Warm-up

Problem 1. Sort the following functions in increasing order of asymptotic growth

$$n, n^3, n\log n, n^n, \frac{3^n}{n^2}, n!, \sqrt{n}, 2^n$$

Solution 1.

$$\sqrt{n}, n, n\log n, n^3, 2^n, \frac{3^n}{n^2}, n!, n^n$$

Problem 2. Sort the following functions in increasing order of asymptotic growth

$$\log\log n, \log n!, 2^{\log\log n}, n^{\frac{1}{\log n}}$$

Solution 2.

$$n^{\frac{1}{\log n}}, \log\log n, 2^{\log\log n}, \log n!$$

Problem 3. Consider the following pseudocode fragment.

```
1: function STARS(n)
2:     for i ← 1; i ≤ n; i++ do
3:         Print '*' i times
```

a) Using the O-notation, upperbound the running time of STARS.

b) Using the Ω-notation, lowerbound the running time of STARS to show that your upperbound is in fact asymptotically tight.

Solution 3.

a) The first iteration prints 1 star, second prints two, third prints three and so on. The total number of stars is $1 + 2 + \cdots + n$, namely,

$$\sum_{j=1}^{n} j \leq \sum_{j=1}^{n} n = n^2 = O(n^2).$$

b) Assume for simplicity that n is even. To lowerbound the running time, we consider only the number of stars printed during the last $\frac{n}{2}$ iterations. Since this is part of the full execution, analyzing only this part gives a lower bound on the total running time. The main observation we need is that for each of the considered iterations, we print at least $n/2$ stars, allowing us to lower bound the total number of stars printed:

$$\sum_{j=1}^{n} j \geq \sum_{j=n/2+1}^{n} \frac{n}{2} = \frac{n^2}{4} = \Omega(n^2).$$

Problem 4. Recall the problem we covered in lecture: Given an array A with n entries, find $0 \leq i \leq j < n$ maximizing $A[i] + \cdots + A[j]$.

Prove that the following algorithm is incorrect: Compute the array B as described in the lectures. Find i minimizing $B[i]$, find j maximizing $B[j+1]$, return (i, j).

Come up with the smallest example possible where the proposed algorithm fails.

Solution 4. Let $A = [1, -2]$, so $B = [0, 1, -1]$. The algorithm return $i = 2$ and $j = 0$. Which does not even obey $i \leq j$.

Problem solving

Problem 5. Given an array A consisting of n integers, we want to compute the upper triangle matrix C where

$$C[i][j] = \frac{A[i] + A[i+1] + \cdots + A[j]}{j - i + 1}$$

for $0 \leq i \leq j < n$. Consider the following algorithm for computing C:

```
1: function SUMMING_UP(A)
2:     C ← new matrix of size(A) by size(A)
3:     for i ← 0; i < n; i++ do
4:         for j ← i; i < n; j++ do
5:             Compute average of entries A[i : j]
6:             Store result in C[i, j]
7:     return C
```

a) Using the O-notation, upperbound the running time of SUMMING-UP.

b) Using the Ω-notation, lowerbound the running time of SUMMING-UP.

Solution 5.

a) The number of iterations is $n + n - 1 + \cdots + 1 = \binom{n+1}{2}$, which is bounded by n^2. In the iteration corresponding to indices (i, j) we need to scan $j - i + 1$ entries from A, so it takes $O(j - i + 1) = O(n)$. Thus, the overall time is $O(n^3)$.

b) In an implementation of this algorithm, Line 5 would be computed with a for loop; when $i < \frac{1}{4}n$ and $j > \frac{3}{4}n$, this loop would iterate least $n/2$ times, which takes $\Omega(n)$ time. There are $n^2/16$ pairs (i, j) of this kind, which is $\Omega(n^2)$. Thus, the overall time is $\Omega(n^3)$.

Problem 6. Come up with a more efficient algorithm for computing the above matrix $C[i][j] = \frac{A[i]+A[i+1]+\cdots+A[j]}{j-i+1}$ for $0 \leq i \leq j < n$. Your algorithm should run in $O(n^2)$ time.

Solution 6. The idea is very simple, suppose we had at our disposal an array B whose ith entry is the entries i through $n - 1$ of A; in other words,

$$B[i] = \sum_{k=0}^{i-1} A[k].$$

Then $C[i][j]$ is simply $\frac{B[j+1]-B[i]}{j-i+1}$. If we can compute B in $O(n^2)$ time we are done. In fact, we can compute B is just $O(n)$ time.

```
1: function SUMMING_UP_FAST(A)
2:     B[0] ← 0
3:     for i ← 1; i < n; i++ do
4:         B[i] ← B[i − 1] + A[i − 1]
5:     for i ← 0; i < n; i++ do
6:         for j ← i; i < n; j++ do
7:             C[i][j] ← (B[j + 1] − B[i])/(j − i + 1)
8:     return C
```

The correctness of the algorithm is clear since

$$C[i][j] = \frac{B[j+1] - B[i]}{j - i + 1} = \frac{\sum_{k=0}^{j} A[k] - \sum_{k=0}^{i-1} A[k]}{j - i + 1} = \frac{\sum_{k=i}^{j} A[k]}{j - i + 1}.$$

as desired.

For the time complexity, we note that the for loop in Line 3 runs in $O(n)$ time and the nested for loops starting in Line 5 runs in $O(n^2)$ time, yielding the desired overall complexity.

Problem 7. Give a formal proof of the transitivity of the O-notation. That is, for function f, g, and h show that if $f(n) = O(g(n))$ and $g(n) = O(h(n))$ then $f(n) = O(h(n))$.

Solution 7. Since $f = O(g)$, it follows that there exists $n_0 > 0$ and $c > 0$ such that $f(n) \leq cg(n)$ for all $n > n_0$. Since $g = O(h)$, it follows that there exists $n_0' > 0$ and $c' > 0$ such that $g(n) \leq ch(n)$ for all $n > n_0$.

It follows that for all $n > \max(n_0, n_0')$ we have

$$f(n) \leq c\, g(n) \leq c\, c'\, h(n).$$

Thus, if we define $n_0'' = \max(n_0, n_0')$ and $c'' = c\, c'$, the above inequality means $f = O(h)$.

Problem 8. Using O-notation, show that the follow snippet of code runs in $O(n^2)$ time. As an extra challenge, it can be shown that it actually runs in $O(n)$ time.

```
1: function ALGORITHM(n)
2:     j ← 0
3:     for i ← 0; i < n; i++ do
4:         while j ≥ 1 and [condition that can be checked in O(1) time] do
5:             j ← j − 1
6:         j ← j + 1
7:     return j
```

Solution 8. Showing $O(n^2)$ time: Line 2 and 5-7 take $O(1)$ time each, as they contain only assignments and basic mathematical operations. Checking the condition of the while-loop on line 4 takes $O(1)$ time as well. Since the while-loop checks for j being positive and the body of the loop decrements j, the loop can be executed at most j times. Since j starts out at 0 and is incremented once in the for-loop, we get that $j \leq n$. Hence, the while-loop on line 4 takes at most $O(n)$ time. As the while loop is inside the for-loop on line 3, which is executed n times, the total time of $O(n^2)$ follows.

Showing $O(n)$ time: To improve the analysis to $O(n)$ time, we need to observe that while j can indeed be decremented a large number of times (depending on the exact condition checked in the while-loop), it can still be decremented at most n times over the entire execution of the algorithm. This is because in every iteration of the for-loop j is incremented once and since the while-loop check for j being positive the while-loop can't execute more than n times over the entire execution of the algorithm. Hence, our running time is actually $O(n)$ time for line 4-5 plus $O(n)$ time for line 3 and 6, instead of the multiplication we used above.

Why does the while-loop contain an arbitrary condition that can be checked in constant time? This algorithm is one of the key components of the Knuth-Morris-Pratt string-matching algorithm, the first linear-time string matching algorithm, which you'll see in COMP2022/2922. That algorithm uses an extra condition that can be checked in constant time, as it needs to check more conditions, but those conditions can checked in constant time.

Problem 9. Given a string (which you can see as an array of characters) of length n, determine whether the string contains k identical consecutive characters. You can assume that $2 \leq k \leq n$. For example, when we take $k = 3$, the string "abaaab"

contains three consecutive a's and thus we should return true, while the string "abaaba" doesn't contain three consecutive characters that are the same.

Your task is to design an algorithm that always correctly returns whether the input string contains k identical consecutive characters. Your solution should run in $O(n)$ time. In particular, k isn't a constant and your running time shouldn't depend on it.

Solution 9. We will loop through the string while maintaining two variables: *last* stores the last character we've seen so far and *count* stores how many consecutive characters before and including *last* are the same as *last*. Initially, we set *last* to the first character of the string and *count* to 1. In every iteration of the loop, we check if the next character is the same as *last*. If so, we increment *count* and once *count* equals k, we return true. If not, we update *last* to be the character that we just processed and reset *count* to 1. If we manage to fully scan the string without outputting true along the way, we return false.

In pseudocode this algorithm could look something like this.

```
1:  function TEST_THREE_CONSECUTIVE(S, k)
2:      last ← S[0]
3:      count ← 1
4:      for i ← 1; i < n; i++ do
5:          if S[i] = last then
6:              count ← count + 1
7:              if count = k then
8:                  return true
9:          else
10:             last ← S[i]
11:             count ← 1
12:     return false
```

To show that this algorithm is correct, we focus on showing that *last* and *count* correctly store the last processed character and the number of identical consecutive characters before and including *last*. This is the invariant of the algorithm. We start by showing that the invariant holds initially: by setting *last* to $S[0]$ and *count* to 1, we ensure that after processing the first character, *last* is indeed set to the last (and only) processed character and *count* stores the number of identical consecutive characters before and including *last*: there are no characters before *last*, so there's only one consecutive character which is equal to *last* itself.

Next we show that the invariant is maintained after processing the next character. In order to do so, we assume that the invariant holds before we process this character (making this essentially an inductive argument). Now let's see what happens when we process the next character. We'll first consider the case where $S[i]$ doesn't match *last*. In this case we failed to find k consecutive characters that are identical and since $S[i]$ differs from last, the next substring that we should consider starts from $S[i]$ and thus currently we have seen 1 consecutive identical characters. Hence, by setting *last* to $S[i]$ and *count* to 1, we maintain the invariants. If *last* and $S[i]$ are equal, we can extend our current substring of identical characters using $S[i]$

and hence its length increases by 1. Hence, we don't have to update *last*, but we do need to increment *count*.

How does this help us to prove the correctness of the algorithm? Since we maintain this invariant, we know that at any point in the algorithm we have the correct length of the longest substring of identical characters that includes the last character. Hence, when we check if this length is k, we know whether we have found a substring of sufficient length. Hence, when we return true, we know we indeed found a substring satisfying the condition. On the other hand, when we return false, we know that we scanned the entire string and at no point did we have a substring of identical characters of length k (since we check for this every time we extend our substring by one character). Hence, when we return false there indeed was no substring satisfying our condition. Together these two statements imply the correctness of our algorithm.

Analyzing the running time of this algorithm is fairly straightforward: line 2-3 and 5-12 all consist of (a constant number of) assignments, simple comparisons, and/or simple math operations and the return statements return booleans, all of which takes $O(1)$ time. The loop on lines 4-11 loops through (at most) the remaining $n-1$ characters of the string, performing $O(1)$ time operations each time. Hence, the loop takes at most $(n-1) \cdot O(1)$ or $O(n)$ time. Since we saw that the operations outside the loop all take constant time, the loop dominates the running time, which comes down to $O(n)$ time in total, as required.

Problem 10. Given an array with n integer values, we would like to know if there are any duplicates in the array. Design an algorithm for this task and analyze its time complexity.

Solution 10. The straightforward solution is to do a double for loop over the entries of the array returning "found duplicates" right away when we find a pair of identical elements, and "found no duplicates" at the end if we exit the for loops.

This algorithm is correct, since we compare every pair of elements, thus if there exists a duplicate pair, we consider this pair at some point in the execution. Similarly, if no duplicates exist, we can safely return this after checking all pairs.

The complexity of this solution is $O(n^2)$, since there are $O(n^2)$ pairs of elements to consider and performing a single such comparison takes $O(1)$ time.

A better solution is to sort the elements and then do a linear time scan testing adjacent positions. The correctness follows from the fact that in a sorted array, duplicate integers have to occur in consecutive indices. As we shall see later in class one can sort an array of length n in $O(n \log n)$ time, which also dominates the running time of this algorithm, as scanning through n elements and performing a single comparison for consecutive integers takes $O(1)$ time per pair and $O(n)$ time in total.

Warm-up

Problem 1. Suppose we implement a stack using a singly linked list. What would be the complexity of the push and pop operations? Try to be as efficient as possible.

Solution 1. To push an element we add it at be beginning of the list. To pop an element we delete and return the first element of the list. Both operations take $O(1)$ time.

Problem 2. Suppose we implement a queue using a singly linked list. What would be the complexity of the enqueue and dequeue operations? Try to be as efficient as possible.

Solution 2. To enqueue an element we add it at be end of the list. To dequeue an element we remove it from the front of the list. To keep the enqueue operations efficient, we can keep a pointer to the last element of the list, so that both operations take $O(1)$.

Problem solving

Problem 3. Given a singly linked list, we would like to traverse the elements of the list in reverse order.

a) Design an algorithm that uses $O(1)$ extra space. What is the best time complexity you can get?

b) Design an algorithm that uses $O(\sqrt{n})$ extra space. What is the best time complexity you can get?

You are not allowed to modify the list, but you are allowed to use position/cursors to move around the list.

Solution 3.

a) We keep a cursor (position) that initially points to the last element of the list. We iteratively scan the list until we find the node before the cursor, visit the element at the cursor, and update the cursor to the previous node. The time complexity is $\Theta(n + n - 1 + \cdots + 1) = \Theta(n^2)$.

b) We store $\sqrt{n}$ cursors evenly spaced along the list. We traverse the span between two of these cursors using the previous strategy. Each of these segments is $\sqrt{n}$ long, so each segment takes $\Theta(\sqrt{n}^2) = \Theta(n)$ time to traverse. There are $\sqrt{n}$ many such segments, so the total time is $\Theta(n^{3/2})$.
We can even do better if we are more aggressive on the data that we store. To scan between two cursors, we can traverse the chunk in the forward direction storing all the elements on a stack of size $\sqrt{n}$. Then we can use the smaller stack to traverse that segment in reverse order in $O(\sqrt{n})$ time. In this way, the space is still $O(\sqrt{n})$ and the overall time is $O(n)$.

Problem 4. Consider the problem of given an integer n, generating all possible permutations of the numbers $\{1, 2, \ldots, n\}$. Provide a recursive algorithm for this problem.

Solution 4. The helper function outputs permutations of the input array A that have the first i elements fixed.

