Problem Set 1

Data Structures and Algorithms, Fall 2020

Due: September 17, in class.

Problem 1

Consider the following algorithm to sort n numbers stored in an array A. First find the smallest element of A and swap it with the element stored in $A[1]$. Then find the second smallest element of A and swap it with the element stored in $A[2]$. Continue in this manner for $n-1$ times and we are done.

(a) Give the pseudocode of this algorithm.

(b) Give the best-case and worst-case running times of this algorithm in Θ-notation.

(c) What loop invariant does this algorithm maintain? State it, prove it, and then use it to show the correctness of the algorithm.

Problem 2

Given $c_0, c_1, \cdots, c_n$ and a value for x, we can evaluate a polynomial

$$P(x) = \sum_{k=0}^{n} c_k x^k = c_0 + x(c_1 + x(c_2 + \cdots + x(c_{n-1} + xc_n)\cdots))$$

using the following algorithm:

```
PolyEval($x, c_0, c_1, \cdots, c_n$)
1: $y \leftarrow 0$
2: for ($i = n$ downto $0$) do
3:     $y \leftarrow c_i + xy$
```

(a) Give the runtime of `PolyEval` in Θ-notation.

(b) What loop invariant does `PolyEval` maintain? State it, prove it, and then use it to show the correctness of `PolyEval`.

Problem 3

(a) Let $f(n) : \mathbb{N}^+ \mapsto \mathbb{R}^+$ and $g(n) : \mathbb{N}^+ \mapsto \mathbb{R}^+$ be two functions, prove that $\max\{f(n), g(n)\} = \Theta(f(n) + g(n))$.

(b) Let $a \in \mathbb{R}$ and $b \in \mathbb{R}^+$ be two constants, prove that $(n+a)^b = \Theta(n^b)$.

(c) Among asymptotic notations $o, O, \omega, \Omega, \Theta$, which can be used to describe the relationship between $\sqrt{n}$ and $n^{\sin(n)}$? You do not need to prove your answer.

Problem 4

Sort the following functions from asymptotically smallest to asymptotically largest, indicating ties if there are any. You do not need to prove your answer. To simplify notation, write $f(n) \ll g(n)$ to mean $f(n) = o(g(n))$ and $f(n) = g(n)$ to mean $f(n) = \Theta(g(n))$.

$\lg(\lg^* n)$	$2^{\lg^* n}$	$(\sqrt{2})^{\lg n}$	n^2	$n!$	$(\lg n)!$
$(3/2)^n$	n^3	$\lg^2 n$	$\lg(n!)$	2^{2^n}	$n^{1/\lg n}$
$\ln \ln n$	$\lg^* n$	$n \cdot 2^n$	$n^{\lg \lg n}$	$\ln n$	1
$2^{\lg n}$	$(\lg n)^{\lg n}$	e^n	$4^{\lg n}$	$(n+1)!$	$\sqrt{\lg n}$
$\lg^*(\lg n)$	$2^{\sqrt{2 \lg n}}$	n	2^n	$n \lg n$	$2^{2^{n+1}}$

Problem 5

Explain how to implement a FIFO queue using two stacks. You should give a brief overview of your implementation, then provide pseudocode for `enqueue` and `dequeue` operations. Assuming `push` and `pop` each takes $\Theta(1)$ time, analyze the running time of `enqueue` and `dequeue`.

Problem 6

Design a MINSTACK data structure that can store comparable elements and supports stack operations `push(x)`, `pop()`, as well as the `min()` operation, which returns the minimum value currently stored in the data structure. All operations should run in $O(1)$ time in your implementation. (You may assume there exists a STACK data structure which supports `push` and `pop`, and both operations run in $\Theta(1)$ time.) You should give a brief overview of your MINSTACK data structure, then provide pseudocode for each of the three operations. You should also discuss the space complexity of your implementation.

Problem 7

Design a RANDOMQUEUE data structure. This is an implementation of the QUEUE interface in which the `remove` operation removes an element that is chosen uniformly at random among all the elements currently in the queue. You may assume you have access to a `random` function which takes a positive integer as input: `random(x)` returns a uniformly chosen random integer in $[1, x]$, in $O(1)$ time. You should give a brief overview of your data structure, then provide pseudocode for `add` and `remove` operations. You should also discuss the time complexity of `add` and `remove` operations. (To get full credit, `add` and `remove` operations should run in $O(1)$ time in your implementation. You may assume the number of elements in the queue never exceeds N.)

Problem 8

You are given an infix expression in which each operator is in $\{+, \times, !\}$ and each operand is a single digit positive integer. Write an algorithm to convert it to postfix. (For example, given "$2 + 3 \times 3!$", whose value is 20, your algorithm should output "$233! \times +$".) You should give a brief overview of your algorithm, then provide pseudocode, and finally discuss its time complexity. (To get full credit, you your algorithm need to have $O(n)$ time complexity.)

Problem Set 2

Data Structures and Algorithms, Fall 2020

Due: September 24, in class.

Problem 1

Prove that the solution of recurrence $T(n) = 2T(\lfloor n/2 \rfloor + 1) + n$ is in $O(n \lg n)$.

Problem 2

(a) Use the recursion tree method to find a good asymptotic upper bound for the recurrence $T(n) = T(n-2) + T(n/4) + n$. Then, use the substitution method to prove your answer.

(b) Give an asymptotic *tight* bound for the recurrence $T(n) = T(\alpha n) + T((1-\alpha)n) + cn$, where $a \in (0, 1)$ is a constant and $c > 0$ is also a constant. You do *not* need to prove your answer.

Problem 3

Let A be an array containing n distinct numbers. If $i < j$ and $A[i] > A[j]$, then the pair (i, j) is called an *inversion* of A.

(a) List all inversions of the array $\langle 2, 3, 8, 6, 1 \rangle$.

(b) What is the relationship between the running time of insertion sort and the number of inversions in the input array? To get full credit, prove your answer is correct.

(c) Give an $O(n \lg n)$ time algorithm that can count the number of inversions of a size n array. Your algorithm does not need to list all inversions. You do *not* need to prove your algorithm is correct. *(Hint: Modify the merge sort algorithm.)*

Problem 4

In this class, we often assume that parameter passing during procedure calls takes constant time, even if an N-element array is being passed. This assumption is valid in many systems because a pointer to the array is passed, not the array itself. This problem examines the implications of three different parameter-passing strategies:

1. An array is passed by pointer. Time $= \Theta(1)$.
2. An array is passed by copying. Time $= \Theta(N)$, where N is the size of the array.
3. An array is passed by copying only the subrange that might be accessed by the called procedure. Time $= \Theta(q - p + 1)$ if the subarray $A[p, \cdots, q]$ is passed.

For each of the three methods above, give the recurrence for the worst-case running time of merge sort when arrays are passed using that method, and give a good upper bound on the solution of the recurrence. Use N to denote the size of the original problem and n to denote the size of a subproblem.

Problem 5

Denote the running time of the *fastest* algorithm to square an n-bit integer as $T_1(n)$, and denote the running time of the *fastest* algorithm to multiply two n-bit integers as $T_2(n)$. Professor F. Lake claims that $T_1(n) = o(T_2(n))$, i.e., it is asymptotically faster to square an n-bit integer than to multiply two n-bit integers. Is this claim true or false? Prove your answer.

Problem 6

Recall the FINDMAXIMUMSUBARRAY algorithm introduced in Section 4.1 in the textbook CLRS.[1] Modify it to obtain an $O(n)$ time divide-and-conquer algorithm for the maximum subarray problem. Your algorithm should not leverage dynamic programming.[2] In particular, your algorithm should not look like an answer for Exercise 4.1-5 in CLRS. *(Hint: (a) The recurrence for the runtime of the modified algorithm should be $T(n) = 2T(n/2) + O(1)$. (b) In the "conquer" step, instead of just computing the maximum subarray in $A[low, \cdots, high]$, compute and return some more information so that the "combine" step is faster. In particular, finding "maximum crossing subarray" should be fast in the combine step.)*

Problem 7

You are having a party with n other friends, each of which plays either as a citizen or a werewolf. You do not know who are citizens or who are werewolves, but all your friends do. There are always more citizens than there are werewolves, but n can be an arbitrary positive integer.

Your allowed "query" operation is as follows: You pick two people and ask each of them if the other person is a citizen or a werewolf. When you do this, a citizen must tell the truth about the identity of the other person, whereas a werewolf does not have to. (That is, a werewolf may lie about the identity of the other person.) Your algorithm should be correct regardless of the behavior of the werewolves.

(a) Devise an algorithm that, given a player as input, can determine whether the player is a citizen using $O(n)$ queries. Prove the correctness of your algorithm carefully.

(b) Devise a divide-and-conquer algorithm that finds a citizen using $O(n \lg n)$ queries. Prove the correctness of your algorithm carefully. Do not devise a linear time algorithm here, as we would like you to practice with divide-and-conquer. *(Hint: Split a group of people into two groups, think about what invariant must hold for at least one of these two groups. Also, you may find solution for part (a) useful.)*

(c) [Bonus Question][3] Devise an algorithm that finds a citizen using $O(n)$ queries. Prove the correctness of your algorithm carefully. *(Hint: Don't be afraid to sometimes "throw away" a pair of people once you have asked them to identify each other.)*

[1] Recall CLRS refers to "Introduction to Algorithms (3ed)".

