. Repeat the following twice:
  - (a) Pick a random integer  $A \in \mathbb{Z}_N^*$ .
  - (b) If  $A^{N-1} \neq 1 \pmod{N}$ , then return “composite”.
2. If the algorithm did not return “composite” in step 1, return “probably prime”.

**Solution:** False. If  $N$  is a Carmichael number, the algorithm always returns “probably prime”. Therefore, the error probability for the case that  $N$  is a Carmichael number is 1.

**Problem 4. Short Answer [19 points] (4 parts)**

Give *brief*, but complete, answers to the following questions.

- (a) [5 points] You are maintaining a collection of linked lists that support the following operations:

INSERT(item, list): insert item into list (cost = 1).

SUM(list): sum the items in list, and replace the list with one item that is the sum (cost = length of list).

Show that, starting with empty lists, we can assign an amortized cost of 2 to each INSERT and an amortized cost of 1 to each SUM.

**Solution:** We'll maintain the invariant that every item has one credit. Insert gets 2 credits, which covers one for the actual cost and one to satisfy the invariant. Sum gets one credit, because the actual cost of summing is covered by the credits in the list, but then the result of the sum will need one credit to maintain the invariant.

Many people confused credit arguments and potential arguments and tried to use a potential function even though one was never defined. These are equivalent, but different. You should only use one at a time.

- (b) [4 points] Suppose we insert an element into a 2-3 tree, and the INSERT algorithm splits  $k$  nodes. Give an *exact* (not big-O) upper bound on the number of nodes in the tree that are created or modified in this case.

**Solution:** Each split of node  $x$  creates one new node ( $x$ 's sibling) and modifies one (i.e.  $x$ ), so two nodes are created / modified for each split. The split also results in promoting a key to  $x$ 's parent, which may further cause the parent node to split. If the parent node splits, the modification to the parent is already accounted for in the splitting of the parent node. Thus, two nodes are created / modified for each split. On the other hand, we need to account for an additional modification to the final node (closest to the root) where the split stops. Or, if the split continues all the way to the root, a new root is created. Therefore, for  $k$  splits, a total of  $2k + 1$  nodes are created / modified. Alternatively, when a node  $x$  splits, one may delete  $x$  entirely, and replace  $x$  with two new nodes (each getting half of  $x$ 's keys). In which case, three nodes are created / modified / deleted, and the answer is then  $3k + 1$ .

(c) [6 points] Suppose that you are given an array  $A$  of  $n$  bits that either contains all zeros or contains  $2n/3$  zeros and  $n/3$  ones in some arbitrary order. Your goal is to determine whether  $A$  contains any ones.

1. Give an exact lower bound in terms of  $n$  (not using asymptotic notation) on the worst-case running time of any deterministic algorithm that solves this problem.
2. Give a randomized algorithm that runs in  $O(1)$  time and gives the right answer with probability at least  $1/3$ .
3. Give a randomized algorithm that runs in  $O(1)$  time and gives the right answer with probability at least  $5/9$ .

**Solution:**

1. Any correct deterministic algorithm must look at  $2n/3 + 1$  entries, because if it didn't, it could see all zeros even when there was a one somewhere.
2. Pick a random location. If it is a 1, output "has ones". Otherwise, output "doesn't have ones". Clearly the algorithm runs in  $O(1)$  time. If the array is all zeros, it is always correct.

However if the array contains  $1/3$  ones, the algorithm may make a mistake. Since there are  $n/3$  ones, you will select a 1 value at random and correctly output "has ones" with probability  $1/3$ .

3. Pick two random locations. The correctness and  $O(1)$  runtime are identical as above. Again, if the array is all zeros, the algorithm is always correct.

A mistake arises if the array contains ones, but the algorithm picks two zeros. Picking a single zero occurs with probability  $2/3$ , and picking two zeros independently occurs with probability  $(2/3)^2 = 4/9$ . Therefore, the probability that the algorithm is correct is  $1 - (4/9) = 5/9$ .

- (d) [4 points] Show that, if all edge weights in a graph are distinct, then the minimum spanning tree is unique.

**Solution:** Suppose (for contradiction) that there were two different MSTs  $T_1$  and  $T_2$  of a graph  $G = (V, E)$  with distinct edge weights; let  $e$  be the lightest edge in  $E(T_1) \setminus E(T_2) \cup E(T_2) \setminus E(T_1)$ , where  $E(T)$  denotes the set of edges in a tree  $T$ . Wlog, assume that  $e \in T_1$ . Consider the graph  $G' = (V, E(T_2) \cup \{e\})$ .  $G'$  has a cycle that contains  $e$ . In this cycle, there must be an edge  $e' \in E(T_2) \setminus E(T_1)$  (otherwise  $T_1$  will contain the cycle). By the definition,  $e'$  must have greater weight than  $e$ . Consider the graph  $G'' = (V, E(T_2) \cup \{e\} \setminus \{e'\})$ .  $G''$  is connected and has exactly  $n - 1$  edges. Therefore,  $G''$  is a MST. Also, the total weight of  $G''$  is less than  $T_2$ , a contradiction.

**Problem 5. Tree cover** [9 points] Let  $T$  be a rooted tree with  $n$  nodes  $\{1, 2, \dots, n\}$  and parent array  $P[1, 2, \dots, n]$  where  $P[i] = j$  if the node  $j$  is the parent of  $i$ , and  $P[i] = i$  if  $i$  is the root of the tree. Thus the edge set of the tree is  $\{\{i, P[i]\} \mid P[i] \neq i\}$ .

We say that a vertex  $c$  **covers** an edge  $e = \{u, v\}$  if  $c$  is one of the endpoints of  $e$ , that is,  $c = u$  or  $c = v$ . The **vertex cover problem** is the following: given a tree  $T$ , find a minimum-size subset of vertices  $S \subseteq \{1, 2, \dots, n\}$  such that  $S$  covers all edges, i.e., every edge  $\{i, P[i]\}$  in the tree is covered by at least one vertex in  $S$ .

Give a polynomial-time greedy algorithm to solve the problem above and prove its correctness. You may assume that each node  $x$  has an attribute  $x.\text{depth}$  that stores the depth of  $x$  in the tree.

**Solution:** Our algorithm works as follows:

1. Sort all the nodes in a nonincreasing order of  $\text{depth}$ .
2. Set  $S$  to  $\emptyset$ .
3. Scan through all the nodes in the sorted array, for a node  $x$ , if the edge  $(x, P[x])$  has not been covered, add  $P[x]$  ( $x$ 's parent node) into  $S$ .

Running time analysis: This algorithm takes  $O(n \lg n)$  to sort, and  $O(n)$  to scan through the sorted array (i.e., each step in the scan can be implemented in  $O(1)$  time). Therefore, the total running time is  $O(n \lg n)$ . The running time can be improved to  $O(n)$  by using COUNTING-SORT (see book).

Proof of correctness: Observe that at the time when we consider a vertex  $x$  in our algorithm, all edges incident to vertices before  $x$  are already covered. When we consider a vertex  $x$ , if edge  $(x, P[x])$  which has not been covered, in order to cover this edge we have to either pick  $x$  or  $P[x]$  (picking both is wasteful).

1. Greedy choice property: Picking  $P[x]$  is the best option since  $P[x]$  covers as least as many (uncovered) edges as  $x$  does. .
2. Optimal substructure:  $P[x]$  covers all (uncovered) edges that  $x$  covers (which is  $(x, P[x])$ ). Therefore, if there is an optimal solution that picks  $x$ , there should be another optimal solution that replaces  $x$  by  $P[x]$ . Therefore, picking  $P[x]$  should lead to an optimal solution.

**Problem 6. Average in 2-3 Trees [9 points] (2 parts)** You are the IT consultant who is managing the salary database for a big firm. The firm salary database is designed as a 2-3 tree<sup>1</sup> in which each node in the tree stores the salary for one or two employee (as in regular 2-3 tree).

Every month the firm has to submit a report to the Department of Fairness about the salary for low-income employees in the firm. In particular, the report should contain the average salary of all employees with salary at most  $x$  for some  $x > 0$ . In order to compile the report efficiently, you are asked to implement a new operation  $\text{AVERAGE}(x)$  in the database which returns the average salary of all employees whose salary is at most  $x$ .

- (a) [6 points] What extra information needs to be stored at each node? Describe how to answer an  $\text{AVERAGE}(x)$  query in  $O(\lg n)$  time using this extra information.

**Solution:** Each node  $x$  should store  $x.size$  - the size of the subtree rooted at  $x$  - and  $x.sum$  - the sum of all the key values in the subtree rooted at  $x$ . For a value  $x > 0$ , let  $S_x$  be the set of all keys less than or equal to  $x$ . Let  $A_x$  and  $B_x$  be the sum and the size of  $S_x$ .

We can compute  $A_x$  as follows. Let  $u$  be the leaf with smallest key larger than  $x$ . Finding  $u$  from the root only takes  $O(\lg n)$  time by using  $\text{SEARCH}$  in a 2-3 tree. Now consider the path from the root of the tree to  $u$ . Clearly,  $A_x$  is the sum of all leaves that are on the left of this path. Therefore,  $A_x$  can be computed by summing up all  $y.sum$ 's for every node  $y$  that is a left sibling of a node in the path. Since there are only  $\lg n$  such nodes  $y$ 's, computing  $A_x$  only takes  $O(\lg n)$  time.

Computing  $B_x$  is similar: instead of summing up  $y.sum$ , we sum up  $y.size$ . Therefore, it also takes  $O(\lg n)$  time to compute  $B_x$ .

Therefore,  $\text{AVERAGE}(x)$  which is  $\frac{A_x}{B_x}$  can be answered in  $O(\lg n)$  time.

---

<sup>1</sup>so that it is efficient to Search, Insert, or Delete by salary

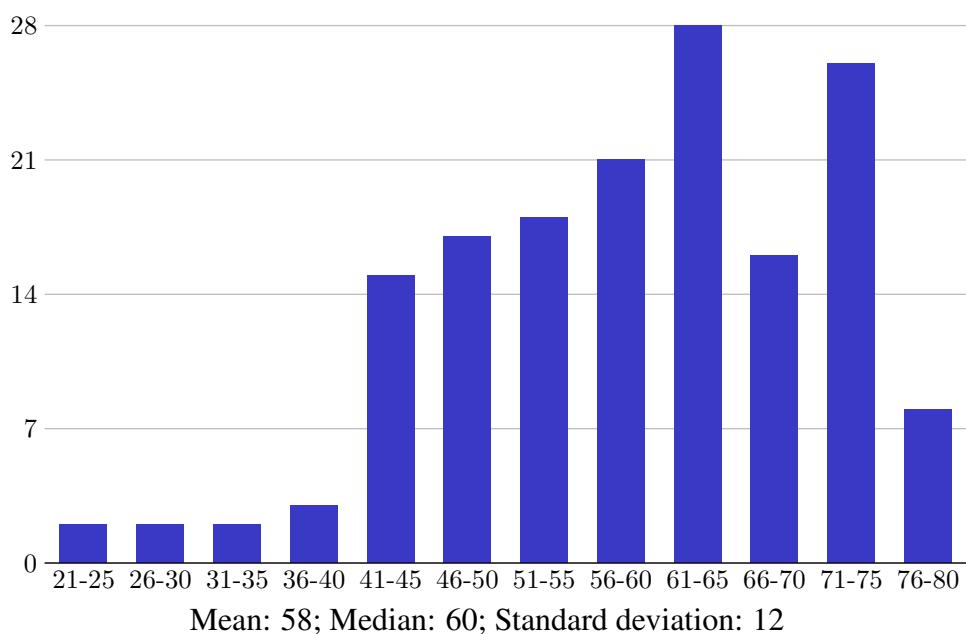
- (b) [3 points] Describe how to modify INSERT to maintain this information. Briefly justify that the worst-case running time for INSERT remains  $O(\lg n)$ .

**Solution:** Maintaining  $x.size$  is similar to what was covered in recitation and homework. Maintaining  $x.sum$  is exactly the same: when a node  $x$  gets inserted, we simply increase  $y.sum$  for every ancestor  $y$  of  $x$  by the amount  $x.key$ . Handling overflow for  $x.sum$  is exactly the same as  $x.size$ . Hence, INSERT still runs in worst-case time  $O(\lg n)$ .

## SCRATCH PAPER

## SCRATCH PAPER

## Quiz 1 Solutions



**Problem 0. Name.** [1 point] Write your name on every page of this exam booklet! Don't forget the cover.

**Problem 1. True or False.** [24 points] (8 parts)

Circle **T** or **F** for each of the following statements to indicate whether the statement is true or false, and briefly explain why. Your justification is worth more points than your true-or-false designation.

- (a) **T F** The solution to the recurrence  $T(n) = 2^n T(n - 1)$  is  $T(n) = \Theta((\sqrt{2})^{n^2+n})$ .  
 (Assume  $T(n) = 1$  for  $n$  smaller than some constant  $c$ ).

**Solution:** [3 points] True. Let  $T(0) = 1$ .

$$\text{Then } T(n) = 2^n \cdot 2^{n-1} \cdot 2^{n-2} \dots 2^1 = 2^{(n+(n-1)+(n-2)+\dots+1)}.$$

$$\text{Because } \sum_{i=1}^n i = n(n+1)/2, \text{ we therefore have } T(n) = 2^{(n^2+n)/2} = (\sqrt{2})^{n^2+n}.$$

Some students also correctly solved the problem by using the substitution method.  
 Some students made the mistake of multiplying the exponents instead of adding them. Also, it has to be noted that  $2^{n^2/2} \neq \Theta(2^{n^2})$ .

- (b) **T F** The solution to the recurrence  $T(n) = T(n/6) + T(7n/9) + O(n)$  is  $O(n)$ .  
 (Assume  $T(n) = 1$  for  $n$  smaller than some constant  $c$ ).

**Solution:** [3 points] True. Using the substitution method:

$$\begin{aligned} T(n) &\leq cn/6 + 7cn/9 + an \\ &\leq 17cn/18 + an \\ &\leq cn - (cn/18 - an) \end{aligned}$$

This holds if  $c/18 - a \geq 0$ , so it holds for any constant  $c$  such that  $c \geq 18a$ .

Full credit was also given for solutions that uses a recursion tree, noting that the total work at level  $i$  is  $(17/18)^i n$ , which converges to  $O(n)$ .

- (c) **T F** In a simple, undirected, connected, weighted graph with at least three vertices and unique edge weights, the heaviest edge in the graph is in no minimum spanning tree.

**Solution:** [3 points] False. If the heaviest edge in the graph is the only edge connecting some vertex to the rest of the graph, then it must be in every minimum spanning tree.

- (d) **T F** The weighted task scheduling problem with weights in the set  $\{1, 2\}$  can be solved optimally by the same greedy algorithm used for the unweighted case.

**Solution:** [3 points] False. The algorithm will fail given the set of tasks (given in the form  $((s_i, f_i), w_i)$ ):  
 $\{((0, 1), 1), ((0, 2), 2)\}$ .

- (e) **T F** Two polynomials  $p, q$  of degree at most  $n - 1$  are given by their coefficients, and a number  $x$  is given. Then one can compute the multiplication  $p(x) \cdot q(x)$  in time  $O(\log n)$ .

**Solution:** [3 points] False. We need at least  $\Theta(n)$  time to evaluate each polynomial on  $x$  and to multiply the results. Some students argued incorrectly that it must take  $O(n \log n)$  using FFT, but FFT overkills because it computes all coefficients, not just one.

- (f) **T F** Suppose we are given an array  $A$  of  $n$  distinct elements, and we want to find  $n/2$  elements in the array whose median is also the median of  $A$ . Any algorithm that does this must take  $\Omega(n \log n)$  time.

**Solution:** [3 points] False. It's possible to do this in linear time using SELECT: first find the median of  $A$  in  $\Theta(n)$  time, and then partition  $A$  around its median. Then we can take  $n/4$  elements from either side to get a total of  $n/2$  elements in  $A$  whose median is also the median of  $A$ .

- (g) **T F** There is a density  $0 < \rho < 1$  such that the asymptotic running time of the Floyd-Warshall algorithm on graphs  $G = (V, E)$  where  $|E| = \rho |V|^2$  is better than that of Johnson's algorithm.

**Solution:** [3 points] False. The asymptotic running time of Floyd-Warshall is  $O(V^3)$ , which is at best the same asymptotic running time of Johnson's (which runs in  $O(VE + V \log V)$  time), since  $E = O(V^2)$

- (h) **T F** Consider the all pairs shortest paths problem where there are also weights on the vertices, and the weight of a path is the sum of the weights on the edges and vertices on the path. Then, the following algorithm finds the weights of the shortest paths between all pairs in the graph:

APSP-WITH-WEIGHTED-VERTICES( $G, w$ ):

- 1 **for**  $(u, v) \in E$
- 2     Set  $w'(u, v) = (w(u) + w(v))/2 + w(u, v)$
- 3     Run Johnson's algorithm on  $G, w'$  to compute the distances  $\delta'(u, v)$  for all  $u, v \in V$ .
- 4 **for**  $u, v \in V$
- 5     Set  $d_{uv} = \delta'(u, v) + \frac{1}{2}(w(u) + w(v))$

**Solution:** [3 points] True. Any shortest path from  $u$  to  $v$  in the original graph is still a shortest path in the new graph. For some path  $\{v_0, v_1, \dots, v_k\}$ , we have:

$$\begin{aligned}
 w'(v_0 \rightsquigarrow v_k) &= \sum_{i=0}^{k-1} ((w(v_i) + w(v_{i+1}))/2 + w(v_i, v_{i+1})) \\
 &= \sum_{i=0}^{k-1} w(v_i, v_{i+1}) + \frac{1}{2} \left( \sum_{i=0}^{k-1} w(v_i) + \sum_{i=1}^k w(v_i) \right) \\
 &= \sum_{i=0}^{k-1} w(v_i, v_{i+1}) + \sum_{i=0}^k w(v_i) - \frac{1}{2}(w(v_0) + w(v_k)) \\
 &= w(v_0 \rightsquigarrow v_k) - \frac{1}{2}(w(v_0) + w(v_k))
 \end{aligned}$$

Therefore, the order of all paths from  $v_0$  to  $v_k$  remains unchanged so Johnson's algorithm in line 3. finds the correct path, and the adjustment in line 5 finds the correct length  $d_{v_0 v_k}$ .

**Problem 2. Translation** [25 points] (5 parts)

You have been hired to manage the translation process for some documentation. Unfortunately, different sections of the documentation were written in different languages:  $n$  languages in total. Your boss wants the entire documentation to be available in all  $n$  languages.

There are  $m$  different translators for hire. Some of those translators are volunteers that do not get any money for their services. Each translator knows *exactly* two different languages and can translate back and forth between them. Each translator has a non-negative hiring cost (some may work for free). Unfortunately, your budget is too small to hire one translator for each pair of languages. Instead, you must rely on chains of translators: an English-Spanish translator and a Spanish-French translator, working together, can translate between English and French. Your goal is to find a minimum-cost set of translators that will let you translate between every pair of languages.

We may formulate this problem as a connected undirected graph  $G = (V, E)$  with non-negative (i.e., zero or positive) edge weights  $w$ . The vertices  $V$  are the languages for which you wish to generate translations. The edges  $E$  are the translators. The edge weight  $w(e)$  for a translator  $e$  gives the cost for hiring the translator  $w(e)$ . A subset  $S \subseteq E$  of translators can be used to translate between  $a, b \in V$  if and only if the subgraph  $G_S = (V, S)$  contains a path between  $a$  and  $b$ . The set  $S \subseteq E$  is a translation network if and only if  $S$  can be used to translate between all pairs  $a, b \in V$ .

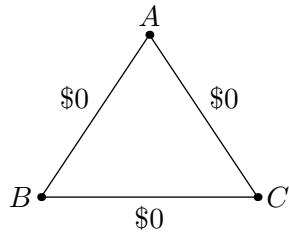
- (a) Prove that each minimum spanning tree of  $G$  is also a minimum-cost translation network.

**Solution:** [5 points] Let  $T$  be some minimum spanning tree of  $G$ . Because  $T$  is a spanning tree, there is a path in  $T$  between any pair of vertices. Hence,  $T$  is a translation network.

For the sake of contradiction, suppose that  $T$  is not a minimum-cost translation network. Then there must be some translation network  $S$  with total cost strictly smaller than  $T$ . Because  $S$  connects every pair of vertices, it must have some spanning tree  $T^*$  as a subgraph. Because all edge weights are nonnegative, we have  $w(T^*) \leq w(S) < w(T)$ . Hence,  $T$  is not the minimum spanning tree. This contradiction means that all spanning trees are also minimum-cost translation networks.

- (b) Give an example of a minimum-cost translation network that is not a minimum spanning tree of  $G$ .

**Solution:** [5 points] If the graph of translators contains a cycle of translators all willing to work for \$0, then it is possible to hire all of the translators in the cycle without increasing the overall cost of the translation network. The smallest example of this is the following:



All three MSTs of the graph have total cost \$0, so any translation network of cost \$0 has minimum cost. Hence, we can take all translators in the cycle to get a minimum-cost translation network that is not an MST.

- (c) Give an efficient algorithm that takes  $G$  as input, and outputs a minimum-cost translation network of  $G$ . State the runtime of your algorithm in terms of the number of languages  $n$  and the number of potential translators  $m$ .

**Solution:** [5 points] We saw in part (a) that every minimum spanning tree is a minimum-cost translation network of  $G$ . Hence, to find a minimum-cost translation network, it is sufficient to find a minimum spanning tree of  $G$ . We may do so using Kruskal's algorithm, for a runtime of  $\Theta(m \log n)$ , or using Prim's algorithm, for a runtime of  $\Theta(m + n \log n)$ .

Your bosses have decided that the previous approach to translation doesn't work. When attempting to translate between Spanish and Portuguese — two relatively similar languages — it degrades the translation quality to translate from Spanish to Tagalog to Mandarin to Portuguese. There are certain clusters of languages that are more closely related than others. When translating between two languages that lie within the same cluster, such as Spanish and Portuguese, the translation is of high quality when the sequence of languages used to translate between them is completely contained within the cluster.

More formally, the language set  $V$  can be divided into disjoint clusters  $C_1, \dots, C_k$ . Each cluster  $C_i$  contains languages that are fairly similar; each language is contained in exactly one cluster. Your bosses have decided that a translation between  $a, b \in C_i$  is high-quality if and only if all of the languages used on the path from  $a$  to  $b$  are also in  $C_i$ . The translator set  $S$  is a high-quality translation network if and only if it is a translation network, and for any language cluster  $C_i$  and any languages  $a, b \in C_i$ ,  $S$  can be used for a high-quality translation between  $a$  and  $b$ .

- (d) Suppose that  $S$  is a minimum-cost high-quality translation network. Let  $S_i = S \cap (C_i \times C_i)$  be the part of the network  $S$  that lies within the cluster  $C_i$ . Show that  $S_i$  is a minimum-cost translation network for the cluster  $C_i$ .

**Solution:** [5 points] Let  $S_i^*$  be a minimum-cost translation network for  $C_i$ . Because  $S$  is a high-quality translation network,  $S_i$  must contain a path between every pair of nodes in  $C_i$ , so  $S_i$  is a translation network for  $C_i$ . For the sake of contradiction, assume that  $S_i$  is not minimum-cost. Then  $w(S_i) > w(S_i^*)$ .

Consider the translation network  $S^* = (S - S_i) \cup S_i^*$ . Then  $w(S^*) = w(S) - w(S_i) + w(S_i^*) < w(S)$ . Because  $S_i^*$  is a translation network of  $C_i$ , replacing  $S_i$  with  $S_i^*$  will not disconnect any pair of vertices in the graph. Furthermore, any pair of vertices connected by a path in  $S$  that lay inside a particular cluster will be connected by a path in  $S^*$  that lies within the same cluster. Hence,  $S^*$  is a high-quality translation network with cost strictly less than  $S$ . This contradicts the definition of  $S$ , so  $S_i$  must be a minimum-cost translation network.

- (e) Give an efficient algorithm for computing a minimum-cost high-quality translation network. Analyze the runtime of your algorithm in terms of the number of languages  $n$  and the number of translators  $m$ .

**Solution:** [5 points] The idea behind this algorithm is to first compute one MST for each individual cluster, and then to compute a global MST using the remaining edges. More specifically, we do the following:

1. For each edge  $(u, v)$ , if there is some cluster  $C_i$  such that  $u, v \in C_i$ , then add  $(u, v)$  to the set  $E_i$ . Otherwise, add  $(u, v)$  to the set  $E_{global}$ .

2. For each cluster  $C_i$ , run Kruskal's algorithm on the graph  $(C_i, E_i)$  to get a minimum-cost translation network  $T_i$  for the cluster. Take the union of these minimum to get a forest  $T$ .
3. Construct an empty graph  $G_{global}$  on nodes  $\{1, \dots, k\}$ . For each edge  $(u, v)$  in  $E_{global}$ , where  $u \in C_i$  and  $v \in C_j$ , check whether the edge  $(i, j)$  is in the graph  $G_{global}$ . If so, set  $w(i, j) = \min\{w(i, j), w(u, v)\}$ . Otherwise, add the edge  $(i, j)$  to the graph  $G_{global}$ . In either case, keep a mapping  $source(i, j) = (u^*, v^*)$  such that  $w(i, j) = w(u^*, v^*)$ .
4. Run Kruskal's algorithm on the graph  $G_{global}$  to get  $T_{global}$ .
5. For each edge  $(i, j) \in T_{global}$  add the edge  $source(i, j)$  to  $T$ .

We begin by examining the runtime of this algorithm. The first step requires us to be able to efficiently discover the cluster  $C_i$  that contains each vertex, which can be precomputed in time  $\Theta(m)$  and stored with the vertices for efficient lookup. So this filtering step requires  $\Theta(1)$  lookup per edge, for a total of  $\Theta(m)$  time.

The second step is more complex. We run Kruskal's algorithm on each individual cluster. So for the cluster  $C_i$ , the runtime is  $\Theta(|E_i| \lg |C_i|)$ . The total runtime here is:

$$\sum_{i=1}^k a \cdot |E_i| \lg |C_i| \leq \sum_{i=1}^k a \cdot |E_i| \lg n = (a \lg n) \sum_{i=1}^k |E_i| \leq am \lg n$$

Hence, the total runtime for this step is  $\Theta(m \lg n)$ .

The third step also requires care. We can store the graph to allow us to efficiently lookup  $w(\cdot, \cdot)$  and to tell whether an edge  $(i, j)$  has been added to the graph. We can also store  $source(\cdot, \cdot)$  to allow for  $\Theta(1)$  lookups. So the runtime here is bounded by  $\Theta(m)$ . The fourth step is Kruskal's again, only once, on a graph with  $\leq n$  vertices and  $\leq m$  edges, so the total runtime is  $\Theta(m \lg n)$ . The final step involves a lookup for each edge  $(i, j) \in T$ , for a total runtime of  $\Theta(k) \leq \Theta(n)$ . Hence, the runtime is dominated by the two steps involving Kruskal. So the total worst-case runtime is  $\Theta(m \lg n)$ .

Next we consider the correctness of this algorithm. Suppose that the result of this process is not the minimum-cost high-quality translator network. Then there must be a high-quality translator network  $S$  that has strictly smaller cost. Because  $S$  is a minimum-cost high-quality translator network, we know that the portion of  $S$  contained in the cluster  $C_i$  is a minimum-cost translator network for  $C_i$ , which has the same cost as the minimum spanning tree for that cluster computed in step 2 of the algorithm. So the total weight of all inter-cluster edges in  $S$  must be strictly less than the total weight of all inter-cluster edges in  $T$ . But the set of all inter-cluster edges in  $T$  formed a minimum spanning tree on the cluster, so any strictly smaller set of edges cannot span the set of all clusters. So  $S$  cannot be a translation network. This contradicts our assumption, and so  $T$  must be a minimum-cost high-quality translation network.

**Problem 3. All Pairs Shortest Red/Blue Paths.** [18 points] (3 parts)

You are given a directed graph  $G = (V, E)$  with edge weights  $w : E \rightarrow \mathbb{R}$ . In addition, each edge of the graph is either red or blue. The shortest red/blue path from vertex  $i \in V$  to vertex  $j \in V$  is defined as the shortest path from  $i$  to  $j$  among those paths that go through *exactly* one red edge (if there are no such paths, the length of the shortest red/blue path is  $\infty$ ).

We can represent this graph with two  $n \times n$  matrices of edge weights,  $W_r$  and  $W_b$ , where  $W_r$  contains the weights of all red edges, and  $W_b$  contains the weights of all blue edges.

- (a) Given the Floyd-Warshall algorithm below, how would you modify the algorithm to obtain the lengths of the shortest paths that only go through blue edges?

FLOYD-WARSHALL( $W$ ):

```

1   $n = W.\text{rows}$ 
2   $D^{(0)} = W$ 
3  for  $k = 1$  to  $n$ 
4    let  $D^{(k)} = (d_{ij}^{(k)})$  be a new  $n \times n$  matrix
5    for  $i = 1$  to  $n$ 
6      for  $j = 1$  to  $n$ 
7         $d_{ij}^{(k)} = \min(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)})$ 
8  return  $D^{(n)}$ 
```

**Solution:** [6 points] In order to find shortest paths going through only blue edges, it suffices to ignore the red edges and run Floyd-Warshall on only the blue edges of the graph. We make the following changes:

- Replace each occurrence of  $W$  with  $W_b$  in lines 1 and 2.
- Replace the matrices  $D^{(k)}$  with blue versions  $D_b^{(k)}$  in lines 2, 4, and 8.
- Replace the matrix elements  $d_{ij}^{(k)}$  with blue versions  $d_{b,ij}^{(k)}$  in lines 4 and 7.

While the second and third changes above are unnecessary for this part, they lay the groundwork for future parts below.

- (b) How would you modify your algorithm from part (a) to keep track not only of shortest paths with only blue edges, but also those with exactly one red edge, and to output the lengths of the shortest red/blue paths for all pairs of vertices in this graph?

**Solution:** [6 points] Add a new set of matrices  $D_r^{(k)}$  that give lengths of shortest paths with exactly one red edge and intermediate vertices up to  $k$ . The resulting pseudocode is as follows:

RED-BLUE-FLOYD-WARSHALL( $W_r, W_b$ ):

```

1    $n = W_r.rows$ 
2    $D_b^{(0)} = W_b$ 
3    $D_r^{(0)} = W_r$ 
4   for  $k = 1$  to  $n$ 
5       let  $D_b^{(k)} = (d_{b,ij}^{(k)})$ ,  $D_r^{(k)} = (d_{r,ij}^{(k)})$  be new  $n \times n$  matrices
6       for  $i = 1$  to  $n$ 
7           for  $j = 1$  to  $n$ 
8                $d_{b,ij}^{(k)} = \min(d_{b,ij}^{(k-1)}, d_{b,ik}^{(k-1)} + d_{b,kj}^{(k-1)})$ 
9                $d_{r,ij}^{(k)} = \min(d_{r,ij}^{(k-1)}, d_{r,ik}^{(k-1)} + d_{b,kj}^{(k-1)}, d_{b,ik}^{(k-1)} + d_{r,kj}^{(k-1)})$ 
10  return  $D_r^{(n)}$ 
```

(c) Prove the correctness of your algorithm using a loop invariant.

**Solution:** [6 points] The procedure for keeping track of paths that go through only blue edges is exactly equivalent to running Floyd-Warshall on the subgraph that contains only the blue edges of  $G$ , which is sufficient to show the correctness of  $D_b^{(k)}$  for all  $k$ .

Loop invariant: at the end of every iteration of the for loop from  $k = 1$  to  $n$ , we have both the length of the shortest path for every pair  $(i, j)$  going through only blue edges and the length of the shortest path for every pair  $(i, j)$  going through exactly one red edge, in each using intermediate vertices only up to  $k$ .

Initialization: At initialization  $k = 0$ , there are no intermediate vertices. The only blue paths are blue edges, and the only paths with exactly one red edge (red/blue paths) are red edges, by definition.

Maintenance: Each iteration gets the shortest blue-edges-only path going from  $i$  to  $j$  using intermediate vertices up through  $k$  accurately, due to the correctness of the Floyd-Warshall algorithm.

For the paths including exactly one red edge, for a given pair  $(i, j)$ , there are two cases: either the shortest red/blue path from  $i$  to  $j$  using intermediate vertices through  $k$  does not go through  $k$ , or it does. If it does not go through  $k$ , then the length of this path is equal to  $d_{r,ij}^{(k-1)}$ . If it does go through  $k$ , then the red edge on this path is either between  $i$  and  $k$  or between  $k$  and  $j$ . Because of the optimal substructure of shortest paths, we can therefore break this case down into two subcases: the shortest path length is equal to  $\min(d_{r,ik}^{(k-1)} + d_{b,kj}^{(k-1)}, d_{b,ik}^{(k-1)} + d_{r,kj}^{(k-1)})$ . Line 9 in the algorithm above finds the minimum path length of these three different possibilities, so this iteration must find the shortest path length from  $i$  to  $j$  going through exactly one red edge, using only intermediate vertices through  $k$ .

Termination: After  $n$  passes through the loop, all paths with intermediate vertices up to  $n$  have been included, and as there are only  $n$  vertices, shortest paths using all vertices as intermediates will be discovered.

**Problem 4. Telephone Psetting.** [12 points] (3 parts)

Upon realizing that it was 8:30 PM on Wednesday and he had not yet started his 6.046 pset, Ben Bitdiddle found  $n - 1$  other students (for a total of  $n$  students) in the same situation, and they decided to do the pset, which coincidentally had  $n$  problems in total, together.

Their brilliant plan for finishing the pset in time was to sit in a circle and assign one problem to each student, so that for  $i = 0, 1, \dots, n-1$ , student  $i$  did problem number  $i$ , and wrote up a solution for problem  $i$  meriting  $p(i)$  points. Then, they each copied the solutions to all the other problems from the student next to them, so that student  $i$  was copying from student  $i - 1$  (and student 0 was copying from student  $n - 1$ ).

Unfortunately, they were in such a hurry that the copying chain degraded the quality of the solutions: by the time student  $i$ 's solution to problem  $i$  reached student  $j$ , where  $d = j - i \pmod{n}$ ,  $d \in \{0, 1, \dots, n - 1\}$ , the solution was only worth  $\frac{1}{d+1}p(i)$  points.

- (a) Write a formula that describes the total pset score  $S(x)$  for student  $x$ , where the total pset score is the sum of the scores that student  $x$  got on each of the  $n$  problems.

**Solution:** [3 points]  $S(x) = \sum_{i=0}^{n-1} \frac{1}{((x-i) \pmod{n}) + 1} \cdot p(i)$

Alternatively, it is also correct to give the equivalent sum:

$$S(x) = \sum_{i=0}^{n-1} \frac{1}{i+1} \cdot p((x-i) \pmod{n})$$

- (b) Describe a simple  $O(n^2)$  algorithm to calculate the pset scores of all the students.

**Solution:** [3 points] Use the formula given in part (a) to calculate each student's score. Because calculating the score requires summing  $n$  numbers, it takes  $O(n)$  time to calculate a single score, and therefore  $O(n^2)$  time to calculate all the scores.

- (c) Describe a  $O(n \log n)$  algorithm to calculate the pset scores of all the students.

**Solution:** [6 points] The formula in the solution to part (a) describes a convolution: letting  $g(y) = \frac{1}{y+1}$ , the formula in part (a) can be written as  $S : \mathbb{Z}_n \rightarrow \mathbb{R}$ , where  $S(x) = \sum_{i \in \mathbb{Z}_n} p(i)g(x-i) = (p \otimes g)(x)$ . The algorithm is as follows:

1. Apply FFT on  $p$  and  $g$  to get  $\hat{p}$  and  $\hat{g}$  (time  $\Theta(n \log n)$ ).
2. Compute the transformed convolution  $\hat{S} = \hat{p} \cdot \hat{g}$  (time  $\Theta(n)$ ).
3. Apply the inverse FFT to get back the convolution  $S$  (time  $\Theta(n \log n)$ ).

Alternatively, define two polynomials:

$P_1(x) = \sum_{i=0}^{2n-1} p(i \bmod n)x^i$  and  $P_2(x) = \sum_{i=0}^{n-1} \frac{1}{i+1}x^i$ . Then student  $j$ 's pset score is equal to the coefficient of  $x^{n+j}$  in the product of the polynomials  $P_1$  and  $P_2$ . Because we are multiplying two polynomials of degree at most  $2n - 1$ , we can apply the polynomial multiplication method seen in class, which is outlined above: apply the FFT to get the evaluations of  $P_1$  and  $P_2$  on the roots of unity of order  $2n$ , pointwise multiply, and finally apply the inverse FFT to get the coefficients of the product.

## SCRATCH PAPER

## Practice Quiz 2

## Quiz 2

This take-home quiz contains 5 problems worth 25 points each, for a total of 125 points. Each problem should be answered on a separate sheet (or sheets) of 3-hole punched paper.

Mark the top of each problem with your name, 6.046J/18.410J, the problem number, your recitation time, and your TA. **Your exam is due between 9:00 and 11:00 A.M. on Friday, April 29, 2005, in the Stata lobby outside 32-123.** Late exams will not be accepted unless you obtain a Dean's Excuse or make prior arrangements with your recitation instructor. You must hand in your own exam in person.

The quiz should take you about 10 hours to do, but you have four days in which to do it. Plan your time wisely. Do not overwork, and get enough sleep. Ample partial credit will be given for good solutions, especially if they are well written. Of course, the better your asymptotic bounds, the higher your score. Bonus points will be given for exceptionally efficient or elegant solutions.

**Write-ups:** Each problem should be answered on a separate sheet (or sheets, stapled separately for each problem) of 3-hole punched paper. Mark the top of each problem with your name, 6.046J/18.410J, the problem number, your recitation time, and your TA. Your solution to a problem should start with a topic paragraph that provides an executive summary of your solution. This executive summary should describe the problem you are solving, the techniques you use to solve it, any important assumptions you make, and the running time your algorithm achieves.

Write up your solutions cleanly and concisely to maximize the chance that we understand them. Be explicit about running time and algorithms. For example, don't just say you *sort n* numbers, state that you are using heapsort, which sorts the  $n$  numbers in  $O(n \lg n)$  time in the worst case. When describing an algorithm, give an English description of the main idea of the algorithm. Use pseudocode only if necessary to clarify your solution. Give examples, and draw figures. Provide succinct and convincing arguments for the correctness of your solutions. Do not regurgitate material presented in class. Cite algorithms and theorems from CLRS, lecture, and recitation to simplify your solutions.

Part of the goal of this exam is to test engineering common sense. If you find that a question is unclear or ambiguous, make reasonable assumptions in order to solve the problem, and state clearly in your write-up what assumptions you have made. Be careful what you assume, however, because you will receive little credit if you make a strong assumption that renders a problem trivial.

**Bugs, etc.:** If you think that you've found a bug, send email to `6046-staff@theory.csail.mit.edu`. Corrections and clarifications will be sent to the class via email and posted on the class website. Check your email and the class website daily to avoid missing potentially important announcements. If you did not receive an email last Friday reminding you about Quiz 2, then you are not on the class email list and you should let your recitation instructor know immediately.

**Policy on academic honesty:** This quiz is “limited open book.” You may use your course notes, the CLRS textbook, basic reference materials such as dictionaries, and any of the handouts posted on the course web page, but *no other sources whatsoever may be consulted*. For example, you may not use notes or solutions from other times that this course or other related courses have been taught, or materials on the World-Wide Web. (These materials will not help you, but you may not use them anyhow.) Of prime importance, you may not communicate with any person except members of the 6.046 staff about any aspect of the exam until after noon on Friday, April 29, even if you have already handed in your exam.

If at any time you feel that you may have violated this policy, it is imperative that you contact the course staff immediately. If you have any questions about what resources may or may not be used during the quiz, send email to [6046-staff@theory.csail.mit.edu](mailto:6046-staff@theory.csail.mit.edu).

**Survey:** Attached to this exam is a survey on your experiences with the exam, especially as they relate to academic honesty. Please detach the survey, fill it out, and hand it in when you hand in your exam. Responses to the survey will be anonymous. No attempt will be made to associate a response with a person. This information will be used to gauge the usefulness of the exam, and summary statistics will be provided to the class.

**PLEASE REREAD THESE INSTRUCTIONS ONCE A DAY DURING THE EXAM.  
GOOD LUCK, AND HAVE FUN!**

### Problem 1. Static Graph Representation

Let  $G = (V, E)$  be a sparse undirected graph, where  $V = \{1, 2, \dots, n\}$ . For a vertex  $v \in V$  and for  $i = 1, 2, \dots$ , out-degree( $v$ ), define  $v$ 's ***ith neighbor*** to be the  $i$ th smallest vertex  $u$  such that  $(v, u) \in E$ , that is, if you sort the vertices adjacent to  $v$ , then  $u$  is the  $i$ th smallest.

Construct a representation of the graph  $G$  to support the following queries:

- **DEGREE( $v$ )**: returns the degree of vertex  $v$ .
- **LINKED( $u, v$ )**: output TRUE if an edge connects vertices  $u$  and  $v$ , and FALSE otherwise.
- **NEIGHBOR( $v, i$ )**: returns  $v$ 's  $i$ th neighbor.

Your data structure should use asymptotically as little space as possible, and the operations should run asymptotically as fast as possible, but space is more important than time. Analyze your data structure in terms of both space and time.

### Problem 2. Video Game Design

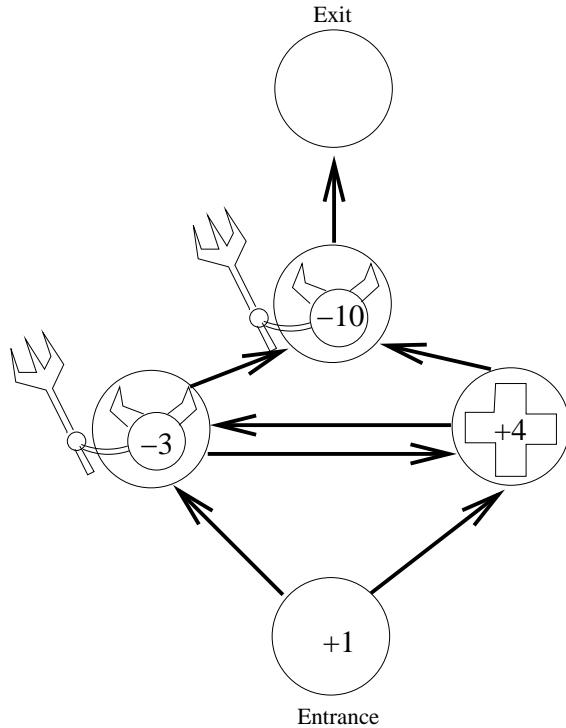
Professor Cloud has been consulting in the design of the most anticipated game of the year: ***Take-home Fantasy***. One of the levels in the game is a maze that players must navigate through multiple rooms from an entrance to an exit. Each room can be empty, contain a monster, or contain a life potion. As the player wanders through the maze, points are added or subtracted from her ***life points***  $L$ . Drinking a life potion increases  $L$ , but battling a monster decreases  $L$ . If  $L$  drops to 0 or below, the player dies.

As shown in Figure 1, the maze can be represented as a digraph  $G = (V, E)$ , where vertices correspond to rooms and edges correspond to (one-way) corridors running from room to room. A vertex-weight function  $f : V \rightarrow \mathbb{Z}$  represents the room contents:

- If  $f(v) = 0$ , the room is empty.
- If  $f(v) > 0$ , the room contains a life potion. Every time the player enters the room, her life points  $L$  increase by  $f(v)$ .
- If  $f(v) < 0$ , the room contains a monster. Every time the player enters the room, her life points  $L$  drop by  $|f(v)|$ , killing her if  $L$  becomes nonpositive.

The ***entrance*** to the maze is a designated room  $s \in V$ , and the ***exit*** is another room  $t \in V$ . Assume that a path exists from  $s$  to every vertex  $v \in V$ , and that a path exists from every vertex  $v \in V$  to  $t$ . The player starts at the entrance with  $L = L_0$  life points, i.e.  $L_0$  is the value of  $f$  for the entrance. The figure shows  $L_0 = 1$ .

Professor Cloud has designed a program to put monsters and life potions randomly into the maze, but some mazes may be impossible to safely navigate from entrance to exit unless the player enters with a sufficient number  $L_0 > 0$  of life points. A path from  $s$  to  $t$  is “safe” if the player stays alive along the way, i.e., her life-points never become non-positive. Define a maze to be ***r-admissible*** if a safe path through the maze exists when the player begins with  $L_0 = r$ .



**Figure 1:** An example of a 1-admissible maze.

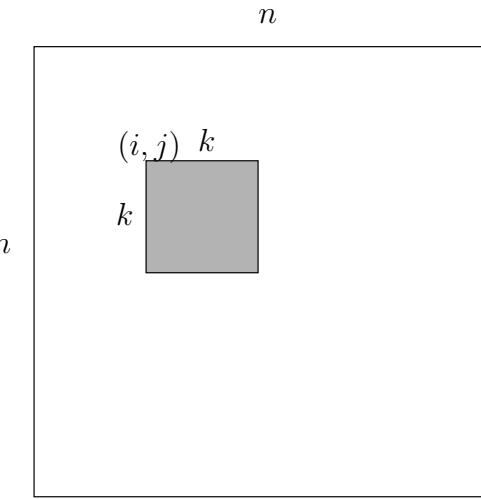
Help the professor by designing an efficient algorithm to determine the minimum value  $r$  so that a given maze is  $r$ -admissible, or determine that no such  $r$  exists. (For partial credit, solve the problem determining whether a maze is  $r$ -admissible for a given  $r$ .)

### Problem 3. *Image Filtering*

**Two-dimensional filtering** is a common operation in vision and image processing. An image is represented as an  $n \times n$  matrix of real values. As shown in Figure 2, the idea is to pass a  $k \times k$  window across the matrix, and for each of the possible placements of the window, the filter computes the “product” of all the values in the window. The “product” is not typically ordinary multiplication, however. For this problem, we shall assume it is an associative and commutative binary operation  $\otimes$  with identity element  $e$ , that is,  $x \otimes e = e \otimes x = x$ . For example, the product could be  $+$  with identity element  $0$ ,  $\times$  with  $1$ ,  $\min$  with  $\infty$ , etc. Importantly, you may not assume that  $\otimes$  has an inverse operation, such as  $-$  for  $+$ .

To be precise, given an  $n \times n$  image

$$A = \begin{pmatrix} a_{00} & a_{01} & \dots & a_{0(n-1)} \\ a_{10} & a_{11} & \dots & a_{1(n-1)} \\ \vdots & \vdots & \ddots & \vdots \\ a_{(n-1)0} & a_{(n-1)1} & \dots & a_{(n-1)(n-1)} \end{pmatrix},$$



**Figure 2:** The output element  $b_{ij}$  is the “product” of all the  $a$ ’s in the shaded square.

the  $(k \times k)$ -*filtered* image is the  $n \times n$  matrix

$$B = \begin{pmatrix} b_{00} & b_{01} & \dots & b_{0(n-1)} \\ b_{10} & b_{11} & \dots & b_{1(n-1)} \\ \vdots & \vdots & \ddots & \vdots \\ b_{(n-1)0} & b_{(n-1)1} & \dots & b_{(n-1)(n-1)} \end{pmatrix},$$

where for  $i, j = 0, 1, \dots, n - 1$ ,

$$b_{ij} = \bigotimes_{x=i}^{i+k-1} \bigotimes_{y=j}^{j+k-1} a_{xy}.$$

(For convenience, if  $x \geq n$  or  $y \geq n$ , we assume that  $a_{xy} = e$ .)

Give an efficient algorithm to compute the  $(k \times k)$ -filter of an input matrix  $A$ . While analyzing your algorithm, do not treat  $k$  as a constant. That is, express your running time in terms of  $n$  and  $k$ . (For partial credit, solve the problem in one dimension.)

#### Problem 4. ViTo Design

You are designing a new-and-improved digital video recorder, called ViTo. In the ViTo software, a television show  $i$  is represented as a triple: a **channel number**  $c_i$ , a **start time**  $s_i$ , and an **end time**  $e_i$ . The ViTo owner inputs a list of  $n$  shows to watch and for each show  $i = 1, 2, \dots, n$ , assigns it a **pleasure rating**  $r_i$ . Since shows may overlap, and the ViTo can record only one show at a time, the ViTo should record the subset of the shows that maximize the aggregate “pleasure.” Since the owner receives no pleasure from watching only part of a show, the ViTo never records partial shows. Design an efficient algorithm for the ViTo to select the best subset of shows to record.

**Problem 5. *Growing a Graph***

We wish to build a data structure that supports a dynamically growing directed graph  $G = (V, E)$ . Initially, we have  $V = \{1, 2, \dots, n\}$  and  $E = \emptyset$ . The user grows the graph with the following operation:

- INSERT-EDGE( $u, v$ )**: Insert a directed edge from vertex  $u$  to vertex  $v$ , that is,  $E \leftarrow E \cup \{(u, v)\}$ .

In addition, at any time the user can query the graph for whether two vertices are connected:

- CHECK-PATH( $u, v$ )**: Return TRUE if a directed path from vertex  $u$  to vertex  $v$  exists; otherwise, return FALSE.

The user grows the graph until it is fully connected. Since the number of edges increases monotonically and the user never inserts the same edge twice, the total number of INSERT-EDGE operations is exactly  $n(n - 1)$ . During the time that the graph is growing, the user performs  $m$  CHECK-PATH operations which are intermixed with the  $n(n - 1)$  INSERT-EDGE's. Design a data structure that can efficiently support any such sequence of operations.

## Quiz 2 Solutions

### Problem 1. Static Graph Representation

Let  $G = (V, E)$  be a sparse undirected graph, where  $V = \{1, 2, \dots, n\}$ . For a vertex  $v \in V$  and for  $i = 1, 2, \dots$ , out-degree( $v$ ), define  $v$ 's ***i*th neighbor** to be the  $i$ th smallest vertex  $u$  such that  $(v, u) \in E$ , that is, if you sort the vertices adjacent to  $v$ , then  $u$  is the  $i$ th smallest.