The correctness of the algorithm hinges on the fact that while the helper function and its many recursive calls may modify the array during their execution, when a call to a helper function finally returns, the input array is always restored to the state it was in when the call started executing. For a formal argument we prove by induction the property that when we call `helper(A, i)` the algorithm outputs all of the array A that leaves all the entries $A[0 : i]$ fixed while trying all permutations of $A[i : n]$. The full proof is left to the reader.

```
1: function PERMUTATIONS-RECURSIVE(n)
2:     # input: integer n
3:     # do: print all permutations of order n
4:     A ← [1, 2, ..., n]
5:     HELPER(A, 0)
```

```
1: function HELPER(A, i)
2:     if i = size(A) then
3:         Print A
4:     for j ← i; j < n; j++ do
5:         Swap A[i] and A[j]
6:         HELPER(A, i + 1)
7:         Swap A[i] and A[j]
```

Problem 5. Consider the problem of given an integer n, generating all possible permutations of the numbers $\{1, 2, \ldots, n\}$. Provide a non-recursive algorithm for this problem using a stack.

Solution 5. For the non-recursive version we simulate the calls to the helper function with a stack. We use the tuple (c, i, j) to denote stages of a call. The tuple ("start", i, j) corresponds to the start of the for loop for some choice of (i, j) and the

tuple ("finish", i, j) to the part of the body of the for loop after the recursive call to `helper`.

```
1:  function PERMUTATIONS(n)
2:      # input: integer n
3:      # do: print all permutations of order n
4:      A ← [1, 2, ..., n]
5:      S ← a stack with the tuple ("start", 0, 0)
6:      while S is not empty do
7:          c, i, j ← S.pop()
8:          if c = "start" then
9:              if i = n then
10:                 Print A
11:             else
12:                 A[i], A[j] ← A[j], A[i]
13:                 S.push("finish", i, j)
14:                 S.push("start", i + 1, i + 1)
15:         if c = "finish" then
16:             A[i], A[j] ← A[j], A[i]
17:             if j < n − 1 then
18:                 S.push("start", i, j + 1)
```

Problem 6. Using only two stacks, provide an implementation of a queue. Analyze the time complexity of enqueue and dequeue operations.

Solution 6. The simplest solution is to push elements as they arrive into the first stack. When we are required to carry out a dequeue operation, we transfer all the elements to the second stack, pop once to later return the element on the top of the second stack, and then transfer back all the remaining elements back to the first stack.

This strategy works because when we transfer the elements from one stack to the next, we reverse the order of the elements. Before we transfer things, the most recent element to be queued is at the top of the first stack. After we transfer we have the oldest element queued at the top of the second stack. Finally, when we transfer the elements back to the first, we go back to the original stack order.

If the queue holds n elements each enqueue operation takes $O(1)$ time and each dequeue takes $O(n)$ time since we need to transfer all n elements twice.

Problem 7. We want to extend the queue that we saw during the lectures with an operation GETAVERAGE() that returns the average value of all elements stored in the queue. This operation should run in $O(1)$ time and the running time of the other queue operations should remain the same as those of a regular queue.

a) Design the GETAVERAGE() operation. Also describe any changes you make to the other operations, if any.

b) Briefly argue the correctness of your operation(s).

c) Analyse the running time of your operation(s).

Solution 7.

a) Recall that the average is the total sum of the elements divided by the number of elements. Since we already store the size of the queue, it suffices to add a single new variable that stores the sum of the elements, say *sum*. When a new element get enqueued or dequeued, the sum needs to be updated.

```
1: function getAverage()
2:     if isEmpty() then
3:         return "queue empty"
4:     else
5:         return sum/size
```

```
1: function newEnqueue(e)
2:     sum ← sum + e
3:     enqueue(e)
```

```
1: function newDequeue()
2:     e ← dequeue()
3:     sum ← sum − e
4:     return e
```

b) The correctness follows directly from the definition of average, assuming we maintain the sum correctly. Since we add the enqueued element's value to the sum and subtract it when it's dequeued, the sum is maintained correctly.

c) The getAverage operation checks if the queue is empty, which takes $O(1)$ time and either throws an error ($O(1)$ time) or returns the required division of two integers ($O(1)$ time). Hence, the total running time is $O(1)$ as required. We modified the enqueue and dequeue operations. Adding the new element to the *sum* takes $O(1)$ time, so enqueue still runs in $O(1)$ time. Similarly, subtracting the removed element from the *sum* takes $O(1)$ time, so dequeue still runs in $O(1)$ time.

Warm-up

Problem 1. Let A be an array holding n distinct integer values. We say that a tree T is a *pre-order realization of* A if T holds the values in A and a pre-order traversal of T visits the values in the order they appear in A.

Design an algorithm that given an array produces a pre-order realization of it.

Solution 1. Let T be a tree where every internal node has a right child and no left child. Such a tree is simply a path connecting the root to its only leaf and its pre-order traversal visits the nodes along the path from the root going down to its leaf. Therefore, assigning the values of A to the path from the root going down is a pre-order realization of A.

It is worth noting that given any tree on n nodes, we can assign the values of A to its nodes such that T is a pre-order realization of A. The above construction is just easier to describe and implement.

Problem 2. Let A be an array holding n distinct integer values. We say that a tree T is a *post-order realization of* A if T holds the values in A and a post-order traversal of T visits the values in the order they appear in A.

Design an algorithm that given an array produces a post-order realization of it.

Solution 2. Let T be a tree where every internal node has a right child and no left child. Such a tree is simply a path connecting the root to its only leaf and its post-order traversal visits the nodes along the path from the leaf going up towards the root. Therefore, assigning the values of A to the path from the leaf going up is a post-order realization of A.

It is worth noting that given any tree on n nodes, we can assign the values of A to its nodes such that T is a post-order realization of A. The above construction is just easier to describe an implement.

Problem solving

Problem 3. Design a linear time algorithm that given a tree T computes for every node u in T the size of the subtree rooted at u.

Solution 3. We use a recursive helper function. When the function is called at some node u in T, we recursively compute the size of the left and right subtrees of u, add 1, set the result to be the size of the subree at u, and return the value to be reused further up in the recursion.

```
1: function SUBTREE-SIZE(T)
2:     SIZE-HELPER(T.root)
```

```
1: function SIZE-HELPER(u)
2:     u.sub_size ← 1
3:     for w ∈ u.children() do
4:         u.sub_size ← u.sub_size+ SIZE-HELPER(w)
5:     return u.sub_size
```

We claim that the call SIZE-HELPER(u) sets $u.sub\_size$ to the size of the subtree rooted at u and returns this value. The correctness of this claim rests on the inductive assumption that the claim is true for recursive calls to smaller subtrees (the ones defined by the children of u) and the fact that

$$size(u) = 1 + \sum_{w \text{ : child of } u} size(w).$$

In order to analyze the running time of our algorithm, we first notice that the only thing that SUBTREE-SIZE does is call SUBTREE-HELPER. Hence, the running time of that algorithm is $O(1)$ time for the function call plus the time needed by SUBTREE-HELPER. In SUBTREE-HELPER, line 2 and 5 take constant time. Ignoring the recursive call, line 3-4 take time proportional in the number of children of u: $O(1) + \sum_{w \text{ : child of } u} O(1)$ or $O(1+$ #children of $u)$. Hence the total running time of this function is $O(1+$ #children of $u)$. We note that SUBTREE-HELPER is called exactly once for each node. Hence, over all nodes in the tree its running time is $\sum_{u \in T} O(1+$ #children of $u)$. Since the total number of children in the tree is $n-1$ (only the root isn't a child of another node) this implies that the total running time is $O(n)$.

Problem 4. In a binary tree there is a natural ordering of the nodes on a given level of the tree, i.e., the left-to-right order that you get when you draw the tree. Design an algorithm that given a tree T and a level k, visits the nodes in level k in this natural order. Your algorithm should perform the whole traversal in $O(n)$ time.

Solution 4. We perform an in-order traversal over all nodes in the tree, but we do not execute the VISIT routine on every node. Instead we keep track of the level of the node we are traversing and we call VISIT only when the level of the node we are currently traversing equals k.

The extra bookkeeping needed to track the level we are at does not increase the running time of the standard in-order traversal, which is $O(n)$.

Problem 5. Design an algorithm that given a binary tree T and a node u, returns the node that would be visited after u in a pre-order traversal. Your algorithm should *not* compute the full traversal and then search for u in that traversal.

Solution 5. If $u.left \neq \emptyset$ then $u.left$ is the next node. Else, if $u.left = \emptyset$ and $u.right \neq \emptyset$ then $u.right$ is the next node. Otherwise, let v be the first ancestor of u that has a right child and $v.right$ is not u's ancestor (that includes u itself), then $v.right$ is the next node. If no such node v exists then u is the last node in the traversal and there is no next node.

The algorithm may take up to $O(n)$ time to find the next node.

Problem 6. Design an algorithm that given a binary tree T and a node u, returns the node that would be visited after u in an in-order traversal. Your algorithm should *not* compute the full traversal and then search for u in that traversal.

Solution 6. If $u.right \neq \emptyset$ then the next node is the left-most descendant of $u.right$ (that is we follow the left child pointer until we reach a node without a left child). Else, we move to the parent of u, call it v. If we arrive at v from its left subtree, then v is the next node; otherwise, we keep updating v until we find such a node. If we happen to reach the root, then u is the last node in the traversal and there is no next node.

The algorithm may take up to $O(n)$ time to find the next node.

Problem 7. Design an algorithm that given a binary tree T and a node u, returns the node that would be visited after u in a post-order traversal. Your algorithm should *not* compute the full traversal and then search for u in that traversal.

Solution 7. If u does not have a parent, then u is the last node in the traversal and there is no next node. Otherwise, let v be the parent of u. If u is v's right child, then v is the next node. If u is v's left child and v has no right child, then v is the next node. If u is v's left child and v has a right child, then we start a post-order traversal at $v.right$.

The algorithm may take up to $O(n)$ time to find the next node.

Problem 8. The balance factor of a node in a binary tree is the absolute difference in height between its left and right subtrees (if the left/right subtree is empty we consider its height to be -1). Design an algorithm for computing the balance factor of **every** node in the tree in $O(n)$ time.

Solution 8. We use a recursive helper function that takes as input a node u and computes the balance factor at u and returns the height of the subtree rooted at u.

```
1: function BALANCE(T)
2:     BALANCE-HELPER(T.root)
```

```
1: function BALANCE-HELPER(u)
2:     left_height ← −1
3:     right_height ← −1
4:     if u.left ≠ nil then
5:         left_height ← BALANCE-HELPER(u.left)
6:     if u.right ≠ nil then
7:         right_height ← BALANCE-HELPER(u.right)
8:     u.balance ← |left_height − right_height)|
9:     return 1 + max(left_height, right_height)
```

We claim that BALANCE-HELPER(u) sets $u.balance$ to the balance factor at u and returns the height of u. To prove the correctness of this claim we use the inductive assumption that the algorithm returns the correct height for the left and right

subtrees. If both are empty then BALANCE-HELPER returns 0 and sets $u.balance$ to 0, which is the correct thing to do. If one of them is non-empty we return its height plus 1 and set $u.balance$ to its height, which again is correct. Finally, if both are non-empty, we return $1 + \max(left\_height, right\_height)$ and set $u.balance$ to $|left\_height - right\_height|$, which again is correct.

Each call to BALANCE-HELPER(u) takes $O(1)$ time, not counting the work done in recursive calls. Therefore, the total running time is $O(n)$ time.

Problem 9. Describe an algorithm for performing an Euler tour traversal of a binary tree that runs in linear time and **does not** use a stack or recursion.

Solution 9. In an Euler tour traversal of a binary tree, our current position is determined by the vertex we are at and whether we are visiting this vertex from the left, from the bottom, or from the right. We encode this position with a pair (u, W) where u is a node in the tree and $W \in \{L, B, R\}$. For each position the following function returns the next state.

```
1:  function EULER-NEXT(u, W)
2:      if W = "L" and u.left ≠ nil then
3:          return (u.left, "L")
4:      if W = "L" and u.left = nil then
5:          return (u, "B")
6:      if W = "B" and u.right ≠ nil then
7:          return (u.right, "L")
8:      if W = "B" and u.right = nil then
9:          return (u, "R")
10:     if W = "R" and u is the root then
11:         return "the end"
12:     if W = "R" and u = u.parent.left then
13:         return (u.parent, "B")
14:     if W = "R" and u = u.parent.right then
15:         return (u.parent, "R")
```

Correctness follows directly from the definition of the Euler tour traversal.

We only do a constant number of comparisons, so the function takes $O(1)$ time. Since the function is executed exactly three time per node (once for each of the three states), the running time per node is $O(1)$. Hence the total running time if $O(n)$ for the tree.

Problem 10. For any pair of nodes in a tree there is a unique simple path (one that does not repeat vertices) connecting them. Design a linear time algorithm that finds the longest such path in a tree.

Solution 10. Here is a sketch of a solution. The longest path either connects the root and a leaf or it connects two leaves. In the first case, if the longest path connects the root to a leaf, we compute the deepest node by computing the height of every node and finding the maximum. Its height is the length of this path.

In the second case, if we let u be the lowest common ancestor of the leaves and let v and w be the two children of u that are ancestors of these two leaves, then the length of the path is

$$height(v) + height(w) + 2.$$

Therefore, we can compute in $O(n)$ time the heights of all subtrees and then we can find the internal vertex maximizing the above formula.

The issue of how to find the actual path can be resolved by not only remembering the height of each subtree but also through which child there is a path to a leaf attaining that height.

Warm-up

Problem 1. Give the full pseudocode for doing a trinode restructure at x, y, z where x is the left child of y and y is the left child of z.

Solution 1. Recall that x, y, z are internal nodes holding keys and due to their relation, $x.key < y.key < z.key$. First we identify the subtrees T_0, T_1, T_2, T_3 as in the picture of a single rotation from class (please refer to the picture from the slides). Then we update the left and right pointers of each node to reflect the updated tree structure.

```
1: T_0 ← x.left
2: T_1 ← x.right
3: T_2 ← y.right
4: T_3 ← z.right
5: y.left, y.right ← x, z
6: x.left, x.right ← T_0, T_1
7: z.left, z.right ← T_2, T_3
```

Problem 2. Consider the implementation of a binary search tree on n keys where the keys are stored in the internal nodes. Prove using induction that the height of the tree is at least $\log_2 n$.

Solution 2. Let $N(h)$ be the maximum number of keys that a tree of height h can have. Note that $N(1) = 1$ and $N(2) = 3$. For $h > 1$, we have $N(h) = 2N(h-1) + 1$ since we can take apart $N(h)$ into its left and right subtrees (each one no greater than $N(h-1)$) plus the root.

Using this inequality we can prove by induction that $N(h) = 2^h - 1$. The base case is $h = 1$, which is clearly true. For the inductive case, we note that $N(h) = 2N(h-1) + 1 = 2(2^{h-1} - 1) + 1 = 2^h - 1$, where the second inequality follows from our inductive hypothesis.