[2] If you do not know what is "dynamic programming", just ignore this sentence.

[3] You are *not* required to solve bonus questions. But you get extra credit if you do!

Problem Set 3

Data Structures and Algorithms, Fall 2020

Due: October 10, in class.

Problem 1

(a) Prove that in a heap containing n nodes, there are at most $\lceil n/2^{h+1} \rceil$ nodes of height h.

(b) Assume operation $\textsc{HeapDel}(A, i)$ deletes the element at index position i from max-heap A. Give an implementation of $\textsc{HeapDel}(A, i)$ that runs in $O(\lg n)$ time for an n-element max-heap.

*Problem 2 [Bonus Problem]

Prove that when all elements are distinct, the best-case running time of $\textsc{HeapSort}$ is $\Omega(n \lg n)$.

Problem 3

We call an $m \times n$ matrix *magical* if: (a) for each row, elements in that row are in sorted order (i.e., non-decreasing) from left to right; and (b) for each column, elements in that column are in sorted order from top to bottom. Some entries in a magical matrix may be ∞, and we treat them as if no elements were in those entries. Therefore, a magical matrix can store at most mn elements.

(a) Draw a 4×4 magical matrix containing elements $\{9, 16, 3, 2, 4, 8, 5, 14, 12\}$.

(b) Describe an algorithm to implement $\textsc{ExtractMin}$ on a nonempty $m \times n$ magical matrix that runs in $O(m + n)$ time. That is, the algorithm finds the minimum element in the matrix, removes it from the matrix, and returns the value of that element. The modified matrix should still be a magical matrix. *(Hint: Think about* $\textsc{MaxHeapify}$.*)* You need to prove the correctness of your algorithm, and you need to argue your algorithm indeed runs in $O(m + n)$ time.

(c) Describe an algorithm that can insert an element into a non-full $m \times n$ magical matrix in $O(m + n)$ time. You need to prove the correctness and argue the time complexity of your algorithm.

(d) Describe an algorithm to use an $n \times n$ magical matrix to sort n^2 numbers in $O(n^3)$ time. Your algorithm should not use any other sorting method as a subroutine. You need to prove the correctness and argue the time complexity of your algorithm.

(e) Describe an $O(m + n)$ time algorithm to determine whether a given number is in a given $m \times n$ magical matrix. You do *not* need to prove the correctness of your algorithm, but you need to argue the time complexity of your algorithm.

Problem 4

(a) Prove that quicksort's best-case running time is $\Omega(n \lg n)$.

(b) Recall the randomized quicksort introduced in class. When randomized quicksort runs, how many calls are made to the random number generator $\textsc{Random}$ in the worst case? How about in the best case? Give your answers in terms of Θ-notation. You do not need to prove your answers.

Problem 5

(a) When all input elements are equal, what is the running time of quicksort? What about randomized quicksort? Give your answers using asymptotic notations. You do not need to prove your answers.

(b) Modify the PARTITION procedure to produce a procedure PARTITION$'(A, p, r)$, which permutes the elements of $A[p, \cdots, r]$ and returns two indices q and t, where $p \leq q \leq t \leq r$, such that

- all elements of $A[q, \cdots, t]$ are equal;
- each element in $A[p, \cdots, q-1]$ is less than $A[q]$;
- each element in $A[t+1, \cdots, r]$ is greater than $A[q]$.

Also, discuss whether your PARTITION$'(A, p, r)$ procedure is stable. Notice, to get full credit, your PARTITION$'(A, p, r)$ procedure should take $\Theta(r-p)$ time and be in-place. You do not need to prove the correctness of your procedure.

(c) Redo part (a), assuming using your PARTITION$'(A, p, r)$ from part (b) as the partition procedure.

Problem 6

One way to improve randomized quicksort is to partition around a pivot that is chosen more carefully than by picking a random element from the subarray. One common approach is the median-of-3 method: choose the pivot as the median of a set of 3 elements randomly selected from the subarray. For this problem, let us assume that the elements in the input array $A[1, \cdots, n]$ are distinct and that $n \geq 3$. We denote the sorted output array by $A'[1, \cdots, n]$. Using the median-of-3 method to choose the pivot element x, define $p_i = \Pr(x = A'[i])$.

(a) Give an exact formula for p_i as a function of n and i for $i = 2, 3, \cdots, n-1$. (Note that $p_1 = p_n = 0$.)

(b) By what amount have we increased the likelihood of choosing the pivot as $A'[\lfloor (n+1)/2 \rfloor]$, the median of $A[1, \cdots, n]$, compared with the ordinary implementation? Assume that $n \to \infty$, and give the limiting ratio of these probabilities.

(c) If we define a "good" split to mean choosing the pivot as $x = A'[i]$, where $n/3 \leq i \leq 2n/3$, by what amount have we increased the likelihood of getting a good split compared with the ordinary implementation? *(Hint: Approximate the sum by an integral.)*

(d) Does the median-of-3 method reduce the (asymptotic) best-case or worst-case running time of randomized quicksort? If your answer is negative, discuss why the method could be considered as an "improvement" for randomized quicksort.

Problem 7

```
STOOGESORT($A[0, \cdots, n-1]$)
1: if ($n == 2$ and $A[0] > A[1]$) then
2:     SWAP($A[0], A[1]$).
3: else if ($n > 2$) then
4:     $m \leftarrow \lceil 2n/3 \rceil$.
5:     STOOGESORT($A[0, \cdots, m-1]$).
6:     STOOGESORT($A[n-m, \cdots, n-1]$).
7:     STOOGESORT($A[0, \cdots, m-1]$).
```

(a) What is the running time of the algorithm? Prove your answer. *(Hint: Ignore the ceiling.)*

(b) Prove that the algorithm actually sorts its input.

(c) Is the algorithm still correct if we replace $m \leftarrow \lceil 2n/3 \rceil$ with $m \leftarrow \lfloor 2n/3 \rfloor$. Prove your answer.

(d) Prove that the number of swaps executed by the algorithm is at most $\binom{n}{2}$.

For Problem 8 and Problem 9, you only need to solve one of them. Pick the one you prefer! Nevertheless, you are welcome to submit solution for both. 😜

Problem 8 [Randomized Algorithm]

The following algorithm finds the second smallest element in an unsorted array:

```
RANDOM2NDMIN($A[1, \cdots, n]$)
1: $min1 \leftarrow \infty, min2 \leftarrow \infty$.
2: for ($i = 1$ to $n$ in random order) do
3:     if ($A[i] < min1$) then
4:         $min2 \leftarrow min1$.
5:         $min1 \leftarrow A[i]$.
6:     else if ($A[i] < min2$) then
7:         $min2 \leftarrow A[i]$.
8: return $min2$.
```

(a) How many times does the algorithm execute line 4 in the worst-case?

(b) What is the probability that line 4 is executed during the n^{th} iteration of the loop?

(c) What is the exact expected number of executions of line 4?

(d) How many times does the algorithm execute line 7 in the worst-case?

(e) What is the probability that line 7 is executed during the n^{th} iteration of the loop?

(f) What is the exact expected number of executions of line 7?

(Hint: You may find "linearity of expectation" helpful when solving (c) and (f).)

Problem 9 [Divide-and-Conquer Algorithm]

In this problem we will develop a divide-and-conquer algorithm for the following geometric task.

The Closest Pair Problem:
Input: A set of n points in the plane, $\{p_1 = (x_1, y_1), p_2 = (x_2, y_2), \cdots, p_n = (x_n, y_n)\}$.
Output: The closest pair of points. That is, the pair $p_i \neq p_j$ that minimizes $\sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$.

For simplicity, assume that n is a power of two, and that all the x-coordinates x_i are distinct, as are the y-coordinates. Here's a high-level overview of the algorithm:

- Find a value x for which exactly half the points have $x_i < x$, and half have $x_i > x$. On this basis, split the points into two groups, L and R.
- Recursively find the closest pair in L and in R. Say these pairs are $p_L, q_L \in L$ and $p_R, q_R \in R$, with distances d_L and d_R respectively. Let d be the smaller of these two distances.
- It remains to be seen whether there is a point in L and a point in R that are less than distance d apart from each other. To this end, discard all points with $x_i < x - d$ or $x_i > x + d$ and sort the remaining points by y-coordinate.
- Now, go through this sorted list, and for each point, compute its distance to the *seven* subsequent points in the list. Let p_M, q_M be the closest pair found in this way.
- The answer is one of the three pairs $\{p_L, q_L\}, \{p_R, q_R\}, \{p_M, q_M\}$, whichever is closest.

(a) Prove that the algorithm is correct. *(Hint: You may find the following property helpful: any square of size $d \times d$ in the plane contains at most four points of L.)*

(b) Write down the pseudocode for the algorithm and argue its running time is $O(n \lg^2 n)$.