Construct a representation of the graph  $G$  to support the following queries:

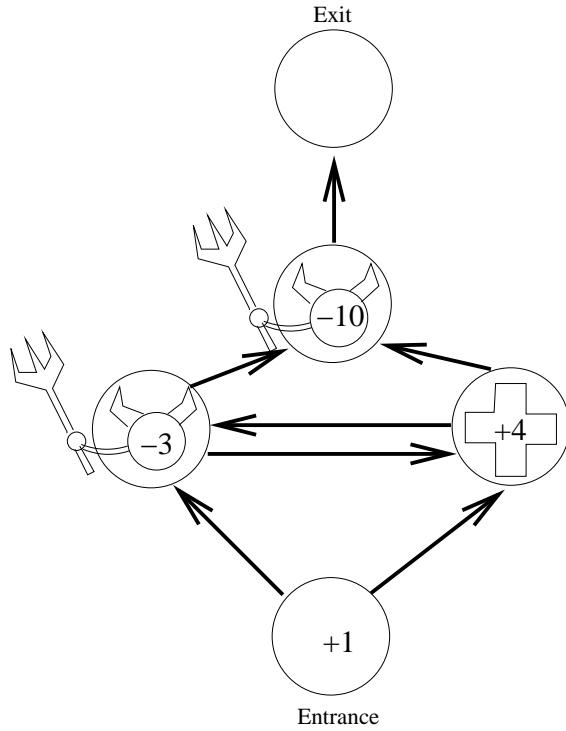
- $\text{DEGREE}(v)$ : returns the degree of vertex  $v$ .
- $\text{LINKED}(u, v)$ : output TRUE if an edge connects vertices  $u$  and  $v$ , and FALSE otherwise.
- $\text{NEIGHBOR}(v, i)$ : returns  $v$ 's  $i$ th neighbor.

Your data structure should use asymptotically as little space as possible, and the operations should run asymptotically as fast as possible, but space is more important than time. Analyze your data structure in terms of both space and time.

**Solution:** We give a solution that uses  $\Theta(E)$  space for the data structure and takes  $\Theta(1)$  time for each of the three operations. Create a hash table that contains key  $(u, v)$  if  $(u, v) \in E$  or key  $(u, i)$  if vertex  $u$  has an  $i$ -th neighbor. It is possible that for some  $u$  and  $v = i$ , vertex  $u$  has both an adjacent vertex  $v$  and an  $i$ -th neighbor. This is handled by storing satellite data with each record of the hash table. For the record with key  $(u, i)$  in the hash table, if  $u$  has a neighbor  $v = i$ , then indicate so using a bit in the record; if  $u$  has an  $i$ -th neighbor, then store the corresponding neighbor vertex index in the record. Also, for every vertex  $u$  that is connected to some other vertex, store its degree in the record for key  $(u, 1)$ .

Thus, for each vertex  $u$  in the first coordinate of the key, the hash table has at most  $\text{degree}(u) + \text{degree}(u) = 2 \text{degree}(u)$  entries. The total number of entries is thus at most  $\sum_{u=1}^n 2 \text{degree}(u) = 4|E|$ . By using *perfect hashing*, and choosing a suitable hash function through a small number of random samplings (during data structure construction), we can make the lookup time  $\Theta(1)$  and space requirement linear in the number of entries stored, i.e.,  $\Theta(E)$  (see CLRS, page 249, Corollary 11.12). We can use the same family of hash functions as in CLRS by converting each 2-tuple  $(u, v)$  into a distinct number  $(u - 1)n + v$  in the range of  $[1 \dots n^2]$ . The total space requirement of the degree array and hash table is thus  $\Theta(V + E)$ .

Then,  $\text{DEGREE}(v)$  looks up the key  $(v, 1)$  in the hash table. If found, it returns the degree value stored in the record. Otherwise, it returns 0, since  $v$  is not adjacent to any vertex. This takes  $\Theta(1)$  time.  $\text{LINKED}(u, v)$  looks up the key  $(u, v)$  in the hash table and returns TRUE if it exists and the associated bit in the record is set. This is  $\Theta(1)$  time.  $\text{NEIGHBOR}(v, i)$  looks up the key  $(v, i)$  in the hash table – if it exists and a neighbor vertex is stored in the record, it returns its index. This is also  $\Theta(1)$  time.



**Figure 1:** An example of a 1-admissible maze.

### Problem 2. Video Game Design

Professor Cloud has been consulting in the design of the most anticipated game of the year: *Take-home Fantasy*. One of the levels in the game is a maze that players must navigate through multiple rooms from an entrance to an exit. Each room can be empty, contain a monster, or contain a life potion. As the player wanders through the maze, points are added or subtracted from her **life points**  $L$ . Drinking a life potion increases  $L$ , but battling a monster decreases  $L$ . If  $L$  drops to 0 or below, the player dies.

As shown in Figure 1, the maze can be represented as a digraph  $G = (V, E)$ , where vertices correspond to rooms and edges correspond to (one-way) corridors running from room to room. A vertex-weight function  $f : V \rightarrow \mathbb{Z}$  represents the room contents:

- If  $f(v) = 0$ , the room is empty.
- If  $f(v) > 0$ , the room contains a life potion. Every time the player enters the room, her life points  $L$  increase by  $f(v)$ .
- If  $f(v) < 0$ , the room contains a monster. Every time the player enters the room, her life points  $L$  drop by  $|f(v)|$ , killing her if  $L$  becomes nonpositive.

The **entrance** to the maze is a designated room  $s \in V$ , and the **exit** is another room  $t \in V$ . Assume that a path exists from  $s$  to every vertex  $v \in V$ , and that a path exists from every vertex  $v \in V$  to  $t$ .

The player starts at the entrance with  $L = L_0$  life points, i.e.  $L_0$  is the value of  $f$  for the entrance. The figure shows  $L_0 = 1$ .

Professor Cloud has designed a program to put monsters and life potions randomly into the maze, but some mazes may be impossible to safely navigate from entrance to exit unless the player enters with a sufficient number  $L_0 > 0$  of life points. A path from  $s$  to  $t$  is “safe” if the player stays alive along the way, i.e., her life-points never become non-positive. Define a maze to be ***r-admissible*** if a safe path through the maze exists when the player begins with  $L_0 = r$ .

Help the professor by designing an efficient algorithm to determine the minimum value  $r$  so that a given maze is  $r$ -admissible, or determine that no such  $r$  exists. (For partial credit, solve the problem determining whether a maze is  $r$ -admissible for a given  $r$ .)

### Solution:

We give an algorithm with running time  $O(VE \lg r)$ , where  $r$  is the minimum life needed at entry for the maze to be admissible. The problem is solved in two parts.

- An algorithm to determine if a given maze is admissible for a given  $r$ : This is done by using a modified version of Bellman-Ford. For every node  $u$ , this algorithm computes the maximum number of points  $q[u]$  that the player can have on reaching  $u$ . Since the player will die if she reaches  $u$  with negative points, the value of  $q[u]$  is either  $-\infty$  (denoting that the player cannot reach  $u$ ) or positive. Thus if  $q[t]$  is positive ( $t$  is the exit node), then the graph is  $r$ -admissible.
- We know that the minimum  $r$  cannot be less than 1. Thus we use a combination of exponential and binary search to find the minimum value of  $r$ . We use the modified Bellman-Ford  $\log r$  times to find the minimum  $r$ .

The running time of the algorithm is  $O(VE \log r)$ , where  $r$  is the minimum  $r$ , where the maze is  $r$ -admissible.

**Determining Admissibility for a given  $r$**  We use a modified version of Bellman-Ford algorithm. Given an  $r$ , for every node  $u$  we find the maximum (positive) number of points  $q[u]$  the player can have when she reaches  $u$ . If  $q[t]$  is positive, then the graph is  $r$ -admissible.

For each vertex  $u \in V$ , we maintain  $p[u]$  which is a lower bound on  $q[u]$ . We initialize all the  $p[u]$ 's to  $-\infty$ , except the entrance, which is initialized to  $r$ . As we run the Bellman-Ford Algorithm and relax edges, the value of  $p[u]$  increases until it converges to  $q[u]$  (if there are no positive weight cycles). The important point to note is that reaching a node with negative points is as good as not reaching it at all. Thus, we modify  $p[u]$  only if it becomes positive, otherwise  $p[u]$  remains  $-\infty$ . We change the relaxation routine to incorporate this as follows.

```

V-RELAX( $u, v$ )
1 if  $(u, v) \in E$ 
2   then if  $((p[v] < p[u] + f[v]) \text{ and } (p[u] + f[v] > 0))$ 
3     then  $p[v] \leftarrow p[u] + f[v]$ 
4    $\pi[v] \leftarrow u$ 

```

After all the edges have been relaxed  $V$  times, if there are no negative weight cycles, all  $p[u]$ 's will have converged to the corresponding  $q[u]$ 's (the maximum number of points you can have on reaching vertex  $u$ ). If  $q[t]$  is positive at this point, then the player can reach there with positive life points and thus the graph is  $r$ -admissible. If  $p[t]$  is not positive, however, we relax all the edges one more time (just like Bellman-Ford). If  $p[u]$  of any node changes, we have found a positive weight cycle which is reachable from  $s$  starting with  $r$  points. Thus the player can go around the cycle enough times to collect all the necessary points to reach  $t$  and thus the graph is  $r$ -admissible. If we don't find a reachable positive weight cycle and  $p[t]$  is  $-\infty$ , then the graph is not  $r$  admissible. The correctness of the algorithm follows from the correctness of Bellman-Ford, and the running time is  $O(VE)$ .

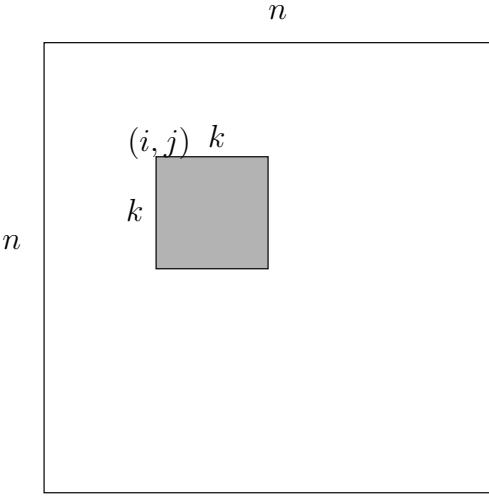
**Finding the minimum  $r$  for which the graph is  $r$ -admissible** Given the above sub-routine, we now find the minimum  $r$ . We first check if the graph is 1-admissible. If it is, we return 1 as the answer. If it is not, then we check if it is 2-admissible and then 4-admissible and so on. Thus on the  $i$ th step we check if the graph is  $2^{i-1}$ -admissible. Eventually, we find  $k$  such that the graph is not  $2^{k-1}$ -admissible, but it is  $2^k$ -admissible. Thus the minimum value of  $r$  lies between these two values. Then we binary search between  $r = 2^{k-1}$  and  $r = 2^k$  to find the right value of  $r$ .

Analysis: The number of iterations is  $k + O(\lg r) = O(\lg r)$ , since  $k = \lfloor \lg r \rfloor$ . Thus you have to run Bellman-Ford  $O(\lg r)$  times, and the total running time is  $O(VE \lg r)$ .

**Alternate Solutions** Some people visited nodes in DFS or BFS order starting from the exit, relaxing edges to find the minimum number of points needed to get from any node  $u$  to the exit. The problem with this approach is that in the presence of positive weight cycles, the algorithm runs for  $O(M(V + E))$  time, where  $M$  is the total sum of all monster points. This number can be big even if the real  $r$  is small. Some people did the same thing, except with Bellman-Ford instead of search, which gives a running time of  $O(MVE)$ . There were a couple of other clever solutions which ran in  $O(V^2E)$  time.

### Problem 3. *Image Filtering*

**Two-dimensional filtering** is a common operation in vision and image processing. An image is represented as an  $n \times n$  matrix of real values. As shown in Figure 2, the idea is to pass a  $k \times k$  window across the matrix, and for each of the possible placements of the window, the filter computes the ‘product’ of all the values in the window. The ‘product’ is not typically ordinary



**Figure 2:** The output element  $b_{ij}$  is the ‘product’ of all the  $a$ ’s in the shaded square.

multiplication, however. For this problem, we shall assume it is an associative and commutative binary operation  $\otimes$  with identity element  $e$ , that is,  $x \otimes e = e \otimes x = x$ . For example, the product could be  $+$  with identity element  $0$ ,  $\times$  with  $1$ ,  $\min$  with  $\infty$ , etc. Importantly, you may not assume that  $\otimes$  has an inverse operation, such as  $-$  for  $+$ .

To be precise, given an  $n \times n$  image

$$A = \begin{pmatrix} a_{00} & a_{01} & \cdots & a_{0(n-1)} \\ a_{10} & a_{11} & \cdots & a_{1(n-1)} \\ \vdots & \vdots & \ddots & \vdots \\ a_{(n-1)0} & a_{(n-1)1} & \cdots & a_{(n-1)(n-1)} \end{pmatrix},$$

the  **$(k \times k)$ -filtered** image is the  $n \times n$  matrix

$$B = \begin{pmatrix} b_{00} & b_{01} & \cdots & b_{0(n-1)} \\ b_{10} & b_{11} & \cdots & b_{1(n-1)} \\ \vdots & \vdots & \ddots & \vdots \\ b_{(n-1)0} & b_{(n-1)1} & \cdots & b_{(n-1)(n-1)} \end{pmatrix},$$

where for  $i, j = 0, 1, \dots, n - 1$ ,

$$b_{ij} = \bigotimes_{x=i}^{i+k-1} \bigotimes_{y=j}^{j+k-1} a_{xy}.$$

(For convenience, if  $x \geq n$  or  $y \geq n$ , we assume that  $a_{xy} = e$ .)

Give an efficient algorithm to compute the  $(k \times k)$ -filter of an input matrix  $A$ . While analyzing your algorithm, do not treat  $k$  as a constant. That is, express your running time in terms of  $n$  and  $k$ . (For partial credit, solve the problem in one dimension.)

**Solution:**

We can solve the two-dimensional filtering problem in  $\Theta(n^2)$  time by first reducing the problem to two one-dimensional filtering problems and then showing how a one-dimensional filter on  $n$  elements can be solved in  $\Theta(n)$  time. We assume that  $k \leq n$ , since filtered values for  $k > n$  are the same as for  $k = n$ . The  $\Theta(n^2)$ -time algorithm is optimal, since there are  $n^2$  values to compute.

Define the intermediate matrix  $C$  by

$$C = \begin{pmatrix} c_{00} & c_{01} & \cdots & c_{0(n-1)} \\ c_{10} & c_{11} & \cdots & c_{1(n-1)} \\ \vdots & \vdots & \ddots & \vdots \\ c_{(n-1)0} & c_{(n-1)1} & \cdots & c_{(n-1)(n-1)} \end{pmatrix},$$

where for  $i, j = 0, 1, \dots, n-1$ ,

$$c_{ij} = \bigotimes_{y=j}^{j+k-1} a_{iy},$$

that is,  $C$  is the one-dimensional  $k$ -filter on each row of  $A$ . We have

$$\begin{aligned} b_{ij} &= \bigotimes_{x=i}^{i+k-1} \bigotimes_{y=j}^{j+k-1} a_{xy} \\ &= \bigotimes_{x=i}^{i+k-1} c_{xj}, \end{aligned}$$

and thus  $B$  is just the one-dimensional  $k$ -filter on each column of  $C$ .

It remains to devise an efficient method to compute one-dimensional  $k$ -filters. The naive algorithm takes  $\Theta(kn)$  time to solve the one-dimensional problem for an array of length  $n$ . Using this one-dimensional algorithm to solve the two-dimensional problem costs  $\Theta(kn^2)$  to compute  $C$  from  $A$  and another  $\Theta(kn^2)$  to compute  $B$  from  $C$ , resulting in  $\Theta(kn^2)$  overall. Many students found a way to compute the one-dimensional problem in  $\Theta(n \lg k)$ , resulting in a two-dimensional solution of  $\Theta(n^2 \lg k)$ . In fact, as some students discovered, the one-dimensional problem can be solved in  $\Theta(n)$  time, leading to a two-dimensional solution of  $\Theta(n^2)$ .

The  $\Theta(n)$ -time solution for the one-dimensional problem works as follows. Let the input array be  $A = \langle a_0, a_1, \dots, a_{n-1} \rangle$  and the  $k$ -filtered output array be  $B = \langle b_0, b_1, \dots, b_{n-1} \rangle$ , where

$$b_i = \bigotimes_{x=i}^{i+k-1} a_x.$$

Assume without loss of generality that  $n$  is evenly divisible by  $k$ , since otherwise, we can pad the end of  $a$  with identity elements  $e$  to make  $n$  a multiple of  $k$  without more than doubling  $n$ .

The idea is to divide the arrays into blocks of  $k$  elements. Observe that any window of  $k$  elements starting at a given location  $i$  consists of the product of a suffix of one block and a prefix of the next

block. Thus, we compute prefixes and suffixes of each block as follows. For  $i = 0, 1, \dots, n - 1$ , define

$$f_i = \begin{cases} e & \text{if } i \bmod k = 0, \\ f_{i-1} \otimes a_{i-1} & \text{otherwise;} \end{cases}$$

and for  $i = n - 1, n - 2, \dots, 0$ , define

$$g_i = \begin{cases} a_i & \text{if } (i+1) \bmod k = 0, \\ a_i \otimes f_{i+1} & \text{otherwise.} \end{cases}$$

These two arrays can be computed in  $\Theta(n)$  time, and then we obtain the output array by computing, for  $i = 0, 1, \dots, n - 1$ ,

$$b_i = g_i \otimes f_{i+k},$$

which also takes  $\Theta(n)$  time.

As an example, consider a one-dimensional 4-filter:

$$\begin{aligned} b_0 &= (a_0 \otimes a_1 \otimes a_2 \otimes a_3) &= g_0 \otimes f_4 \\ b_1 &= (a_1 \otimes a_2 \otimes a_3) \otimes (a_4) &= g_1 \otimes f_5 \\ b_2 &= (a_2 \otimes a_3) \otimes (a_4 \otimes a_5) &= g_2 \otimes f_6 \\ b_3 &= (a_3) \otimes (a_4 \otimes a_5 \otimes a_6) &= g_3 \otimes f_7 \\ b_4 &= (a_4 \otimes a_5 \otimes a_6 \otimes a_7) &= g_4 \otimes f_8 \\ b_5 &= (a_5 \otimes a_6 \otimes a_7) \otimes (a_8) &= g_5 \otimes f_9 \\ b_6 &= (a_6 \otimes a_7) \otimes (a_8 \otimes a_9) &= g_6 \otimes f_{10} \\ &\vdots \end{aligned}$$

#### Problem 4. ViTo Design

You are designing a new-and-improved digital video recorder, called ViTo. In the ViTo software, a television show  $i$  is represented as a triple: a **channel number**  $c_i$ , a **start time**  $s_i$ , and an **end time**  $e_i$ . The ViTo owner inputs a list of  $n$  shows to watch and for each show  $i = 1, 2, \dots, n$ , assigns it a **pleasure rating**  $r_i$ . Since shows may overlap, and the ViTo can record only one show at a time, the ViTo should record the subset of the shows that maximize the aggregate ‘pleasure.’ Since the owner receives no pleasure from watching only part of a show, the ViTo never records partial shows. Design an efficient algorithm for the ViTo to select the best subset of shows to record.

**Solution:** Assume ViTo has enough harddisk space to record any subset of the programs. We use a dynamic programming approach to the problem.

First let’s show the optimal substructure. Let  $show_i$  denote the triple  $(c_i, s_i, e_i)$ ,  $shows$  denote the whole list of shows  $show_1, \dots, show_n$ , and  $shows(t)$  denote the subset of shows  $show_j$  such that  $e_j < t$ , i.e. all the shows with ending times before time  $t$ . Consider an optimal solution

$show_{i_1}, show_{i_2}, \dots, show_{i_k}$ . Then  $show_{i_1}, show_{i_2}, \dots, show_{i_{k-1}}$  must be an optimal solution to the subproblem  $shows(s_{i_k})$  since if not, we could cut and paste a better solution to this subproblem, append  $show_{i_k}$  to it, and get a better solution than the optimal one.

Sort the shows by ending time, so  $e_i \leq e_j$  for  $i < j$  and if  $e_i = e_j$  then  $s_i < s_j$ . (if two or more shows start and end at the same time, then we can just keep the one with the maximum pleasure  $r_i$ , breaking ties arbitrarily). This can be done with counting sort in linear time: with  $n$  shows, there are only a maximum of  $2n$  possible starting/ending times, and there are only 24 hours in a day, i.e. a limited (constant) amount of time, therefore the range of possible times is  $O(n)$ . Relabel the shows so that  $show_1, \dots, show_n$  are in sorted order. Let  $shows^i$  be the list of shows up to and including the  $i$ -th show, i.e.  $shows^i = show_1, show_2, \dots, show_i$ .

Note that  $shows(t)$ , the subset of shows  $show_j$  such that  $e_j < t$ , is equal to  $show^{k(t)}$  for some  $k(t)$ . The aggregate pleasure  $p(i)$  of an optimal solution for shows in  $show^i$  is:

$$p(i) = \begin{cases} 0 & \text{if } i < 1; \\ \max\{p(k(s_i)) + r_i, p(i-1)\} & \text{otherwise.} \end{cases}$$

The optimal solution is:

$$record(i) = \begin{cases} \{\} & \text{if } i < 1; \\ record(k(s_i)) \cup \{show_i\} & \text{if } p(k(s_i)) + r_i > p(i-1); \\ record(i-1) & \text{otherwise.} \end{cases}$$

Running Time = time to sort the shows + time to find  $p(n) = O(n)$ .

### Problem 5. Growing a Graph

We wish to build a data structure that supports a dynamically growing directed graph  $G = (V, E)$ . Initially, we have  $V = \{1, 2, \dots, n\}$  and  $E = \emptyset$ . The user grows the graph with the following operation:

- **INSERT-EDGE( $u, v$ )**: Insert a directed edge from vertex  $u$  to vertex  $v$ , that is,  $E \leftarrow E \cup \{(u, v)\}$ .

In addition, at any time the user can query the graph for whether two vertices are connected:

- **CHECK-PATH( $u, v$ )**: Return TRUE if a directed path from vertex  $u$  to vertex  $v$  exists; otherwise, return FALSE.

The user grows the graph until it is fully connected. Since the number of edges increases monotonically and the user never inserts the same edge twice, the total number of INSERT-EDGE operations is exactly  $n(n - 1)$ . During the time that the graph is growing, the user performs  $m$  CHECK-PATH operations which are intermixed with the  $n(n - 1)$  INSERT-EDGE's. Design a data structure that can efficiently support any such sequence of operations.

**Solution:** To solve this problem, we keep an  $n \times n$  **transitive-closure** matrix  $T$  that keeps track of whether there exists a directed path between each pair of vertices. We give an algorithm such that each CHECK-PATH operation takes  $O(1)$  time, and a sequence of  $n(n - 1)$  INSERT-EDGE operations take a total of  $O(n^3)$  time in the worst case. Combining these bounds, any sequence of  $m$  CHECK-PATH and  $n(n - 1)$  INSERT-EDGE operations takes a total of  $O(n^3 + m)$  time. We later improve the data structure to deal with the case in which  $m$  is small, to get a total time of  $O(\min\{n^3 + m, n^2m\})$ .

Our data structure maintains a transitive-closure matrix  $T = (t_{uv})$  such that

$$t_{uv} = \begin{cases} 1 & : \text{ if there exists a directed path from } u \text{ to } v \text{ in } G , \\ 0 & : \text{ otherwise .} \end{cases}$$

The matrix  $T$  is similarly to an adjacency matrix, except that instead of keeping track of the existence of edges  $u \rightarrow v$ , it keeps track of paths  $u \rightsquigarrow v$ . Note that the 1's in the  $u$ -th row correspond to all the vertices that  $u$  can reach, and the 1's in the  $u$ -th column correspond to all the vertices that can reach  $u$ . We initialize  $t_{uu} = 1$  because there is a directed path (of no edges) from a vertex to itself.

Given  $T$ , the implementation of CHECK-PATH( $u, v$ ) is straightforward: just query the value of  $t_{uv}$ . This query can be performed in constant time, so CHECK-PATH runs in constant time. Pseudocode for CHECK-PATH is given below:

```
CHECK-PATH( $u, v$ )
1 if  $t_{uv} = 1$ 
2   then return TRUE
3   else return FALSE
```

The tricky part of the data structure is maintaining the matrix  $T$  on an INSERT-EDGE( $u, v$ ). When the edge  $(u, v)$  is added, we check each vertex  $x$ . If  $x$  can reach  $u$ , and  $x$  cannot already reach  $v$ , then we update the matrix to indicate that  $u$  can reach all the vertices that  $v$  can reach (in addition to the vertices that it could reach before). In other words, let  $R_w$  be the set of vertices that the vertex  $w$  can reach (i.e., the set of indices of 1's in the  $w$ -th row in  $T$ ). Then when adding  $(u, v)$ , we iterate over all  $x \in V$ . For each  $x$  such that  $u \in R_x$  and  $v \notin R_x$ , we set  $R_x \leftarrow R_x \cup R_v$ . Pseudocode for INSERT-EDGE is given below:

```
INSERT-EDGE( $u, v$ )
1 for  $x \leftarrow 1$  to  $n$ 
2   do if  $t_{xu} = 1$  and  $t_{xv} = 0$   $\triangleright x$  can reach  $u$  but not  $v$ 
3     then for  $y \leftarrow 1$  to  $n$ 
4       do  $t_{xy} \leftarrow \max\{t_{xy}, t_{vy}\}$   $\triangleright$  If  $v \rightsquigarrow y$ , add  $x \rightsquigarrow y$  to  $T$ 
```

**Correctness.** The following theorem proves that our algorithm is correct.

**Theorem 1** *The INSERT-EDGE operation maintains the invariant that  $t_{xy} = 1$  iff there exists a directed path from  $x$  to  $y$  in  $G$ .*

*Proof.* We prove by induction on INSERT-EDGE operations. That is, we assume that the transitive-closure matrix is correct up to (before) a particular  $\text{INSERT-EDGE}(u, v)$  operation, and then we show that it is correct after that operation. We do not have to prove anything for CHECK-PATH as that operation does not modify the matrix.

First, suppose that  $x \rightsquigarrow y$  before the edge  $(u, v)$  is added. Then  $t_{xy} = 1$  before the INSERT-EDGE operation. The only place  $t_{xy}$  can be updated is in line 4, and if so, it keeps its value of 1. This behavior is correct because adding edges cannot destroy a path.

Suppose that  $x \not\rightsquigarrow y$  before the edge  $(u, v)$  is added, but  $x \rightsquigarrow y$  after the edge is added. Therefore, it must be the case the path from  $x$  to  $y$  uses the edge  $(u, v)$ . Therefore, we have  $x \rightsquigarrow u$  and  $v \rightsquigarrow y$  before the  $\text{INSERT-EDGE}(u, v)$ , so by assumption  $t_{xu} = 1$  and  $t_{vy} = 1$ . Furthermore, it must also be true that  $x \not\rightsquigarrow v$  before the addition of  $(u, v)$ , or we would violate the assumption that  $x \not\rightsquigarrow y$ . Thus, we reach line 4, and  $t_{xy} \leftarrow t_{vy} = 1$ .

The last case to consider is the one in which  $x \not\rightsquigarrow y$  after the operation. We need to make sure that we have  $t_{xy} = 0$  in this case. If there is no path, then  $t_{xy} = 0$  before the addition of  $(u, v)$ . Moreover, there is no path that uses  $(u, v)$ , so either  $t_{xu} = 0$  or  $t_{vy} = 0$ . If  $t_{xu} = 0$ , we don't enter the loop in line 2, so the update in line 4 is not performed. If  $t_{xu} = 1$ , then  $t_{vy} = 0$ , and line 4 sets the value of  $t_{xy} \leftarrow 0$ .  $\square$

**Analysis.** Now let us examine the runtime of our algorithm. Each CHECK-PATH operation is just a table lookup, which takes  $O(1)$  time. The analysis of INSERT-EDGE is slightly more complicated. We can trivially bound the worst-case cost of INSERT-EDGE to  $O(n^2)$  because we have nested for loops, each iterating over  $n$  items and doing constant work in line 4. We can show a tighter bound on a sequence of  $n(n - 1)$  INSERT-EDGE operations using aggregate analysis. Each time INSERT-EDGE runs, the outer loop (line 1) executes, performing the constant work from line 2 on  $n$  items. Thus, the contribution of the outer loop totals to  $O(n^3)$ . The inner loop (line 3) executes only when  $t_{xv} = 0$ , and when it finishes,  $t_{xv} = 1$ . Thus, for a particular vertex  $x$ , the inner loop can be executed at most  $n$  times (actually,  $n - 1$ , as we begin with  $t_{xx} = 1$ ). Since there are  $n$  vertices, the inner loop can run at most  $n^2$  times in total for a total  $O(n^3)$  work in the worst case. Thus, the total runtime for  $n(n - 1)$  INSERT-EDGES and  $m$  CHECK-PATHs is  $O(n^3 + m)$ .

**Slight improvements.** There is another data structure with  $O(1)$  cost for each INSERT-EDGE but  $O(n^2)$  for each CHECK-PATH. To implement this data structure, we can use an adjacency list: we keep an array  $A[1..n]$  of size  $n$  indexed by vertex and keep a (linked) list of all the outgoing edges from the corresponding vertex. To perform an  $\text{INSERT-EDGE}(u, v)$ , simply insert  $v$  at the front of  $A[u]$  in  $O(1)$  time. (Note that edges are inserted only once, so we do not have to worry about  $v$  being present in the list already.) To perform  $\text{CHECK-PATH}(u, v)$ , we run some sort of search, let's say breadth-first search, starting at vertex  $u$ . If  $v$  is encountered at any point along the search,

return TRUE. If not, return FALSE. Correctness of this algorithm should be somewhat obvious. BFS takes  $O(V + E) = O(n^2)$  time. Thus, the total runtime of the sequence of operations is  $O(n^2 + n^2m)$ .

This data structure is probably worse than the one given above. It seems safe to assume that  $m \gg n$  as you probably query each vertex at least once. Assuming that  $m \gg n^2$  is also reasonable. If you do not want make these assumptions, and you know  $m$  ahead of time, you can choose the appropriate data structure.

It turns out that we can also combine both data structures to achieve the better of the two bounds even if  $m$  is not known ahead of time. To do this, we use the adjacency-list data structure until there have been  $n$  queries. Once we reach the  $n$ -th query (CHECK-PATH), we construct the transitive-closure matrix and then use the matrix for all subsequent operations. Construction of the matrix takes  $O(n^3)$  time by simply running BFS from each vertex  $u$  and marking each reachable vertex  $v$  by  $t_{uv} \leftarrow 1$ . Thus, if  $m \leq n$ , we use only the adjacently list, to get a total runtime of  $O(n^2m)$ . If  $m \geq n$ , we first use the adjacency list for a total of  $O(n^3)$  work, then we transform to the transitive-closure matrix in  $O(n^3)$  time, then we use the matrix for all subsequent operations, which comes to a total of  $O(n^3 + m)$ . Thus, this data structure achieves a runtime of  $O(\min\{n^3 + m, n^2m\})$  in the worst case.

## Quiz 2 Practice

### Problem 1. Trip Planning with a Gas Tank

You want to drive from some location  $s$  to some location  $t$  without running out of gas while spending as little money as possible. You are given a map of the road system designated by a set of locations  $V$  and a set of directed roads  $E$ , where  $(u, v) \in E$  means that there is a road going from  $u \in V$  to  $v \in V$ . You know  $w(u, v)$ , the length of the road from  $u$  to  $v$ . You are also told which locations  $S \subseteq V$  have gas stations. When you come to a node  $u$  with a gas station, you have the option to fill up the tank of your car by paying  $c(u)$  dollars (the gas is free - you only pay for the parking).

Assuming you start with a full tank containing  $k$  units of gas, give an efficient algorithm that finds a cheapest route from  $s$  to  $t$  such that you never run out of gas.

Note: it is okay to run out of gas at the same moment you reach a gas station.

#### Solution:

Here is the best solution to the problem. A discussion of other solutions is given below. It is worth noting here that a greedy approach does not work because of the nature of the constraints.

#### Algorithm:

The fastest way we know to solve this problem is to build a new graph over the gas stations and run a standard shortest-path algorithm on that graph.

Specifically, we want to create a new graph  $G'$  whose vertices are the locations with gas stations plus  $s$  and  $t$ , i.e.  $V' = S \cup \{s, t\}$ . Let  $\delta(u, v)$  be the length of the shortest path between  $u$  and  $v$  in  $G$ . Now, an edge between  $u$  and  $v$  exists in  $G'$  if and only if  $\delta(u, v) < k$ , i.e. we can get from  $u$  to  $v$  on one tank of gas. Finally, the edge weights  $w'(u, v)$  in  $G'$  are the cost of filling up at  $v$ , i.e.  $w'(u, v) = c(v)$ .

Our algorithm will be as follows:

1. Construct  $G'$ .
2. Find the shortest path in  $G'$ .
3. Recover the path in  $G$  that is associated with the path found in  $G'$ .

There are many ways to do this with varying runtimes. The easiest solution is to use Floyd-Warshall to find the all-pairs shortest paths in  $G$ . Then, we can run Bellman-Ford on  $G'$ . Once we

have run Bellman-Ford, we can look back at the results from Floyd-Warshall to determine which partial paths corresponded to the path found in  $G'$ .

Floyd-Warshall requires  $O(V^3)$  time and  $O(V^2)$  space. Bellman-Ford will require  $O(VE)$  time and certainly  $O(V)$  space plus the space for  $G'$ . This will give an overall running time of  $O(V^3)$  and space requirement of  $O(V^2)$ .

We can speed this up by recognizing that neither distances nor costs are normally negative. Thus, we can reasonably assume that  $w(u, v) \geq 0$  and  $c(u) \geq 0$  for all  $u, v$ . As a result, we can find all-pairs shortest paths by running Dijkstra's algorithm  $|V|$  times (essentially Johnson's algorithm without reweighting.) We can make this faster by observing that we only really need to run Dijkstra's algorithm starting at each of the gas stations. Thus, the time to generate  $G'$  is the time to run  $|S|$  versions of Dijkstra's algorithm, which is  $O(S \cdot (E + V \log V))$  using the Fibonacci heap implementation. Finally, we can use Dijkstra's algorithm again to find the shortest path in  $G'$  in  $O(S^2 + S \log S) = O(S^2)$  time. The running time will be dominated by the computation of  $G$  for a total of  $O(S \cdot E + SV \log V)$ .

The space requirement when using Dijkstra's algorithm is a little more complicated. Dijkstra's algorithm inherently requires  $O(V)$  space. However, we need to somehow recover the path in  $G$ . The easiest way to do this is to store the results from all  $|S|$  original calls to Dijkstra's algorithm. This will require  $O(S \cdot V)$  space. This is probably the most practical approach from a time perspective, but you can actually reduce the space requirement to  $O(S^2 + V)$  with some tricks. Instead of saving the results from Dijkstra's algorithm, we discard the information not necessary to generate  $G'$ . The graph  $G'$  itself can easily be stored in  $O(S^2)$  space, so producing  $G'$  requires  $O(S^2 + V)$ . (Note that it is possible that  $|E'| > E$ , so the best bound we can give on  $|E'|$  is  $O(S^2)$ .) Now, to recover the path in  $G$ , we run Dijkstra's algorithm again to find the information we discarded. We will rerun Dijkstra's algorithm at most  $|S|$  times, so the asymptotic running time is not affected. Finally, the extra information we keep corresponds to a path. Consequently, it contains at most  $|V|$  nodes. Thus, we will need at most  $O(V)$  space for the extra information we find. This gives an overall space requirement of  $O(S^2 + V)$  (plus the space requirement for the input graph  $G$ ).

### Correctness:

To see that this is indeed correct, first observe that any path in  $G'$  corresponds directly to a path in  $G$ , since each edge itself corresponds to a path in  $G$ . Thus, if the cheapest route in  $G'$  is not a cheapest path, it must be more expensive than the cheapest route in  $G$ .

Now, let  $P$  be the cheapest route in  $G$ , and decompose it into subsections between gas stations at which we fill up the tank. Clearly, each subsection corresponds to a distance that can be traversed on one tank of gas, which implies that the section generates an edge in  $G'$ . It follows that each subsection of  $P$  has an edge in  $G'$  and therefore the entire path must have a counterpart in  $G'$ . Moreover, observe that the cost of a path in  $G'$  is exactly the gas stations along the route, since you pay  $c(u)$  on any incoming edge to  $u$  but never on an outgoing edge. Thus, each route in  $G$  corresponds to a route of equivalent cost in  $G'$ .

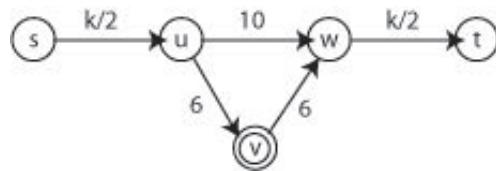
Therefore, the cheapest route in  $G'$  must also correspond to a route in  $G'$ , meaning that there is a path through  $G'$  with optimal cost.

**Comments and Other Ideas:**

Many people tried to copy each gas station so that there was one copy where the person filled up their gas tank and one where they didn't. (Either that, or they split a path when it reached a gas station.) This is handled in the above algorithm by allowing edges in  $G'$  to represent partial paths that go through gas stations.

There were also a handful of people who attempted to remove (one-by-one) all the vertices that were not gas stations. This is a valid idea, but the approaches presented did not have good analysis of the runtime necessary to perform this operation.

The insidiously tantalizing (yet wrong) approach to this problem is to modify Dijkstra's algorithm such that it ranks first by cost then by gas amount. The modification doesn't change the runtime or other properties of Dijkstra's, giving a possible  $O(E + V \log V)$  algorithm. Unfortunately, this doesn't work. Consider the following graph of 5 nodes (with one gas station at  $v$ ):



The greedy choice will pick the path to  $w$  avoiding the gas station at  $v$ , but won't have enough gas to get to  $t$ . Thus, the above algorithm won't work.

Another interesting approach uses the correct algorithm but replace Dijkstra's with a modification on BFS. The result is a slightly faster running time; however, it fails in a manner similar to the greedy approach. When you decide that a node is not reachable in this problem, you need to make sure you checked the shortest path to that node. This is not possible with BFS.

There was one significant alternative to constructing  $G'$ . If you assume that all distances are integers, then you can maintain a data structure at each node with  $k$  entries and change the notion of "visited" in Dijkstra's algorithm. (Alternatively, you can make  $k$  copies of each vertex to get a new graph with the same idea.) Now, you extend a path as long as the amount of gas in the tank is above the maximum already attained at that node. Then, since weights are integers, there are only  $k$  possible amounts of gas you can have in the tank, so you can go through each node at most  $k$  times. (Note that without the integer assumption, you may have  $2^{|E|}$  possible distinct amounts of gas, so this approach doesn't work anymore.) Apart from the assumption that  $k$  is an integer, the only problem with this algorithm is that  $k$  may be very large, resulting in high runtimes.

**Problem 2. Karp and Rabin compute the median**

Karp and Rabin each hold a set of  $n$  numbers, denoted by  $K$  and  $R$ , respectively. The sets  $K$  and  $R$  are disjoint. They would like to compute the median of the union set  $K \cup R$ . However, they live on different coasts, and they can communicate only by exchanging simple messages. Each message can either contain a number from their sets, or an integer in the range  $[1, n]$ .

Give an efficient communication protocol that Karp and Rabin can use to compute the median of the set  $K \cup R$ . The efficiency of the protocol is measured by the number of messages exchanged.

**Solution:** The most efficient solution to this problem uses  $O(\log n)$  messages in the worst case. There are many ways to achieve this. However, all efficient solutions have some “binary search flavor”. We note that a solution using  $O(n)$  messages can be achieved by transferring the whole data set to one of the parties and performing the computation locally.

Define  $\text{rank}_S(a)$  to be the number of elements in  $S$  that are smaller than  $a$ . Note that according to this definition, the smallest element has rank 0. The first observation is that, for any element  $a \in K$ , it is possible for Karp and Rabin to determine  $\text{rank}_{K \cup R}(a)$  using only constant number of messages. This is done as follows: Karp sends  $a$  to Rabin; Karp (or Rabin, respectively) compute the rank  $r_K(a)$  of  $a$  in  $K$  (or  $r_R(a)$  of  $a$  in  $R$ , respectively) using the local information; Karp and Rabin exchange the ranks and set the rank of  $a$  in  $K \cup R$  to be  $r_K(a) + r_R(a)$ .

According to the definition in CLRS, p. 183, the median of  $K \cup R$  is the element  $a$  with rank  $n - 1$  (but we accepted solutions with more exotic median definitions). Assume for the time being that the median is contained in  $K$ . Since the function  $\text{rank}_{K \cup R}(a)$  is monotone in the argument  $a$  (i.e., for any  $a, a' \in K \cup R$ , if  $a < a'$  then  $\text{rank}_{K \cup R}(a) < \text{rank}_{K \cup R}(a')$ ), we can find an element with rank  $n - 1$  using binary search on elements of  $K$ . We first convert the set  $K$  into a sorted array  $SoK[1 \dots n]$ . Then we perform binary search on the array  $SoK$  with respect to the ranks. Conceptually, we binary search for the value  $n - 1$  in the sorted array  $\text{rank}_{K \cup R}(SoK[1]), \dots, \text{rank}_{K \cup R}(SoK[n])$ . Each time we need to compare  $\text{rank}_{K \cup R}(SoK[i])$  to  $n - 1$ , we exchange  $O(1)$  messages to compute the rank. Since binary search terminates after  $O(\log n)$  steps, it follows that the median is located using  $O(\log n)$  messages. The correctness of this procedure follows directly from the correctness of the binary search procedure.

Therefore, the problem is solved *if* the median element is held by Karp. If this is not the case, the aforementioned protocol fails to locate an element in  $K$  with rank  $n - 1$ . In that case the protocol is repeated, with the roles of Karp and Rabin reversed.

This concludes the description of the simplest variant of the protocol. It exchanges  $O(\log n)$  messages, performs  $O(n \log n)$ -time computation, and is in-place, assuming that an in-place sorting algorithm is used. Various optimizations are possible. They do not decrease the asymptotic number of messages exchanged, but can reduce the running time to  $O(n)$ .

There are many variants of the aforementioned procedure. For example, the elements  $a$  for which we compute  $\text{rank}_{K \cup R}(a)$  could be chosen from  $K$  in odd rounds and from  $R$  in even rounds. We could also choose those elements at random from  $K$  or  $R$ , in a manner analogous to the Randomized Select procedure. This however leads only to  $O(\log n)$  messages *in the expectation*. In addition, the analysis of Randomized Select needs to be modified slightly, given that at each step we choose a random element from either  $K$  or  $R$ , not from the whole data set.

### Problem 3. The lightest path

Consider a weighted connected graph  $G$ , with  $m$  edges and  $n$  vertices. Each edge weight is an integer in the range  $[1, 10n]$ . Consider any two nodes  $u, v \in G$ , and a path  $P$  between  $u$  and  $v$ . The *weight* of the path  $P$  is defined as the *maximum* weight of any edge in  $P$ .

Give an efficient algorithm that given the graph  $G$  and vertices  $u$  and  $v$ , finds the minimum weight path between  $u$  and  $v$  in  $G$ .

**Solution:**

A common approach taken by students was to modify Dijkstra's algorithm. Instead of storing the shortest known distance to a node, store the smallest known path weight to the node. In particular, change the RELAX subroutine (which potentially updates the value at node  $v$  using the best-known path to  $u$ ) to the following:

```
RELAX(u,v,w)
1      if  $d[v] > \max(d[u], w(u, v))$ 
2          then  $d[v] \leftarrow \max(d[u], w(u, v))$ 
3           $\pi[v] \leftarrow u$ 
```

We essentially replace the '+' operator with a 'max' operator. The arguments used in the proof of correctness for Dijkstra's algorithm still holds after doing so. In particular, path weights are nondecreasing whenever they are extended (this is true even if we allowed negative edge weights) because the max operator will always keep the larger value. Thus we know that for the set of vertices that has not yet been relaxed, the vertex that has the minimum known cost can not get an improved cost by going through any other intermediary vertex. It is always therefore safe to relax that vertex and add it to the set of finished vertices to which we have found optimal paths. Interestingly enough, this problem loses optimal substructure in that it is no longer a strict requirement that all subpaths must be optimal for the entire path to be optimal. That is, if two parts of the path contain two different edges that both have the maximum edge weight required for the path, if we improve the first half of the path, the entire path is not made more optimal. Points were taken off for making incorrect arguments and for not making any.

Simply using a Fibonacci heap results in a running time of  $O(m + n \lg n)$ . However, because the edge weights can only be integers in the range  $[1, 10n]$ , a monotone priority queue of size  $10n$  can be used as described in Problem Set 5 since the only path weights possible are the edge weights. This results in an overall running time of  $O(n + m)$ . (Note: Because the graph is connected, we know that  $n = O(m)$ , so we can also simplify the running time to  $O(m)$ .)

The other main approach students took was to "grow" a copy of the graph by adding in the edges of the original graph in order of their edge weights. Once the two nodes  $u$  and  $v$  are connected, we know that the last edge added to the graph is the minimum path weight possible. The simplest way of implementing this algorithm is to first sort the edges by edge weight (by using counting sort, we can get  $O(m + n)$  because the  $m$  edge weights can only be of  $10n$  different values), add the edges of the smallest edge weight not yet in the graph, test the connectivity of  $u$  and  $v$ , and then iterate. Testing connectivity can be done by using Dijkstra's (rather suboptimal), using BFS or DFS

(which takes  $O(m + n)$  each time for  $O(mn)$  total running time), or maintaining the connected components using the Union-Find data structure described in CLRS (for  $O(m\alpha(n))$  time). While the last of these techniques depends on growing the graph edge by edge, the former two techniques can be sped up by performing a binary search on the minimum edge weight required to connect  $u$  and  $v$ . In other words, try the graph with only edge weights at most  $5n$ , then either  $2.5n$  or  $7.5n$ , etc. This would yield a running time of  $O(m \log n)$  when using BFS or DFS to test connectivity.

The Union-Find approach is implemented by first making each vertex into a set by itself. Then, whenever an edge is added, we union the two sets that contain the endpoints of the edge. We test connectivity by finding the set representatives of  $u$  and  $v$  and check if they're the same. Using the union by rank heuristic gives an amortized bound of  $O(\alpha(n))$  time per Union-Find operation, where  $\alpha(n)$  is the inverse Ackerman function and grows incredibly slowly (its value is on the order of 4 or 5 on reasonable input sizes).

Another clever solution actually achieves  $O(m)$  running time also by growing the graph but saves time by only keeping track of which nodes are reachable from  $u$  using a variable stored at each node. At the beginning,  $u$  is the only one marked as reachable from  $u$ . When adding the edges in sorted order, if one endpoint of the edge is marked as reachable and the other isn't, mark the second vertex as reachable and perform a BFS to mark everything connected to that node as reachable. If  $v$  is ever encountered, we're done. If neither endpoint of the added edge is reachable or both of them are, simply insert the edge and do nothing. This algorithm only performs a constant amount of work when adding an edge to the graph for a total of  $O(m)$  for all the edges, and the series of BFSs only traverse each vertex and edge once for a total of  $O(m + n)$ , so the entire algorithm runs in  $O(m)$ .

Another related approach is to build a minimum spanning tree. In an MST, the path between  $u$  and  $v$  necessarily has the minimum path weight possible in the graph. Thus we can simply run an MST algorithm and use DFS to find the path between  $u$  and  $v$ . Since the MST is a tree, there is only one unique path for the MST. To see how the path in the MST is correct, it is useful to think of how Kruskal's algorithm works. (This is fact pretty much like the Union-Find approach except we don't add edges that are unnecessary.) We sort the edges by weight, and we keep adding edges if they do not form cycles. If the path in the MST is wrong, then that means that there is another path in the graph that connects through smaller weight edges in favor of some larger edge used in the MST. Obviously, not all of those edges are in the MST, or otherwise the MST would not be a tree or we would be taking this path. In the process of Kruskal's algorithm, we would have considered adding all those lighter edges before having added in the larger edge. If we chose not to add some of those edges, it would be because adding them would not increase the connectivity; the MST would already have contained a path that connected the proposed edge's endpoints with lesser or equal cost. This implies that if this better path of lighter edges existed, it or another path of equal or lesser cost would be added into the MST before the larger edge was added. Because this path is in existence, the larger edge can no longer be part of the path between  $u$  and  $v$ . Similar to the Dijkstra's algorithm approach, the traditional running time for finding the MST is not quite  $O(m)$  but can be improved because of the integer edge weights.

All of these approaches can be performed using  $O(m + n)$  space, and only  $O(n)$  space for im-

plementing Dijkstra's. Both were an acceptable use of space. Some students' solutions stored the entire path at each node, resulting in using  $O(n^2)$  space and therefore  $\Omega(n^2)$  time, depending on what else their algorithm did. Note that this modification to Dijkstra's is entirely unnecessary since the method shown in CLRS of simply maintaining the predecessor of a vertex is sufficient and only uses  $O(n)$  space. The path can be reconstructed after running Dijkstra's simply by tracing back and following the predecessors.

**Problem 4. Updating Max-Flow** Let  $G = (V, E)$  be a flow network with source  $s$ , sink  $t$ , and integer capacities. Suppose that we are given a maximum flow in  $G$  obtained using the Ford-Fulkerson method. Suppose that the capacity of a single edge  $(u, v) \in E$  is increased by 1. Give an  $O(V + E)$ -time algorithm to update the maximum flow.

**Solution:** Let  $f^*$  denote the maximum flow of  $G$  given to us by the Ford-Fulkerson method. By the max-flow min-cut theorem (27.7 in CLR), the fact that  $f^*$  is a maximum flow implies that the residual network  $G_{f^*}$  has no augmenting path. When we increase the capacity of  $(u, v)$  by 1, the residual capacity  $c_{f^*}(u, v)$ , defined as  $c(u, v) - f^*(u, v)$ , also increases by 1. Notice that  $f^*$  is a valid flow in the modified flow network  $G'$ . Thus, the value of the maximum flow cannot decrease; it can either stay the same or increase by at most 1 (an increase by more than 1 would contradict the maximality of  $f^*$  in  $G$ ).

