CS 473U: Undergraduate Algorithms, Fall 2006
Homework 0

Due Friday, September 1, 2006 at noon in 3229 Siebel Center

Name:	
Net ID:	Alias:

☐ I understand the Homework Instructions and FAQ.

- Neatly print your full name, your NetID, and an alias of your choice in the boxes above, and submit this page with your solutions. We will list homework and exam grades on the course web site by alias. For privacy reasons, your alias should not resemble your name, your NetID, your university ID number, or (God forbid) your Social Security number. Please use the same alias for every homework and exam.

  Federal law forbids us from publishing your grades, even anonymously, without your explicit permission. **By providing an alias, you grant us permission to list your grades on the course web site; if you do not provide an alias, your grades will not be listed.**

- Please carefully read the Homework Instructions and FAQ on the course web page, and then check the box above. This page describes what we expect in your homework solutions—start each numbered problem on a new sheet of paper, write your name and NetID on every page, don't turn in source code, analyze and prove everything, use good English and good logic, and so on—as well as policies on grading standards, regrading, and plagiarism. **See especially the policies regarding the magic phrases "I don't know" and "and so on".** If you have *any* questions, post them to the course newsgroup or ask in lecture.

- This homework tests your familiarity with prerequisite material—basic data structures, big-Oh notation, recurrences, discrete probability, and most importantly, induction—to help you identify gaps in your knowledge. **You are responsible for filling those gaps on your own.** Each numbered problem is worth 10 points; not all subproblems have equal weight.

#	1	2	3	4	5	6*	Total
Score							
Grader							

Please put your answers to problems 1 and 2 on the same page.

1. Sort the functions listed below from asymptotically smallest to asymptotically largest, indicating ties if there are any. **Do not turn in proofs**, but you should probably do them anyway, just for practice.

   To simplify your answers, write $f(n) \ll g(n)$ to mean $f(n) = o(g(n))$, and write $f(n) \equiv g(n)$ to mean $f(n) = \Theta(g(n))$. For example, the functions $n^2, n, \binom{n}{2}, n^3$ could be sorted either as $n \ll n^2 \equiv \binom{n}{2} \ll n^3$ or as $n \ll \binom{n}{2} \equiv n^2 \ll n^3$.

   $$\begin{array}{cccccccc} \lg n & \ln n & \sqrt{n} & n & n \lg n & n^2 & 2^n & n^{1/n} \\ n^{1+1/\lg n} & \lg^{1000} n & 2^{\sqrt{\lg n}} & (\sqrt{2})^{\lg n} & \lg^{\sqrt{2}} n & n^{\sqrt{2}} & (1+\frac{1}{n})^n & n^{1/1000} \\ H_n & H_{\sqrt{n}} & 2^{H_n} & H_{2^n} & F_n & F_{n/2} & \lg F_n & F_{\lg n} \end{array}$$

   In case you've forgotten:

   - $\lg n = \log_2 n \neq \ln n = \log_e n$
   - $\lg^3 n = (\lg n)^3 \neq \lg \lg \lg n$.
   - The harmonic numbers: $H_n = \sum_{i=1}^{n} 1/i \approx \ln n + 0.577215\ldots$
   - The Fibonacci numbers: $F_0 = 0$, $F_1 = 1$, $F_n = F_{n-1} + F_{n-2}$ for all $n \geq 2$

2. Solve the following recurrences. State tight asymptotic bounds for each function in the form $\Theta(f(n))$ for some recognizable function $f(n)$. Proofs are *not* required; just give us the list of answers. **Don't turn in proofs**, but you should do them anyway, just for practice. Assume reasonable but nontrivial base cases. **If your solution requires specific base cases, state them.** Extra credit will be awarded for more exact solutions.

   (a) $A(n) = 2A(n/4) + \sqrt{n}$

   (b) $B(n) = 3B(n/3) + n/\lg n$

   (c) $C(n) = \dfrac{2C(n-1)}{C(n-2)}$ *[Hint: This is easy!]*

   (d) $D(n) = D(n-1) + 1/n$

   (e) $E(n) = E(n/2) + D(n)$

   (f) $F(n) = 4F\left(\left\lceil \dfrac{n-8}{2} \right\rceil + \left\lfloor \dfrac{3n}{\log_\pi n} \right\rfloor\right) + 6\dbinom{n+5}{2} - 42n\lg^7 n + \sqrt{13n-6} + \dfrac{\lg\lg n + 1}{\lg n \lg\lg\lg n}$

   (g) $G(n) = 2G(n-1) - G(n-2) + n$

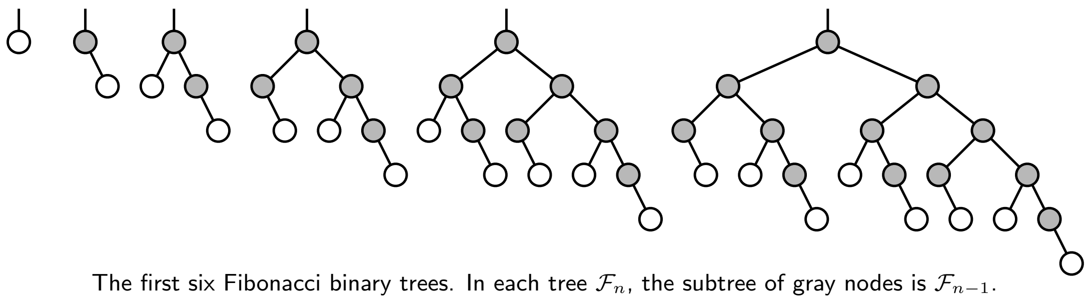
   (h) $H(n) = 2H(n/2) - 2H(n/4) + 2^n$

   (i) $I(n) = I(n/2) + I(n/4) + I(n/6) + I(n/12) + n$

   *(j) $J(n) = \sqrt{n} \cdot J(2\sqrt{n}) + n$
   [Hint: First solve the secondary recurrence $j(n) = 1 + j(2\sqrt{n})$.]

3. The nth *Fibonacci binary tree* $\mathcal{F}_n$ is defined recursively as follows:

   - $\mathcal{F}_1$ is a single root node with no children.
   - For all $n \geq 2$, $\mathcal{F}_n$ is obtained from $\mathcal{F}_{n-1}$ by adding a right child to every leaf and adding a left child to every node that has only one child.



The first six Fibonacci binary trees. In each tree $\mathcal{F}_n$, the subtree of gray nodes is $\mathcal{F}_{n-1}$.

   (a) Prove that the number of leaves in $\mathcal{F}_n$ is precisely the nth Fibonacci number: $F_0 = 0$, $F_1 = 1$, and $F_n = F_{n-1} + F_{n-2}$ for all $n \geq 2$.

   (b) How many nodes does $\mathcal{F}_n$ have? For full credit, give an *exact*, closed-form answer in terms of Fibonacci numbers, and prove your answer is correct.

   (c) Prove that the left subtree of $\mathcal{F}_n$ is a copy of $\mathcal{F}_{n-2}$.

4. Describe and analyze a data structure that stores set of n records, each with a numerical *key* and a numerical *priority*, such that the following operation can be performed quickly:

   RANGETOP(a, z) : return the highest-priority record whose key is between a and z.

   For example, if the (key, priority) pairs are

$$(3,1),\ (4,9),\ (9,2),\ (6,3),\ (5,8),\ (7,5),\ (1,4),\ (0,7),$$

   then RANGETOP$(2, 8)$ would return the record with key 4 and priority 9 (the second record in the list).

   You may assume that no two records have equal keys or equal priorities, and that no record has a key equal to a or z. Analyze both the size of your data structure and the running time of your RANGETOP algorithm. For full credit, your data structure must be as small as possible *and* your RANGETOP algorithm must be as fast as possible.

   [Hint: How would you compute the number *of keys between a and z? How would you solve the problem if you knew that a is always $-\infty$?]*

5. Penn and Teller agree to play the following game. Penn shuffles a standard deck[1] of playing cards so that every permutation is equally likely. Then Teller draws cards from the deck, one at a time without replacement, until he draws the three of clubs (3♣), at which point the remaining undrawn cards instantly burst into flames.

   The first time Teller draws a card from the deck, he gives it to Penn. From then on, until the game ends, whenever Teller draws a card whose value is smaller than the last card he gave to Penn, he gives the new card to Penn.[2] To make the rules unambiguous, they agree beforehand that $A = 1$, $J = 11$, $Q = 12$, and $K = 13$.

   (a) What is the expected number of cards that Teller draws?
   (b) What is the expected *maximum* value among the cards Teller gives to Penn?
   (c) What is the expected *minimum* value among the cards Teller gives to Penn?
   (d) What is the expected number of cards that Teller gives to Penn?

   Full credit will be given only for *exact* answers (with correct proofs, of course).

*6. **[Extra credit]**[3]

   Lazy binary is a variant of standard binary notation for representing natural numbers where we allow each "bit" to take on one of three values: 0, 1, or 2. Lazy binary notation is defined inductively as follows.

   - The lazy binary representation of zero is 0.
   - Given the lazy binary representation of any non-negative integer n, we can construct the lazy binary representation of $n + 1$ as follows:
     (a) increment the rightmost digit;
     (b) if any digit is equal to 2, replace the rightmost 2 with 0 and increment the digit immediately to its left.

   Here are the first several natural numbers in lazy binary notation:

   0, 1, 10, 11, 20, 101, 110, 111, 120, 201, 210, 1011, 1020, 1101, 1110, 1111, 1120, 1201, 1210, 2011, 2020, 2101, 2110, 10111, 10120, 10201, 10210, 11011, 11020, 11101, 11110, 11111, 11120, 11201, 11210, 12011, 12020, 12101, 12110, 20111, 20120, 20201, 20210, 21011, 21020, 21101, 21110, 101111, 101120, 101201, 101210, 102011, 102020, 102101, 102110, . . .