(c) Can you improve the algorithm so that its running time is reduced to $O(n \lg n)$?

Problem Set 4

Data Structures and Algorithms, Fall 2020

Due: October 15, in class.

Problem 1

Suppose that you are given a sequence of n elements to sort. The input sequence consists of n/k subsequences, each containing k elements. The elements in a given subsequence are all smaller than the elements in the succeeding subsequence and larger than the elements in the preceding subsequence. Thus, all that is needed to sort the whole sequence of length n is to sort the k elements in each of the n/k subsequences.

(a) Bob claims an $\Omega(n \lg k)$ lower bound on the number of comparisons needed to solve this variant of the sorting problem. The basis idea of his proof is: "any comparison based sorting algorithm needs $\Omega(k \lg k)$ comparisons to sort k elements; since we need to sort n/k subsequences each containing k elements, we need $(n/k) \cdot \Omega(k \lg k) = \Omega(n \lg k)$ comparisons." Unfortunately, this proof is invalid. Can you see why?

(b) Interestingly enough, indeed $\Omega(n \lg k)$ comparisons are needed to solve this variant of the sorting problem. Can you provide a valid proof for this claim?

Problem 2

(a) Develop an algorithm that, given n integers in the range 0 to k, preprocesses its input and then answers any query about how many of the n integers fall into a range $[a, b]$ in $O(1)$ time. Here, $a \in \mathbb{N}$ and $b \in \mathbb{N}$. Your algorithm should use $O(n + k)$ time for preprocessing.

(b) You are given an array of integers, where different integers may have different numbers of digits, but the total number of digits over all the integers is n. Develop an algorithm to sort the array in $O(n)$ time.

Problem 3

Suppose that, instead of sorting an array, we just require that the elements increase on average. More precisely, we call an n-element array A *k-sorted* if, for all $1 \leq i \leq n - k$, the following holds:

$$\frac{\sum_{j=i}^{i+k-1} A[j]}{k} \leq \frac{\sum_{j=i+1}^{i+k} A[j]}{k}$$

(a) Prove that an n-element array is k-sorted if and only if $A[i] \leq A[i + k]$ for all $1 \leq i \leq n - k$.

(b) Give an algorithm that k-sorts an n-element array in $O(n \lg (n/k))$ time.

(c) Give an algorithm that sorts a k-sorted array of length n in $O(n \lg k)$ time.

(d) Prove that when k is a constant, k-sorting an n-element array requires $\Omega(n \lg n)$ time.

Problem 4

Assume n is a power of 2. Develop an algorithm that finds the second smallest of n elements using at most $n + \lg n - 2$ comparisons. Argue your algorithm indeed uses at most $n + \lg n - 2$ comparisons.

Problem 5

(a) In the algorithm QUICKSELECT we introduced in class, the input elements are divided into groups of 5. Will the algorithm work in linear time if they are divided into groups of 7? What about groups of 3? You need to justify your answer.

(b) Assume n and k are both some power of 2. The k^{th} quantiles of an n-element set are the $k-1$ order statistics that divide the sorted set into k equal-sized sets. Develop an $O(n \lg k)$ time algorithm to list the k^{th} quantiles of a set. That is, for every $1 \leq i \leq k-1$, find the $(in/k)^{\text{th}}$ order statistic of the set.

Problem 6

For n distinct elements $x_1, x_2, \cdots, x_n$ with positive weights $w_1, w_2, \cdots, w_n$ such that $\sum_{i=1}^{n} w_i = 1$, the *weighted median* is the element x_k satisfying $\sum_{x_i < x_k} w_i < 1/2$ and $\sum_{x_i > x_k} w_i \leq 1/2$.

For example, if the elements are 0.1, 0.35, 0.05, 0.1, 0.15, 0.05, 0.2 and each element equals its weight (that is, $w_i = x_i$ for all i), then the median is 0.1, but the weighted median is 0.2.

Develop an algorithm that computes the weighted median in $O(n)$ worst-case time.

Problem 7

The pre/in/post-order numbering of a binary tree labels the nodes of a binary tree with the integers $0, \cdots, n-1$ in the order that they are encountered by a pre/in/post-order traversal. (See Figure 1.)

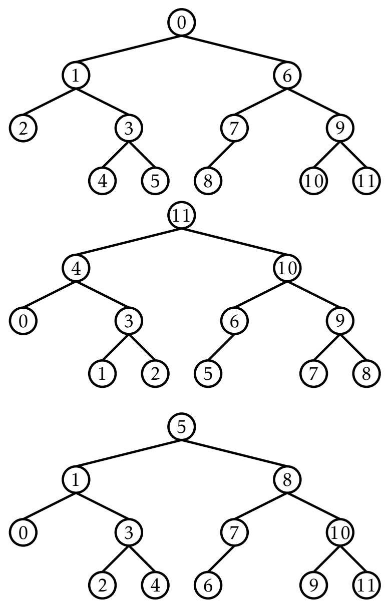


Figure 1: Pre-order, post-order, and in-order numberings of a binary tree.

Suppose you are given a set of nodes with pre-order and in-order numbers assigned. Prove there is at most one binary tree with this pre/in-order numbering and develop an algorithm to construct that tree.

*Problem 8 [Bonus Problem]

(a) Suppose we are given a full binary tree with pre-, post-, and in-order numbers assigned to the nodes. (That is, for each node, there is a data field storing the pre-, post-, and in-order numbers of this node. Also, recall that a binary tree is full if each node has either zero or two children.) Given a node u, show how these numbers can be used to determine the size of the subtree rooted at u, in $O(1)$ time.

(b) Show that the shape of any binary tree on n nodes can be represented using at most $2n+1$ bits.

Problem Set 5

Data Structures and Algorithms, Fall 2020

Due: October 22, in class.

Problem 1

(a) Suppose that we have numbers between 1 and 1000 in a binary search tree, and we want to search for the number 363. Which of the following sequences could *not* be the sequence of nodes examined?

- 2, 252, 401, 398, 330, 344, 397, 363.
- 924, 220, 911, 244, 898, 258, 362, 363.
- 925, 202, 911, 240, 912, 245, 363.
- 2, 399, 387, 219, 266, 382, 381, 278, 363.
- 935, 278, 347, 621, 299, 392, 358, 363.

(b) Professor Bunyan thinks he has discovered a remarkable property of binary search trees. Suppose that the search for key k in a binary search tree ends up in a leaf. Consider three sets: A, the keys to the left of the search path; B, the keys on the search path; and C, the keys to the right of the search path. (That is, $A = (\cup_{b \in B}\{\text{nodes in the left subtree of } b\}) \setminus B$ and $C = (\cup_{b \in B}\{\text{nodes in the right subtree of } b\}) \setminus B$.) Professor Bunyan claims that any three keys $a \in A$, $b \in B$, and $c \in C$ must satisfy $a \leq b \leq c$ (if there are such a, b, and c). Do you think the claim is true? You need to justify your answer.

Problem 2

Recall that for each node x in a binary search tree, it keeps a pointer $x.left$ to its left child, a pointer $x.right$ to its right child, and a pointer $x.p$ to its parent. Suppose that instead of each node x keeping the pointer $x.p$, it keeps $x.succ$, pointing to x's successor. Give pseudocode for SEARCH, INSERT, and DELETE on a binary search tree T using this representation. These procedures should operate in time $O(h)$, where h is the height of the tree T.

Problem 3

Equal keys pose a problem for the implementation of binary search trees.

(a) Recall the TREEINSERT procedure introduced in Section 12.3 in the textbook CLRS. What is the asymptotic total runtime of TREEINSERT when used to insert n items with identical keys into an initially empty binary search tree?

We propose to improve TREEINSERT by testing before line 5 to determine whether $z.key = x.key$ and by testing before line 11 to determine whether $z.key = y.key$. If equality holds, we implement one of the following strategies. For each strategy, find the asymptotic total runtime of inserting n items with identical keys into an initially empty binary search tree. (The strategies are described for line 5, in which we compare the keys of z and x. Substitute y for x to arrive at the strategies for line 11.)

(b) Keep a boolean flag $x.b$ at node x, and set x to either $x.left$ or $x.right$ based on value of $x.b$, which alternates between FALSE and TRUE each time we visit x while inserting a node with same key as x.

(c) Keep a list of nodes with equal keys at x, and insert z into the list.

Problem 4

(a) Describe a red-black tree that realizes the largest possible ratio of red internal nodes to black internal nodes. What is this ratio? What tree has the smallest possible ratio, and what is the ratio? (You do *not* need to prove the described trees have the largest or the smallest ratio.)

(b) Argue that an arbitrary n-node binary search tree can be transformed into any other arbitrary n-node binary search tree using $O(n)$ rotations. *(Hint: First show that at most $n-1$ right rotations suffice to transform the tree into a right-going chain.)*

Problem 5

(a) Show the red-black trees that result after successively inserting the keys 41, 38, 31, 12, 19, 8 into an initially empty red-black tree. Note, you need to show the red-black tree after *each* insertion.