If there is no augmenting path in the residual network  $G'_{f^*}$  then  $f^*$  is a maximum flow (max-flow min-cut theorem). Otherwise, there is an augmenting path  $p$  in  $G'_{f^*}$  and we can ship some additional amount of net flow along the edges of an augmenting path  $p$ . This amount is equal to the residual capacity of  $p$ , which is greater than zero by the definition of the residual network. However, by the integrality theorem (27.11 in CLR), it must be an integer, because the capacities and flows of all the edges are integers. It follows that the residual capacity must be at least 1. Since the flow cannot increase by more than one unit, the residual capacity of  $p$  must be exactly 1.

To update the maximum flow, modify the capacity of  $(u, v)$  and construct the residual network of  $G'$  with flow  $f^*$ . This can be done in  $O(V + E)$  (see section 27.2). Next, find an augmenting path  $p$ , using either BFS or DFS, which take  $O(V + E)$ . If there is an augmenting path  $p$ , for every edge  $(v_1, v_2)$  on  $p$ , increment  $f^*(v_1, v_2)$  by 1. The resulting flow is the maximum in  $G'$ . If not, no update of the maximum flow is necessary:  $f^*$  is the maximum flow of  $G$  with the capacity of  $(u, v)$  increased by 1.

### Problem 5. Search Engine Excerpts

Most popular Internet search engines feature document excerpts for each found document. The goal of search engines is to provide the smallest document excerpt possible that helps the user realize if the found document is relevant to their search. One possible heuristic is to find the smallest excerpt from a found document that includes all of the keywords used in the search.

You are given  $k$  search keywords, and a document that is a sequence of  $N$  words. All  $k$  search keywords are present in the document.

Describe an efficient algorithm that finds a smallest range  $[i, j]$  in the document ( $i, j \in [1, N]$ ), that contains at least one instance of every search keyword.

Note: you can assume that each word or search keyword is represented by a number. **Solution:**

There were three  $O(N)$  solutions to this problem, as well as numerous variations that provided slower runtimes. All solutions begin by hashing each keyword  $k_i$  into a table  $T$  (either using perfect hashing, or some mechanism for resolving false positives in lookups).

The first general solution involves a “sliding window” pass over the document words from 1 to  $N$ . More specifically, two pointers,  $i$  and  $j$ , define a window over the document. The value associated with each  $k_i$  in  $T$  tracks the number of instances of  $k_i$  within this window, and a counter  $m$  tracks the number of entries with non-zero values. Both  $i$  and  $j$  start at index 1 of the document.  $j$  then slides forwards until  $m = 0$ . Whenever a keyword is identified during the slide (in constant time via hash table lookup), the appropriate counter in  $C$  is incremented. Whenever an entry’s value goes from zero to one,  $m$  is incremented.

Once  $m = k$ , we have a complete excerpt. To make the excerpt minimal, we now slide  $i$  forward, decrementing hash table entries and  $m$  when keywords are encountered, until incrementing  $i$  would make our excerpt incomplete. At this point we record our range as the current best complete range, and then slide  $i$  to the next keyword instance. We then repeat our process of sliding  $j$ , then  $i$ , and updating our best observed range, until  $j$  hits the end of our document.

This solution takes  $O(N)$  time and  $O(N)$  space.

The second general solution was to traverse the document, tracking the latest instances of all  $k_i$ , and recording best ranges at each new keyword. More specifically, a linked list  $L$  of latest keyword instances is created during initialization, and populated with one element per  $k_i$  that stores the value -1. The values of the hash table  $T$  are each assigned one of these list elements. We now start a pointer  $i$  at the first document word and scan to the right. At each identified keyword at index  $j$  (again in constant time via  $T$ ), the linked list element for  $k_i$  has its value updated to  $j$ , and is spliced to the end of the linked list  $L$  (because we only ever increase values, causing elements to move to the end of the list, we are in effect implementing a monotone priority queue). Whenever a list element value is changed from -1, a counter  $m$  is incremented. Once  $m = k$ , the interval  $[1, i]$ , contains at least one instance of each keyword. Whenever we find a new keyword, we take the value of the head of our linked list and our current index to form a best complete range, and continue updating this value as  $i$  slides forward. We terminate when  $j$  reaches the end of the list.

This solution takes  $O(N)$  time and  $O(N)$  space.

The third general solution involves splitting the keyword instances into  $k$  separate linked lists and processing them in quasi-parallel. More specifically, the document is first run through once, and linked lists of matched instances (again identified in  $O(1)$ ) are formed off of each  $k_i$  entry in  $T$ . These lists are of necessity sorted, and have aggregate length  $O(N)$ . The range defined by the heads of each of these linked lists will define our current complete excerpt range. We record the maximum head pointer in  $max$  after a  $O(k)$  search. We now begin scanning the document a second time. Whenever a keyword  $k_i$  is matched, three actions occur. First, the range defined by the head

element of the list hashed to be  $k_i$  and  $\max$  is checked for being the best range (which we track during the scan). Second, the head element of the list  $k_i$  hashes to is removed. Third, the new head element is compared to  $\max$ , and if found greater, takes  $\max$ 's place. The scan stops whenever any of the linked lists is emptied.

This solution takes  $O(N)$  time and space.

In general, full credit was given for a bugless  $O(N)$  solution. Penalties were applied for bugs, significant omissions, extremely confusing writeups, or incomplete correctness or analysis. Partial credit was given to  $O(N \lg k)$  solutions that used different data structures to either track latest keyword instances for solution two, or to find minimums in solution three. Less credit was given for similar solutions that used less efficient data structures to get  $O(Nk)$  algorithms. Less credit was given to  $O(N^2)$  algorithms that used a different, more exhaustive approach.

### Problem 6. Languages of the new empire

In the glory days of the Roman empire, Julius Caesar seeks to establish efficient communications between Rome and the myriad provinces it commands in close and remote corners of the empire. One-way broadcast messages are sent frequently from Rome to all of the  $N$  provinces along previously-specified paths, to announce new taxes, new wars, and other urgent matters. The problem is that many different languages are spoken across the empire, and thus a lot of consecutive translations may need to happen along the way to a remote province when a message is sent, causing unwanted delays. Each province speaks only one of the  $K$  total languages in the empire, and Caesar insisted that they receive messages in their own language (his magnanimity knows no limits).

Communication happens by messengers carrying stone tablets, along a graph  $G = (V, E)$  of roads and sailing routes. Since all roads lead to Rome and only three-way intersections are present, the communication graph is a binary tree, with Rome at the root and provinces at the leaves. At each intersection (each internal node) lies an outpost where a single stone tablet arrives, and is then copied and passed on (exactly once for each of the two outgoing paths), in the same language when possible, or translated to a new language when necessary. For every broadcast, a single message tablet is created in Rome, in Latin, and arrives at Rome's outpost, where it starts its journey throughout the empire. After each intersection, exactly one tablet, in a specified language, leaves down each subsequent path. Since translation of messages takes a long time, you need to select the language of tablets to be used along each segment in a way that minimizes the total cost of needed translations.

In addition to the graph  $G$ , and the language spoken at each province, you are given the cost of translating from each language to each other language, represented as a  $K \times K$  matrix  $M$  of positive numbers for the cost  $M[i, j]$  of translating from language  $i$  to language  $j$ . Note that translation costs need not be symmetrical, that the cost of translating a language to any other language is always strictly positive, and that all diagonal entries of matrix  $M$  are zero as no translation is necessary from a language to itself. You can also assume that the costs reported in  $M[i, j]$  are optimal,

namely you cannot get a lower translation cost from  $i$  to  $j$  by translating through a series of other languages.

Design an efficient algorithm to select the language of tables to be used on each segment (and the translations done at each outpost) in order to minimize the total cost of translations. For partial credit, you can simplify the problem by assuming that any translation has unit cost, i.e.  $M[i, j] = 1$  for  $i \neq j$ .

### Solution:

We set up a Dynamic Programming (DP) Cost table  $Cost_v[1..K]$  for each vertex  $v$ , to be filled in starting from the leaves and towards the root. At each entry  $Cost_v[i]$ , we store the minimum cost of translations for sending a message from vertex  $v$  to all of the provinces in that subtree, if language  $i$  is used in the road leading to that outpost from Rome. We update the  $Cost_v$  table for each outpost  $v$  based on the two tables  $Cost_{v_{right}}$  and  $Cost_{v_{left}}$  at its two children  $v_{right}$  and  $v_{left}$ , by evaluating all combinations of costs at that intersection and taking the min. Once the min is found, we also store the choice of assignment leading to the optimal score at each vertex in a table  $Assignment_v[i]$ , with entries  $(j, k)$  if language choices  $j$  and  $k$  at the children led to minimum value, thus enabling a traceback back towards the leaves after the root is reached, in order to find the final language assignment for each outpost  $v$ .

We start by the proof that DP is applicable here. First, there's only a finite number of subproblems, as a leaf with  $N$  leaves has  $N - 1$  internal vertices, each of which has only  $K$  possible languages. Second, the problem exhibits optimal substructure: a solution using the minimum cost of translations to get from Rome to any of the provinces must be made of optimal solutions to get from any of the internal outposts to the provinces it leads to. The proof is by contradiction using a cut-and-paste argument: if a solution from a given province existed with a smaller cost of translations, that portion could be replaced in the overall solution, leading to an overall solution better than the optimal, hence a contradiction.

Next, we set up the dynamic programming update rule, to compute the table of optimal translation costs for a given outpost  $v$  whose children  $v_{right}$  and  $v_{left}$  have their optimal costs already computed. In the general case, the cost of using language  $l$  at outpost  $v$  would be computed as:

$$Cost_v[l] = \min_{i,j} \{Cost_{v_{right}[i]} + Cost_{v_{left}[j]} + M[l, i] + M[l, j]\}$$

This general update rule would have a cost of  $k^3$ , as each of  $k$  entries are filled in, each time testing  $k^2$  combinations of assignments for the two children. However, since translation costs are independent between the two sides, a speedup is possible:

$$Cost_v[l] = \min_i \{Cost_{v_{right}[i]} + M[l, i]\} + \min_j \{Cost_{v_{left}[j]} + M[l, j]\}$$

This independence relies on the fact that no optimal solution requires that two consecutive translations are necessary at a single node  $v$  (from  $i \rightarrow j$  and then  $j \rightarrow k$ ) and that only  $i \rightarrow j$  and  $i \rightarrow k$  solutions are allowed at any one node. The reason for this is that instead of sending  $i$  to node  $v$ , and then doing two consecutive translations at that node, we can simply translate from  $i$  to  $j$  at the parent of  $v$ , and then send  $j$  down that node which we then translate to  $k$  at node  $v$ .

The optimized update rule allows us to pay only cost  $k^2$  for each outpost, since each of  $k$  entries is updated in  $O(k)$  time. Note that  $\text{Assignment}_v[i]$  records the assignment of the left and right children of  $v$  that lead to the optimal translation cost when the language at  $v$  is set to be  $i$ . Also note that the two subloops can be run in any order, as the 'relaxation' step is never surpassing the optimal solution (additional relaxation steps don't hurt).

We initialize the DP computation by setting the cost table for each leaf  $p$  (corresponding to province  $p$ ) as follows:

$\text{Cost}_p[j] = M[j, i]$ , where  $i$  is the language spoken at province  $p$ .

(an alternative strategy may be to set  $\text{Cost}_p[i] = 0$  and  $\text{Cost}_p[j] = \infty$  for all other languages, as we know that no optimal solution will use a different language at the last outpost than the language spoken at that province).

Upon termination, the root contains a vector of the value of the globally optimal solution for each choice of language starting at the Rome outpost. Since the incoming language there is always Latin, we have to add to that value  $\text{Cost}_{\text{Root}}[i]$  the cost of translating Latin to that language  $M[\text{Latin}, i]$ . We choose the language  $i$  minimizing that sum, set that language for the Rome outpost. The  $\text{Assignment}_{\text{Rome}}[\text{best}]$  then starts the series of pointers to children vertex assignments that we trace back to construct the optimal solution, assigning languages to all outposts.

The run time analysis for the entire algorithm is simple.  $O(N * K^2)$  for computing the optimal solution, since a binary tree with  $n$  leaves has  $n - 1$  internal vertices, and since the table of each vertex is computed exactly once, and each computation takes  $O(k^2)$  time. The traceback takes linear time as each optimal solution is immediately looked up by following the  $\text{Assignment}_v[\text{best}]$  pointers. The space requirements are  $O(N * K)$  for storing  $K$  values and  $K$  tuples at each vertex.

*Notes:*

Note that a greedy algorithm is not possible. The reason is that the local information within a subtree is insufficient to make a globally-optimal choice. For example, if A and B at the root of a subtree have equal cost within the subtree, the choice will not be resolved until we know whether A and B are dominant outside the subtree. More generally, even if some language L1 is giving lowest cost within a subtree, another language L2 may in fact come to be cheapest for that subtree considering the outside information.

In particular, even for the unit-cost version of the problem, simple voting schemes will not work (of counting the provinces beneath a node that speak each language, and choosing the majority). The reason is that it's the layout of the languages that matters, not the total count. For example, the subtree (B,(B,(B,(B,(B,(A,A),(A,((A,A),(A,A)))))))) contains 7 A's and 6 B's, hence at every node, the majority-voting scheme will choose A, leading to a total of 6 translations, from the top-most A into each individual B. The optimal solution however starts with a top-most B and only requires 1 translation from B to A at the root of the all-A subtree.

It should also be noted that the overlapping subproblems nature of the problem can only be exploited in bottom-up algorithms that move from the leaves to the root. The reason is that optimal solutions to the subproblems (given a starting language at the root) can only be computed for

fully-explored subtrees at a time. This is not possible from the root down, and would lead to exponential-time algorithms that have to travel all the way down to the leaves at each iteration to collect information.

To receive full credit, solutions had to demonstrate a working Dynamic Programming (DP) solution, including set-up of cost variables, bottom-up propagation of costs, a precise and correct update rule with initialization and termination conditions, a traceback step after appropriate update of max pointers, and a proof of correctness and analysis of running time. Partial credit was received for Greedy solutions that propagated sums to the root and traced back to maximum-frequency choices, as they demonstrated the principles of DP, despite the misunderstanding described above.

## Quiz 2 Solutions

- Do not open this quiz booklet until you are directed to do so. Read all the instructions first.
- When the quiz begins, write your name on every page of this quiz booklet.
- The quiz contains four multi-part problems. You have 120 minutes to earn 108 points.
- This quiz booklet contains **16** pages, including this one. Extra sheets of scratch paper are attached. Please detach them before turning in your quiz.
- This quiz is closed book. You may use one handwritten A4 or  $8\frac{1}{2}'' \times 11''$  crib sheet. No calculators or programmable devices are permitted.
- 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, since the pages may be separated for grading.
- Do not waste time and paper rederiving facts that we have studied. It is sufficient to cite known results.
- 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.
- 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.
- Good luck!

Problem	Points	Grade	Initials
1	40		
2	25		
3	25		
4	18		
Total	108		

Name: Solutions \_\_\_\_\_  
Circle your recitation letter and the name of your recitation instructor:

David      A      B      Steve      C      D      Hanson      E      F

**Problem -1. True or False, and Justify [40 points]**

Circle **T** or **F** for each of the following statements, and briefly explain why. The better your argument, the higher your grade, but be brief. Your justification is worth more points than your true-or-false designation. Parts **(a)** through **(d)** are worth **2 points** each and parts **(e)** through **(l)** are worth **4 points** each.

- (a) T F** A preorder traversal of a binary search tree will output the values in sorted order.

**Solution:** False. Consider a BST with 2 at the root and 1 as left child and 3 as right child. The preorder is 2, 1, 3. An *inorder* traversal of a BST outputs the values in sorted order.

- (b) T F** The cost of searching in an AVL tree is  $\Theta(\lg n)$ .

**Solution:** True. An AVL tree is a balanced binary search tree. Therefore, searching in an AVL tree costs  $\Theta(\lg n)$ .

- (c) **T F** In a topological ordering of the vertices of a DAG, if a vertex  $v$  comes after a vertex  $u$  in the ordering, then there is a directed path from  $u$  to  $v$ .

**Solution:** False. A vertex  $u$  may come before  $v$  in the topologically sort, but that does not mean there is actually a path from  $u$  to  $v$ . In particular,  $v$  may have no in-edges.

- (d) **T F** The transitive closure of a directed graph  $G = (V, E)$  can be computed in  $O(V^3)$  time.

**Solution:** True. We can use FLOYD-WARSHALL to compute the all pairs shortest paths. If there is no directed path from  $i$  to  $j$  in  $G$ , then the shortest path from  $i$  to  $j$  is  $\infty$ .

- (e) **T F** Any mixed sequence of  $m$  increments and  $n$  decrements to a  $k$ -bit binary counter (that is initialized to zero and is always non-negative) takes  $O(m + n)$  time in the worst case.

**Solution:** False. If we allow decrement operations, then a series of  $n + m$  increment and decrement operations could take  $O(kn + km)$  in the worst case.

- (f) **T F** Suppose we have a directed graph  $G = (V, E)$  that is strongly connected. For any depth-first search of  $G$ , if all the forward edges of  $G$  (with respect to the depth-first forest) are removed from  $G$ , the resulting graph is still strongly connected.

**Solution:** True. In a DFS on  $G$ , *forward edges* are those nontree edges  $(u, v)$  connecting a vertex  $u$  to a descendant  $v$  in the depth-first tree. Since  $v$  is a descendant of  $u$  on the DFS tree, there are tree edges that create a directed path from  $u$  to  $v$ . Therefore, after removing edge  $(u, v)$ , there is still a directed path from  $u$  to  $v$ , so  $G$  is still strongly connected.

- (g) **T F** Consider a directed acyclic graph  $G = (V, E)$  and a specified source vertex  $s \in V$ . Every edge in  $E$  has either a positive or a negative edge weight, but  $G$  has no negative cycles. Dijkstra's Algorithm can be used to find the shortest paths from  $s$  to all other vertices.

**Solution:** False. Counterexample  $G = \{w(s, b) = -3, w(s, c) = 2, w(b, c) = 4\}$ .

- (h) **T F** Suppose someone has run FLOYD-WARSHALL on a weighted graph  $G = (V, E)$  and gives you a 3-dimensional array  $D$  in which  $D[i, j, k] = d_{ij}^{(k)}$  (for  $0 \leq i, j, k \leq n$ ) is the length of the shortest path from vertex  $i$  to vertex  $j$  for which all intermediate vertices are in the set  $\{1, 2, \dots, k\}$ . The array  $D$  can be used to determine if  $G$  contains a negative cycle in  $O(V)$  time.

**Solution:** True. For each pair of vertices  $i, j \in V$ , FLOYD-WARSHALL computes  $d_{ij}^{(0)}, d_{ij}^{(1)}, \dots, d_{ij}^{(n)}$ . Thus, if there is a negative cycle, then  $d_{ii}^{(n)}$  will be negative for some  $i$ .

- (i) **T F** Given a bipartite graph  $G$  and a matching  $M$ , we can determine if  $M$  is maximum in  $O(V + E)$  time.

**Solution:** True. Modified BFS is used to determine if there is an augmenting path in time  $O(V + E)$ . If there is no augmenting path, then the matching is maximum.

- (j) **T F** Given a random skip list on  $n$  elements in which each element has height  $i$  with probability  $(1/3)^{i-1}(2/3)$ , the expected number of elements with height  $\lg_3 n$  is  $O(1)$ .

**Solution:** True. Let  $X_i$  be the indicator random variable that is a 1 if element  $i$  is in the set of elements with height at least  $\lg_3 n$  and 0 otherwise. An element has height at least  $\lg_3 n$  with probability  $(2/3)(1/3)^{\lg_3 n - 1} = 2/n$ . Thus,  $E[X_i] = 2/n$  and  $E[\sum_{i=1}^n X_i] = \sum_{i=1}^n E[X_i] = 2 = O(1)$ .

- (k) **T F** Suppose you are given an undirected connected graph with integer edge weights in which each vertex is degree 2. An MST can be computed for this graph in  $O(V)$  time.

**Solution:** True. The graph must be a cycle with  $O(V)$  edges. We can remove the edge with the highest weight in  $O(V)$  time.

- (l) **T F** Consider an undirected, weighted graph  $G$  in which every edge has a weight between 0 and 1. Suppose you replace the weight of every edge with  $1 - w(u, v)$  and compute the minimum spanning tree of the resulting graph using Kruskal's algorithm. The resulting tree is a maximum cost spanning tree for the original graph.

**Solution:** True. Let  $G'$  be the graph in which every edge has weight  $1 - w(u, v)$ . Let MaxST denote the maximum spanning tree in  $G$  and let MST denote the minimum spanning tree in  $G'$ . Suppose MST does correspond to MaxST in  $G$ . Then there is a spanning tree in  $G$  with higher weight, implying that if we replace each edge with weight  $1 - w(u, v)$ , we will find a spanning tree in  $G'$  that has less weight than MST, which is a contradiction.

**Problem -2. Short Answer Problems** [25 points]

Give *brief*, but complete, answers to the following questions. Each problem part is worth **5 points**.

**Amortized Analysis** (re-written from Fall 98)

You are to maintain a collection of lists and support the following operations.

- (i) *insert(item, list)*: insert item into list (cost = 1).
  - (ii) *sum(list)*: sum the items in list, and replace the list with a list containing one item that is the sum (cost = length of list).
- (a) Use the Accounting Method to show that the amortized cost of an *insert* operation is  $O(1)$  and the amortized cost of a *sum* operation is  $O(1)$ .

**Solution:** We will maintain the invariant that every item has one credit. Insert gets 2 credits, which covers one for the actual cost and one to satisfy the invariant. Sum gets one credit, because the actual cost of summing is covered by the credits in the list, but then the result of the sum will need one credit to maintain the invariant.

A common error was not putting a credit on the newly created sum.

- (b) Use the Potential Method to show that the amortized cost of an *insert* operation is  $O(1)$  and the amortized cost of a *sum* operation is  $O(1)$ .

**Solution:** We define  $\Phi(list)$  to be the number of elements in the list. Then the amortized cost of an insert operation is  $\hat{c}_i = c_i + \Phi_i - \Phi_{i-1}$ . The actual cost  $c_i$  is 1. The change in potential is 1. So the amortized cost is 2. For a sum operation, the actual cost,  $c_i$  is  $k$  and the change in potential is  $\Phi_i - \Phi_{i-1} = 1 - (k) = 1 - k$ , so the amortized cost of a sum is 1.

**Balanced Search Trees**

- (c) Show that the number of node splits performed when inserting a single node into a 2-3 tree can be as large as  $\Omega(\lg n)$ , where  $n$  is the number of keys in the tree. (E.g. give a general, worst-case example.)

**Solution:** Suppose that every node in an  $n$ -node 2-3 tree is full – i.e., all internal nodes have three children. Then the height of the tree is  $\log_3 n = \Omega(\log n)$ . [Common error: I don't care that its height is  $O(\log n)!$ ] When we insert into a leaf node, it's full—so we have to have to do something with the fourth element ... that is, bump the median element up to its parent. But its parent is full, too—so we have to bump up an element from there. And so on, all the way up to the root. The numbers of splits is  $\Omega(\text{height}) = \Omega(\log n)$ .

### Faster MST Algorithms

- (d) Given an undirected and connected graph in which all edges have the same weight, give an algorithm to compute an MST in  $O(E)$  time.

**Solution:** Run DFS and keep only the tree edges.

Some common errors were: (1) not running in linear time (union-find is NOT constant!) and (2) not producing a tree (a graph where every node has degree  $\geq 1$  is not necessarily connected!).

**Competitive Analysis** (written by DLN)

- (e) Your TA can be bribed to give you an A+ in 6.046 if you give him at least  $x$  dollars, where  $x$  is a positive integer. *However, you do not know what  $x$  is.* Since your TA is forgetful, you must give him  $x$  dollars at once; if you give him less, he just keeps it and forgets who gave it to him. Your goal is to get an A+ and minimize the amount paid to the TA. You have the following strategy: you pay your TA 1 dollar and keep doubling the amount you pay until you get an A+. The following is pseudocode for this strategy.

BRIBE-TA

```

1 paid  $\leftarrow$  0
2 amount  $\leftarrow$  1
3 While TA has not given you an A+
4   Pay TA amount dollars
5   paid  $\leftarrow$  paid + amount
6   amount  $\leftarrow$  2 * amount
7 Return paid
```

Analyze the competitive ratio of the Bribe-TA algorithm.

**Solution:** We will show that the Bribe-TA algorithm is 4-competitive, because *paid* is at most  $4x$  dollars and  $x$  dollars is the minimum amount we can pay.

Suppose  $2^k \leq x \leq 2^{k+1}$ . Then the total amount paid to the TA is:

$$\sum_{i=0}^{k+1} 2^i = 2^{k+2} - 1 \leq 4x.$$

**Problem -3. Dynamic Programming [25 points]**

Santa Claus is packing his sleigh with gifts. His sleigh can hold no more than  $c$  pounds. He has  $n$  different gifts, and he wants to choose a subset of them to pack in his sleigh. Gift  $i$  has utility  $u_i$  (the amount of happiness gift  $i$  induces in some child) and weight  $w_i$ . We define the weight and utility of a *set of gifts* as follows:

- The weight of a set of gifts is the *sum* of their weights.
- The utility of a set of gifts is the *product* of their utilities.

For example, if Santa chooses two gifts such that  $w_1 = 3, u_1 = 4$  and  $w_2 = 2, u_2 = 2$ , then the total weight of this set of gifts is 5 pounds and the total utility of this set of gifts is 8. All numbers mentioned are positive integers and for each gift  $i$ ,  $w_i \leq c$ . Your job is to devise an algorithm that lets Santa maximize the utility of the set of gifts he packs in his sleigh without exceeding its capacity  $c$ .

- (a) [5 points] A greedy algorithm for this problem takes the gifts in order of increasing weight until the sleigh can hold no more gifts. Give a small example to demonstrate that the greedy algorithm does not generate an optimal choice of gifts.

**Solution:** Suppose that  $c = 2$  and the gifts have weights 1 and 2 and  $u_i = w_i$  for each gift. If Santa chooses gift 1, then he can not fit gift 2, so the total utility he obtains is 1 rather than 2.

**(b) [10 points]** Give a recurrence that can be used in a dynamic program to compute the maximum utility of a set of gifts that Santa can pack in his sleigh. Remember to evaluate the base cases for your recurrence.

**Solution:** Let  $H(k, x)$  be the maximal achievable utility if the gifts are drawn from 1 through  $k$  (where  $k \leq n$ ) and weigh at most  $x$  pounds.

For  $1 \leq k \leq n$ :

If  $x - w_k \geq 0$ ,  $H(k, x) = \max\{H(k - 1, x - w_k) \cdot u_k, H(k - 1, x), u_k\}$

Otherwise (if  $x - w_k < 0$ ),  $H(k, x) = H(k - 1, x)$

**Base cases:**  $H(0, x) = 0$  for all integers  $x$ ,  $1 \leq x \leq c$ .

Some common errors were (1) forgetting to include  $u_i$  in the recurrence for the case in which  $H(k - 1, x - w_k)$  is 0, (2) neglecting one dimension, e.g. weight, (3) using + instead of \* for utility, (4) writing the recurrence as  $H(i, c)$  instead of (general)  $H(i, x)$ .

- (c) [7 points] Write pseudo-code for a dynamic program that computes the maximum utility of a set of gifts that Santa can pack in his sleigh. What is the running time of your program?

**Solution:** We give the following pseudo-code for a dynamic program that takes as input a set of gifts,  $\mathcal{G} = \{g_1, g_2, \dots, g_n\}$ , a maximum weight,  $c$ , and outputs the set of gifts with the maximum utility.

MAXIMUM-UTILITY( $\mathcal{G}, c$ )

- 1 For  $1 \leq x \leq c$
- 2      $H(0, x) = 0$
- 3 For  $1 \leq k \leq n$
- 4     For  $1 \leq x \leq c$
- 5         If  $x - w_k < 0$
- 6              $H(k, x) = H(k - 1, x)$
- 7         If  $x - w_k \geq 0$
- 8              $H(k, x) = \max\{H(k - 1, x - w_k) \cdot u_k, H(k - 1, x), u_k\}$
- 9 Return  $H(n, c)$

The running time of this algorithm is  $O(c \cdot n)$ .

- (d) [3 points] Modify your pseudo-code in part (c) to output the actual set of gifts with maximum utility that Santa packs in his sleigh.

**Solution:** The following pseudo-code takes as input the table  $H$  computed in part (c) and variables  $k, n$  and outputs a set of gifts that yield utility  $H(k, n)$ .

OUTPUT-GIFTS( $H, k, x$ )

- 1 If  $k = 0$
- 2     Output  $\emptyset$
- 3 Else If  $H(k, x) = H(k - 1, x - w_k) \cdot u_k$
- 4     Output  $g_k$
- 5     OUTPUT-GIFTS( $H, k - 1, x - w_k$ )
- 6 Else if  $H(k, x) = H(k - 1, x)$
- 7     Output OUTPUT-GIFTS( $H, k - 1, x$ )
- 8 Else if  $H(k, x) = u_k$
- 9     Output  $g_k$

**Problem -4. Detecting Costly Cycles [18 points]**

You are given a weighted, directed graph  $G = (V, E)$ , in which each edge has a weight between 0 and 1, i.e. for all  $(i, j) \in E$ ,  $0 \leq w(i, j) \leq 1$ . We say a directed cycle with  $c$  edges is *costly* if the sum of the weights of the edges on the cycle is more than  $c - 1$ .

- (a) [10 points]** Give an  $O(V^3)$  algorithm to find the minimum-cost directed cycle in  $G$ .  
 (Assume the graph  $G$  contains no self-loops, so a directed cycle contains at least two edges.)

**Solution:** Run FLOYD-WARSHALL to find all pairs shortest paths. Then for every  $i, j \in V$ , find the shortest path from  $i$  to  $j$  (i.e.  $d_{ij}^{(n)}$ ) and the shortest path from  $j$  to  $i$  (i.e.  $d_{ji}^{(n)}$ ). These two paths make up the minimum cost cycle that goes through vertices  $i$  and  $j$ . Thus, if we find the minimum-cost cycle that goes through  $i, j$  for all  $i, j \in V$ , then the minimum of these is the minimum-cost cycle in the graph. Below is pseudo-code for finding the cost of the minimum-cost cycle in a directed graph  $G$ .

```

MIN-COST-CYCLE( $G$ )
1 Run FLOYD-WARSHALL on  $G$ 
2  $c \leftarrow \infty$ 
3 For all pairs  $i \neq j \in V$ 
4   If  $d_{ij}^{(n)} + d_{ji}^{(n)} < c$ 
5      $c = d_{ij}^{(n)} + d_{ji}^{(n)}$ 
6 Return  $c$ 

```

Common errors included running FLOYD-WARSHALL and then finding the minimum  $d_{ii}^{(n)}$  for all  $i \in V$ . This does not work if you use FLOYD-WARSHALL as implemented in CLRS because  $d_{ii}^{(1)}$  is initialized to 0 and since  $G$  contains no negative cycles,  $d_{ii}^{(n)}$  will be 0 for all  $i \in V$ . This approach could work if you run FLOYD-WARSHALL and initialize  $d_{ii}^{(1)} = \infty$  for all  $i \in V$ . However, for full credit for this answer, you needed to argue why this implementation of FLOYD-WARSHALL is correct.

Another common incorrect solution was to run DFS and “find all cycles”, sort them according to cost and return the minimum. Note that there may be exponentially many cycles, so any algorithm that finds all cycles is not efficient.

- (b) [8 points]** Give an  $O(V^3)$  algorithm to determine whether  $G$  contains a *costly* cycle. Argue that your algorithm is correct and analyze its running time. (**Hint:** Use part **(a)** on a modified graph.)

**Solution:** Consider the graph  $G'$  in which every edge has weight  $w'(i, j) = 1 - w(i, j)$ . If any cycle in  $G$  with  $c$  edges has value greater than  $c - 1$ , then the same cycle in  $G'$  has weight less than 1. Therefore, we simply need to find the minimum-cost cycle in  $G'$ . If the minimum-cost cycle is less than 1, then  $G$  contains a costly cycle. Finding a minimum-cost cycle in  $G'$  can be done in  $O(V^3)$  time using part **(a)**. Correctness follows from the following claim. A cycle in  $G$  is costly iff the corresponding cycle in  $G'$  has cost less than 1. In other words, consider some cycle  $C$  which contains a subset of  $c$  edges in  $G$ . Then we have:

$$\sum_{ij \in C} w_{ij} > c - 1 \iff \sum_{ij \in C} (1 - w_{ij}) = c - \sum_{ij \in C} w_{ij} < c - (c - 1) = 1.$$

Most people used the correct modification of the graph, since that was a hint given in problem **1(l)**. However, many people incorrectly argued correctness by claiming that a minimum-cost cycle in  $G'$  corresponds to a *maximum*-cost cycle in  $G$ . This is not true: a minimum-cost cycle in  $G'$  corresponds to a cycle in  $G$  which has the smallest difference between its cost and its cardinality, but it might not be the maximum-cost cycle.

## Quiz 2 Solutions

- Do not open this quiz booklet until you are directed to do so. Read all the instructions first.
- When the quiz begins, write your name on every page of this quiz booklet.
- The quiz contains **four** multi-part problems. You have 120 minutes to earn 120 points.
- This quiz booklet contains **18** pages, including this one. An extra sheet of scratch paper is attached.
- This quiz is closed book. You may use one handwritten A4 or  $8\frac{1}{2}'' \times 11''$  crib sheet. No calculators or programmable devices are permitted.
- Write your solutions in the space provided. Extra scratch paper may be provided if you need more room, although your answer should fit in the given space.
- Do not waste time and paper re-deriving facts that we have studied. It is sufficient to cite known results.
- 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. Generally, a problem's point value is an indication of how much time to spend on it.
- 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.
- Good luck!

Problem	Points	Grade	Initials
1	30		
2	35		
3	25		
4	30		
Total	120		

Name: Solutions \_\_\_\_\_  
Circle your recitation:

Brian 11

Brian 12

Jen 12

Jen 1

Brian 2

**Problem 1. True or False, and Justify** [30 points] (7 parts)

Circle **T** or **F** for each of the following statements, and briefly explain why. The better your argument, the higher your grade, but be brief. Your justification is worth more points than your true-or-false designation.

- (a) **T F** [4 points] A 2-3-4 tree is special case of a B-tree where leaves can have different depths.

**Solution:** False. In a 2-3-4 tree, all leaves have the same depth.

- (b) **T F** [4 points] Searching on a skip list takes expected  $\Theta(\log n)$  time, but could take  $\Omega(n)$  time with non-zero probability.

**Solution:** True. A skip list could be of any height or be a simple linked list, depending on its random choices.

- (c) **T F** [4 points] To determine whether two binary search trees on the same set of keys have identical tree structures, one could perform an inorder tree walk on both and compare the output lists.

**Solution:** False. An inorder tree walk will simply output the elements of a tree in sorted order. Thus, an inorder tree walk on any binary search tree of the same elements will produce the same output.

- (d) **T F** [4 points] In an undirected weighted graph with distinct edge weights, both the lightest and the second lightest edge are in some MST.

**Solution:** True. First, since the edge weights are distinct there is only a single MST. Let  $e_1$  and  $e_2$  be the lightest and second lightest edge. In Kruskal's algorithm,  $e_1$  is always the first edge added. Since  $e_2$  cannot possibly create a cycle, it will necessarily be the next edge added by Kruskal's algorithm. So, by the correctness of Kruskal, the two lightest edges are always in the MST.

- (e) **T F** [5 points] Dijkstra's algorithm works correctly on graphs with negative-weight edges, as long as there are no negative-weight cycles.

**Solution:** False. Consider the directed graph  $V = \{A, B, C\}$  and  $E = \{AB, AC, CB\}$ . Let  $w(AB) = 1$ ,  $w(AC) = 2$  and  $w(CB) = -2$ . Dijkstra will first add the edge  $AB$  to the shortest path tree. However, the shortest path tree would be  $T = \{AC, CB\}$ .

- (f) **T F** [5 points] In a graph  $G = (V, E)$ , suppose that each edge  $e \in E$  has an integer weight  $w(e)$  such that  $1 \leq w(e) \leq n$ . Then there is a an  $o(m \log n)$ -time algorithm to find a minimum spanning tree in  $G$ .

**Solution:** True. Run Kruskal using Counting sort. Sorting edges will take  $O(m + n) = O(m)$ . The remainder of Kruskal's algorithm will take  $O(m\alpha(m, n))$  time, which is  $o(m \log n)$ .

- (g) **T F** [4 points] In the comparison model, there is an  $\Omega(m \log n)$  lower bound on performing  $m$  UNION and FIND-SET operations in any disjoint-set data structure storing  $n$  total elements.

**Solution:** False. Using the Union-Find data structure from class, we can perform  $m$  operations on a set of initial size  $n$  in  $\Theta(m\alpha(m, n))$ .

**Problem 2. Short Answer** [35 points] (3 parts)

Give *brief*, but complete, answers to the following questions.

- (a) [10 points] We wish to define a universal hash family  $\mathcal{H} = \{h_1, h_2, h_3\}$  where  $U = \{a, b, c\}$  and  $h_i : U \rightarrow \{0, 1\}$ . Complete the following table to make  $\mathcal{H}$  universal. The function  $h_1$  has already been defined for you:

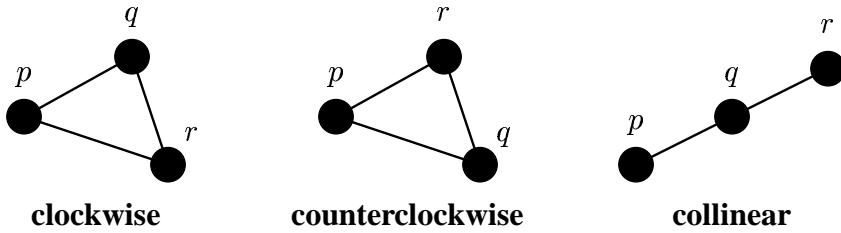
	$a$	$b$	$c$
$h_1$	0	0	1
$h_2$			
$h_3$			

**Solution:**

We need to ensure that none of the three pairs  $(a, b)$ ,  $(a, c)$  or  $(b, c)$  collide with more than  $1/2$  probability. Since  $a$  and  $b$  already collide once, we must have that  $h_2(a) \neq h_2(b)$  and  $h_3(a) \neq h_3(b)$ . Then they will collide with probability  $1/3$ . The value  $c$  can collide with each of  $a$  and  $b$  exactly once, so we can have  $h_2(a) = h_2(c)$  and  $h_2(b) = h_2(c)$ . One valid solution is:

	$a$	$b$	$c$
$h_1$	0	0	1
$h_2$	0	1	0
$h_3$	0	1	1

- (b) [10 points] Suppose that you are given a set  $\{p_1, p_2, \dots, p_n\}$  of  $n$  points in two dimensions. Give an  $O(n^2 \lg n)$ -time algorithm to detect whether any three points are *collinear*, that is, whether any three points lie on a common line. You may assume that you have a subroutine that computes in  $O(1)$  time whether three given points  $p, q, r$  are oriented clockwise, oriented counterclockwise, or collinear, as shown in the figure below. (Such an “orientation test” subroutine was given in lecture.) You may not use hashing.



**Solution:**

**for** each point  $x$ :

Use the orientation test as a comparator to sort the other  $n - 1$  points radially with respect to  $x$ .

If any two points are collinear with respect to  $x$ , **return** True.

**return** False

As shown in class, we will use the clockwise/counterclockwise/collinear test as a less/greater/equal comparator. We can sort the points with respect to a given  $x$  in  $O(n \log n)$  time. If any triplet is collinear, it will be detected and the loop will halt. Since we loop through every possible  $x$ , this algorithm will take  $O(n^2 \log n)$  time.

- (c) [15 points] You are given three bags of steel pipes, denoted  $A$ ,  $B$  and  $C$ , such that each bag contains  $n$  pieces of pipe with lengths in the range  $[0, 5n]$ . You may assume that each piece of pipe has a unique integer length. You may attach a single pipe  $a \in A$  to a single pipe  $b \in B$  and to a single pipe  $c \in C$  to produce a pipe with length  $a + b + c$ .

Give an  $O(n \log n)$ -time algorithm to determine both:

- Every possible length that may be obtained by attaching a triplet of pipes from  $A$ ,  $B$  and  $C$ .
- The number of triplets that could be attached to produce a given length.

**Solution:**

Represent  $A$ ,  $B$  and  $C$  as polynomials of degree  $5n$  as follows:

$$A(x) = x^{a_1} + x^{a_2} + \dots + x^{a_n}, B(x) = x^{b_1} + x^{b_2} + \dots + x^{b_n}, C(x) = x^{c_1} + x^{c_2} + \dots + x^{c_n},$$

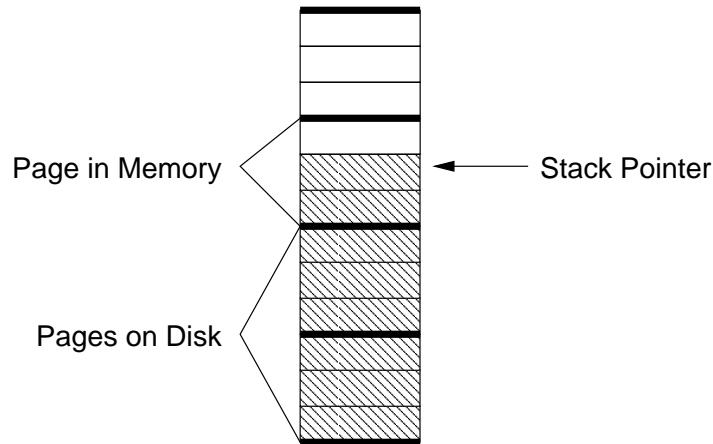
Multiply  $A$ ,  $B$ , and  $C$  in time  $\Theta(n \log n)$  using the FFT to obtain a coefficient representation  $d_0, d_1, \dots, d_{3n}$ . In other words  $D(x) = d_0 + d_1x + d_2x^2 + \dots + d_{3n}x^{3n}$ . Each triplet  $a_i, b_j, c_k$  will account for one term  $x^{a_i}x^{b_j}x^{c_k} = x^{a_i+b_j+c_k} = x^m$ . Therefore,  $d_m$  will be the number of such pairs  $a_i + b_j + c_k = m$ .

**Problem 3. Amortized Stack Analysis** [25 points] (2 parts)

Recall the standard implementation of a stack in an array. A *stack pointer* stores the address of the top element of the stack. The  $\text{PUSH}(x)$  operation increments the stack pointer, then stores a new element  $x$  at the top of the stack. The  $\text{POP}$  operation returns the element at the top of the stack, then decrements the stack pointer.

In this problem, we wish to implement a stack on a computer with a small amount of fast memory and a large amount of slow disk space. The disk space is partitioned into  $p$ -element clusters called *pages*. The computer can read or write individual elements in memory, but can read or write only entire pages on disk. Reading or writing an element in memory takes  $\Theta(1)$  time while reading or writing a page on disk takes  $\Theta(p)$  time.

Consider a stack that stores the top page of the stack array in memory and the remainder on disk, as in the following diagram:



In this diagram, each page has  $p = 3$  words. Whenever the stack pointer crosses a page boundary (indicated by the thick lines) there is a disk operation that incurs  $\Theta(p)$  cost.

- (a) [10 points] What is the worst-case running time to perform  $n$  stack operations using this implementation? Express your solution in terms of  $n$  and  $p$  throughout this problem.

**Solution:** Consider the alternating sequence

PUSH, PUSH, POP, POP . . .

occurring at a page boundary. The second PUSH requires writing the first word of the next page, and the second POP requires reading in the previous page again. Thus,  $n$  operations can make  $\Theta(n)$  disk accesses, taking a total of  $\Theta(np)$  time.

- (b) [15 points] Suppose you are allowed to store two stack pages in memory. Implement PUSH and POP so that the amortized running time for any stack operation is  $O(1)$ . You may assume that you have  $O(1)$  extra memory for bookkeeping.

**Solution:**

Keep the page currently needed in memory, as in part (a), but also keep the previously used page. Keep a bit marking which of the two pages in memory has been least recently used (LRU). Whenever a new page must be read in, save the LRU page to disk if modified, and read in the new page. When executing many PUSH operations, memory will first be  $p_0$  (only one page valid), then  $p_0p_1$  (while doing the  $(p + 1)$ st through  $(2p)$ th pushes), then  $p_2p_1$ , and then  $p_2p_3$ .

The stack pointer will point to the top of any fresh page read from disk. In other words, there is  $p$  data and  $p$  free space available in memory for our stack operations. We must either perform  $p$  PUSH or  $p$  POP operations before the next page must be read.

Therefore each disk access must be preceded by  $p$  stack operations. Using the accounting method, pay  $1/p$ th of our disk access cost, i.e.  $\Theta(1)$  for every stack operation. After  $p$  stack operations, we will have accumulated  $\Theta(p)$  value that can pay for any necessary disk accesses. Therefore, stack operations have a  $\Theta(1)$  amortized cost.

**Problem 4.** Package Shipping [30 points] (3 parts)

You work for a small manufacturing company and have recently been placed in charge of shipping items from the factory, where they are produced, to the warehouse, where they are stored. Every day the factory produces  $n$  items which we number from 1 to  $n$  in the order that they arrive at the loading dock to be shipped out. As the items arrive at the loading dock over the course of the day they must be packaged up into boxes and shipped out. Items are boxed up in contiguous groups according to their arrival order; for example, items 1 . . . 6 might be placed in the first box, items 7 . . . 10 in the second, and 11 . . . 42 in the third.

Items have two attributes, *value* and *weight*, and you know in advance the values  $v_1 \dots v_n$  and weights  $w_1 \dots w_n$  of the  $n$  items. There are two types of shipping options available to you:

**Limited-Value Boxes:** One of your shipping companies offers insurance on boxes and hence requires that any box shipped through them must contain no more than  $V$  units of value. Therefore, if you pack items into such a “limited-value” box, you can place as much weight in the box as you like, as long as the total value in the box is at most  $V$ .

**Limited-Weight Boxes:** Another of your shipping companies lacks the machinery to lift heavy boxes, and hence requires that any box shipped through them must contain no more than  $W$  units of weight. Therefore, if you pack items into such a “limited-weight” box, you can place as much value in the box as you like, as long as the total weight inside the box is at most  $W$ .

Please assume that every individual item has a value at most  $V$  and a weight at most  $W$ . You may choose different shipping options for different boxes. Your job is to determine the optimal way to partition the sequence of items into boxes with specified shipping options, so that shipping costs are minimized.

- (a) [10 points] Suppose limited-value and limited-weight boxes each cost \$1 to ship. Describe an  $O(n)$  greedy algorithm that can determine a minimum-cost set of boxes to use for shipping the  $n$  items. Justify why your algorithm produces an optimal solution.

**Solution:**

We use a greedy algorithm that always attempts to pack the largest possible prefix of the remaining items that still fits into some box, either limited-value or limited-weight. The algorithm scans over the items in sequence, maintaining a running count of the total value and total weight of the items encountered thus far. As long as the running value count is at most  $V$  or the running weight count is at most  $W$ , the items encountered thus far can be successfully packed into some type of box. Otherwise, if we reach a item  $j$  whose value and weight would cause our counts to exceed  $V$  and  $W$ , then prior to processing item  $j$  we first package up the items scanned thus far (up to item  $j - 1$ ) into an appropriate box and zero out both counters. Since the algorithm spends only a constant amount of work on each item, its running time is  $O(n)$ .

Why does the greedy algorithm generate an optimal solution (minimizing the total number of boxes)? Suppose that it did not, and that there exists an optimal solution different from the greedy solution that uses fewer boxes. Consider, among all optimal solutions, one which agrees with the greedy solution in a maximal prefix of its boxes. Let us now examine the sequence of boxes produced by both solutions, and consider the first box where the greedy and optimal solutions differ. The greedy box includes items  $i \dots j$  and the optimal box includes items  $i \dots k$ , where  $k < j$  (since the greedy algorithm always places the maximum possible number of items into a box). In the optimal solution, let us now remove items  $k + 1 \dots j$  from the boxes in which they currently reside and place them in the box we are considering, so now it contains the same set of items as the corresponding greedy box. In so doing, we clearly still have a feasible packing of items into boxes and since the number of boxes has not changed, this must still be an optimal solution; however, it now agrees with the greedy solution in one more box, contradicting the fact that we started with an optimal solution agreeing maximally with the greedy solution.

- (b)** [20 points] Suppose limited-value boxes cost  $\$C_v$  and limited-weight boxes cost  $\$C_w$ . Give an  $O(n^2)$  algorithm that can determine the minimum cost required to ship the  $n$  items, and briefly justify its correctness.

**Solution:**

We use dynamic programming. Let  $A[j]$  denote the optimal cost for packing just items  $j \dots n$  into boxes. We compute the sequence of subproblem solutions  $A[n] \dots A[1]$  in reverse order as follows (we take  $A[n+1] = 0$  as a base case). For every item  $i$ , let  $V(i)$  be the largest item index such that  $v_i + v_{i+1} + \dots + v_{V(i)} \leq V$ , and let  $W(i)$  be the largest item index such that  $w_i + w_{i+1} + \dots + w_{W(i)} \leq W$ . We can compute  $V(i)$  and  $W(i)$  for a item  $i$  by scanning forward from  $i$  and maintaining a running count of the total value and weight from item  $i$  onward. Applying this to every item, we can compute  $V(i)$  and  $W(i)$  for all items in  $O(n^2)$  time. We can now easily compute solutions to our DP subproblems via the following formula:

$$A[i] = \min(A[V(i) + 1] + C_v, A[W(i) + 1] + C_w)$$

The two cases above correspond to the decision of whether or not to use a limited-value or limited-weight box as the first box when packing items  $i \dots n$  in sequence. Regardless of the type of box we select, we clearly want to place as many items as possible in that box — this follows from the greedy proof above. The DP algorithm requires only  $O(n)$  time after computing the  $V(i)$ 's and  $W(i)$ 's, which requires  $O(n^2)$  time.