   (a) Prove that in any lazy binary number, between any two 2s there is at least one 0, and between two 0s there is at least one 2.
   (b) Prove that for any natural number N, the sum of the digits of the lazy binary representation of N is exactly $\lfloor \lg(N + 1) \rfloor$.

[1] In a standard deck of 52 cards, each card has a *suit* in the set $\{\spadesuit, \heartsuit, \clubsuit, \diamondsuit\}$ and a *value* in the set $\{A, 2, 3, 4, 5, 6, 7, 8, 9, 10, J, Q, K\}$, and every possible suit-value pair appears in the deck exactly once. Actually, to make the game more interesting, Penn and Teller normally use razor-sharp ninja throwing cards.

[2] Specifically, he hurls them from the opposite side of the stage directly into the back of Penn's right hand.

[3] The "I don't know" rule does not apply to extra credit problems. There is no such thing as "partial extra credit".

CS 473U: Undergraduate Algorithms, Fall 2006
Homework 1

Due Tuesday, September 12, 2006 in 3229 Siebel Center

Starting with this homework, groups of up to three students can submit or present a single joint solution. If your group is submitting a written solution, please remember to **print the names, NetIDs, and aliases of *every* group member on *every* page**. Please remember to submit **separate, individually stapled** solutions to each of the problems.

---

1. Recall from lecture that a *subsequence* of a sequence A consists of a (not necessarily contiguous) collection of elements of A, arranged in the same order as they appear in A. If B is a subsequence of A, then A is a *supersequence* of B.

   (a) Describe and analyze a **simple** recursive algorithm to compute, given two sequences A and B, the length of the *longest common subsequence* of A and B. For example, given the strings $\texttt{\underline{AL}GO\underline{RIT}HM}$ and $\texttt{\underline{AL}T\underline{R}U\underline{I}S\underline{T}IC}$, your algorithm would return 5, the length of the longest common subsequence `ALRIT`.

   (b) Describe and analyze a **simple** recursive algorithm to compute, given two sequences A and B, the length of a *shortest common supersequence* of A and B. For example, given the strings `ALGORITHM` and `ALTRUISTIC`, your algorithm would return 14, the length of the shortest common supersequence $\texttt{\overline{\underline{AL}}\underline{G}\overline{T}\underline{O}\overline{\underline{R}U\underline{I}S\underline{T}}\underline{H}\overline{I}\underline{M}\overline{C}}$.

   (c) Let $|A|$ denote the length of sequence A. For any two sequences A and B, let $\mathrm{lcs}(A, B)$ denote the length of the longest common subsequence of A and B, and let $\mathrm{scs}(A, B)$ denote the length of the shortest common supersequence of A and B.
   Prove that $\boldsymbol{|A| + |B| = \mathrm{lcs}(A, B) + \mathrm{scs}(A, B)}$ for all sequences A and B. *[Hint: There is a simple non-inductive proof.]*

   In parts (a) and (b), we are *not* looking for the most efficient algorithms, but for algorithms with simple and correct recursive structure.

2. You are a contestant on a game show, and it is your turn to compete in the following game. You are presented with an $m \times n$ grid of boxes, each containing a unique number. It costs \$100 to open a box. Your goal is to find a box whose number is larger than its neighbors in the grid (above, below, left, and right). If you spend less money than your opponents, you win a week-long trip for two to Las Vegas and a year's supply of Rice-A-Roni™, to which you are hopelessly addicted.

   (a) Suppose $m = 1$. Describe an algorithm that finds a number that is bigger than any of its neighbors. How many boxes does your algorithm open in the worst case?

   (b) Suppose $m = n$. Describe an algorithm that finds a number that is bigger than any of its neighbors. How many boxes does your algorithm open in the worst case?

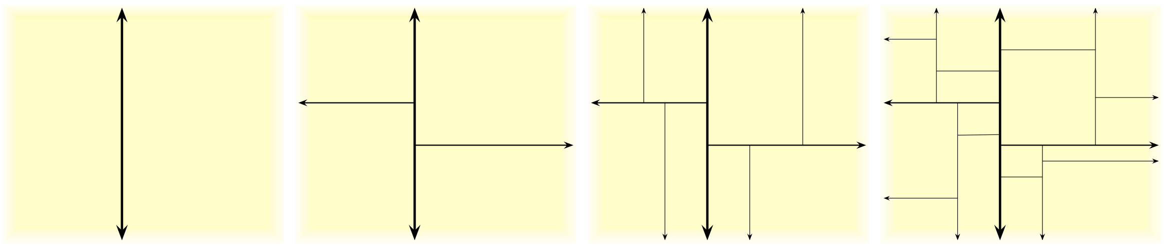
   *(c) **[Extra credit]**[1] Prove that your solution to part (b) is asymptotically optimal.

[1]The "I don't know" rule does not apply to extra credit problems. There is no such thing as "partial extra credit".

3. A kd-tree is a rooted binary tree with three types of nodes: horizontal, vertical, and leaf. Each vertical node has a *left* child and a *right* child; each horizontal node has a *high* child and a *low* child. The non-leaf node types alternate: non-leaf children of vertical nodes are horizontal and vice versa. Each non-leaf node v stores a real number p_v called its *pivot value*. Each node v has an associated *region* $R(v)$, defined recursively as follows:

   - $R(\mathit{root})$ is the entire plane.
   - If v is is a horizontal node, the horizontal line $y = p_v$ partitions $R(v)$ into $R(\mathit{high}(v))$ and $R(\mathit{low}(v))$ in the obvious way.
   - If v is is a vertical node, the vertical line $x = p_v$ partitions $R(v)$ into $R(\mathit{left}(v))$ and $R(\mathit{right}(v))$ in the obvious way.

   Thus, each region $R(v)$ is an axis-aligned rectangle, possibly with one or more sides at infinity. If v is a leaf, we call $R(v)$ a *leaf box*.


The first four levels of a typical kd-tree.

   Suppose T is a perfectly balanced kd-tree with n leaves (and thus with depth exactly $\lg n$).

   (a) Consider the horizontal line $y = t$, where $t \neq p_v$ for all nodes v in T. *Exactly* how many leaf boxes of T does this line intersect? *[Hint: The parity of the root node matters.]* Prove your answer is correct. A correct $\Theta(\cdot)$ bound is worth significant partial credit.

   (b) Describe and analyze an efficient algorithm to compute, given T and an arbitrary horizontal line ℓ, the number of leaf boxes of T that lie *entirely above* ℓ.

CS 473U: Undergraduate Algorithms, Fall 2006
Homework 2

Due Tuesday, September 19, 2006 in 3229 Siebel Center
Remember to turn in in separate, individually stapled solutions to each of the problems.

---

1. You are given an $m \times n$ matrix M in which each entry is a 0 or 1. A *solid block* is a rectangular subset of M in which each entry is 1. Give a correct efficent algorithm to find a solid block in M with maximum area.

$$\begin{array}{ccccc}
1 & 1 & 0 & 1 & 1 \\ \cline{2-4}
0 & \multicolumn{1}{|c}{1} & 1 & \multicolumn{1}{c|}{1} & 0 \\
1 & \multicolumn{1}{|c}{1} & 1 & \multicolumn{1}{c|}{1} & 1 \\ \cline{2-4}
1 & 1 & 0 & 1 & 1
\end{array}$$

An algorithm that runs in $\Theta(n^c)$ time will earn $19 - 3c$ points.

2. You are a bus driver with a soda fountain machine in the back and a bus full of very hyper students, who are drinking more soda as they ride along the highway. Your goal is to drop the students off as quickly as possible. More specifically, every minute that a student is on your bus, he drinks another ounce of soda. Your goal is to drop the students off quickly, so that in total they drink as little soda as possible.

   You know how many students will get off of the bus at each exit. Your bus begins partway along the highway (probably not at either end), and moves at a constant rate. You must drive the bus along the highway- however you may drive forward to one exit then backward to an exit in the other direction, switching as often as you like (you can stop the bus, drop off students, and turn around instantaneously).

   Give an efficient algorithm to drop the students off so that they drink as little soda as possible. The input to the algorithm should be: the bus route (a list of the exits, together with the travel time between successive exits), the number of students you will drop off at each exit, and the current location of your bus (you may assume it is at an exit).

3. Suppose we want to display a paragraph of text on a computer screen. The text consists of n words, where the ith word is p_i pixels wide. We want to break the paragraph into several lines, each exactly P pixels long. Depending on which words we put on each line, we will need to insert different amounts of white space between the words. The paragraph should be fully justified, meaning that the first word on each line starts at its leftmost pixel, and *except for the last line*, the last character on each line ends at its rightmost pixel. There must be at least one pixel of whitespace between any two words on the same line.

   Define the *slop* of a paragraph layout as the sum over all lines, *except the last*, of the cube of the number of extra white-space pixels in each line (not counting the one pixel required between every adjacent pair of words). Specifically, if a line contains words i through j, then the amount of extra white space on that line is $P - j + i - \sum_{k=i}^{j} P_k$. Describe a dynamic programming algorithm to print the paragraph with minimum slop.

CS 473U: Undergraduate Algorithms, Fall 2006
Homework 3

Due Wednesday, October 4, 2006 in 3229 Siebel Center

Remember to turn in separate, individually stapled solutions to each of the problems.

