
CS 61B

Spring 2024

Final
Tuesday, May 7, 2024

Your Name: _____ [Solutions](#)

Your SID: _____ Location: _____

SID of person to your left: _____ Right: _____

This exam is worth **100 points**, and consists of 8 questions. Questions are ordered by topic, not by difficulty.

Question:	1	2	3	4	5	6	7	8	Total
Points:	9	22	4	7	5	16	22	15	100

- ☐ indicates only one circle should be filled in. ☐ indicates more than one box may be filled in. **Please fill in the shape completely.** If you change your response, **erase as completely as possible.**
- Anything you write that you ~~cross out~~ will not be graded.
- If you write multiple answers, your answer is ambiguous, or the bubble/checkbox is not entirely filled in, we will grade according to the worst interpretation.
- Unless otherwise specified, all data structures and algorithms behave according to their implementation in lecture, with no additional optimizations.
- If an implementation detail (e.g. tiebreaking scheme, linked list topology) is relevant, it will be explicitly noted in the question.

Coding questions:

- You may write at most one statement per blank and you may not use more blanks than provided.
- Your answer will be reformatted according to the 61B style guidelines. For example, any if-statement requires at least three lines for the purposes of determining line count.
- You may not use ternary operators, lambdas, streams, or multiple assignment.
- Unless otherwise specified, you may assume that everything on the reference card has been imported.

Write the statement below in the same handwriting you will use on the rest of the exam. Failure to do so may result in a voided exam.

I have neither given nor received help on this exam (or quiz), and have rejected any attempt to cheat. If these answers are not my own work, I may be deducted up to 9,876,543,210 points.

Signature: _____

1 Vroom (9 points)

For each of the following lines, write the result of each statement: Write “CE” if a compiler error is raised on that line, “RE” if a runtime error occurs on that line, “OK” if the line runs properly but doesn’t print anything, and the printed result if the line runs properly and prints something. If a line errors, assume that the program continues to run as if that line did not exist. Each blank is worth 1 point.

```

1      public class TwoWheel {
2          public int topSpeed = 200;
3          private boolean counterSteer = true;
4          public void rideWith(TwoWheel t) { System.out.println("I'm TwoWheel"); }
5          public void printSteer() { System.out.println(counterSteer); }
6      }
7      public class Motorcycle extends TwoWheel {
8          public Motorcycle() { topSpeed = 400; }
9          public void rideWith(TwoWheel t) { System.out.println("I'm moto"); }
10     }
11     public class Bicycle extends TwoWheel {
12         public Bicycle() { topSpeed = 100; }
13         public void rideWith(TwoWheel t) { System.out.println("I'm bike"); }
14     }

```

TwoWheel scooter = new TwoWheel(); a: OK

Despite no explicitly defined constructor in the TwoWheel class, this does not CE. This is because Java generates and uses a default constructor with no parameters if no constructor is defined.

Motorcycle kawi = new Motorcycle(); b: OK

Bicycle trek = new Bicycle(); c: OK

Motorcycle.rideWith(kawi); d: CE

You cannot call a nonstatic method from a Class.

kawi.rideWith(trek); e: I'm moto

((TwoWheel) trek).rideWith(scooter); f: I'm bike

During compile time, we look for rideWith(TwoWheel) in the TwoWheel class, which we find. During runtime, we use trek’s dynamic type (Bicycle) to run rideWith(TwoWheel).

The following subparts are independent; the change in each part does not carry over to other parts.

- (g) (1 point) If we change the signature on line 5 to `public static void printSteer()`, will the code still compile?

☐ Yes

☒ No

Solution: We would be trying to access a non-static variable (`counterSteer`) inside of a static method, which is not allowed and will cause a compiler error.

- (h) (1 point) If we change line 3 to `private static boolean counterSteer = true`, will the code still compile?

☒ Yes

☐ No

Solution: In contrast with part (g), we are allowed to access a static variable from a non-static method.

- (i) (1 point) If we change line 2 to `private int topSpeed = 200`, will the code still compile?

☐ Yes

☒ No

Solution: Private variables are only accessible inside their own classes. In `Motorcycle`'s and `Bicycle`'s constructors, we try to access the variable `topSpeed`, which, if made private, won't be defined. The fact that `Motorcycle` and `Bicycle` are subclasses of `TwoWheel` makes no difference; we would have to change `topSpeed`'s access modifier to `protected` for it to be accessible in its subclasses.

A common point of confusion is whether or not the classes in this problem are defined in the same file, or in different files – they are defined in different files, because of Java's design of one top-level class per file (e.g. a file called `Moped.java` must have only one top-level class called `Moped`). However, this line of questioning is indicative of a larger conceptual misunderstanding of what the `private` keyword does. The `private` keyword makes a variable or method only accessible within the class that it is defined, NOT within the file that it is defined.

