Neatly print your name(s) (first name first, with no commas), your network ID(s), and the alias(es) you used for Homework 0 in the boxes above.

Required Problems

1. [10 Points] Small 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 and 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.

(a) In the United States, there is a simple greedy algorithm that always results in the smallest number of coins: subtract the largest coin and recursively give change for the remainder. El Generalissimo does not approve of American capitalist greed. Show that there is a set of coin values for which the greedy algorithm does not always give the smallest possible number of coins.

(b) 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 $A$. 
2. **[10 Points] Party Time**

A company is planning a party for its employees. The employees in the company are organized into a strict hierarchy, that is, a tree with the company president at the root. The organizers of the party have assigned a real number to each employee measuring how 'fun' the employee is. In order to keep things social, there is one restriction on the guest list: an employee cannot attend the party if their immediate supervisor is present. On the other hand, the president of the company must attend the party, even though she has a negative fun rating; it's her company, after all. Give an algorithm that makes a guest list for the party that maximizes the sum of the 'fun' ratings of the guests.

3. **[10 Points] Fast Partition**

(a) **[5 Points]** Consider the following NP-complete problem.

**Problem:** PARTITION

**Instance:** A finite set \( A \) and a "size" \( s(a) \in \mathbb{Z}^+ \) for each \( a \in A \).

**Question:** Is there a subset \( A' \subseteq A \) such that

\[
\sum_{a \in A'} s(a) = \sum_{a \notin A \setminus A'} s(a) ?
\]

Now suppose that \( s(a) \leq |A| \) for each \( a \in A \), give an algorithm that solve PARTITION in \( O(n^3) \) time, where \( n = |A| \).

(b) **[5 Points]** Suppose you have one machine and a set of \( n \) tasks \( a_1, a_2, ..., a_n \). Each task \( a_i \) has a processing time \( t_i \), a profit \( p_i \), and a deadline \( d_i \). The machine can process only one task at a time, and task \( a_i \) must run uninterruptedly for \( t_i \) consecutive time units to complete. If you complete task \( a_i \) by its deadline \( d_i \), you receive a profit \( p_i \). But you receive no profit if you complete it after its deadline. As an optimization problem, you are given the processing times, profits and deadlines for a set of \( n \) tasks, and you wish to find a schedule that completes all the tasks and returns the greatest amount of profit.

Give a polynomial-time algorithm for the decision problem, assuming that all processing times are integers from 1 to \( n \).

4. **[10 Points] Bitonic euclidean traveling-salesman problem**

(Based on CLRS 15-1)

The euclidean traveling-salesman problem is the problem of determining the shortest closed tour that connects a given set of \( n \) points in the plane. Figure [1](a) below shows the solution to a 7-point problem. The general problem is NP-complete, and its solution is therefore believed to require more than polynomial time.

J.L. Bentley has suggested that we simplify the problem by restricting our attention to bitonic tours, that is, tours that start at the leftmost point, go strictly left to right to the rightmost point, and then go strictly right to left back to the starting point. Figure [1](b) shows the shortest bitonic tour of the same 7 points. In this case, a polynomial-time algorithm is possible.
Describe an $O(n^2)$-time algorithm for determining an optimal bitonic tour. You may assume that no two points have the same x-coordinate. [Hint: Scan left to right, maintaining optimal possible for the two parts of the tour.]

5. [10 Points] Moving on a checkerboard
(Based on CLRS 15-6)

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:

- (a) the square immediately above,
- (b) the square that is one up and one left (but only if the checker is not already in the leftmost column),
- (c) the square that is one up and one 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 $p(x,y)$ for all pairs $(x,y)$ 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. Your 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?

6. [20 Points] Narrow Graphs
Suppose you are given a graph $G = (V, E)$. The $n$ vertices $v_1, v_2, ..., v_n$ of the graph $G$ are ordered in such a way that for any edge $(v_i, v_j) \in E$, we have $|i - j| \leq c$, where $c$ is a small constant.

(a) [5 Points] Give a linear time algorithm to find largest clique in $G$.

(b) [5 Points] Give a linear time algorithm to find the largest independent set in the graph.

(c) [5 Points] Give a linear time algorithm to find the minimum number $k$ such that the graph can be $k$-Colored.