---

1. Consider a perfect tree of height h, where every non-leaf node has 3 children. (Therefore, each of the 3^h leaves is at distance h from the root.) Every leaf has a boolean value associated with it - either 0 or 1. Every internal node gets the boolean value assigned to the majority of its children. Given the values assigned to the leaves, we want to find an algorithm that computes the value (0 or 1) of the root.

   It is not hard to find a (deterministic) algorithm that looks at every leaf and correctly determines the value of the root, but this takes $O(3^h)$ time. Describe and analyze a *randomized* algorithm that, on average, looks at asymptotically fewer leaves. That is, the expected number of leaves your algorithm examines should be $o(3^h)$.

2. We define a *meldable heap* to be a binary tree of elements, each of which has a priority, such that the priority of any node is less than the priority of its parent. (Note that the heap does **not** have to be balanced, and that the element with greatest priority is the root.) We also define the priority of a heap to be the priority of its root.

   The *meld* operation takes as input two (meldable) heaps and returns a single meldable heap H that contains all the elements of both input heaps. We define *meld* as follows:

   - Let H_1 be the input heap with greater priority, and H_2 the input heap with lower priority. (That is, the priority of $root(H_1)$ is greater than the priority of $root(H_2)$.) Let H_L be the left subtree of $root(H_1)$ and H_R be the right subtree of $root(H_1)$.
   - We set $root(H) = root(H_1)$.
   - We now flip a coin that comes up either "Left" or "Right" with equal probability.
     - If it comes up "Left", we set the left subtree of $root(H)$ to be H_L, and the right subtree of $root(H)$ to be $meld(H_R, H_2)$ (defined recursively).
     - If the coin comes up "Right", we set the right subtree of $root(H)$ to be H_R, and the left subtree of $root(H)$ to be $meld(H_L, H_2)$.
   - As a base case, melding any heap H_1 with an empty heap gives H_1.

   (a) Analyze the expected running time of $meld(H_a, H_b)$ if H_a is a (meldable) heap with n elements, and H_b is a (meldable) heap with m elements.

   (b) Describe how to perform each of the following operations using only melds, and give the running time of each.

   - $DeleteMax(H)$, which deletes the element with greatest priority.
   - $Insert(H, x)$, which inserts the element x into the heap H.
   - $Delete(H, x)$, which - given a pointer to element x in heap H - returns the heap with x deleted.

3. Randomized Selection. Given an (unsorted) array of n distinct elements and an integer k, SELECTION is the problem of finding the kth smallest element in the array. One easy solution is to sort the array in increasing order, and then look up the kth entry, but this takes $\Theta(n \log n)$ time. The randomized algorithm below attempts to do better, at least on average.

```
QuickSelect(Array $A$, $n$, $k$)
$pivot \leftarrow$ Random(1, $n$)
$S \leftarrow \{x \mid x \in A, x < A[pivot]\}$
$s \leftarrow |S|$
$L \leftarrow \{x \mid x \in A, x > A[pivot]\}$
if $(k = s + 1)$
   return $A[pivot]$
else if $(k \leq s)$
   return QuickSelect($S, s, k$)
else
   return QuickSelect($L, n - (s + 1), k - (s + 1)$)
```

Here we assume that Random(a, b) returns an integer chosen uniformly at random from a to b (inclusive of a and b). The pivot position is randomly chosen; S is the set of elements smaller than the pivot element, and L the set of elements larger than the pivot. The sets S and L are found by comparing every other element of A to the pivot. We partition the elements into these two 'halves', and recurse on the appropriate half.

(a) Write a recurrence relation for the expected running time of QuickSelect.

(b) Given any two elements $x, y \in A$, what is the probability that x and y will be compared?

(c) Either from part (a) or part (b), find the expected running time of QuickSelect.

4. **[Extra Credit]**: In the previous problem, we found a $\Theta(n)$ algorithm for selecting the kth smallest element, but the constant hidden in the $\Theta(\cdot)$ notation is somewhat large. It is easy to find the *smallest* element using at most n comparisons; we would like to be able to extend this to larger k. Can you find a randomized algorithm that uses $n + \Theta(k \log k \log n)$[1] expected comparisons? (Note that there is no constant multiplying the n.)

Hint: While scanning through a random permutation of n elements, how many times does the smallest element seen so far change? (See HBS 0.) How many times does the kth smallest element so far change?

[1]There is an algorithm that uses $n + \Theta(k \log(n/k))$ comparisons, but this is even harder.

CS 473U: Undergraduate Algorithms, Fall 2006
Homework 4

Due Tuesday, October 10, 2006 in 3229 Siebel Center

Remember to submit **separate, individually stapled** solutions to each of the problems.

---

1. Chicago has many tall buildings, but only some of them have a clear view of Lake Michigan. Suppose we are given an array $A[1..n]$ that stores the height of n buildings on a city block, indexed from west to east. Building i has a good view of Lake Michigan if every building to the east of i is shorter than i. We present an algorithm that computes which buildings have a good view of Lake Michigan. Use the taxation method of amortized analysis to bound the amortized time spent in each iteration of the for loop. What is the total runtime?

```
GOODVIEW($A[1..n]$):
  Initialize a stack $S$
  for $i = 1$ to $n$
      while ($S$ not empty and $A[i] \geq A[S.\text{top}]$)
          POP($S$)
      PUSH($S, i$)
  return $S$
```

2. Design and analyze a simple data structure that maintains a list of integers and supports the following operations.

   (a) CREATE(): creates and returns a new list L

   (b) PUSH(L, x): appends x to the end of L

   (c) POP(L): deletes the last entry of L and returns it

   (d) LOOKUP(L, k): returns the kth entry of L

   Your solution may use these primitive data structures: arrays, balanced binary search trees, heaps, queues, single or doubly linked lists, and stacks. If your algorithm uses anything fancier, you must give an explicit implementation. Your data structure should support all operations in amortized constant time. In addition, your data structure should support LOOKUP() in worst-case $O(1)$ time. At all times, your data structure should use space which is linear in the number of objects it stores.

3. Consider a computer game in which players must navigate through a field of landmines, which are represented as points in the plane. The computer creates new landmines which the players must avoid. A player may ask the computer how many landmines are contained in any simple polygonal region; it is your job to design an algorithm which answers these questions efficiently.

   You have access to an efficient static data structure which supports the following operations.

- CREATES($\{p_1, p_2, \ldots, p_n\}$): creates a new data structure S containing the points $\{p_1, \ldots, p_n\}$. It has a worst-case running time of $T(n)$. Assume that $T(n)/n \geq T(n-1)/(n-1)$, so that the average processing time of elements does not decrease as n grows.
- DUMPS(S): destroys S and returns the set of points that S stored. It has a worst-case running time of $O(n)$, where n is the number of points in S.
- QUERYS(S, R): returns the number of points in S that are contained in the region R. It has a worst-case running time of $Q(n)$, where n is the number of points stored in S.

Unfortunately, the data structure does not support point insertion, which is required in your application. Using the given static data structure, design and analyze a dynamic data structure that supports the following operations.

(a) CREATED(): creates a new data structure D containing no points. It should have a worst-case constant running time.

(b) INSERTD(D, p): inserts p into D. It should run in amortized $O(\log n) \cdot T(n)/n$ time.

(c) QUERYD(D, R): returns the number of points in D that are contained in the region R. It should have a worst-case running time of $O(\log n) \cdot Q(n)$.

CS 473U: Undergraduate Algorithms, Fall 2006
Homework 5

Due Tuesday, October 24, 2006 in 3229 Siebel Center
Remember to turn in in separate, individually stapled solutions to each of the problems.

---

1. Makefiles:
   In order to facilitate recompiling programs from multiple source files when only a small number of files have been updated, there is a UNIX utility called 'make' that only recompiles those files that were changed after the most recent compilation, *and* any intermediate files in the compilation that depend on those that were changed. A Makefile is typically composed of a list of source files that must be compiled. Each of these source files is dependent on some of the other files that must be compiled. Thus a source file must be recompiled if a file on which it depends is changed.

   Assuming you have a list of which files have been recently changed, as well as a list for each source file of the files on which it depends, design and analyze an efficient algorithm to recompile only the necessary files. DO NOT worry about the details of parsing a Makefile.

2. Consider a graph G, with n vertices. Show that if any two of the following properties hold for G, then the third property must also hold.

   - G is connected.
   - G is acyclic.
   - G has $n-1$ edges.

3. The weight of a spanning tree is the sum of the weights on the edges of the tree. Given a graph, G, describe an efficient algorithm (the most efficient one you can) to find the k lightest (with least weight) spanning trees of G.

   Analyze the running time of your algorithm. Be sure to prove your algorithm is correct.

CS 473U: Undergraduate Algorithms, Fall 2006
Homework 6

Due Wednesday, November 8, 2006 in 3229 Siebel Center

Remember to turn in separate, individually stapled solutions to each of the problems.

---

1. Dijkstra's algorithm can be used to determine shortest paths on graphs with some negative edge weights (as long as there are no negative cycles), but the worst-case running time is much worse than the $O(E+V \log V)$ it takes when the edge weights are all positive. Construct an infinite family of graphs - with negative edge weights - for which the asymptotic running time of Dijkstra's algorithm is $\Omega(2^{|V|})$.

2. It's a cold and rainy night, and you have to get home from Siebel Center. Your car has broken down, and it's too windy to walk, which means you have to take a bus. To make matters worse, there is no bus that goes directly from Siebel Center to your apartment, so you have to change buses some number of times on your way home. Since it's cold outside, you want to spend as little time as possible waiting in bus shelters.