2 Polynomials (22 points)

An *Integer Polynomial* is an expression of the form

$$a_n x^n + \cdots + a_2 x^2 + a_1 x + a_0$$

where all a_i , $0 \leq i \leq n$, are integers. A *nonzero term* of a polynomial is a single $a_i x^i$ with $a_i \neq 0$. The *degree* of a polynomial is the largest exponent n associated with a nonzero term, or 0 for the polynomial 0. Consider the `Polynomial` class below, which represents an integer polynomial as a linked list of terms:

```
public class Polynomial {
    private class Term {
        public int coef;
        public int exp;
        public Term next;

        public Term(int coef, int exp) {
            this.coef = coef;
            this.exp = exp;
        }
    }

    private Term terms;

    public Polynomial() {
        terms = new Term(0, -1);
    }

    public double eval(double input) {
        // Implemented below
    }

    public void addTerm(int coef, int exp) {
        // Implemented below
    }
}
```

Note the following:

- The special term $(0, -1)$ is used as a sentinel value, which is placed at the **end** of the linked list.
- The linked list contains only the nonzero terms of the polynomial, ordered from greatest exponent to least exponent. Thus, every polynomial has a unique representation.
- The linked-list topology is not circular; the sentinel term always has a **next** of `null`.

For example, the polynomial $-3x^{500} + 5x^3 + 4x + 7$ has four nonzero terms, and has degree 500; it is represented by the linked list $(-3, 500) \rightarrow (5, 3) \rightarrow (4, 1) \rightarrow (7, 0) \rightarrow (0, -1)$.

The polynomial 0 has no nonzero terms and has degree 0; it is represented by the linked list $(0, -1)$.

The Polynomial class, reprinted for your convenience:

```
public class Polynomial {
    private class Term {
        public int coef;
        public int exp;
        public Term next;

        public Term(int coef, int exp) { ... }
    }

    private Term terms;

    public Polynomial() {
        terms = new Term(0, -1);
    }
}
```

- (a) (7 points) Implement the `eval` method, which receives a value `input` and returns the result of evaluating the polynomial when $x = \text{input}$.

For example, the polynomial $5x^3 + 4x + 7$ evaluated at the input 2 would be $(5 \cdot 2^3) + (4 \cdot 2) + (7) = 55$. If instead, this polynomial is evaluated at 0.5, the result would be $(5 \cdot 0.5^3) + (4 \cdot 0.5) + (7) = 9.625$.

```
public double eval(double input) {
    double result = 0;

    Term currTerm = terms;

    while (currTerm.exp != -1) { // See alternate solutions below

        result += currTerm.coef * Math.pow(input, currTerm.exp);

        currTerm = currTerm.next;
    }

    return result;
}
```

Solution:

Common mistakes: Many students attempted to use `List` or `Deque` interface methods such as `add`, `size`, or `getFirst`. This is incorrect, as the `Polynomial` class does not implement either of these interfaces. Furthermore, use of these methods demonstrates insufficient understanding that the `Polynomial` class itself forms a linked-list structure with the inner class `Term`, accessible through the instance variable `terms`.

Another common error is trying to use a `prev` attribute or similar. The `Polynomial` class specifically forms a **non-circular singly-linked list**, so any attempt at using `prev` attributes, or trying to “wrap around” to the start of the list, is incorrect.

Alternate solutions: There are many alternate solutions for blank 2 (the while loop condition). The following are **valid** alternate solutions:

- `currTerm.exp != -1`: The sentinel term will always exist and has an exponent of -1.
- `currTerm.exp > -1`: All nonzero terms will have a non-negative exponent, since $0 \leq n$.
- `currTerm.coef != 0`: We are guaranteed that all nonzero terms will have a nonzero coefficient. Additionally, the sentinel term will always exist and has a coefficient of 0.
- `currTerm.next != null`: We’re guaranteed that the only term with no `next` term is the sentinel term.
- Any combination of the above answers are also valid.

The following (non-exhaustive list of) answers are **invalid**:

- `currTerm != null`: This fails when `input == 0`, as the sentinel term will try to evaluate `Math.pow(0, -1)` which is equal to $\frac{1}{0}$. This will cause the return value of `eval` to be `NaN` (Not a Number).
- `currTerm.exp >= -1`: This will result in a `NullPointerException`. Though we do not enter the while loop after we iterate through the sentinel term, the while loop condition `currTerm.exp != -1` fails because `currTerm` is `null` by this point.
- `currTerm.next.exp != -1`: For polynomials with 2 or more terms, this does not account for the last nonzero term, so will give an incorrect answer. For polynomials with less than 2 terms, this will result in a `NullPointerException`, since the `currTerm.next` in the while loop condition is `null`.
- `currTerm.next.coef != 0`: Same reasoning as the above item.
- `currTerm.next.next != null`: Same reasoning as the above item.