(b) Following part (a), now show the red-black trees that result from the successive deletion of the keys in the order 8, 12, 19, 31, 38, 41. Again, you need to show the red-black tree after *each* deletion.

Problem 6

An *AVL tree* (named after inventors Adelson-Velsky and Landis) is a binary search tree that is height balanced: for each node x, the heights of the left and right subtrees of x differ by at most 1. To implement an AVL tree, we maintain an extra attribute in each node: $x.h$ is the height of node x. As for any other binary search tree T, we assume that $T.root$ points to the root node.

(a) Prove that an AVL tree with n nodes has height $O(\log n)$. *(Hint: Prove that an AVL tree of height h has at least F_h nodes, where F_h is the h^{th} Fibonacci number.)*

(b) To insert into an AVL tree, first place a node into the appropriate place in binary search tree order. Now, the tree might no longer be height balanced: the heights of the left and right children of some node might differ by 2. Describe a procedure BALANCE(x), which takes a subtree rooted at x whose left and right children are height balanced and have heights that differ by at most 2, i.e., $|x.right.h - x.left.h| \leq 2$, and alters the subtree rooted at x to be height balanced. *(Hint: Use rotations.)*

(c) Using part (b), describe a recursive procedure AVLINSERT(x, z) that takes a node x within an AVL tree and a newly created node z (whose key has already been filled in), and adds z to the subtree rooted at x, maintaining the property that x is the root of an AVL tree. As in TREEINSERT from the textbook, assume that $z.key$ has already been filled in and that $z.left = NULL$ and $z.right = NULL$; also assume that $z.h = 0$. Thus, to insert the node z into the AVL tree T, we call AVLINSERT$(T.root, z)$.

(d) Argue that AVLINSERT, run on an n-node AVL tree, takes $O(\lg n)$ time and performs $O(1)$ rotations.

Problem 7

(a) Assume you are given a size n sorted array $A = (x_1, \cdots, x_n)$. Design an algorithm that builds a random treap containing elements in A in $O(n)$ worst-case time.

(b) Recall that we say an event happens "with high probability (in n)" if it happens with probability at least $1 - 1/n$. Prove the following basic facts about skip lists, where n is the number of keys.

1. The max level is $O(\log n)$ with high probability.
2. The number of nodes is $O(n)$ in expectation.
3. **(Bonus Question)** The number of nodes is $O(n)$ with high probability.

(Hint: You may need to apply concentration inequalities like Chernoff bounds to solve the bonus question. You can learn more about these concentration inequalities from Chapter 4 of the "Probability and Computing" book [original version, translated version].)

Problem Set 6

Data Structures and Algorithms, Fall 2020

Due: October 29, in class.

Problem 1

Suppose we have stored n keys in a hash table of size m, with collisions resolved by chaining, and that we know the length of each chain, including the length L of the longest chain. Describe a procedure that selects a key uniformly at random from among the keys in the hash table and returns it in expected time $O(L \cdot (1 + 1/\alpha))$. Here, α is the load factor. You may assume there exists a function $\textsc{Random}(x, y)$ that returns an integer chosen uniformly at random from $[x, y]$, in $O(1)$ time.

Problem 2

(a) Consider a version of the division method in which $h(k) = k \mod m$, where $m = 2^p - 1$, k is a character string interpreted in radix 2^p, and $p > 1$ is an integer. (For example, if we use the 7-bit ASCII encoding, then $p = 7$ and string AB has key value $65 \times 128 + 66$.) Show that if we can derive string x from string y by permuting its characters, then x and y hash to the same value.

(b) Consider inserting the keys 10,22,31,4,15,28,17,88,59 into a hash table of length $m = 11$ using open addressing with the auxiliary hash function $h'(k) = k$. Illustrate the result of inserting these keys using linear probing, using quadratic probing with $c_1 = 1$ and $c_2 = 3$, and using double hashing with $h_1(k) = k$ and $h_2(k) = 1 + (k \mod (m - 1))$. (It suffices to show the eventual hash table.)

Problem 3

Define a family $\mathcal{H}$ of hash functions from a finite set U to a finite set B to be ϵ-universal if for all pairs of distinct elements k and l in U,

$$\Pr[h(k) = h(l)] \leq \epsilon$$

where the probability is over the choice of the hash function h drawn at random from the family $\mathcal{H}$. Show that an ϵ-universal family of hash functions must have

$$\epsilon \geq \frac{1}{|B|} - \frac{1}{|U|}$$

Problem 4

Design a data structure to support the following two operations for a dynamic multiset S of integers, which allows duplicate values:

- $\textsc{Insert}(S, x)$ inserts x into S.
- $\textsc{DelLargeHalf}(S)$ deletes the largest $\lceil |S|/2 \rceil$ elements from S.

Explain how to implement this data structure so that, starting from an empty set, any sequence of m $\textsc{Insert}$ and $\textsc{DelLargeHalf}$ operations runs in $O(m)$ time in total. Your implementation should also include a way to output the elements of S in worst-case $O(|S|)$ time.

Problem 5

Suppose that a counter begins at a number with b 1s in its binary representation, rather than at 0. Show that the cost of performing n INC operations is $O(n)$ if $n = \Omega(b)$. Do not assume that b is constant.

Problem 6

To conserve space for a stack, it is proposed to shrink it when its size is some fraction of the number of allocated cells. This supplements the array-doubling strategy for growing it. Assume we stay with the policy that the array size is doubled whenever the stack size grows beyond the current array size. Evaluate each of the following proposed shrinking policies, using amortized costs if possible. Do they offer constant amortized time per stack operation (i.e., push and pop)? You need to justify your answer. (The current array size is denoted as N.)

(a) If a pop results in fewer than $N/2$ stack elements, reduce the array to $N/2$ cells.

(b) If a pop results in fewer than $N/4$ stack elements, reduce the array to $N/4$ cells.

(c) If a pop results in fewer than $N/4$ stack elements, reduce the array to $N/2$ cells.

Problem 7

Consider the following modified algorithm for incrementing a binary counter.

```
INC($B[0, \cdots, \infty]$)
1: $i \leftarrow 0$
2: while ($B[i] = 1$) do
3:     $B[i] \leftarrow 0$
4:     $i \leftarrow i + 1$
5: $B[i] \leftarrow 1$
6: SOMETHINGELSE($i$)
```

The only difference from the standard algorithm is the function call at the end, to a black-box subroutine called SOMETHINGELSE. Suppose we call INC n times, starting with a counter with value 0. The amortized time of each INC clearly depends on the running time of SOMETHINGELSE. Let $T(i)$ denote the worst-case running time of SOMETHINGELSE(i). For example, we proved in class that INC algorithm runs in $O(1)$ amortized time when $T(i) = 0$.

(a) What is the amortized time per INC if $T(i) = 4$? Justify your answer.

(b) What is the amortized time per INC if $T(i) = 2^i$? Justify your answer.

(c) What is the amortized time per INC if $T(i) = 4^i$? Justify your answer.

Problem Set 7

Data Structures and Algorithms, Fall 2020

Due: November 5, in class.

Problem 1

(a) Give a *tight* asymptotic bound on the running time of the following sequence of operations, assuming the linked-list representation and the weighted-union heuristic. You need to justify your answer.

$$\text{MAKESET}(x_1), \cdots, \text{MAKESET}(x_n), \text{UNION}(x_2, x_1), \text{UNION}(x_3, x_2), \cdots, \text{UNION}(x_n, x_{n-1})$$

(b) Suppose you are given a collection of trees representing a partition of the set $\{1, 2, \cdots, n\}$ into disjoint subsets. You have no idea how these trees were constructed. You are also given an array $node[1, \cdots, n]$, where $node[i]$ is a pointer to the tree node containing element i. You can create a new array $label[1, \cdots, n]$ using the following algorithm. Prove that if we implement FIND using path compression, LABELEVERYTHING runs in $O(n)$ time in the worst case.

```
LABELEVERYTHING
1: for ($i \leftarrow 1$ to $n$) do
2:     $label[i] \leftarrow$ FIND($node[i]$)
```

Problem 2

Suppose we want to maintain an array $X[1, \cdots, n]$ of bits, which are all initially zero, subject to the following operations.

- LOOKUP(i): Given an index i, return $X[i]$.
- BLACKEN(i): Given an index $i < n$, set $X[i] \leftarrow 1$.
- NEXTWHITE(i): Given an index i, return the smallest index $j \geq i$ such that $X[j] = 0$. (Because we never change $X[n]$, such an index always exists.)

If we use the array $X[1, \cdots, n]$ itself as the only data structure, it is trivial to implement LOOKUP and BLACKEN in $O(1)$ time and NEXTWHITE in $O(n)$ time. But you can do better! Describe data structures that support LOOKUP in $O(1)$ worst-case time and the other two in the following time bounds.

(a) The amortized time for BLACKEN is $O(\log n)$, and the worst-case time for NEXTWHITE is $O(1)$.

(b) The worst-case time for BLACKEN is $O(\log n)$, and the amortized time for NEXTWHITE is $O(\log^* n)$.