Some student suggested a greedy solution in this part: find the largest prefixes for each the limited-value box and limited-weight box, and choose the option that results in the lowest cost per item. However, this greedy strategy does not give an optimal solution in the following example.

Suppose  $W = 5$ ,  $V = 5$ ,  $C_v = 2$  and  $C_w = 5$ . Consider the 5 items with the following  $(v_i, w_i)$  values:

$$(5, 3), (5, 2), (3, 2), (1, 1), (1, 2)$$

The greedy algorithm will pick VLB, WLB and VLB in the above order, costing a total of \$9. On the other hand, the optimal solution should choose a WLB followed by a VLB, with a total cost of \$7. Therefore, the greedy algorithm does not always give an optimal solution.

- (c) [3 points] **(Extra Credit)** Can you reduce the running time of the algorithm from (b) to  $O(n)$ ?

**Solution:**

We can speed up the algorithm above by computing the  $V(i)$ 's and  $W(i)$ 's in  $O(n)$  total time. Note that  $V(i)$  and  $W(i)$  are both monotonically non-decreasing functions of  $i$ . Therefore, as we scan forward for  $i = 1 \dots n$ , the values of  $V(i)$  and  $W(i)$  can only move forward as well. The following pseudocode illustrates how we can exploit this monotonicity to speed up the total computation:

```
1    $V(0) \leftarrow 0, TotalValue \leftarrow 0$ 
2    $W(0) \leftarrow 0, TotalWeight \leftarrow 0$ 
3   for  $i \leftarrow 1$  to  $n$ :
4        $V(i) \leftarrow V(i - 1), TotalValue \leftarrow TotalValue - v_{i-1}$ 
5        $W(i) \leftarrow W(i - 1), TotalWeight \leftarrow TotalWeight - w_{i-1}$ 
6       while  $TotalValue + v_{V(i)+1} \leq V$ :
7            $V(i) \leftarrow V(i) + 1, TotalValue \leftarrow TotalValue + v_{V(i)}$ 
8           while  $TotalWeight + w_{W(i)+1} \leq W$ :
9                $W(i) \leftarrow W(i) + 1, TotalWeight \leftarrow TotalWeight + w_{W(i)}$ 
```

## SCRATCH PAPER

## SCRATCH PAPER

## SCRATCH PAPER

## Quiz 2 Solutions

### Problem 1. Faulty Turbines

You live in Windyland where the winds are *always* blowing at 50 miles an hour. To harness the energy of this windy bonanza, Windyland has laid out  $N^2$  windturbines on an  $N \times N$  grid – that is, for each  $1 \leq i \leq N, 1 \leq j \leq N$ , a turbine is placed at location  $(i, j)$ . Unfortunately, some  $m$  (very few compared to  $N$ ) of these turbines are faulty. When the wind passes through at 50 miles per hour, a good turbine generates 1 mega-joule per second, while a faulty turbine generates only a 1/2 mega-joule per second. Windyland is trying to determine the location of the faulty turbines. To help this quest, they have a built-in test mechanism  $\text{TEST}(i, j)$  that tells them the total amount of energy generated by the turbines in the set  $[1 \dots i] \times [1 \dots j]$ , where they get to specify  $i, j \in [1 \dots N]$ . However every  $\text{TEST}$  requires them to shut down the entire grid for an hour and is thus extremely expensive to run.

Give an algorithm that Windyland could use to find all the faulty turbines using as few  $\text{TESTS}$  as possible. Specify the running time of your algorithm as a function of  $m$  and  $N$ . Your algorithm should be efficient when  $m \ll N$ .

**Solution: Executive Summary.** Given an  $N \times N$  matrix of 0/1's with  $m$  ones, we try to find the ones efficiently. To solve the problem, we use binary search (more abstractly, divide-and-conquer). We obtain an algorithm that does  $O(m \log \frac{n^2}{m})$  tests and has the same running time.

**Detailed solution.** First we design a procedure  $\text{TEST-SUBRECT}(i_1, j_1, i_2, j_2)$ ,  $i_1 \leq i_2$  and  $j_1 \leq j_2$ , that returns the amount of power generated by the turbines in the rectangle  $(i_1, j_1, i_2, j_2)$ . This is done as follows:

$$\text{TEST-SUBRECT}(i_1, j_1, i_2, j_2) = \text{TEST}(i_2, j_2) - \text{TEST}(i_1 - 1, j_2) - \text{TEST}(i_2, j_1 - 1) + \text{TEST}(i_1 - 1, j_1 - 1).$$

(This is easy verifiable by a picture.)

Next imagine an abstract tree on the grid, which is in fact a *quad-tree*. The quad-tree is constructed as follows. Suppose, wlog,  $n$  is a power of 2. Divide the grid into 4 squares (of side-length  $n/2$ ). Then subdivide each square into another 4 squares (of side-length  $n/4$ ), and so forth. Now the quad-tree has as its root the entire grid, and it has 4 children, corresponding to the squares of side-length  $n/2$ . These squares also have 4 children each, each of side-length  $n/4$ , and so forth. At the leaf level, we have squares of side-length 1, i.e., they correspond to turbines.

The algorithm is now simple: design a procedure  $\text{SEARCH}(node)$  that returns all faulty turbines in the square corresponding to  $node$ . For this, test each of the 4 subtrees of  $node$  if they have faulty turbines (using  $\text{TEST-SUBRECT}$ ) and recurse into the subtrees that have.

To see the correctness and running time of this algorithm consider the following argument. Paint black all nodes that have faulty turbines inside. Then, the black nodes are exactly the union of the

paths from all faulty turbines to the root. There are thus  $m \log n$  black nodes (the depth of the tree is  $\log n$ ). We perform tests on the black nodes and their children only.

A more attentive counting of the black nodes actually gives a bound of  $O(m \log \frac{n}{\sqrt{m}})$ . The reason is that previously we overcounted the black nodes, especially near the root. The number of black nodes is maximized when all the first  $\log_4 m$  level of the quad tree are black, and, in the rest, we have  $m$  disjoint black paths. In this case the number of black nodes is  $m + m(\log n - \log_4 m) = O(m \log \frac{n^2}{m})$ .

Thus the running time is  $O(m \log n^2/m)$ . Consequently, we can at most the same number of TESTs.

One can prove that the above number of TESTs is optimal. A easy bound of  $\Omega(\frac{m \log(n^2/m)}{\log m})$  can be obtained as follows. One can do the same lower bound for this problem as for the sorting: we draw a tree of queries, with each node having  $d+1$  children (depending on the answer of faulty turbines, which is a number in  $\{0, \dots, d\}$ ). Then, there are  $\binom{n^2}{m} = 2^{O(m \log n^2/m)}$  total leaves, and the lower bound is the log of that, divided by the log of the degree,  $m+1$ . To obtain the optimal lower bound, partition the array in to  $n^2/m$  parts, each with 1 faulty turbine, and prove that to uncover the turbine in each part takes  $\Omega(\log n^2/m)$ , and that in total one needs  $m$  times that (note that this the latter is not an immediate corollary).

**Grading (out of 30 points).** Most students gave the right idea of divide-and-conquer by partitioning in subrectangles and recursing into non-empty ones. Another option was to find the columns with faulty turbines and to find faulty turbines inside the columns (both with binary search). We gave full credit for  $O(m \log n)$  solutions.

Most students lost 7 points for not giving complete analysis of the running time. The following argument was insufficient: “to find one fault, we need  $O(\log n)$  time (because of binary search); and so, for  $m$  faults, we need  $m$  times that”. This is not an immediate implication, and one needed to justify this step (e.g., see above). Also, many students lost 1-4 points for either not specifying the computational running time (besides the number of tests performed) or having a computationally inefficient algorithm. Note that this was the requirement for all problems (as stated in the preamble). Otherwise, points were deducted for imprecise description of the algorithm (e.g., for not showing how to compute the number of faults in a general subrectangle). Solutions with  $\Omega(N)$  tests received at most 15 points.

## Problem 2. Tax Status

You live in a community of  $n$  people who have approached you for help with a tax question. They would like to find out how many of them will have to pay taxes, how many would receive tax credits (in this hypothetical example we assume that the IRS does give money to people with sufficiently low income), and how many will neither owe taxes nor receive any credit.

The amount that someone pay in taxes is a monotone non-decreasing function  $f(x)$  of their income  $x$  (so  $f(x) > f(y)$  if  $x > y$ ). The IRS has provided you with software to compute  $f(x)$ , but in

their characteristic style (showing lack of proper 6.046 training), this software is extremely slow and takes  $\sqrt{n}$  time to compute the  $f(x)$  on any *single* input  $x$ .

You have available to you the incomes of all people in your community in the form of an *unsorted* array  $X[1 \dots n]$ , where  $X[i]$  is the income of the  $i$ th person. Give an efficient algorithm to compute an array  $Y[1 \dots n]$  where  $Y[i]$  indicates the tax status of person  $i$ , i.e.,  $Y[i] = +$  if the  $i$ th person owes taxes (i.e.,  $f(x) > 0$ ),  $Y[i] = -$  if the  $i$ th person is owed money by the IRS, and  $Y[i] = 0$  if the  $i$ th person neither owes taxes, nor is owed money by the IRS. Analyze the running time of your algorithm as a function of  $n$ .

You may assume all incomes are distinct.

**Solution: Executive Summary.** Search for the two “boundary” values of  $X$  in the community  $k, j$ , where  $k$  is the index of the person who makes the most money but is still owed money by the IRS, and  $j$  is the index of the person who makes the most money of those that neither owe taxes nor are owed money by the IRS. Then a linear scan through  $X$  allows you to determine the values of  $Y$ . In order to find the boundaries, find a query to  $X$ , using the linear time median algorithm, which allows you to successively rule out half of the remaining elements from consideration as possible candidates for the boundary.

### Detailed Solution.

In order to find the value of  $k$  (finding the value of  $j$  is analogous), the plan is to find a single query that allows you to rule out half of the indices from consideration. Find the median  $m$  of the  $X[i]$ ’s using the linear time median algorithm. Query  $f$  on  $X[m]$ . If  $X[m] \geq 0$ , then construct a new list which contains only those people for which  $X[i] \leq X[m]$  (by the monotonicity of  $f$ , the indices that have been thrown out must have  $X$  values which are  $\geq X[m] \geq 0$ , and therefore cannot be the boundary), otherwise, construct a new list which contains only those indices for which  $X[i] > X[m]$  (again, by the monotonicity of  $f$ , the indices that have been thrown out must have  $X$  values which are  $\leq X[m] < 0$  and therefore cannot be the boundary). Repeating this  $O(\log n)$  times brings you down to a list of constant size.

Once  $k, j$  have been found, scan through the indices of  $X$  and for each  $i$ , set the value of  $Y[i]$  by comparing  $X[i]$  to  $X[k]$  and  $X[j]$ . That is, set  $Y[i]$  to  $-$  if  $X[i]$  is smaller than or equal to  $X[k]$ , set  $Y[i]$  to  $0$  if  $X[k] < X[i] \leq X[j]$ , and set  $Y[i]$  to  $+$  otherwise.

Time analysis: Since half of the indices are ruled out from consideration after each query,  $O(\log n)$  queries to  $f$  will be made, taking  $O(\sqrt{n} \log n)$  time. The extra time to implement the search for each boundary value is given by  $T_b(n) = T(n/2) + c \cdot n$  which gives  $T_b(n) = O(n)$ . Once the two boundary values are found, the scan to set the values of  $Y$  takes linear time. The total time is  $T(n) = O(\sqrt{n} \log n + 2T_b(n) + n) = O(n)$ .

**Grading (out of 30 points).** Many sorted  $X$  and then did two binary searches for the boundaries. This requires  $\theta(n \log n)$  time, which is slower. Done correctly, this solution received 20 points.

Some used randomized selection. Depending on how well the analysis was done (which is harder than the deterministic case), up to 27 points were given for such a solution.

One common mistake was to give a recurrence for  $T(n)$  in which the cost of the queries to  $f$  was incorporated. The problem is that the cost of a query is  $\sqrt{n_0}$  for  $n_0$  the original number of people in the community, and does not decrease with the level. One point was taken off for this type of mistake.

Up to three points were taken off for mishandling the case in which several people satisfy  $f(X[i]) = 0$ .

### Problem 3. Repair Work

A hurricane just hit Cambridge and wiped out all of the roads. You need to repair the roads to connect up all the public buildings as quickly as possible. Every road connects two buildings, and the time to repair the road between buildings  $i$  and  $j$  is  $t_{ij}$ , where  $t_{ij}$  is an integer between 1 and 10. Let  $T_{\text{upandrunning}}$  be the minimum time you need to get enough roads built to connect all the public buildings (you can only work on one road at a time – no parallelism here!).

Give an efficient algorithm to compute  $T_{\text{upandrunning}}$ . Assume there are  $n$  buildings and  $m$  roads, and that your input comes in the form of an array of  $n$  adjacency lists, where the  $i$ th list specifies all the roads incident to building  $i$ , the other endpoint for each road, and the time it takes to repair the road. Express the running time of your algorithm as a function of  $n$  and  $m$ . (The more efficient your algorithm, the better your score.)

**Solution: Executive Summary:** The problem can be restated as finding the weight of a minimum spanning tree in the underlying graph. Using that all the edge weights are small integers, we modify Prim's algorithm to run in time  $O(m)$ . This is achieved by using a different implementation of the Priority Queue inside that allows Extract-min and Decrease-Key operations in  $O(1)$  time.

**Detailed Solution:** Let  $G = (V, E)$  be the graph obtained by letting  $V$  be the set of public buildings in Cambridge and  $E$  be the set of roads. Assign to every road  $ij \in E$  the weight  $w(ij) = t_{ij}$ . Then  $T_{\text{upandrunning}}$  is simply the weight of a minimum spanning tree (MST) of  $G$ . We can find an MST of  $G$  using either Prim or Kruskal's algorithm. In the implementation seen in lecture, the running time of Prim is  $O(nT_{\text{Extract-min}} + mT_{\text{Decrease-Key}})$ , where  $n = |V|$ ,  $m = |E|$ , and  $T_{\text{Extract-min}}$  and  $T_{\text{Decrease-Key}}$  are the time needed by the respective operation in the Priority Queue, which is used to keep track of the vertices not yet covered by the tree so far. Recall that the priority of a vertex in the Priority Queue is equal to the weight of the smallest edge connecting the vertex to any vertex in the tree constructed so far (or “ $\infty$ ” if none exists).

In this problem, we are given an additional constraint that the edge weights are small integers. Using this, we can construct a Priority Queue where both extract-min and decrease-key operations take  $O(1)$ . There are many ways to do this. One way is to keep 2 arrays  $A$  and  $B$ .  $A$  is an array of length 11, where, for  $1 \leq i \leq 10$ ,  $A[i]$  contains a doubly linked list containing all the vertices with priority  $i$ , and  $A[11]$  contains a doubly linked list with the vertices with infinite priority.  $B$ , on the other hand, is an array of size  $n = |V|$  such that for every  $v \in V$ ,  $B[v]$  has a pointer to the node corresponding to the vertex  $v$  in the linked list of  $A$  (or a null pointer if  $v$  is not in the priority queue anymore).

To extract the minimum element of the queue, we can do the following: Find the smallest  $i$  for which  $A[i]$  is not empty. Remove the first element in the linked list in  $A[i]$  (call this element  $v$ ). Then, set  $B[v]$  to be a null pointer and output  $v$ . Since  $A$  is of size 11, this takes  $O(1)$  time.

To change the priority of some vertex  $v$  from  $k'$  to  $k$  in the priority queue, we simply use  $B[v]$  to locate  $v$  in its corresponding doubly linked list. Then, we remove  $v$  from the list of  $A[k']$  (this can be done in constant time by changing the pointer of the predecessor and successor of the node in the list). Then, we insert  $v$  to the beginning of the linked list in  $A[k]$  and modify  $B[v]$  to point the corresponding node associated to  $v$ . This also takes  $O(1)$  time.

Thus, both  $T_{\text{Extract-min}}$  and  $T_{\text{DecreaseKey}}$  run in constant time, and therefore, Prim's algorithm takes time  $O(nT_{\text{Extract-min}} + mT_{\text{DecreaseKey}}) = O(n + m) = O(m)$ .

**Grading comments:** Any solution achieving this running time received 30 points. Many solutions were of this flavor with some mistakes. The most common mistake was to have only array  $A$  without  $B$  to help search in the priority queue. This kind of solution did not take into account the time needed to find an element in the priority queue  $A$  in order to decrease its key. This mistake was penalized with 8 points.

There were some alternative correct solutions. The most common one was an implementation of the previous priority queue  $A$  with arrays instead of linked list and an implementation of a priority queue without decrease-key method (Simply reinsert the elements to the queue with the new (smaller) priority, and modify the extract-min method to just check if the element had been extracted before or not. A careful analysis shows that the running time of Prim in this case is still  $O(m)$ ). Also, there were some solutions that used a modification of Prim in which the priority queue holds edges instead of vertices.

Any solution that applied a direct implementation of Prim using binary heaps or a direct application of Kruskal (giving a running time of  $O(m \log n)$ ) received 15 points. Better implementations that used Fibonacci Heaps (with running time  $O(m + n \log n)$ ) or used Counting-Sort to sort the edges as the first step in Kruskal (giving a running time  $O(m\alpha(n))$ , where  $\alpha(\cdot)$  is the inverse of the Ackermann function) received up to 20 points.

#### Problem 4. Brady Bunch Marriage

In a small farming village there lives an old man with  $n$  sons, and on the farm next to his there lives an old woman with  $n$  daughters. In fact, they are the only two families remaining on the planet after the Bubonic plague wiped out everyone else in the world. Both the man and woman would like very much to have grandchildren in order to re-populate the Earth, but none of their children are yet married. Clearly, the only way for them to have grandchildren is to intermarry their children between their two families. But before they start making matches, the old man and woman agree on the following rule:

*If son A marries daughter B, then no son younger than son A may marry a daughter older than daughter B, and no son older than son A may marry a daughter younger than daughter B.*

This rule prevents age-crossings in marriages between the two families, which the old man and woman are afraid may lead to infidelity, tearing the two families apart (and thus endangering the future of our entire species).

In addition, village records over the last 14 generations show that the number of children a couple has is affected by the couple's height difference—the closer in height the husband and wife are, the more kids they can expect to have! If they are more than 12 inches apart, they will not have any children. Before he died of Bubonic plague, the village statistician found the exact formula for the expected number of children that a couple with a height difference of  $d$  inches (in absolute value), would have. This quantity, denoted by  $C(d)$ , is given by the following formula:

$$C(d) = \begin{cases} \frac{12-d}{2} & \text{if } d < 12 \\ 0 & \text{otherwise} \end{cases} \quad (1)$$

The old man and woman would like to maximize the expected number of grandchildren their children will give them. Your task is to give an efficient algorithm (in the number of children in each family,  $n$ ), to help them find the best way to intermarry their children (where best is defined as yielding the highest expected number of grandchildren). Assume that the algorithm is given as input four arrays:

- $A[1 \dots n]$  where  $A(i)$  denotes the age of the  $i^{th}$  son.
- $B[1 \dots n]$  where  $B(i)$  denotes the age of the  $i^{th}$  daughter.
- $G[1 \dots n]$  where  $G(i)$  denotes the height of the  $i^{th}$  son.
- $H[1 \dots n]$  where  $H(i)$  denotes the height of the  $i^{th}$  daughter.

Note that it may be preferable to leave some children unmarried for the social good. Be sure to prove the correctness and running time of your algorithm—the fate of humanity rests in your hands!

**Solution: Executive Summary:** We will solve this using Dynamic Programming. To find the recursive substructure, consider the oldest son. Either he can marry the oldest daughter, or he can marry one of the  $n - 1$  younger daughters, or he can not marry anyone. Each of these cases reduces to a smaller subproblem, in a matrix of  $n \times n$  subproblems. Since each problem only depends on (at most) 3 subproblems, DP will find the optimal matching in  $\Theta(n^2)$  time.

**Detailed Solution:** At first, sort sons and daughters in increasing order of their ages. It takes  $\Theta(n \log n)$  time, and now we can assume  $A$  and  $B$  are sorted.  $T$  is a  $n \times n$  matrix, where the element  $T(i, j)$  represents the highest expected number of grandchildren by considering only the youngest  $i$  sons and the youngest  $j$  daughters. (Our goal is to compute  $T(n, n)$ .) To find the recursive formula for  $T(i, j)$ , take the  $i^{th}$  youngest son and consider 3 possible options he can choose when the youngest  $i$  sons and the youngest  $j$  daughters are intermarried. Either he can marry the  $j^{th}$  daughter, or he can marry one of  $j - 1$  younger daughters, or he cannot marry anyone. The best way for his choice can be decided by comparing these three cases can be expressed as the

following recursion:

$$T(i, j) = \max \left\{ \begin{array}{l} \frac{12 - |G(i) - H(j)|}{2} + T(i - 1, j - 1), \\ T(i, j - 1), \\ T(i - 1, j) \end{array} \right\}.$$

Using this recursion, the table  $T$  can be filled up starting from the base case  $T(i, 0) = T(0, i) = 0$ , and  $T(n, n)$  is the highest expected number of grandchildren achievable. To find the explicit marriage strategy that gives  $T(n, n)$ , we maintain another  $n \times n$  matrix  $S$ . By assuming  $S(i, 0) = S(0, i) = \emptyset$ ,  $S(i, j)$  can be computed together with  $T(i, j)$  as follows:

$$S(i, j) = \left\{ \begin{array}{ll} (i, j) \cup S(i - 1, j - 1), & \text{if } T(i, j) = \frac{12 - |G(i) - H(j)|}{2} + T(i - 1, j - 1) \\ S(i, j - 1), & \text{if } T(i, j) = T(i, j - 1) \\ S(i - 1, j), & \text{if } T(i, j) = T(i - 1, j) \end{array} \right..$$

$S(n, n)$  is the best way we are looking for, and it takes  $\Theta(n^2)$  time to fill up both  $S$  and  $T$  completely. Note that to fill up  $S$  in  $\Theta(n^2)$  time we must either build the lists up using pointers to smaller lists, rather than copying the sub-list into the bigger cell, or we must store directions in  $S$ : “EAST”, “SOUTHEAST”, and “SOUTH” which tell us which path to take through  $S$  after we fill in the whole matrix—following these directions will allow us to read off the path (and thus the optimal marriages) in linear time.

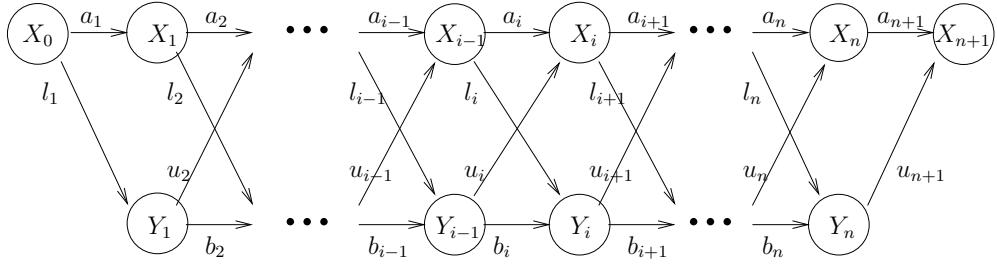
The total running time is  $\Theta(n^2)$  because the sorting time  $\Theta(n \log n)$  at the initial stage is dominated by  $\Theta(n^2)$ .

**Grading comments:** Many students designed Dynamic Programming with the  $n \times n$  table, but used a slow recursion which took  $\Theta(n)$  time to compute. It lead to the  $\Theta(n^3)$  running time, which got at most 23 points. Several solutions were naive or brute-force solutions, which got at most 7 points if their analysis was right. Any fundamentally incorrect algorithm got at most 10 points. (Some students used a greedy scheme, but it does not work.) Not writing down the recursion or making a small mistake in the recursion was a 3 to 5 point penalty. Incorrect analysis or no analysis at all took around 5 to 7 points off. Most implementations of the right  $\Theta(n^2)$  algorithm got 28 to 30 points depending on the clarity of the writeup.

### Problem 5. Dynamic Navigation

Recall (from Lecture 9) the nightmare that Professor Rubinfeld faces when driving to work every morning. She can either take the path from her home  $X_0$  to  $X_1 \rightarrow X_2 \rightarrow \dots \rightarrow X_n$  to her work  $X_{n+1}$ , or she can take the path from  $X_0 \rightarrow Y_1 \rightarrow Y_2 \rightarrow \dots \rightarrow Y_n \rightarrow X_{n+1}$ , or switch, as many times as she wants, from  $X_i \rightarrow Y_{i+1}$  or  $Y_i \rightarrow X_{i+1}$ . The delay in getting from  $X_{i-1} \rightarrow X_i$  is  $a_i$ , while the delay in getting from  $Y_{i-1} \rightarrow Y_i$  is  $b_i$ . The switching delay from  $X_i \rightarrow Y_{i+1}$  is  $\ell_i$  and the delay in switching from  $Y_i \rightarrow X_{i+1}$  is  $u_i$ . For an illustration, see figure 1.

Her goal was to get to work as quickly as possible and this is still the case. But now she has a new feature to help her cope with the nightmare. A generous non-profit institution is monitoring



**Figure 1:** The routes from Professor Rubinfeld's home,  $X_0$ , to her work,  $X_{n+1}$ .

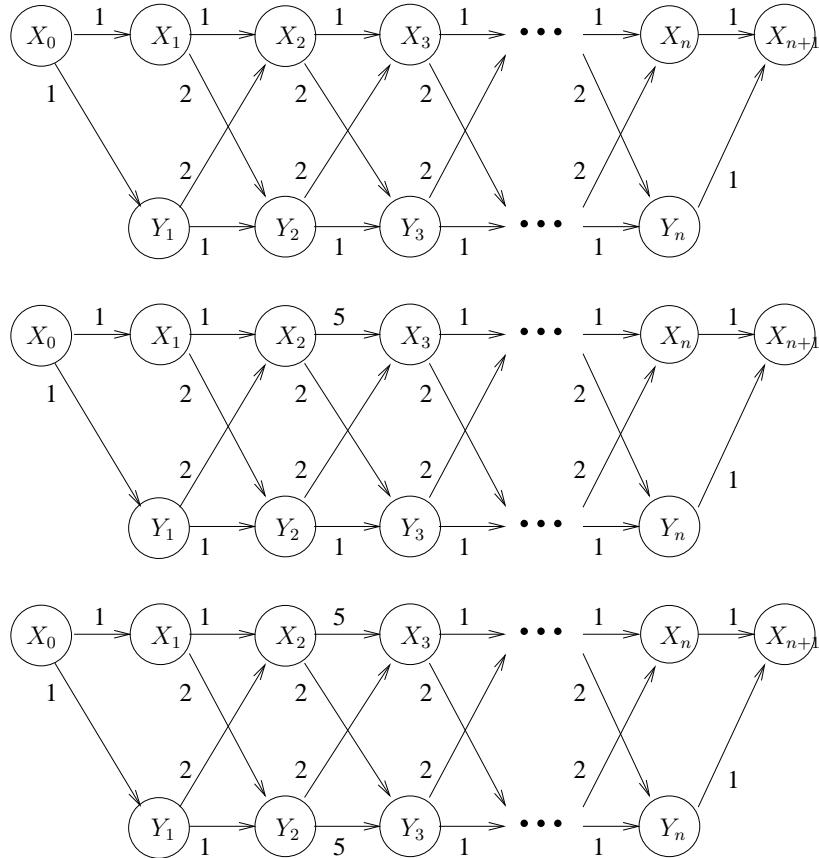
all the streets and can provide live updates on the current delays. A typical update is of the form  $\text{UPDATE}(U, W, i, v)$ , where  $U, W \in \{X, Y\}$  and the implication is that the associated street from  $U_{i-1} \rightarrow W_i$  now has a delay of  $v$ . These updates are beamed directly to her laptop, possibly even as she drives. Professor Rubinfeld would like to be able to execute two kinds of queries: **CURRENT-TRAVEL-TIME** which returns the current total travel time from  $X_0$  to  $X_{n+1}$ ; and **NEXT( $U, i$ )** where  $U \in \{X, Y\}$  which should return the next destination she should drive to to get to  $X_{n+1}$  by the shortest path, if she were currently at  $U_i$ .

Design a data structure to initialize the data structure and handle the updates and queries efficiently. (Your score will depend on the time complexity of the initialization, the updates, the queries, as well as the space complexity of the data structure. So specify all of these in your solution.)

The following example may be illustrative. Assume that initially all horizontal delays equal 1 (i.e.,  $a_i = b_i = 1$ ) and all switches cost 2 units of delay except for  $l_1 = u_{n+1} = 1$ , then the following table gives an example of a sequence of updates/queries and the desired responses. An illustration of the example appears in figure 2.

Query/Update	Desired Response
<code>INITIALIZE(<math>a_1, \dots, a_{n+1}, b_2, \dots, b_n, \ell_1, \dots, \ell_n, u_2, \dots, u_{n+1}</math>)</code>	ACK
<code>CURRENT-TRAVEL-TIME</code>	$n + 1$
<code>NEXT(<math>X, 2</math>)</code>	$X_3$
<code>UPDATE(<math>X, X, 3, 5</math>)</code>	ACK
<code>CURRENT-TRAVEL-TIME</code>	$n + 1$
<code>NEXT(<math>X, 2</math>)</code>	$Y_3$
<code>NEXT(<math>X, 3</math>)</code>	$X_4$
<code>UPDATE(<math>Y, Y, 3, 5</math>)</code>	ACK
<code>CURRENT-TRAVEL-TIME</code>	$n + 2$
<code>NEXT(<math>X, 2</math>)</code>	$Y_3$
<code>NEXT(<math>Y, 2</math>)</code>	$X_3$

**Solution: Executive Summary:** The problem asks to maintain shortest paths in a dynamic setting. The solution involves augmenting a data structure, a simple balanced binary tree, which maintains shortest path lengths of some, but not all, intervals from  $X_i/Y_i$  to  $X_j/Y_j$  so that changing



**Figure 2:** An example under the update shown in the table. The first picture shows the initial configuration, the second picture shows the configuration after the operation  $\text{UPDATE}(X, X, 3, 5)$ , and the third picture shows the configuration after the operation  $\text{UPDATE}(Y, Y, 3, 5)$ . For example, after we perform  $\text{UPDATE}(X, X, 3, 5)$  (second picture), it is now faster to switch from  $X_2$  to  $X_3$ .

any one edge length changes the path length of at most  $\log n$  of the intervals that we maintain. The resulting strategy is implemented below using  $O(n)$  space, so that **INITIALIZE** takes  $O(n)$  time, **UPDATE** and **NEXT** take  $O(\log n)$  time while **CURRENT-TRAVEL-TIME** takes  $O(1)$  time.

**Detailed Solution:** We maintain a nearly full balanced static binary search tree with  $n + 1$  leaves. The  $i$ th leaf represents the  $i$ th transition point from  $U_{i-1}$  to  $W_i$  for  $U, W \in \{X, Y\}$ . The tree is keyed on the index  $i$  and so every internal node represents a contiguous interval (or path) from  $i$  to  $j$ , such that its children represent the intervals  $i$  to  $k$  and  $k$  to  $j$ , where  $k = (i + j)/2$ . A node representing the interval  $i$  to  $j$  maintains the shortest path length from  $U_{i-1}$  to  $W_j$  for  $U, W \in \{X, Y\}$ . Note that this is information that can be computed locally given the information for a node's children. This will allow us to modify the tree in  $O(\log n)$  time during an update.

**INITIALIZE:** We build the tree bottom up. Every node  $v$ , representing say the interval  $[i, j]$  has four fields,  $v.XX$ ,  $v.XY$ ,  $v.YX$  and  $v.YY$  representing the shortest paths from  $X_i$  to  $X_j$  etc. If the node  $v$  has two children  $v_1$  covering the interval  $[i, k]$  and  $v_2$  covering the interval  $[k, j]$  then

the recurrence giving the four field values for  $v$  is as follows, for  $U, V \in \{X, Y\}$ :

$$v.UV = \min\{v_1.UX + v_2.XV, v_1.UY + v_2.YV\}.$$

It is clear that this algorithm runs in  $O(n)$  time.

**UPDATE( $U, W, i, v$ ):** We modify the cost of the edge  $U_{i-1} \rightarrow W_i$  to  $v$ , and then walk up the tree from the leaf  $i - 1 \rightarrow i$  to the root, updating the information at all nodes using the recurrence above. Clearly this takes  $O(\log n)$  time.

**CURRENT-TRAVEL-TIME:** Simply returns the value  $r.XX$  where  $r$  is the root node. This takes  $O(1)$  time.

**NEXT( $U, i$ ):** As a helper routine we first design an algorithm **TIME-REMAINING( $U, i$ )** which computes the length of the shortest path from  $U_i$  to  $X_{n+1}$ . This algorithm walks up from the leaf for  $i \rightarrow i + 1$  to the root. At a node  $v_1$  representing the interval  $[j, k]$  with  $j \leq i < k$  it maintains the information for the shortest path length from  $U_i$  to  $X_k$  and  $U_i$  to  $Y_k$ . It then uses the information from sibling  $v_2$  of  $v$  (if  $v_1$  is the left child) to compute this information at the parent  $v$  of  $v_1$ . When this algorithm reaches the root, it now has the shortest path length from  $U_i$  to  $X_{n+1}$ .

Now using **TIME-REMAINING** it is easy to compute the next step from  $U_i$ . One simply has to go to the node  $W_{i+1}$  for which the edge length from  $U_i$  to  $W_{i+1}$  equals  $\text{TIME-REMAINING}(U, i) - \text{TIME-REMAINING}(W, i + 1)$ . Determining this requires computing **TIME-REMAINING** thrice, where each invocation takes  $O(\log n)$  time. Thus **NEXT** takes  $O(\log n)$  time.

**Grading comments:** Unfortunately, a large number of solutions simply recomputed all path lengths, either during **UPDATE**, or during the queries. Such solutions are roughly analogous to the use of arrays (sorted or unsorted) to maintain dynamic sets — either the update, or the query, takes linear time. Such solutions got somewhere between 5 to 10 points. Some solutions additionally pointed out the possibility of doing better using balanced trees, without being able to give the algorithm. For noticing the possibility that one can do better, the writeups got up to 5 more points. Most implementations of the above algorithm got 25-30 points depending on the clarity of the writeup.

## Quiz 2

### Problem 1. Trip Planning with a Gas Tank

You want to drive from some location  $s$  to some location  $t$  without running out of gas while spending as little money as possible. You are given a map of the road system designated by a set of locations  $V$  and a set of directed roads  $E$ , where  $(u, v) \in E$  means that there is a road going from  $u \in V$  to  $v \in V$ . You know  $w(u, v)$ , the length of the road from  $u$  to  $v$ . You are also told which locations  $S \subseteq V$  have gas stations. When you come to a node  $u$  with a gas station, you have the option to fill up the tank of your car by paying  $c(u)$  dollars (the gas is free - you only pay for the parking).

Assuming you start with a full tank containing  $k$  units of gas, give an efficient algorithm that finds a cheapest route from  $s$  to  $t$  such that you never run out of gas.

Note: it is okay to run out of gas at the same moment you reach a gas station.

#### Solution:

Here is the best solution to the problem. A discussion of other solutions is given below. It is worth noting here that a greedy approach does not work because of the nature of the constraints.

#### Algorithm:

The fastest way we know to solve this problem is to build a new graph over the gas stations and run a standard shortest-path algorithm on that graph.

Specifically, we want to create a new graph  $G'$  whose vertices are the locations with gas stations plus  $s$  and  $t$ , i.e.  $V' = S \cup \{s, t\}$ . Let  $\delta(u, v)$  be the length of the shortest path between  $u$  and  $v$  in  $G$ . Now, an edge between  $u$  and  $v$  exists in  $G'$  if and only if  $\delta(u, v) < k$ , i.e. we can get from  $u$  to  $v$  on one tank of gas. Finally, the edge weights  $w'(u, v)$  in  $G'$  are the cost of filling up at  $v$ , i.e.  $w'(u, v) = c(v)$ .

Our algorithm will be as follows:

1. Construct  $G'$ .
2. Find the shortest path in  $G'$ .
3. Recover the path in  $G$  that is associated with the path found in  $G'$ .

There are many ways to do this with varying runtimes. The easiest solution is to use Floyd-Warshall to find the all-pairs shortest paths in  $G$ . Then, we can run Bellman-Ford on  $G'$ . Once we

have run Bellman-Ford, we can look back at the results from Floyd-Warshall to determine which partial paths corresponded to the path found in  $G'$ .

Floyd-Warshall requires  $O(V^3)$  time and  $O(V^2)$  space. Bellman-Ford will require  $O(VE)$  time and certainly  $O(V)$  space plus the space for  $G'$ . This will give an overall running time of  $O(V^3)$  and space requirement of  $O(V^2)$ .

We can speed this up by recognizing that neither distances nor costs are normally negative. Thus, we can reasonably assume that  $w(u, v) \geq 0$  and  $c(u) \geq 0$  for all  $u, v$ . As a result, we can find all-pairs shortest paths by running Dijkstra's algorithm  $|V|$  times (essentially Johnson's algorithm without reweighting.) We can make this faster by observing that we only really need to run Dijkstra's algorithm starting at each of the gas stations. Thus, the time to generate  $G'$  is the time to run  $|S|$  versions of Dijkstra's algorithm, which is  $O(S \cdot (E + V \log V))$  using the Fibonacci heap implementation. Finally, we can use Dijkstra's algorithm again to find the shortest path in  $G'$  in  $O(S^2 + S \log S) = O(S^2)$  time. The running time will be dominated by the computation of  $G$  for a total of  $O(S \cdot E + SV \log V)$ .

The space requirement when using Dijkstra's algorithm is a little more complicated. Dijkstra's algorithm inherently requires  $O(V)$  space. However, we need to somehow recover the path in  $G$ . The easiest way to do this is to store the results from all  $|S|$  original calls to Dijkstra's algorithm. This will require  $O(S \cdot V)$  space. This is probably the most practical approach from a time perspective, but you can actually reduce the space requirement to  $O(S^2 + V)$  with some tricks. Instead of saving the results from Dijkstra's algorithm, we discard the information not necessary to generate  $G'$ . The graph  $G'$  itself can easily be stored in  $O(S^2)$  space, so producing  $G'$  requires  $O(S^2 + V)$ . (Note that it is possible that  $|E'| > E$ , so the best bound we can give on  $|E'|$  is  $O(S^2)$ .) Now, to recover the path in  $G$ , we run Dijkstra's algorithm again to find the information we discarded. We will rerun Dijkstra's algorithm at most  $|S|$  times, so the asymptotic running time is not affected. Finally, the extra information we keep corresponds to a path. Consequently, it contains at most  $|V|$  nodes. Thus, we will need at most  $O(V)$  space for the extra information we find. This gives an overall space requirement of  $O(S^2 + V)$  (plus the space requirement for the input graph  $G$ ).

### Correctness:

To see that this is indeed correct, first observe that any path in  $G'$  corresponds directly to a path in  $G$ , since each edge itself corresponds to a path in  $G$ . Thus, if the cheapest route in  $G'$  is not a cheapest path, it must be more expensive than the cheapest route in  $G$ .

Now, let  $P$  be the cheapest route in  $G$ , and decompose it into subsections between gas stations at which we fill up the tank. Clearly, each subsection corresponds to a distance that can be traversed on one tank of gas, which implies that the section generates an edge in  $G'$ . It follows that each subsection of  $P$  has an edge in  $G'$  and therefore the entire path must have a counterpart in  $G'$ . Moreover, observe that the cost of a path in  $G'$  is exactly the gas stations along the route, since you pay  $c(u)$  on any incoming edge to  $u$  but never on an outgoing edge. Thus, each route in  $G$  corresponds to a route of equivalent cost in  $G'$ .

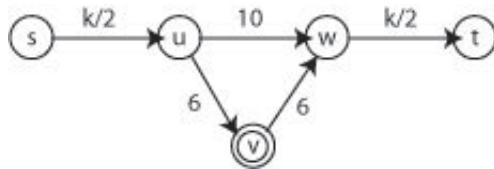
Therefore, the cheapest route in  $G'$  must also correspond to a route in  $G'$ , meaning that there is a path through  $G'$  with optimal cost.

**Comments and Other Ideas:**

Many people tried to copy each gas station so that there was one copy where the person filled up their gas tank and one where they didn't. (Either that, or they split a path when it reached a gas station.) This is handled in the above algorithm by allowing edges in  $G'$  to represent partial paths that go through gas stations.

There were also a handful of people who attempted to remove (one-by-one) all the vertices that were not gas stations. This is a valid idea, but the approaches presented did not have good analysis of the runtime necessary to perform this operation.

The insidiously tantalizing (yet wrong) approach to this problem is to modify Dijkstra's algorithm such that it ranks first by cost then by gas amount. The modification doesn't change the runtime or other properties of Dijkstra's, giving a possible  $O(E + V \log V)$  algorithm. Unfortunately, this doesn't work. Consider the following graph of 5 nodes (with one gas station at  $v$ ):



The greedy choice will pick the path to  $w$  avoiding the gas station at  $v$ , but won't have enough gas to get to  $t$ . Thus, the above algorithm won't work.

Another interesting approach uses the correct algorithm but replace Dijkstra's with a modification on BFS. The result is a slightly faster running time; however, it fails in a manner similar to the greedy approach. When you decide that a node is not reachable in this problem, you need to make sure you checked the shortest path to that node. This is not possible with BFS.

There was one significant alternative to constructing  $G'$ . If you assume that all distances are integers, then you can maintain a data structure at each node with  $k$  entries and change the notion of "visited" in Dijkstra's algorithm. (Alternatively, you can make  $k$  copies of each vertex to get a new graph with the same idea.) Now, you extend a path as long as the amount of gas in the tank is above the maximum already attained at that node. Then, since weights are integers, there are only  $k$  possible amounts of gas you can have in the tank, so you can go through each node at most  $k$  times. (Note that without the integer assumption, you may have  $2^{|E|}$  possible distinct amounts of gas, so this approach doesn't work anymore.) Apart from the assumption that  $k$  is an integer, the only problem with this algorithm is that  $k$  may be very large, resulting in high runtimes.

**Problem 2. Karp and Rabin compute the median**

Karp and Rabin each hold a set of  $n$  numbers, denoted by  $K$  and  $R$ , respectively. The sets  $K$  and  $R$  are disjoint. They would like to compute the median of the union set  $K \cup R$ . However, they live on different coasts, and they can communicate only by exchanging simple messages. Each message can either contain a number from their sets, or an integer in the range  $[1, n]$ .

Give an efficient communication protocol that Karp and Rabin can use to compute the median of the set  $K \cup R$ . The efficiency of the protocol is measured by the number of messages exchanged.

**Solution:** The most efficient solution to this problem uses  $O(\log n)$  messages in the worst case. There are many ways to achieve this. However, all efficient solutions have some “binary search flavor”. We note that a solution using  $O(n)$  messages can be achieved by transferring the whole data set to one of the parties and performing the computation locally.

Define  $\text{rank}_S(a)$  to be the number of elements in  $S$  that are smaller than  $a$ . Note that according to this definition, the smallest element has rank 0. The first observation is that, for any element  $a \in K$ , it is possible for Karp and Rabin to determine  $\text{rank}_{K \cup R}(a)$  using only constant number of messages. This is done as follows: Karp sends  $a$  to Rabin; Karp (or Rabin, respectively) compute the rank  $r_K(a)$  of  $a$  in  $K$  (or  $r_R(a)$  of  $a$  in  $R$ , respectively) using the local information; Karp and Rabin exchange the ranks and set the rank of  $a$  in  $K \cup R$  to be  $r_K(a) + r_R(a)$ .

According to the definition in CLRS, p. 183, the median of  $K \cup R$  is the element  $a$  with rank  $n - 1$  (but we accepted solutions with more exotic median definitions). Assume for the time being that the median is contained in  $K$ . Since the function  $\text{rank}_{K \cup R}(a)$  is monotone in the argument  $a$  (i.e., for any  $a, a' \in K \cup R$ , if  $a < a'$  then  $\text{rank}_{K \cup R}(a) < \text{rank}_{K \cup R}(a')$ ), we can find an element with rank  $n - 1$  using binary search on elements of  $K$ . We first convert the set  $K$  into a sorted array  $SoK[1 \dots n]$ . Then we perform binary search on the array  $SoK$  with respect to the ranks. Conceptually, we binary search for the value  $n - 1$  in the sorted array  $\text{rank}_{K \cup R}(SoK[1]), \dots, \text{rank}_{K \cup R}(SoK[n])$ . Each time we need to compare  $\text{rank}_{K \cup R}(SoK[i])$  to  $n - 1$ , we exchange  $O(1)$  messages to compute the rank. Since binary search terminates after  $O(\log n)$  steps, it follows that the median is located using  $O(\log n)$  messages. The correctness of this procedure follows directly from the correctness of the binary search procedure.

Therefore, the problem is solved *if* the median element is held by Karp. If this is not the case, the aforementioned protocol fails to locate an element in  $K$  with rank  $n - 1$ . In that case the protocol is repeated, with the roles of Karp and Rabin reversed.

This concludes the description of the simplest variant of the protocol. It exchanges  $O(\log n)$  messages, performs  $O(n \log n)$ -time computation, and is in-place, assuming that an in-place sorting algorithm is used. Various optimizations are possible. They do not decrease the asymptotic number of messages exchanged, but can reduce the running time to  $O(n)$ .

There are many variants of the aforementioned procedure. For example, the elements  $a$  for which we compute  $\text{rank}_{K \cup R}(a)$  could be chosen from  $K$  in odd rounds and from  $R$  in even rounds. We could also choose those elements at random from  $K$  or  $R$ , in a manner analogous to the Randomized Select procedure. This however leads only to  $O(\log n)$  messages *in the expectation*. In addition, the analysis of Randomized Select needs to be modified slightly, given that at each step we choose a random element from either  $K$  or  $R$ , not from the whole data set.

### Problem 3. The lightest path

Consider a weighted connected graph  $G$ , with  $m$  edges and  $n$  vertices. Each edge weight is an integer in the range  $[1, 10n]$ . Consider any two nodes  $u, v \in G$ , and a path  $P$  between  $u$  and  $v$ . The *weight* of the path  $P$  is defined as the *maximum* weight of any edge in  $P$ .

Give an efficient algorithm that given the graph  $G$  and vertices  $u$  and  $v$ , finds the minimum weight path between  $u$  and  $v$  in  $G$ .

**Solution:**

A common approach taken by students was to modify Dijkstra's algorithm. Instead of storing the shortest known distance to a node, store the smallest known path weight to the node. In particular, change the RELAX subroutine (which potentially updates the value at node  $v$  using the best-known path to  $u$ ) to the following:

```
RELAX(u,v,w)
1      if  $d[v] > \max(d[u], w(u, v))$ 
2          then  $d[v] \leftarrow \max(d[u], w(u, v))$ 
3           $\pi[v] \leftarrow u$ 
```

We essentially replace the '+' operator with a 'max' operator. The arguments used in the proof of correctness for Dijkstra's algorithm still holds after doing so. In particular, path weights are nondecreasing whenever they are extended (this is true even if we allowed negative edge weights) because the max operator will always keep the larger value. Thus we know that for the set of vertices that has not yet been relaxed, the vertex that has the minimum known cost can not get an improved cost by going through any other intermediary vertex. It is always therefore safe to relax that vertex and add it to the set of finished vertices to which we have found optimal paths. Interestingly enough, this problem loses optimal substructure in that it is no longer a strict requirement that all subpaths must be optimal for the entire path to be optimal. That is, if two parts of the path contain two different edges that both have the maximum edge weight required for the path, if we improve the first half of the path, the entire path is not made more optimal. Points were taken off for making incorrect arguments and for not making any.

Simply using a Fibonacci heap results in a running time of  $O(m + n \lg n)$ . However, because the edge weights can only be integers in the range  $[1, 10n]$ , a monotone priority queue of size  $10n$  can be used as described in Problem Set 5 since the only path weights possible are the edge weights. This results in an overall running time of  $O(n + m)$ . (Note: Because the graph is connected, we know that  $n = O(m)$ , so we can also simplify the running time to  $O(m)$ .)

The other main approach students took was to "grow" a copy of the graph by adding in the edges of the original graph in order of their edge weights. Once the two nodes  $u$  and  $v$  are connected, we know that the last edge added to the graph is the minimum path weight possible. The simplest way of implementing this algorithm is to first sort the edges by edge weight (by using counting sort, we can get  $O(m + n)$  because the  $m$  edge weights can only be of  $10n$  different values), add the edges of the smallest edge weight not yet in the graph, test the connectivity of  $u$  and  $v$ , and then iterate. Testing connectivity can be done by using Dijkstra's (rather suboptimal), using BFS or DFS

(which takes  $O(m + n)$  each time for  $O(mn)$  total running time), or maintaining the connected components using the Union-Find data structure described in CLRS (for  $O(m\alpha(n))$  time). While the last of these techniques depends on growing the graph edge by edge, the former two techniques can be sped up by performing a binary search on the minimum edge weight required to connect  $u$  and  $v$ . In other words, try the graph with only edge weights at most  $5n$ , then either  $2.5n$  or  $7.5n$ , etc. This would yield a running time of  $O(m \log n)$  when using BFS or DFS to test connectivity.

The Union-Find approach is implemented by first making each vertex into a set by itself. Then, whenever an edge is added, we union the two sets that contain the endpoints of the edge. We test connectivity by finding the set representatives of  $u$  and  $v$  and check if they're the same. Using the union by rank heuristic gives an amortized bound of  $O(\alpha(n))$  time per Union-Find operation, where  $\alpha(n)$  is the inverse Ackerman function and grows incredibly slowly (its value is on the order of 4 or 5 on reasonable input sizes).