(d) [5 Points] Give a linear time algorithm that computes a Hamiltonian cycle in $G$, if such a cycle exists.

![Figure 2](image_url)

Figure 2: (a) The original polygonal line with 14 vertices. (b) A new polygonal line with 6 vertices. (c) The distance between $p_5$ on the original polygonal line and the simplification segment $p_4p_6$. The error of $p_5$ is $\text{error}(p_5) = \text{dist}(p_5, p_4p_6)$.

7. [10 Points] Polygonal Line Simplification

You are given a polygonal line $\gamma$ made out of $n$ vertices in the plane. Namely, you are given a list of $n$ points in the plane $p_1, ..., p_n$, where $p_i = (x_i, y_i)$. You need to display this polygonal line on the screen, however, you realize that you might be able to draw a polygonal line with considerably less vertices that looks identical on the screen (because of the limited resolution of the screen). It is crucial for you to minimize the number of vertices of the polygonal line. (Because, for example, your display is a remote Java applet running on the user computer, and for each vertex of the polygon you decide to draw, you need to send the coordinates of the points through the network which takes a long time. So the fewer vertices you send, the snappier your applet would be.)

So, given such a polygonal line $\gamma$, and a parameter $k$, you would like to select $k$ vertices of $\gamma$ that yield the “best” polygonal line that looks like $\gamma$. (Please see the Figure 2)

Namely, you need to build a new polygonal line $\gamma'$ and minimize the difference between the two polygonal lines. The polygonal line $\gamma'$ is built by selecting $k$ vertices $\{p_{i_1}, p_{i_2}, ..., p_{i_k}\}$ from $\gamma$. It is required that $i_1 = 1$, $i_k = n$, and $i_j < i_{j+1}$ for $j = 1, 2, ..., k - 1$. 


We define the error between $\gamma$ and $\gamma'$ by how far from $\gamma'$ are the vertices of $\gamma$. More formally, The difference between the two polygonal lines is

$$\text{error}(\gamma, \gamma') = \sum_{j=1}^{k-1} \sum_{m=i_j+1}^{i_{j+1}-1} \text{dist}(p_m, p_i_j p_{i_j+1}).$$

Namely, for every vertex not in the simplification, its associated error, is the distance to the corresponding simplified segment. The overall error is the maximum over all vertices. See Figure 2(c).

You can assume there you are provided with a subroutine that can calculate $\text{dist}(u, vw)$ in constant time, where $\text{dist}(u, vw)$ is the distance between the point $u$ and the segment $vw$.

Give an $O(n^3)$ time algorithm to find the $\gamma'$ that minimizes $\text{error}(\gamma, \gamma')$. 
Practice Problems

1. Consider the following strange sorting algorithm. For simplicity we will assume that \( n \) is always some positive power of 2 (i.e. \( n = 2^i \), for some positive integer \( i > 0 \)).

\[
\text{4-StupidSort}(A[0..n-1]):
\]

\[
\text{if } n \leq 8 \\
\quad \text{INSERTIONSORT}(A[0..n-1])
\]

\[
\text{else } /\!\!/ n > 8 \quad */
\]

\[
\quad \text{for } i \leftarrow 0 \text{ to } 2
\]

\[
\quad \quad \text{for } j \leftarrow 2 \text{ to } i
\]

\[
\quad \quad \quad \text{4-StupidSort}(A[jn/4..(j+2)n/4-1])
\]

(a) Prove that 4-StupidSort actually sorts its input.

(b) State a recurrence (including the base case(s)) for the number of comparisons executed by 4-StupidSort.

(c) Solve the recurrence, and prove that your solution is correct. Does the algorithm deserve its name?

(d) Show that the number of swaps executed by 4-StupidSort is at most \( \left( \binom{n}{2} \right) \).