(c) [Bonus Question] BLACKEN in $O(1)$ worst-case, and NEXTWHITE in $O(\log^* n)$ amortized.

Problem 3

Given an adjacency matrix for a directed graph $G = (V, E)$, design an algorithm that determines whether G contains a vertex satisfying the following property: the vertex has in-degree $|V| - 1$ and out-degree 0. To get full credit, your algorithm should only take $O(|V|)$ time.

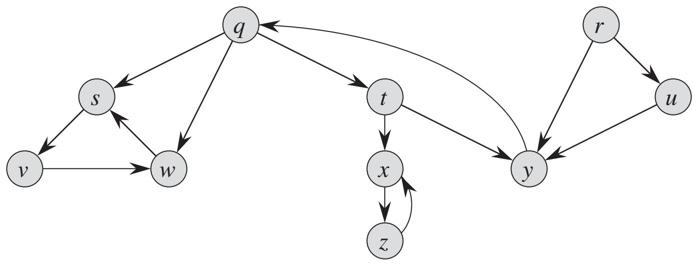
Problem 4

(a) Argue that in a breadth-first search, the value $u.d$ assigned to a vertex u is independent of the order in which the vertices appear in each adjacency list. Nonetheless, show (via example) that the breadth-first tree computed by BFS can depend on the ordering within adjacency lists.

(b) Give an example of a directed graph $G = (V, E)$, a source vertex $s \in V$, and a set of tree edges $E_\pi \subseteq E$ such that for each vertex $v \in V$, the unique simple path in the graph (V, E_π) from s to v is a shortest path in G, yet the set of edges E_π cannot be produced by running BFS on G, no matter how the vertices are ordered in each adjacency list.

Problem 5

(a) Show how DFS works on the following graph. Specifically, assume that the DFS procedure considers the vertices in alphabetical order, and assume that each adjacency list is ordered alphabetically. Show the discovery and finishing times for each vertex, and show the classification of each edge.



(b) Give a counterexample to the conjecture that if a directed graph G contains a path from u to v, and if $u.d < v.d$ in a depth-first search of G, then v is a descendant of u in the depth-first forest produced.

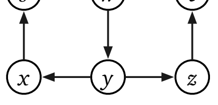
Problem 6

Modify the pseudocode for depth-first search so that it prints out every edge in the directed graph G, together with its type.

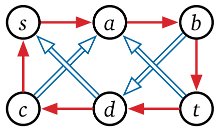
Problem 7

[Choose either one of the following two problems, but you are welcome to submit solutions for both.]

(a) Suppose you are given a directed graph $G = (V, E)$ and two vertices s and t. Describe and analyze an algorithm to determine if there is a walk in G from s to t (possibly repeating vertices and/or edges) whose length is divisible by 3. For example, given the graph shown below, your algorithm should return `True`, because the walk $s \to w \to y \to x \to s \to w \to t$ has length 6. To get full credit, your algorithm should run in $O(|V| + |E|)$ time.



(b) Suppose you are given a directed graph G where some edges are red and the remaining edges are blue. Describe and analyze an algorithm to find the shortest walk in G from one vertex s to another vertex t in which no three consecutive edges have the same color. That is, if the walk contains two red edges in a row, the next edge must be blue, and if the walk contains two blue edges in a row, the next edge must be red. For example, given the following graph as input, your algorithm should return the integer 7, because $s \rightarrow a \rightarrow b \Longrightarrow d \rightarrow c \Longrightarrow a \rightarrow b \rightarrow t$ is the shortest legal walk from s to t. To get full credit, your algorithm should run in $O(|V| + |E|)$ time.



Problem Set 8

Data Structures and Algorithms, Fall 2020

Due: November 19, in class.

Problem 1

One way to perform topological sorting on a directed acyclic graph $G = (V, E)$ is to repeatedly find a vertex of in-degree 0, output it, and remove it and all of its outgoing edges from the graph. Assuming we use adjacency-list representation, explain how to implement this idea so that it runs in time $O(|V|+|E|)$.

Problem 2

(a) Give an $O(|V| + |E|)$ time algorithm to compute the component graph of a directed graph $G = (V, E)$. Make sure that there is at most one edge between two vertices in the component graph your algorithm produces.

(b) Consider the two-pass SCC algorithm we introduced in class. Professor Bacon claims that the algorithm would still work correctly if it used the transpose graph in the second depth-first search and scanned the vertices in order of increasing finishing times. Do you agree with Professor Bacon? You need to justify your answer.

Problem 3

You are given a directed graph $G = (V, E)$ in which each node $u \in V$ has an associated price p_u which is a positive integer. Define the array `cost` as follows: for each $u \in V$, `cost[u]` is the price of the cheapest node reachable from u (including u itself). Your goal is to design an algorithm that fills in the entire `cost` array (i.e., for all vertices).

(a) Give an $O(|V| + |E|)$ time algorithm that works for directed acyclic graphs.

(b) Give an $O(|V| + |E|)$ time algorithm that works for all directed graphs.

Problem 4 [Bonus Problem]

A directed graph $G = (V, E)$ is "sort-of-connected" if, for every pair of vertices u and v, either u is reachable from v or v is reachable from u (or both). Give an $O(|V| + |E|)$ time algorithm to determine whether a directed graph is sort-of-connected. To get full credit, also prove your algorithm is correct.

Problem 5

(a) Let $G = (V, E)$ be a connected, undirected graph with a real-valued weight function w defined on E. Let A be a subset of E that is included in some minimum spanning tree for G, let $(S, V - S)$ be any cut of G that respects A, and let (u, v) be a safe edge for A crossing $(S, V - S)$. Professor Bacon claims (u, v) is a light edge for the cut. Do you agree with Professor Bacon? You need to justify your answer.

(b) Professor Bacon proposes a new divide-and-conquer algorithm for computing minimum spanning trees, which goes as follows. Given a graph $G = (V, E)$, partition the set V of vertices into two sets V_1 and V_2 such that $|V_1|$ and $|V_2|$ differ by at most 1. Let E_1 be the set of edges that are incident only on vertices in V_1, and let E_2 be the set of edges that are incident only on vertices in V_2. Recursively solve a minimum-spanning-tree problem on each of the two subgraphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$. Finally, select the minimum-weight edge in E that crosses the cut (V_1, V_2), and use this edge to unite the resulting two minimum spanning trees into a single spanning tree. Do you think Professor Bacon's algorithm is correct? You need to justify your answer.

Problem 6

(a) Let $G = (V, E)$ be an undirected graph. Prove that if all its edge weights are distinct, then it has a unique minimum spanning tree.

(b) Let T be a minimum spanning tree of a graph $G = (V, E)$, and let V' be a subset of V. Let T' be the subgraph of T induced by V', and let G' be the subgraph of G induced by V'. Show that if T' is connected, then T' is a minimum spanning tree of G'.

Problem 7

Let $G = (V, E)$ be an undirected, connected graph whose weight function is $w : E \to \mathbb{R}$, and suppose that all edge weights are distinct. We define a second-best minimum spanning tree as follows. Let $\mathcal{T}$ be the set of all spanning trees of G, and let T' be a minimum spanning tree of G. Then a second-best minimum spanning tree is a spanning tree T such that $w(T) = \min_{T'' \in \mathcal{T} - \{T'\}} \{w(T'')\}$.

(a) Let T be the minimum spanning tree of G. Prove that G contains edges $(u, v) \in T$ and $(x, y) \notin T$ such that $T - \{(u, v)\} \cup \{(x, y)\}$ is a second-best minimum spanning tree of G.

(b) Let T be a spanning tree of G and, for any two vertices u and v, let `max(u,v)` denote an edge of maximum weight on the unique simple path between u and v in T. Describe an $O(|V|^2)$ time algorithm that, given T, computes `max(u,v)` for all (u, v) pairs.

(c) Give an $O(|V|^2 + |E| \lg |V|)$ time algorithm to compute a second-best minimum spanning tree of G.

Problem Set 9

Data Structures and Algorithms, Fall 2020

Due: November 26, in class.

Problem 1

Suppose that all edge weights in a graph are integers in the range from 1 to $|V|$. How fast can you make Kruskal's algorithm run? What if the edge weights are integers in the range from 1 to W for some constant W? You need to describe your algorithms and analyze their time complexities.

Problem 2

You are given a graph $G = (V, E)$ with positive edge weights, and an MST $T = (V, E')$ with respect to these weights. You may assume G and T are given as adjacency lists. Now suppose the weight of a particular edge $e \in E$ is modified from $w(e)$ to a new value $\hat{w}(e)$. You wish to quickly update the MST T to reflect this change, without recomputing the entire tree from scratch. There are four cases. In each case give an $O(|V| + |E|)$ time algorithm for updating the tree.

(a) $e \notin E'$ and $\hat{w}(e) > w(e)$.

(b) $e \notin E'$ and $\hat{w}(e) < w(e)$.

(c) $e \in E'$ and $\hat{w}(e) < w(e)$.