- (b) (2 points) What is the asymptotic runtime of `eval`, in terms of n , the degree of the polynomial? Provide the most informative bound possible, in either Θ , Ω , or O notation.

$O(n)$

Solution: The maximum number of terms that we can have in a degree n polynomial is $n + 1$, so in the worst case `eval` will take $\Theta(n)$ time. However, the best case is if we have some polynomial that looks like $a_n x^n$, which will be represented by the linked list $(a_n, n) \rightarrow (0, -1)$. In this case, regardless of how large n is, we only have to traverse 2 items, meaning that the best case runtime of `eval` is $\Theta(1)$. Thus, the most informative bound possible is $O(n)$.

Note that n is the degree of the polynomial, **not** the number of terms in the polynomial. For example, in the expression $3x^{500} + 2x^{250} + 5$, n is 500, not 3.

- (c) (11 points) Implement the `addTerm` method, which receives a coefficient and exponent representing a nonzero term, and updates the polynomial to include that term.

For example, the polynomial $5x^3 + 4x + 7$, after adding $2x^2$, would become the polynomial $5x^3 + 2x^2 + 4x + 7$, with the underlying linked list $(5, 3) \rightarrow (2, 2) \rightarrow (4, 1) \rightarrow (7, 0) \rightarrow (0, -1)$.

You may assume that `coef != 0`, `exp >= 0`, and that the polynomial does not already contain a nonzero term with the same exponent.

```
public void addTerm(int coef, int exp) {

    Term newTerm = new Term(coef, exp);

    if (newTerm.exp > terms.exp) { // Alternate answer: exp > terms.exp

        newTerm.next = terms;

        terms = newTerm;
        return;
    }
    Term curr = terms;

    while (curr.next.exp > newTerm.exp) { // Alternate answer: curr.next.exp > exp

        curr = curr.next;
    }

    newTerm.next = curr.next;

    curr.next = newTerm;
}
```

Solution: In the if statement, we handle the case that `newTerm` has an exponent greater than the currently largest term's exponent, and thus `newTerm` should go at the front of the linked list. We assign `newTerm`'s next pointer to point to the currently largest term, and then we reassign `terms` to point to `newTerm`. If we entered this if statement, we can return out of the method early.

In the while loop, we traverse the linked list to find the correct spot to insert our `newTerm`. However, we must be careful with how we do it. Specifically, since our linked list is singly-linked, we must make sure to "look ahead" one element in our while loop condition, hence why we are checking `curr.next.exp` instead of `curr.exp`. If we were to incorrectly use `curr.exp`, we would not be able to assign the correct node's next pointer to point to `newTerm`. We do not have to worry about any `NullPointerExceptions`

due the presence of the sentinel node and the constraints placed upon the problem that `exp >= 0`. It is also valid to assume that `terms` has at least one nonzero node at this point, since if `terms` contained only the sentinel node, then we would have entered the if statement, and exited from the method early.

Finally on the last 2 lines, we assign `newTerm`'s next pointer and reassign `curr`'s next pointer to point to `newTerm`.

- (d) (2 points) What is the asymptotic runtime of `addTerm`, in terms of n , the degree of the **original** polynomial, and k , the exponent of the new term? Provide the most informative bound possible, in either Θ , Ω , or O notation.

$O(n)$

Solution: As in part (b), note that n is the degree of the polynomial, **not** the number of terms in the polynomial.

In the worst case, we could have a polynomial of degree n with n terms like $x^n + x^{n-1} + \dots + x^2 + x$, with the constant term missing, and we could try to add the constant term at the end. This would require traversing the entire linked list to reach the last term, which takes $\Theta(n)$ time.

In the best case, we could have a single-term polynomial of degree n like x^n , and we could again try to add a constant term at the end. No matter how large n gets, the runtime in this case is the same (we only have to scan one item in the linked list), namely $\Theta(1)$ time.

Since the worst case is $\Theta(n)$ and the best case is $\Theta(1)$, we can say that the runtime is $O(n)$ in all cases.

Note that in the intended solution, k is not used in our answer, because even as k grows large, the runtime bound does not change. For example, if we had the n -term polynomial from earlier, $x^n + x^{n-1} + \dots + x^2 + x$, as k grows large (i.e. larger than n), we actually end up putting the new large term at the front of the linked list (taking constant time), which is still within the $O(n)$ bound we provided.

Alternate answers:

We gave credit to $O(n - k)$, $O(|n - k|)$, and $O(\max(1, n - k))$ as alternate answers involving k .

$O(\max(1, n - k))$ correctly captures the behavior of the function as both n and k grow large. In particular, these bounds capture the fact that if n and k are both large, and $k > n$, then the runtime is constant (add to the front of the linked list). Also, if n and k are both large, but $k < n$, then the runtime is indeed roughly $O(n - k)$, since that's the most number of terms we would need to traverse before inserting the new item. For example, if we had $n = 500$ and $k = 400$, and our expression was $x^{500} + x^{499} + x^{498} + \dots + 1$, then it would take $n - k = 500 - 400 = 100$ operations to scan to the x^{100} spot and add that new term.