2. The following randomized algorithm selects the \( r \)th smallest element in an unsorted array \( A[1..n] \). For example, to find the smallest element, you would call \text{RANDOMSELECT}(A, 1); to find the median element, you would call \text{RANDOMSELECT}(A, \lfloor n/2 \rfloor). Recall from lecture that \text{Partition} splits the array into three parts by comparing the pivot element \( A[p] \) to every other element of the array, using \( n-1 \) comparisons altogether, and returns the new index of the pivot element.

\[
\text{RANDOMSELECT}(A[1..n], r):
\]

\[
p \leftarrow \text{RANDOM}(1, n)
\]

\[
k \leftarrow \text{PARTITION}(A[1..n], p)
\]

\[
\text{if } r < k \\
\quad \text{return RANDOMSELECT}(A[1..k-1], r)
\]

\[
\text{else if } r > k \\
\quad \text{return RANDOMSELECT}(A[k+1..n], r-k)
\]

\[
\text{else}
\]

\[
\text{return } A[k]
\]

(a) State a recurrence for the expected running time of RANDOMSELECT, as a function of \( n \) and \( r \).

(b) What is the exact probability that RANDOMSELECT compares the \( i \)-th smallest and \( j \)-th smallest elements in the input array? The correct answer is a simple function of \( i, j, \) and \( r \). [Hint: Check your answer by trying a few small examples.]

(c) Show that for any \( n \) and \( r \), the expected running time of RANDOMSELECT is \( \Theta(n) \). You can use either the recurrence from part (a) or the probabilities from part (b). For extra credit, find the exact expected number of comparisons, as a function of \( n \) and \( r \).
(d) What is the expected number of times that \texttt{RandomSelect} calls itself recursively?

3. Solve the following randomization problems:

(a) Given a function \texttt{RandBit()} that returns 0 or 1 with equal probability, write a function \texttt{Rand(n)} that returns a random integer between 0 and \( n - 1 \) (inclusive) with uniform distribution over those possible values.

(b) Given a function \texttt{RandReal()} that returns a random real number between 0 and 1, describe a procedure that generates a random permutation of the integer set \( \{1, \ldots, n\} \) where every possible permutation has the same probability of being returned. You can assume that \texttt{RandReal()} never returns the same number twice.

(c) Given a function \texttt{Rand(n)} as describe above, describe a linear-time algorithm that outputs a permutation of the integer set \( \{1, \ldots, n\} \) where all permutations have equal probability. You can assume that \texttt{Rand(n)} takes constant time.
4. You are watching a stream of packets go by one at a time, and want to take a random sample of \( k \) distinct packets from the stream. You face several problems:

- You only have room to save \( k \) packets at any one time.
- You do not know the total number of packets in the stream.
- If you choose not to save a packet as it goes by, it is gone forever.

In each of the three scenarios below, devise a scheme so that whenever the packet stream terminates you are left holding a subset of \( k \) packets chosen uniformly at random from the entire packet stream. If the total number of packets in the stream is less than \( k \), you should hold all of these packets. (hint: To verify your solution, imagine that you now repeat the same experiment with the same stream sent in the reverse order. The probability to get the same output in the two experiments should be the same.)

(a) Prior to watching the stream you know that the total number of packets is some number \( n \). Also, you have room to save all \( n \) packets (i.e. \( k = n \)).

(b) In this scenario you the know the values of both \( n \) and \( k \) in advance.

(c) Here you still have room to hold \( k \) packets. However, you have no idea how many packets will flow through in the stream (i.e., \( n \) is unknown in advance).

5. Recall from lecture the notion of an edit distance between two words (Lecture 2, Section 2.7). There we derived an algorithm EDITDISTANCE to compute the minimum number of legal transformations required to convert one string into another (i.e. the edit distance). Let us now consider the problem assuming we have some additional information. Namely, suppose we are given a parameter \( k \) that is an upper limit on the edit distance of the two string parameters (\( k \) is some positive finite integer). Write a new procedure K-EDITDISTANCE to achieve the same results as EDITDISTANCE, this time taking advantage of the additional \( k \) parameter. What are the new and improved space and time complexities?

6. **Finding maximal longest monotonically increasing subsequence**

(a) Give an \( O(n^2) \)-time algorithm to find the longest monotonically increasing subsequence of a sequence of \( n \) numbers. Recall that a monotonically increasing sequence is a sequence \( s_1, \ldots, s_n \) where \( i \leq j \Rightarrow s_i \leq s_j \) for all \( i \) and \( j \) in \( \{1, \ldots, n\} \).