(d) $e \in E'$ and $\hat{w}(e) > w(e)$. *[For this part, you will get full credit if you can find an $O((|V| + |E|) \lg |V|)$ time algorithm; and you will get bonus credit for finding an $O(|V| + |E|)$ time algorithm.]*

Problem 3

Not just any greedy approach to the activity-selection problem produces a maximum-size set of mutually compatible activities. Give an example to show that the approach of selecting the activity of least duration from among those that are compatible with previously selected activities does not work. Do the same for the approaches of always selecting the compatible activity that overlaps the fewest other remaining activities and always selecting the compatible remaining activity with the earliest start time.

Problem 4

Suppose that in a 0-1 knapsack problem, the order of the items when sorted by increasing weight is the same as their order when sorted by decreasing value. Give an efficient greedy algorithm to find an optimal solution to this variant of the knapsack problem, and prove that your algorithm is correct.

Problem 5

Suppose that a data file contains a sequence of 8-bit characters such that all 256 characters are about equally common: the maximum character frequency is less than twice the minimum character frequency. Prove that Huffman coding in this case is no more efficient than using an ordinary 8-bit fixed-length code.

Problem 6

Let X be a set of n intervals on the real line. A proper coloring of X assigns a color to each interval, so that any two overlapping intervals are assigned different colors. Devise a greedy algorithm to compute the minimum number of colors needed to properly color X. Assume that your input consists of two arrays $L[1 \cdots n]$ and $R[1 \cdots n]$, representing the left and right endpoints of the intervals in X. You need to argue the correctness of your algorithm and analyze its time complexity.

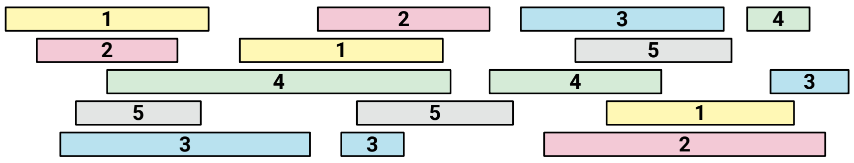


Figure 1: A proper coloring of a set of intervals using five colors.

Problem 7

One day Alex got tired of climbing in a gym and decided to take a large group of climber friends outside to climb. They went to a climbing area with a huge wide boulder, not very tall, with several marked hand and foot holds. Alex quickly determined an "allowed" set of moves that her group of friends can perform to get from one hold to another.

The overall system of holds can be described by a rooted tree T with n vertices, where each vertex corresponds to a hold and each edge corresponds to an allowed move between holds. The climbing paths converge as they go up the boulder, leading to a unique hold at the summit, represented by the root of T.

Alex and her friends decided to play a game, where as many climbers as possible are simultaneously on the boulder and each climber needs to perform a sequence of exactly k moves. Each climber can choose an arbitrary hold to start from, and all moves must move away from the ground. Thus, each climber traces out a path of k edges in the tree T, all directed toward the root. However, no two climbers are allowed to touch the same hold; the paths followed by different climbers cannot intersect at all.

Describe a greedy algorithm to compute the maximum number of climbers that can play this game, and analyze the correctness and time complexity of your algorithm. Your algorithm is given a rooted tree T and an integer k as input, and it should compute the largest possible number of disjoint paths in T, where each path has length k. Do not assume that T is a binary tree. For example, given the tree below as input, your algorithm should return the integer 8.

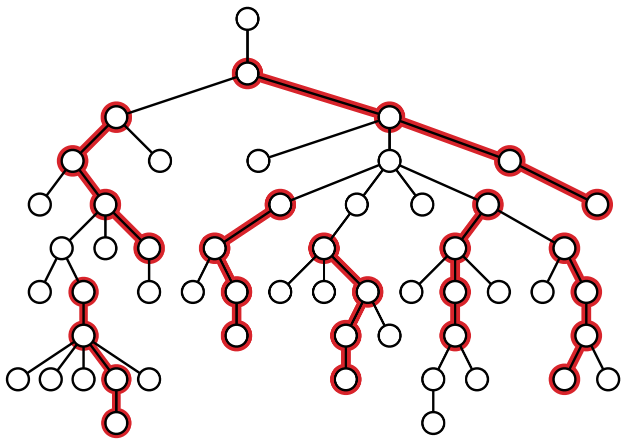


Figure 2: Seven disjoint paths of length $k = 3$. This is *not* the largest such set of paths in this tree.

Problem Set 10

Data Structures and Algorithms, Fall 2020

Due: Dec 3, in class.

Problem 1

Suppose there are n people with the sizes of their feet given in an array $P[1 \cdots n]$, and n pairs of shoes with sizes given in an array $S[1 \cdots n]$. Design an algorithm to assign a pair of shoes to each person, so that the average difference between the size of a person's feet and the size of his/her assigned pair of shoes is as small as possible. Specifically, the algorithm should compute a permutation $\pi : [1 \cdots n] \to [1 \cdots n]$ such that $(1/n) \sum_{i=1}^{n} |P[i] - S[\pi(i)]|$ is minimized. To get full credit, your algorithm should have time complexity $O(n \log n)$, and you need to prove the correctness of your algorithm.

Bonus Problem

Consider the following generalized set cover problem: Given a universe U of n elements, a collection of subsets of U, $\mathcal{S} = \{S_1, \cdots, S_k\}$, and a cost function $c : \mathcal{S} \to \mathbb{Q}^+$, find a minimum *cost* sub-collection of $\mathcal{S}$ that covers all elements of U.[1]

This problem is suspected to be hard, in the sense that there might not exist a polynomial (with respect to the length of the input) time algorithm that can solve the problem exactly. However, good approximation algorithms do exist for this problem. In particular, if the optimal solution incurs a cost of OPT, we can efficiently find a solution that costs at most $O(OPT \cdot \ln n)$. Can you devise one such algorithm? Prove your algorithm indeed gives a solution that costs at most $O(OPT \cdot \ln n)$, and analyze the runtime of your algorithm. *(Hint: Be Greedy!)*

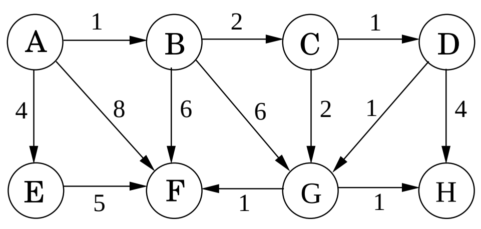
Problem 2

(a) Give a weighted, directed graph $G = (V, E)$ with no negative-weight cycles. Use this graph to demonstrate minimum spanning trees and shortest path trees are not necessarily identical.

(b) Given a weighted, directed graph $G = (V, E)$ with no negative-weight cycles, let m be the maximum over all vertices $v \in V$ of the minimum number of edges in a shortest path from the source s to v. Suggest a simple change to the Bellman-Ford algorithm that allows it to terminate in $m + 1$ passes, even if m is not known in advance.

(c) Suppose Dijkstra's algorithm is run on the following graph, starting at node A. (I) Draw a table showing the intermediate distance values of all the nodes at each iteration of the algorithm. (II) Show the final shortest-path tree.

[1]Recall in class we have discussed another version of the problem, in which the goal is to find the minimum *size* sub-collection of $\mathcal{S}$ that covers all elements of U.



Problem 3

You are given a set of cities, along with the pattern of highways between them, in the form of an undirected graph $G = (V, E)$. Each stretch of highway $e \in E$ connects two of the cities, and you know its length in miles, l_e. You want to get from city s to city t. There's one problem: your car can only hold enough gas to cover L miles. There are gas stations in each city, but not between cities. Therefore, you can only take a route if every one of its edges has length $l_e \leq L$.

(a) Given the limitation on your car's fuel tank capacity, show how to determine in $O(|V| + |E|)$ time whether there is a feasible route from s to t. You need to briefly argue the correctness of your algorithm.

(b) You are now planning to buy a new car, and you want to know the minimum fuel tank capacity that is needed to travel from s to t. Give an $O((|V| + |E|) \log |V|)$ time algorithm to determine this. You need to briefly argue the correctness of your algorithm.

Problem 4

Let $G = (V, E)$ be a weighted, directed graph with weight function w. Assume G has no negative-weight cycles. Give an $O(|V| \cdot |E|)$-time algorithm to find, for each $v \in V$, the value $\min_{u \in V}\{dist(u, v)\}$. Also briefly argue the correctness of your algorithm. *(Hint: Modify the Bellman-Ford algorithm. You cannot solve this problem by using an $O(|V|^3)$-time all-pairs-shortest-path algorithm, as it might be the case that $|E| \in o(|V|^2)$.)*

Problem 5

The PERT chart formulation discussed in class is somewhat unnatural. In a more natural structure, vertices would represent jobs and edges would represent sequencing constraints; that is, edge (u, v) would indicate that job u must be performed before job v. We would then assign weights to vertices instead of edges. In this problem, you are given a DAG graph $G = (V, E)$ with a weight function $w : V \to \mathbb{R}^+$. Assume G has a unique source s and a unique sink t. Design an $O(|V| + |E|)$ time algorithm that computes, for each $v \in V$:

- The earliest start time of the job represented by v, assuming the job represented by s starts at 0.
- The latest start time of the job represented by v, without affecting the project's duration.
- Whether v is in some critical path.