Taking the log on both sides of $N(h) = 2^h - 1$ we get $\log_2 N(h) < h$, which is precisely what we are after.

Problem 3. Consider the implementation of a binary search tree on n keys where the keys are stored in the internal nodes. Prove using induction that the number of external nodes is $n + 1$.

Solution 3. We give a proof by induction on n. Our base case is $n = 1$, which consists of a tree with one internal node and two external nodes. For the inductive case, consider any tree T with n keys. Let w be an internal node with two external children (there is always at least one) and consider the resulting tree after deleting w from the tree, which involves replacing w and its two external children with one external node. The resulting tree has $n - 1$ keys so by the induction hypothesis it has n external leaves. Inserting w back where it used to be, causes the tree to lose

the leaf that replaced w, but it gains the two leaves of w. Hence, in total it gains one leaf, so the number of leaves in T is $n + 1$.

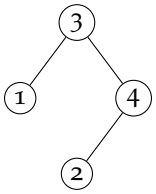
Problem solving

Problem 4. Consider the following algorithm for testing if a given binary tree has the binary search tree property.

```
1: function TEST-BST(T)
2:     for u ∈ T do
3:         if u.left ≠ nil and u.key < u.left.key then
4:             return False
5:         if u.right ≠ nil and u.key > u.right.key then
6:             return False
7:     return True
```

Either prove the correctness of this algorithm or provide a counter example where it fails to return the correct answer.

Solution 4. The algorithm is not correct as it returns True on the following tree lacking the binary search tree property.



Problem 5. Consider the following operation on a binary search tree: LARGEST() that returns the largest key in the tree.

Give an implementation that runs in $O(h)$ time, where h is the height of the tree.

Solution 5. Starting at the root, follow the right pointer until we reach a node u that has no right subtree and return the key of that node.

Since in a binary search tree we have that for every node u $u.left.key < u.key < u.right.key$, the above algorithm correctly finds the largest key in the tree.

The algorithm clearly runs in $O(h)$ time, as it takes $O(1)$ time per level of the tree it visits.

Problem 6. Consider the following operation on a binary search tree: SECOND-LARGEST() that returns the second largest key in the tree.

Give an implementation that runs in $O(h)$ time, where h is the height of the tree.

Solution 6. First we find the node w holding the largest key in the tree. If w has a left child, then the second largest key must be in this subtree, so we again search for the largest key in $w.left$.

Otherwise, the second largest key can be found at w's parent, provided that w has a parent, i.e., w is not the root of the tree. If w is the root of the tree and its left subtree is empty, then w is the only node in the tree and so the second-largest element is not defined and we can throw an exception.

The correctness follows from the binary search tree property. After finding the largest key, the second largest key is either its parent or the largest value in its left subtree. Since we know that every key in the left subtree of w is larger than the key of w's parent, the choice of the algorithm follows. If w has no left subtree, we know that there are no keys between w and its parent, as all keys in the left subtree of w's parent and those higher up in the tree are smaller than w's parent's key by the binary search tree property.

Since the algorithm takes $O(1)$ time per level of the tree it visits, the total running time is $O(h)$.

Problem 7. Consider the following operation on a binary search tree: MEDIAN() that returns the median element of the tree (the element at position $\lfloor n/2 \rfloor$ when considering all elements in sorted order).

Give an implementation that runs in $O(h)$, where h is the height of the tree. You are allowed to augment the insertion and delete routines to keep additional data at each node.

Solution 7. We can augment the nodes in our binary search trees with a size attribute. For a node u, $u.size$ is the number of nodes in the subtree rooted at u. When inserting a new key at some node w, we only need to increase by 1 the size of ancestors of w, which takes $O(h)$ time. Similarly, when we delete a node w with at least one external child, we only need to decrease by 1 the size of ancestors of w. Hence, the insertion and deletion times remain the same.

We implement a more powerful operation called POSITION(k), which returns the kth key (in sorted order) stored in the tree. When searching for the kth element at some node u, we check if $u.left.size \geq k$ in which case we know that the kth element is in the left subtree and we search for the kth key in $u.left$. Otherwise, if $u.left.size = k - 1$, the kth key is at the root u. Finally, if $u.left.size < k - 1$, then we know that the kth element is in the right subtree, so we search for the $(k - u.left.size - 1)$th key there, i.e., k minus the number of nodes smaller than the root of $u.right.key$.

The complexity is $O(h)$ because we do $O(1)$ work at each node and we only traverse one path from the root to the element we are searching for.

Problem 8. Consider the following binary search tree operation: REMOVE_ALL(k_1, k_2) that removes from the tree any key $k \in [k_1, k_2]$.

Give an implementation of this operation that runs in $O(h + s)$ where h is the height of the tree and s is the number of keys we are to remove.

Solution 8. We solve a simpler problem first; REMOVE-GREATER(w, k): remove all nodes with key at least a given key k from the tree rooted at w (the problem of removing nodes smaller than the given key is symmetric).

We do a search for k that takes us to a node u (which may hold k or may be an external node). Let v be any node on the path from u to the root that has a right child that is not on the path from u to the root. We can delete the nodes in those right subtrees right away since they are guaranteed to be greater than k. After that we may need to remove some of the nodes on the path from u to the root. Let v be a node in the path from u to the root. If u is a right descendant of v then there is no need to remove v as its key is smaller than k. If u is left descendant of v then after the first pruning step v has one external child so we can delete it in $O(1)$ time as we argued in class. Finally, we remove the bifurcation node u with the usual delete procedure.

Now to implement the REMOVE-ALL(k_1, k_2) operation, we perform the usual range search until we find the first node u such that $k_1 \leq u.key \leq k_2$. Then we call REMOVE-GREATER$(u.left, k_1)$ and REMOVE-SMALLER$(u.right, k_2)$. Finally, we delete u.

As we saw in class we can divide the nodes into boundary nodes (on the search path from the root to k_1 or k_2), inside nodes (those inside the range but not on the boundary), and outside nodes (those outside the range but not the boundary). The calls to REMOVE-GREATER and REMOVE-SMALLER run in $O(h + s)$ where h is the height of the tree and s is the number of deleted nodes, since the work done is proportional to the number of inside nodes, which is at most s, and the length of the search path, which is at most h. Finally, the removal of the bifurcation node u takes the usual $O(h)$ time. Hence, the overall complexity is $O(h + s)$.

Warm-up

Problem 1. Come up with an instance showing that SELECTION-SORT takes $\Omega(n^2)$ time in the worst case.

Solution 1. Any array will takes $\Omega(n^2)$: For each of the first $n/2$ iterations of the top level for loop, the inner loop runs for at least $n/2$ iterations, each one taking constant time, so $\Omega(n^2)$ overall.

Problem 2. Come up with an instance showing that INSERTION-SORT takes $\Omega(n^2)$ time in the worst case.

Solution 2. Consider the array sorted in descending order. In the ith iteration of the for loop when we need to insert $A[i]$ we need to move all the entries from 0 to $i-1$ one position to the right. Therefore, the last $n/2$ iterations of the for loop will cause the inner while loop to iterate at least $n/2$ times, so $\Omega(n^2)$ time overall.

Problem solving

Problem 3. Come up with an instance showing that HEAP-SORT takes $\Omega(n \log n)$ time in the worst case.

Solution 3. Imagine a heap with keys $1, \ldots, n$, such that $n = 2^h - 1$; that is, the last level (i.e., level $h-1$) is full. Imagine the last level of the heap has the keys $2^h - 1, 2^h - 2, \ldots, 2^{h-1}$ listed from left to right. Now suppose we do 2^{h-1} remove-min operations. On each operation, after moving the key of the last node (call it k) to the root, we need to perform a down-heap operation to repair the heap-order property. Note that every node in level $h-2$ and up is smaller than k, while every node in level $h-1$ has a larger key; thus the down-heap operation brings k down to level $h-2$, which takes $\Omega(h)$ work. Thus, the total time spent on these operations is $\Omega(2^{h-1}h) = \Omega(n \log n)$.

Problem 4. Given an array A with n integers, an inversion is a pair of indices $i < j$ such that $A[i] > A[j]$. Show that the in-place version of INSERTION-SORT runs in $O(n + I)$ time where I is the total number of inversions.

Solution 4. Let I_i be the number of inversions at the beginning of the iteration of the for loop corresponding to $i \in [1, n)$ and I_n be the number of inversions at the end; thus we have $I_1 = I$ and $I_n = 0$. Note that if the inner while loop runs for k_i iterations for a given value of i, it removes k_i inversions and thus $I_i = I_{i+1} + k_i$. Let W be the total number of iteration of the inner while loop. Then,

$$W = \sum_{i=1}^{n-1} k_i = \sum_{i=1}^{n-1} (I_i - I_{i+1}) = I_1 - I_n = I.$$

The total running time is clearly $O(n + W)$, so the total running time in $O(n + I)$.

Problem 5. Given an array A with n distinct integers, design an $O(n \log k)$ time algorithm for finding the kth value in sorted order.

Solution 5. We use a priority queue that supports MAX and REMOVE-MAX operations. We start by inserting the first k elements from A into the priority queue. Then we scan the rest of array comparing the current entry $A[i]$ to the maximum value in the priority queue, if $A[i]$ is greater than the current maximum, we can safely ignore it as it cannot be the kth value due to the fact that the priority queue holds k values smaller than $A[i]$. On the other hand, if $A[i]$ is less than the current maximum, we remove the maximum from the queue and insert $A[i]$. After we are done processing all the entries in A, we return the maximum value in the priority queue.

To argue the correctness of the algorithm we can use induction to prove that after processing $A[i]$, the k smallest elements in $A[0, i]$ are in the queue. Since we return the largest element in the queue after processing the whole array A (which we just argued holds the k smaller elements in A), we are guaranteed to return the kth element of A.

The time complexity is dominated by $O(n)$ plus the time it takes to perform the REMOVE-MAX and INSERT operations in the priority queue. In the worst case we perform n such operations each one taking $O(\log k)$ time, when we implement the priority queue as a heap. Thus, the overall time complexity is $O(n \log k)$.

Problem 6. Given k sorted lists of length m, design an algorithm that merges the list into a single sorted lists in $O(mk \log k)$ time.

Solution 6. We keep a priority queue holding pointers to positions in the lists where the priority of a pointer is the value it points to. Initially, we add the head of each list to the priority queue. We iteratively remove the minimum priority pointer, add it at the end of the merged list, and then, provided we are not already at the end of that list, insert the next element of the list where that pointer came from.

To argue the correctness of the algorithm, notice that the minimum value in the queue never goes down, as the next pointer we add can only have larger value than the minimum we just removed because the input lists are sorted. Therefore the output is in sorted order.

For the time complexity, note that we always keep at most k items in the priority queue, so each remove and insert operations takes $O(\log k)$, when using a heap. Since there are a total of mk elements, the total time is $O(mk \log k)$.

Warm-up

Problem 1. Consider a hash table of size $N > 1$, and the hash function such that $h(k) = k \bmod 2$ for every k. We insert a dataset S of size $n < N$. After that, what is the typical running times of GET for chaining and open addressing (as a function of n)?

Solution 1. $\Theta(n)$ for both. Since the hash function maps all elements to only two values, 0 and 1, there are at least $n/2$ elements hashed to the same value – for instance, 0. In the case of chaining, that means that at least $n/2$ elements of S are now part of a linked list, and so doing a GET on any of those boils down to doing a linear search into a linked list of size $\Theta(n)$: $\Theta(n)$. In the open addressing case, similarly we have a sequence of at least $n-1$ contiguous positions corresponding to those elements: indeed, everything is mapped to either 0 or 1, so after the first 0 everything collides at 1 and every single following index until an empty slot is found (and if there is no element mapped to 0, then all n insertions collide at 1), and so we end up doing a search in an unsorted array of size $\Theta(n)$.
Upshot: the choice of hash function matters! Don't come up with your own, it may be a bad idea...

Problem 2. Suppose you are given a hash function h mapping 10-digit strings to integers in $\{1, 2, \ldots, 10000\}$. Show that there is some dataset S of size $1,000,000$ such that all keys of S are hashed to the *same* value.

Solution 2. There are 10^{10} different 10-digit strings. By the Pigeonhole Principle, there exists some $i \in \{1, 2, \ldots, 10000\}$ such that $|h^{-1}(i)| \geq \frac{10^{10}}{10000} = 10^6$ (where $h^{-1}(i) \subseteq S$ is the set of keys k such that $h(k) = i$) . Let S be this set $h^{-1}(i)$.
Upshot: for every hash function h, there exists some dataset $S = S(h)$ (depends on the hash function!) for which h is arbitrarily bad. So all we can ask for is that hash functions be good for *most* (i.e., "typical" for our applications) datasets, not *all*.

Problem 3. Work out the details of implementing cycle detection in cuckoo hashing based on the number of iterations of the eviction sequence.

Solution 3. There are $2N$ entries in total, so if the cuckoo eviction process runs more than $4N$ times, we are guaranteed to have a cycle: there can be $2N$ shifts to move all entries to their alternative position and then another $2N$ to move everything back to their initial positions. By adding a counter that is incremented every time we evict an entry, this is easily checked in $O(1)$ time after every eviction.

Problem 4. Work out the details of implementing cycle detection in cuckoo hashing based on keeping a flag for each entry.

Solution 4. We augment each entry to have a boolean attribute called flag. Assuming that initially all entries are unflagged (i.e., flag is set to false), the put routine

sets the flag of every item it evicts. If we ever run into a flagged item, we have a cycle.

However, if we successfully find an empty slot, we must trace back our steps along the eviction sequence to unflag all the entries flagged. That way when we need to execute the next put all entries are unflagged again.

The running time is proportional to the length of the eviction sequence.

Problem solving

Problem 5. Design a sorted hash table data structure that performs the usual operations of a hash table with the additional requirement that when we iterate over the items, we do so in the order in which they were inserted into the hash table. Iterating over the items should take $O(n)$ time where n is the number of items stored in the hash table. Your data structure should only add $O(1)$ time to the standard put, get, and delete operations.

Solution 5. In addition to the hash table, we keep a doubly linked list of the entries and augment each entry in the hash table to also have a pointer to its position in the list.

When we need to put a new item, after inserting it into the hash table in the usual way, we add it to the end of the list and set the pointer of the entry in the hash table accordingly in $O(1)$ time.

When we remove an item, after removing it from the hash table in the usual way, we use the pointer to the doubly linked list to remove it from there as well in $O(1)$ time.

When we update an existing item, after updating it in the usual way, we move its position in the list to the end in $O(1)$ time or we leave it in place depending on how we want to interpret this case; i.e., we care about the order of when the key of the item arrived or when the value of the item arrived.

Whenever we need to iterate over the entries, we use the linked list. Since iterating through all elements of a doubly linked list takes $O(n)$ time, this satisfies our requirements.

Problem 6. Given an array with n integers, design an algorithm for finding a value that is most frequent in the array. Your algorithm should run in $O(n)$ expected time.