   From a computer in Siebel Center, you can access an online copy of the MTD bus schedule, which lists bus routes and the arrival time of every bus at each stop on its route. Describe an algorithm which, given the schedule, finds a way for you to get home that minimizes the time you spend at bus shelters (the amount of time you spend on the bus doesn't matter). Since Siebel Center is warm and the nearest bus stop is right outside, you can assume that you wait inside Siebel until the first bus you want to take arrives outside. Analyze the efficiency of your algorithm and prove that it is correct.

3. The Floyd-Warshall all-pairs shortest path algorithm computes, for each $u, v \in V$, the shortest path from u to v. However, if the graph has negative cycles, the algorithm fails. Describe a modified version of the algorithm (*with the same asymptotic time complexity*) that correctly returns shortest-path distances, even if the graph contains negative cycles. That is, if there is a path from u to some negative cycle, and a path from that cycle to v, the algorithm should output $dist(u, v) = -\infty$. For any other pair u, v, the algorithm should output the length of the shortest directed path from u to v.

CS 473U: Undergraduate Algorithms, Fall 2006
Homework 6

Due at **4 p.m. on** Friday, November 17, 2006 in 3229 Siebel Center

---

Remember to turn in separate, individually stapled solutions to each of the problems.

---

1. Given an undirected graph $G(V, E)$, with three vertices $u, v, w \in V$, you want to know whether there exists a path from u to w via v. (That is, the path from u to w must use v as an intermediate vertex.) Describe an efficient algorithm to solve this problem.

2. *Ad-hoc Networks*, made up of cheap, low-powered wireless devices, are often used on battlefields, in regions that have recently suffered from natural disasters, and in other situtations where people might want to monitor conditions in hard-to-reach areas. The idea is that a large collection of the wireless devices could be dropped into the area from an airplane (for instance), and then they could be configured into an efficiently functioning network.

   Since the devices are cheap and low-powered, they frequently fail, and we would like our networks to be reliable. If a device detects that it is likely to fail, it should transmit the information it has to some other device (called a *backup*) within range of it. The range is limited; we assume that there is a distance d such that two devices can communicate if and only if they are within distance d of each other. To improve reliability, we don't want a device to transmit information to a neighbor that has already failed, and so we require each device v to have at least k backup devices that it could potentially contact, all of which must be within d meters of it. We call this the *backup set* of v. Also, we do not want any device to be in the backup set of too many other devices; if it were, and it failed, a large fraction of our network would be affected.

   The input to our problem is a collection of n devices, and for each pair u, v of devices, the distance between u and v. We are also given the distance d that determines the range of a device, and parameters b and k. Describe an algorithm that determines if, for each device, we can find a backup set of size k, while also requiring that no device appears in the backup set of more than b other devices.

3. **UPDATED:** Given a piece of text T and a pattern P (the 'search string'), an algorithm for the string-matching problem either finds the first occurrence of P in T, or reports that there is none. Modify the Knuth-Morris-Pratt (KMP) algorithm so that it solves the string-matching problem, even if the pattern contains the wildcards '?' and '*'. Here, '?' represents any *single* character of the text, and '*' represents any substring of the text (including the empty substring). For example, the pattern "A?B*?A" matches the text "ABACBCABBCCACBA" starting in position 3 (in three different ways), and position 7 (in two ways). For this input, your algorithm would need to return '3'.

   UPDATE: You may assume that the pattern you are trying to match containst at most 3 blocks of question marks; the usage of '*' wildcards is stll unrestricted. Here, a block refers to a string of consecutive '?'s in the pattern. For example, AAB??ACA???????BB contains 2 blocks of question marks; A?B?C?A?C contains 4 blocks of question marks.

4. In the two-dimensional pattern-matching problem, you are given an $m \times n$ matrix M and a $p \times q$ pattern P. You wish to find all positions (i, j) in M such that the the submatrix of M between rows i and $i + p - 1$ and between columns j and $j + q - 1$ is identical to P. (That is, the $p \times q$ sub-matrix of M below and to the right of position (i, j) should be identical to P.) Describe and analyze an efficient algorithm to solve this problem.[1]

[1]Note that the normal string-matching problem is the special case of the 2-dimensional problem where $m = p = 1$.

CS 473U: Undergraduate Algorithms, Fall 2006
Homework 8

Due Wednesday, December 6, 2006 in 3229 Siebel Center

Remember to submit **separate, individually stapled** solutions to each of the problems.

---

1. Given an array $A[1..n]$ of $n \geq 2$ distinct integers, we wish to find the second largest element using as few comparisons as possible.

   (a) Give an algorithm which finds the second largest element and uses at most $n + \lceil \lg n \rceil - 2$ comparisons in the worst case.

   *(b) Prove that every algorithm which finds the second largest element uses at least $n + \lceil \lg n \rceil - 2$ comparisons in the worst case.

2. Let R be a set of rectangles in the plane. For each point p in the plane, we say that the *rectangle depth* of p is the number of rectangles in R that contain p.

   (a) (Step 1: Algorithm Design) Design and analyze a polynomial-time algorithm which, given R, computes the maximum rectangle depth.

   (b) (Step 2: ???) Describe and analyze a polynomial-time reduction from the maximum rectangle depth problem to the maximum clique problem.

   (c) (Step 3: Profit!) In 2000, the Clay Mathematics Institute described the Millennium Problems: seven challenging open problems which are central to ongoing mathematical research. The Clay Institute established seven prizes, each worth one million dollars, to be awarded to anyone who solves a Millennium problem. One of these problems is the P = NP question. In (a), we developed a polynomial-time algorithm for the maximum rectangle depth problem. In (b), we found a reduction from this problem to an NP-complete problem. We know from class that if we find a polynomial-time algorithm for any NP-complete problem, then we have shown P = NP. Why hasn't Jeff used (a) and (b) to show P = NP and become a millionaire?

3. Let G be a complete graph with integer edge weights. If C is a cycle in G, we say that the *cost* of C is the sum of the weights of edges in C. Given G, the traveling salesman problem (TSP) asks us to compute a Hamiltonian cycle of minimum cost. Given G, the traveling salesman cost problem (TSCP) asks us to compute the cost of a minimum cost Hamiltonian cycle. Given G and an integer k, the traveling salesman decision problem (TSDP) asks us to decide if there is a Hamiltonian cycle in G of cost at most k.

   (a) Describe and analyze a polynomial-time reduction from TSP to TSCP.

   (b) Describe and analyze a polynomial-time reduction from TSCP to TSDP.

   (c) Describe and analyze a polynomial-time reduction from TSDP to TSP.

(d) What can you conclude about the relative computational difficulty of TSP, TSCP, and TSDP?

4. Let G be a graph. A set S of vertices of G is a *dominating set* if every vertex in G is either in S or adjacent to a vertex in S. Show that, given G and an integer k, deciding if G contains a dominating set of size at most k is NP-complete.

1. Probability
   (a) n people have checked their hats with a hat clerk. The clerk is somewhat absent-minded and returns the hats uniformly at random (with no regard for whether each hat is returned to its owner). On average, how many people will get back their own hats?
   (b) Let S be a uniformly random permutation of $\{1, 2, \ldots, n-1, n\}$. As we move from the left to the right of the permutation, let X denote the smallest number seen so far. On average, how many different values will X take?

2. A *tournament* is a directed graph where each pair of distinct vertices u, v has either the edge uv or the edge vu (but not both). A *Hamiltonian path* is a (directed) path that visits each vertex of the (di)graph. Prove that every tournament has a Hamiltonian path.

3. Describe and analyze a data structure that stores a set of n records, each with a numerical *key*, such that the following operation can be performed quickly:

   `Foo`(a): return the sum of the records with keys at least as large as a.

   For example, if the keys are:

   $$3\ 4\ 9\ 6\ 5\ 8\ 7\ 1\ 0$$

   then `Foo`(2) would return 42, since $3, 4, 5, 6, 7, 8, 9$ are all larger than 2 and $3+4+5+6+7+8+9 = 42$.

   You may assume that no two records have equal keys, and that no record has a key equal to a. Analyze both the size of your data structure and the running time of your `Foo` algorithm. Your data structure must be as small as possible and your `Foo` algorithm must be as fast as possible.

1. The Acme Company is planning a company party. In planning the party, each employee is assigned a *fun value* (a positive real number). The goal of the party planners is to maximize the total fun value (sum of the individual fun values) of the employees invited to the party. However, the planners are not allowed to invite both an employee and his direct boss. Given a tree containing the boss/underling structure of Acme, find the invitation list with the highest allowable fun value.

2. An *inversion* in an array A is a pair i, j such that $i < j$ and $A[i] > A[j]$. (In an n-element array, the number of inversions is between 0 and $\binom{n}{2}$.)

   Find an efficient algorithm to count the number of inversions in an n-element array.

3. A *tromino* is a geometric shape made from three squares joined along complete edges. There are only two possible trominoes: the three component squares may be joined in a line or an L-shape.

   (a) Show that it is possible to cover all but one square of a 64 x 64 checkerboard using L-shape trominoes. (In your covering, each tromino should cover three squares and no square should be covered more than once.)

   (b) Show that you can leave *any* single square uncovered.

   (c) Can you cover all but one square of a 64 x 64 checkerboard using *line* trominoes? If so, which squares can you leave uncovered?

1. **Moving on a Checkerboard**

   Suppose that you are given an $n \times n$ checkerboard and a checker. You must move the checker from the bottom edge of the board to the top edge of the board according to the following rule. At each step you may move the checker to one of three squares:

   1) the square immediately above
   2) the square that is one up and one to the left (but only if the checker is not already in the leftmost column)
   3) the square that is one up and one to the right (but only if the checker is not already in the rightmost column)

   Each time you move from square x to square y, you receive $p(x, y)$ dollars. You are given a list of the values $p(x, y)$ for each pair (x, y) for which a move from x to y is legal. Do not assume that $p(x, y)$ is positive.