$O(|n - k|)$ was given credit, although it is not as informative as $O(\max(1, N - K))$. We gave this answer credit because it is neither more nor less informative than $O(n)$. In particular, when $k > n$, we get a bound of $O(k - n)$. This bound is not as informative, because we would be in the case where we insert at the front of the list, and the correct tight bound is $O(1)$. However, when $n > k$, the bound of $O(n - k)$ does end up being more informative than the bound of $O(n)$.

$O(n - k)$ is not correct under a strict definition of O -bounds. For example, if $k = 600$ and $n = 500$ (which would trigger a constant-time insertion at the front), this bound gives us a rough runtime of -100 , which is not correct. However, we gave this answer credit because some online sources indicated

that O-bounds are only defined in the context of positive-valued functions, and this is not something we really discussed in 61B.

3 Disjoint Sets (4 points)

- (a) (1 point) True/False: The array below is a valid representation of a Weighted Quick Union.

1	0	-1	-1	-1
---	---	----	----	----

☐ True

☒ False

Solution: Node 0 claims that it is a direct child of node 1, but node 1 claims that it is a direct child of node 0. This is a direct contradiction, thus the representation must be invalid.

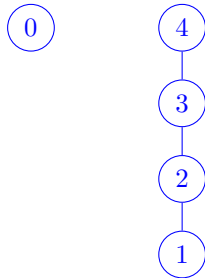
- (b) (1 point) True/False: The array below is a valid representation of a Weighted Quick Union.

-1	2	3	4	-4
----	---	---	---	----

☐ True

☒ False

Solution: This is the representation of the Weighted Quick Union:



While the underlying array is logically consistent, this disjoint set structure could not have possibly been generated using a Weighted Quick Union. This is because Weighted Quick Unions with N nodes are guaranteed to be height $\lfloor \log_2 N \rfloor$ or less (where a tree with a single node is height 0). With 5 nodes in the Weighted Quick Union, we expect the height to be $\lfloor \log_2 5 \rfloor = 2$ at most. Since our Weighted Quick Union is of height 3, this could not possibly be a valid Weighted Quick Union.

- (c) (1 point) You are given a mystery Disjoint Set implementation, and want to figure out if it is a **Quick Union**, or a **Weighted Quick Union**.

In order to determine this, you create a new instance of your Disjoint Set with 100 elements, and observe the values in the underlying array as you apply `union` operations. What is the minimum number of `union` operations needed to distinguish between these two disjoint set implementations?

☒ 1

☐ 3

☒ 2

☐ 4

Solution: The intended answer was 2, but after the exam, students found that 1 is a valid alternate answer. Points were awarded if you selected 1 or 2. Both answers are explained below.

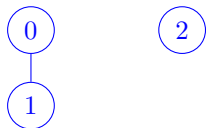
2 operation solution:

The exact numbers (of the 100 elements) that we connect don't actually affect the answer, so we'll arbitrarily use elements 0, 1, 2, 3, 4, ... The analysis is the same if you chose to connect up some other subset of the 100 elements.

The new instance of your Disjoint Set object starts with all 100 elements disconnected (in their own sets).

With a single `union` operation, all we can do is connect up two elements into a set with two elements. In both the Quick Union (QU) and Weighted Quick Union (WQU) implementations, the result is the same.

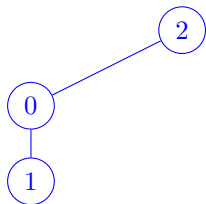
This is what the Disjoint Sets object (either QU or WQU) looks like after `union(0, 1)`:



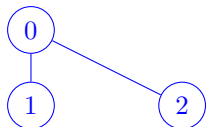
Recall that in the QU implementation, we always hang the root of the first tree under the root of the second tree (where first and second refer to the order of arguments passed into `union`). By contrast, in the WQU implementation, we always hang the root of the smaller tree under the root of the larger tree.

The key realization to solve this problem is to note that after one call to `union`, we have two trees of unequal size. This means that we can attempt to hang the larger tree underneath the smaller tree, and see if it succeeds, to distinguish between the QU and WQU implementations.

In particular, we can call `union(0, 2)` in an attempt to hang the larger tree (rooted at 0) underneath the smaller tree (rooted at 2). In a QU implementation, this will succeed, since QU takes the first argument (tree rooted at 0) and hangs that tree underneath the second argument (tree rooted at 2):



By contrast, in a WQU implementation, the smaller tree (rooted at 2) will be attached below the larger tree (rooted at 0):



At this point, by observing the underlying array (which represents these tree structures), we can determine whether the Disjoint Sets was using a QU or WQU operation. Therefore, the answer is 2 `union` operations, and we are done.