Solution 6. Keep a hash table with $2n$ entries using linear probing where the keys are the integers in the input array and the value is the frequency of the key.

We scan the integers in the array, updating their frequency in our hash table. That is, when processing k, we first try to get the entry for k. If there is no entry, we put $(k, 1)$; otherwise, if there is already an entry (k, f) we updated with $(k, f + 1)$. At the end we do a scan of the table to find the integer with maximum frequency.

Given that the load factor of the hash table is $\leq 1/2$, the hash table operations take $O(1)$ expected time. Finally, scanning the whole hash table to find the maximum frequency integer take $O(n)$ time.

It is worth noting that we can avoid the scan of the hash table if we also keep track of the maximum frequency item we have seen so far. Every time we update the frequency of a number, we check if its frequency is larger than the maximum frequency so far, and if so, we update it.

Problem 7. A multimap is a data structure that allows for multiple values to be associated with the same key. The method GET(k) should return all the values associated with key k. Describe an implementation where this method runs in $O(1+s)$ expected time, where s is the number of values associated with k.

Solution 7. *(Sketch)* Each entry, instead of having a single value, has a linked list of values associated with the key. When putting a new item (k, v) we add v to the list in the entry associated with k. When getting a key k, we find it in $O(1)$ expected time and we go over the list in the entry associated with k (which takes $O(s)$ time).

Problem 8. Suppose that you have a group of n people and you would like to know if there are two people that share a birthday. Design an $O(1)$ time algorithm that given the information about the n people's birthdays, finds a pair that shares a birthday, or reports that no such pair exists.

Solution 8. Map the birthday to a number in $[1, 365]$ and use a hash function on integers to build a hash table of size $2 * 365$ using linear probing. We iterate over the elements in the list, for each person p we treat their birthday as a key k and try to get the entry with k. If there is no such key we add (k, p) to the hash table; otherwise, if we find the item (k, p') we have our pair p, p' of people sharing a birthday and we can stop.

If we scan the whole set of n people without finding a match, we report that none share a birthday.

For the time complexity, we note that if $n > 365$ we are guaranteed to find a matching pair of people in the first 366 entries of the array so the algorithm terminates after $O(1)$ iterations and each iteration takes $O(1)$ time since the hash table has $O(1)$ size.

Problem 9. In computational linguistics, texts (such as a book or an article) are modelled as a sequence of words. A k-gram is a sequence of k consecutive words. A common task in language modelling requires that we compute the frequency of all k-grams that appear in the text.

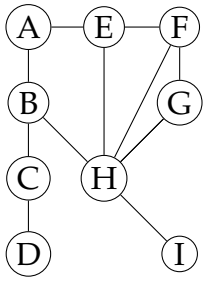
Given a text with n words, design an $O(n)$ expected time algorithm that computes the frequency of all k-grams that appear at least once in the text.

Solution 9. We use an approach similar Problem 6, but instead we use the k-grams in the text as our keys. To design a hash function for these keys we can use polynomial evaluation (since the order of the words of the k-gram matters).

Note that there can be at most $n - k + 1$ distinct k-grams in the text, so keeping a hash table of size $2n$ using linear probing yields the desired running time.

Warm-up

Problem 1. Consider the following undirected graph.



a) Starting from A, give the layers the breadth-first search algorithm finds.

b) Starting from A, give the order in which the depth-first search algorithm visits the vertices.

Solution 1.

a) $L_0 = \{A\}$
$L_1 = \{B, E\}$
$L_2 = \{C, F, H\}$
$L_3 = \{D, G, I\}$

b) There are a number of different solution possible, when the algorithm can choose multiple nodes to recurse on. This example picks the lexicographically first node, but other choices are just as valid.
$A, B, C, D, H, E, F, G, I$

Problem solving

Problem 2. An undirected graph $G = (V, E)$ is said to be bipartite if its vertex set V can be partitioned into two sets A and B such that $E \subseteq A \times B$. Design an $O(n + m)$ algorithm to test if a given input graph is bipartite using the following guide:

a) Suppose we run BFS from some vertex $s \in V$ and obtain layers $L_1, \ldots, L_k$. Let (u, v) be some edge in E. Show that if $u \in L_i$ and $v \in L_j$ then $|i - j| \leq 1$.

b) Suppose we run BFS on G. Show that if there is an edge (u, v) such that u and v belong to the same layer then the graph is not bipartite.

c) Suppose G is connected and we run BFS. Show that if there are no intra-layer edges then the graph is bipartite.

d) Put together all the above to design an $O(n + m)$ time algorithm for testing bipartiness.

Solution 2.

a) If $i = j$ then we are done. Otherwise, assume without loss of generality that u is discovered by BFS before v; in other words, assume $i < j$. When processing u, BFS scans the neighborhood of u. Since v ends up in layer L_j and $j > i$, this means that v had not been discovered yet at the time we started processing u. But then that means that v will be placed in the next layer since (u, v) is an edge; in other words, $j = i + 1$, and the property follows.

b) Suppose $u, v \in L_i$ and $(u, v) \in E$. Then we can form an odd cycle by taking the shortest path from s to u, the edge (u, v) and the shortest path from v to s; this cycle has length $2i + 1$. Now if the graph was bipartite the vertices in the cycle should alternate between the A set and the B set, but that is not possible if the cycle has odd length.

c) Let A be the even layers $L_0, L_2, \ldots$, and let B be the odd layers $L_1, L_3, \ldots$. Because the graph is connected, every vertex belongs to some layer, so (A, B) is a partition of V. Because there are no intra-layer edges, we have $E \subseteq A \times B$.

d) Given an input graph G, find its connected components. For each component, we run BFS and check if there are intra-layer edges; if not, partition the vertices into odd and even layer vertices.
The correctness of the algorithm follows readily from the previous tasks.
We use DFS to compute the connected components of G, which takes $O(n + m)$ time. Since these connected components together form the entire graph, running BFS on each of them has total running time $O(n + m)$. (Note that we can't say much about the running time per connected component, since they may vary from $O(1)$ to $\Omega(n)$ in size.) Checking for each edge if its endpoints are in different layers takes $O(1)$ time per edge, so $O(m)$ time in total. Hence, the running time is dominated by the DFS and BFS computations, which take $O(n + m)$ time.

Problem 3. Give an $O(n)$ time algorithm to detect whether a given undirected graph contains a cycle. If the answer is yes, the algorithm should produce a cycle. (Assume adjacency list representation.)

Solution 3. We run DFS with a minor modification. Every time we scan the neighborhood of a vertex u, we check if the neighbor v has been discovered before and whether it is different than u's parent. If we can find such a vertex then we have our cycle: $v, u, \text{parent}[u], \text{parent}[\text{parent}[u]], \ldots, v$.

We only need to argue that this algorithm runs in $O(n)$ time. Consider the execution of the algorithm up the point when we discovered the cycle. After the $O(n)$ time spent initializing the arrays needed to run DFS, each call to DFS-VISIT takes time that is proportional to the edges discovered. However, up until the time we find the cycle we have only discovered tree edges. So the total number of edges is upper bounded by $n - 1$. Thus, the overall running time in $O(n)$.

Problem 4. Let $G = (V, E)$ be an n vertex graph. Let s and t be two vertices. Argue that if $\text{dist}(s, t) > n/2$ then there there exists a vertex $u \neq s, t$ such that every path from s to t goes through u.

Solution 4. Run BFS starting from s. Let $L_0, L_1, \ldots$ be the layers discovered by the algorithm. Recall that $L_0 = \{s\}$ and let L_k be the layer t belongs to. By our assumption that $\text{dist}(s,t) > n/2$, it follows that $k > n/2$. We claim that there is a layer L_i for some $1 < i < k$, that has a single node; we call this a small layer. Otherwise, i.e., if all layers between L_0 and L_k had two or more vertices, then

$$\sum_{i=1}^{k-1} |L_i| \geq 2(k-1) > 2\,(n/2-1) = n-2.$$

But this cannot be, since there are at most $n-2$ vertices between s and t.

Now that we know that there is a small layer, let's see how we can use this. Let u be the vertex in small layer L_i. We claim that u is in every s-t path. Suppose, for the sake of contradiction, that there is a path from s to t that does not go through u. Such a path must have an edge (x,y) where $x \in L_a$ and $y \in L_b$ and $a < i < b$. But this cannot be since when BFS visited x, it should have added y to layer L_{a+1}, whereas $a < i < b$ implies that $b - a > 1$. A contradiction. Hence, u must appear in every s-t path.

Problem 5. In a directed graph, a *get-stuck* vertex has in-degree $n-1$ and out-degree 0. Assume the adjacency matrix representation is used. Design an $O(n)$ time algorithm to test if a given graph has a get-stuck vertex. Yes, this problem can be solved without looking at the entire input matrix.

Solution 5. Let A be the adjacency matrix of the graph; assume vertices are labeled $1, \ldots, n$. For ease of explanation, we say that an entry (u,v) of the matrix is 1 if there is an edge from u to v, and 0 otherwise. If i is a get-stuck vertex then the ith row of A is the all zeros vector, and the ith column of A is the all ones vector with the exception of its ith entry, which is zero.

Now starting on the top left corner of the matrix consider the following walk:

1. If we are in some entry (j,j), then we move right.
2. Otherwise, if we are at (i,j) for $i \neq j$, then we move down if we see a 1 and we move right if we see a 0.

The walk ends when we would exit the matrix (i.e., try to access a non-existing cell), either through the right side of (n,n) or the right side of another cell.

We claim that if there is a get-stuck vertex i we will leave the matrix through the right side and on i's row. To see this, we first observe that our walk always stays above the diagonal of the matrix. This implies that we will eventually reach column i. Since column i consists of only ones above the diagonal, when our walk reaches it, we will move down until we reach (i,i). Next, because the whole row i is made up of zeros, we'd move right at all of these entries, thus exiting through the right side on the ith row.

If there is no get-stuck vertex we will still fall through the right side, so the row we exit at should still be checked.

An $O(n)$ time algorithm for finding a get-stuck vertex would then perform the walk. If the walk exists through the right side on row i, then we check if i is indeed a get-stuck vertex (we only need to check $2(n-1)$ entries of A) and output

accordingly. Hence, the algorithm inspects only $O(n)$ entries and spends $O(1)$ time per entry, thus the algorithm runs in $O(n)$ time in total.

Problem 6. Let G be an undirected graph with vertices numbered $1 \dots n$. For a vertex i define $\text{small}(i) = \min\{j : j \text{ is reachable from } i\}$, that is, the smallest vertex reachable from i. Design an $O(n + m)$ time algorithm that computes $\text{small}(i)$ for *every* vertex in the graph.

Solution 6. *(Sketch)* Run DFS. Each tree in the DFS forest is a connected component in the graph. For each component C find the node $a \in C$ with the smallest label; for each vertex $b \in C$, set $\text{small}(b) = a$.

The algorithm is correct as every vertex in a connected component can reach the same set of vertices, namely, the whole connected component.

Running DFS takes $O(n + m)$ time. After that doing the scan of each component and setting the small values takes $O(n)$ time. Thus the total running time is $O(n + m)$.

Problem 7. In a connected undirected graph $G = (V, E)$, a vertex $u \in V$ is said to be a cut vertex if its removal disconnects G; namely, $G[V - u]$ is not connected.

The aim of this problem is to adapt the algorithm for cut edges from the lecture, to handle cut vertices.

a) Derive a criterion for identifying cut vertices that is based on the `down-and-up`$[\cdot]$ values defined in the lecture.

b) Use this criterion to develop an $O(n + m)$ time algorithm for identifying all cut vertices.

Solution 7. Recall that the main idea for the cut edges algorithm, was to run DFS on G and for every $v \in V$ set $\text{level}(v)$ of the vertex v in the DFS tree. Then we defined down-and-up(u) be the highest level (closer to the root) we can reach by taking any number of DFS tree edges down and one single back edge up.

a) First note that the leaves of the DFS tree cannot be cut vertices, since the rest of the DFS tree keeps the graph together.
Secondly, note that if the root has two or more children, then it must be a cut vertex since there are no edges connecting its children subtrees (we only have DFS tree edges or back edges connecting a node to one of its ancestors).
Finally, consider an internal node u other than the root. Note that after we remove u, it may be the case that the subtree rooted at one of its children gets disconnected from the rest of the graph. Let v be one of u's children. If $\textsc{down-and-up}(v) < \textsc{level}(u)$ then after we remove u the subtree rooted at v remains connected by the edge that connects some descendant of v to some ancestor of u. Otherwise, if $\textsc{down-and-up}(v) \geq \textsc{level}(u)$ then u is a cut vertex.

b) The algorithm first runs DFS and computes level(u) and DOWN-AND-UP(u) for all $u \in V$. If the root has two or more children, it is declared a cut vertex. For any other internal node u, we check if it has a child v such that DOWN-AND-UP$(v) \geq$ LEVEL(u). If this is the case, we declare u to be a cut vertex.
The correctness of the algorithm rests on the observations made in the previous point.
The running time is dominated by running DFS and computing LEVEL and DOWN-AND-UP, which can be done in $O(n+m)$ time.

Problem 8. Let T be a rooted tree. For each vertex $u \in T$ we use T_u to denote the subtree of T made up by u and all its descendants. Assume each vertex $u \in T$ has a value $A[u]$ associated with it. Let $B[u] = \min\{A[v] : v \in T_u\}$. Design an $O(n)$ time algorithm that given A, computes B.

Solution 8. Perform a post-order traversal of the tree (you can do this in a graph just as well as in the tree data structure, as long as you keep track from which edge you used when you first arrived at a vertex). Recall that this means that for each vertex u, you first visit the subtree of each child before visiting u. When visiting u, compute $B[u]$ as the minimum of $A[u]$ and $B[v]$ for each child v of u.

The correctness of the algorithm rests on the following observation

$$B[u] = \min_{x \in T_u} A[u] = \min(A[u], \min_{x \in T_{v_1}} A[x], \dots \min_{x \in T_{v_k}} A[x]) = \min(A[u], B[v_1], \dots, B[v_k]),$$

where $v_1, \dots, v_k$ are the children of u.

The time complexity is $O(n)$ because of the amount of work done at each vertex u is $O(|N(u)|)$. Adding up over all vertices we get that the overall running time is $O(n)$ because $\sum_{u \in T} |N(u)| = 2(n-1)$.

Problem 9. Let G be a connected undirected graph. Design a linear time algorithm for finding all cut edges by using the following guide:

a) Derive a criterion for identifying cut edges that is based on the `down-and-up`$[\cdot]$ values defined in the lecture.

b) Use this criterion to develop an $O(n+m)$ time algorithm for identifying all cut edges.