Problem 6

Let $c_1, c_2, \cdots, c_n$ be various currencies; for instance, c_1 might be dollars, c_2 pounds, and c_3 lire. For any two currencies c_i and c_j, there is an exchange rate $r_{i,j}$; this means that you can purchase $r_{i,j}$ units of currency c_j in exchange for one unit of c_i. These exchange rates satisfy the condition that $r_{i,j} \cdot r_{j,i} < 1$, so that if you start with a unit of currency c_i, change it into currency c_j and then convert back to currency c_i, you end up with less than one unit of currency c_i (the difference is the cost of the transaction).

(a) Describe and analyze an efficient algorithm for the following problem: Given a set of exchange rates $r_{i,j}$, and two currencies s and t, find the most advantageous sequence of currency exchanges for converting currency s into currency t.

The exchange rates are updated frequently, reflecting the demand and supply of the various currencies. Occasionally the exchange rates satisfy the following property: there is a sequence of currencies $c_{i_1}, c_{i_2}, \cdots, c_{i_k}$ such that $r_{i_1,i_2} \cdot r_{i_2,i_3}, \cdots \cdot r_{i_{k-1},i_k} \cdot r_{i_k,i_1} > 1$. This means that by starting with a unit of currency c_{i_1} and then successively converting it to currencies $c_{i_2}, c_{i_3}, \cdots, c_{i_k}$ and finally back to c_{i_1}, you would end up with more than one unit of currency c_{i_1}. Such anomalies last only a fraction of a minute on the currency exchange, but they provide an opportunity for risk-free profits.

(b) Describe and analyze an efficient algorithm for detecting the presence of such an anomaly.

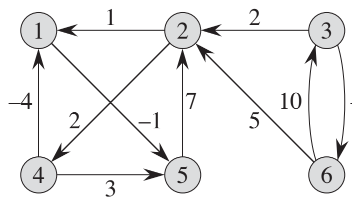
Problem Set 11

Data Structures and Algorithms, Fall 2020

Due: Dec 10, in class.

Problem 1

(a) Run the Floyd-Warshall algorithm on the following weighted, directed graph. Show the $dist$ values for all pairs of nodes after each iteration of the outer-most loop.



(b) How to use the Floyd-Warshall algorithm's output to detect the presence of a negative-weight cycle?

Problem 2

(a) Use Johnson's algorithm to find the shortest paths between all pairs of vertices in the graph used in Problem 1.(a), visualize algorithm's execution via a figure similar to Figure 25.6 in the CLRS textbook.

(b) Suppose that we run Johnson's algorithm on a directed graph G with weight function w. Assume G does not contain negative-weight cycles. Show that if G contains a 0-weight cycle c, then $\hat{w}(u, v) = 0$ for every edge (u, v) in c.

Problem 3

Suppose that we wish to maintain the transitive closure of a directed graph $G = (V, E)$ as we insert edges into E. That is, after each edge has been inserted, we want to update the transitive closure of the edges inserted so far. Assume that the graph G has no edges initially and that we represent the transitive closure as a boolean matrix.

(a) Show how to update the transitive closure of a graph $G = (V, E)$ in $O(|V|^2)$ time when a new edge is added to G.

(b) Give an example of a graph G and an edge e such that $\Omega(|V|^2)$ time is required to update the transitive closure after the insertion of e into G, no matter what algorithm is used.

(c) Describe an algorithm for updating the transitive closure as edges are inserted into the graph. For any sequence of x insertions, your algorithm should run in total time $\sum_{i=1}^{x} t_i = O(|V|^3)$, where t_i is the time to update the transitive closure upon inserting the ith edge. Prove that your algorithm attains this time bound.

Problem 4

(a) Describe the subproblem graph for matrix-chain multiplication with an input chain of length n. How many vertices does it have? How many edges does it have, and which edges are they?

(b) Draw the recursion tree for the MergeSort algorithm on an array of 16 elements. Explain why memoization fails to speed up a good divide-and-conquer algorithm such as MergeSort.

(c) Imagine that you wish to exchange one currency for another. You realize that instead of directly exchanging one currency for another, you might be better off making a series of trades through other currencies, winding up with the currency you want. Suppose that you can trade n different currencies, numbered $1 \cdots n$, where you start with currency 1 and wish to wind up with currency n. You are given, for each pair of currencies i and j, an exchange rate r_{ij}, meaning that if you start with d units of currency i, you can trade for dr_{ij} units of currency j. A sequence of trades may entail a commission, which depends on the number of trades you make. Let c_k be the commission that you are charged when you make k trades. Show that if commissions c_k are arbitrary values, then the problem of finding the best sequence of exchanges from currency 1 to currency n does not necessarily exhibit optimal substructure.

Problem 5

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 of coins.

(b) Suppose El Generalissimo decides to impose a currency system where the coin denominations are consecutive powers $b^0, b^1, b^2, \cdots, b^k$ of some integer $b \geq 2$. Prove that despite El Generalissimo's disapproval, the greedy algorithm described in part (a) does make optimal change in this currency system.

(c) Design an algorithm to determine, given a target amount T and a sorted array $c[1 \cdots n]$ of coin denominations, the smallest number of coins needed to make T cents in change. Assume that $c[1] = 1$, so that it is possible to make change for any amount T. Remember to analyze the time complexity of your proposed algorithm. For full credit, your algorithm should have $O(nT)$ runtime.

Problem 6

A palindrome is any string that is exactly the same as its reversal, like `I`, or `DEED`, or `RACECAR`, or `AMANAPLANACATACANALPANAMA`.

(a) Describe and analyze an algorithm to find the length of the *longest subsequence* of a given string that is also a palindrome. For example, the longest palindrome subsequence of the string

`MAHDYNAMICPROGRAMZLETMESHOWYOUTHEM`

is `MHYMRORMYHM`; thus, given that string as input, your algorithm should return 11. For full credit, your algorithm should have $O(n^2)$ runtime for an input string of length n.

(b) Describe and analyze an algorithm to find the length of the *shortest supersequence* of a given string that is also a palindrome. For example, the shortest palindrome supersequence of `TWENTYONE` is `TWENTOYOTNEWT`, so given the string `TWENTYONE` as input, your algorithm should return 13. For full credit, your algorithm should have $O(n^2)$ runtime for an input string of length n.

Problem 7

This problem asks you to devise algorithms to compute optimal subtrees in *unrooted* trees—connected acyclic undirected graphs. In this problem. a *subtree* of an unrooted tree is any connected subgraph.

(a) Suppose you are given an unrooted tree T with weights on its *edges*, which may be positive, negative, or zero. Describe and analyze an algorithm to find a *path* in T with maximum total weight. Your algorithm only need to return the weight value. You may assume a path only contains distinct vertices. For full credit, your algorithm should have $O(n)$ runtime assuming T contains n vertices.

(b) Suppose you are given an unrooted tree T with weights on its *vertices*, which may be positive, negative, or zero. Describe and analyze an algorithm to find a subtree of T with maximum total weight. Your algorithm only need to return the weight value. For full credit, your algorithm should have $O(n)$ runtime assuming T contains n vertices. (This was a 2016 Google interview question.)

(We only ask for returning the weight value so as to keep the code cleaner. Nonetheless, you should also think about how to efficiently reconstruct the optimal solution. It's another good exercise!)

Problem Set 12

Data Structures and Algorithms, Fall 2020

Due: Dec 17, in class.

Problem 1

After graduating from university, you decide to interview for a position at the Wall Street bank **Long Live Boole**. The managing director of the bank, Eloob Egroeg, poses a 'solve-or-die' problems to each new employee, which they must solve within 1 hour. Those who fail to solve the problem are fired!

Entering the bank for the first time, you notice that the employee offices are organized in a straight row, with a large T or F printed on the door of each office. Furthermore, between each adjacent pair of offices, there is a board marked by one of the symbols $\wedge$, $\vee$, or $\oplus$. When you ask about these arcane symbols, Eloob confirms that T and F represent the boolean values TRUE and FALSE, and the symbols on the boards represent the standard boolean operators AND, OR, and XOR. He also explains that these letters and symbols describe whether certain combinations of employees can work together successfully. At the start of any new project, Eloob hierarchically clusters his employees by adding parentheses to the sequence of symbols, to obtain an unambiguous boolean expression. The project is successful if this parenthesized boolean expression evaluates to T.

For example, if the bank has three employees, and the sequence of symbols on and between their doors is $T \wedge F \oplus T$, there is exactly one successful parenthesization scheme: $(T \wedge (F \oplus T))$. However, if the list of door symbols is $F \wedge T \oplus F$, there is no way to add parentheses to make the project successful.

Eloob finally poses his question: Describe an algorithm to decide whether a given sequence of symbols can be parenthesized so that the resulting boolean expression evaluates to T. Your input is an array $S[0 \cdots 2n]$, where $S[i] \in \{T, F\}$ when i is even, and $S[i] \in \{\wedge, \vee, \oplus\}$ when i is odd.