(b) Give an \( O(n \lg n) \)-time algorithm to find the longest monotonically increasing subsequence of a sequence of \( n \) numbers. (Hint: Observe that the last element of a candidate subsequence of length \( i \) is at least as large as the last element of a candidate subsequence of length \( i - 1 \). Maintain candidate subsequences by linking them through the input sequence.)
7. Let an integer array \( A[0..n-1] \) represent a histogram with \( n \) buckets, each of width 1 (see the figure below). Informally, a *bucket* is a single-valued bar in the histogram while *width* is measured by the number of array cells beneath an object (i.e. a single bucket can have width greater than one). For clarity, assume throughout this entire problem that \( n \geq 1 \) and that each of the \( n \) array entries are integers in some predetermined finite range \([0..M]\).

A \( k \)-cover of a histogram is an approximation in the form of a new histogram of the same total width, but with \( k \) buckets (possibly of \( k \) different widths) instead of the \( n \) buckets in the original histogram (see the figures below). Each of the new \( k \) buckets have a width which is a non-negative integer number\(^1\). The height of a new bucket is an integer number in the range \([0..M]\).

\[ A[] : [4, 3, 6, 0, 2, 4, \ldots, n-1] \]

Histogram interpretation of an integer array.

The error of a \( k \)-cover is the area of the geometric difference between the original histogram and its cover (i.e. the total area between the original histogram graph and the covering

\(^1\)A new bucket must cover all or none of any given original bucket (i.e. cannot split an original bucket between two or more new buckets).
histogram graph). The illustrations above should clarify this quantity. A best k-cover of a given histogram is a k-cover whose error is no larger than any other k-cover of that same histogram (for the same value of k). The covers shown above are definitely not optimal.

(a) How many bits are required to store a histogram?

(b) How many bits are required to store a k-cover of a histogram?

(c) Write an algorithm 1-COVER(A[l..r]) that determines the best 1-cover of the histogram A[l..r] and returns the value (height) of that histogram (NOTE: we are only covering A from index l through index r). You can assume that 0 ≤ l ≤ r < n.

(d) Using 1-COVER, write an algorithm K-COVER(A[0..n−1], k, W[0..k−1], H[0, .., k−1]) that determines the best k-cover of A[0..n−1] and assigns the corresponding bucket widths of the k-cover into the array W[0..k−1] and their heights into H[0, .., k−1].

(e) What are the space and running time complexities of K-COVER?

(f) Describe an algorithm that receives as an input an histogram and constructs a data-structure in O(n log n) time, so that given query interval [l..r], it computes 1-COVER(A[l..r]) in O(log M log^2 n) time. (It is possible to do even better than that.)

8. 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 requiring 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 ≤ j ≤ n, because Champaign needs no more victories to win. P(i, 0) = 0, 1 ≤ i ≤ 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 ≥ 1 and j ≥ 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 \(P(n, n)\)).
9. The traditional Devonian/Cornish drinking song “The Barley Mow” has the following pseudolyrics\(^2\), where \(\text{container}[i]\) is the name of a container that holds \(2^i\) ounces of beer\(^3\):

<table>
<thead>
<tr>
<th>BarleyMow((n)):</th>
</tr>
</thead>
<tbody>
<tr>
<td>“Here’s a health to the barley-mow, my brave boys,”</td>
</tr>
<tr>
<td>“Here’s a health to the barley-mow!”</td>
</tr>
<tr>
<td>“We’ll drink it out of the jolly brown bowl,”</td>
</tr>
<tr>
<td>“Here’s a health to the barley-mow!”</td>
</tr>
<tr>
<td>“Here’s a health to the barley-mow, my brave boys,”</td>
</tr>
<tr>
<td>“Here’s a health to the barley-mow!”</td>
</tr>
</tbody>
</table>

for \(i \leftarrow 1\) to \(n\)

“We’ll drink it out of the container\([i]\), boys,”

“Here’s a health to the barley-mow!”

for \(j \leftarrow i\) downto \(1\)

“The container\([j]\),”

“And the jolly brown bowl!”

“Here’s a health to the barley-mow!”

“Here’s a health to the barley-mow, my brave boys,”

“Here’s a health to the barley-mow!”