   Give an algorithm that figures out the set of moves that will move the checker from somewhere along the bottom edge to somewhere along the top edge while gathering as many dollars as possible. You algorithm is free to pick any square along the bottom edge as a starting point and any square along the top edge as a destination in order to maximize the number of dollars gathered along the way. What is the running time of your algorithm?

2. **Maximizing Profit**

   You are given lists of values $h_1, h_2, \ldots, h_k$ and $l_1, l_2, \ldots, l_k$. For each i you can choose $j_i = h_i$, $j_i = l_i$, or $j_i = 0$; the only catch is that if $j_i = h_i$ then j_{i-1} must be 0 (except for $i = 1$). Your goal is to maximize $\sum_{i=1}^{k} j_i$.

   Give an efficient algorithm that returns the maximum possible value of $\sum_{i=1}^{k} j_i$.

3. **Maximum alternating subsequence**

   An *alternating sequence* is a sequence $a_1, a_2, \ldots$ such that no three consecutive terms of the sequence satisfy $a_i > a_{i+1} > a_{i+2}$ or $a_i < a_{i+1} < a_{i+2}$.

   Given a sequence, efficiently find the longest alternating subsequence it contains. What is the running time of your algorithm?

1. **Championship Showdown**

   What excitement! The Champaign Spinners and the Urbana Dreamweavers have advanced to meet each other in the World Series of Basketweaving! The World Champions will be decided by a best of $2n-1$ series of head-to-head weaving matches, and the first to win n matches will take home the coveted Golden Basket (for example, a best-of-7 series requires four match wins, but we will keep the generalized case). We know that for any given match there is a constant probability p that Champaign will win, and a subsequent probability $q = 1-p$ that Urbana will win.

   Let $P(i,j)$ be the probability that Champaign will win the series given that they still need i more victories, whereas Urbana needs j more victories for the championship. $P(0,j) = 1$, $1 \le j \le n$, because Champaign needs no more victories to win. $P(i,0) = 0$, $1 \le i \le n$, as Champaign cannot possibly win if Urbana already has. $P(0,0)$ is meaningless. Champaign wins any particular match with probability p and loses with probability q, so

   $$P(i,j) = p \cdot P(i-1,j) + q \cdot P(i,j-1)$$

   for any $i \ge 1$ and $j \ge 1$.

   Create and analyze an $O(n^2)$-time dynamic programming algorithm that takes the parameters n, p, and q and returns the probability that Champaign will win the series (that is, calculate $P(n,n)$).

2. **Making Change**

   Suppose you are a simple shopkeeper living in a country with n different types of coins, with values $1 = c[1] < c[2] < \cdots < c[n]$. (In the U.S., for example, $n = 6$ and the values are $1, 5, 10, 25, 50$, and 100 cents.) Your beloved benevolent dictator, El Generalissimo, has decreed that whenever you give a customer change, you must use the smallest possible number of coins, so as not to wear out the image of El Generalissimo lovingly engraved on each coin by servants of the Royal Treasury.

   Describe and analyze a dynamic programming algorithm to determine, given a target amount A and a sorted array $c[1..n]$ of coin values, the smallest number of coins needed to make A cents in change. You can assume that $c[1] = 1$, so that it is possible to make change for any amount A.

3. **Knapsack**

   You are a thief, who is trying to choose the best collection of treasure (some subset of the n treasures, numbered 1 through n) to steal. The weight of item i is $w_i \in \mathbb{N}$ and the profit is $p_i \in \mathbb{R}$. Let $C \in \mathbb{N}$ be the maximum weight that your knapsack can hold. Your goal is to choose a subset of elements $S \subseteq \{1, 2, \ldots, n\}$ that maximizes your total profit $P(S) = \sum_{i \in S} p_i$, subject to the constraint that the sum of the weights $W(S) = \sum_{i \in S} w_i$ is not more than C.

   Give an algorithm that runs in time $O(Cn)$.

1. **Randomized Edge Cuts**

   We will randomly partition the vertex set of a graph G into two sets S and T. The algorithm is to flip a coin for each vertex and with probability 1/2, put it in S; otherwise put it in T.

   (a) Show that the expected number of edges with one endpoint in S and the other endpoint in T is exactly half the edges in G.

   (b) Now say the edges have weights. What can you say about the sum of the weights of the edges with one endpoint in S and the other endpoint in T?

2. **Skip Lists**

   A *skip list* is built in layers. The bottom layer is an ordinary sorted linked list. Each higher layer acts as an "express lane" for the lists below, where an element in layer i appears in layer $i + 1$ with some fixed probability p.

```
1
1-----4---6
1---3-4---6-----9
1-2-3-4-5-6-7-8-9-10
```

   (a) What is the probability a node reaches height h.

   (b) What is the probability any node is above $c \log n$ (for some fixed value of c)?
   Compute the value explicitly when $p = 1/2$ and $c = 4$.

   (c) To search for an entry x, scan the top layer until you find the last entry y that is less than or equal to x. If $y < x$, drop down one layer and in this new layer (beginning at y) find the last entry that is less than or equal to x. Repeat this process (dropping down a layer, then finding the last entry less than or equal to x) until you either find x or reach the bottom layer and confirm that x is not in the skip list. What is the expected search time?

   (d) Describe an efficient method for insertion. What is the expected insertion time?

3. **Clock Solitaire**

   In a standard deck of 52 cards, put 4 face-down in each of the 12 'hour' positions around a clock, and 4 face-down in a pile in the center. Turn up a card from the center, and look at the number on it. If it's number x, place the card face-up next to the face-down pile for x, and turn up the next card in the face-down pile for x (that is, the face-down pile corresponding to hour x). You win if, for each Ace $\leq x \leq$ Queen, all four cards of value x are turned face-up before all four Kings (the center cards) are turned face-up.

   What is the probability that you win a game of Clock Solitaire?

1. **Simulating Queues with Stacks**

   A *queue* is a first-in-first-out data structure. It supports two operations *push* and *pop*. Push adds a new item to the back of the queue, while pop removes the first item from the front of the queue. A *stack* is a last-in-first-out data structure. It also supports push and pop. As with a queue, push adds a new item to the back of the queue. However, pop removes the last item from the back of the queue (the one most recently added).

   Show how you can simulate a queue by using two stacks. Any sequence of pushes and pops should run in amortized constant time.

2. **Multistacks**

   A *multistack* consists of an infinite series of stacks $S_0, S_1, S_2, \ldots$, where the ith stack S_i can hold up to 3^i elements. Whenever a user attempts to push an element onto any full stack S_i, we first move all the elements in S_i to stack S_{i+1} to make room. But if S_{i+1} is already full, we first move all its members to S_{i+2}, and so on. To clarify, a user can only push elements onto S_0. All other pushes and pops happen in order to make space to push onto S_0. Moving a single element from one stack to the next takes $O(1)$ time.

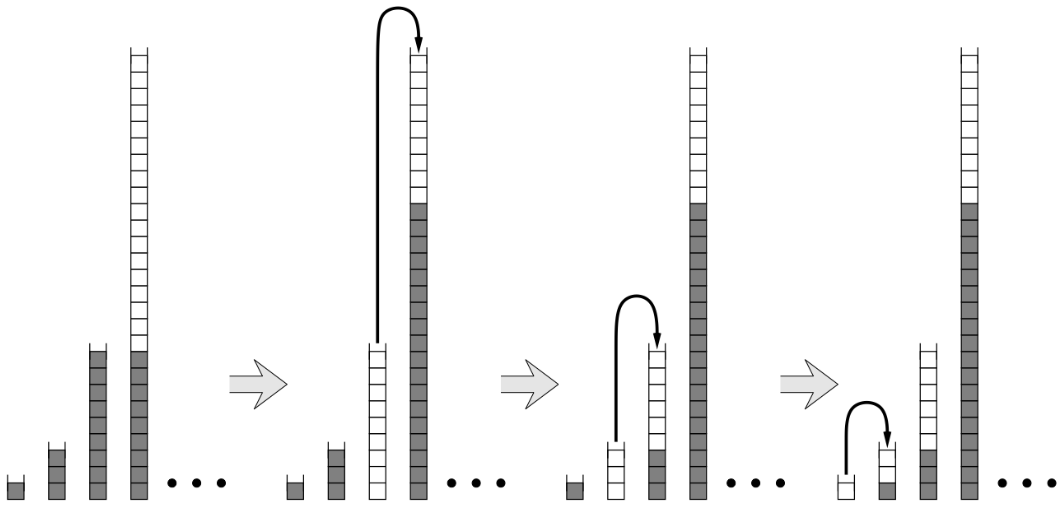


Figure 1. Making room for one new element in a multistack.

   (a) In the worst case, how long does it take to push one more element onto a multistack containing n elements?

   (b) Prove that the amortized cost of a push operation is $O(\log n)$, where n is the maximum number of elements in the multistack.

3. **Powerhungry function costs**

   A sequence of n operations is performed on a data structure. The ith operation costs i if i is an exact power of 2, and 1 otherwise. Determine the amortized cost of the operation.

1. **Representation of Integers**

   (a) Prove that any positive integer can be written as the sum of distinct *nonconsecutive* Fibonacci numbers–if F_n appears in the sum, then neither F_{n+1} nor F_{n-1} will. For example: $42 = F_9 + F_6$, $25 = F_8 + F_4 + F_2$, $17 = F_7 + F_4 + F_2$.