Another clever solution actually achieves  $O(m)$  running time also by growing the graph but saves time by only keeping track of which nodes are reachable from  $u$  using a variable stored at each node. At the beginning,  $u$  is the only one marked as reachable from  $u$ . When adding the edges in sorted order, if one endpoint of the edge is marked as reachable and the other isn't, mark the second vertex as reachable and perform a BFS to mark everything connected to that node as reachable. If  $v$  is ever encountered, we're done. If neither endpoint of the added edge is reachable or both of them are, simply insert the edge and do nothing. This algorithm only performs a constant amount of work when adding an edge to the graph for a total of  $O(m)$  for all the edges, and the series of BFSs only traverse each vertex and edge once for a total of  $O(m + n)$ , so the entire algorithm runs in  $O(m)$ .

Another related approach is to build a minimum spanning tree. In an MST, the path between  $u$  and  $v$  necessarily has the minimum path weight possible in the graph. Thus we can simply run an MST algorithm and use DFS to find the path between  $u$  and  $v$ . Since the MST is a tree, there is only one unique path for the MST. To see how the path in the MST is correct, it is useful to think of how Kruskal's algorithm works. (This is fact pretty much like the Union-Find approach except we don't add edges that are unnecessary.) We sort the edges by weight, and we keep adding edges if they do not form cycles. If the path in the MST is wrong, then that means that there is another path in the graph that connects through smaller weight edges in favor of some larger edge used in the MST. Obviously, not all of those edges are in the MST, or otherwise the MST would not be a tree or we would be taking this path. In the process of Kruskal's algorithm, we would have considered adding all those lighter edges before having added in the larger edge. If we chose not to add some of those edges, it would be because adding them would not increase the connectivity; the MST would already have contained a path that connected the proposed edge's endpoints with lesser or equal cost. This implies that if this better path of lighter edges existed, it or another path of equal or lesser cost would be added into the MST before the larger edge was added. Because this path is in existence, the larger edge can no longer be part of the path between  $u$  and  $v$ . Similar to the Dijkstra's algorithm approach, the traditional running time for finding the MST is not quite  $O(m)$  but can be improved because of the integer edge weights.

All of these approaches can be performed using  $O(m + n)$  space, and only  $O(n)$  space for im-

plementing Dijkstra's. Both were an acceptable use of space. Some students' solutions stored the entire path at each node, resulting in using  $O(n^2)$  space and therefore  $\Omega(n^2)$  time, depending on what else their algorithm did. Note that this modification to Dijkstra's is entirely unnecessary since the method shown in CLRS of simply maintaining the predecessor of a vertex is sufficient and only uses  $O(n)$  space. The path can be reconstructed after running Dijkstra's simply by tracing back and following the predecessors.

#### Problem 4. Maximum triangular subset

Assume you are given a set  $S$  of  $n$  distinct numbers  $\{s_1, s_2, \dots, s_n\}$  in the range  $[1, 10n]$ . A subset  $T \subseteq S$  is called *triangular* iff (1)  $|T| \geq 3$  and (2) for every triple of distinct numbers  $x, y, z \in T$ ,

$$x + y > z, x + z > y, \text{ and } z + y > x.$$

Give an efficient algorithm to find a triangular subset of  $S$  with maximum size (i.e largest number of elements). For example, if  $S = \{4, 1, 8, 2, 5, 3, 9, 10, 7\}$  then a triangular subset with maximum size is  $T = \{8, 5, 9, 10, 7\}$ .

#### Solution:

**Executive summary** We want to find a subset  $T$  of  $S$  with the largest size such that (1)  $|T| \geq 3$  and (2) for every triple of distinct numbers  $x, y, z \in T$ ,

$$x + y > z, z + x > y, \text{ and } y + z > x$$

In our solution, we will prove following claims: (1) in any triangular subsets, the sum of the two smallest numbers is strictly greater than the largest number and (2) if  $S$  is sorted, there is a triangular subset with the largest size  $T$  such that  $T$  is a subsequence of  $S$ . Using these claims, we give an algorithm that finds  $T$ , by first sorting the elements of  $S$  in increasing order, and then scanning through the numbers from the smallest to the largest. The algorithm runs in  $\Theta(n)$  time in the worst case. The space requirement is also  $\Theta(n)$ .

**Proof of claims** Let  $A = a_1, a_2, \dots, a_n$  be the sorted sequence of elements in  $S$ .

First, we prove that a subset  $U$  is triangular if and only if sum of the two smallest numbers in  $U$  is greater than the largest number (in  $U$ ): By definition, it is obvious that if  $U$  is triangular, sum of the two smallest numbers is greater than the largest. Now, assume that  $U$  is a subset such that sum of two smallest numbers is greater than the largest. Let  $a, b$  be two smallest numbers and  $c$  be the largest in  $U$ . For any distinct triple  $x < y < z \in U$ :

$$x + y \geq a + b > c \geq z, x + z > y, z + y > x$$

. Therefore,  $U$  is a triangular subset.

Second, we can prove there exists a maximum size triangular subset  $U$  such that  $a_i$  and  $a_{i+1}$  are the two smallest numbers in  $U$  for some  $1 \leq i < n$ : Let  $U_l$  be a maximum size triangular subset. Let  $a_j$  and  $a_k$ ,  $1 \leq j < k \leq n$  be the two smallest numbers in  $U$  and  $a_l$  be the largest number,

$$a_{k-1} + a_k \geq a_j + a_k > a_l$$

Therefore  $(U - \{a_j\}) \cup \{a_{k-1}\}$  is also a maximum size triangular subset.

From the proved arguments, we can derive that there is a maximum size triangular subset  $U$  which is a consecutive subsequence of  $A$ , i.e  $U = \{a_i, a_{i+1}, a_{i+2}, \dots, a_{j-1}, a_j\}$ : By the second argument, let  $U$  be a maximum size triangular subset such that  $a_i$  and  $a_{i+1}$  are the two smallest number in  $U$ , for some  $1 \leq i < n$ . By the first argument,  $U$  must be a consecutive subsequence of  $A$ , otherwise we can increase size of  $U$  by “filling in” missing numbers in the middle and get a bigger triangular subset.

**Algorithm** Therefore, we just need to find the largest  $j_i > i$ , for all  $1 \leq i \leq n$ , such that the sequences  $U_i = \{a_i, a_{i+1}, a_{i+2}, \dots, a_{j_i-1}, a_{j_i}\}$  are a triangular subsets and take the longest sequence. For  $i = 1$ ,  $j_1$  can be found in  $O(n)$  time: let  $j_1 = 2$  and keep increasing  $j_1$  until  $a_1 + a_2 \leq a_{j_1+1}$ . For  $i > 1$ , we can  $j_i$  can be found by let  $j_i = j_{i-1}$  and keep increasing  $j_i$  until  $a_i + a_{i+1} \leq a_{j_i+1}$ . The triangular subset with largest size found in this process is a maximum size triangular subset.

**Running time and space analysis** Since all the numbers in  $S$  are small integers ( $[1, 2, \dots, 10n]$ ), it takes  $\Theta(n)$  time to sort  $S$  using counting sort. It also takes  $O(n)$  time to find  $j_1$ . By amortized analysis for all  $i > 1$ , the total running time to find  $j_i$  is  $\Theta(n)$  since we start increasing  $j_i$  at  $j_{i-1}$ . Therefore, our algorithm only take  $\Theta(n)$  time. The space requirement is also  $\Theta(n)$  (auxiliary space for sorted  $S$ ).

### Problem 5. Search Engine Excerpts

Most popular Internet search engines feature document excerpts for each found document. The goal of search engines is to provide the smallest document excerpt possible that helps the user realize if the found document is relevant to their search. One possible heuristic is to find the smallest excerpt from a found document that includes all of the keywords used in the search.

You are given  $k$  search keywords, and a document that is a sequence of  $N$  words. All  $k$  search keywords are present in the document.

Describe an efficient algorithm that finds a smallest range  $[i, j]$  in the document ( $i, j \in [1, N]$ ), that contains at least one instance of every search keyword.

Note: you can assume that each word or search keyword is represented by a number. **Solution:**

There were three  $O(N)$  solutions to this problem, as well as numerous variations that provided slower runtimes. All solutions begin by hashing each keyword  $k_i$  into a table  $T$  (either using perfect hashing, or some mechanism for resolving false positives in lookups).

The first general solution involves a “sliding window” pass over the document words from 1 to  $N$ . More specifically, two pointers,  $i$  and  $j$ , define a window over the document. The value associated with each  $k_i$  in  $T$  tracks the number of instances of  $k_i$  within this window, and a counter  $m$  tracks the number of entries with non-zero values. Both  $i$  and  $j$  start at index 1 of the document.  $j$  then slides forwards until  $m = 0$ . Whenever a keyword is identified during the slide (in constant time via hash table lookup), the appropriate counter in  $C$  is incremented. Whenever an entry’s value goes from zero to one,  $m$  is incremented.

Once  $m = k$ , we have a complete excerpt. To make the excerpt minimal, we now slide  $i$  forward, decrementing hash table entries and  $m$  when keywords are encountered, until incrementing  $i$  would make our excerpt incomplete. At this point we record our range as the current best complete range, and then slide  $i$  to the next keyword instance. We then repeat our process of sliding  $j$ , then  $i$ , and updating our best observed range, until  $j$  hits the end of our document.

This solution takes  $O(N)$  time and  $O(N)$  space.

The second general solution was to traverse the document, tracking the latest instances of all  $k_i$ , and recording best ranges at each new keyword. More specifically, a linked list  $L$  of latest keyword instances is created during initialization, and populated with one element per  $k_i$  that stores the value -1. The values of the hash table  $T$  are each assigned one of these list elements. We now start a pointer  $i$  at the first document word and scan to the right. At each identified keyword at index  $j$  (again in constant time via  $T$ ), the linked list element for  $k_i$  has its value updated to  $j$ , and is spliced to the end of the linked list  $L$  (because we only ever increase values, causing elements to move to the end of the list, we are in effect implementing a monotone priority queue). Whenever a list element value is changed from -1, a counter  $m$  is incremented. Once  $m = k$ , the interval  $[1, i]$ , contains at least one instance of each keyword. Whenever we find a new keyword, we take the value of the head of our linked list and our current index to form a best complete range, and continue updating this value as  $i$  slides forward. We terminate when  $j$  reaches the end of the list.

This solution takes  $O(N)$  time and  $O(N)$  space.

The third general solution involves splitting the keyword instances into  $k$  separate linked lists and processing them in quasi-parallel. More specifically, the document is first run through once, and linked lists of matched instances (again identified in  $O(1)$ ) are formed off of each  $k_i$  entry in  $T$ . These lists are of necessity sorted, and have aggregate length  $O(N)$ . The range defined by the heads of each of these linked lists will define our current complete excerpt range. We record the maximum head pointer in  $max$  after a  $O(k)$  search. We now begin scanning the document a second time. Whenever a keyword  $k_i$  is matched, three actions occur. First, the range defined by the head element of the list hashed to be  $k_i$  and  $max$  is checked for being the best range (which we track during the scan). Second, the head element of the list  $k_i$  hashes to is removed. Third, the new head element is compared to  $max$ , and if found greater, takes  $max$ ’s place. The scan stops whenever any of the linked lists is emptied.

This solution takes  $O(N)$  time and space.

In general, full credit was given for a bugless  $O(N)$  solution. Penalties were applied for bugs, significant omissions, extremely confusing writeups, or incomplete correctness or analysis. Partial

credit was given to  $O(N \lg k)$  solutions that used different data structures to either track latest keyword instances for solution two, or to find minimums in solution three. Less credit was given for similar solutions that used less efficient data structures to get  $O(Nk)$  algorithms. Less credit was given to  $O(N^2)$  algorithms that used a different, more exhaustive approach.

### Problem 6. Languages of the new empire

In the glory days of the Roman empire, Julius Caesar seeks to establish efficient communications between Rome and the myriad provinces it commands in close and remote corners of the empire. One-way broadcast messages are sent frequently from Rome to all of the  $N$  provinces along previously-specified paths, to announce new taxes, new wars, and other urgent matters. The problem is that many different languages are spoken across the empire, and thus a lot of consecutive translations may need to happen along the way to a remote province when a message is sent, causing unwanted delays. Each province speaks only one of the  $K$  total languages in the empire, and Caesar insisted that they receive messages in their own language (his magnanimity knows no limits).

Communication happens by messengers carrying stone tablets, along a graph  $G = (V, E)$  of roads and sailing routes. Since all roads lead to Rome and only three-way intersections are present, the communication graph is a binary tree, with Rome at the root and provinces at the leaves. At each intersection (each internal node) lies an outpost where a single stone tablet arrives, and is then copied and passed on (exactly once for each of the two outgoing paths), in the same language when possible, or translated to a new language when necessary. For every broadcast, a single message tablet is created in Rome, in Latin, and arrives at Rome's outpost, where it starts its journey throughout the empire. After each intersection, exactly one tablet, in a specified language, leaves down each subsequent path. Since translation of messages takes a long time, you need to select the language of tablets to be used along each segment in a way that minimizes the total cost of needed translations.

In addition to the graph  $G$ , and the language spoken at each province, you are given the cost of translating from each language to each other language, represented as a  $K \times K$  matrix  $M$  of positive numbers for the cost  $M[i, j]$  of translating from language  $i$  to language  $j$ . Note that translation costs need not be symmetrical, that the cost of translating a language to any other language is always strictly positive, and that all diagonal entries of matrix  $M$  are zero as no translation is necessary from a language to itself. You can also assume that the costs reported in  $M[i, j]$  are optimal, namely you cannot get a lower translation cost from  $i$  to  $j$  by translating through a series of other languages.

Design an efficient algorithm to select the language of tables to be used on each segment (and the translations done at each outpost) in order to minimize the total cost of translations. For partial credit, you can simplify the problem by assuming that any translation has unit cost, i.e.  $M[i, j] = 1$  for  $i \neq j$ .

### Solution:

We set up a Dynamic Programming (DP) Cost table  $\text{Cost}_v[1..K]$  for each vertex  $v$ , to be filled in starting from the leaves and towards the root. At each entry  $\text{Cost}_v[i]$ , we store the minimum cost of translations for sending a message from vertex  $v$  to all of the provinces in that subtree, if language  $i$  is used in the road leading to that outpost from Rome. We update the  $\text{Cost}_v$  table for each outpost  $v$  based on the two tables  $\text{Cost}_{v_{right}}$  and  $\text{Cost}_{v_{left}}$  at its two children  $v_{right}$  and  $v_{left}$ , by evaluating all combinations of costs at that intersection and taking the min. Once the min is found, we also store the choice of assignment leading to the optimal score at each vertex in a table  $\text{Assignment}_v[i]$ , with entries  $(j, k)$  if language choices  $j$  and  $k$  at the children led to minimum value, thus enabling a traceback back towards the leaves after the root is reached, in order to find the final language assignment for each outpost  $v$ .

We start by the proof that DP is applicable here. First, there's only a finite number of subproblems, as a leaf with  $N$  leaves has  $N - 1$  internal vertices, each of which has only  $K$  possible languages. Second, the problem exhibits optimal substructure: a solution using the minimum cost of translations to get from Rome to any of the provinces must be made of optimal solutions to get from any of the internal outposts to the provinces it leads to. The proof is by contradiction using a cut-and-paste argument: if a solution from a given province existed with a smaller cost of translations, that portion could be replaced in the overall solution, leading to an overall solution better than the optimal, hence a contradiction.

Next, we set up the dynamic programming update rule, to compute the table of optimal translation costs for a given outpost  $v$  whose children  $v_{right}$  and  $v_{left}$  have their optimal costs already computed. In the general case, the cost of using language  $l$  at outpost  $v$  would be computed as:

$$\text{Cost}_v[l] = \min_{i,j} \{\text{Cost}_{v_{right}[i]} + \text{Cost}_{v_{left}[j]} + M[l, i] + M[l, j]\}$$

This general update rule would have a cost of  $k^3$ , as each of  $k$  entries are filled in, each time testing  $k^2$  combinations of assignments for the two children. However, since translation costs are independent between the two sides, a speedup is possible:

$$\text{Cost}_v[l] = \min_i \{\text{Cost}_{v_{right}[i]} + M[l, i]\} + \min_j \{\text{Cost}_{v_{left}[j]} + M[l, j]\}$$

This independence relies on the fact that no optimal solution requires that two consecutive translations are necessary at a single node  $v$  (from  $i \rightarrow j$  and then  $j \rightarrow k$ ) and that only  $i \rightarrow j$  and  $i \rightarrow k$  solutions are allowed at any one node. The reason for this is that instead of sending  $i$  to node  $v$ , and then doing two consecutive translations at that node, we can simply translate from  $i$  to  $j$  at the parent of  $v$ , and then send  $j$  down that node which we then translate to  $k$  at node  $v$ .

The optimized update rule allows us to pay only cost  $k^2$  for each outpost, since each of  $k$  entries is updated in  $O(k)$  time. Note that  $\text{Assignment}_v[i]$  records the assignment of the left and right children of  $v$  that lead to the optimal translation cost when the language at  $v$  is set to be  $i$ . Also note that the two subloops can be run in any order, as the 'relaxation' step is never surpassing the optimal solution (additional relaxation steps don't hurt).

We initialize the DP computation by setting the cost table for each leaf  $p$  (corresponding to province  $p$ ) as follows:

$$\text{Cost}_p[j] = M[j, i], \text{ where } i \text{ is the language spoken at province } p.$$

(an alternative strategy may be to set  $Cost_p[i] = 0$  and  $Cost_p[j] = \inf$  for all other languages, as we know that no optimal solution will use a different language at the last outpost than the language spoken at that province).

Upon termination, the root contains a vector of the value of the globally optimal solution for each choice of language starting at the Rome outpost. Since the incoming language there is always Latin, we have to add to that value  $Cost_{Root}[i]$  the cost of translating Latin to that language  $M[Latin, i]$ . We choose the language  $i$  minimizing that sum, set that language for the Rome outpost. The  $Assignment_{Rome}[best]$  then starts the series of pointers to children vertex assignments that we trace back to construct the optimal solution, assigning languages to all outposts.

The run time analysis for the entire algorithm is simple.  $O(N * K^2)$  for computing the optimal solution, since a binary tree with  $n$  leaves has  $n - 1$  internal vertices, and since the table of each vertex is computed exactly once, and each computation takes  $O(k^2)$  time. The traceback takes linear time as each optimal solution is immediately looked up by following the  $Assignment_v[best]$  pointers. The space requirements are  $O(N * K)$  for storing  $K$  values and  $K$  tuples at each vertex.

#### Notes:

Note that a greedy algorithm is not possible. The reason is that the local information within a subtree is insufficient to make a globally-optimal choice. For example, if A and B at the root of a subtree have equal cost within the subtree, the choice will not be resolved until we know whether A and B are dominant outside the subtree. More generally, even if some language L1 is giving lowest cost within a subtree, another language L2 may in fact come to be cheapest for that subtree considering the outside information.

In particular, even for the unit-cost version of the problem, simple voting schemes will not work (of counting the provinces beneath a node that speak each language, and choosing the majority). The reason is that it's the layout of the languages that matters, not the total count. For example, the subtree  $(B, (B, (B, (B, (B, (A, A), (A, ((A, A), (A, A))))))))$  contains 7 A's and 6 B's, hence at every node, the majority-voting scheme will choose A, leading to a total of 6 translations, from the top-most A into each individual B. The optimal solution however starts with a top-most B and only requires 1 translation from B to A at the root of the all-A subtree.

It should also be noted that the overlapping subproblems nature of the problem can only be exploited in bottom-up algorithms that move from the leaves to the root. The reason is that optimal solutions to the subproblems (given a starting language at the root) can only be computed for fully-explored subtrees at a time. This is not possible from the root down, and would lead to exponential-time algorithms that have to travel all the way down to the leaves at each iteration to collect information.

To receive full credit, solutions had to demonstrate a working Dynamic Programming (DP) solution, including set-up of cost variables, bottom-up propagation of costs, a precise and correct update rule with initialization and termination conditions, a traceback step after appropriate update of max pointers, and a proof of correctness and analysis of running time. Partial credit was received for Greedy solutions that propagated sums to the root and traced back to maximum-frequency choices, as they demonstrated the principles of DP, despite the misunderstanding described above.

## Quiz 2 Solutions

This take-home quiz contains 5 problems worth 25 points each, for a total of 125 points. Your quiz solutions are due **between 7pm and 8pm on Wednesday, November 18, 2009 across from 32-141**. Late quizzes will not be accepted unless you obtain a Dean's Excuse or make prior arrangements with the course staff. You must hand in your own quiz solutions in person.

**Guide to this quiz:** Every problem (except one part about NP-completeness) asks you to design an efficient algorithm for a given problem. Your goal is to find the ***most efficient algorithm possible***. Generally, the faster your algorithm, the more points you receive. For two asymptotically equal bounds, worst-case bounds are better than expected or amortized bounds. The best possible solution will receive full points if well written, but ample partial credit will be given for correct solutions, especially if they are well written. Bonus points may be awarded for exceptionally efficient or elegant solutions.

Plan your time wisely. Do not overwork, and get enough sleep. Your very first step should be to write up the most obvious algorithm for every problem, even if it is exponential time, and then work on improving your solutions, writing up each improved algorithm as you obtain it. In this way, at all times, you have a complete quiz that you could hand in.

**Policy on academic honesty:** The rules for this take-home quiz are like those for an in-class quiz, except that you may take the quiz home with you. As during an in-class quiz, you may not communicate with any person except members of the 6.046 staff about any aspect of the quiz during the exam period, even if you have already handed in your quiz solutions.

This take-home quiz is “limited open book.” You may use your course notes, the CLRS textbook, and any of the materials posted on the course web page, but *no other sources whatsoever may be consulted*. For example, you may not use notes or solutions to problem sets, exams, etc. from other times that this course or other related courses have been taught. **You may not use any materials on the World-Wide Web, including OCW.** You probably won’t find information in these other sources that will help directly with these problems, but you may not use them regardless.

If at any time you feel that you may have violated this policy, it is imperative that you contact the course staff immediately. If you have any questions about what resources may or may not be used during the quiz, please send email to `6046-staff@csail.mit.edu`.

**Write-ups:** Answer **each problem on a separate sheet (or set of stapled sheets)** of paper. Mark the top of each problem with your name, 6.046J/18.410J, the problem number, your recitation time and your TA’s name. Your write-up for a problem should start with a topic paragraph that provides an executive summary of your solution. This executive summary should describe the problem you are solving, the techniques you use to solve it, any important assumptions you make, and the asymptotic bounds on the running time your algorithm achieves, including whether they are worst-case, expected, or amortized.

Write your solutions cleanly and concisely to maximize the chance that we understand them. When describing an algorithm, give an English description of the main idea of the algorithm. Adopt suitable notation. Use pseudocode if necessary to clarify your solution. Give examples, draw figures, and state invariants. A long-winded description of an algorithm's execution should not replace a succinct description of the algorithm itself.

Provide short and convincing arguments for the correctness of your solutions. Do not regurgitate material presented in class. Cite algorithms and theorems from CLRS, lecture, and recitation to simplify your solutions. Do not waste effort proving facts that can simply be cited.

Be explicit about running time and algorithms. For example, don't just say that you sort  $n$  numbers, state that you are using MERGE-SORT, which sorts the  $n$  numbers in  $O(n \lg n)$  time in the worst case. If the problem contains multiple variables, analyze your algorithm in terms of all the variables, to the extent possible.

Part of the goal of this quiz is to test your engineering common sense. If you find that a question is unclear or ambiguous, make reasonable assumptions in order to solve the problem, and state clearly in your write-up what assumptions you have made. Be careful what you assume, however, because you will receive little credit if you make a strong assumption that renders a problem trivial.

**Bugs:** If you think that you've found a bug, send email to `6046-staff@csail.mit.edu`. (If you did not find a bug, and ask us a question, our answer will likely be "state your assumptions".) Corrections and clarifications will be sent to the class via email and posted on the class website. Check your email and the class website daily to avoid missing potentially important announcements.

**PLEASE REREAD THESE INSTRUCTIONS ONCE A DAY DURING THE EXAM.**

**GOOD LUCK, AND HAVE FUN!**

### Problem 1. Stripper Display

Your company ShowOff has invented a new bioalgorithmic manufacturing process called Stripper that produces, at very low cost, a batch of  $n$  strips of organic LEDs, each consisting of an  $m \times 1$  array of pixels. Unfortunately, the process is error-prone, so many of the LEDs are damaged. Thus the  $i$ th strip can be represented by an  $m \times 1$  vector  $s_i$  of 0s (broken) and 1s (working). For example, one batch with  $n = 6$  and  $m = 8$  might look like this:

$$s_1 = \begin{pmatrix} 0 \\ 1 \\ 0 \\ 1 \\ 0 \\ 1 \\ 1 \\ 1 \\ 1 \\ 0 \\ 0 \\ 0 \end{pmatrix}; \quad s_2 = \begin{pmatrix} 0 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 0 \\ 0 \\ 1 \\ 1 \end{pmatrix}; \quad s_3 = \begin{pmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix}; \quad s_4 = \begin{pmatrix} 0 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \end{pmatrix}; \quad s_5 = \begin{pmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix}; \quad s_6 = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \end{pmatrix}.$$

Unsatisfied by one-dimensional displays, you want to assemble these strips into an  $m \times n$  matrix, using any order of columns you want, to maximize the area of the biggest rectangle filled with 1s. (Then you will cut out that large working rectangle and sell it.) In the example above, the optimal solution is any matrix that places strips  $s_2, s_4, s_5$  as consecutive columns, yielding an all-1s rectangle of area 21 (these three columns and rows 2–8).

Design the most efficient algorithm you can to find a permutation of columns  $s_1, s_2, \dots, s_n$  to form an  $m \times n$  binary matrix with the largest possible rectangle of all 1s.

#### Solution:

**Executive summary:** The algorithm finds the max rectangle by sweeping through all rows and calculating the sizes of all possible rectangles one can have by aligning the bottom of the rectangles with the  $i$ th row. The basic idea is to traverse through each row  $i$ , and maintain the “height” of each column, i.e., the number of ones in that column extending upwards from row  $i$ . Given the heights of each column, one has to find a subset  $S$  of columns maximizing  $|S| \cdot \min(S)$ . This subset can be found by trying values for  $\min(S)$  and counting how many columns have heights that are greater or equal.

**Algorithm:** The straightforward approach would be to first calculate the heights extending upward from each row for each strip, and store this information in a matrix  $H$  (i.e.,  $H[i, j]$  = height of the sequence of working LEDs extending upwards from the  $i$ th row of the  $j$ th strip). Then, traverse through each row  $i = 1, 2, \dots, m$ , sort the columns in a given row  $i$  by their heights in descending order, and find the subset  $S$  of columns maximizing  $|S| \cdot \min(S)$ . The running time of such algorithm would depend on the sorting algorithm used, because the sorting dominates the work done in each row.

The more efficient approach is to sort each row as we traverse the rows to count the heights. The key observation is that, while the sweep line advances to the next row, the height for a column is either incremented or reset to zero from one row to the next. This means that one can maintain the sorted order in linear time: place the values that become zero to the end, and the order for the rest is maintained naturally (because the relative order among those columns with nonzero heights are not changed).

We use an array  $count$  to maintain the height of each column. If in the current row, the value in a column is 0 then its  $count$  will be set to 0. Otherwise, its  $count$  is incremented. We can use a linked list  $\pi$  to maintain the sorted order of the columns. When a column's height becomes zero, the entry corresponding to that column is removed from the list and appended to the end of the list. When a column's height is incremented, do nothing (because all unremoved entries in the list are also incremented).

The pseudocode is shown below.

```

1  For all  $i = 1, 2, \dots, n$ , set  $count_i = 0$ 
2  Let  $\pi$  be a linked list of length  $n$  with entries initially set to  $1, 2, \dots, n$ 
3  For all  $i = 1, 2, \dots, n$ , let  $pointer_i$  be the pointer that keep track of the entry of value  $i$  in  $\pi$ .
4  Set  $A_{max} = 0$ 
5  for each row  $i$  of the matrix in order from 1 to  $m$ 
6    do
7      for each column  $j$ 
8        do
9          if the entry  $(i, j)$  of the matrix is 1
10            then
11               $count_i = count_i + 1$ 
12            else
13               $count_i = 0$ 
14              Use  $pointer_i$  to remove the entry of value  $i$  from  $\pi$  and append it to the end of  $\pi$ 
15
16   $\triangleright$  At this point,  $\pi$  is an order of the column such that  $count_{\pi_1} \geq count_{\pi_2} \geq \dots \geq count_{\pi_n}$ .
17  for  $i = 1, 2, \dots, n$ 
18    do
19      Let  $k = \pi_i$ 
20      Let  $A = count_k \cdot i$ 
21      if  $A > A_{max}$ 
22        then
23           $A_{max} = A$ 
24           $\pi_{max} = \pi$ 
25
26  Output  $\pi_{max}$ 
```

**Running time:** The running time to maintain the linked list and update  $count$  is linear ( $O(N)$ ). Therefore, the total running time is  $O(MN)$ .

**Alternate solutions:** We summarize a few approaches we have seen in the submissions. Some people chose to do the straightforward approach. In this case, depending on the sorting algorithm used, the running time is either  $O(M(N + M))$  (e.g., counting sort or bucket sort) or  $O(M(N \log N))$  (e.g., merge sort).

Another (slightly less efficient) approach taken by many people is to count the number of strips that have consecutive working LEDs from row  $i$  to row  $j$  (inclusive, where  $1 \leq i \leq j \leq m$ ), and find the max rectangle by maximizing  $|S| \cdot (j - i + 1)$ , where  $S$  is the subset of columns with consecutive working LEDs between those rows. Using dynamic programming, one can count the number of columns with consecutive working LEDs between rows  $i$  and  $j$  in  $O(NM^2)$  — to determine whether a strip has working LEDs between rows  $i$  and  $j$ , one can look at the solution to the subproblem for working LEDs between rows  $i$  and  $j - 1$ , and whether the  $j$ th LED is working. After obtaining this information, maximizing  $|S| \cdot (j - i + 1)$  takes either  $O(M^2)$  or  $O(NM^2)$ , depending on how this information is stored. In either case, the total running time is just  $O(NM^2)$ .

Yet another approach with a similar idea taken by some people is as follows. Each row (fixed column order) is represented as a bit string:  $R_i = b_1 b_2 b_3 \dots b_n$ , where  $b_j$  in  $R_i$  is 1 if the  $i$ th row in  $j$ th strip is working, and 0 if otherwise. We calculate the results of ANDing all possible subsets of consecutive rows. The final string from ANDing these rows gives us the columns that have consecutive working LEDs between these rows, indicated by the positions and number of 1's in the string. We maximize  $|S| \cdot (\# \text{ of } 1\text{'s in } \text{AND}(S))$ , where  $S$  is the subset of consecutive rows. Again, one can use dynamic programming to find the result of ANDing all possible subsets of consecutive rows, where the base case is all subsets of size one:  $R_1, R_2, \dots, R_m$ , the subproblems are subsets of size  $k = 1, 2, \dots, m$ , with choices of which row is the first row. This can be done in  $O(NM^2)$  (number of subproblems =  $M$ , choices per subproblem =  $M$ , and each subproblem takes  $O(N)$  to compute).

Some people tried to take the greedy approach, where they greedily choose which column to append to the set next. This approach does not work. What's locally optimal here may not lead to a globally optimal choice.

## Problem 2. Around the Country

Based on your 6.046 performance, you have won a prize sponsored by Algorithmic Airlines: a free ticket to travel around the continental United States. You begin your trip at Bangor, Maine, which is the easternmost point served by the airline. Then you must travel only from east to west, until you reach Eureka, California, which is the western-most point served by the airline. Finally, you must come back, only by west-to-east travel, until you return to Bangor. However, during your trip, no city may be visited more than once, except for Bangor which must be visited exactly twice (at the beginning and the end of the trip). For publicity reasons, you are not allowed to use any other airline or any other means of transportation.

You want to get the most out of your free ticket by visiting the most cities possible. Being Algorithmic Airlines, they have provided you with an electronic list of cities served by the airline and a list of direct flights between pairs of cities. Design the most efficient algorithm you can to find an itinerary which visits as many cities as possible while satisfying the above conditions.

### Solution:

**Executive summary:** Dynamic programming. Let  $n$  be the number cities. We number the cities from 1 to  $n$  in order from east to west: 1 is the easternmost city and  $n$  is the western-most city. We have  $n^2$  subproblems, named  $D[i, j]$  for  $1 \leq i, j \leq n$ : find the length of the optimal path that starts from city  $i$ , travels west to visit city  $n$ , and travels east to return to city  $j$ . Clearly,  $D[1, 1]$  is the length of the optimal path we want to find. We spend  $O(n)$  time per subproblem, for a total running time of  $O(n^3)$ .

**Dynamic program:** We write a recurrence for  $D[i, j]$  based on the following case analysis:

- If  $i = j = n$ , then the obvious solution is 1.  $D[n, n] = 1$ .
- If  $1 < i = j < n$ , there is no such path since the any feasible path should not visit city  $i$  more than once.  $D[i, j] = -\infty$ .
- If  $i < j$ , consider an optimal path. Let  $i_1$  be the first city after  $i$  in the path.

Observe that, if we remove  $i$  from this path, the remaining should be an optimal solution for the problem “finding the optimal path starting from  $i_1$ , visiting  $n$  and returning to  $j$ ”. Therefore,  $D[i, j] = D[i_1, j] + 1$ .

Also, observe that there should be no  $k > i$  such that  $(i, k) \in E$  and  $D[k, j] > D[i_1, j]$ . Otherwise, we can find a path that is longer than the optimal path (by starting from  $i$ , visiting  $k$  and then following the optimal path). Contradiction.

Putting the observations together,  $D[i, j]$  can be computed as  $\max\{D[k, j] : i < k \leq n \text{ and } (i, k) \in E\} + 1$ .

- If  $j < i$ , similar to the case  $i < j$ ,  $D[i, j] = \max\{D[i, k] : j < k \leq n \text{ and } (j, k) \in E\} + 1$ .
- If  $i = j = 1$ , similar to the case  $i < j$ ,  $D[1, 1] = \max\{D[k, 1] : 1 < k \leq n \text{ and } (1, k) \in E\} + 1$ .

Thus we obtain the following recurrence:

$$D[i, j] = \begin{cases} 1 & \text{for } i = n \text{ and } j = n, \\ -\infty & \text{for } 1 \leq i = j < n, \\ \max\{D[k, j] : i < k \leq n \text{ and } (i, k) \in E\} + 1 & \text{for } i < j, \\ \max\{D[i, k] : j < k \leq n \text{ and } (j, k) \in E\} + 1 & \text{for } i > j. \end{cases}$$

Implementing this recurrence with recursion and memoization, we have  $O(n^2)$  subproblems and spend  $O(n)$  time per subproblem, so the total running time is  $O(n^3)$ . Alternatively, we can compute the table  $D[i, j]$  bottom-up using the following pseudocode:

```

1    $D[n, n] = 1$ 
2   for  $i = 1, 2, \dots, n - 1$ 
3       do  $D[i, i] = -\infty$ 
4   for  $i = 1, 2, \dots, n - 1$ 
5       do  $D[i, i] = -\infty$ 
6   for  $i = n, n - 1, \dots, 2, 1$ 
7       do
8           for  $j = n, n - 1, \dots, 2, 1$ 
9               do
10              if  $i < j$ 
11                  then Find  $k > i$  such that  $(i, k)$  is an edge and  $D[k, j]$  is maximum.
12                   $D[i, j] = D[k, j] + 1$ 
13
14              if  $i > j$ 
15                  then Find  $k > i$  such that  $(i, k)$  is an edge and  $D[k, j]$  is maximum.
16                   $D[i, j] = D[i, k] + 1$ 

```

**Finding an optimal path:** Once the table  $D$  is computed, we can find the actual optimal path by backtracking on table  $D$  to find out which subproblems were used to compute  $D[1, 1]$ . Detailed pseudocode:

```

1 Let  $forward$  be an empty linked list.           $\triangleright$  to record the forward path from 1 to  $n$ 
2 Let  $backward$  be an empty linked list.          $\triangleright$  to record the backward path from  $n$  to 1
3 Set  $i = 1$  and  $j = 1$ .
4 Append 1 to  $forward$ .
5 Append 1 to  $backward$ .
6 while  $i \neq n$  or  $j \neq n$ 
7     do
8         if  $i < j$ 
9             then Find  $k_i$  such that  $k_i > i$  and  $D[i, 1] = D[k_i, 1] + 1$ .
10            Set  $i = k_i$ .
11            Append  $k_i$  to the end of  $forward$ .
12        if  $j < i$ 
13            then Find  $k_j$  such that  $D[n, k_j] = D[n, k_j] + 1$ .
14            Set  $j = k_j$ .
15            Append  $k_j$  to the head of  $backward$ .
16    Output the concatenation of  $forward$  and  $backward$  (with duplicate (at  $n$ ) removed).

```

**Optimization:** We can improve the running time bound to  $O(VE)$  using the handshaking lemma, because the running time is proportional to  $n$  times the sum of the degrees of the vertices.

**Alternate solutions:** A few people chose to reduce the problem to a max-cost flow problem, and solve it using linear programming. To do this, start by creating a flow network with a node for

each city. To make sure that you don't visit a city multiple times, split every node into two nodes, and connect the two with a directed edge. Set all edge capacities and costs in the flow network to 1. Finally, create a source and a sink, and connect them to Bangor and Eureka (respectively) with an edge of capacity 2. To solve the original problem, we need to get 2 units of flow across the network, and do so with the maximum possible cost. We can do this by solving a linear program that maximizes

$$\sum_{(u,v) \in E} f(u, v)$$

while satisfying the ordinary flow constraints. Since we need our flow to be integral, a correct solution must show how a fractional solution obtained by solving the linear program can be turned into an integral solution. One way to show that is to note that we will only get a non-integral flow when the network branches with an equal number of edges on both sides of the branching. In that case, we can just round the flow on one side of the branching to get an integral flow. To obtain the optimal tour, note that the max-cost flow on the network will be forced to take two separate paths from Bangor to Eureka. Take one of the paths and reverse it, and the two paths will constitute the optimal tour. This algorithm is correct but not as fast as the dynamic program.

There is also the obvious, and exponential, solution obtained by enumerating all the possible tours from Bangor to Eureka and back, checking the validity of the tour, and comparing the lengths to come up with the longest one. Enumerating the paths might take  $O(4^n)$ ,  $O(2^{n \lg n})$ , or  $O(2^n)$ , depending on how you do it, and evaluating the validity and length of the tour might take  $O(n)$  or  $O(n + |E|)$ .

A few people attempted to solve the problem using DFS or BFS in different ways, while making the wrong assumption that the running time of these algorithms is linear in the size of the original graph. The running time of these algorithms is linear in the size of the search tree, which is exponential in the size of the original graph if we are to get a correct solution.

A few people made the mistake of trying to solve the problem by finding the longest path in the DAG constructed by concatenating the two DAGs corresponding to the eastbound and westbound flights, while making sure that no city is visited twice. This approach assumes that our problem has the same optimal substructure as this longest (or shortest) path problem, which is not the case. To see why this is the case, note that finding a longest path does not guarantee an optimal roundtrip, because we might settle for a shorter path on each way in order to maximize the total length or the round trip.

Another related and common mistake is to assume that our problem has the greedy-choice property (for some choice), and attempt to get the optimal tour in a greedy manner. Because our problem does not have this property, greedy algorithms will not be optimal, even if they produce a valid solution.

### Problem 3. TwitDate

You have founded an algorithmic dating company TwitDate that suggests to participating Twitter users who they should ask out on a date. At the foundation of your plan is that personality is

fundamentally one-dimensional, that like personalities attract, and that people prefer to ask others out via word-of-mouth through public tweets (messages on Twitter). For example, Angelina Jolie (@AngelinaJolie) follows Larry King (LarryKingHeart), who follows solidity28, who follows Brad Pitt (thepitts). So you might suggest to Brad Pitt that he ask Angelina Jolie out by tweeting that he fancies her, with the intent that solidity28 and then Larry King retweet the message, which then Angelina Jolie will see.

You have downloaded the database of all Twitter follow relations, and represented it as a directed graph  $G = (V, E)$ , with edges directed from follower to followee. You have also precomputed your patented one-dimensional personality measure  $P(u)$  for each Twitter user  $u$ . Design the most efficient algorithm you can that, for every user  $u$ , finds a user  $\heartsuit[u]$  reachable by a chain of follows  $u \rightarrow \dots \rightarrow \heartsuit[u]$  and, subject to this constraint, minimizes  $|P(u) - P(\heartsuit[u])|$ .

**Solution: Reduction to graph problem:** Let  $G = (V, E)$  be the “follow” graph. The vertex set  $V$  represents the people, and for any two vertices  $u, v \in V$ , there is a directed edge  $(u, v)$  if  $u$  follows  $v$  on Twitter. We say that a vertex  $v$  is **reachable** from a vertex  $u$  if there is a directed path from  $u$  to  $v$  in  $G$ .

**Executive summary:** The basic idea is to compute the *transitive closure* of the graph (i.e., for each vertex  $v \in V$ , compute the set reachable vertices from vertex  $v$ ). Then, for each vertex  $v$ , scan through the set of reachable vertices from  $v$  to find the best match for  $v$ . We can solve the problem in  $O(\min(V^2 + VE, V^{2.376} \log V))$  by taking the best of two algorithms. The first algorithm uses BFS or DFS, runs in  $O(V^2 + VE)$  time and is best if  $E = O(V^{1.376} \log V)$ . The second algorithm uses Coppersmith–Winograd matrix multiplication, runs in  $O(V^{2.376} \log V)$  time and is best if  $E = \omega(V^{1.376} \log V)$ .

**Algorithm 1:** The  $O(V^2 + VE)$  algorithm works as follows. For each vertex  $v \in V$ , use BFS or DFS to find the set of vertices that are reachable from  $v$ . Then, we scan through the set of vertices that are reachable from  $v$  to find the best match. The running time for each execution of BFS (or DFS) is  $O(E)$ . Because we have to make  $V$  executions of BFS (or DFS), one for each vertex, the total running time to compute the transitive closure is  $O(VE)$ . Once the transitive closure is found, the running time to find the best match for each vertex  $v$  is  $O(V)$ . Therefore, the total running time is  $O(V^2 + VE)$ .

**Algorithm 2:** The  $O(V^{2.376} \log V)$  algorithm works as follows. Let  $A$  be the adjacency matrix for  $G$ , i.e.,  $A_{uv} = 1$  if  $(u, v) \in E$ , and  $A_{uv} = 0$  otherwise. Let  $B = (A + I)^V$  where  $I$  is the identity matrix. We define the **positivity indicator** of  $B$  to be the following matrix:

$$C_{uv} = \begin{cases} 1 & \text{if } B_{uv} > 0, \\ 0 & \text{otherwise.} \end{cases}$$

As we will prove later, for any pair of vertices  $u, v \in V$ ,  $C_{uv} = 1$  if and only if  $v$  is reachable from  $u$  in  $G$ . Therefore, in order to find the best match for a vertex  $u$ , we only need to look at the row  $u$  of  $C$  and find  $v$  such that  $C_{uv} = 1$  and  $|P(u) - P(v)|$  minimizes.

The detail of the algorithm is shown in the following pseudocode:

```

1 Let  $A$  be the adjacency matrix for  $G$ .
2 Let  $A[u, u] = 1$  for all  $u \in V$ .
3  $C = \text{COMPUTE}(A, |V|)$             $\triangleright$  Computing  $B = A^{|V|}$ 
4 for each  $u \in V$ 
5   do
6      $\text{Best-Match}(u) = \operatorname{argmin}_v \{|P(v) - P(u)| : B[u, v] > 0\}$ 

```

where the function COMPUTE can be implemented as follows:

```

COMPUTE( $A, n$ )
1 if  $n = 1$ 
2   then
3     return  $A$ 
4 if  $n = 1 \pmod{2}$ 
5   then
6      $D = \text{COMPUTE}(A, n - 1)$ 
7      $C = D \times A$ 
8     Replace all positive entries of  $C$  by 1.
9   return  $C$ 
10 else
11    $D = \text{COMPUTE}(A, n/2)$ 
12    $C = D \times D$ 
13   Replace all positive entries of  $C$  by 1.
14   return  $C$ 

```

**Running time:** In the algorithm above, computing  $C$  takes  $O(\log V)$  matrix multiplications each of which can be done in  $O(V^{2.376})$  time using Coppersmith–Winograd matrix multiplication algorithm. Therefore, computing  $C$  takes  $O(V^{2.376} \log V)$ . Computing the best matches for all vertices from  $C$  takes only  $O(V^2)$ . Therefore, the overall running time of this algorithm is  $O(V^{2.376} \log V)$ . Note that in this algorithm, we don't compute the matrix  $B = (A + I)^V$  directly because the entries in this matrix may have exponential values.

**Proof of correctness:** It is sufficient to prove the claim above that, for any pair of vertices  $u, v \in V$ ,  $C_{uv} = 1$  if  $v$  is reachable from  $u$  in  $G$ , and  $C_{uv} = 0$  otherwise.

Consider the sequence of matrices  $C^{(1)}, C^{(2)}, \dots, C^{(V)}$  such that  $C^{(i)}$  is the positivity indicator of the matrix  $B^{(i)} = (A + I)^i$ . (Clearly, by definition,  $C^{(V)} = C$ .) We will prove the following stronger claim (which implies the claim we want to prove):

**Claim:** For any  $1 \leq i \leq V$ , for any pair of vertices  $u, v \in V$ ,  $C_{uv}^{(i)} = 1$  if there is a path of length at most  $i$  from  $u$  to  $v$  in  $G$ , and  $C_{uv}^{(i)} = 0$  otherwise.

**Proof of claim:** By induction on  $i$ .

- Case  $i = 1$ ,  $B^{(i)} = A + I$ . Clearly,  $B_{uv}^{(i)} > 0$  if and only if  $u = v$  or  $(u, v) \in E$ . Therefore, there must be a path of length at most 1 from  $u$  to  $v$ .
- Case  $i > 1$ . By the induction hypothesis,  $C_{uv}^{(i-1)} = 1$  if and only if  $v$  is reachable from  $u$  in  $G$  by a path of length at most  $k - 1$ .

Consider any pair of vertices  $u, v \in V$  such that  $C_{uv}^{(k)} = 1$  (thus,  $B_{uv}^{(k)} > 0$ ). Since  $B^{(k)} = B^{(k-1)} \cdot (A + I)$ ,

$$B_{uv}^{(k)} = \sum_{w=1}^n B_{uw}^{(k-1)} \cdot (A + I)_{vw}$$

Therefore, there must exist some  $w$  such that  $B_{uw}^{(k-1)} > 0$  and  $(A + I)_{vw} > 0$ . This implies that  $w$  is reachable from  $u$  by a path of length at most  $k - 1$  and  $v$  is reachable from  $w$  by a path of length at most 1. Therefore,  $v$  is reachable from  $u$  by a path of length at most  $k$ .

Conversely, consider any pair of vertices  $u, v \in V$  such that  $v$  is reachable from  $u$  by a path of length at most  $k$ . Let  $P = (u = w_1, w_2, \dots, w_{p-1}, w_p = v)$  with  $p \leq k$  be one such path. Clearly,  $A_{w_{p-1}v} = 1$  by definition. Furthermore, since  $w_{p-1}$  is reachable from  $u$  by a path of length at most  $k - 1$ ,  $B_{uw_{p-1}}^{(k-1)} > 0$ . Therefore,

$$B_{uv}^{(k)} = \sum_{w=1}^n B_{uw}^{(k-1)} \cdot (A + I)_{vw} > B_{uw_{p-1}}^{(k-1)} \cdot (A + I)_{w_{p-1}v} > 0$$

Therefore,  $C_{uv}^{(k)} = 1$ .

**Alternate solutions:** Some students reduced this problem to the all-pair shortest-path problem. This approach is also correct because, for any pair of vertices  $u, v \in V$ ,  $v$  is only reachable from  $u$  if and only if the shortest path from  $u$  to  $v$  is less than infinity. However, the running time is always worse (because some computation was wasted to compute the *shortest* path). In particular, the Floyd-Warshall algorithm can be used to achieve running time  $O(V^3)$ . Johnson's algorithm is slightly better and achieves running time  $O(V^2 \log V + VE)$ .

Some students made the very nice observation that, by identifying and contracting *strongly connected components* (see CLRS), the problem can be reduced to finding transitive closure on an acyclic graph. However, this trick does not help improve the running time.

#### Problem 4. Rein à Trois

The American Kidney Association (a.k.a. AKA) has started a new service for finding matching pairs of kidney donors and kidney recipients. In this service, a pair of people (e.g., spouses, siblings, or other relatives) can register as a donor-recipient pair: the donor of the pair offers a kidney for donation to someone else, in exchange for the recipient of the pair receiving a kidney from someone. The donor would have given her/his kidney directly to the recipient in the pair, except that their blood types differ, making them **incompatible** for kidney transplant. The AKA stores a list of all donor-recipient pairs that have registered, along with the blood types of every person. (Each person can register for only one pair.)

You're heading a new project called ThreeWay that looks for three pairs  $a, b, c$  where  $a$ 's donor is compatible with (has the same blood type as)  $b$ 's recipient,  $b$ 's donor is compatible with  $c$ 's recipient, and  $c$ 's donor is compatible with  $a$ 's recipient. In this case,  $a, b, c$  form a *rein à trois*.

- (a) Design the most efficient algorithm you can to find a rein à trois, or report that none exist.

**Solution: Executive summary:** After appropriate reduction, this problem is equivalent to finding a directed triangle in a directed graph  $G = (V, E)$ , where  $|V| = b$  is the number of blood types and  $|E| = n$  is the number of donor-recipient pairs. (Note that  $b \leq n$ .) We can solve this problem in  $O(\min\{nb, n + b^{2.376}\})$  time by taking the best of two algorithms. The  $O(nb)$  algorithm is an optimized brute force, and is best when  $n = O(b^{1.376})$  (relatively sparse graphs). The  $O(n + b^{2.376})$  algorithm is based on Coppersmith–Winograd matrix multiplication, and is best when  $n = \Omega(b^{1.376})$  (relatively dense graphs).