Solution 9.

a) First note that only DFS-tree edges can be cut edges because other edges can never disconnect the graph (the DFS tree keeps the graph in one piece).
Let (u, v) be a DFS-tree edge. Assume that u is v's parent in the DFS tree. If (u, v) is not a cut edge then some vertex in T_v (the subtree rooted at v) must have a back edge to u or a higher vertex; i.e., if DOWN-AND-UP$(v) \leq$ LEVEL(u).

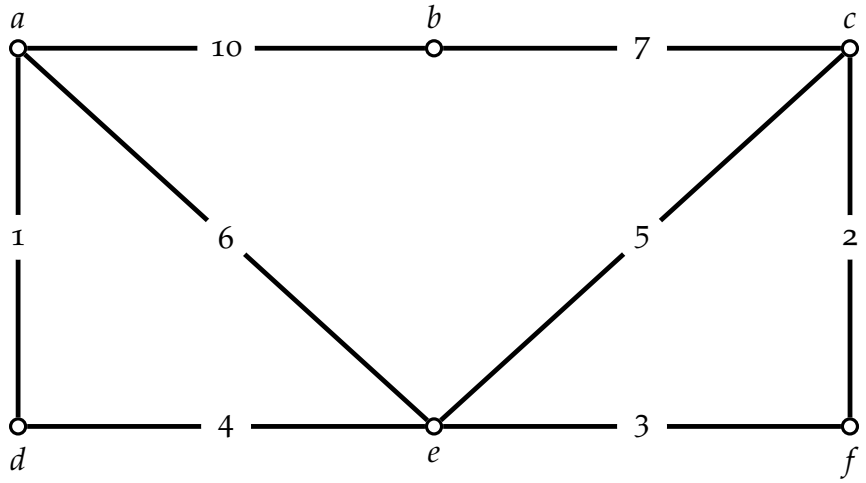
b) The algorithm first runs DFS and computes UP$[u]$ at the highest level up the tree that we can reach from u by taking a back edge. Then we compute DOWN-AND-UP$(u) = \min_{v \in T_u}$ UP(v) for all $u \in V$ (see previous problem). Then for each DFS-tree edge (u, v), where v is a child of u, the algorithm declares (u, v) to be a cut edge if DOWN-AND-UP$(v) >$ LEVEL (u).
The correctness of the algorithm rests on observations made in the previous point.
The running time is dominated by running DFS and computing LEVEL, UP and DOWN-AND-UP, which can be done in $O(n + m)$ time.

Warm-up

Problem 1. Consider Dijkstra's shortest path algorithm for undirected graphs. What changes (if any) do we need to make to this algorithm for it to work for directed graphs and maintain its running time?

Solution 1. The only thing we need to ensure is that when updating $D[z]$, we need to consider only the vertices where there is a directed edge from u to z. In the pseudocode in the lecture, this is implicitly handled by only considering the neighbors of u (i.e., the vertices that can be reached by following an edge from u), so we don't need to change anything.

Problem 2. Consider the following weighted undirected graph G:



Your task is to compute a minimum weight spanning tree T of G:

a) Which edges are part of the MST?

b) In which order does Kruskal's algorithm add these edges to the solution.

c) In which order does Prim's algorithm (starting from a) add these edges to the solution.

Solution 2.

a) $(a,d),(b,c),(c,f),(d,e),(e,f)$

b) The edges are considered in order of their weight, so they are added in the order: $(a,d),(c,f),(e,f),(d,e),(b,c)$. Note that when (c,e) is considered (c,f) and (e,f) are already present, so it isn't added. Similarly, (a,e) and (a,b) aren't added.

c) Prim's algorithm grows the MST from the starting vertex, each time adding the cheapest edge that connects the MST to a new vertex, so the edges are added in the order: $(a,d),(d,e),(e,f),(c,f),(b,c)$.

Problem solving

Problem 3. Let $G = (V, E)$ be an undirected graph with edge weights $w : E \to R^+$. For all $e \in E$, define $w_1(e) = \alpha w(e)$ for some $\alpha > 0$, $w_2(e) = w(e) + \beta$ for some $\beta > 0$, and $w_3(e) = w(e)^2$.

a) Suppose p is a shortest s-t path for the weights w. Is p still optimal under w_1? What about under w_2? What about under w_3?

b) Suppose T is minimum weight spanning tree for the weights w. Is T still optimal under w_1? What about under w_2? What about under w_3?

Solution 3.

a) For w_1 we note that for any two paths p and p' if $w(p) \leq w(p')$ then

$$w_1(p) = \alpha w(p) \leq \alpha w(p') = w_1(p').$$

so if a path is shortest under w it remains shortest under w_1.
This is now longer the case with w_2 and w_3. Consider a graph with three vertices and three edges, basically a cycle. Suppose one edge has weight 1 and the other edges have weight $1/2$. Consider the shortest path between the endpoints of the edge with weight one. The two possible paths have total weight 1, so both of them are shortest. Now imagine adding 1 to all edge weights; that is, consider w_2 with $\beta = 1$. The shortest path in w_2 is unique—the edge with weight 1 in w. Now imagine squaring the edge weights; that is, consider w_3. The shortest path in w_3 is unique—the path with two edges of weight $1/2$, which under w_3 have weight $1/4$ each.

b) We claim that the optimal spanning tree does not change. This is because the relative order of the edge weights is the same under w, w_1, w_2, and w_3. Therefore, if we run Kruskal's algorithm the edges will be sorted in the same way and the output will be the same in all cases.

Problem 4. It is not uncommon for a given optimization problem to have multiple optimal solutions. For example, in an instance of the shortest s-t path problem, there could be multiple shortest paths connecting s and t. In such situations, it may be desirable to break ties in favor of a path that uses the fewest edges.

Show how to reduce this problem to a standard shortest path problem. You can assume that the edge lengths ℓ are positive integers.

a) Let us define a new edge function $\ell'(e) = M\ell(e)$ for each edge e. Show that if P and Q are two s-t paths such that $\ell(P) < \ell(Q)$ then $\ell'(Q) - \ell'(P) \geq M$.

b) Let us define a new edge function $\ell''(e) = M\ell(e) + 1$ for each edge e. Show that if P and Q are two s-t paths such that $\ell(P) = \ell(Q)$ but P uses fewer edges than Q then $\ell''(P) < \ell''(Q)$.

c) Show how to set M in the second function so that the shortest s-t path under ℓ'' is also shortest under ℓ and uses the fewest edges among all such shortest paths.

Solution 4.

a) Since the edge lengths are integer valued, if $\ell(P) < \ell(Q)$ then $\ell(Q) - \ell(P) \geq 1$. It follows then that $\ell'(Q) - \ell'(P) = M\ell(Q) - M\ell(P) = M(\ell(Q) - \ell(P)) \geq M$.

b) Let $|P|$ be the number of edges used in the path P. Then $\ell''(P) = M\ell(P) + |P|$. It follows that $\ell''(P) = M\ell(P) + |P| < M\ell(Q) + |Q| = \ell''(Q)$.

c) Set M to be the number of vertices in the graph.
From task (b) we know that if P is optimal under ℓ'' then it will have the fewest edges among paths that have length $\ell(P)$. Now suppose that there is another path Q such that $\ell(Q) < \ell(P)$; then $\ell(P) - \ell(Q) \geq 1$ and therefore $M\ell(P) \geq M + M\ell(Q)$. Notice that since $|Q| < n$, we have

$$M + M\ell(Q) > |Q| + M\ell(Q) = \ell''(Q)$$

and also

$$M\ell(P) \leq |P| + M\ell(P) \leq \ell''(P).$$

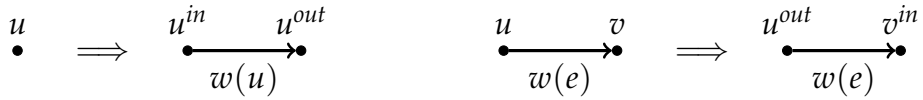
Putting all these inequalities together we get

$$\ell''(P) \geq M\ell(P) \geq M + M\ell(Q) > \ell''(Q)$$

which contradicts the optimality of P in ℓ''.

Problem 5. Consider the following generalization of the shortest path problem where in addition to edge lengths, each vertex has a cost. The objective is to find an s-t path that minimizes the total length of the edges in the path plus the cost of the vertices in the path. Design an efficient algorithm for this problem.

Solution 5. We reduce the problem of finding a path with minimum edges plus vertex weight to a standard shortest path problem with weights on the edges only. Before we do this, recall that an undirected graph can be modelled as a directed graph by replacing every undirected edge (u, v) by two directed edges (u, v) and (v, u). Hence, in the remainder, we'll focus on directed graphs. Given a directed graph $G = (V, E)$ and weights w we construct a new graph $G' = (V', E')$ and weights w'. For each vertex $u \in V$ we create two copies u^{in} and u^{out}, which we connect with an edge (u^{in}, u^{out}) with weight $w(u)$. For each edge $(u, v) \in E$ we create an edge (u^{out}, v^{in}) with weight $w(e)$.



It is trivial to check that for each path $p = \langle u_1, u_2, \ldots, u_k \rangle$ in the input graph G we have a path $p' = \langle u_1^{out}, u_2^{in}, u_2^{out}, \ldots, u_k^{in} \rangle$. Furthermore, the edge weight of p' equals the edge plus vertex weight of p.

It is easy to see that the graph G' can be constructed in $O(m+n)$ time. Furthermore, $|V'| = 2|V|$ and $|E'| = |V| + |E|$. Therefore Dijkstra's algorithm runs in $O(m + n \log n)$, where $m = |E|$ and $n = |V|$.

Problem 6. Consider the IMPROVING-MST algorithm for the MST problem.

```
1: function IMPROVING-MST(G, w)
2:     T ← some spanning tree of G
3:     for e ∈ E \ T do
4:         T ← T + e
5:         C ← unique cycle in T
6:         f ← heaviest edge in C
7:         T ← T − f
8:     return T
```

Prove its correctness and analyze its time complexity. To simplify things, you can assume the edge weights are distinct.

Solution 6. Let $T_0, T_1, T_2, \ldots, T_m$ be the trees kept by the algorithm in each of its m iterations. Consider some edge $(u, v) \in E$ that did not make it to the final tree. It could be that (u, v) was rejected right away by the algorithm, or it was in T for some time and then it was removed. Suppose this rejection took place on the ith iteration. Let p be the u-v in path in T_i. Since (u, v) was rejected, all edges in p have weight less than $w(u, v)$. It is easy to show using induction that this is true not only in T_i but in all subsequent trees T_j for $j \geq i$.

Let (x, y) be an edge in the final tree T_m. Remove (x, y) from T_m to get two connected components X and Y. The Cut Property states that if (x, y) is the lightest edge in the cut (X, Y) then every MST contains (x, y). Suppose for the sake of contradiction that there is some rejected edge $(u, v) \in \text{cut}(X, Y)$ such that $w(u, v) < w(x, y)$. This means that (u, v) is not the heaviest edge in the unique u-v path in T_m, which contradicts the conclusion of the previous paragraph. Thus, (x, y) belongs to every MST. Since this holds for every edge in T_m, it follows that T_m itself is an MST.

Regarding the time complexity, we note that finding the cycle in $T + e$ can be done $O(n)$ time using DFS. Similarly finding the heaviest edge and removing it takes $O(|C|) = O(n)$ time. There are m iterations, so the overall time complexity is $O(nm)$.

Problem 7. Consider the REVERSE-MST algorithm for the MST problem.

```
1: function REVERSE-MST(G, w)
2:     Sort edges in decreasing weight w
3:     T ← E
4:     for e ∈ E in decreasing weight do
5:         if T − e is connected then
6:             T ← T − e
7:     return T
```

Prove its correctness and analyze its time complexity. To simplify things, you can assume the edge weights are distinct.

Solution 7. Suppose e was one of the edges the algorithm kept. If e belongs to the optimal solution we are done, so assume for the sake of contradiction that this is not the case.

If the algorithm decided to keep e then it must be that $T - e$ was not connected, say $T - e$ had two connected components X and Y. Notice that all edges in the cut (X, Y) have a weight larger than $w(e)$. Therefore, we can take the optimal solution, add e to form a cycle C. This cycle must contain one edge f from cut(X, Y), so we could remove f and improve the cost of the optimal solution, a contradiction. Therefore, e must be part of the optimal solution. Since this holds for all e in the output the algorithm is optimal.

Regarding the time complexity, we note that checking connectivity can be done $O(n + m)$ time using DFS. There are m iterations, so the overall time complexity is $O(m(n + m))$, which is $O(m^2)$ assuming the graph G is connected.

Problem 8. A computer network can be modeled as an undirected graph $G = (V, E)$ where vertices represent computers and edges represent physical links between computers. The maximum transmission rate or *bandwidth* varies from link to link; let $b(e)$ be the bandwidth of edge $e \in E$. The bandwidth of a path P is defined as the minimum bandwidth of edges in the path, i.e., $b(P) = \min_{e \in P} b(e)$.

Suppose we number the vertices in $V = \{v_1, v_2, \ldots, v_n\}$. Let $B \in R^{n \times n}$ be a matrix where B_{ij} is the maximum bandwidth between v_i and v_j. Give an algorithm for computing B. Your algorithm should run in $O(n^3)$ time.

Solution 8. Let T be a maximum weight spanning tree, that is, the spanning tree maximizing $w(T) = \sum_{e \in T} w(e)$. We claim that in this tree for any two vertices $u, v \in V$, the unique u-v path in T is a maximum bandwidth path. Indeed, assume that there is a pair of vertices u and v contradicting our claim. That is, let e be the minimum bandwidth edge in the unique u-v path in T and let p be an alternative u-v path p in the graph whose bandwidth is $b(p) > w(e)$. Now remove e from T breaking the tree into two connected components A and B having u on one side and v on the other. Then the path p must contain at least one edge, call it f, with one endpoint in A and another endpoint in B. Since $b(p) > w(e)$ then $w(f) > w(e)$, and so $T - e + f$ is also spanning tree with greater weight. A contradiction.

The maximum spanning tree can be computed by modifying Prim's or Kruskal's minimum spanning tree algorithm. For Prim's algorithm, instead of picking the lightest edge in every iteration, we pick the heaviest edge. For Kruskal's algorithm, we sort the edges in decreasing weight.

After computing T we could loop over every pair of vertices v_i and v_j, computing the lightest edge in the unique v_i-v_j path in T. This takes $O(n)$ time per pair of vertices, so we get overall $O(n^3)$ time.

With a better approach, you can also do the final computation in $O(n^2)$ time. After computing T we can loop over the vertices. For each starting vertex v_i we will recursively visit its neighbours in T, keeping track of the minimum weight edge traversed so far. At each vertex v_j we visit, where W is the minimum weight

seen so far, we will update B_{ij} to W, then visit all unvisited neighbours of v_j in T, using $\min\{W, w(v_j, u)\}$ as the minimum weight seen so far at u. Since T contains $O(n)$ vertices and edges, each traversal will take $\sum_{v \in V} O(\deg_T(v)) = O(n)$ time, and so all n traversals together will take $O(n^2)$ time in total. Since both Prim's and Kruskal's algorithm can be run in less than $O(n^2)$ time, the total running time is $O(n^2)$.