   (b) Prove that any integer (positive, negative, or zero) can be written in the form $\sum_i \pm 3^i$, where the exponents i are distinct non-negative integers. For example $42 = 3^4 - 3^3 - 3^2 - 3^1$, $25 = 3^3 - 3^1 + 3^0$, $17 = 3^3 - 3^2 - 3^0$.

2. **Minimal Dominating Set**

   Suppose you are given a rooted tree T (not necessarily binary). You want to label each node in T with an integer 0 or 1, such that every node either has the label 1 or is adjacent to a node with the label 1 (or both). The *cost* of a labeling is the number of nodes with label 1. Describe and analyze an algorithm to compute the minimum cost of any labeling of the given tree T.

3. **Names in Boxes**

   The names of 100 prisoners are placed in 100 wooden boxes, one name to a box, and the boxes are lined up on a table in a room. One by one, the prisoners are led into the room; each may look in at most 50 boxes, but must leave the room exactly as he found it and is permitted no further communication with the others.

   The prisoners have a chance to plot their strategy in advance, and they are going to need it, because unless *every single prisoner finds his own name* all will subsequently be executed. Find a strategy for them which has probability of success exceeding 30%. You may assume that the names are distributed in the boxes uniformly at random.

   (a) Calculate the probability of success if each prisoner picks 50 boxes uniformly at random.

   *(b) Consider the following strategy.

   The prisoners number themselves 1 to 100. Prisoner i begins by looking in box i. There he finds the name of prisoner j. If $j \neq i$, he continues by looking in box j. As long as prisoner i has not found his name, he continues by looking in the box corresponding to the last name he found.

   Describe the set of permutations of names in boxes for which this strategy will succeed.

   *(c) Count the number of permutations for which the strategy above succeeds. Use this sum to calculate the probability of success. You may find it useful to do this calculation for general n, then set $n = 100$ at the end.

   (d) We assumed that the names were distributed in the boxes uniformly at random. Explain how the prisoners could augment their strategy to make this assumption unnecessary.

1. **Dynamic MSTs**

   Suppose that you already have a minimum spanning tree (MST) in a graph. Now one of the edge weights changes. Give an efficient algorithm to find an MST in the new graph.

2. **Minimum Bottleneck Trees**

   In a graph G, for any pair of vertices u, v, let bottleneck(u, v) be the maximum over all paths p_i from u to v of the minimum-weight edge along p_i. Construct a spanning tree T of G such that for each pair of vertices, their bottleneck in G is the same as their bottleneck in T.

   One way to think about it is to imagine the vertices of the graph as islands, and the edges as bridges. Each bridge has a maximum weight it can support. If a truck is carrying stuff from u to v, how much can the truck carry? We don't care what route the truck takes; the point is that the smallest-weight edge on the route will determine the load.

3. **Eulerian Tours**

   An *Eulerian tour* is a "walk along edges of a graph" (in which successive edges must have a common endpoint) that uses each edge exactly once and ends at the vertex where it starts. A graph is called Eulerian if it has an Eulerian tour.

   Prove that a connected graph is Eulerian iff each vertex has even degree.

1. **Alien Abduction**

   Mulder and Scully have computed, for every road in the United States, the exact probability that someone driving on that road won't be abducted by aliens. Agent Mulder needs to drive from Langley, Virginia to Area 51, Nevada. What route should he take so that he has the least chance of being abducted?

   More formally, you are given a directed graph $G = (V, E)$, where every edge e has an independent safety probability $p(e)$. The *safety* of a path is the product of the safety probabilities of its edges. Design and analyze an algorithm to determine the safest path from a given start vertex s to a given target vertex t.

2. **The Only SSSP Algorithm**

   In the lecture notes, Jeff mentions that all SSSP algorithms are special cases of the following generic SSSP algorithm. Each vertex v in the graph stores two values, which describe a tentative shortest path from s to v.

   - $\text{dist}(v)$ is the length of the tentative shortest $s \rightsquigarrow v$ path.
   - $\text{pred}(v)$ is the predecessor of v in the shortest $s \rightsquigarrow v$ path.

   We call an edge *tense* if $\text{dist}(u) + w(u \to v) < \text{dist}(v)$. Our generic algorithm repeatedly finds a tense edge in the graph and *relaxes* it:

   ```
   Relax($u \to v$):
     $\text{dist}(v) \leftarrow \text{dist}(u) + w(u \to v)$
     $\text{pred}(v) \leftarrow u$
   ```

   If there are no tense edges, our algorithm is finished, and we have our desired shortest path tree. The correctness of the relaxation algorithm follows directly from three simple claims. The first of these is below. Prove it.

   - When the algorithm halts, if $\text{dist}(v) \neq \infty$, then $\text{dist}(v)$ is the total weight of the predecessor chain ending at v:
   $$s \to \cdots \to (pred(pred(v)) \to pred(v) \to v.$$

3. **Can't find a Cut-edge**

   A cut-edge is an edge which when deleted disconnects the graph. Prove or disprove the following. Every 3-regular graph has no cut-edge. (A common approach is induction.)

1. **Max-Flow with vertex capacities**

   In a standard $s-t$ Maximum-Flow Problem, we assume edges have capacities, and there is no limit on how much flow is allowed to pass through a node. In this problem, we consider the variant of Maximum-Flow and Minimum-Cut problems with node capacities.

   More specifically, each node, n_i, has a capacity c_i. The edges have unlimited capacity. Show how you can model this problem as a standard Max-flow problem (where the weights are on the edges).

2. **Emergency evacuation**

   Due to large-scale flooding in a region, paramedics have identified a set of n injured people distributed across the region who need to be reushed to hospitals. There are k hospitals in the region, and each of the n people needs to be brought to a hospital that is within a half-hour's driving time of their current location.

   At the same time, we don't want to overload any hospital by sending too many patients its way. We'd like to distribute the people so that each hospital receives at most $\lceil n/k \rceil$ people.

   Show how to model this problem as a Max-flow problem.

3. **Tracking a Hacker**

   A computer network (with each edge weight 1) is designed to carry traffic from a source s to a destination t. Recently, a computer hacker destroyed some of the edges in the graph. Normally, the maximum $s-t$ flow in G is k. Unfortunately, there is currently no path from s to t. Fortunately, the sysadmins know that the hacker destroyed at most k edges of the graph.

   The sysadmins are trying to diagnose which of the nodes of the graph are no longer reachable. They would like to avoid testing each node. They are using a monitoring tool with the following behavior. If you use the command *ping(v)*, for a given node v, it will tell you whether there is currently a path from s to v (so *ping(t)* will return `False` but *ping(s)* will return `True`).

   Give an algorithm that accomplishes this task using only $O(k \log n)$ pings. (You may assume that any algorithm you wish to run on the original network (before the hacker destroyed edges) runs for free, since you have a model of that network on your computer.)

1. **Updating a maximum flow**

   Suppose you are given a directed graph $G = (V, E)$, with a positive integer capacity c_e on each edge e, a designated source $s \in V$, and a designated sink $t \in V$. You are also given a maximum $s - t$ flow in G, defined by a flow value f_e on each edge e. The flow $\{f_e\}$ is *acyclic*: There is no cycle in G on which all edges carry positive flow.

   Now suppose we pick a specific edge $e^* \in E$ and reduce its capacity by 1 unit. Show how to find a maximum flow in the resulting capacitated graph in time $O(m + n)$, where m is the number of edges in G and n is the number of nodes.

2. **Cooking Schedule**

   You live in a cooperative apartment with n other people. The co-op needs to schedule cooks for the next n days, so that each person cooks one day and each day there is one cook. In addition, each member of the co-op has a list of days they are available to cook (and is unavailable to cook on the other days).

   Because of your superior CS473 skills, the co-op selects you to come up with a schedule for cooking, so that everyone cooks on a day they are available.

   (a) Describe a bipartite graph G so that G has a perfect matching if and only if there is a feasible schedule for the co-op.

   (b) A friend of yours tried to help you out by coming up with a cooking schedule. Unfortunately, when you look at the schedule he created, you notice a big problem. $n - 2$ of the people are scheduled for different nights on which they are available: no problem there. But the remaining two people are assigned to cook on the same night (and no one is assigned to the last night).

   You want to fix your friend's mistake, but without having to recompute everything from scratch. Show that it's possible, using his "almost correct" schedule to decide in $O(n^2)$ time whether there exists a feasible schedule.

3. **Disjoint paths in a digraph**

   Let $G = (V, E)$ be a directed graph, and suppose that for each node v, the number of edges into v is equal to the number of edges out of v. That is, for all v,

$$|\{(u, v) : (u, v) \in E\}| = |\{(v, w) : (v, w) \in E\}|.$$

   Let x, y be two nodes of G, and suppose that there exist k mutually edge-disjoint paths from x to y. Under these conditions, does it follow that there exist k mutually edge-disjoint paths from y to x. Give a proof or a counterexample with explanation.

1. **String matching: an example**

   (a) Build a finite automata to search for the string "bababoon".

   (b) Use the automata from part (a) to build the prefix function for Knuth-Morris-Pratt.

   (c) Use the automata or the prefix function to search for "bababoon" in the string "babybaboon-buysbananasforotherbabybababoons".

2. **Cooking Schedule Strikes Back**

   You live in a cooperative apartment with n other people. The co-op needs to schedule cooks for the next $5n$ days, so that each person cooks five days and each day there is one cook. In addition, each member of the co-op has a list of days they are available to cook (and is unavailable to cook on the other days).

   Because of your success at headbanging last week, the co-op again asks you to compose a cooking schedule. Unfortunately, you realize that no such schedule is possible Give a schedule for the cooking so that no one has to cook on more than 2 days that they claim to be unavailable.