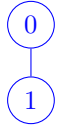
1 operation solution:

This alternate solution exploits the different ways in which the underlying array represents the QU and WQU implementations.

In particular, in the QU implementation, when an item is the root, the corresponding index in the underlying array contains the integer `-1` (to indicate that this item has no parent).

By contrast, in the WQU implementation, when an item is the root, the corresponding index in the underlying array contains the negative of the number of items in that subtree. Unlike in QU, we need to store the number of items in the subtree, since the `union` operation in WQU requires us to check which subtree is larger than the other.

Therefore, to distinguish QU from WQU, all we need to do is run `union(0, 1)`:



In a QU implementation, the underlying array would be $[-1, 0]$. Index 0 contains integer -1 , since item 0 is the root. Index 1 contains integer 0, since item 1's parent is item 0.

By contrast, in a WQU implementation, the underlying array would be $[-2, 0]$. Index 0 contains integer -2 , since item 0 is the root, and the subtree rooted at 0 contains 2 items. Index 1 contains integer 0, since item 1's parent is 0.

Since the two underlying arrays are different at this point, we can distinguish between QU and WQU.

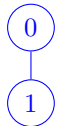
- (d) (1 point) You are given a mystery Disjoint Set implementation, and want to figure out if it is a **Weighted Quick Union**, or a **Weighted Quick Union with Path Compression**.

In order to determine this, you create a new instance of your Disjoint Set with 100 elements, and observe the values in the underlying array as you apply `union` operations. What is the minimum number of `union` operations needed to distinguish between these two disjoint set implementations?

- ☐ 1
 ☐ 3
☐ 2
 ☒ 4

Solution: Similar to the last subpart, the exact numbers (of the 100 elements) that we connect don't affect the answer, so we'll use elements $0, 1, 2, 3, 4, \dots$, without loss of generality. (Note: "Without loss of generality," or WLOG, is often used in math proofs to say that our choice was arbitrary for the sake of writing a clean solution, but other choices would have also worked just fine.)

As in the previous subpart, a single `union` operation doesn't help us distinguish anything, since the resulting Disjoint Sets object looks the same whether it's using Weighted Quick Union (WQU) or Weighted Quick Union with Path Compression (WQUPC). After `union(0, 1)`, the Disjoint Sets object looks like this:



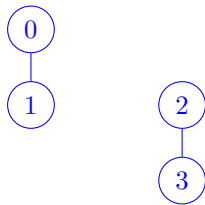
At this point, we might try to use the same trick as the previous question (attempt to hang the larger tree under the smaller tree), but that won't help us distinguish between WQU and WQUPC, since both implementations always hang the smaller tree under the larger tree.

Recall that WQUPC is different from WQU because as we climb the tree to find the root, we tie all intermediate items seen along the way directly to the root.

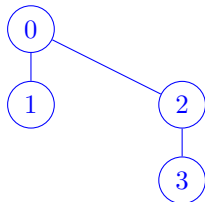
The Disjoint Set object so far has a maximum height of 1 so far (the tree rooted at 0), so we'll never be able to see WQUPC tying an intermediate item directly to the root. All subordinate items (just 1 so far) are already directly tied to the root.

The key realization to solve this problem is to note that in order to see intermediate items being tied directly to the root by WQUPC (or not tied directly to the root by WQU), we have to create a tree of height at least 2.

Recall from lecture that we can create a height-2 tree by creating two trees of height 1, and then unioning them together. In particular, we'll call `union(2, 3)` to create the second height-1 tree:

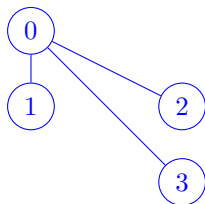


Then, we'll call `union(0, 2)` to combine the two height-1 trees into a height-2 tree:

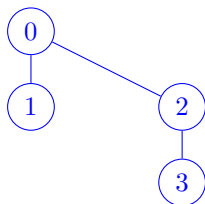


Now that we have a height-2 tree, we can test if intermediate items get connected to the root, which would allow us to check if this is a WQUPC or a WQU. For example, we can run `union(3, 0)`.

If the implementation was a WQUPC, we would see 3 tied to the root, since we have to climb $3 \rightarrow 2 \rightarrow 0$ to find the root associated with 3:



However, if this implementation was a WQU, we would not change anything about the underlying trees, since the root of 3 (i.e. 0) and the root of 0 (i.e. 0) are the same:



In total, we had to call `union(0, 1)`, `union(2, 3)`, and `union(0, 2)` to create the height-2 tree. Then, we had to call `union(3, 0)` to check if 3 gets tied to the root. In total, we made 4 union calls to distinguish between WQU and WQUPC.