(a) Suppose each container name \(\text{container}[i]\) is a single word, and you can sing four words a second. How long would it take you to sing \(\text{BarleyMow}(n)\)? (Give a tight asymptotic bound.)

(b) If you want to sing this song for \(n > 20\), you’ll have to make up your own container names, and to avoid repetition, these names will get progressively longer as \(n\) increases\(^4\). Suppose \(\text{container}[n]\) has \(\Theta(\log n)\) syllables, and you can sing six syllables per second. Now how long would it take you to sing \(\text{BarleyMow}(n)\)? (Give a tight asymptotic bound.)

(c) Suppose each time you mention the name of a container, you drink the corresponding amount of beer: one ounce for the jolly brown bowl, and \(2^i\) ounces for each \(\text{container}[i]\). Assuming for purposes of this problem that you are at least 21 years old, exactly how many ounces of beer would you drink if you sang \(\text{BarleyMow}(n)\)? (Give an exact answer, not just an asymptotic bound.)

\(^2\)Pseudolyrics are to lyrics as pseudocode is to code.

\(^3\)One version of the song uses the following containers: nipperkin, gill pot, half-pint, pint, quart, pottle, gallon, half-anker, anker, firkin, half-barrel, barrel, hogshead, pipe, well, river, and ocean. Every container in this list is twice as big as its predecessor, except that a firkin is actually 2.25 ankers, and the last three units are just silly.

\(^4\)“We’ll drink it out of the hemisemidemiyottapint, boys!”
10. Suppose we want to display a paragraph of text on a computer screen. The text consists of \( n \) words, where the \( i \)th 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 white space 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 \( P - j + i - \sum_{k=i}^{j-1} p_k \). Describe a dynamic programming algorithm to print the paragraph with minimum slop.

11. You are at a political convention with \( n \) delegates. Each delegate is a member of exactly one political party. It is impossible to tell which political party a delegate belongs to. However, you can check whether any two delegates are in the *same* party or not by introducing them to each other. (Members of the same party always greet each other with smiles and friendly handshakes; members of different parties always greet each other with angry stares and insults.)

   (a) Suppose a majority (more than half) of the delegates are from the same political party. Give an efficient algorithm that identifies a member of the majority party.

   (b) Suppose exactly \( k \) political parties are represented at the convention and one party has a *plurality*: more delegates belong to that party than to any other. Present a practical procedure to pick a person from the plurality party as parsimoniously as possible. (Please.)

12. Give an algorithm that finds the second smallest of \( n \) elements in at most \( n + \lceil \lg n \rceil - 2 \) comparisons. [Hint: divide and conquer to find the smallest; where is the second smallest?]

13. A company is planning a party for its employees. The employees in the company are organized into a strict hierarchy, that is, a tree with the company president at the root. The organizers of the party have assigned a real number to each employee measuring how *fun* the employee is. In order to keep things social, there is one restriction on the guest list: an employee cannot attend the party if their immediate supervisor is present. On the other hand, the president of the company *must* attend the party, even though she has a negative fun rating; it's her company, after all. Give an algorithm that makes a guest list for the party that maximizes the sum of the *fun* ratings of the guests.

14. Suppose you have a subroutine that can find the median of a set of \( n \) items (*i.e.*, the \( \lfloor n/2 \rfloor \) smallest) in \( O(n) \) time. Give an algorithm to find the \( k \)th biggest element (for arbitrary \( k \)) in \( O(n) \) time.
15. You're walking along the beach and you stub your toe on something in the sand. You dig around it and find that it is a treasure chest full of gold bricks of different (integral) weight. Your knapsack can only carry up to weight $n$ before it breaks apart. You want to put as much in it as possible without going over, but you cannot break the gold bricks up.

(a) Suppose that the gold bricks have the weights $1, 2, 4, 8, \ldots, 2^k$, $k \geq 1$. Describe and prove correct a greedy algorithm that fills the knapsack as much as possible without going over.

(b) Give a set of 3 weight values for which the greedy algorithm does not yield an optimal solution and show why.

(c) Give a dynamic programming algorithm that yields an optimal solution for an arbitrary set of gold brick values.