Warm-up

Problem 1. Construct the Huffman tree of the following phrase: "data structure"

Solution 1. We first determine the frequency of each character in the phrase (including the space):

character	frequency
a	2
c	1
d	1
e	1
r	2
s	1
t	3
u	2
space	1

After creating a single-node tree for each of these, the Huffman tree algorithm repeatedly merges the two trees with smallest total frequency. Assuming ties are broken lexicographically, it performs the following merges:

1. *c* and *d* form a new tree *cd* with frequency 2
2. *e* and *s* form a new tree *es* with frequency 2
3. space and *a* form a new tree *a_* with frequency 3
4. *cd* and *es* form a new tree *cdes* with frequency 4
5. *r* and *u* form a new tree *ru* with frequency 4
6. *a_* and *t* form a new tree *at_* with frequency 6
7. *cdes* and *ru* form a new tree *cdersu* with frequency 8
8. and finally *cdersu* and *at_* form a new tree with frequency 14

The resulting tree can be found below:

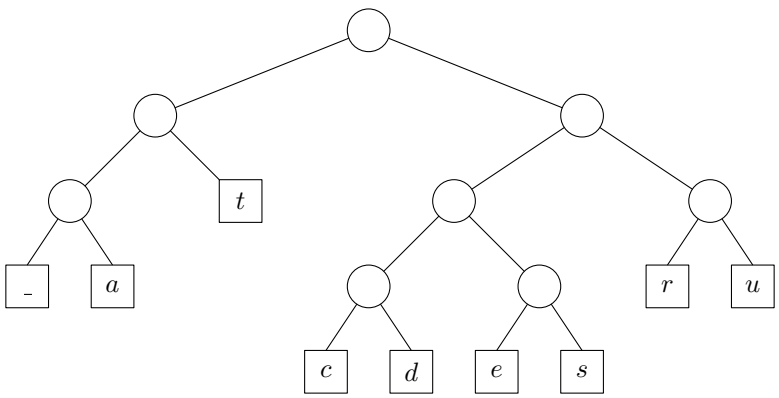


Figure 1: The resulting Huffman tree.

Problem 2. During the lecture we saw that the Fractional Knapsack algorithm's correctness depends on which greedy choice we make. We saw that there are instances where always picking the "highest benefit" or always picking the "smallest weight" doesn't give the optimal solution.

For each of these two strategies, give an infinite class of instances where the algorithm gives the optimal solution, thus showing that *you can't show correctness of an algorithm by using a finite number of example instances*.

Solution 2. The trivial solution that works for both strategies is any instance consisting of a single item with weight at most W (or any instance where the sum of the weights of all items is at most W).

A less trivial solution where not all items fit in the knapsack could for example be the following. For the "highest benefit" strategy, we construct the following class of instances: We have n items (numbered 0 to $n-1$) and item i has benefit $b_i = c \cdot 2^i$ (for and $c > 0$) and weight $w_i = 2^i$. For any integer W, picking the highest benefit items that fit in the knapsack corresponds to picking those i that are 1 in the binary representation of W, filling the knapsack entirely. The solution will have benefit $c \cdot W$.

For the "smallest weight" strategy, we can take the same items when we restrict the values of W. Specifically, by picking W such that $W = \sum_{j=0}^{k} 2^j$ for some positive integer k, we ensure that the k least significant bits of the binary representation of W are all ones and all others are zeroes. The "smallest weight" strategy picks the items starting from the smallest weight (i.e., the least significant bit) and solution will have benefit $c \cdot W$.

Problem solving

Problem 3. Given vectors $a, b \in R^n$, consider the problem of finding a permutation a' and b' of a and b respectively that minimizes

$$\sum_{1 \le i \le n} |a_i - b_i|.$$

Prove or disprove the correctness (i.e., that it always returns the optimal solution) of the following algorithm that greedily tries to pair up similar coordinates.

```
1:  function GREEDY-MATCHING-V1(a, b)
2:      A ← multiset of values in a
3:      B ← multiset of values in b
4:      a' ← new empty list
5:      b' ← new empty list
6:      while A is not empty do
7:          x, y ← a pair in (A, B) minimizing |x − y|
8:          Append x to a'
9:          Append y to b'
10:         Remove x from A and y from B
```

```
11:     return (a', b')
```

Solution 3. The algorithm does not always return the optimal solution. Consider the instance $A = (1, 4)$ and $B = (3, 6)$. Since the difference between 3 and 4 is the pair that minimizes their difference, the algorithm ends up with the following pairs: $(3, 4)$ and $(1, 6)$. These pairs have total difference of 6, where the optimal solution would have a difference of 4 using $(1, 3)$ and $(4, 6)$.

Problem 4. Given vectors $a, b \in R^n$, consider the problem of finding a permutation a' and b' of a and b respectively that minimizes

$$\sum_{1 \le i \le n} |a_i - b_i|.$$

Prove or disprove the correctness (i.e., that it always returns the optimal solution) of the following algorithm that greedily tries to pair up similar coordinates.

```
1:  function GREEDY-MATCHING-V2(a, b)
2:      A ← multiset of values in a
3:      B ← multiset of values in b
4:      a' ← new empty list
5:      b' ← new empty list
6:      while A is not empty do
7:          x ← smallest in A
8:          y ← smallest in B
9:          Append x to a'
10:         Append y to b'
11:         Remove x from A
12:         Remove y from B
13:     return (a', b')
```

Solution 4. Note that the algorithm sorts the coordinates of a and b in non-decreasing order. In this case, the algorithm is optimal.

Let a^* and b^* be the optimal solution. We use an exchange argument. First note that we can assume that a^* is listed in non-decreasing order, because applying the same permutation to both a^* and b^* does not change the value of the objective. If b^* is not in sorted order, there must be an inversion, i.e., an index i such that $b^*_i > b^*_{i+1}$. The contribution of the indices i and $i + 1$ to the objective is

$$|a^*_i - b^*_i| + |a^*_{i+1} - b^*_{i+1}|$$

since $a^*_i \le a^*_{i+1}$, a case analysis argument (which is a bit involved, so we defer this to the end of the proof) can be used to show that

$$|a^*_i - b^*_i| + |a^*_{i+1} - b^*_{i+1}| \ge |a^*_i - b^*_{i+1}| + |a^*_{i+1} - b^*_i|,$$

therefore, by swapping the entries b^*_i and b^*_{i+1} we have one fewer inversion without increasing the objective value.

We can keep on performing this exchange argument until b^* is sorted. Thus, our algorithm is optimal.

It remains to prove that $|a_i^* - b_i^*| + |a_{i+1}^* - b_{i+1}^*| \geq |a_i^* - b_{i+1}^*| + |a_{i+1}^* - b_i^*|$. This is equivalent to proving that $|a_i^* - b_i^*| + |a_{i+1}^* - b_{i+1}^*| - (|a_i^* - b_{i+1}^*| + |a_{i+1}^* - b_i^*|) \geq 0$. Recall that we know that $a_i^* \leq a_{i+1}^*$ and $b_i^* > b_{i+1}^*$.

- Case 1: $a_i^* \geq b_i^*$ and $a_{i+1}^* \geq b_{i+1}^*$, which implies $b_{i+1}^* < b_i^* \leq a_i^* \leq a_{i+1}^*$. In this case $|a_i^* - b_i^*| + |a_{i+1}^* - b_{i+1}^*| - (|a_i^* - b_{i+1}^*| + |a_{i+1}^* - b_i^*|)$ can be rewritten to $a_i^* - b_i^* + a_{i+1}^* - b_{i+1}^* - (a_i^* - b_{i+1}^* + a_{i+1}^* - b_i^*)$. We can see that for every term occur both positively and negatively, thus this sums to 0.

- Case 2: $a_i^* \geq b_i^*$ and $b_{i+1}^* > a_{i+1}^*$. We have that $a_i^* \geq b_i^* > b_{i+1}^*$ and that $b_{i+1}^* > a_{i+1}^* \geq a_i^*$. These two contradict each other, so this case can't occur.

- Case 3: $b_i^* > a_i^*$ and $a_{i+1}^* \geq b_{i+1}^*$, which implies $b_{i+1}^* \leq a_i^* \leq a_{i+1}^* \leq b_i^*$ (and at least one of these is strict inequality). In this case $|a_i^* - b_i^*| + |a_{i+1}^* - b_{i+1}^*| - (|a_i^* - b_{i+1}^*| + |a_{i+1}^* - b_i^*|)$ can be rewritten to $b_i^* - a_i^* + a_{i+1}^* - b_{i+1}^* - (a_i^* - b_{i+1}^* + b_i^* - a_{i+1}^*)$. This is equal to $2a_{i+1}^* - a_i^*$, which is at least 0, since $a_i^* \leq a_{i+1}^*$.

- Case 4: $b_i^* > a_i^*$ and $b_{i+1}^* > a_{i+1}^*$, which implies $a_i^* \leq a_{i+1}^* < b_{i+1}^* < b_i^*$. In this case $|a_i^* - b_i^*| + |a_{i+1}^* - b_{i+1}^*| - (|a_i^* - b_{i+1}^*| + |a_{i+1}^* - b_i^*|)$ can be rewritten to $b_i^* - a_i^* + b_{i+1}^* - a_{i+1}^* - (b_{i+1}^* - a_i^* + b_i^* - a_{i+1}^*)$. We can see that for every term occur both positively and negatively, thus this sums to 0.

Hence, in all cases we have that $|a_i^* - b_i^*| + |a_{i+1}^* - b_{i+1}^*| - (|a_i^* - b_{i+1}^*| + |a_{i+1}^* - b_i^*|) \geq 0$ and thus $|a_i^* - b_i^*| + |a_{i+1}^* - b_{i+1}^*| \geq |a_i^* - b_{i+1}^*| + |a_{i+1}^* - b_i^*|$.

Problem 5. Suppose we are to construct a Huffman tree for a string over an alphabet $C = c_1, c_2, \ldots, c_k$ with frequencies $f_i = 2^i$. Prove that every internal node in T has an external-node child.

Solution 5. We prove by induction that at the start of the ith iteration of the algorithm, the set of trees the algorithm has is $\{c_1, \ldots c_i\}, \{c_{i+1}\}, \ldots \{c_k\}$. The base case $i = 1$ is trivial, since all trees are singletons. For the induction step, assume that after iteration $i-1$ we have $\{c_1, \ldots c_{i-1}\}, \{c_i\}, \ldots \{c_k\}$. In the ith iteration, the algorithm will merge $\{c_1, \ldots c_{i-1}\}$ and $\{c_i\}$. This is because $\sum_{1 \leq j < i} f_j = f_i - 2$.

Therefore, in the ith iteration the algorithm creates a new internal node that has the external node c_i as a child.

Problem 6. Design a greedy algorithm for the following problem (see Figure 2): Given a set of n points $\{x_1, ..., x_n\}$ on the real line, determine the smallest set of unit-length intervals that contains all points.

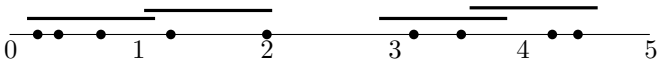


Figure 2: Covering points with unit intervals.

Solution 6. We sort the points from left to right. We start a new unit-length interval at the leftmost point and we find the leftmost point p that isn't covered by this interval. From p we start another interval. This process of finding the leftmost uncovered point and starting a new interval there is repeated until all points are covered.

In order to see that the algorithm is correct, we start from any optimal solution σ and convert it to our greedy solution. Let I be the leftmost interval (the interval with the leftmost left endpoint) of σ that is different from the greedy intervals. We shift interval I to the right until its left endpoint is at a point without leaving any points uncovered in the process (if there are points to its left, these are covered by intervals to the left of I, since those match the greedy solution). This way we can convert every segment in σ that differs from the greedy solution to segment in the greedy solution.

The running time of this algorithm is dominated by the sorting step, which takes $O(n \log n)$ time. Once the input is sorted, the rest of the algorithm just scans through the input, performing only $O(1)$ time operations in the process (checking if a point is contained in the current interval, and starting a new interval when needed).

Problem 7. Suppose we are to schedule print jobs on a printer. Each job j has an associated weight $w_j > 0$ (representing how important the job is) and a processing time t_j (representing how long the job takes). A schedule σ is an ordering of the jobs that tells the printer in which order to process the jobs. Let C_j^σ be the completion time of job j under the schedule σ.

Design a greedy algorithm that computes a schedule σ minimizing the sum of weighted completion times, that is, minimizing $\sum_j w_j C_j^\sigma$.

Solution 7. An optimal schedule can be obtained by sorting the jobs in increasing $\frac{t_j}{w_j}$ order; assume for simplicity that no two jobs have the same ratio.

To show that the schedule is optimal, we use an exchange argument. Suppose the optimal solution is σ and there are two consecutive jobs, say i and k such that $\frac{t_i}{w_i} > \frac{t_k}{w_k}$. We build another schedule τ where we swap the positions of i and k, and leave the other jobs unchanged. We need to argue that this change decreases the cost of the schedule. Notice that the completion time of jobs other than i and k is the same in σ and τ. Thus,

$$\sum_j w_j C_j^\sigma - \sum_j w_j C_j^\tau = w_i C_i^\sigma + w_k C_k^\sigma - w_i C_i^\tau - w_k C_k^\tau. \tag{1}$$

Let $X = C_i^\sigma - t_i$, the time when job i starts in σ, which equals the time when job k starts in τ. Then $C_i^\sigma = X + t_i$, $C_k^\sigma = X + t_i + t_k$, $C_k^\tau = X + t_k$, and $C_i^\tau = X + t_i + t_k$. If we plug these values into (1) and simplify, we get

$$\sum_j w_j C_j^\sigma - \sum_j w_j C_j^\tau = -w_i t_k + w_k t_i. \tag{2}$$

The proof is finished by noting that $\frac{t_i}{w_i} > \frac{t_k}{w_k}$ implies $-w_i t_k + w_k t_i > 0$ and

therefore

$$\sum_j w_j C_j^\sigma > \sum_j w_j C_j^\tau.$$

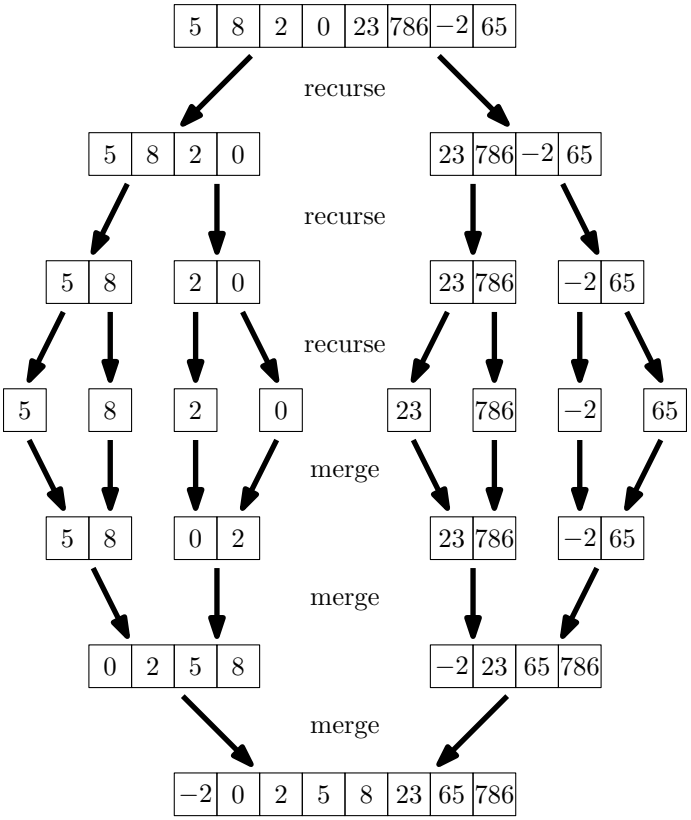
Hence, the schedule σ is not optimal.