Note: This solution did not prove that distinguishing cannot be done in 3 or fewer calls (and you didn't have to prove this on the exam). The rough intuition for why 4 is the minimum is: you need at least 3 calls to create the height-2 tree in the first place (without a height-2 tree, there is no way to distinguish). Then, you need to call `union` on the height-3 tree at least once to check if intermediate items are tied to the root.

Note: Many students thought the answer was 3, because the call to `union(0, 2)`, which combines the two height-1 trees into a height-2 tree, will also tie 3 to the root of the combined height-2 tree (0).

However, this is not how the implementation of WQUPC works. In WQUPC, we only tie items to the

root as a side effect of calling the `find` method, which climbs up the tree to find the root associated with an item.

When calling `union(0, 2)`, we call `find(0)` to see that 0's root is 0, and we call `find(2)` to see that 2's root is 2. Neither operation ties any additional items to the root. Then, after both calls to `find` are done, we hang one tree under the other tree (which does not involve any further calls to `find` and will not tie any more items to the root).

Even if you called something like `union(1, 3)` to combine the trees instead, `find(1)` (which returns 0) and `find(3)` (which returns 2) will not cause 3 to be tied to 0, because there is no point at which we climb from 3 and reach 0.

4 Heaps (7 points)

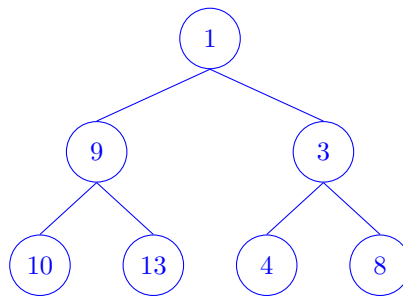
For the following question, all heap implementations leave the 0th index empty. The empty index is denoted with a $-$.

- (a) (1 point) Consider the following min-heap in array format: $[-, 1, 9, 3, 10, 13, 4, 8]$.

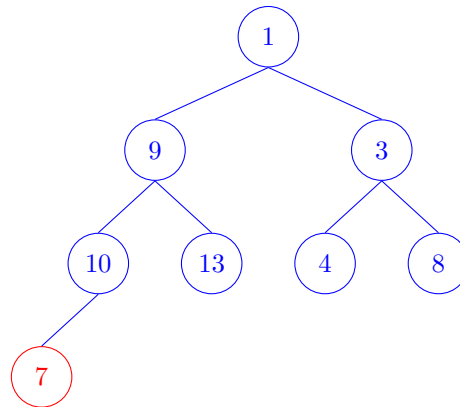
What is the resulting array representation after inserting 7?

- ☐ $[-, 1, 9, 3, 7, 13, 4, 8, 10]$
☐ $[-, 1, 3, 7, 9, 13, 4, 8, 10]$
☐ $[-, 1, 3, 7, 13, 9, 4, 8, 10]$
☒ $[-, 1, 7, 3, 9, 13, 4, 8, 10]$

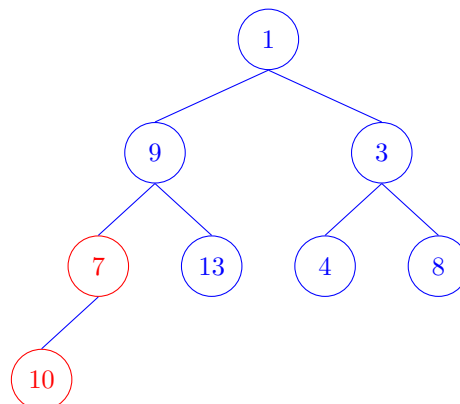
Solution: First, we'll draw the heap diagram corresponding to this array representation:



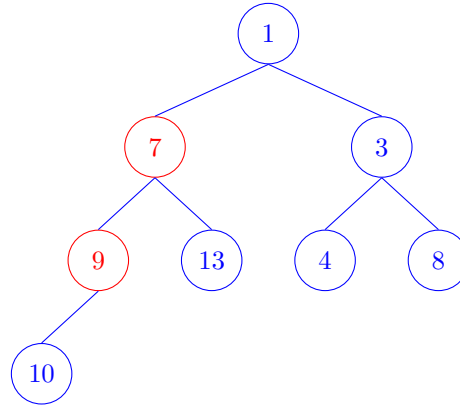
To add 7, we first add it to the next available spot in the complete heap:



Then, we'll make 7 climb as far up as it can go. 7 is less than 10, so these two items should swap:



7 is also less than 9, so these two items should swap:



7 is not less than 1, so these two items should not swap, and 7 has moved into its correct place. (You can verify that the min-heap property holds on this result.)

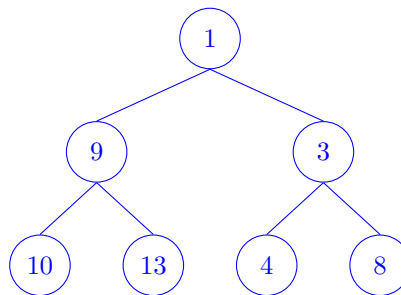
Finally, we convert this heap diagram back into its array representation by reading the node values in level order (top-to-bottom, left-to-right).