Problem 2

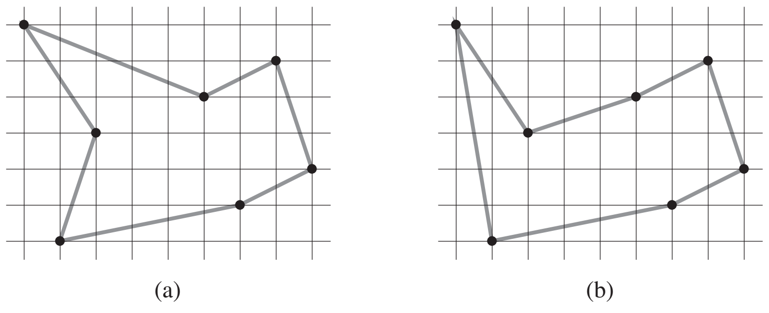
Suppose you are given an array $A[1 \cdots n]$ of numbers, which may be positive, negative, or zero, and which are not necessarily integers. Describe and analyze an algorithm that finds the largest product of elements in a contiguous subarray $A[i \cdots j]$.

For example, given the array $[-6, 12, -7, 0, 14, -7, 5]$ as input, your algorithm should return 504; and given the one-element array $[-374]$ as input, your algorithm should return 1. (The empty interval is still an interval!) For the sake of analysis, assume that comparing, adding, or multiplying any pair of numbers takes $O(1)$ time. For full credit, your algorithm should have $O(n)$ time complexity and $O(1)$ space complexity.

Problem 3

Suppose we are given a set of n points in the plane, and we wish to find the shortest closed tour that connects all n points. Part (a) of the following figure shows the solution to a 7-point problem. The general problem is NP-hard, and its solution is therefore believed to require more than polynomial time.

We can simplify the problem by restricting our attention to *bitonic tours*, that is, tours that start at the leftmost point, go strictly rightward to the rightmost point, and then go strictly leftward back to the starting point. Part (b) of the following figure shows the shortest bitonic tour of the same 7 points. For finding optimal bitonic tours, a polynomial-time algorithm is possible.


(a) (b)

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 and that all operations on real numbers take unit time. *(Hint: Scan left to right, maintaining optimal possibilities for the two parts of the tour.)*

Problem 4

(a) Give a dynamic-programming solution to the 0-1 knapsack problem that runs in $O(nW)$ time, where n is the number of items and W is the maximum weight of items that the thief can put in his knapsack.

(b) Is the dynamic-programming algorithm you designed in part (a) a polynomial-time algorithm? Justify your answer.

Problem 5

Show that the class $\mathbf{P}$, viewed as a set of languages, is closed under union, intersection, concatenation, complement, and Kleene star. That is, if $L_1, L_2 \in \mathbf{P}$, then $L_1 \cup L_2 \in \mathbf{P}$, $L_1 \cap L_2 \in \mathbf{P}$, $L_1 L_2 \in \mathbf{P}$, $\overline{L_1} \in \mathbf{P}$, and $L_1^* \in \mathbf{P}$.

Problem 6

(a) Consider the language GRAPH-ISOMORPHISM $= \{\langle G_1, G_2 \rangle : G_1$ and G_2 are isomorphic graphs$\}$. Prove that GRAPH-ISOMORPHISM $\in \mathbf{NP}$.

(b) Let ϕ be a boolean formula constructed from the boolean input variables $x_1, x_2, \cdots, x_k$, negations ($\neg$), ANDs ($\wedge$), ORs ($\vee$), and parentheses. The formula ϕ is a *tautology* if it evaluates to 1 for every assignment of 1 and 0 to the input variables. Define TAUTOLOGY as the language of boolean formulas that are tautologies. Show that TAUTOLOGY $\in \mathbf{coNP}$. (Read the textbook to understand the definition of $\mathbf{coNP}$.)

Problem 7

(a) Prove that $\mathbf{P} \subseteq \mathbf{coNP}$.

(b) Prove that if $\mathbf{NP} \neq \mathbf{coNP}$, then $\mathbf{P} \neq \mathbf{NP}$.

Problem Set 13

Data Structures and Algorithms, Fall 2020

Due: Dec 24, in class.

Problem 1

(a) A language L is *complete* for a language class $\mathcal{C}$ with respect to polynomial-time reductions if $L \in \mathcal{C}$ and $L' \leq_P L$ for all $L' \in \mathcal{C}$. Show that $\emptyset$ and $\{0,1\}^*$ are the only languages in $\mathbf{P}$ that are not complete for $\mathbf{P}$ with respect to polynomial-time reductions.

(b) Show that, with respect to polynomial-time reductions, L is complete for $\mathbf{NP}$ if and only if $\overline{L}$ is complete for $\mathbf{coNP}$.

Problem 2

(a) Suppose that someone gives you a polynomial-time algorithm to decide boolean formula satisfiability. Describe how to use this algorithm to find a satisfying assignment in polynomial time. You need to argue your algorithm indeed runs in polynomial time.

(b) We have shown in class that 3-SAT is NP-complete. However, it is known that 2-SAT $\in \mathbf{P}$, where 2-SAT is the language containing all satisfiable boolean formulas in CNF with exactly 2 literals per clause. Prove that 2-SAT $\in \mathbf{P}$ by giving a polynomial time algorithm. You need to argue your algorithm indeed runs in polynomial time.

Problem 3

Given an integer $m \times n$ matrix A and an integer m-vector b, the 0-1 integer programming problem asks whether there exists an integer n-vector x with elements in the set $\{0,1\}$ such that $Ax \leq b$. Prove that 0-1 integer programming is NP-complete. *(Hint: reduce from* 3-SAT.*)*

Problem 4

(a) Suppose that, in addition to edge capacities, a flow network has vertex capacities too. That is each vertex v has a limit $l(v)$ on how much flow can pass though v. Show how to transform a flow network $G = (V, E)$ with vertex capacities into an equivalent flow network $G' = (V', E')$ without vertex capacities, such that a maximum flow in G' has the same value as a maximum flow in G. How many vertices and edges does G' have?

(b) Suppose that you are given a flow network G, and G has edges entering the source s. Let f be a flow in G in which one of the edges (v, s) entering the source has $f(v, s) = 1$. Prove that there must exist another flow f' with $f'(v, s) = 0$ such that $|f| = |f'|$. Give an $O(|E|)$-time algorithm to compute f', given f, and assuming that all edge capacities are integers.

(c) Let $G = (V, E)$ be a bipartite graph with vertex partition $V = L \cup R$, and let G' be its corresponding flow network. Give a good upper bound (that is, as large as possible) on the length of any augmenting path found in G' during the execution of the Ford-Fulkerson algorithm. You need to give an example to justify your bound can be attained. *(Hint: a constant is not the desired answer.)*

Problem 5

The edge connectivity of an undirected graph is the minimum number of edges that must be removed to disconnect the graph. For example, the edge connectivity of a tree is 1, and the edge connectivity of a cyclic chain of vertices is 2. Show how to determine the edge connectivity of an undirected graph $G = (V, E)$ by running a maximum-flow algorithm on at most $|V|$ flow networks, each having $O(|V|)$ vertices and $O(|E|)$ edges. You need to argue the correctness of your algorithm.

Problem 6

You are helping arranging the work schedules of doctors in a hospital. In particular, you need to ensure there is at least one doctor covering each vacation day.

More specifically, there are k vacation periods (e.g., the week of Christmas, the July 4th weekend, the Thanksgiving weekend), each spanning several contiguous days. Let D_j be the set of days included in the j^{th} vacation period. We refer to the union of all these days, $\bigcup_j D_j$, as the set of all vacation days.

There are n doctors at the hospital, and doctor i has a set of vacation days S_i when he or she is available to work. (This may include certain days from a given vacation period but not others; so, for example, a doctor may be able to work the Friday, Saturday, or Sunday of Thanksgiving weekend, but not the Thursday.)

Design and analyze an algorithm that takes this information and determines whether it is possible to select a single doctor to work on each vacation day, subject to the following two constraints:

- For a given parameter c, each doctor should be assigned to work at most c vacation days total, and only days when he or she is available.
- For each vacation period j, each doctor should be assigned to work at most one of the days in the set D_j. (In other words, although a particular doctor may work on several vacation days over the course of a year, he or she should not be assigned to work two or more days of the Thanksgiving weekend, or two or more days of the July 4th weekend, etc.)

Problem 7

Suppose you are given a flow network $G = (V, E)$ with integer edge capacities and an integer-valued maximum flow f in G. Describe and analyze algorithms for each of the following two operations:

(a) `Increment`(e): Increase the capacity of edge e by 1 and update the maximum flow.

(b) `Decrement`(e): Decrease the capacity of edge e by 1 and update the maximum flow.

Both algorithms should modify f so that it is still a maximum flow, more quickly than recomputing a maximum flow from scratch. In particular, each algorithm should run in $O(|V| + |E|)$ time.