Finally, for the running time analysis, we note that sorting the jobs takes $O(n \log n)$ time. Once the jobs are sorted, we only need to output them in that order (which takes $O(n)$ time), hence the algorithm takes $O(n \log n)$ time.

Warm-up

Problem 1. Sort the following array using merge-sort: $A = [5, 8, 2, 0, 23, 786, -2, 65]$. Give all arrays on which recursive calls are made and show how they are merged back together.

Solution 1. Below all arrays are shown. Until all arrays have size 1, all splits represent recursive calls. Since arrays of size 1 are sorted, after this the arrays are merged back together in order to create the final sorted array.



Problem 2. Consider the following algorithm.

```
1: function REVERSE(A)
2:     if |A| = 1 then
3:         return A
4:     else
5:         B ← first half of A
6:         C ← second half of A
7:         return concatenate REVERSE(C) with REVERSE(B)
```

Let $T(n)$ be the running time of the algorithm on an instance of size n. Write down the recurrence relation for $T(n)$ and solve it by unrolling it.

Solution 2. The divide step takes $O(n)$ time when we explicitly copy it. For the recursion, we recurse on two parts of size $n/2$ each. The conquer step concatenates the two arrays, which involves copying over at least one of the two, so this takes $O(n)$ time. Solving a base case when $n = 1$ takes constant time, since we just return the single element array.

Hence, the recursion is:

$$T(n) = \begin{cases} T(n) = 2T(n/2) + O(n) & \text{for } n > 1 \\ O(1) & \text{for } n = 1 \end{cases}$$

which solves to

$$T(n) = 2 \cdot T\left(\frac{n}{2}\right) + O(n) = c \cdot n + 2 \cdot c \cdot \frac{n}{2} + 4 \cdot c \cdot \frac{n}{4} + ... = O(n \log n)$$

Problem solving

Problem 3. Given an array A holding n objects, we want to test whether there is a *majority* element; that is, we want to know whether there is an object that appears in more than $n/2$ positions of A.

Assume we can test equality of two objects in $O(1)$ time, but we cannot use a dictionary indexed by the objects. Your task is to design an $O(n \log n)$ time algorithm for solving the majority problem.

a) Show that if x is a majority element in the array then x is a majority element in the first half of the array or the second half of the array

b) Show how to check in $O(n)$ time if a candidate element x is indeed a majority element.

c) Put these observation together to design a divide an conquer algorithm whose running time obeys the recurrence $T(n) = 2T(n/2) + O(n)$

d) Solve the recurrence by unrolling it.

Solution 3.

a) If x is a majority element in the original array then, by the pigeon-hole principle, x must be a majority element in at least one of the two halves. Suppose that n is even. If x is a majority element at least $n/2 + 1$ entries equal x. By the pigeon hole principle either the first half or the second half must contain $n/4 + 1$ copies of x, which makes x a majority element within that half.

b) We scan the array counting how many entries equal x. If the count is more than $n/2$ we declare x to be a majority element. The algorithm scans the array and spends $O(1)$ time per element, so $O(n)$ time overall.

c) If the array has a single element, we return that element as the majority element. Otherwise, we break the input array A into two halves, recursively call the algorithm on each half, and then test in A if either of the elements returned by the recursive calls is indeed a majority element of A. If that is the case we return the majority element, otherwise, we report that there is "no majority element".
To argue the correctness, we see that if there is no majority element of A then the algorithm must return "no majority element" since no matter what the recursive call returns, we always check if an element is indeed a majority element. Otherwise, if there is a majority element x in A we are guaranteed that one of our recursive calls will identify x (by part a)) and the subsequent check will lead the algorithm to return x.
Regarding time complexity, breaking the main problem into the two subproblems and testing the two candidate majority elements can be done in $O(n)$ time. Thus we get the following recurrence

$$T(n) = \begin{cases} T(n) = 2T(n/2) + O(n) & \text{for } n > 1 \\ O(1) & \text{for } n = 1 \end{cases}$$

d)

$$T(n) = c \cdot n + 2 \cdot c \cdot \frac{n}{2} + 4 \cdot c \cdot \frac{n}{4} + ... = O(n \log n).$$

Problem 4. Let A be an array with n distinct numbers. We say that two indices $0 \leq i < j < n$ form an inversion if $A[i] > A[j]$. Modify merge sort so that it computes the number of inversions of A.

Solution 4. We augment the standard merge sort algorithm so that in addition to returning the sorted input array, it also returns the number of inversions. If our array consists of a single element, there can't be any inversions, so we return the array and 0.

Let L be the sorted left half of the input array and R be the sorted right half of the input array. We can count on the recursive calls to count the inversions within L and within R, but we need to modify the merge procedure to count the number of inversions (i, j) where i is in the first half and j is in the second half.

For each element $e \in R$ let $L(e)$ be the number of elements in L that are greater than e. Note that the number of inversions (i, j) where i is in the first half of the input array and j is in the second half of the input array equals $\sum_{e \in R} L(e)$. Furthermore, $L(e)$ can be computed on the fly when e is added to the merged array since elements that remain in L at that time are precisely those elements of L that are greater than e. The pseudocode of this algorithm is given below.

```
1: function MERGE-AND-COUNT(L, R)
2:     result ← new array of length |L| + |R|
```

```
    count ← 0
    l, r ← 0, 0
    while l + r < |result| do
        index ← l + r
        if r ≥ |R| or (l < |L| and L[l] < R[r]) then
            result[index] ← L[l]
            l ← l + 1
        else
            result[index] ← R[r]
            count ← count + |L| − l
            r ← r + 1
    return result, count
```

```
function COUNT-INVERSIONS(A)
    if |A| = 1 then
        return A, 0
    else
        B ← first half of A
        C ← second half of A
        sortedB, countB ← COUNT-INVERSIONS(B)
        sortedC, countC ← COUNT-INVERSIONS(C)
        sortedBC, countBC ← MERGE-AND-COUNT(sortedB, sortedC)
        return sortedBC, countB + countC + countBC
```

We focus on the correctness of computing the inversions, since the correctness of sorting was argued during the lecture. The correctness can be proven using a short inductive argument on n, the size of A: Our base case occurs when $n = 1$, in which case no inversions can occur and thus we indeed return 0.

For our inductive hypothesis, we assume that for all arrays of size k the algorithm computes the number of inversions correctly. We now show that the algorithm computes the number of inversions for $k + 1$ correctly as follows: it splits the array in two halves, each of size at most k, so by our induction hypothesis the number of inversions in them is computed correctly. Thus, if we can show that we correctly compute the number of inversions having one element in L and one in R, the correctness of our algorithm follows for $k + 1$. Fortunately, this is indeed the case, as the number of such inversions is exactly the number of unprocessed element in L (i.e., $|L| - l$) when we put an element of R in the sorted array.

The running time analysis is the same as that for merge sort. We get the following recursion

$$T(n) = \begin{cases} T(n) = 2T(n/2) + O(n) & \text{for } n > 1 \\ O(1) & \text{for } n = 1 \end{cases}$$

which solves to

$$T(n) = c \cdot n + 2 \cdot c \cdot \frac{n}{2} + 4 \cdot c \cdot \frac{n}{4} + ... = O(n \log n).$$

Problem 5. Given a sorted array A containing distinct non-negative integers, find the smallest non-negative integer that isn't stored in the array. For simplicity, you can assume there is such an integer, i.e., $A[n-1] > n-1$

Example: $A = [0, 1, 3, 5, 7]$, result: 2

Solution 5. Our approach is very similar to binary search: we consider the middle element, determine which half contains a missing element and recurse on that. So the main question is how to determine whether a half contains a missing element. This can be done by checking if the integer stored in $A[i] = i$. If $A[i] = i$, we know that there can be no missing values in the first i elements, since all integers are distinct and non-negative. If $A[i] > i$, this means that there is some value missing and hence, we can recurse on the first half. Note that because the integers are distinct, $A[i]$ can't be less than i. Our base case is when we have an array of size 1 left, in which case we can output our smallest missing integer.

```
1: function FIND-MISSING-IN-RANGE(A, low, high)
2:     if low = high then
3:         return low
4:     else
5:         mid ← ⌊(low + high)/2⌋
6:         if A[mid] = mid then
7:             return FIND-MISSING-IN-RANGE(A, mid + 1, high)
8:         else
9:             return FIND-MISSING-IN-RANGE(A, low, mid)
```

```
1: function FIND-MISSING(A)
2:     return FIND-MISSING-IN-RANGE(A, 0, n − 1)
```

Correctness follows directly from the fact that the integers are sorted and distinct, as well as that it's given that there actually is a missing integer. To argue that we indeed recurse on the correct half of the array, we note that we maintain the invariant that we recurse on the half that is missing the smallest element (implied by our check above as to whether $A[i] = i$). Since the size of the array that we recurse on decreases with each iteration, this implies that we eventually reach an array of size 1, which thus must contain the smallest missing element.

The running time analysis is the same as that for binary search. The recursion is

$$T(n) = \begin{cases} T(n) = T(n/2) + O(1) & \text{for } n > 1 \\ O(1) & \text{for } n = 1 \end{cases}$$

which solves to $O(\log n)$.

Problem 6. Design an $O(n)$ time algorithm for the majority problem.

Solution 6. (Sketch) Our base case for a single element stays the same as before. In order to get a linear time algorithm we need a different strategy from the one in the earlier question. Assume for now that n is an even number. First, we pair up

arbitrarily objects and compare the objects in each pair. For each pair, if the objects are different discard both; however, if the objects are the same, keep only one copy. If n happens to be odd, we pick an arbitrary object, test whether it is the majority element; if not, we eliminate it and run the even case routine.

Either way, at most half the elements remain after this filtering step. We keep doing this until only one element remains, at which point we can test in linear time if it is indeed a majority element.

The correctness rests on the fact that if an element is a majority element in the original array, it remains a majority element in the filtered array. (Why?)

Since the filtering can be carried out in linear time and the size of the problem halves after each filtering step, we get the recurrence you get is

$$T(n) = \begin{cases} T(n) = T(n/2) + O(n) & \text{for } n > 2 \\ O(1) & \text{for } n \leq 2 \end{cases}$$

which solves to $T(n) = c \cdot n + c \cdot \frac{n}{2} + c \cdot \frac{n}{4} + ... = O(n)$.

Warm-up

Problem 1. The product of two $n \times n$ matrices X and Y is a third $n \times n$ matrix $Z = XY$, where the (i, j) entry of Z is $Z_{ij} = \sum_{k=1}^{n} X_{ik} Y_{kj}$. Suppose that X and Y are divided into four $n/2 \times n/2$ blocks each:

$$X = \begin{bmatrix} A & B \\ C & D \end{bmatrix} \text{ and } Y = \begin{bmatrix} E & F \\ G & H \end{bmatrix}.$$

Using this block notation we can express the product of X and Y as follows

$$XY = \begin{bmatrix} AE + BG & AF + BH \\ CE + DG & CF + DH \end{bmatrix}.$$

In this way, one multiplication of $n \times n$ matrices can be expressed in terms of 8 multiplications and 4 additions that involve $n/2 \times n/2$ matrices. Let $T(n)$ be the time complexity of multiplying two $n \times n$ matrices using this recursive algorithm.

a) Derive the recurrence for $T(n)$. (Assume adding two $k \times k$ matrices takes $O(k^2)$ time.)

b) Solve the recurrence by unrolling it.

Solution 1.

a) Breaking each matrix into four sub-matrices takes $O(n^2)$ time and so does putting together the results from the recursive call. Therefore the recurrence is

$$T(n) = \begin{cases} 8T(n/2) + O(n^2) & \text{for } n > 1 \\ O(1) & \text{for } n = 1 \end{cases}$$

b) $T(n) = c \cdot n^2 + 8 \cdot c \cdot n^2/4 + 64 \cdot c \cdot n^2/16 + ... = O(n^3)$. Therefore, this is no better that the standard algorithm for computing the product of two n by n matrices.

Problem 2. Similar to the integer multiplication algorithm, there are algebraic identities that allow us to express the product of two $n \times n$ matrices in terms of 7 multiplications and $O(1)$ additions involving $n/2 \times n/2$ matrices. Let $T(n)$ be the time complexity of multiplying two $n \times n$ matrices using this recursive algorithm.

a) Derive the recurrence for $T(n)$. (Assume adding two $k \times k$ matrices takes $O(k^2)$ time.)

b) Solve the recurrence by unrolling it.

Solution 2.

a) Breaking each matrix into four sub-matrices takes $O(n^2)$ time and so does putting together the results from the recursive call. Therefore the recurrence is

$$T(n) = \begin{cases} 7T(n/2) + O(n^2) & \text{for } n > 1 \\ O(1) & \text{for } n = 1 \end{cases}$$

b) $T(n) = O(n^{\log_2 7})$, which is slightly better that the standard algorithm for computing the product of two n by n matrices.
The above can be shown using an argument similar to the one we used for $T(n) = 3T(n/2) + O(n)$ during the lecture. When you think of the recursion tree, we get:
Level 1: 1 instance of size n, which takes $c \cdot n^2$ time
Level 2: 7 instances of size $(n/2)^2 = n^2/4$, which take $(7/4) \cdot c \cdot n^2$ time
Level 3: 49 instances of size $(n/4)^2 = n^2/16$, which take $(49/16) \cdot c \cdot n^2$ time
...
Level k: 7^k instances of size $n^2/4^k$, which take $(7/4)^k \cdot c \cdot n^2$ time
Since we halve the size every time, there are $\log n$ levels. Summing all this up we get

$$\sum_{i=0}^{\log n} (7/4)^k \cdot c \cdot n^2 = c \cdot n^2 \cdot \sum_{i=0}^{\log n} (7/4)^k.$$

When we apply the bound on geometric series to this with $r = 7/4 > 1$, we get

$$c \cdot n^2 \cdot (7/4)^{\log n + 1} / (7/4 - 1) = (4c/3) \cdot n^2 \cdot (7/4)^{\log n + 1}.$$

We focus on $(7/4)^{\log n + 1}$, which is the same $(7/4) \cdot (7/4)^{\log n}$ or $(7/4) \cdot n^{\log 7/4}$. Observe that $\log 7/4 = \log 7 - \log 4 = \log 7 - 2$. Plugging this back into the total running time, we get

$$(4c/3) \cdot n^2 \cdot (7/4) \cdot n^{\log 7 - 2} = (7c/3) \cdot n^{\log 7} = O(n^{\log 7}).$$

Problem solving

Problem 3. Your friend Alex is very excited because they have discovered a novel algorithm for sorting an array of n numbers. The algorithm makes three recursive calls on arrays of size $\frac{2n}{3}$ and spends only $O(1)$ time per call.

```
1: function NEW-SORT(A)
2:     if |A| < 3 then
3:         Sort A directly
4:     else
```

```
5:      NEW-SORT(A[0 : 2n/3])
6:      NEW-SORT(A[n/3 : n])
7:      NEW-SORT(A[0 : 2n/3])
```

Alex thinks their breakthrough sorting algorithm is very fast but has no idea how to analyze its complexity or prove its correctness. Your task is to help Alex:

a) Find the time complexity of NEW-SORT.

b) Prove that the algorithm actually sorts the input array.