- (b) (1 point) Consider the following min-heap in array format: $[-, 1, 9, 3, 10, 13, 4, 8]$.

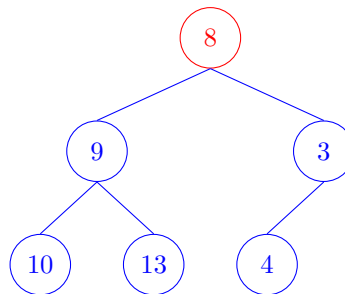
What is the resulting array representation after 2 remove operations?

- ☒ $[-, 4, 9, 8, 10, 13]$
- ☐ $[-, 4, 8, 9, 10, 13]$
- ☐ $[-, 4, 8, 9, 13, 10]$
- ☐ $[-, 4, 9, 8, 13, 10]$

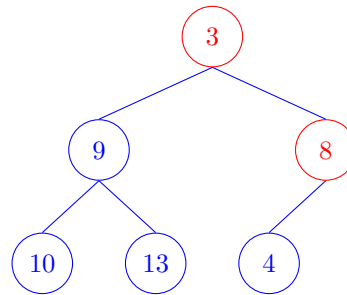
Solution: As in the previous subpart, we first draw the heap diagram:



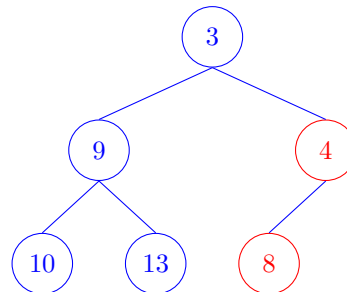
To remove from the heap, we first delete the root (1), and swap the last item (i.e. bottom-right) into the root:



Now, we sink 8 as far down as it can go. As we sink, we should always swap with the smaller of the two children (if you tried swapping with the larger of the two children, the resulting min-heap would not be correct). 8 is not less than both 9 and 3, so we should swap 8 and 3:

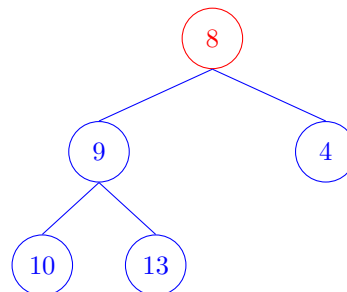


8 is not less than 4, so we should swap these two items:

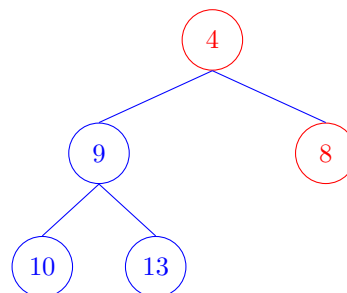


At this point, the 8 cannot sink any further, so the first remove operation is done.

To remove again from the heap, we first delete the root (3), and swap the last item (i.e. bottom-right) into the root:



Now, we need to sink the new root (8 again) as far down as needed. 8 is not less than 9 and 4, so we need to swap 8 and 4:



At this point, the 8 cannot sink any further, so the second remove operation is done.

Converting this resulting heap diagram back into its array representation gives us the answer.

- (c) (2.5 points) Construct a max-heap with the values 1, 2, 3, 4, 5, 6, 7, such that the underlying array has 21 inversions.

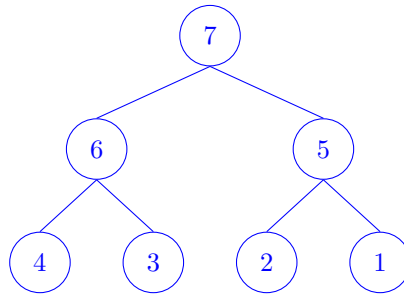
(Note: This is the maximum possible number of inversions in a valid max-heap with 7 elements.)

—	7	6	5	4	3	2	1
---	---	---	---	---	---	---	---

Solution: An inversion is defined as a pairing of indices i and j where $i < j$ and $\text{arr}[i] > \text{arr}[j]$.

There are 21 inversions in this array: 7-6, 7-5, 7-4, 7-3, 7-2, 7-1, 6-5, 6-4, 6-3, 6-2, 6-1, 5-4, 5-3, 5-2, 5-1, 4-3, 4-2, 4-1, 3-2, 3-1, 2-1. Drawing this array will also show that it is a valid max-heap.

One way to solve this problem is to ignore the max-heap part temporarily, and first try to find any array with 7 items and 21 inversions. To create an array with the highest number of inversions, we can put the items in descending order, such that the first item (7) has an inversion with the 6 items after it, then the next item (6) has an inversion with the 5 items after it, and so on. The total number of inversions is $6 + 5 + 4 + 3 + 2 + 1 = 21$. At this point, we just have to draw the corresponding heap to verify that it is a valid max-heap.