3. **String matching on Trees**

   You are given a rooted tree T (not necessarily binary), in which each node has a character. You are also given a pattern $P = p_1 p_2 \cdots p_l$. Search for the string as a subtree. In other words, search for a subtree in which p_i is on a child of the node containing p_{i-1} for each $2 \leq i \leq l$.

1. **Self-reductions**

   In each case below assume that you are given a black box which can answer the decision version of the indicated problem. Use a polynomial number of calls to the black box to construct the desired set.

   (a) Independent set: Given a graph G and an integer k, does G have a subset of k vertices that are pairwise nonadjacent?

   (b) Subset sum: Given a multiset (elements can appear more than once) $X = \{x_1, x_2, \ldots, x_k\}$ of positive integers, and a positive integer S does there exist a subset of X with sum exactly S?

2. **Lower Bounds**

   Give adversary arguments to prove the indicated lower bounds for the following problems:

   (a) Searching in a sorted array takes at least $1 + \lfloor \lg_2 n \rfloor$ queries.

   (b) Let M be an $n \times n$ array of real values that is increasing in both rows and columns. Prove that searching for a value requires at least n queries.

3. ***k*-coloring**

   Show that we can solve the problem of constructing a k-coloring of a graph by using a polynomial number of calls to a black box that determines whether a graph has such a k-coloring. (Hint: Try reducing via an intermediate problem that asks whether a partial coloring of a graph can be extended to a proper k-coloring.)

1. **NP-hardness Proofs: Restriction**
   Prove that each of the following problems is NP-hard. In each part, find a special case of the given problem that is equivalent to a known NP-hard problem.
   (a) **Longest Path**
   Given a graph G and a positive integer k, does G contain a path with k or more edges?
   (b) **Partition into Hamiltonian Subgraphs**
   Given a graph G and a positive integer k, can the vertices of G be partitioned into at most k disjoint sets such that the graph induced by each set has a Hamiltonian cycle?
   (c) **Set Packing**
   Given a collection of finite sets C and a positive integer k, does C contain k disjoint sets?
   (d) **Largest Common Subgraph**
   Given two graphs G_1 and G_2 and a positive integer k, does there exist a graph G_3 such that G_3 is a subgraph of both G_1 and G_2 and G_3 has at least k edges?

2. **Domino Line**
   You are given an unusual set of dominoes; each domino has a number on each end, but the numbers may be arbitarily large and some numbers appear on many dominoes, while other numbers only appear on a few dominoes. Your goal is to form a line using all the dominoes so that adjacent dominoes have the same number on their adjacent halves. Either give an efficient algorithm to solve the problem or show that it is NP-hard.

3. **Set Splitting**
   Given a finite set S and a collection of subsets C is there a partition of S into two sets S_1 and S_2 such that no subset in C is contained entirely in S_1 or S_2? Show that the problem is NP-hard. (Hint: use NAE-3SAT, which is similar to 3SAT except that a satisfying assingment does not allow all 3 variables in a clause to be true.)

You have 120 minutes to answer four of these five questions.

Write your answers in the separate answer booklet.

1. **Multiple Choice.**

Each of the questions on this page has one of the following five answers:

A: $\Theta(1)$ B: $\Theta(\log n)$ C: $\Theta(n)$ D: $\Theta(n \log n)$ E: $\Theta(n^2)$

Choose the correct answer for each question. Each correct answer is worth $+1$ point; each incorrect answer is worth $-\frac{1}{2}$ point; each "I don't know" is worth $+\frac{1}{4}$ point. Your score will be rounded to the nearest *non-negative* integer. You do *not* need to justify your answers; just write the correct letter in the box.

(a) What is $\dfrac{5}{n} + \dfrac{n}{5}$?

(b) What is $\displaystyle\sum_{i=1}^{n} \frac{n}{i}$?

(c) What is $\displaystyle\sum_{i=1}^{n} \frac{i}{n}$?

(d) How many bits are required to represent the nth Fibonacci number in binary?

(e) What is the solution to the recurrence $T(n) = 2T(n/4) + \Theta(n)$?

(f) What is the solution to the recurrence $T(n) = 16T(n/4) + \Theta(n)$?

(g) What is the solution to the recurrence $T(n) = T(n-1) + 1/n^2$?

(h) What is the worst-case time to search for an item in a binary search tree?

(i) What is the worst-case running time of quicksort?

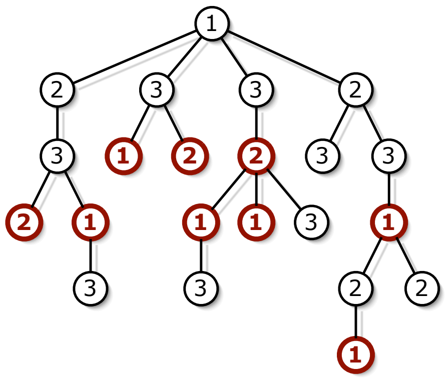
(j) What is the running time of the fastest possible algorithm to solve Sudoku puzzles? A Sudoku puzzle consists of a 9×9 grid of squares, partitioned into nine 3×3 sub-grids; some of the squares contain digits between 1 and 9. The goal of the puzzle is to enter digits into the blank squares, so that each digit between 1 and 9 appears exactly once in each row, each column, and each 3×3 sub-grid. The initial conditions guarantee that the solution is unique.

2							4	
	7		5					
				1		9		
6		4			2			
	8						5	
			9			3		7
		1		4				
					3		8	
	5							6

A Sudoku puzzle. **Don't try to solve this during the exam!**

2. Oh, no! You have been appointed as the gift czar for Giggle, Inc.'s annual mandatory holiday party! The president of the company, who is certifiably insane, has declared that *every* Giggle employee must receive one of three gifts: (1) an all-expenses-paid six-week vacation anywhere in the world, (2) an all-the-pancakes-you-can-eat breakfast for two at Jumping Jack Flash's Flapjack Stack Shack, or (3) a burning paper bag full of dog poop. Corporate regulations prohibit any employee from receiving the same gift as his/her direct supervisor. Any employee who receives a better gift than his/her direct supervisor will almost certainly be fired in a fit of jealousy. How do you decide what gifts everyone gets if you want to minimize the number of people that get fired?

   More formally, suppose you are given a rooted tree T, representing the company hierarchy. You want to label each node in T with an integer 1, 2, or 3, such that every node has a different label from its parent.. The *cost* of an labeling is the number of nodes that have smaller labels than their parents. Describe and analyze an algorithm to compute the minimum cost of any labeling of the given tree T. (Your algorithm does *not* have to compute the actual best labeling—just its cost.)



A tree labeling with cost 9. Bold nodes have smaller labels than their parents.
This is *not* the optimal labeling for this tree.

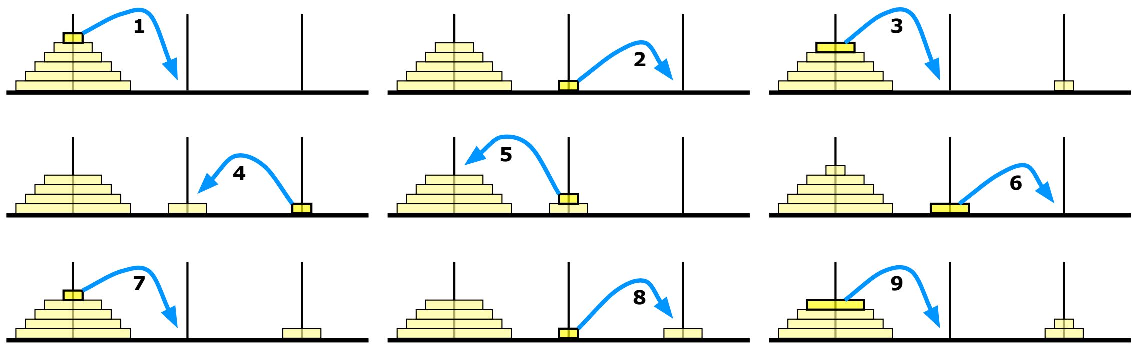
3. Suppose you are given an array $A[1..n]$ of n distinct integers, sorted in increasing order. Describe and analyze an algorithm to determine whether there is an index i such that $A[i] = i$, in $o(n)$ time. *[Hint: Yes, that's **little**-oh of n. What can you say about the sequence $A[i] - i$?]*

4. Describe and analyze a *polynomial-time* algorithm to compute the length of the longest common subsequence of two strings $A[1..m]$ and $B[1..n]$. For example, given the strings 'DYNAMIC' and 'PROGRAMMING', your algorithm would return the number 3, because the longest common subsequence of those two strings is 'AMI'. You must give a complete, self-contained solution; don't just refer to HW1.

5. Recall that the Tower of Hanoi puzzle consists of three pegs and n disks of different sizes. Initially, all the disks are on one peg, stacked in order by size, with the largest disk on the bottom and the smallest disk on top. In a single move, you can transfer the highest disk on any peg to a different peg, except that you may never place a larger disk on top of a smaller one. The goal is to move all the disks onto one other peg.

   Now suppose the pegs are arranged in a row, and you are forbidden to transfer a disk directly between the left and right pegs in a single move; every move must involve the middle peg. How many moves suffice to transfer all n disks from the left peg to the right peg under this restriction? **Prove your answer is correct.**

   For full credit, give an *exact* upper bound. A correct upper bound using $O(\cdot)$ notation (with a proof of correctness) is worth 7 points.



The first nine moves in a restricted Towers of Hanoi solution.