**Reduction:** We construct the directed graph  $G = (V, E)$  with a vertex for each blood type, and an edge  $(u, v) \in E$  if there is a pair whose donor has blood type  $u$  and whose recipient has blood type  $v$ . [We could also define it the other way around.] A rein à trois  $a, b, c$  then corresponds to the three edges in a directed triangle in  $G$ . We can construct an adjacency-matrix representation of this graph in  $O(n + b^2)$  time (which is smaller than the target running time) by scanning through the list of pairs of people (with their blood types) and setting the corresponding entry in the matrix to 1. From this representation, we can compute the adjacency-list representation of the graph in  $O(b^2)$  additional time.

**Algorithm 1:** The  $O(VE) = O(nb)$  algorithm works as follows: for each edge  $(u, v) \in E$ , test whether  $u$  and  $v$  have a common neighbor, i.e., a vertex  $x$  such that  $(v, x) \in E$  and  $(x, u) \in E$ . There are  $|E|$  edges to test, and each test can be performed in  $O(V)$  time by scanning  $v$ 's row and  $u$ 's column in the adjacency matrix in parallel, checking for a common 1 entry.

**Algorithm 2:** The  $O(n + b^{2.376})$  algorithm uses the matrix-multiplication view of transitive closure, introduced in Lecture 11. Specifically, the adjacency matrix  $A$  indicates whether it is possible to go from one vertex to another via a directed path of length 1. The ring-world product  $A \odot A$  indicates whether it is possible to go from one vertex to another via a directed path of length 2. And  $A \odot A \odot A$  indicates whether it is possible to go from one vertex to another via a directed path of length 3. Therefore,  $A \odot A \odot A$  has a 1 on its diagonal if and only if there is a directed triangle. The two  $b \times b$  products can be computed using Coppersmith–Winograd matrix multiplication (as mentioned in lecture), for a cost of twice  $O(b^{2.376})$ . We can then scan the diagonal in  $O(b)$  time. If we find a 1 on the diagonal, that is, a vertex  $v$  on a directed triangle, we can find the corresponding triangle in  $O(n)$  time by scanning all edges and testing whether they form a triangle with  $v$ . Otherwise, there are no directed triangles.

**Alternate solutions:** We received a wide range of correct solutions, from one of the two algorithms above to an  $O(n^4)$  algorithm. Many students found one of the algorithms described above, but no one found both and combined the algorithms to improve the overall performance as a function of  $n$  and  $b$ . (A couple of students did so for two other algorithms, though.) The obvious brute-force algorithm tries all triples of pairs and checks compatibility of each, which takes  $O(n^3)$  time. An improvement is to first translate pairs into a graph of blood types and then test all triples of blood types for existence of a triangle of edges, which takes  $O(n + b^3)$  time. Many students found the  $O(VE)$  algorithm for triangle finding, but used it on the graph  $G' = (V', E')$  whose vertices represent pairs and whose edges represent compatibilities among pairs. Unfortunately, in this graph,  $|V'| = n$  and  $|E'|$  can be  $\Theta(n^2)$ , leading to a worst-case running time of  $\Theta(n^3)$ .

A common mistake was to use depth-first or breadth-first search to find triangles, which is incorrect; as far as the field knows, there is no way to modify these algorithms to find triangles in linear time.

[**State-of-the-art:** The best known running time for triangle finding in a directed graph  $G = (V, E)$  is  $O(\min\{E^{1.408}, V^{2.376}\})$  time. The  $O(V^{2.376})$  algorithm is the second algorithm described above. The  $O(E^{1.408})$  algorithm is described in the following paper: <http://www.cs.bris.ac.uk/~ian/graphs/part.pdf> (Theorem 3.5). Obviously we did not expect anyone to find this algorithm. If you found the quiz problem interesting, though, you might take a look now.]

- (b) Prove that it is NP-complete to decide whether there are  $k$  disjoint instances of rein à trois, that is,  $k$  rein-à-trois triples  $(a_1, b_1, c_1), (a_2, b_2, c_2), \dots, (a_k, b_k, c_k)$  involving  $3k$  distinct pairs of people. In your NP-hardness reduction, you may reduce from any of the NP-complete problems listed in Lectures 15, 16, or 17 or in the textbook.

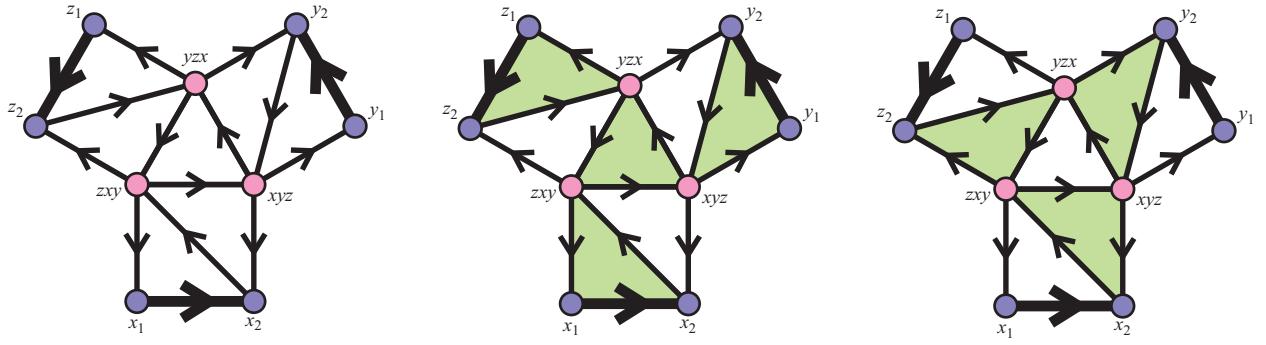
**Solution: Executive summary:** First we show that the problem is in NP using the obvious certificate. Then we prove that the problem is NP-hard by a polynomial-time reduction from Three-Dimensional Matching.

**In NP:** To prove that the problem is in NP, we give a polynomial-length certificate for YES instances verifiable by a polynomial-time algorithm. The certificate is simply the solution: a list of  $k$  rein-à-trois triples. Given such a certificate, we simply need to check two properties. First, each triple must form an instance of rein à trois; blood-type compatibility can be checked in  $O(1)$  time per triple. Second, the triples must be disjoint, which we can check in  $O(n^2)$  time by naïvely trying all pairs or in  $O(n \lg n)$  time by sorting. (We cannot use the  $O(n)$  hashing solution because the algorithm must be deterministic.)

**NP-hardness reduction:** Three-Dimensional Matching (3DM) is one of the problems listed as NP-hard in Lecture 15. We reduce 3DM to the rein-à-trois problem, which proves that the latter is at least as hard and therefore also NP-hard. We are given an instance of 3DM, which consists of three disjoint sets  $X, Y, Z$ , a set  $T \subseteq X \times Y \times Z$

of triples, and a positive integer  $k$ . We construct a rein-à-trois instance, which we represent as a positive integer  $k'$  and a directed graph  $G = (V, E)$ , whose vertices represent blood types and whose edges represent donor–recipient pairs. Our goal is for there to be  $k'$  disjoint instances of rein à trois (which correspond to  $k'$  edge-disjoint triangles in  $G$ ) if and only if there are  $k$  disjoint triples in  $T$ .

The reduction replaces each element in  $x \in X \cup Y \cup Z$  with two distinct vertices  $x_1, x_2$  connected by a directed edge  $(x_1, x_2)$ . Then, for each triple  $(x, y, z) \in T$ , we create the gadget shown in Figure 1 (left), by adding three new vertices  $xyz, yzx, zxy$  and connecting them to  $x_1, x_2, y_1, y_2, z_1, z_2$  as shown. Finally, we set  $k' = 3|T| + k$ . Note that this reduction takes polynomial time.



**Figure 1:** Left: Gadget for each triple  $(x, y, z)$ . The bold edges are shared among multiple gadgets. Middle: Four triangles corresponding to chosen triple. Right: Three triangles corresponding to unchosen triple.

Claim 1: If there are  $k$  disjoint triples  $S \subseteq T$ , then  $G$  has  $k'$  edge-disjoint triangles in  $G$ . Namely, for each triple  $(x, y, z) \in S$ , we include the four triangles  $(x_1, x_2, zxy), (y_1, y_2, xyz), (z_1, z_2, yzx), (zxy, xyz, yzx)$  (Figure 1, middle); and for each triple  $(x, y, z) \in T \setminus S$ , we include the three triangles  $(x_2, zxy, xyz), (y_2, xyz, yzx), (z_2, yzx, zxy)$  (Figure 1, right). The number of triangles is clearly  $3|T| + k$ . Edge-disjointness of the triangles holds within each gadget, and across gadgets because each element  $x \in X \cup Y \cup Z$  appears in only one triple in  $S$  and hence the corresponding edge  $(x_1, x_2)$  appears in only one triangle.

Claim 2: If there are  $k'$  edge-disjoint triangles in  $G$ , then there are  $k$  disjoint triples  $S \subseteq T$ . Here we use that the only four-triangle solution within a gadget is the one in Figure 1 (middle), which requires the use of the edges  $(x_1, x_2), (y_1, y_2)$ , and  $(z_1, z_2)$ , which means that no other triples with  $x, y$ , or  $z$  can have a four-triangle solution. Thus we choose  $S$  to consist of the triples having four rein-à-trois triangles in their corresponding gadget. The remaining (unchosen) triples might as well use the three-triangle solution in Figure 1 (middle) in their gadget, because three triangles is the best possible, and this particular three-triangle solution does not use up any of the shared edges  $(x_1, x_2), (y_1, y_2)$ , and  $(z_1, z_2)$ . Therefore we obtain  $k' - 3|T| = k$  disjoint triples in  $S$ .

**Alternate solutions:** Many tried to reduce from 3DM, but without any gadgets. The resulting “natural” reduction is incorrect. The few who got correct solutions generally solved a different problem, in which the triangles must be vertex-disjoint instead of edge-disjoint. From the rein-à-trois perspective, this corresponds to allowing a more flexible definition of compatibility. This is a different problem, but we graded as if it were the same. In this variation, correct reductions were obtained from 3SAT, Graph 3-Coloring, and Independent Set.

### Problem 5. The Producer

You are the producer for a soon-to-be cult-classic movie, *6.046 Returns*, and you need both actors and investors. You have constructed a list of all  $n$  available actors (mostly former 6.046 students), with actor  $i \in \{1, 2, \dots, n\}$  charging  $a_i$  dollars. For funding, you have found  $m$  available investors. Investor  $j$  offers  $b_j$  dollars toward your budget, but only on the condition that actors  $L_j \subseteq \{1, 2, \dots, n\}$  are included in the movie. (The offer is “all or nothing”: all actors in  $L_j$  must be chosen in order to receive any funding from investor  $j$ , and then the funding is  $b_j$ .) Your profit is the sum of the payments from investors minus the sum of the payments to actors.

Design the most efficient algorithm you can to maximize your profit. (You have no limits on the number of actors or the amount of funding.)

#### Solution:

**Executive summary:** We can represent this problem as a linear program, and use a linear program solver to solve it in polynomial time. The most straightforward way to solve this problem via linear programming is to formulate this problem as an integer linear program, then relax the integrality constraints on the variables to get a linear program. One can prove that the solution to this relaxed linear program may be transformed into a solution for the original problem. A viable alternative solution, which is faster than linear programming, is to reduce this problem to maximum flow on a particular graph.

**Exponential-time solutions:** The simple exponential-time solution to this problem is to try all possible combinations of investors, compute the total profit from those investors, and choose the combination that maximizes this total profit. A similar exponential-time solution may iterate over all possible combinations of actors.

**Linear-programming solution:** Our problem can be rewritten as the following integer linear programming (ILP) problem:

$$\begin{aligned} \text{maximize} \quad & \sum_{j=1}^m b_j y_j - \sum_{i=1}^n a_i x_i \\ \text{subject to} \quad & x_i \geq y_j \quad \text{for every } 1 \leq i \leq n, 1 \leq j \leq m \text{ such that } i \in L_j \\ & x_i = 0 \text{ or } 1 \quad \text{for each } i = 1, 2, \dots, n \\ & y_j = 0 \text{ or } 1 \quad \text{for each } j = 1, 2, \dots, m \end{aligned}$$

In this ILP, the variable  $x_i$  corresponds to the actor  $i$  and the variable  $y_j$  corresponds to investor  $j$ .  $x_i = 1$  represents the event that actor  $i$  is hired, and  $x_i = 0$  means that actor  $i$  is not hired.  $y_j = 1$  indicates that the offer from investor  $j$  is accepted, and  $y_j = 0$  indicates that investor  $j$ 's offer is not accepted.

If we relax the requirement that the solution must be integral, we will get the following linear programming (LP) problem:

$$\begin{array}{ll} \text{maximize} & \sum_{j=1}^m b_j y_j - \sum_{i=1}^n a_i x_i \\ \text{subject to} & x_i \geq y_j \quad \text{for every } 1 \leq i \leq n, 1 \leq j \leq m \text{ such that } i \in L_j \\ & x_i \leq 1 \quad \text{for each } i = 1, 2, \dots, n \\ & y_j \leq 1 \quad \text{for each } j = 1, 2, \dots, m \\ & x_i \geq 0 \quad \text{for each } i = 1, 2, \dots, n \\ & y_j \geq 0 \quad \text{for each } j = 1, 2, \dots, m \end{array}$$

Clearly the optimal objective value for this LP is at least as large as the optimal objective value for the ILP. Therefore, if we can show that there is an optimal solution for the LP that is integral, then this solution must also be the optimal solution for the ILP.

Let  $(\vec{x}^*, \vec{y}^*)$  be an optimal solution for the LP. There are two possible cases:

1.  $\vec{x}^*$  is **integral** (i.e.,  $x_i^*$  is either 0 or 1 for every  $i = \{1, 2, \dots, n\}$ ): Assume that  $\vec{y}^*$  is not integral. There must be some  $y_j^*$  such that  $0 < y_j^* < 1$ . Clearly, by setting  $y_j^* = 1$  we will obtain a better solution and no constraint is violated. This gives us a contradiction, and therefore  $\vec{y}^*$  must also be integral.
2.  $\vec{x}^*$  is **not integral**: Let  $x_{\min}^*$  be the minimum positive value among all  $x_i^*$ 's (i.e.,  $x_{\min}^* = \min_{x_i^* > 0} x_i^*$ ). Since  $\vec{x}^*$  is not integral,  $x_{\min}^*$  is not integral. Let  $S_{\min} = \{i : x_i^* = x_{\min}^*\}$ . Clearly,  $S_{\min}$  is non-empty by definition. Let  $T_{\min} = \{j : y_j^* > 0 \text{ and } L_j \cap S_{\min} \neq \emptyset\}$ . By the constraints in the LP,  $y_j < x_{\min}^*$  for all  $j \in T_{\min}$ . Let  $a = \sum_{i \in S_{\min}} a_i$  and  $b = \sum_{j \in T_{\min}} b_j$ . There are 3 possible cases:
  - i.  $a > b$ : By setting  $x_i^* = 0$  for all  $i \in S_{\min}$  and  $y_j^* = 0$  for all  $j \in T_{\min}$ , the objective value will be increased by
 
$$\sum_{i \in S_{\min}} x_i a_i - \sum_{j \in T_{\min}} y_j b_j \geq a \cdot x_{\min} - b \cdot x_{\min} > 0.$$

Therefore, we can obtain a better solution and no constraint is violated, which contradicts the optimality of our original solution.

- iii.  $a < b$ : Let  $x_1^*$  be the next smallest positive value in  $\vec{x}$  (i.e.,  $x_1^* = \min_{x_i^* > x_{\min}^*} x_i^*$ ). Then by increasing  $x_i^*$  for all  $i \in S_{\min}$  and  $y_j^*$  for all  $j \in T_{\min}$  by  $x_1 - x_{\min}$ , the objective value will be increased by

$$(b - a)(x_1 - x_{\min}) > 0.$$

Therefore, we can obtain a better solution and no constraint is violated, which again contradicts the optimality of our original solution.

- ii.  $a = b$ : By setting  $x_i^* = 0$  for all  $i \in S_{\min}$  and  $y_j^* = 0$  for all  $j \in T_{\min}$  we will obtain another optimal solution for the LP with smaller number positive variables and no constraint is violated.

Therefore,  $(\vec{x}^*, \vec{y}^*)$  is either integral or there is an algorithm to obtain another optimal solution from  $(\vec{x}^*, \vec{y}^*)$  with smaller number of positive variables.

Our algorithm works as follows:

- 1 Set up a LP problem based on the input.
- 2 Solve the LP problem (using an efficient LP solver) to obtain a solution  $(\vec{x}^*, \vec{y}^*)$ .
- 3 **while**  $(\vec{x}^*, \vec{y}^*)$  is not integral
- 4     **do**
- 5          $x_{\min}^* = \min_{x_i^* > 0} x_i^*$
- 6          $S_{\min} = \{i : x_i^* = x_{\min}^*\}$
- 7          $T_{\min} = \{j : y_j^* > 0 \text{ and } L_j \cap S_{\min} \neq \emptyset\}$
- 8         Set  $x_i^* = 0$  for all  $i \in S_{\min}$
- 9         Set  $y_j^* = 0$  for all  $j \in T_{\min}$
- 10      Output  $(\vec{x}^*, \vec{y}^*)$ .

**Maximum-flow solution:** A faster algorithm for solving this problem uses a reduction to maximum flow. One may represent this problem using a flow network containing one vertex  $y_j$  for each investor  $j$ , one vertex  $x_i$  for each actor  $i$ , one source vertex  $s$ , and one sink vertex  $t$ . Connect  $s$  to each  $x_i$  with an edge of capacity  $a_i$ , and connect each  $y_j$  to  $t$  with an edge of capacity  $b_j$ . Finally, for each actor  $i$  and each investor  $j$ , if  $i \in L_j$  then connect  $x_i$  to  $y_j$  with an infinite capacity edge. Finding the maximum flow through this flow network will expose a minimum cut of this flow network.

We can find an  $(S, T = V \setminus S)$  cut in this flow network that has finite capacity, so the minimum  $(S, T)$  cut of this flow network must also have finite capacity. Observe that the finiteness of the minimum cut of this flow network must enforce the all-or-nothing constraint of the problem. Suppose some actor vertex  $x_i$  is in the  $S$  portion of the minimum cut, and suppose some actor vertex  $y_j$ , where  $i \in L_j$ , is in the  $T$  portion of this cut. Because  $i \in L_j$ , by construction of this flow network, there must exist an edge of infinite capacity from  $x_i$  to  $y_j$ . Therefore, there must exist an edge of infinite capacity from  $S$  to  $T$ , and the total capacity of this cut must be infinite. This contradicts our assertion that the minimum cut must have finite capacity, so this scenario cannot happen. Therefore, if an investor vertex  $y_j$  is in the  $T$  portion of this cut, then all actor vertices  $x_i$  where  $i \in L_j$  must also be in the  $T$  portion of this cut.

We can derive the optimal profit for this problem from the size of the minimum cut in this flow network. Note that the total profit from the optimal selection of actors  $i$  and investors  $j$  is

$$\sum_{j \in OPT} b_j - \sum_{i \in OPT} a_i.$$

Meanwhile, the capacity of the minimum cut is

$$\sum_{j: y_j \in S} b_j + \sum_{i: x_i \in T} a_i = \sum_j b_j - \sum_{j: y_j \in T} b_j + \sum_{i: x_i \in T} a_i.$$

The sum of all investor contributions is fixed for a given problem instance, so the capacity of the minimum cut must maximize the value of  $\sum_{j: y_j \in T} b_j - \sum_{i: x_i \in T} a_i$ , which is exactly the equation for the profit of the original problem. Consequently, the minimum cut of this flow network maximizes the profit of the original problem, and because maximum flow is equivalent to minimum cut, finding the maximum flow of this flow network gives us an optimal solution to the original problem.

A similar solution to this problem swaps the positions of the actor and investor nodes in this flow network, and therefore chooses the actors and investors on the  $S$  side of a minimum cut.

**Alternate attempts:** Many people tried to solve this problem using dynamic programming. The difficulty with such a solution is that this problem does not exhibit optimal substructure, so any feasible dynamic programming solution would be equivalent to an exponential time algorithm. Similarly, greedy strategies to this problem do not work due to this problem's lack of optimal substructure.

---

## Quiz 2

### Cover Sheet

Problem	Points	Grade	Initials
1	32		
2	18		
3	20		
4	30		

Name: \_\_\_\_\_

R01	R02	R03	R04	R05	R06
F10	F11	F12	F1	F2	F3
Rafael	Shaunak	Lin	Lin	Yu	Yu
R07	R08	R09	R10		
F11	F12	F1	F2		
Piotr	Piotr	Tom	Tom		

## INSTRUCTIONS

This take-home quiz contains 4 problems worth a total of 100 points. Your quiz solutions are due **at 11am on Monday, November 21, 2011**. Late quizzes will not be accepted unless you obtain a Dean's support or make prior arrangements with the course staff. You must submit your solutions electronically.

**Guide to this quiz:** For problems that ask you to design an efficient algorithm for a certain problem, your goal is to find the *most efficient algorithm possible*. Generally, the faster your algorithm, the more points you receive. For two asymptotically equal bounds, worst-case bounds are better than expected or amortized bounds. The best possible solution will receive full points if well written, but ample partial credit will be given for correct solutions, especially if they are well written. Bonus points may be awarded for exceptionally efficient or elegant solutions.

Plan your time wisely. Do not overwork, and get enough sleep. Your very first step should be to write up the most obvious algorithm for every problem, even if it is exponential time, and then work on improving your solutions, writing up each improved algorithm as you obtain it. In this way, at all times, you have a complete quiz that you could hand in.

**Policy on academic honesty:** The rules for this take-home quiz are like those for an in-class quiz, except that you may take the quiz home with you. As during an in-class quiz, you may not communicate with any person except members of the 6.046 staff about any aspect of the quiz, even if you have already handed in your quiz solutions. To communicate with the staff, you have to send an email to `6046-tas@csail.mit.edu`. We will not respond to your question if you send an email to the personal email of a staff member. In addition, you may not discuss any aspect of the quiz with anyone except the course staff **until November 30, 2011**. Note that this date is **after** the due date of the quiz.

This take-home quiz is “limited open book.” You may use your course notes, the CLRS textbook, and any of the materials posted on the course web page, but *no other sources whatsoever may be consulted*. For example, you may not use notes or solutions to problem sets, exams, etc. from other times that this course or other related courses have been taught. **You may not use any materials on the World-Wide Web, including OCW.** You probably won’t find information in these other sources that will help directly with these problems, but you may not use them regardless.

If at any time you feel that you may have violated this policy, it is imperative that you contact the course staff immediately. If you have any questions about what resources may or may not be used during the quiz, please send email to `6046-staff@csail.mit.edu`.

**Write-ups:** Type your answers up on separate files (the templates will be given), and submit all your answers electronically, as you would do in the problem set. We *strongly encourage* you to submit your problems typed in LaTeX, since this makes it easier for us to read your solutions and understand them.

**Your solutions are due, electronically, at 11am on Monday the 21st of November. There is no hardcopy submission option for this test.**

*Your write-up for a problem should start with a topic paragraph that provides an executive summary of your solution.* This executive summary should describe the problem you are solving, the techniques you use to solve it, any important assumptions you make, and the asymptotic bounds on the running time your algorithm achieves, including whether they are worst-case, expected, or amortized.

Write your solutions cleanly and concisely to maximize the chance that we understand them. When describing an algorithm, give an English description of the main idea of the algorithm. Adopt suitable notation. Use pseudocode if necessary to clarify your solution. Give examples, draw figures, and state invariants. A long-winded description of an algorithm's execution should not replace a succinct description of the algorithm itself. *Points will be taken off for overly convoluted solutions to the problems.*

Provide short and convincing arguments for the correctness of your solutions. Do not regurgitate material presented in class. Cite algorithms and theorems from CLRS, lecture, and recitation to simplify your solutions. Do not waste effort proving facts that can simply be cited.

Be explicit about running time and algorithms. For example, don't just say that you sort  $n$  numbers, state that you are using MERGE-SORT, which sorts the  $n$  numbers in  $O(n \lg n)$  time in the worst case. If the problem contains multiple variables, analyze your algorithm in terms of all the variables, to the extent possible.

Part of the goal of this quiz is to test your engineering common sense. If you find that a question is unclear or ambiguous, make reasonable assumptions in order to solve the problem, and state clearly in your write-up what assumptions you have made. Be careful what you assume, however, because you will receive little credit if you make a strong assumption that renders a problem trivial.

**Bugs:** If you think that you've found a bug, send email to `6046-staff@csail.mit.edu`. Corrections and clarifications will be sent to the class via email and posted on the class website. Check your email and the class website daily to avoid missing potentially important announcements.

**Good Luck!**

**Problem 1. Minimum Spanning Trees and Matchings [32 points] (4 parts)**

In all four parts of this problem,  $G = (V, E, w)$  is a connected, weighted, undirected graph. We define  $w(e)$  to be the weight of edge  $e$ .

- (a) Suppose that  $T$  is a minimum spanning tree of  $G$ . If we increase the weight of each edge of  $G$  by the same positive amount  $\delta$ , is  $T$  still guaranteed to be a minimum spanning tree? Give an argument for why it will be, or a counterexample showing that it may not be.
- (b) Let  $e$  be an edge of maximal weight in  $G$ . Suppose we add a single new edge  $e'$  to  $G$  to obtain the graph  $G' = (V, E \cup \{e'\}, w)$ . The weight function  $w$  is extended so that  $w(e')$  is defined, and  $w(e')$  is less than  $w(e)$  – that is, the new edge does not have maximal weight. Show that the total weight of a minimum spanning tree of  $G'$  is at least  $w(e') - w(e)$  plus the total weight of a minimum spanning tree of  $G$ .
- (c) We now consider maximum-weight matchings in  $G$ . Suppose that we increase the weight of each edge of  $G$  by the same positive amount  $\delta$ . Is it possible that a matching of maximum weight in  $G$  will no longer be maximum-weight after the change? Give an argument for why it will be, or a counterexample showing that it may not be.
- (d) Finally, we consider maximum-cardinality matchings in  $G$ . A maximum-cardinality matching is a matching with the maximum possible number of edges. Suppose we add a single new edge  $e'$  to  $G$  to obtain the graph  $G'$  with edge set  $E \cup \{e'\}$ . Show that the size of a maximum-cardinality matching in  $G'$  is not more than one larger than the size of a maximum-cardinality matching in  $G$ .

**Solution:** (a) Yes. Since each tree in  $G$  has  $(|V| - 1)$  edges, after the increase of weight, the total weight of each tree will be increased by the same amount  $(|V| - 1)\delta$ .  $T$  is therefore still the minimum spanning tree in the new graph.

(b) Let  $T$  and  $T'$  be the minimum spanning tree of  $G$  and  $G'$  respectively. By adding  $\{e'\}$  to  $T$ , we create a cycle in  $(T \cup \{e'\})$  that contains  $\{e'\}$ . The new minimum spanning tree  $T'$  in  $G'$  can be obtained by replacing the edge that has maximum weight in the cycle with  $\{e'\}$ . We call the edge  $e_m$ , then  $w(T') = w(T) - w(e_m) + w(e') \geq w(T) - w(e) + w(e')$ .

(c) Yes. it is possible. A simple example is a graph with two vertices and one edge  $e$  connecting the two vertices that has negative weight  $w(e)$ . Before the increase of weight, the maximum-weight matching is empty. If  $\delta > -w(e)$ , then after the increase of weight, the maximum weight matching is  $\{e\}$ .

(d) Let the the maximum-cardinality matching in  $G$  and  $G'$  be  $M$  and  $M'$  respectively. We consider two cases. First, If  $e' \notin M'$ , then  $M'$  is a feasible matching in  $G$ . By the definition of maximum carnality matching,  $|M'| \leq |M|$ . Second, if  $e' \in M'$ , then  $M' \setminus \{e'\}$  is a feasible matching in  $G$ ,  $|M' \setminus \{e'\}| \leq |M|$  or equivalently  $|M'| \leq |M| + 1$ . In either case, the size of  $M'$  is at most one larger than the size of  $M$ .

**Problem 2. Gerrymandering in LineLand [18 points]**

There are  $n$  residents of LineLand, residing at the points  $1, 2, \dots, n$  of the Line which is their world. There are two political parties in LineLand: the Cardinals and the Cyans. It is known to everyone if LineLander  $i$  is a Cardinal or a Cyan, for each  $i$ ,  $i = 1, 2, \dots, n$ .

It is time to "redistrict" – i.e. to divide LineLand into  $m$  political districts. Here  $m$  is significantly smaller than  $n$ .

Each district must be a contiguous segment of LineLand: it must include LineLander  $i$  up to LineLander  $j$ , inclusive, for some  $i$  and  $j$ ,  $i \leq j$ .

Thus, the first district goes from 1 up to  $a_1$ , the second from  $a_1 + 1$  up to  $a_2$ , and so on, until the last district is from  $a_{m-1} + 1$  up to  $a_m = n$ .

To be fair, each district must have  $n/m$  LineLanders, more or less. More precisely, by the LineLand Constitution, a district must contain at least  $r = \lceil n/(2m) \rceil$  LineLanders, and may not contain more than  $s = \lfloor 3n/(2m) \rfloor$  LineLanders.

Once the districts are drawn, the LineLand Parliament is created by picking one resident uniformly at random from each district. (So the Parliament has size  $m$ .)

Explain what algorithm you would use to determine the districts (i.e., to determine  $a_1, \dots, a_m$ ), given  $n$ , the number  $m$  of districts desired, and the political affiliation of each LineLander, in such a way that would maximize the expected number of Cardinal members in Parliament. (Also explain how efficient your algorithm is, as a function of  $n$  and  $m$ .)

**Solution:** We will solve this problem using dynamic programming. Our subproblem will be

$$f(i, j) = \text{most expected Cardinals from dividing } i, \dots, n \text{ into } j \text{ regions.}$$

Thus the solution to our original problem is  $f(1, m)$ . Our base cases will be

$$\begin{aligned} f(n, 0) &= 0, \\ f(i, 0) &= -\infty, \quad \forall i < n, \\ f(i, j) &= -\infty, \quad \forall i > n - r, j > 0. \end{aligned}$$

The second and third base cases handle the cases where it is impossible to correctly divide up the region. Given these base cases, our recurrence will be

$$f(i, j) = \max_{k \in \{r, \dots, s\}} \left( f(i + k, j - 1) + \frac{\# \text{Cardinals in } i, \dots, k - 1}{k} \right).$$

Essentially, the dynamic program tries all possible district sizes for the first district in the region  $i, \dots, n$  and selects the best one by evaluating the expected contribution of that district plus the most expected Cardinals achievable from the remaining districts.

The dynamic program is  $O(nm)$  subproblems, and each subproblem takes  $O(n/m)$  time to evaluate (since  $s$  is  $O(n/m)$ ), so that total running time is  $O(n^2)$ .

**Problem 3. The Ivory Tower [20 points]**

A group of undergraduates, graduate students and professors work in an ivory tower. Each professor  $p$  is willing to work with a subset  $U(p)$  of the undergrads and a subset  $G(p)$  of the graduate students; a *research team* is comprised of an undergrad, a graduate student and a professor who is willing to work with both. No one can belong to more than one research team.

Find an efficient algorithm for determining the maximum number of research teams that can be formed given the professors' preferences.

**Solution:**

**Maximum Flow:** We will solve this problem with maximum flows. Let's first explore the following idea where a valid team is represented by a flow from undergraduate  $-i$ , professor  $-i$ , graduate. Like usual, we create a node for each person in  $P, U, G$ . And connect a directed edge  $(u_i, p_j)$  if professor  $p_j$  is willing to work with undergraduate  $u_i$ . Similarly, connect a directed edge  $(p_j, g_k)$  if professor  $p_j$  is willing to work with graduate student  $g_k$ . We then create a source  $s$  connected to each undergraduate as well as a sink  $t$  connected from every graduate. All edges have capacity 1. However, there is a problem with this construction so far. Notice that each professor can only be part of 1 research group, we must ensure a "vertex capacity" of 1 on each professor node. We can accomplish this by breaking each professor node into two nodes  $p$  and  $p'$ , and create an edge of capacity 1 from  $p$  to  $p'$ . All the incoming edges from undergraduates will be connected to  $p$ , whereas all the outgoing edges to graduates will be connected from  $p'$ . Thus, a flow path from  $s$  to  $t$  will go through  $u, p, p',$  and  $g$ . The two node gadget with a capacity 1 edge for each professor ensures that there cannot be multiple flows across it.

**Maximum Matching:**

We will prove that the size of the maximum matching  $M$  in the graph used for maximum flow (without the source and sink) has value  $P + k$ , where  $k$  is the maximum number of research teams. Given this, we can thus find  $k$  in  $O((P + U + G)^{2.5})$  time as the maximum matching can be found in  $O(\sqrt{VE})$ .

First we will argue that if there is a valid assignment of  $k$  research teams, then there is a matching of size  $P + k$ . For each of the  $k$  research teams, match the undergrad to the first professor vertex and the grad student to the second professor vertex, for a total of  $2k$  matched edges. For each of the professors not in any of the  $k$  research teams, neither of its vertices is matched so we can select the edge between them, for a total of  $P - k$  edges. Thus we have a matching of size  $2k + P - k = P + k$ .

Next we will argue that if there is a maximal matching  $M$  of size  $P + k$ , then there is a valid assignment of  $k$  research teams. Observe that every edge in the graph is adjacent to at least one professor vertex. Thus, we can write  $M = \sum_{p_i \in P} M(p_i)$  where  $M(p_i)$  is the number of edges in  $M$  adjacent to the vertices corresponding to professor  $p_i$ . Observe that in any maximal matching, for a given professor, at least one of its two vertices must be matched. This is because there is an edge between the two vertices so if neither was matched then we could match them to each other. Thus,  $M(p_i) \geq 1$ . Since  $p_i$  corresponds to exactly two vertices,  $M(p_i) \leq 2$ . Therefore  $M = (P - k) + 2k = P + k$  where  $k$  is the number of professors with two edges in the matching. If a professor has two edges in the matching, then one must go to an undergrad and one must go to a grad student thus the set of  $k$  professors matched to two vertices corresponds to a valid assignment of  $k$  research teams.

**Problem 4. Red-Blue Network Flow [30 points] (2 parts)**

You are given a flow network  $G = (V, E)$  with distinguished source vertex  $s$  and distinguished sink vertex  $t$ . As usual, the edges in  $E$  are directed edges. The edges in  $E$  are now, however, of two types: blue and red. The blue edges are of the usual type: each blue edge  $e$  has a capacity  $c(e)$  that is the *maximum possible* amount of flow that can pass through that edge. The red edges are built out of strange alien technology: each red edge  $e$  must have at least a certain amount  $d(e)$  of flow passing through that edge. That is,  $d(e)$  is the *minimum required* amount of flow that must pass through that edge.

Other than the fact that some edges are now red, everything is as usual (e.g. flow must be conserved at all intermediate vertices, etc.). Each edge is either red or blue. We call such a network problem a “red-blue network flow problem”. As usual, the goal is to find a flow of maximum possible value.

- (a) Show that it is possible to compute a maximum red-blue network flow, or determine that no legal flow exists, in polynomial time.
- (b) You now discover how to “turn off” red edges. If a red edge  $e$  is on, at least  $d(e)$  units of flow must pass over it; if it is off, no flow can pass over it. You can turn some red edges on and others off. Show that it is now NP-complete to determine the maximum flow possible in a red-blue network.

**Solution:** (a) The problem can be formulated as a linear programming problem. Let  $f(u, v)$  be the flow on edge  $(u, v) \in E$ ,  $B$  and  $C$  be the sets of blue and red edges respectively. The LP is

$$\begin{aligned} & \max \sum_v f(s, v) \quad s.t. \\ & f(u, v) \leq c(u, v) \quad \text{for any } (u, v) \in B, \quad f(u, v) \geq d(u, v) \quad \text{for any } (u, v) \in C \\ & f(u, v) \geq 0, \quad \sum_v f(u, v) = \sum_w f(w, u) \quad \text{for any } u \in \{V - s, t\} \end{aligned} \quad (1)$$

The LP can be solved in weakly polynomial time by ellipsoid or interior point algorithm.

(b). Since the NP-complete is for decision problems, the question can be interpreted as determine if there is a legal  $k$  flow in the flow network. To prove the problem is NP-hard, we can reduce the SUBSET-SUM problem to the red-blue flow network problem. The SUBSET-SUM is the decision problem that determines if there is a subset in  $S = \{x_1, x_2, \dots, x_n\}$  whose sum equal to  $k$ .

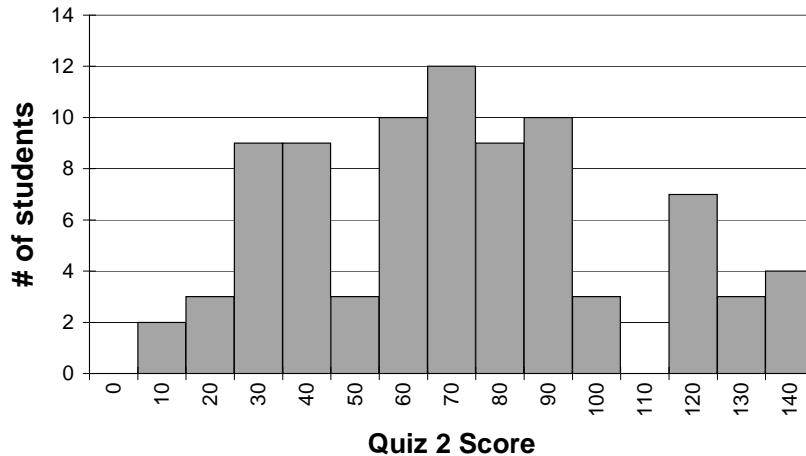
We first create a red-blue flow network with source  $s$ , sink  $t$  and another  $n$  vertices. The edges in the flow network are as follows,

$(s, u_i)$  for  $i = 1, 2, \dots, n$  are red edges that connects  $s$  to  $n$  vertices that represent  $n$  variables in the subset.  $c(s, u_i) = x_i$

$(u_i, t)$  for  $i = 1, 2, \dots, n$  are blue edges that connects  $u$  to sink  $t$ .  $d(u_i, t) = x_i$ .

The way we create the graph guarantees that the flow from  $s$  to  $t$  through  $u_i$  can either be  $x_i$  if the red edge  $(s, u_i)$  is on or 0 if the red edge is off. The flow in the red-blue network is therefore equal to the sum of the  $x_i$  whose corresponding red edge is on. Determine if there exists a flow  $k$  is the same as determine if there is a subset sum  $k$ . Creating the red-blue network takes polynomial running time. Since SUBSET-SUM is NP-complete, the red-blue flow network problem is NP-hard. Given the assignment of flow on each edge and the on/off assignment of red edges, it takes polynomial time to check if flow  $k$  exists, so the problem is in NP. Combining with the NP-hardness proof, the red-blue network-flow problem is NP-complete.

## Quiz 2 Solutions



### Problem 1. Ups and downs

Moonlighting from his normal job at the National University of Technology, Professor Silvermeadow performs magic in nightclubs. The professor is developing the following card trick. A deck of  $n$  cards, labeled  $1, 2, \dots, n$ , is arranged face up on a table. An audience member calls out a range  $[i, j]$ , and the professor flips over every card  $k$  such that  $i \leq k \leq j$ . This action is repeated many times, and during the sequence of actions, audience members also query the professor about whether particular cards are face up or face down. The trick is that there are no actual cards: the professor performs these manipulations in his head, and  $n$  is huge.

Unbeknownst to the audience, the professor uses a computational device to perform the manipulations, but the current implementation is too slow to work in real time. Help the professor by designing an efficient data structure that supports the following operations on  $n$  cards:

- $\text{FLIP}(i, j)$ : Flip over every card in the interval  $[i, j]$ .
- $\text{IS-FACE-UP}(i)$ : Return TRUE if card  $i$  is face up and FALSE if card  $i$  is face down.

**Solution:** Let  $F$  be the number of FLIP operations requested by the audience. Notice that performing a single flip  $\text{FLIP}(i, j)$  is equivalent to performing two flips,  $\text{FLIP}(i, \infty)$  and  $\text{FLIP}(j + 1, \infty)$ .

One fast solution is to use a dynamic order-statistic tree, where an element with key  $i$  represents a flip of the interval  $[i, \infty)$ . For both  $x = i$  and  $x = j + 1$ , FLIP inserts  $x$  into the tree  $T$  if  $x \notin T$ , and deletes  $x$  from  $T$  if  $x \in T$ . Is-FACE-UP is implemented by returning true if the number of elements in the tree less than  $i$  (call it  $z$ ) is even, and false if  $z$  is odd. One way to compute  $z$  is to insert  $i$  into  $T$ , set  $z$  to be one less than the rank of  $i$ , and then delete  $i$  from  $T$ . This data structure requires  $O(\min\{F, n\})$  space and supports FLIP and Is-FACE-UP operations each in  $O(\lg(\min\{F, n\}))$  time. A completely correct solution of this type received full credit.

The following list describes other types of solutions that students submitted and the approximate number of points awarded to each.

1. Another solution is to create a static 1-d range tree  $T$  containing the elements from 1 to  $n$  as its keys. Each node  $x$  in the tree stores a bit that corresponds to a flip of the interval of all elements in the subtree rooted at  $x$ .  $\text{FLIP}(i, j)$  performs a range query on  $[i, j]$ , and flips the stored bit for the  $O(\lg n)$  disjoint subtrees that are found by the query. Is-FACE-UP( $i$ ) walks down the range tree to the node for  $i$  and returns true if the sum of the bits of nodes along the path from the root to  $i$  is even. This data structure uses  $O(n)$  space and supports both operations in  $O(\lg n)$  time. This type of solution received about 20 points.
2. Other students assumed that  $F \ll n$ , and stored each of the flip intervals  $[i, j]$  in an interval tree. The Is-FACE-UP( $i$ ) procedure simply queries the interval tree and returns true if the number of intervals that overlap  $i$  is even. If the tree stores overlapping flip intervals, then FLIP and Is-FACE-UP run in  $O(\lg F)$  time and  $O(F \lg F)$  time, respectively. If we do not allow overlapping intervals in the tree, then FLIP runs in  $O(F \lg F)$  time but Is-FACE-UP runs in  $O(\lg F)$  time. This type of solution received about 13 points.
3. A simple slow solution is to use a bit-vector of length  $n$  to record which bits are flipped. This solution supports FLIP and Is-FACE-UP in  $O(j - i)$  and  $O(1)$  time, respectively. Similarly, another simple solution that stores all  $F$  flip-intervals in a linked list supports FLIP and Is-FACE-UP in  $O(1)$  and  $O(F)$  time, respectively. Either of these solutions received about 8 points.
4. Any solution that ran slower than the simple slow solution (for example,  $O(\lg n + j - i)$  for FLIP) received about 5 points.

### Problem 2. The Data Center

The world-famous architect Gary O'Frank has been commissioned to design a new building, called the Data Center. Gary wants his top architectural protégé to design a scale model of the Data Center using precision-cut sticks, but he wants to preclude the model from inadvertently containing any right angles. Gary fabricates a set of  $n$  sticks, labeled  $1, 2, \dots, n$ , where stick  $i$  has length  $x_i$ . Before giving the sticks to the protégé, he shows them to you and asks you whether it is possible to create a right triangle using any three of the sticks. Give an efficient algorithm for determining whether there exist three sticks  $a$ ,  $b$ , and  $c$  such that the triangle formed from them — having sides of lengths  $x_a$ ,  $x_b$ , and  $x_c$  — is a right triangle (that is,  $x_a^2 + x_b^2 = x_c^2$ ).

**Solution:** Let  $X[1..n]$  be the array with stick sizes. We are asked to determine whether there exist distinct indices  $i$ ,  $j$ , and  $k$  such that  $X[i]^2 + X[j]^2 = X[k]^2$ .

As an initial step we sort  $X[1..n]$  in ascending order. This can be done in worst-case  $O(n \lg n)$  time by for example heapsort. As soon as the array is sorted we first observe that it is sufficient to check  $X[i]^2 + X[j]^2 = X[k]^2$  for indices  $i$ ,  $j$ , and  $k$  with  $i < j < k$ .

In order to obtain a worst-case  $O(n^2)$  algorithm, we increase  $k$  from 1 to  $n$  in an outerloop and we search in linear time in an innerloop for indices  $i$  and  $j$ ,  $i < j < k$ , with  $X[i]^2 + X[j]^2 = X[k]^2$ .

```

1 Sort  $X[1..n]$  in ascending order
2  $k \leftarrow 1$ 
3 while  $k \leq n$ 
4   do  $i, j \leftarrow 1, k - 1$ 
5     while  $i < j$ 
6       if  $X[i]^2 + X[j]^2 = X[k]^2$  then return true
7       if  $X[i]^2 + X[j]^2 < X[k]^2$ 
8         then  $i \leftarrow i + 1$ 
9       else  $j \leftarrow j - 1$ 
10  return false

```

We will prove the invariant:

$$[1 \leq i' < j' \leq k - 1 \text{ and } X[i']^2 + X[j']^2 = X[k]^2] \text{ implies } [i \leq i' < j' \leq j].$$

Initially,  $i = 1$  and  $j = k - 1$  and the invariant holds. For the inductive step, assume that the invariant holds when we enter the innerloop. Notice that  $X[1..n]$  is sorted in increasing order. If  $X[i]^2 + X[j]^2 < X[k]^2$ , then, for  $i < j' \leq j$ ,

$$X[i]^2 + X[j']^2 \leq X[i]^2 + X[j]^2 < X[k]^2$$

and the invariant holds for  $i \leftarrow i + 1$ . If  $X[i]^2 + X[j]^2 > X[k]^2$ , then, for  $i \leq i' < j$ ,

$$X[i']^2 + X[j]^2 \geq X[i]^2 + X[j]^2 > X[k]^2$$

and the invariant holds for  $j \leftarrow j - 1$ .

If the innerloop finishes, then  $i = j$  and the invariant shows that there are no indices  $i'$  and  $j'$ ,  $1 \leq i' < j' \leq k - 1$ , such that  $X[i']^2 + X[j']^2 = X[k]^2$ . This proves the correctness of the algorithm. The innerloop has worst-case  $O(n)$  running time, hence, together with the outerloop the algorithm runs in worst-case  $O(n^2)$  time.

**Problem 3. Nonnegativizing a matrix by pivoting**

A matrix  $M[1 \dots n, 1 \dots n]$  contains entries drawn from  $\mathbb{R} \cup \{\infty\}$ . Each row contains at most 10 finite values, some of which may be negative. The goal of the problem is to transform  $M$  so that every entry is nonnegative by using only *pivot* operations:

```

PIVOT( $M, i, x$ )
1 for  $j \leftarrow 1$  to  $n$ 
2   do  $M[i, j] \leftarrow M[i, j] + x$ 
3    $M[j, i] \leftarrow M[j, i] - x$ 

```

Give an efficient algorithm to determine whether there exists a sequence of pivot operations with various values for  $i$  and  $x$  such that, at the end of the sequence,  $M[i, j] \geq 0$  for all  $i, j = 1, 2, \dots, n$ .

**Solution:**

Suppose that we have a sequence  $\langle \text{PIVOT}(M, i_1, x_1), \text{PIVOT}(M, i_2, x_2), \dots, \text{PIVOT}(M, i_k, x_k) \rangle$  of pivots. Since each pivot operation simply adds to columns and subtracts from rows, by the associativity and commutativity of addition, the order of the pivot operations is irrelevant. Moreover, since  $\text{PIVOT}(M, i_1, x_1)$  followed by  $\text{PIVOT}(M, i_2, x_2)$  is equivalent to  $\text{PIVOT}(M, i_1, x_1 + x_2)$ , there is no need to pivot more than once on any index. Thus, any sequence of pivots is equivalent to a sequence of exactly  $n$  pivots  $\langle \text{PIVOT}(M, 1, x_1), \text{PIVOT}(M, 2, x_2), \dots, \text{PIVOT}(M, n, x_n) \rangle$ , where some of the  $x_i$  may be 0.

The only pivots that affect a matrix entry  $M[i, j]$  are  $\text{PIVOT}(M, i, x_i)$  and  $\text{PIVOT}(M, j, x_j)$ . Thus, after the entire sequence of pivots has been performed, the resulting matrix  $M'$  has entries  $M'[i, j] = M[i, j] + x_i - x_j$ . We want every entry  $M'[i, j]$  to be nonnegative, which is to say that  $M[i, j] + x_i - x_j \geq 0$ , or  $x_j - x_i \leq M[i, j]$ . Thus, we only need to solve a set of difference constraints. We can ignore the difference constraints where  $M[i, j] = \infty$ , which leaves at most  $10n$  difference constraints in  $n$  variables. It takes us  $O(n^2)$  time to find the at-most  $10n$  finite entries, and we can use the Bellman-Ford algorithm [CLRS, Section 24.4] to solve them in  $O(n \cdot (10n)) = O(n^2)$  time, for a total of  $O(n^2)$  time.

#### Problem 4. Augmenting the Queueinator™

By applying his research in warm fission, Professor Uriah's company is now manufacturing and selling the Queueinator™, a priority-queue hardware device which can be connected to an ordinary computer and which effectively supports the priority-queue operations `INSERT` and `EXTRACT-MIN` in  $O(1)$  time per operation. The professor's company has a customer, however, who actually needs a "double-ended" priority queue that supports not only the operations `INSERT` and `EXTRACT-MIN`, but also `EXTRACT-MAX`. Redesigning the Queueinator™ hardware to support the extra operation will take the professor's company a year of development. Help the professor by designing an efficient double-ended priority queue using software and one or more Queueinator™ devices.

#### Solution:

Keep two Queueinators,  $MIN$  and  $MAX$ , where  $MAX$  has the keys negated, of roughly equal size and keep an element  $mid$  which is the smallest element in  $MAX$ . While inserting an element with key  $k$ , compare it to  $mid$  and insert it in  $MIN$  if  $k < mid$  and into  $MAX$  otherwise. For `EXTRACT-MIN`, extract from  $MIN$  and for `EXTRACT-MAX`, extract from  $MAX$ . If one empties, extract all the elements from the other into a sorted array. Split the array into half and insert the smaller half into  $MIN$  and the larger half into  $MAX$ . This solution provides  $O(1)$  amortized cost for all operations.