Solution 3.

a) Let $T(n)$ be the running time of the algorithm on an array of length n. We observe that the base case is of constant size and thus sorting it takes constant time. For arrays of size at least 3, we recurse three times and each recursion is performed on an array of size $2n/3$. By using the indices during the recursion instead of explicitly copying the array, the divide step takes $O(1)$ time. The algorithm doesn't have a conquer step. Hence, the recursion is

$$T(n) = \begin{cases} 3T(2n/3) + O(1) & \text{for } n \geq 3 \\ O(1) & \text{for } n < 3 \end{cases}$$

which is $O\left(n^{\log_{3/2} 3}\right)$. That's roughly $O(n^{2.71})$. Much worse than the other sorting algorithms we know! Alex is heartbroken...

b) The algorithm, however, is correct. Think of the bottom $\frac{1}{3}$ of the elements in sorted order. After Line 6 is executed we are guaranteed that these element lie in the first $\frac{2}{3}$ of the array and thus Line 7 correctly places them in the first $\frac{1}{3}$ of the array. A similar argument can be made about the top $\frac{1}{3}$ of the elements in sorted order ending in the last $\frac{1}{3}$ of the array. Therefore, the middle $\frac{1}{3}$ of the elements in sorted order are placed in the middle $\frac{1}{3}$ of the array. Within each $\frac{1}{3}$ of the array the elements are sorted, so the array is indeed sorted at the end of the algorithm.

Problem 4. Suppose we are given an array A with n distinct numbers. We say an index i is locally optimal if $A[i] < A[i-1]$ and $A[i] < A[i+1]$ for $0 < i < n-1$, or $A[i] < A[i+1]$ for if $i = 0$, or $A[i] < A[i-1]$ for $i = n-1$.

Design an algorithm for finding a locally optimal index using divide and conquer. Your algorithm should run in $O(\log n)$ time.

Solution 4. First we test whether $i = 0$ or $i = n-1$ are locally optimal entries. Otherwise, we know that $A[0] > A[1]$ and $A[n-2] < A[n-1]$. If $n \leq 4$, it is easy to see that either $i = 1$ or $i = 2$ is locally optimal and we can check that in $O(1)$ time.

Otherwise, pick the middle position in the array (for example $i = \lfloor n/2 \rfloor$) and test whether i is locally optimal. If it is we are done, otherwise $A[i-1] < A[i]$

or $A[i] > A[i+1]$; in the former case we recur on $A[0,\ldots,i]$ and in the latter we recur on $A[i,\ldots,n-1]$. Either way, we reduce the size of the array by half and we maintain that the half that we recurse on contains a locally optimal index (this can be argued similarly to how we argued that binary search recurses on the half that contains the element that we're searching for, if it exists).

The above invariant immediately implies the correctness of our algorithm, since if the half we recurse on contains the index we're searching for and our array gets smaller with each recursive call, we eventually end up in the base case, which is easily checked. Note that our recursion could end earlier, if the middle element happens to be locally optimal, in which case we also found what we were looking for.

Rather than creating a new array each time we recur (which would take $O(n)$ time) we can keep the working subarray implicitly by maintaining a pair of indices (b,e) telling us that we are working with the array $A[b,\ldots,e]$. Thus, each call demands $O(1)$ work plus the work done by recursive calls. This leads to the following recurrence,

$$T(n) = \begin{cases} T(n/2) + O(1) & \text{for } n > 1 \\ O(1) & \text{for } n = 1 \end{cases}$$

which solves to $T(n) = O(\log n)$.

Problem 5. Given two sorted lists of size m and n. Give an $O(\log(m+n))$ time algorithm for finding the kth smallest element in the union of the two lists.

Solution 5. (Sketch.) Let A be the larger array with n elements and B be the smaller array with m elements. Let us assume for simplicity that all numbers are distinct.

Let $\text{rank}(x)$ be the rank of x in the union of A and B. Notice that for each i we have $i \leq \text{rank}(A[i]) \leq i+m$. (Why?) Furthermore, if we compare $A[i]$ to $B[0]$, $B[m/4]$, $B[m/2]$, $B[3m/4]$, and $B[m-1]$ we can better estimate $i+a \leq \text{rank}(A[i]) \leq i+b$ where $b-a \leq m/4$. (Why?) Let us picture this estimate as an interval $I_i = [i+a, i+b]$.

Now observe that if $k < I_i$ (if k is smaller than the left endpoint of I_i) then $k < \text{rank}(A[i])$ so we can ignore every item larger than $A[i]$ from the large array and search for the kth element in the smaller instance. Similarly, if $k > I_i$ then $k > \text{rank}(A[i])$ so we can ignore everything smaller than $A[i]$ from the large array and search for the $(k-i)$th element in the smaller instance.

The only issue is what to do if $k \in I_i$. To get around this, we find the intervals of two indices. Notice that $\text{rank}(A[3n/4]) - \text{rank}(A[n/4]) \geq n/2$ and that $|I_{n/4}|, |I_{3n/4}| \leq m/4 \leq n/4$, therefore, $I_{n/4}$ and $I_{3n/4}$ must be disjoint and therefore either $k > |I_{n/4}|$ or $k < |I_{3n/4}|$. That means that we can always remove the elements that are either smaller than $A[n/4]$ or larger than $A[3n/4]$.

For the time complexity we note that we do not really need to create a new array each time we recur, we can simply keep two pairs of indices, one for A and one for B, that tell us the parts of the array where we are still searching on. Then, in each iteration we get rid of $n/4 \geq (m+n)/8$ elements in $O(1)$ time. Therefore, the combined length of the intervals is at most $\frac{7}{8}$ their previous combined length. Thus,

if we let N measure the combined length of the intervals, we get the recurrence

$$T(n) = \begin{cases} T(7N/8) + O(1) & \text{for } N > 1 \\ O(1) & \text{for } N = 1 \end{cases}$$

which solves to $T(N) = O(\log N)$. Therefore, the algorithm runs in $O(\log(n+m))$ time.

Problem 6. Solve the following recurrences using the Master Theorem or unrolling (the solutions will use the Master Theorem to allow you to practice that). All are $O(1)$ for $n = 1$.

a) $T(n) = 4T(n/2) + O(n^2)$

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

c) $T(n) = 16T(n/4) + O(n)$

d) $T(n) = 2T(n/2) + O(n \log n)$

e) $T(n) = \sqrt{2}T(n/2) + O(\log n)$

f) $T(n) = 3T(n/2) + O(n)$

g) $T(n) = 3T(n/3) + O(\sqrt{n})$

Solution 6. Recall that the Master Theorem applies to recurrences of the following form: $T(n) = aT(n/b) + f(n)$ where $a \geq 1$ and $b > 1$ are constants and $f(n)$ is an asymptotically positive function. There are three cases:

1. If $f(n) = O(n^{\log_b a - \varepsilon})$ for some constant $\varepsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$.
2. If $f(n) = \Theta(n^{\log_b a} \log^k n)$ with $k \geq 0$, then $T(n) = \Theta(n^{\log_b a} \log^{k+1} n)$.
3. If $f(n) = \Omega(n^{\log_b a + \varepsilon})$ with $\varepsilon > 0$, and $f(n)$ satisfies the regularity condition, then $T(n) = \Theta(f(n))$. Regularity condition: $af(n/b) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n (this condition will be satisfied for all recurrences you'll encounter during this unit and we'll ignore it).

Also remember that if $f(n)$ is given in big-O notation, we can only use the above to conclude an upper bound on $T(n)$ in big-O notation and not the Θ used in the theorem. This is because Θ implies that we have both an upper and lower bound on the running time and big-O only gives us an upper bound on $f(n)$.

a) $T(n) = 4T(n/2) + O(n^2) \to T(n) = O(n^2 \log n)$ (Case 2 with $a = 4$, $b = 2$ and $f(n) = n^2$)

b) $T(n) = T(n/2) + O(2^n) \to O(2^n)$ (Case 3 with $a = 1$, $b = 2$ and $f(n) = 2^n$)

c) $T(n) = 16T(n/4) + O(n) \to T(n) = O(n^2)$ (Case 1 with $a = 16$, $b = 4$ and $f(n) = n$)

d) $T(n) = 2T(n/2) + O(n \log n) \to T(n) = O(n \log^2 n)$ (Case 2 with $a = 2$, $b = 2$ and $f(n) = n \log n$)

e) $T(n) = \sqrt{2}T(n/2) + O(\log n) \rightarrow T(n) = O(\sqrt{n})$ (Case 1 with $a = \sqrt{2}$, $b = 2$ and $f(n) = \log n$)

f) $T(n) = 3T(n/2) + O(n) \rightarrow T(n) = O(n^{\log 3})$ (Case 1 with $a = 3$, $b = 2$ and $f(n) = n$)

g) $T(n) = 3T(n/3) + O(\sqrt{n}) \rightarrow T(n) = O(n)$ (Case 1 with $a = 3$, $b = 3$ and $f(n) = \sqrt{n}$)

Warm-up

Problem 1. If you repeatedly perform an experiment whose outcome has a Geometric Distribution with probability of success p, what is the expected number of times you need to repeat the experiment before you get a success?

Solution 1. The expected number of repetitions required before you get a success is $1/p$.

Problem solving

Problem 2. Suppose you only have access to a biased coin that lands heads with probability p and tails with probability $(1-p)$. Show how to design a fair coin without even knowing what p is. How many times does your algorithm flip the biased coin in expectation?

Solution 2. Flip the coin twice. If the outcomes are HT, output Heads. If the outcomes are TH, output Tails. If the outcomes are HH or TT, try again.

The probability of not having to try again is $2p(1-p)$. Therefore, we expect that the number of tries is $1/2p(1-p)$ because that is the expected value of the Geometric Distribution with probability $2p(1-p)$. Each try involves two coin flips of the biased coins, so the expected number of coin flips is therefore $1/p(1-p)$.

Problem 3. Suppose you only have access to a fair coin that lands heads with probability $1/2$ and tails with probability $1/2$. Show how to design an algorithm for uniformly sampling an integer from $\{1,\ldots,n\}$. How many times does your algorithm flip the coin in expectation?

Solution 3. Let $k = \lfloor \log_2 n \rfloor + 1$. Flip the fair coin k many times. Interpret the sequence of heads and tails as a binary number, i.e.,

$$\text{HHTHTTH} \to 1101001.$$

It is easy to see that the number is distributed uniformly from $\{0,\ldots,2^k-1\}$. Note that $\log_2 n \leq k \leq \log_2 n + 1$, so

$$n - 1 \leq 2^k - 1 \leq 2n - 1.$$

If the number constructed happens to be in the range $\{1,\ldots,n\}$, then we output it. Otherwise, we try again.

The probability of not having to try again is at least $1/2$. Therefore, the expected number of tries is 2 because the expected value of a Geometric Distribution with probability $1/2$ is 2. Each try involves flipping $\log_2 n + 1$ fair coins, so the expected number of coin flips is $2(\log_2 n + 1)$.

Problem 4. Suppose you only have access to a random generator for sampling real numbers from the interval $[0, 1]$. Show how to design an algorithm for uniformly sampling points (x, y) from the square $[-1, 1]^2$ (i.e., $-1 \leq x \leq 1$ and $-1 \leq y \leq 1$). How many samples does your algorithm need in expectation?

Solution 4. We sample two numbers x' and y' and return $x = 2x' - 1$ and $y = 2y' - 1$. Since the random generator generates a number from the real interval $[0, 1]$, all x' and y' are equally likely, which implies that all x and y are equally likely. Hence, this generates a point in the square uniformly at random.

Problem 5. Suppose you only have access to a random generator for sampling real numbers from the interval $[0, 1]$. Show how to design an algorithm for uniformly sampling points (x, y) from the unit radius disk centered at the origin (i.e., $x^2 + y^2 \leq 1$). How many samples does your algorithm need in expectation?

Solution 5. Sample a point from the square, if the point lies inside the disk output that; otherwise, reject it and try again.

The probability of not having to try again is $\pi/4$ (the area of the disk is π and that of the square is 4), so the expected number of tries is $4/\pi$. Each try takes two samples, so this approach requires $8/\pi$ samples overall.

Problem 6. Suppose we want to add a new operation to the existing skip list implementation. This operation, RangeSearch($k1$, $k2$), returns all the items with keys in the range $[k1, k2]$. Design this operation and show that it runs in expected time $O(\log n + s)$, where n is the number of elements in the skip list and s is the number of items returned.

Solution 6. Since every element is stored at the lowest layer of the skip list and we are allowed to report the s elements in $O(s)$ time, a simple approach to this problem is to search for $k1$ and after that traverse the bottom layer of the skip list until we reach a node with key greater than $k2$. Note that our search ends at a node with key at most $k1$, so we need to check if it should be reported before proceeding to scan its successors.

This algorithm is correct, since it starts from a node that's at most $k1$ and then scans the bottom level of the skip list until it reaches a node with key larger than $k2$, reporting all nodes in between.

This algorithm's running time is determined by the length of the search path to $k1$ and the number of elements we traverse until we reach a node with key greater than $k2$. Recall that the expected height of a skip list if $O(\log n)$, so that's the time we spend searching for $k1$. While reporting all nodes in the range $[k1, k2]$, we note that if there are s such nodes to report, we check at most $s + 2$ nodes: the s nodes that need to be reported and at most one node before the range (where our search for $k1$ ends) and one node after the range (where our scan of the lowest level of the skip lists ends). Hence, we spend $O(s)$ time reporting these nodes. In total this leads to an $O(\log n + s)$ expected time algorithm.