1. On an overnight camping trip in Sunnydale National Park, you are woken from a restless sleep by a scream. As you crawl out of your tent to investigate, a terrified park ranger runs out of the woods, covered in blood and clutching a crumpled piece of paper to his chest. As he reaches your tent, he gasps, "Get out… while… you… ", thrusts the paper into your hands, and falls to the ground. Checking his pulse, you discover that the ranger is stone dead.

   You look down at the paper and recognize a map of the park, drawn as an undirected graph, where vertices represent landmarks in the park, and edges represent trails between those landmarks. (Trails start and end at landmarks and do not cross.) You recognize one of the vertices as your current location; several vertices on the boundary of the map are labeled EXIT.

   On closer examination, you notice that someone (perhaps the poor dead park ranger) has written a real number between 0 and 1 next to each vertex and each edge. A scrawled note on the back of the map indicates that a number next to an edge is the probability of encountering a vampire along the corresponding trail, and a number next to a vertex is the probability of encountering a vampire at the corresponding landmark. (Vampires can't stand each other's company, so you'll never see more than one vampire on the same trail or at the same landmark.) The note warns you that stepping off the marked trails will result in a slow and painful death.

   You glance down at the corpse at your feet. Yes, his death certainly looked painful. Wait, was that a twitch? Are his teeth getting longer? After driving a tent stake through the undead ranger's heart, you wisely decide to leave the park immediately.

   Describe and analyze an efficient algorithm to find a path from your current location to an arbitrary EXIT node, such that the total *expected number* of vampires encountered along the path is as small as possible. *Be sure to account for* ***both*** *the vertex probabilities* ***and*** *the edge probabilities!*

2. Consider the following solution for the union-find problem, called *union-by-weight*. Each set leader $\overline{x}$ stores the number of elements of its set in the field *weight*$(\overline{x})$. Whenever we UNION two sets, the leader of the *smaller* set becomes a new child of the leader of the *larger* set (breaking ties arbitrarily).

```
MAKESET(x):
    parent(x) ← x
    weight(x) ← 1
```

```
FIND(x):
    while x ≠ parent(x)
        x ← parent(x)
    return x
```

```
UNION(x, y)
    x̄ ← FIND(x)
    ȳ ← FIND(y)
    if weight(x̄) > weight(ȳ)
        parent(ȳ) ← x̄
        weight(x̄) ← weight(x̄) + weight(ȳ)
    else
        parent(x̄) ← ȳ
        weight(x̄) ← weight(x̄) + weight(ȳ)
```

   Prove that if we use union-by-weight, the *worst-case* running time of FIND is $O(\log n)$.

3. ***Prove or disprove***[1] each of the following statements.

   (a) Let G be an arbitrary undirected graph with arbitrary distinct weights on the edges. The minimum spanning tree of G includes the lightest edge in every cycle in G.

   (b) Let G be an arbitrary undirected graph with arbitrary distinct weights on the edges. The minimum spanning tree of G excludes the heaviest edge in every cycle in G.

4. In Homework 2, you were asked to analyze the following algorithm to find the kth smallest element from an unsorted array. (The algorithm is presented here in iterative form, rather than the recursive form you saw in the homework, but it's exactly the same algorithm.)

```
QUICKSELECT($A[1..n], k$):
  $i \leftarrow 1$; $j \leftarrow n$
  while $i \le j$
      $r \leftarrow$ PARTITION($A[i..j]$, RANDOM($i, j$))
      if $r = k$
          return $A[r]$
      else if $r > k$
          $j \leftarrow r - 1$
      else
          $i \leftarrow r + 1$
```

   The algorithm relies on two subroutines. RANDOM(i, j) returns an integer chosen uniformly at random from the range $[i..j]$. PARTITION$(A[i..j], p)$ partitions the subarray $A[i..j]$ using the pivot value $A[p]$ and returns the new index of the pivot value in the partitioned array.

   What is the *exact* expected number of iterations of the main loop when $k = 1$? ***Prove*** your answer is correct. A correct $\Theta(\cdot)$ bound (with proof) is worth 7 points. You may assume that the input array $A[\,]$ contains n distinct integers.

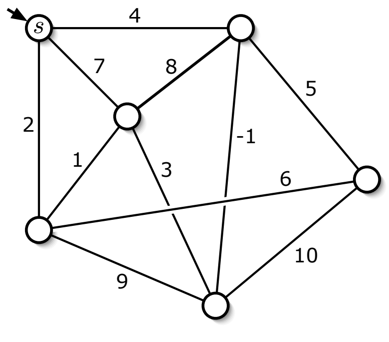
5. Find the following spanning trees for the weighted graph shown below.

   (a) A breadth-first spanning tree rooted at s.

   (b) A depth-first spanning tree rooted at s.

   (c) A shortest-path tree rooted at s.

   (d) A minimum spanning tree.



   You do *not* need to justify your answers; just clearly indicate the edges of each spanning tree. Yes, one of the edges has negative weight.

[1]But not both! If you give us *both* a proof *and* a disproof for the same statement, you will get no credit, even if one of your arguments is correct.

1. A *double-Hamiltonian* circuit in an undirected graph G is a closed walk that visits every vertex in G exactly *twice*, possibly by traversing some edges more than once. ***Prove*** that it is NP-hard to determine whether a given undirected graph contains a double-Hamiltonian circuit.

2. Suppose you are running a web site that is visited by the same set of people every day. Each visitor claims membership in one or more *demographic groups*; for example, a visitor might describe himself as male, 31-40 years old, a resident of Illinois, an academic, a blogger, a Joss Whedon fan[1], and a Sports Racer.[2] Your site is supported by advertisers. Each advertiser has told you which demographic groups should see its ads and how many of its ads you must show each day. Altogether, there are n visitors, k demographic groups, and m advertisers.

   Describe an efficient algorithm to determine, given all the data described in the previous paragraph, whether you can show each visitor exactly *one* ad per day, so that every advertiser has its desired number of ads displayed, and every ad is seen by someone in an appropriate demographic group.

3. Describe and analyze a data structure to support the following operations on an array $X[1..n]$ as quickly as possible. Initially, $X[i] = 0$ for all i.

   - Given an index i such that $X[i] = 0$, set $X[i]$ to 1.
   - Given an index i, return $X[i]$.
   - Given an index i, return the smallest index $j \geq i$ such that $X[j] = 0$, or report that no such index exists.

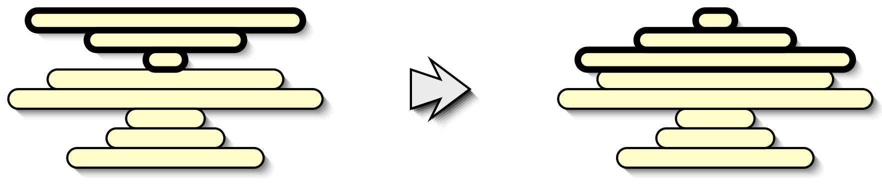
   For full credit, the first two operations should run in *worst-case constant* time, and the amortized cost of the third operation should be as small as possible. *[Hint: Use a modified union-find data structure.]*

4. The next time you are at a party, one of the guests will suggest everyone play a round of Three-Way Mumbledypeg, a game of skill and dexterity that requires three teams and a knife. The official Rules of Three-Way Mumbledypeg (fixed during the Holy Roman Three-Way Mumbledypeg Council in 1625) require that (1) each team *must* have at least one person, (2) any two people on the same team *must* know each other, and (3) everyone watching the game *must* be on one of the three teams. Of course, it will be a really *fun* party; nobody will want to leave. There will be several pairs of people at the party who don't know each other. The host of the party, having heard thrilling tales of your prowess in all things algorithmic, will hand you a list of which pairs of partygoers know each other and ask you to choose the teams, while he sharpens the knife.

   Either describe and analyze a polynomial time algorithm to determine whether the partygoers can be split into three legal Three-Way Mumbledypeg teams, or prove that the problem is NP-hard.

[1]Har har har! Mine is an evil laugh! Now *die!*
[2]It's Ride the Fire Eagle Danger Day!

5. Suppose you are given a stack of n pancakes of different sizes. You want to sort the pancakes so that smaller pancakes are on top of larger pancakes. The only operation you can perform is a *flip*—insert a spatula under the top k pancakes, for some integer k between 1 and n, and flip them all over.



Flipping the top three pancakes.

   (a) Describe an efficient algorithm to sort an arbitrary stack of n pancakes. *Exactly* how many flips does your algorithm perform in the worst case? (For full credit, your algorithm should perform as few flips as possible; an optimal $\Theta()$ bound is worth three points.)

   (b) Now suppose one side of each pancake is burned. *Exactly* how many flips do you need to sort the pancakes *and* have the burned side of every pancake on the bottom? (For full credit, your algorithm should perform as few flips as possible; an optimal $\Theta()$ bound is worth three points.)

6. Describe and analyze an efficient algorithm to find the length of the longest substring that appears both forward and backward in an input string $T[1..n]$. The forward and backward substrings must not overlap. Here are several examples:

   - Given the input string ALGORITHM, your algorithm should return 0.
   - Given the input string RECURSION, your algorithm should return 1, for the substring R.
   - Given the input string REDIVIDE, your algorithm should return 3, for the substring EDI. (The forward and backward substrings must not overlap!)
   - Given the input string DYNAMICPROGRAMMINGMANYTIMES, your algorithm should return 4, for the substring YNAM.

   For full credit, your algorithm should run in $O(n^2)$ time.

7. A *double-Eulerian* circuit in an undirected graph G is a closed walk that traverses every edge in G exactly twice. Describe and analyze a *polynomial-time* algorithm to determine whether a given undirected graph contains a double-Eulerian circuit.