**Proof of Correctness:** The invariant is that all the elements in  $MIN$  are smaller than  $mid$  and all the elements in  $MAX$  are larger than or equal to  $mid$ . The other invariant is that  $MIN$  and  $MAX$  together contain all the elements that have been inserted and not yet been extracted. The invariants are maintained during all operations. Therefore, extractions return the correct results.

**Running time analysis:** The potential function is  $\Phi(i) = 2|MIN| - |MAX|$ , where  $|MIN|$  and  $|MAX|$  are the number of elements in the  $MIN$  and  $MAX$  queueinators respectively. Both  $MIN$  and  $MAX$  are empty initially, and the potential is 0. The potential obviously never becomes negative. Let us look at the amortized costs.

1. **INSERT** Inserts into  $MIN$  or  $MAX$ . The real cost is 1 and the potential either increases by 2 (if you inserted into the bigger queue) or decreases by 1 (if you insert into the smaller queue). Thus the cost is always  $O(1)$ .
2. **EXTRACT-MIN** If the  $MIN$  is non-empty, then the real cost is 1 and the potential either increases or decreases by 2. If  $MIN$  is empty, then you have to rebalance the queues. The potential before the rebalance is equal to  $2 * \text{the number of elements in } MAX$ , and the potential after the rebalance is either 0 (or 2 if the number of elements is odd, but that doesn't change much). The real cost of rebalancing is  $2 * \text{the number of elements in } MAX$ , since all the elements are extracted and then inserted again. Thus the change in potential pays for the rebalancing, and the amortized cost is  $O(1)$ .
3. **EXTRACT-MAX** It is analogous to `EXTRACT-MIN`.

Therefore the amortized cost of the operations is  $O(1)$ .

There were other good solutions, some of them are given below

1. Use 4 queueinators where 2 of them to keep track of the deleted elements. Those solutions lost a couple of points due to the use of the extra hardware.
2. Use extra fields in the items to keep track of deletions. These also lost a couple of points due to extra assumptions that you can change the elements.
3. Use hash tables to keep track of deleted elements. This only provides expected  $O(1)$  amortized time, and assumes that the keys are integers in a range. Therefore, they lost about 5 points.
4. Use direct access tables to keep track of deleted elements, lost about 6-7 points due to the extra space.

The analysis was very important for the solutions and the solutions that had incomplete or incorrect running time analysis lost points for that. In addition, a convincing correctness argument was required for full credit. People who gave the correct algorithm but did not claim or prove amortized bounds lost many points.

### Problem 5. Spam distribution

Professor Hormel is designing a spam distribution network. The network is represented by a rooted tree  $T = (V, E)$  with root  $r \in V$  and nonnegative edge-weight function  $w : E \rightarrow \mathbb{R}$ . Each vertex  $v \in V$  represents a server with one million email addresses, and each edge  $e \in E$  represents a communication channel that costs  $w(e)$  dollars to purchase. A server  $v \in V$  receives spam precisely if the entire path from the root  $r$  to  $v$  is purchased. The professor wants to send spam from the root  $r$  to  $k \leq |V|$  servers (including the root) by spending as little money as possible. Help the professor by designing an algorithm that finds a minimum-weight connected subtree of  $T$  with  $k$  vertices including the root. (For partial credit, solve the problem when each vertex  $v \in V$  has at most 2 children in  $T$ .)

#### Solution:

**Executive Overview.** The solution to this problem is based on dynamic programming. We define  $k|V|$  subproblems, one for each combination of a vertex  $v \in V$  and a number  $k' \leq k$ . Each subproblem is of the form “Find the minimum-weight connected subtree of rooted at  $v$  with  $k'$  vertices.” Each subproblem can be solved in  $O(k)$  time, resulting in a running time of  $O(k^2|V|)$ .

**Notation.** Throughout this solution, we use the following notation:

- $\text{MWCT}(v, i)$ : a minimum-weight connected subtree of  $i$  nodes rooted at  $v$
- $\text{child}(v, i)$ : child  $i$  of node  $v$
- $w(v, i)$ : the weight of the edge from  $v$  to child  $i$
- $W(S)$ : the sum of the weights of all the edges in tree  $S$
- $\deg(v)$ : the number of children of node  $v$

**Optimal Substructure.** This problem exhibits both optimal substructure and overlapping subproblems. For intuition, consider a binary tree  $T$ . Let  $v$  be a node in the tree, and consider a MWCT of  $T$  rooted at  $v$  with  $\ell$  nodes. If  $T_1$  is the left subtree of  $v$  and has  $\ell_1$  nodes, then  $T_1$  is a MWCT rooted at  $v$ 's left child with  $\ell_1$  nodes. Similarly, if  $T_2$  is the right subtree of  $v$  and has  $\ell_2$  nodes, then  $T_2$  is a MWCT rooted at  $v$ 's right child with  $\ell_2$  nodes. The proof follows by a cut-and-paste argument. Assume that  $T_1$  is *not* a MWCT rooted at  $v$ 's left child. Then there is another tree,  $S$ , rooted at  $v$ 's left child with  $\ell_1$  nodes and with less weight than  $T_1$ . We can then reduce the weight of the tree rooted at  $v$  by substituting  $S$  for tree  $T_1$ , contradicting our assumption that the tree rooted at  $v$  is a minimum-weight connected subtree with  $k$  nodes.

We can now observe how to use the overlapping subproblems. Once we have calculated the  $\text{MWCT}(v, \ell)$  for all  $v \in V$  and all  $\ell \leq k'$ , we can determine the  $\text{MWCT}(v, k' + 1)$ . In particular, we choose subtrees rooted at  $v$ 's children that contain  $k'$  nodes in total. That is, if there are  $\ell$  nodes in the left subtree of  $v$ , then there must be  $k' - \ell$  nodes in the right subtree. We choose  $\ell$  to

split the tree between  $v$ 's left and right children to minimize the total weight. We now proceed to generalize this to arbitrary degree trees, and explain the algorithm in more detail.

**Algorithm Description.** Let  $T[v]$  be the subtree of  $T$  rooted at  $v$ . Let  $T[v, c]$  be the subtree of  $T$  rooted at  $v$  consisting of  $v$  and the trees rooted at children  $1 \dots c$ . More formally,  $T[v, c] = v \cup \{T[w] : w = \text{child}_v, i, 1 \leq i \leq c\}$ .

We create two  $|V| \times k \times |V|$  arrays:  $C[1 \dots |V|, 1 \dots k, 1 \dots |V|]$  and  $B[1 \dots |V|, 1 \dots k], 1 \dots |V|$ .  $C[v, k', c]$  holds the cost of the minimum-weight connected subtree of  $T[v, c]$  with  $k'$  nodes. The array  $B$  is used to reconstruct the tree after the dynamic program has terminated.  $B[v, k', c]$  holds the number of children in the subtree of the MWCT of  $T[v, c]$  with  $k'$  nodes rooted at child  $c$ .

Notice that the resulting arrays are three-dimensional tables of size  $k|V|^2$ . Fortunately, we will only have to fill in  $k|V|$  of the entries, leaving the rest empty/uninitialized, as we will see when the algorithm is analyzed. (For simplicity, we assume that arbitrary-sized memory blocks can be allocated in  $O(1)$  time. If memory allocation is more costly, we can reduce the memory usage to  $O(k|V|)$ .)

The main procedure to calculate the MWCT of  $V$  with  $k$  nodes is as follows:

```

MWCT( $V, k$ )
1  for all  $v \in V$ 
2    do for  $c = 1$  to  $\deg(v)$ 
3      do  $C[v, 1, c] \leftarrow 0$ 
4      do  $B[v, 1, c] \leftarrow 0$ 
5  for  $\ell = 2$  to  $k$ 
6    do for all  $v \in V$ 
7      do  $w \leftarrow \text{child}(v, 1)$ 
8       $C[v, \ell, 1] \leftarrow w(v, w) + C[w, \ell - 1, \deg(w)]$ 
9       $B[v, \ell, 1] \leftarrow \ell - 1$ 
10     for  $c = 2$  to  $\deg(v)$ 
11       do  $i \leftarrow \text{FIND-NUM-CHILDREN}(v, \ell, c)$ 
12        $B[v, \ell, c] \leftarrow i$ 
13       if  $i = 0$  then  $C[v, \ell, c] \leftarrow C[v, \ell, c - 1]$ 
14       if  $i = \ell$  then  $C[v, \ell, c] \leftarrow w(v, w) + C[w, \ell - 1, \deg(w)]$ 
15       if  $0 < i < \ell$  then  $C[v, \ell, c] \leftarrow C[v, \ell - i] + w(v, w) + C[w, i, \deg(w)]$ 

```

We begin in lines 1–4 by initializing  $C[v, 1, c] = 0$  for all  $v \in V$  and all  $c \leq |V|$ . These are the smallest subproblems considered: the one node minimum-weight connected subtree rooted at  $v$  consists simply of  $v$  itself, and hence has cost zero.

We proceed with three nested loops to fill in the arrays. In the outer loop (line 5), we iterate  $\ell$  from 2 to  $k$ . In the second loop (line 6), we iterate  $v$  over all nodes in  $V$ . In the inner loop (line 10), we iterate  $c$  from 2 to  $\deg(v)$ .

Within the loop, we are examining the subtree  $T[v, c]$  and the node  $w = \text{child}(v, c)$ . Let  $S$  be the MWCT of  $T[v, c]$  containing  $\ell$  nodes. We want to calculate  $C[v, \ell, c]$ , the cost of  $S$ .

First consider the case where  $c = 1$ . In this case, all the nodes in  $S$  are in the subtree rooted at  $w$ . Therefore,  $S$  consists of node  $v$  and the MWCT of  $w$  with  $\ell - 1$  nodes.

Now consider the case where  $c \geq 2$ . Notice that some of the nodes in  $S$  may be in subtrees rooted at the first  $c - 1$  children of  $v$ , and some may be in the subtree rooted at  $w$ . The function  $\text{FIND-NUM-CHILDREN}(v, \ell, c)$  calculates the number of nodes in  $S$  that are in the subtree rooted at  $w$ ; the remaining  $\ell - i - 1$  are in the subtrees rooted at the first  $c - 1$  children of  $v$ .

It is then easy to determine the cost of  $S$ . If no nodes in  $S$  are in the subtree rooted at  $w$ , then  $S$  is just the MWCT of  $T[v, c - 1]$  with  $\ell$  nodes (line 11). If all the nodes in  $S$  are in the subtree rooted at  $w$ , then  $S$  consists of node  $v$  along with the MWCT of  $w$  with  $\ell - 1$  nodes (line 12). Finally, if  $i$  is between 0 and  $\ell$ , then  $S$  consists of the MWCT of  $T[v, c - 1]$  with  $\ell - i$  nodes, plus the MWCT of  $w$  with  $i$  nodes (line 13).

It remains to discuss the  $\text{FIND-NUM-CHILDREN}$  procedure:

```

FIND-NUM-CHILDREN( $v, \ell, c$ )
1    $w \leftarrow \text{child}(v, c)$ 
2    $\text{min-weight} \leftarrow C[v, \ell, c - 1]$ 
3    $\text{num} \leftarrow 0$ 
4   for  $i = 1$  to  $\ell - 2$ 
5     do  $\text{wt} \leftarrow C[v, \ell - i, c - 1] + w(v, c) + C[w, i, \deg(w)]$ 
6       if  $\text{wt} < \text{min-weight}$ 
7         then  $\text{min-weight} \leftarrow \text{wt}$ 
8          $\text{num} \leftarrow i$ 
9   if  $\text{min-weight} > w(v, w) + C[w, \ell - 1, \deg(w)]$ 
10    then  $\text{num} \leftarrow \ell$ 
11   return  $\text{num}$ 
```

The  $\text{FIND-NUM-CHILDREN}$  procedure compares all possible ways of dividing  $\ell$  nodes between  $T[v, c - 1]$  and the subtree rooted at  $w = \text{child}(v, c)$ . First, it considers the case where all  $\ell$  children are in  $T[v, c - 1]$  and zero children are in the tree rooted at  $w$  (line 2–3). Next, it iterates through the loop  $\ell - 2$  times, placing  $\ell - i$  nodes in  $T[v, c - 1]$  and  $i$  nodes in the subtree rooted at  $w$  (lines 4–8). Finally, it considers the case where  $\ell - 1$  children are in the subtree rooted at  $w$  (lines 9–10). It then returns the choice which minimizes the weight.

The final procedure in the algorithms prints out the tree, performing a depth-first traversal of the resulting tree:

```

PRINT-TREE( $v, k$ )
1 PRINT( $v$ )
2  $\ell \leftarrow k$ 
3 for  $c = \deg(v)$  downto 1
4   do if  $\ell > 0$  and  $B[v, \ell, c] \neq 0$ 
5     then  $w \leftarrow \text{child}(v, c)$ 
6       PRINT-TREE( $v, B[v, \ell, c]$ )
7        $\ell \leftarrow \ell - B[v, \ell, c]$ 

```

The PRINT-TREE procedure starts with the last child of  $v$ , and examines  $B[v, \ell, c]$  to determine the number of nodes in the subtree root at child  $c$ . It then recurses into that subtree, and updates the number of nodes remaining in the MWCT.

**Running Time.** First, examine lines 1–4 of MWCT: the two loops iterate over every edge of every node, resulting on  $O(|E|) = O(|V|)$  operations. Next, the for-loop on line 5 repeats  $O(k)$  time. Now we examine lines 6–15. The innermost loop executes once for every edge of every node, i.e.,  $O(|E|) = O(|V|)$  times. Each iteration of the loop calls FIND-NUM-CHILDREN( $v, \ell, c$ ) which takes  $O(\ell) = O(k)$  time. Therefore each iteration of lines 6–15 costs  $O(k|V|)$ . Therefore the entire cost of MWCT is  $O(k^2|V|)$ . Finally, printing the output costs  $O(|V|)$ .

**Correctness.** The correctness follows by induction on  $k$  using a cut-and-paste argument. The loop invariant maintained by the innermost loop is that for all  $c' \leq c$ ,  $C[v, \ell, c']$  is equal to the cost of the MWCT of  $T[v, c']$ . The loop invariant of the outer loop is that for all  $v'$ , for all  $\ell' \leq \ell$ , for all  $c' \leq \deg(v)$ ,  $C[v, \ell', c']$  is equal to the cost of the MWCT of  $T[v', c']$  with  $\ell'$  nodes. These two invariants need to be proved together by induction. (We omit mention of the  $B$  array here, though its correctness follows by the same argument as follows.)

Consider immediately after the innermost loop has terminated for some  $v$ ,  $\ell$ , and  $c$ . We argue that  $C[v, \ell, c]$  is the cost of the MWCT of  $T[v, c]$ . Assume not. Then there is some other subtree  $S$  of  $T[v, c]$  with  $\ell$  nodes and cost  $< C[v, \ell, c]$ . Consider the subtree of  $S$  rooted at child  $c$  of  $v$  with  $i$  nodes. ( $S$  may be the empty tree.) We know that the cost of this subtree must be at least  $x = C[\text{child}(v, c), i, \deg(\text{child}(v, c))]$ , since by induction  $x$  is the MWCT of child  $c$  of  $v$  with  $i$  nodes. Similarly, consider the subtree of  $S$  consisting of all the nodes in  $T[v, c - 1]$ . Again, the cost of this subtree must be at least  $y = C[v, \ell - i, c - 1]$ , since by induction  $y$  is the MWCT of  $T[v, c - 1]$  with  $\ell - i$  nodes. But FIND-MIN-CHILDREN ensures that the  $C[v, \ell, c]$  is no more than  $x + y + w(v, c)$ , contradicting our assumption. Finally, the inductive step of the outer loop invariant follows from the inner loop invariant, concluding the proof.

### Problem 6. A tomato a day

Professor Kerry loves tomatoes! The professor eats one tomato every day, because she is obsessed with the health benefits of the potent antioxidant lycopene and because she just happens to like them very much, thank you. The price of tomatoes rises and falls during the year, and when the price of tomatoes is low, the professor would naturally like to buy as many tomatoes as she can. Because tomatoes have a shelf-life of only  $d$  days, however, she must eat a tomato bought on day  $i$  on some day  $j$  in the range  $i \leq j < i + d$ , or else the tomato will spoil and be wasted. Thus, although the professor can buy as many tomatoes as she wants on any given day, because she consumes only one tomato per day, she must be circumspect about purchasing too many, even if the price is low.

The professor's obsession has led her to worry about whether she is spending too much money on tomatoes. She has obtained historical pricing data for  $n$  days, and she knows how much she actually spent on those days. The historical data consists of an array  $C[1..n]$ , where  $C[i]$  is the price of a tomato on day  $i$ . She would like to analyze the historical data to determine what is the minimum amount she could possibly have spent in order to satisfy her tomato-a-day habit, and then she will compare that value to what she actually spent.

Give an efficient algorithm to determine the optimal offline (20/20 hindsight) purchasing strategy on the historical data. Given  $d$ ,  $n$ , and  $C[1..n]$ , your algorithm should output  $B[1..n]$ , where  $B[i]$  is the number of tomatoes to buy on day  $i$ .

#### Solution:

You can solve this problem in  $\Theta(n)$  time, which is the fastest possible asymptotic solution because it requires  $\Omega(n)$  time to scan the cost array. For every day  $i$ , you must buy the tomato for that day in the window  $W(i) \equiv [i - d + 1, i] \cap [1, n]$ . Note that the greedy choice works: in the optimal solution, you should buy the tomato for day  $i$  on the minimum cost day  $T_i$  in  $W(i)$ . Once you calculate  $T_i$  for all  $i$ , you can easily compute  $B$  in  $\Theta(n)$  time.

You can compute all  $T_i$  in  $O(n)$  time by computing the sliding window minimum  $T_i$  for each  $i$  in amortized  $O(1)$  time. Use a double-ended queue (deque) of days  $Q = \langle q_1, q_2, \dots, q_m \rangle$  such that day  $q_1$  is the minimum cost day in  $W(i)$  and each entry  $q_{k+1}$  is the minimum cost day of the days in  $W(i)$  following  $q_{k-1}$ . Then  $q_1 = T_i$  and this invariant can be maintained in amortized  $O(1)$  time using the following algorithm:

```

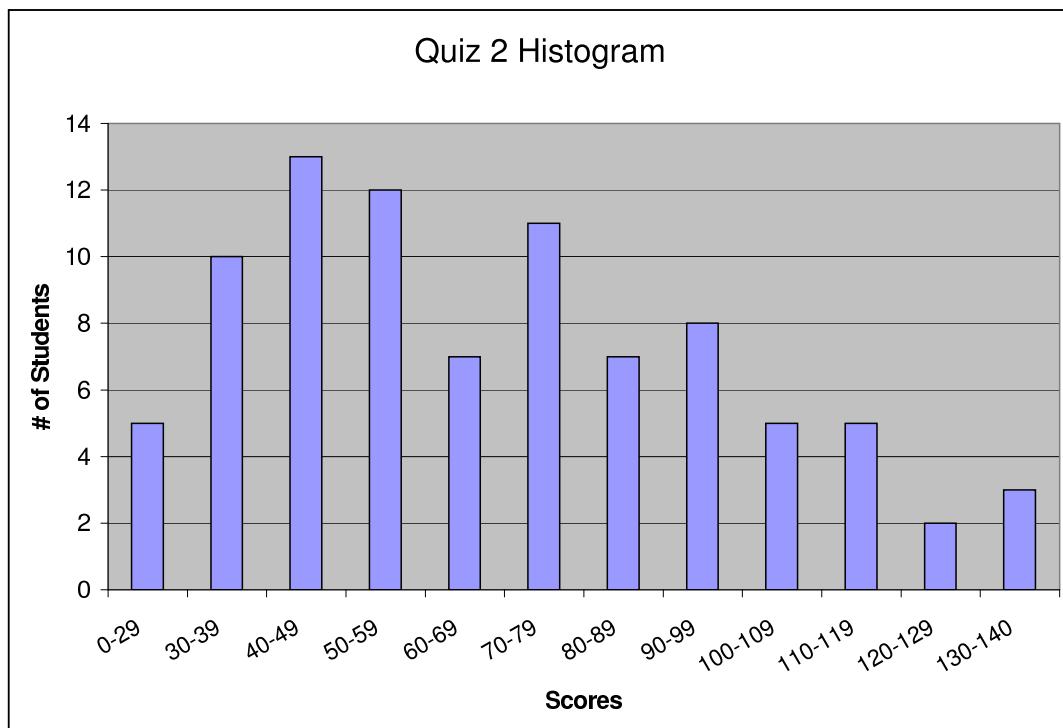
BUYTOMATOES( $d, C[1..n]$ )
1  $Q \leftarrow \emptyset$ 
2  $B[1..n] \leftarrow 0$ 
3 for  $i \leftarrow 1$  to  $n$ 
4   do if not EMPTY( $Q$ ) and HEAD( $Q$ )  $\leq i - d$ 
5     then POP-FRONT( $Q$ )       $\triangleright$  Remove old day
6     while not EMPTY and  $C[i] \leq C[\text{TAIL}(Q)]$ 
7       do POP-BACK( $Q$ )       $\triangleright$  Remove non-minimum-cost days
8       PUSH-BACK( $Q, i$ )         $\triangleright$  Add current day
9        $B[\text{HEAD}(Q)] \leftarrow B[\text{HEAD}(Q)] + 1$ 
10  return  $B$ 

```

Each day  $i$  is added once to  $Q$  and removed at most once from  $Q$ , so the total number of deque operations is  $O(n)$ . Each deque operation can be done in  $O(1)$  time. The loop in the algorithm executes for  $n$  iterations, so the overall running time is  $\Theta(n)$ . The deque requires  $O(\min(n, d))$  space since there can be at most one element for each of the past  $d$  days. The return array  $B$  requires  $\Theta(n)$  space, so the overall space used is  $\Theta(n)$ .

There are other ways to compute  $T_i$  in amortized  $O(1)$  time that also received full credit. Slower solutions that received partial credit include solutions that compute  $T_i$  in  $O(\log d)$  or  $O(d)$  time, solutions that compute  $B$  directly by looking ahead up to  $d$  days at each step, dynamic programming solutions, and solutions that sort the tomato prices and maximize the number of tomatoes purchased at lower prices.

## Quiz 2 Solutions



### Problem 1. Big Bully

The year is 2048. The world has been taken over by unruly orcs. Each orc is either a *captain* or a *soldier*. Roughly, an orc X turns into a captain if any orc bigger than X in X's neighborhood is already the solider of another captain, and X is the biggest orc remaining in X's neighborhood. On the other hand, X ends up being a soldier if there is an orc in X's neighborhood that is bigger than X and has turned into a captain. Your goal is to find out which orcs are captains and which orcs are soldiers in a one-dimensional cave. More precisely:

**Given:** An array  $A[1..n]$  of distinct integers (defining “orc size”), and an integer  $k$  (defining “neighborhood size”).

**Compute:**  $B[1..n]$ , where each  $B[i] \in \{\text{captain}, \text{soldier}\}$ , such that, for every  $i$ ,

- $B[i] = \text{captain}$  if, for every  $j \in \{i - k, \dots, i + k\}$ , either  $B[j] = \text{soldier}$  or  $A[j] \leq A[i]$ .
- $B[i] = \text{soldier}$  if there exists a  $j \in \{i - k, \dots, i + k\}$  such that  $B[j] = \text{captain}$  and  $A[j] > A[i]$ .

(For this definition, assume that  $A[i] = -\infty$  and  $B[i] = \text{soldier}$  for  $i < 1$  and  $i > n$ .)

Give an efficient algorithm for this problem. An ideal solution works well for all values of  $k$  relative to  $n$ . For partial credit, solve the special cases when  $k = O(1)$  and/or when  $k = \Omega(n)$ .

**Solution:** We give an  $O(n)$  time algorithm for this problem. Initially every orc is a captain. The algorithm makes two passes over the array  $A$ . The first pass is left to right and the second pass is right to left. Informally, on the first pass we demote orcs that are dominated by their left neighbors and on the second pass we demote orcs that are dominated by their right neighbors.

We use a single auxiliary variable  $p$  to keep a pointer to the orc that is *currently dominating*. Below is the pseudocode for our algorithm:

```

1  for  $i \leftarrow 1$  to  $n$ 
2     $B[i] \leftarrow \text{captain}$ 
3   $p \leftarrow 1$ 
4  for  $i \leftarrow 1$  to  $n$ 
5    if  $B[p] = \text{captain}$  and  $i \leq p + k$  and  $A[i] < A[p]$ 
6      then  $B[i] \leftarrow \text{soldier}$ 
7      else  $p \leftarrow i$ 
8   $p \leftarrow n$ 
9  for  $i \leftarrow n$  to 1
10   if  $B[p] = \text{captain}$  and  $i \geq p - k$  and ( $A[i] < A[p]$  or  $B[i] = \text{soldier}$ )
11     then  $B[i] \leftarrow \text{soldier}$ 
12     else  $p \leftarrow i$ 
```

Let us argue that the output of our algorithm satisfies both constraints specified in the problem statement.

- It is easy to see that the output of our algorithm can not contain two captains neighboring each other, since in the pass where we first observe the bigger captain we would demote the smaller.
- It remains to argue that every soldier neighbors a bigger captain. Note that at the moment that an orc is demoted it is demoted by a bigger captain from its neighborhood. Let  $i$  be a soldier. Assume that orc  $i$  has been demoted by a bigger orc  $j$ . The only problem is that orc  $j$  could have later been demoted by some other orc  $k$ . Note that this may only happen if  $i$  was demoted by  $j$  on the first pass, and  $j$  was demoted by  $k$  on the second pass. Thus orc  $k$  is a captain in the neighborhood of  $i$ , and clearly we have  $A[i] < A[j] < A[k]$ .

There were many different solutions to this problem. The grade was assigned based on the running time (and correctness) of the algorithm.

Several solutions arise from the observation that the largest orc  $i^*$  in the array must be a captain, with all other orcs in the neighborhood of  $i^*$  as soldiers.

This observation suggests the following algorithm. First, start with all orcs unassigned, i.e.,  $B[i] = \text{NULL}$ . Then, repeat the following loop until all orcs are assigned:

1. Scan all unassigned orcs in  $A$ , i.e., those with  $B[i] = \text{NULL}$ , and of those orcs find the orc  $i^*$  with maximum size  $A[i^*]$ . If all orcs have been assigned, then terminate the loop.
2. Mark  $i^*$  as a captain and mark any unassigned orcs  $j \neq i^*$  in the interval  $i^* - k \leq j \leq i^* + k$  as soldiers.

Finding the maximum in the first step takes  $O(n)$ . One can show that we can have at most  $O(n/k)$  captains, giving us a runtime of  $O(n^2/k)$ . In the special case when  $k = \Omega(n)$ , this simple algorithm gives a linear-time solution.

For  $k = o(n/\lg n)$ , the above algorithm can be improved by using some technique (i.e., sorting, building a heap, or a balanced search tree structure) to find the next maximum size unassigned orc more efficiently. If we store the orcs into a max-heap, then we can repeatedly extract the orcs in order of decreasing size. If we extract an unassigned orc  $i$ , then mark  $i$  as a captain and all unassigned orcs in  $i$ 's neighborhood as soldiers. If we extract an assigned orc, then do nothing.

For each of the  $O(n/k)$  orcs we mark as captains, we perform  $O(k)$  work to check its neighborhood. For any element  $i$  which is already a soldier when we extract it, we perform only  $O(1)$  work in labeling. Thus, the cost of only labeling captains and orcs is only  $O(n)$  time. Therefore, the runtime is dominated by the  $O(n \lg n)$  cost to extract all elements from the heap in order of decreasing size.

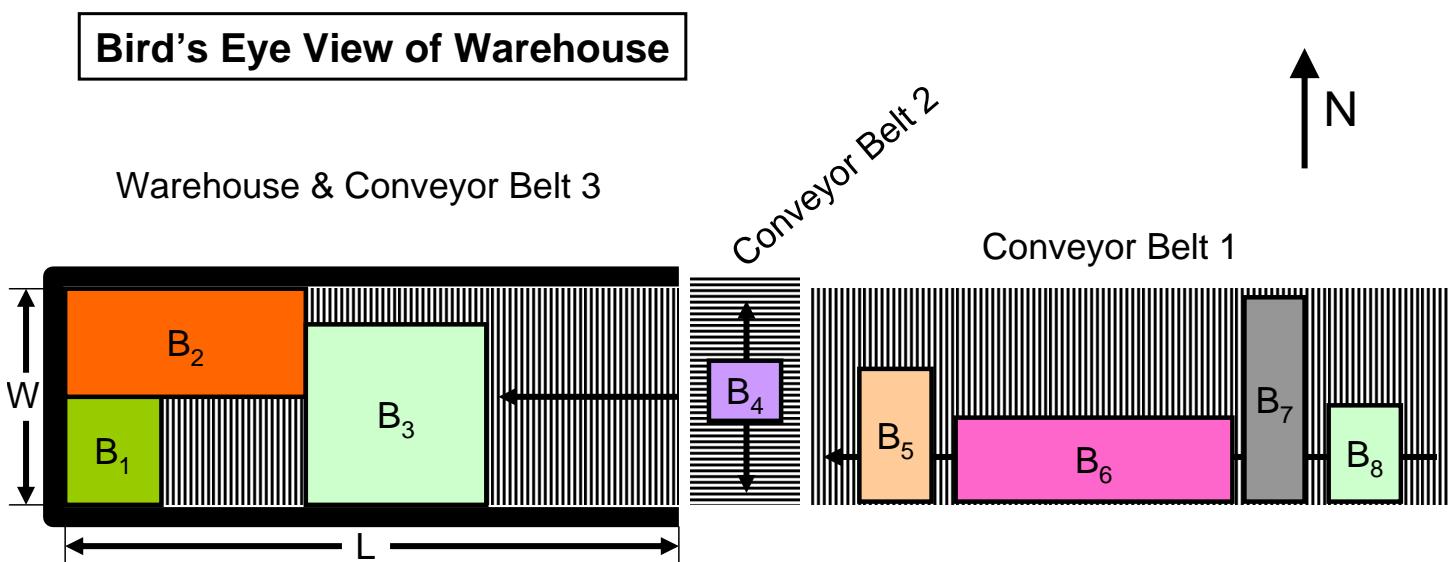
Finally, if we make an additional assumption that the integer orc sizes are each at most  $O(\lg n)$  bits, then we can use radix sort instead of a heap or other comparison-based sort and achieve an  $O(n)$  algorithm.

### Problem 2. The Gorehouse

Frustrated by his previous job experiences, Al Gorism has decided to make a career out of his best skill, developed by playing years of Tetris. He has taken up a job in a warehouse where his daily duty is to pack long rectangular boxes into an  $L \times W$  warehouse, where  $L$  is large, and  $W$  is a small constant (think  $W = 5$ ). Your goal is to design an algorithm to help him pack the boxes.

At the beginning of the day, Al starts with an empty  $L \times W$  warehouse and a sequence of  $n$  boxes lying on Conveyor Belt 1. He measures the boxes, and finds that the  $i$ th box  $B_i$  has an integer length  $\ell_i$  and an integer width  $w_i$ . Having measured all the boxes, his goal for the day is to pack them into the warehouse using three conveyor belts. Conveyor Belt 1 initially contains all boxes; it moves the boxes from east to west and deposits the boxes one at a time, in the order  $B_1, B_2, \dots, B_n$ , onto Conveyor Belt 2. Conveyor Belt 2 can move the boxes north/south. Initially, the southern border of all the boxes are aligned with the southern wall of the warehouse. Thus, when box  $B_i$  reaches Conveyor Belt 2, Al gets to choose how much north this box should be moved before it is transferred to Conveyor Belt 3. Conveyor Belt 3 then moves box  $B_i$  west as far as possible (i.e., until its motion is obstructed by some box in the warehouse or by the western wall of the warehouse).

Give an efficient algorithm to compute how many units  $u_i$  northward to move each box  $B_i$ , for  $i \in \{1, 2, \dots, n\}$ , when it arrives on Conveyor Belt 2, so that all the boxes can be fit into the  $L \times W$  warehouse. If there is no way to pack the boxes into the warehouse, then your algorithm should simply report this fact.



#### Solution:

**Executive Summary:** The solution uses dynamic programming to give an  $O(n \cdot L^W)$  algorithm for this problem.

**Details:** We think of the warehouse as being tiled by  $L \times W$  unit squares. The basic idea is to determine recursively if the first  $i$  boxes can be fit into the profile consisting of the  $L_1$  westmost tiles of the southmost row, the  $L_2$  westmost tiles of the next row and so on up to  $L_W$  westmost tiles of the top row. Let  $P(L_1, \dots, L_W, i)$  denote the indicator variable which is 1 if the first  $i$  tiles can be packed into the profile  $L_1, \dots, L_W$ , and 0 otherwise.

We note that  $P(L_1, \dots, L_W, i)$  satisfies the following conditions:

- If any of the  $L_j$ 's are negative, then  $P(L_1, \dots, L_W, i) = 0$ .
- If all the  $L_j$ 's are non-negative and  $i = 0$ , then  $P(L_1, \dots, L_W, i) = 1$ .
- For  $i > 0$ ,  $P(L_1, \dots, L_W, i) = 1$  if and only if there exists some  $u_i \in \{0, \dots, W - w_i\}$  such that  $P(L'_1, \dots, L'_W, i - 1) = 1$  where  $L'_j = L_j$  if  $j \notin \{u_i + 1 \dots u_i + w_i\}$  and  $L'_j = \min_{k \in \{u_i + 1 \dots u_i + w_i\}} \{L_k - \ell_i\}$  otherwise. (In English,  $L'_1, \dots, L'_W$  give the easternmost profile that would allow the  $i$ th box to be moved  $u_i$  units northwards and still fit within the profile  $L_1, \dots, L_W$ .

We can now compute the  $P(\dots)$  for all input arguments starting with  $i = 1$  to  $n$ . In each iteration we enumerate the  $L_1, \dots, L_W$  in any order and compute  $P(L_1, \dots, L_W, i)$  using the recurrence above. Each  $P(L_1, \dots, L_W, i)$  takes  $O(W^2)$  time to compute. There are  $n \cdot L^W$  such values, thus taking a total of  $O(W^2 \cdot n \cdot L^W)$  time to compute them all. We finally return  $P(L, L, \dots, L, n)$ .

**Notes on solutions:** Alternately, one can also describe the solution in terms of the natural recursive algorithm with memoization based on the profile.

The problem also has an exponential,  $O(W^n)$ , time algorithm. This solution received partial credit (it is slow but correct).

A fair number of solutions proposed a greedy algorithm minimizing the maximum of the  $L_j$ 's after inserting box  $i$ , for every  $i$ . This solution is obviously incorrect and received no credit.

### Problem 3. Budget Shopping

You have just inherited a large toy store, Holidays ‘Я’ Us. Throughout the day, customers come to you for advice of what to buy their relatives, given a hard budget of how much they can spend. To maximize your revenue, you always recommend the most expensive item that they can afford. The store has  $n$  items for sale, of distinct positive prices  $p_1 < p_2 < \dots < p_n$ , and can sell arbitrarily many copies of each item. Over the holiday season, your  $m$  customers come in arbitrary order with positive budgets  $b_1, b_2, \dots, b_m$ , respectively. You convince each customer  $i$  to purchase the predecessor product, i.e., the most expensive product  $p_j < b_i$ .

Although you cannot predict the budgets of customers, you observe that products tend to be sold in “waves”. To model this feature, when you sell product  $p_j$  to customer  $i$  (the  $i$ th sale), you define the *recentness*  $r_i$  of this sale to be the number of customers since that product  $p_j$  was last purchased, i.e.,  $r_i = i - i'$  where  $i'$  is the largest index  $< i$  such that customer  $i'$  purchased product  $p_j$ . As a special case, if customer  $i$  is the first to have purchased the product  $p_j$ , then the recentness of the  $i$ th sale is defined to be  $n$ . You observe that, in practice, the recentness of most purchases is much smaller than  $n$ .

To speed your customer advice, your goal is to build a data structure storing the set of prices that supports predecessor queries as fast as possible. The  $i$ th query  $b_i$  should run faster when the recentness  $r_i$  is small.

**Solution:** The optimal data structure for this problem supports the  $i$ th query  $b_i$  in  $O(\lg r_i)$  time. This bound is not easy to achieve—only one student did so—but there are many simpler structures whose performance adapts to smaller values of  $r_i$ . We will describe some of the most common such partial solutions, and try to show the thought process for successive improvements.

The first solution that should come to your mind for solving the predecessor problem is a bound of  $O(\lg n)$ , either by binary search in a sorted array or by searching in a balanced binary search tree. This bound is already a good start. Most students got at least this far.

A simple idea for adapting to  $r_i$  is to obtain a bound of  $\Theta(r_i)$ . The easiest way to achieve this bound is to maintain the products in a linked list, ordered by the time they were last accessed, with most recent items at the front. (Initially the list can be ordered arbitrarily.) Each node of the list needs to store both the price  $p_j$  of the item and the price  $p_{j+1}$  of the next item, so that the query can tell directly whether this product will be purchased by customer  $b_i$  (when  $p_j < b_i \leq p_{j+1}$ ). A query walks through the linked list, and when it finds the matching product, moves it to the front of the list, thereby keeping the desired list ordering. Many students found this solution.

Of course,  $r_i$  could be larger than  $\lg n$ , so a query should really only look at the first  $O(\lg n)$  products of the list, after which it should fall back to the sorted array or binary search tree. This leads to a bound of  $O(\min\{r_i, \lg n\})$ . To achieve this bound, we add pointers between corresponding nodes in the linked list and in the sorted array or binary search tree. Then a query walks through the first  $\lg n$  nodes of the list; if it doesn’t find the answer there, it does a binary search or tree walk to find the predecessor, and then follows the pointer to the corresponding node in the list. In either case, the query then moves the list node to the front. Many students found this solution.

But if we are using only the first  $\lg n$  products of the list, surely we can store them in a more efficient way? Specifically, if we store the most recent  $\lg n$  products in a balanced binary search tree (e.g., red-black tree), then a query costs  $O(\lg \lg n)$  time if  $r_i \leq \lg n$  and  $O(\lg n)$  time otherwise. This is a big improvement if  $r_i$  is between  $\lg \lg n$  and  $\lg n$ . To actually maintain this “cache” of  $\lg n$  products, we also need to store a linked list of the most recent  $\lg n$  products, ordered by when they were last accessed, and store pointers between corresponding nodes in the linked list and in the binary search tree. When a query finds the answer in the cache tree, we can move the corresponding list node to the front of the list. When a query has to go to the main tree of all products, we kick out the oldest item from the cache (the last product in the list), and replace it with the newly queried item (putting it at the front of the list). Several students found this solution.

If we push this caching idea a little further, we can get a bound of  $O((\lg r_i)^2)$ . The idea is to store caches of exponentially increasing size: 1, 2, 4, 8, ...,  $2^{\lceil \lg n \rceil}$ . The first cache stores the most recent product; the next cache stores the next two most recent products; etc. (The last cache might not be full.) Each product is in exactly one cache. Each cache consists of a balanced binary search tree, keyed by price, and a linked list, ordered most-recent-product-first. A query searches for the budget in the first cache, then the second cache, etc. If the product has recentness  $r_i$ , then we will find it in the cache of size  $2^k$  for some  $k \leq \lceil \lg r_i \rceil$ . Then we move this element to the size-1 cache, kick out the oldest element from that cache and put it into the size-2 cache, kick out the oldest element from that cache and put it into the size-4 cache, etc., until the size- $2^k$  cache which has room for one element. Thus we perform one search, one delete, and one insert in each of the caches from 1 up to  $2^k$ . The total cost is thus  $O(\lg 1 + \lg 2 + \lg 4 + \dots + \lg 2^k) = O(\sum_{i=0}^k \lg 2^i) = O(\sum_{i=0}^k i) = O(k^2) = O((\lg r_i)^2)$ . One student found this solution.

Now, finally, we describe an optimal structure achieving a bound of  $O(\lg r_i)$ . The trouble with exponentially increasing caches is that the running time ends up being an arithmetic sum. To make it a geometric sum, we build caches of doubly exponentially increasing size: 2, 4, 16, 256, ...,  $2^{2^{\lceil \lg \lg n \rceil}}$ . The first cache stores the  $2^{2^0}$  most recently purchased products; the second cache stores the next  $2^{2^1}$  most recently purchased products; etc. (The last cache might not be full.) We represent the size- $2^{2^i}$  cache in two different ways: a balanced search tree (e.g., a red-black tree), keyed on price, and a linked list, ordered most-recent-product-first. Every product appears in exactly one cache, in both the tree and the list forms. We store pointers between corresponding nodes in the tree and list forms. A query searches for the budget in the first cache, then the second cache, etc. If the product has recentness  $r_i$ , then we will find it in the cache of size  $2^{2^k}$  for some  $k \leq \lceil \lg \lg r_i \rceil$ . Then we move this element to the size-2 cache, kick out the oldest element from that cache and put it into the size-4 cache, kick out the oldest element from that cache and put it into the size-16 cache, etc., until the size- $2^{2^k}$  cache which has room for one element. Thus we perform one search, one delete, and one insert in each of the caches from 2 up to  $2^{2^k}$ . The total cost is thus  $O(\lg 2 + \lg 4 + \lg 16 + \dots + \lg 2^{2^k}) = O(\sum_{i=0}^k \lg 2^{2^i}) = O(\sum_{i=0}^k 2^i) = O(2^k) = O(2^{\lg \lg r_i}) = O(\lg r_i)$ . One student found this solution.

In case you are interested, the  $O(\lg r_i)$  bound is called the *working-set property* in the literature, indicating that working with a small subset of elements is faster; it plays an important role in the study of optimal search trees.

### Problem 4. Drowsy Shortest Paths

On the morning of November 27, 2006, you decide to attend the mandatory Lecture 21. Your task is complicated by the fact that lecture is early in the morning and you are still sleepy when trying to figure out the shortest path to get there. Unlike shortest paths when you are fully awake, this *drowsy shortest paths* problem is somewhat different. As before, every edge takes some fixed time to traverse. But when you are sleepy, every time you reach a vertex it takes some time to figure out which outgoing edge to take. Furthermore, the time it takes to resolve the confusion depends on how awake you are. So it might take less time to get to class if you start later (because you are more awake), but you might end up missing the beginning of lecture if you start too late! Your goal is to determine how late you can wake up and still get to lecture in time.

The map of MIT is described by a directed graph  $G = (V, E)$ . Your starting point is a vertex  $s$  and your goal is to reach the 6.046 lecture at vertex  $z$ . If you wake up  $t_0$  units of time before start of lecture, then the time to visit a vertex  $v$  is  $\alpha \cdot t_0$  minutes, where  $\alpha < 1$  is a global nonnegative constant. (So the time to visit a vertex  $v$  is independent of the vertex  $v$ , but it depends on how much sleep you get.) Each edge  $e$  takes some nonnegative real number  $w(e)$  minutes to traverse, independent of when you wake up. Give an efficient algorithm to compute the least amount of time  $t_0$  before lecture that you could wake up and still reach lecture in time.

#### Solution:

**Summary** The best solution we know for this problem uses a modification of the Bellman-Ford algorithm and computes the optimal value of  $t_0$  in  $O(E \cdot \min(V, \frac{1}{\alpha}))$  time. Several other solutions with slower run-times are possible. We describe the Bellman-Ford solution in some detail and briefly discuss some of the other common solutions.

**Main idea** The first thing to note is that for any path  $p$  from  $s$  to  $z$ , the drowsy time  $t_0$  to traverse the path satisfies

$$t_0 = m \cdot \alpha \cdot t_0 + \sum_{e \in p} w(e),$$

where  $\sum_{e \in p} w(e)$  is the regular (non-drowsy) time to traverse  $p$ , and  $m$  is the number of edges in  $p$  (since  $\alpha \cdot t_0$  time is added at each vertex other than  $z$ ). This gives

$$t_0 = \frac{\sum_{e \in p} w(e)}{1 - m\alpha}. \quad (1)$$

The crucial thing to observe now is that if we consider all the paths from  $s$  to  $z$  with a fixed number of edges  $m$ , then the best (smallest) value of  $t_0$  among these  $m$ -edge paths is achieved with a path that has the shortest regular (non-drowsy) time among them. Thus if we let  $\delta^{(m)}(s, z)$  denote the shortest regular time among paths with exactly  $m$  edges, then the smallest drowsy time that can be achieved with a path of  $m$  edges is given by

$$t_0^{(m)} = \frac{\delta^{(m)}(s, z)}{1 - m\alpha}.$$

The best value of  $t_0$  overall is then the smallest among all  $t_0^{(m)}$  (note that we only need to consider values of  $m$  smaller than  $\frac{1}{\alpha}$ , since by Eq. (1),  $m > \frac{1}{\alpha}$  would imply it is not possible to get to lecture in time with any path of  $m$  edges):

$$t_0^* = \min_{1 \leq m \leq \min(|V|-1, \frac{1}{\alpha})} \frac{\delta^{(m)}(s, z)}{1 - m\alpha}.$$

Thus all we need is an algorithm that can compute  $\delta^{(m)}(s, z)$ , the shortest regular time that can be achieved with a path of  $m$  edges, for each  $m$ ; we can then compute the optimal  $t_0^*$  as above.

**Bellman-Ford based algorithm** We can make a simple modification to the Bellman-Ford algorithm to compute the values  $\delta^{(m)}(s, z)$ , and keep track of the smallest  $t_0^{(m)}$  as we vary  $m$  from 1 to  $\min(|V| - 1, \frac{1}{\alpha})$ . The modification ensures that in the  $m$ th outer loop we separate the newly computed  $d$  values from the ones computed in the previous iteration.

```
DROWSY-SHORTEST-PATHS( $G, s, z$ )
1   for each vertex  $v \in V$ 
2     do  $d[v] \leftarrow \infty$ 
3      $d[s] \leftarrow 0$ 
4      $t_0 \leftarrow d[z]$ 
5   for  $m \leftarrow 1$  to  $\min(|V| - 1, \frac{1}{\alpha})$ 
6     do for each vertex  $v \in V$ 
7       do  $d_{\text{old}}[v] \leftarrow d[v]$ 
8        $d[v] \leftarrow \infty$ 
9     for each edge  $(u, v) \in E$ 
10    do if  $d[v] > d_{\text{old}}[u] + w(u, v)$ 
11      then  $d[v] \leftarrow d_{\text{old}}[u] + w(u, v)$ 
12    if  $t_0 > \frac{d[z]}{1 - \alpha \cdot m}$ 
13      then  $t_0 \leftarrow \frac{d[z]}{1 - \alpha \cdot m}$ 
14  return  $t_0$ 
```

**Correctness** From the analysis of Bellman-Ford it follows that for all  $m, v$ , at the end of the  $m$ th iteration, the quantity  $d[v]$  contains the shortest regular time that can be achieved with a path from  $s$  to  $v$  using exactly  $m$  edges. Therefore, at the end of the  $m$ th iteration, the quantity  $d[z]$  is equal to  $\delta^{(m)}(s, z)$  as discussed above. Correctness thus follows from the above discussion.

**Run-time analysis** The running time of the above algorithm is  $O(E \cdot \min(V, \frac{1}{\alpha}))$ ; the space it takes is  $O(V)$ .

**Other algorithms** Several students came up with the above Bellman-Ford based algorithm. Some other slower algorithms that were also found among the solutions are as follows:

### Repeated runs of Dijkstra

There are at least two (correct) solutions that use Dijkstra; both involve  $O(\min(V, \frac{1}{\alpha}))$  runs of Dijkstra and therefore take  $O((V \lg V + E) \cdot \min(V, \frac{1}{\alpha}))$  time. One solution simply uses one run of Dijkstra for each  $m$  to compute  $\delta^{(m)}(s, z)$ ; this can be done by keeping track of the lengths of paths found from  $s$  and not relaxing any edges going out of a vertex to which a path of length  $m$  has been found. The other solution starts by running Dijkstra using the original weights  $w(e)$  to find the shortest (regular-time) path overall, and computing the drowsy time  $t_0$  corresponding to this path (using Eq. (1)); the next run of Dijkstra then uses the new weights  $w(e) + \alpha \cdot t_0$ , finds a corresponding shortest path, and updates  $t_0$  based on the new path; this process is repeated until no improvement is obtained (it can be shown that the number of edges in the path found goes down on each run of Dijkstra, and therefore in this case too, at most  $O(\min(V, \frac{1}{\alpha}))$  runs of Dijkstra are required).

### Dynamic programming

The dynamic programming method used to find all-pairs shortest paths (CLRS, Section 25.1) can be used to give another (albeit slower) algorithm for this problem. The matrices  $L^{(m)}$  computed in that method contain the weights of shortest paths of at most  $m$  edges. It is easy to show that if  $\tilde{\delta}^{(m)}(s, z)$  denotes the weight of a shortest (regular-time) path from  $s$  to  $z$  among paths with *at most*  $m$  edges, we can still compute  $t_0^*$  as

$$t_0^* = \min_{1 \leq m \leq \min(|V|-1, \frac{1}{\alpha})} \frac{\tilde{\delta}^{(m)}(s, z)}{1 - m\alpha}.$$

Since we can obtain the required  $\tilde{\delta}^{(m)}(s, z)$  by computing the matrices  $L^{(m)}$  for each  $m$  ranging from 1 to  $\min(|V| - 1, \frac{1}{\alpha})$ , from the results of Section 25.1 in CLRS, this gives us an  $O(V^3 \cdot \min(V, \frac{1}{\alpha}))$  time algorithm for finding  $t_0^*$ . In fact, this algorithm is redundant in the sense that we really only need to compute one row of each of these matrices, namely the row corresponding to the source vertex  $s$ . This observation gives a quick improvement with this approach, leading to an  $O(V^2 \cdot \min(V, \frac{1}{\alpha}))$  time algorithm.

### Brute force

Exhaustively evaluate all paths from  $s$  to  $z$ . Exponential time, but correct.

### Search for $t_0^*$ using doubling trick with Dijkstra

A completely different approach can be used if one assumes the weights  $w(e)$  and  $t_0$  are all integers. In this case, one can search for the optimal value  $t_0^*$  using a doubling trick as follows. Start with  $t_0 = 1$ ; run Dijkstra with weights  $w(e) + \alpha \cdot t_0$ . If the shortest path returned has (drowsy) traversal time smaller than  $t_0$ , double  $t_0$  and repeat. If the shortest path returned has (drowsy) traversal time greater than  $t_0$ , use binary search between  $t_0/2$  and  $t_0$  to find the right value. If the traversal time is equal to  $t_0$ , return that value of  $t_0$ . This algorithm takes  $O((\lg t_0^*)(V \lg V + E))$  time. The integer assumption for this algorithm is a strong assumption (although it can be shown to also give some useful approximation in the real case).

## Quiz 2 Solutions

**Problem 1. Hacking Skip Lists** After years of painstaking effort, you launch myferretplanetspace.com, an e-commerce site devoted to selling ferrets online. Your list of active customer orders is implemented as a skip list, with all data about an order stored as an element in the skip list. Every time a customer asks about the status of their order, your site queries the customer order list and displays the data. Every order gets a unique ID number that is used to place it in the customer order list. ID numbers are assigned by your web site and are not necessarily increasing, so the most recent order could go anywhere in the skip list. For a while, your site works well; orders can be managed quickly and customers are happy.

But the ferret e-commerce industry is a cutthroat one, and one dark and stormy night your competitors try to slow down your site. There are two operations they can do: add a dummy order to your site, and cancel an order that they added.

- (a) Your system is storing information about  $n$  legitimate orders. Your competitors add  $m$  dummy orders to your system. What effect will this have on the runtime of query, insert, and delete operations for legitimate customer orders the morning after your competitors try to slow your site? Assume that  $m \gg n$ .

**Solution:**

INSERT, DELETE, and SEARCH operations on a skip list normally take  $\Theta(n)$  time, when a skip list has  $n$  elements. Here the number of elements (customer orders) changes from  $n$  to  $n + m$ , so all three operations take  $\Theta(m + n) = \Theta(m)$  time. This could slow the site substantially. For instance, if  $m = 2^n$ , performance will change from  $\Theta(\log n)$  to  $\Theta(n)$ .

- (b) Your competitors find a security loophole in your site that lets them see information about your skip list: specifically, which level each order has been promoted to. They again wish to perform  $m$  operations, with the goal of slowing down later legitimate customer orders as much as possible. What should they do? What effect will this have on the runtime of query, insert, and delete operations for legitimate customer orders the morning after?

**Solution:**

To slow down the site, the competitors should try to make the skip list more like a linked list. They can do this using the following procedure:

1. Insert a dummy order.
2. Delete that order if it was promoted above the lowest level in the skip list.
3. Repeat until all  $m$  operations have been used up.

## Performance

In the morning, INSERT, DELETE, and SEARCH operations will still take  $\log(n)$  time to traverse the levels of the skip list. At the bottom of the skip list, the list of elements that must be traversed will have an expected length of  $\Theta(m/n)$ .

So these operations will take  $\Theta(m/n + \log n)$ .

**Another Acceptable Answer** The problem didn't say that the competitors know the size of  $n$ , the number of orders in the system when they start their attack.

But if they did, they would know the number of levels in the skip list would be about  $\log(n)$ . They could use this to

1. Insert a dummy order.
2. Delete that order if it was promoted above the lowest level in the skip list, unless it is promoted to a level higher than  $\log(n)$  and is the first dummy element promoted to that level.
3. Repeat until all  $m$  operations have been used up.

This strategy would slow performance to  $\Theta(m/n + \log m)$ . We gave full credit for this answer (for students who assumed competitors knew the size of  $n$ ) and full credit for the other answer described (for students who assumed competitors did not know the size of  $n$ ).

### Problem 2. Balanced linear combination

Suppose you are given two vectors  $u, v$  of length  $n$ , such that,  $u_i, v_i > 0$  for all  $i = 1, 2, \dots, n$ , and  $n$  is odd. A vector  $w$  is a *balanced linear combination* of  $u$  and  $v$  if it is a non-trivial linear combination of  $u$  and  $v$ , i.e.

$$w = \alpha u + \beta v \text{ for some } \alpha, \beta \text{ such that } \alpha \neq 0 \text{ and } \beta \neq 0$$

and the median of  $w$  is 0. For example, if  $n = 5$ ,  $u = (1, 3, 4, 2, 5)$  and  $v = (4, 4, 4, 4, 4)$ , then  $w = 4u - 3v = (-8, 0, 4, -4, 8)$  is a balanced linear combination of  $u$  and  $v$ .

Give an efficient algorithm to find a balanced linear combination of  $u$  and  $v$ .

**Solution:** Observe that for any balanced linear combination  $z$  of  $u$  and  $v$ ,

$$z = \alpha u + \beta v$$

There is a *normalized* (the coefficient of  $u$  is 1) balanced linear combination for  $u$  and  $v$

$$z_1 = u + (\beta/\alpha)v$$

since  $\alpha \neq 0$  and  $\beta \neq 0$ .

Therefore, we can narrow down our search to the set of normalized balanced linear combination, i.e. finding appropriate value of  $\beta$  such that the vector

$$x = u + \beta v$$

has median equal to zero. For each  $i = 1, 2, \dots, n$ , calculate the array  $s$  such that  $s_i = v_i/u_i$ . It is straightforward to verify that

- If  $\beta > s_i$ , then  $x_i$  is positive.
- If  $\beta < s_i$ , then  $x_i$  is negative.
- If  $\beta = s_i$ , then  $x_i$  is 0.

Therefore, if  $\beta$  is the median of  $s$ , then median of  $x$  is 0. The problem is equivalent to finding the median of the array  $s$ .

Our algorithm works as follows.

1. Calculate  $s_i$ , for all  $i = 1, 2, \dots, n$ .
2. Find median of  $s$  by deterministic SELECTION.
3. Output  $u - \beta v$ .

All three steps in the algorithm only takes  $O(n)$  time. Therefore, the total running time of this algorithm is  $O(n)$ .

### Problem 3. Graph Sparsification

You are given a connected graph  $G = (V, E)$  such that each edge  $e \in E$  has a distinct non-negative value  $w(e)$ . In order to reduce the cost of storing  $G$ , we want to remove edges from  $G$  such that the number of remaining edges is exactly equal to the number of nodes. However, there are two constraints on which edges can be removed and which edges can be kept. First, each vertex chooses an edge incident to it to keep. An edge will only be kept when it is chosen by some vertex. Any edge that is not chosen by any vertex will be removed. Second, the set of remaining edges must keep  $G$  connected, i.e. for any pair of vertices  $u$  and  $v$ , there is a path from  $u$  to  $v$  only using the chosen edges.

Give an efficient algorithm to find which edge should be chosen by which vertex such that the total value of the remaining edges is maximized.

#### Solution:

##### The set of edges to be chosen:

In our algorithm, we find the *maximum spanning tree* in  $G$  and keep all of its edges. Since this tree only contains  $n - 1$  edges, one more edge needs to be chosen. To find such edge, we look at all edges in  $G$  which are not in the tree and pick the one with maximum weight.

Find the maximum spanning tree: For each edge  $e \in E$ , let  $w_1(e) = C - w(e)$ , where  $C$  is the maximum edge weight in  $G$ . For any spanning tree  $S$  in  $G$ ,  $w_1(S) = (n - 1)C - w(S)$ <sup>1</sup>. Thus, the minimum spanning tree in  $G$  w.r.t weight function  $w_1$  is also the maximum spanning tree in  $G$  w.r.t weight function  $w$ . Prim algorithm can be used to find the maximum spanning tree in  $G$ .

##### The mapping

Let  $S_{max}$  be the maximum spanning tree and  $e_{max} = (u, v)$  be the last edge added to the set. Consider the tree  $S_{max}$  with  $u$  as root. Repeatedly do the following step until only one vertex left in the tree: Pick a leaf  $x$  in the tree, let  $f$  map  $x$  to the edge that connects  $x$  to its parent, and delete  $x$  (along with the edge) from the tree. After  $n - 1$  such steps, the only vertex left is  $u$ . We can complete the mapping  $f$  by letting it map  $u$  to  $e_{max}$ .

**Proof of correctness** We will prove that the mapping  $f$  satisfy all four constraints:

1.  $f(u) = f(v)$  iff  $u = v$ .
2.  $\sum_v w(f(v))$  is maximized.
3.  $f(v)$  is incident to  $v$ , for all vertices  $v$ .
4. the edge set  $\{f(v) | v \in V\}$  is connected.

The first and third constraints is straight forward from the construction of our mappings.

The fourth constraint is guaranteed by the fact that the set of chosen edges contains a spanning tree in  $G$ .

---

<sup>1</sup> $w(S)$  and  $w_1(S)$  are the total edge weight of  $S$  w.r.t weight function  $w$  and  $w_1$  respectively

Therefore, we only need to prove that  $\sum_v w(f(v))$  is maximized. We will prove that  $\sum_v w(f(v)) \geq \sum_v w(g(v))$  for any mapping  $g$  satisfying constraints 1,3 and 4. Let  $A_g$  be the set of edges in  $G$  chosen by  $g$ . Since  $A_g$  has exactly  $n$  edges, the (connected) subgraph of  $G$  formed by  $A_g$  has exactly one cycle. In that cycle, there must be at least one edge  $e_1$  that is not in  $S_{max}$ , thus,  $w(e_1) \leq w(e_{max})$ . Also, consider the set  $A_g - \{e_1\}$ , it has exactly  $n - 1$  edges and still forms a connected subgraph in  $G$ . Therefore, since  $S_{max}$  is the maximum spanning tree in  $G$ ,  $w(A_g - \{e_1\}) \leq w(S_{max})$ . Thus,  $w(A_g) \leq w(S_{max} \cup e_{max})$   $\square$ .

**Running time** The optimal running time is  $O(E + V \log V)$  by using Prim algorithm to find MST. If Kruskal algorithm is used, then the running time is  $O(E \log V)$ , which is not optimal.

### Problem 4. Detecting Compatible Subtrees

Given two rooted trees<sup>2</sup> we say that  $S$  is a *subtree* of  $T$  if there is a one-to-one mapping  $f$  from the vertices of  $S$  to the vertices of  $T$  that preserves all parent-child relationships.<sup>3</sup> In this problem, we consider rooted trees with the additional feature that every vertex is assigned a *label* from some set  $L$  (for example, the labels may come from the set of colors  $L = \{\text{Red, Orange, Yellow, Green, Blue, Purple}\}$ ). Moreover, we are told that only certain pairs of labels are “compatible.” The set of compatible labels can be represented as an undirected graph  $H$ , called a *compatibility graph*, with vertex set  $L$  and edges connecting labels that are compatible.

Given two rooted trees  $S$  and  $T$ , both with labels in  $L$ , we would like to determine whether  $S$  is a subtree of  $T$ , but with the additional constraint that each vertex of  $S$  must be matched with a compatibly labeled vertex of  $T$ . Formally, if there is a function  $f$  with all of the properties in the previous paragraph, and the additional property that the labels of  $v$  and  $f(v)$  are compatible for all  $v \in S$ , we say  $S$  is a *compatibly labeled subtree* of  $T$ . Figure 1 illustrates an example.

- (a) Suppose you are given two labeled rooted trees,  $S$  and  $T$ , along with a compatibility graph  $H$ . Let  $|S| = m$  and  $|T| = n$ , where  $m \leq n$ . **Further suppose that both  $S$  and  $T$  have depth 1.** Give an efficient algorithm to determine if  $S$  is a compatibly labeled subtree of  $T$ . You may assume that for vertices  $u \in S$  and  $v \in T$ , it only takes  $O(1)$  time to check if the labels of  $u$  and  $v$  are compatible.

**Solution:** This problem can be solved in  $O(m^2n)$  time by reducing it to bipartite matching. The algorithm is as follows.

First we check that the root of  $S$  is compatible with the root of  $T$ . As specified in the problem, this can be done in  $O(1)$  time. If the roots are compatible, then we need to check that there is a way to compatibly map the  $m - 1$  leaves of  $S$  to the  $n - 1$  leaves of  $T$ . This is exactly the maximum bipartite matching problem. To solve it we create a bipartite graph. The vertices  $L$  are the leaves of  $S$  and the vertices  $R$  are the leaves of  $T$ . For each  $u \in S$  and  $v \in T$  we check whether they are compatible, and if so, we put an edge between the corresponding nodes of the graph  $G = L \cup R$ . It is clear that the leaves of  $S$  can be compatibly matched to the leaves of  $T$  iff the maximum matching in  $G$  is of size  $m - 1$ .

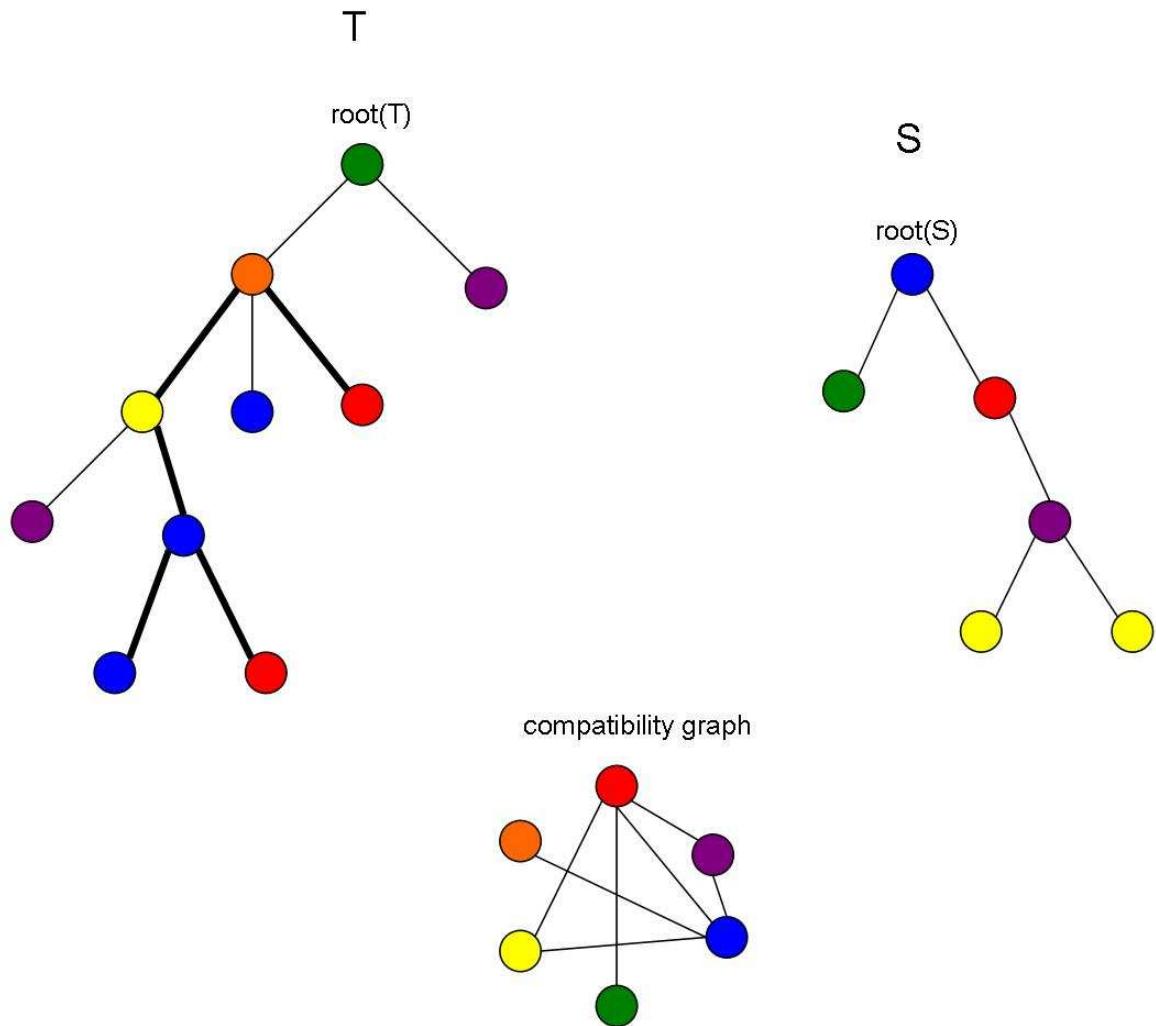
A solution to the maximum bipartite matching problem is given in the book, and was also covered in recitation. To solve the problem, we make all the edges of the bipartite

---

<sup>2</sup>For completeness, we give some formal definitions here: a *rooted tree* is simply a tree where one vertex is designated as the root. For any vertex  $v$ , the *children* of  $v$  are defined as the neighbors of  $v$  which are further from the root, and the *parent* of  $v$  is defined as the neighbor of  $v$  which is closer to the root (if  $v$  itself is not the root). Every vertex has exactly one parent, except the root which has no parents. Rooted trees need not be binary or balanced- any vertex may have an arbitrary number of children. There is no ordering among the children.

The *depth* of a rooted tree is the depth of the deepest vertex (the root has depth 0, the children of the root have depth 1, their children have depth 2, etc.).

<sup>3</sup>In other words,  $f$  must map every vertex of  $S$  to a distinct vertex of  $T$ , in such a way that if  $u$  is the parent (or child) of  $v$  in  $S$ , then  $f(u)$  is the parent (or child, respectively) of  $f(v)$  in  $T$ .



**Figure 1:** Two labeled rooted trees  $S$  and  $T$ . As illustrated by the dark lines,  $S$  is a compatibly labeled subtree of  $T$ .

graph directed from  $L$  to  $R$ . We add a source node  $s$  connected to all nodes in  $L$ , and a sink node  $t$  connected from all nodes in  $R$ . Then we give all edges capacity 1, and solve for the maximum flow on the resulting graph. See CLRS section 26.3 for details.

Now we analyze the algorithm. Constructing the graph  $G$  takes  $O(mn)$  time. The number of edges in  $G$  is bounded by  $O(mn)$ , since in the worst case each node of  $L$  is connected to every node of  $R$ . Solving for the maximum flow using any Ford-Fulkerson algorithm takes  $O(E|f^*|)$  where  $E$  is the number of edges in  $G$  and  $|f^*|$  is the size of the maximum flow. Here  $|f^*|$  is at most  $m - 1 = O(m)$ . So in total, Ford-Fulkerson takes  $O(m^2n)$  time.

Notes on alternative analysis and solutions:

- Some people noted that the Hopcraft-Karp algorithm referred to in the book takes only  $O(\sqrt{V}E)$  time, which in this case comes out to be  $O(mn^{1.5})$ . This answer also received full credit. However, this was not the preferred answer, since we didn't actually see the details of the algorithm or analysis in class, nor do the details appear in the book (they only appear as a problem in the book, they are never fully explained). Similarly, some people cited the  $O(n^\omega)$  bipartite matching algorithm (where  $\omega$  is the best known matrix multiplication exponent) by Mucha and Sankowski. Again, this was unnecessary. This algorithm was mentioned casually in recitation, but we didn't even come close to talking about the details.
- Some people used a variation on bipartite matching to achieve a better runtime when the number of labels is small. These people also created a bipartite graph  $G = L \cup R$ , but the number of nodes in both  $L$  and  $R$  was equal to the number of possible labels. As before, they created a source node  $s$  connected to all nodes in  $L$ , and a sink node  $t$  connected from all nodes in  $R$ . However the capacity of the edge  $(s, i)$  was equal to the number of leaf nodes in  $S$  with label  $i$ , and the capacity of the edge  $(j, t)$  was equal to the number of leaf nodes in  $T$  with label  $j$ . There were also edges with infinite capacity going from  $L$  to  $R$  corresponding to compatible labels. The maximum flow was then found on this graph  $G$ , and if it was equal to  $m - 1$ , the algorithm accepted.

To analyze the running time of this algorithm, let  $k$  be the number of labels. There are then  $O(k^2)$  possible edges on the compatibility graph. Thus the time to construct  $G$  is  $O(n + k^2)$ . To find the maximum flow on  $G$ , we *cannot* necessarily use Hopcraft-Karp or Mucha-Sankowski, since we are no longer solving a strict *matching* problem (we are allowing multiple nodes in  $L$  to match to the same node in  $R$ ). Thus we should use Edmonds-Karp. The running time of Edmonds-Karp on any graph is  $O(\min(E|f^*|, VE^2))$ . In this case, if there are  $k$  possible labels, this comes out to be  $O(\min(k^2m, k^5))$ .

Thus the total running time is  $O(n + \min(k^2m, k^5))$ . If  $k$  is small enough (say  $k < m^{2/5}n^{1/5}$ ), then this offers a speed advantage over the  $O(m^2n)$  algorithm given earlier. However, in the worst case,  $k$  could be as large as  $n$ , in which case the

running time could be  $O(n^2m)$ , slightly worse than the  $O(m^2n)$  algorithm given earlier. Since this solution is incomparable to the previous one (and potentially better when  $k$  is small), it received full credit.

- Several people suggested algorithms that greedily selected compatible vertices, or tried to use some variant of the stable marriage algorithm. Magically these algorithms were purported to run in  $O(mn)$  time. Unfortunately, they did not work. If an algorithm actually did achieve that running time, it would be a major research result. Here's why. Given any bipartite graph, you can turn the perfect-matching problem into a problem about depth-1 subtree compatibility. Just construct a tree  $S$  that has one leaf for every node in  $L$ , and a tree  $T$  that has one leaf for every node in  $R$ , and make the bipartite graph the compatibility graph (also give the roots of  $T$  and  $S$  colors that are known to be compatible). Then there is a perfect matching in the original graph iff  $S$  is a subtree of  $T$ . The best known perfect matching algorithm runs in  $O(n^\omega)$ , where  $\omega$  is around 2.38. So an  $O(mn) = O(n^2)$  algorithm beating that bound would be a major breakthrough.

- (b)** Same as part (a), but now let  $S$  and  $T$  be arbitrary (not necessarily depth 1) labeled rooted trees.

**Solution:** The trick here was to use dynamic programming AND bipartite matching. With both of these tools and some fine-grained analysis, it is possible to obtain an algorithm with  $O(m^2n)$  running time.

First we create an  $m \times n$  array  $M$ . For any  $i \in S$  and  $j \in T$ , the entry  $M[i, j]$  will tell us whether the subtree rooted at  $i$  in  $S$  can be compatibly mapped to the subtree rooted at  $j$  in  $T$ . (Formally, we want  $M[i, j] = \text{TRUE}$  if there is a one-to-one function  $f$  that maps  $i$  to  $j$  and all descendants of  $i$  to descendants of  $j$  in such a way that parent-child relationships are preserved and  $i$  is always compatible with  $f(i)$ . Otherwise, we want  $M[i, j] = \text{FALSE}$ .) Clearly if we can compute  $M[i, j]$  for all  $i \in S$  and  $j \in T$ , then we just need to check whether  $M[\text{root}(S), j] = \text{TRUE}$  for some  $j \in T$  to tell if  $S$  is a compatible subtree of  $T$ .

To compute each entry of  $M[i, j]$ , we first check whether node  $i$  is compatible to node  $j$ . If so, then we need to solve a bipartite matching problem. We build a bipartite graph  $G = L \cup R$  where the vertices of  $L$  correspond to children of  $i$  in  $S$ , and the vertices of  $R$  correspond to children of  $j$  in  $T$ . We connect a vertex  $u$  of  $L$  to a vertex  $v$  of  $R$  if  $M[u, v] = \text{TRUE}$ . Then we find the maximum matching in this bipartite graph, and if it's equal to the number of children of  $i$ , we set  $M[i, j] = \text{TRUE}$ . To insure that we don't recompute answers to subproblems, we iterate over the nodes in order of decreasing depth (from the leaves to the root). The pseudocode for this algorithm follows (it assumes that the nodes are in decreasing depth order, so for any node  $i$ ,  $\text{child}(i) < i$ ):

```

1   for  $i = 1 \dots m$ 
2     for  $j = 1 \dots n$ 
3       if  $i$  is a leaf Then  $M[i, j] = \text{Compatible}(i, j, H)$ 
4       Else
5         Let  $G = \text{Construct-Bipartite-Graph}(\text{children}(i), \text{children}(j), M)$ 
6         If ( $\text{Compatible}(i, j, H)$  AND  $\text{Max-Matching}(G) = |\text{children}(i)|$ )
7           Then  $M[i, j] = \text{TRUE}$ 
8           Else  $M[i, j] = \text{FALSE}$ 

```

The “Compatible” subroutine simply takes two nodes and the compatibility graph, and returns True/False depending on whether the nodes are compatible.. The “Construct-Bipartite-Graph” subroutine uses previously computed entries of  $M$  to tell which children of  $i$  are matchable to which children of  $j$  (note that this subroutine does *not* need access to the compatibility graph, just to previously computed entries of  $M$ ). The “Max-Matching” subroutine is the Ford-Fulkerson algorithm described in part (a).

Now we analyze the running time of this algorithm. First, we give a coarse analysis. To compute each entry  $M[i, j]$ , we are solving a bipartite matching problem no bigger than that in part (a). Thus we are performing  $O(mn)$  bipartite matchings, each taking at most  $O(m^2n)$  time, for a total of at most  $O(m^3n^2)$ . This analysis is correct (and received full credit), but not tight. We can do better as follows:

Let  $m_i$  be the number of children of node  $i$  in  $S$ , and let  $n_j$  be the number of children of node  $j$  in  $T$ . Then the running time of the algorithm is bounded by

$$\begin{aligned}
\sum_i \sum_j O(m_i^2 n_j) &= O\left(\sum_i \sum_j m_i^2 n_j\right) \\
&= O\left(\sum_i m_i^2 \sum_j n_j\right) \\
&= O\left(n \sum_i m_i^2\right) \\
&\leq O\left(n \left(\sum_i m_i\right)^2\right) \\
&= O(nm^2)
\end{aligned}$$

### Problem 5. Feasible Region

Consider an instance of the linear programming problem in two dimensions (i.e., with  $m$  constraints over  $n = 2$  variables). Give an efficient algorithm that reports *all* vertices of the feasible region polygon.

**Solution:** The most efficient algorithm for this problem runs in  $O(m \log m)$  time. The algorithm actually solves a (harder) problem: given a set of given  $m$  half-spaces, find their intersection polygon, i.e., report the vertices sorted by the order they appear on the polygon.

**Near-linear time algorithm.** The  $O(m \log m)$ -time algorithm is based on sorting and the divide-and-conquer approach. First, we sort the half-spaces by their direction angles (ranging from 0 to 360 degrees), resulting in a sequence of half-spaces  $A$ . Then, we split the sequence  $A$  into two sequences  $B$  and  $C$  where  $B$  contains all half-spaces with angles in range  $[0, 180]$ . We compute the intersection  $IB$  of halfspaces in  $B$  and intersection  $IC$  of halfspaces in  $C$  using the same algorithm. In the end we intersect  $IB$  and  $IC$  using essentially the same algorithm.

The intersection  $IB$  is computed using the divide and conquer approach. We split  $B$  into two equal-size sequences  $B'$  and  $B''$ , and recursively compute their intersections  $IB'$  and  $IB''$ . The combine step is as follows. Let  $S' = s'_1 \dots s'_{t'}$  and  $S'' = s''_1 \dots s''_{t''}$  be the sequences of segments defining  $IB'$  and  $IB''$ , respectively. It now suffices to find the unique intersection between the segments in  $S'$  and  $S''$ . First, we check if the first and last segments of  $S'$  and  $S''$  contribute to such intersection; if yes, we are done. If not, then let  $s''_{i_1}$  be the segment in  $S''$  intersected by the line along segment  $s'_1$  (we can find it in  $O(m)$  time). We now observe the segment  $s''_{i_2}$  of  $S''$  that is intersected by the line along  $s'_1$  is such that  $i_2 \geq i_1$ . Therefore, we can find  $i_2$  by incrementing  $i_1$  and checking for intersection of respective segments in  $S''$ . By iterating this approach, we can find the intersection between segments in  $S'$  and  $S''$  in  $O(m)$  time.

**Quadratic-time algorithm.** A simpler (but slower) approach is to use the incremental approach. Specifically, we can construct the intersection of the halfspaces by adding one halfspace at a time. Given the current intersection region  $I$  and a new halfspace  $h$  defined by a line  $l$ , the algorithm finds the two points on the boundary of  $I$  that intersect  $l$ . This can be done by enumerating all segments describing  $I$ , and checking for intersection with  $l$ . Since the number of segments defining  $I$  is at most  $m$ , adding a new halfspace can be accomplished in  $O(m)$  time. The overall algorithm runs in time  $O(m^2)$ .

**Cubic-time algorithm.** Even simpler (but slow) algorithm proceeds by enumerating all possible candidates for vertices in the feasible region, and checking their feasibility. See Lecture 17, slide 9 for details.

## Quiz 2 Solutions

- Do not open this quiz booklet until you are directed to do so. Read all the instructions first.
- The quiz contains **5** problems. You have 120 minutes to earn **115** points.
- This quiz booklet contains **13** pages, including this one, and a sheet of scratch paper.
- This quiz is closed book. You may use two double-sided letter ( $8\frac{1}{2}'' \times 11''$ ) or A4 crib sheet. No calculators or programmable devices are permitted. Cell phones must be put away.
- Write your solutions in the space provided. If you run out of space, continue your answer on the back of the same sheet and make a notation on the front of the sheet.
- Do not waste time deriving facts that we have studied. Just cite results from class.
- When we ask you to “give an algorithm” in this quiz, describe your algorithm in English or pseudocode, and provide a short argument for correctness and running time. You do not need to provide a diagram or example unless it helps make your explanation clearer.
- Do not spend too much time on any one problem. Generally, a problem’s point value is an indication of how many minutes to spend on it.
- 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. Please be neat.
- Do not discuss the quiz with anyone until we give you permission to do so. We still have a handful of people taking make-up quizzes after tonight.
- Good luck!

Problem	Title	Points	Parts	Grade	Initials
0	Name	1	1		
1	True or False	24	8		
2	Short Answers	40	5		
3	4-Colorability	10	1		
4	Priority Queues	10	1		
5	Approximating Knapsack	30	4		
Total		115			

Name: \_\_\_\_\_

F10 R01 Nathan	F11 R02 Nathan	F11 R07 Adam	F12 R03 Bonny	F12 R08 Adam	F1 R04 Bonny	F1 R09 Casey	F2 R05 Hayden	F2 R10 Christina	F3 R06 Hayden
----------------------	----------------------	--------------------	---------------------	--------------------	--------------------	--------------------	---------------------	------------------------	---------------------

**Problem 0. Name.** [1 point] Write your name on every page of this exam booklet! Don't forget the cover.

**Problem 1. True or False.** [24 points] (8 parts)

Circle **T** or **F** for each of the following statements to indicate whether the statement is true or false and briefly explain why.

- (a) **T F** If a linear program has an optimal solution, then the optimal solution is unique.

**Solution:** False. There could be an optimal edge.

- (b) **T F** Adding a constraint to a linear program always strictly decreases the size of the feasible region.

**Solution:** False. A simple counterexample is adding  $x_1 \geq 2$  to a linear program that already has the constraint  $x_1 \geq 4$ .

- (c) **T F** Consider Seidel's algorithm presented in class for 2D Linear Programming which incrementally adds constraints one by one. Instead of adding the constraints in random order, suppose that a little birdy provides them in the order that minimizes the running time. Assume there is a unique optimal solution. Then the little birdy can ensure that the total running time of the algorithm is always linear.

**Solution:** True. The little birdy can start by providing the two constraints that define the optimum solution. Then all future constraints will be satisfied by that solution and no further updates will be needed.

- (d) **T F** Given a universal hash family  $H$ , every nonempty subset  $G \subseteq H$  is also a universal hash family.

**Solution:** False. If  $|G| = 1$ , then the probability of collision among different elements with the same hash is 1. If  $n > m$ , then there must exist two different elements with the same hash by the Pigeonhole principle.

- (e) **T F** When achieving perfect hashing with  $\Theta(n)$  space using a two-level hash table, the first and second level hash tables must both avoid collisions.

**Solution:** False. The first level can have collisions, but the second cannot.

- (f) **T F** Consider the family of hash functions  $H = \{f, g, h\}$  where each function maps  $\{0, 1, 2\} \rightarrow \{0, 1\}$ . In particular:

$$\begin{aligned} f(0) &= 1, f(1) = 0, f(2) = 0 \\ g(0) &= 0, g(1) = 1, g(2) = 0 \\ h(0) &= 0, h(1) = 0, h(2) = 1 \end{aligned}$$

Then  $H$  is a universal hash family.

**Solution:** True. This just follows from the definition. For every  $x \neq y \in \{0, 1, 2\}$ , the probability that  $a(x) = a(y)$  over  $a \in H$  is  $1/3$ , which is less than  $1/m$  where  $m = 2$ .

- (g) T F Finding the maximum independent set on graphs that are binary trees is NP-complete.

**Solution:** False. We can solve this in polynomial time using a dynamic programming algorithm where each subproblem computes the maximum independent set on the subtree rooted at a particular node. This problem is exactly the same as homework problem 2.2 if we set each node's value to 1.

- (h) T F The LRU paging strategy is  $2k$ -competitive, where  $k$  is the size of the cache.

**Solution:** True. LRU is  $k$ -competitive, and by definition, if it's  $k$ -competitive, it's also  $2k$ -competitive.

**Problem 2. Short Answers.** [40 points] (5 parts)

Please answer the following questions.

- (a) [8 points] Consider the  $k$ -Coloring problem: Given a graph  $G = (V, E)$ , use at most  $k$  colors to color the graph such that no two adjacent nodes get the same color. Assume that you are given a graph that is  $k$ -colorable. Provide a polytime algorithm that finds a coloring that uses at most  $\frac{nk}{\log(n)}$  colors. Note that  $k$  is a constant.

**Solution:** Modify the algorithm we used in class for a  $n/\log(n)$  approximation of Clique. Take the set of vertices  $V$ , and partition them into  $n/\log(n)$  groups. For each group, there is at most  $\log(n)$  vertices. Simply brute force over all possible assignment of  $k$  colors to this graph, and find a valid  $k$ -coloring. This takes at most  $O(k^{\log(n)}) = O(nk)$  time for each group. We use a set of  $k$  new colors for each group, and thus in total we use a maximum of  $nk/\log(n)$  colors in this valid coloring.

Common mistake: The algorithm of “pick a node of high degree and k-color its neighbors” does not work since we don’t know how to k-color the neighbors in poly time. The algorithm we did in class only needed to 2-color, which we know how to solve in polytime.

- (b) [8 points] Given a graph with maximum degree  $d$ , give a polynomial time algorithm to find an independent set of size at least  $n/(d + 1)$ .

**Solution:** Naively pick any vertex, put it in the independent set and remove it and its neighbors (which is at most  $d + 1$ ). In each iteration, the algorithm removes at most  $d + 1$  vertices, thus after choosing  $k$  vertices, there are at least  $n - k(d + 1)$  vertices left. Thus if  $k$  is strictly less than  $n/(d + 1)$ , after removing any  $k$  vertices, there will still be remaining vertices that are not adjacent to any of the vertices chosen in the independent set. Thus, this algorithm always gives an independent set of size at least  $n/(d + 1)$ .

- (c) [8 points] The MAXEKSAT problem is similar to KSAT, but attempts to maximize the total number of clauses which are satisfied. It also has the additional constraint that each clause has exactly  $k$  distinct literals. Consider a randomized algorithm for MAXEKSAT which simply chooses a random assignment for each variable. What is the expected number of clauses satisfied by this algorithm?

**Solution:** Let  $m$  be the number of clauses. Any given clause is false with probability  $(1/2^k)$ , and therefore true with probability

$$1 - \frac{1}{2^k}.$$

By linearity of expectation, the expected number of clauses satisfied is  $\left(1 - \frac{1}{2^k}\right)m$ .

- (d) [8 points] Consider a cow facing a fence. The cow wants to get around the fence. It knows that there is an opening  $k$  steps away, but it does not know in which direction the opening is. Give a 3-competitive algorithm for the cow to find the opening in the fence.

**Solution:** Begin by walking  $k$  steps to the right. If you haven't found the hole, walk  $2k$  steps to the left. By then you must have found the opening. It took a total of  $3k$  steps and the optimal offline algorithm would take  $k$  steps, so the algorithm is  $k$ -competitive.

- (e) [8 points] If  $H$  is a universal family of hash functions  $h : \{1 \dots u\} \rightarrow \{1 \dots 7\}$ , show that for any set  $K$  of 4 keys there is always  $h \in H$  that is perfect (no collisions) on  $K$ .

**Solution:** The expected number of collisions is  $\binom{4}{2}/7 = 6/7 < 1$ . By Markov property, the probability that a randomly chosen  $h \in H$  is perfect on  $K$  is strictly greater than zero. Thus there is at least one hash function with zero collisions.

**Problem 3. 4-Colorability [10 points]**

Show that 4-colorability is NP-Complete. Recall that the 4-colorability problem takes as input a graph  $G = (V, E)$ , and outputs whether there exists an assignment of colors to the vertices such that for every edge, the two endpoints have different colors, and at most 4 colors are used.

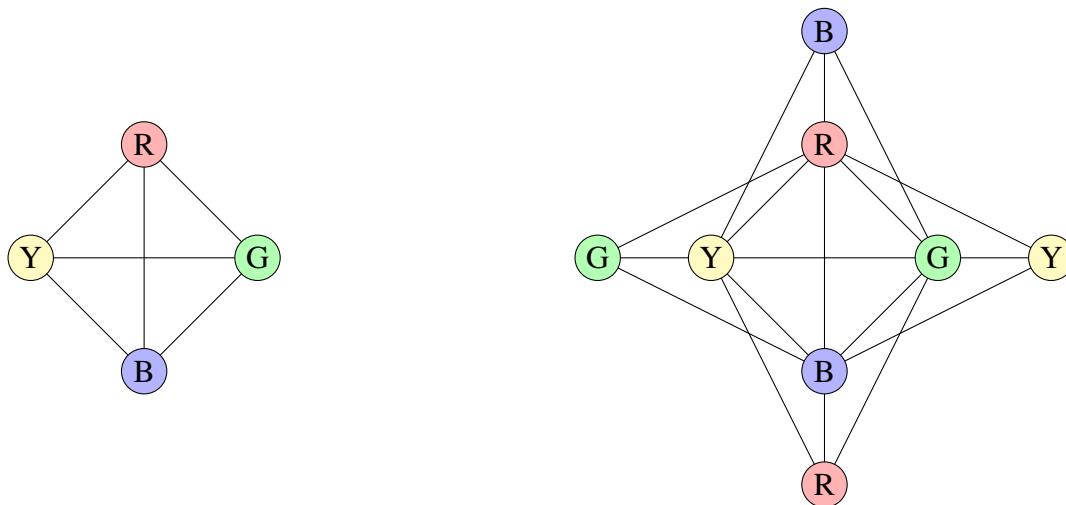
*Hint: Reduce from 3-colorability.*

**Solution:** Firstly, 4-colorability is in NP because our certificate can just be a valid 4-coloring of the graph.

For the reduction, let  $G = (V, E)$  be the graph we want to test for 3-colorability. Add a new vertex  $v$  and draw  $|V|$  new edges connecting  $v$  to every other vertex in  $V$ . Given a valid coloring of this new graph,  $v$  must have a different color from all the other vertices, so if the new graph is 4-colorable, then  $G$  must be 3-colorable. Conversely, if  $G$  is 3-colorable, then the new graph is 4-colorable because you can assign a fourth color to the one vertex you added without violating any constraints. Therefore 4-coloring is NP-hard.

**Grader's Notes.** There were several common mistakes that people made on this problem. One of the most common mistakes was performing the reduction in the wrong direction. Reducing from 3-colorability to 4-colorability means that you must show how to solve 3-colorability *by using 4-colorability as a black box*, not the other way around.

There were also several approaches that some people tried that did not work. One example was adding a vertex  $u$  for every set of three vertices  $v_1, v_2, v_3$  that formed a clique and adding edges  $(u, v_1), (u, v_2), (u, v_3)$ , and asserting that the original graph is 3-colorable if and only if the transformed graph is 4-colorable. This would not work on the graph shown below, with the original graph on the left and the transformed graph on the right:



The transformed graph is 4-colorable, but the original graph is not 3-colorable.

Another common mistake was to simply add a new vertex  $v'_i$  for each  $v_i \in V$  and an edge  $(v_i, v'_i)$ . In order for this to work you would need all the new vertices to be the same color, but this transformation does not constrain them in any way. You could make this work by adding three additional vertices  $u_1, u_2, u_3$  with edges  $(u_1, u_2), (u_1, u_3), (u_2, u_3)$  and connecting each  $v'_i$  to each of the  $u_j$ 's. Since the  $u_j$ 's will use 3 different colors, each  $v'_i$  must take on the fourth color, which means that the original graph needs to be 3-colorable.

A third mistake was to split each edge into two edges and create a vertex in between. In other words, transform  $G = (V, E)$  into  $G' = (V', E')$  where  $V' = \{v_1, \dots, v_n, u_1, \dots, u_m\}$  (where  $n = |V|$  and  $m = |E|$ ) and  $E' = \{(v_i, u_j) \mid \text{edge } j \text{ is incident to vertex } i \text{ in the original graph } G\}$ . The assertion was that  $G$  is 3-colorable if and only if  $G'$  is 4-colorable. However, regardless of the structure of  $G$ ,  $G'$  is always 2-colorable, since you can color each  $v_i$  one color and each  $u_j$  a second color.

**Problem 4. Priority Queues [10 points]**

Consider a priority queue  $P$  which supports the following operations:

- 1.INSERT( $P, x$ ) in  $O(\log n)$
- 2.EXTRACT-MIN( $P$ ) in  $O(\log n)$
- 3.MERGE( $P_1, P_2$ ) in  $O(\log n)$
- 4.MAX-HEAPIFY( $P$ ) in  $O(n)$

Say that we construct a new priority queue  $P^*$  which consists of a normal priority queue  $P$  and a linked-list  $L$ . It implements its operations as follows:

- 1.INSERT( $P^*, x$ ): Append  $x$  to the end of  $L$ .
- 2.EXTRACT-MIN( $P^*$ ):
  - Make  $L$  into a heap (via MAX-HEAPIFY( $L$ ))
  - Merge  $P$  and  $L$  into a new priority queue  $Q$  (via MERGE( $P, L$ ))
  - Extract the minimum from  $Q$  (via EXTRACT-MIN( $P$ )).

Consider the potential function  $\phi = |L|$ . Show that for any sequence of INSERTS and EXTRACT-MINS, the amortized cost of an INSERT operation is  $O(1)$  and the amortized cost of an EXTRACT-MIN operation is  $O(\log n)$ .

**Solution:** The real cost of insertion is  $O(1)$  because we just have to append an element to the end of a linked list.  $\Delta\phi$  is 1, because we increase the size of  $L$  by 1. So the real cost is bounded above by  $O(1)$ .

Let  $s = |L|$ . The real cost of EXTRACT-MIN is  $O(\log n + s)$ . Calling MAX-HEAPIFY( $L$ ) takes  $O(s)$ , merging  $P$  and  $L$  takes  $O(\log n)$ , and extracting the minimum of the new heap takes  $O(\log n)$ .  $\Delta\phi$  is  $-s$ , since we remove all the elements from  $L$ . So then the real cost is bounded above by

$$O(\log n + s - s) = O(\log n).$$

**Problem 5. Approximating Knapsack** [30 points] (4 parts)

Consider the Knapsack optimization problem we saw in recitation. We introduce a modification such that there is only a single object per item type. Given  $n$  items such that item  $i$  has value  $v_i$  and weight  $w_i$ , and given a bag that can hold a total weight at most  $W$ , we want to choose a subset of these items which fits in the bag and has maximum value. Assume that  $v_i/w_i$  is distinct for all items  $i$ , and that  $w_i \leq W$  for all items  $i$ .

- (a) [5 points] One obvious algorithm to try is a greedy algorithm: sort the objects in decreasing order of value per unit weight ( $v_i/w_i$ ) and go through the list in order, taking an object if it still fits in the bag. Show that there are inputs (choices of  $v_i, w_i, W$ ) on which this algorithm might give less than or equal to  $1/10$  of the value of the optimal subset of items.

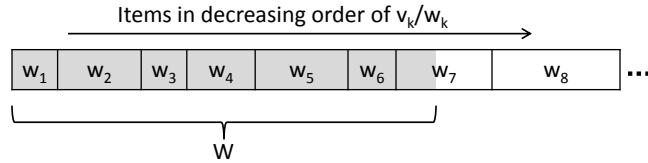
**Solution:** Suppose we have  $W = 20, v_1 = 20, w_1 = 20, v_2 = 2, w_2 = 1$ . The greedy algorithm only takes object 2, for a total value of 2, when we could have had value 20. So we get only a  $1/10$  fraction of the optimal total worth.

- (b) [5 points] The following is an integer linear program for solving Knapsack. Fill in the missing constraint.

$$\begin{aligned} & \text{maximize} \sum_{k=1}^n v_k x_k \\ & \text{subject to (missing constraint)} \\ & \quad x_k \in \{0, 1\} \quad \forall k \end{aligned}$$

**Solution:**  $\sum_{k=1}^n w_k x_k \leq W$

- (c) [10 points] Consider the relaxed Linear Program, where we replace  $x_k \in \{0, 1\}$  by  $x_k \in [0, 1]$ . Show that the unique optimal solution of the relaxed LP is given by the greedy fractional algorithm depicted below.



In words, sort the items in decreasing order by  $v_i/w_i$ . If the item fits in the bag, take it. Let  $j$  denote the first item that does not fully fit in the bag. Take the largest fraction of item  $j$  to completely fill the remaining space in the bag. Thus the optimal solution is to set  $x_i = 1$  for all items considered before item  $j$ , to set  $x_j$  equal to the remaining weight divided by  $w_j$ , and to set  $x_i = 0$  for all items considered after item  $j$ .

**Solution:** Basic idea: Prove by contradiction. Given another solution, it must take less of the first  $j$  items (since the optimal solution maximizes this), and it must take some weight from a less valuable item (value per unit weight). Thus we can just shift the assignment to a more valuable item and increase the objective function.

(Guilio Gueltrini's solution - good job!) Assume that in the best solution there exists some  $i, j$  such that  $x_i < x_j$  and

$$\frac{v_i}{w_i} > \frac{v_j}{w_j}.$$

This assumption is equivalent to assuming that the greedy algorithm is not optimal. We define  $w^*$  to be the space occupied by elements  $i$  and  $j$  in our knapsack, and  $v^*$  to be the value of the fraction of the two objects in the knapsack.

$$w^* = x_i w_i + x_j w_j.$$

$$v^* = x_i v_i + x_j v_j.$$

Now we consider a new solution where  $x'_i w_i + x'_j w_j = w^*$ ,  $x'_i > x_i$ , and  $x'_j < x_j$ . We can always find such a  $x'_i, x'_j$ , since  $x_i < x_j$ , so  $x_i < 1$ , and  $x_j > 0$ . Now we have found a new solution which still satisfies the constraints given by the weights, but has a strictly higher value. The change in the value of items in the knapsack is

$$(x'_i v_i + x'_j v_j) - (x_i v_i + x_j v_j) > 0.$$

This contradicts the assumption that  $x$  was an optimal solution, therefore the initial assumption is wrong, and the greedy solution must be optimal.

A common mistake was to state that proceeding with the greedy algorithm would maximize the total value. This does not actually constitute a proof of optimality.

- (d) [10 points] Consider the following approximation algorithm: Solve the relaxed LP for knapsack. From the previous part, we know that there is at most one item which has a fractional value for  $x_i$ . Denote this item to be  $j$ . Let  $A$  denote the set of items such that  $x_i = 1$ . Compare the total value of the set  $A$  with the value of the single item  $j$ , and choose the more valuable set. Show that this is a 2-approximation algorithm.

**Solution:** Let  $f_{OPT}$  be the value of the true optimal solution of Knapsack. Let  $f_{LP}$  be the value of the solution to the relaxed LP. Let  $f_{APX}$  be the value of the approximation algorithm. We know that  $f_{LP} \geq f_{OPT}$  because the relaxed LP optimizes over a strictly larger feasible region. Then observe that  $\text{value}(A) + v_j$  is greater than  $f_{LP}$ , since it takes the whole item  $j$ . Since the approximation algorithm picks the largest of  $\text{value}(A)$  and  $v_j$ ,

$$f_{APX} = \max(\text{value}(A), v_j) \geq \frac{1}{2}(\text{value}(A) + v_j) \geq \frac{1}{2}f_{LP} \geq \frac{1}{2}f_{OPT}.$$

Therefore this algorithm gives a 2-approximation for knapsack.

Common mistakes: Many people did not really compare to the knapsack solution, but rather stopped at claiming that this approximation is at least half of the optimal for the relaxed LP. Note that we do not need to find an approximation algorithm for the relaxed LP since we know how to solve it exactly. The goal is to approximate knapsack (which is exactly equivalent to the *integer* LP).

Many people also assumed that the optimal solution somehow contains a subset of items  $A \cup j$ , and thus tried to argue that by choosing  $A$  or  $j$  you could not add more valuable items to the set. The optimal solution in fact does not even need to include any of the items in  $A \cup j$ . For example consider a problem with three items: item 1 has weight 6, value 6.5; item 2 has weight 6, value 6; and item 3 has weight 10, value 9. The knapsack has size 10. Then set  $A$  consists of item 1. Item  $j$  denotes item 2. The approximation algorithm will choose item 1. However the optimal set is item 3, which is not even included in  $A \cup j$ .

Also people made assumptions or conclusions about the value of  $x_j$  or the weight of the final approximation output. Note that  $x_j$  can be anything within 0 or 1. There is nothing about the problem that implies it is greater or less than  $\frac{1}{2}$ . Also the final set chosen does not need to have weight at least  $\frac{1}{2}$ . For example the weight of all items in set  $A$  can be very small compared to  $W$ , yet the value of  $A$  can still be larger than  $v_j$ .

## SCRATCH PAPER