- (d) (2.5 points) Construct a max-heap with the values 1, 2, 3, 4, 5, 6, 7, such that the underlying array has 12 inversions.

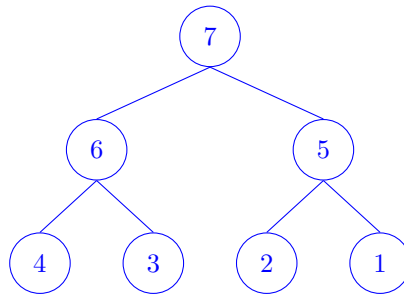
(Note: This is the minimum possible number of inversions in a valid max-heap with 7 elements.)

—	7	3	6	1	2	4	5
---	---	---	---	---	---	---	---

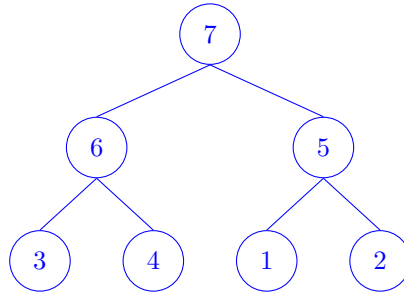
Solution: An inversion is defined as a pairing of indices i and j where $i < j$ and $\text{arr}[i] > \text{arr}[j]$. There are 12 inversions in this array: 7-3, 7-6, 7-1, 7-2, 7-4, 7-5, 3-1, 3-2, 6-1, 6-2, 6-4, 6-5. Drawing this array will also show that it is a valid max-heap.

Method 1:

One way to solve this problem is to first draw any valid max-heap, and then shuffle the items around to try and decrease the number of inversions, while maintaining the max-heap property. As an example, we can start with the 21-inversion array from the previous subpart, $[7, 6, 5, 4, 3, 2, 1]$:

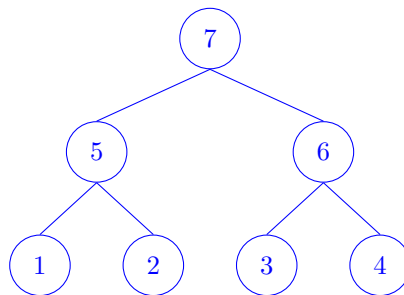


We can swap 4 and 3, and we can also swap 2 and 1, without breaking the max-heap property.



Now, the underlying array is $[7, 6, 5, 3, 4, 1, 2]$, and the number of inversions has decreased to $6 + 5 + 4 + 2 + 2 + 0 = 19$. Note: Each term of the sum corresponds to one item in the array, i.e. 7 has 6 inversions with later items, and 6 has 5 inversions with later items, and 5 has 4 inversions with later items, and 3 has 2 inversions with later items (3-1 and 3-2, but not 3-4), and 4 has 2 inversions with later items, and 1 has no inversions with later items.

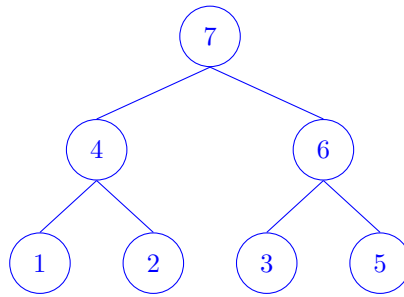
We could also swap the entire left sub-tree with the entire right sub-tree, which should also help reduce the number of inversions, since it will place 5 before 6, and $[1, 2]$ before $[3, 4]$ in the underlying array:



The underlying array is $[7, 5, 6, 1, 2, 3, 4]$, which has $6 + 4 + 4 + 0 + 0 + 0 = 14$ inversions.

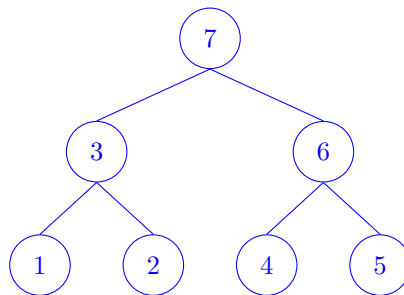
The final few tricks are a bit harder to come up with using this solution (start with 21 inversions and shuffle), but intuitively: 7 has to be the root of the max-heap, since it's the largest item (and max-heaps by definition store the largest item in the root). Therefore, we have no way to get rid of the 6 inversions involving 7 and the other items in the array.

However, putting the number 5 early in the array is costing us 4 inversions. If we placed the number 5 near the end of the array, we could reduce the number of inversions. For example, here's what would happen if we swapped 5 to the end of the array/heap:



Now, the array representation is $[7, 4, 6, 1, 2, 3, 5]$, which has $6 + 3 + 4 + 0 + 0 + 0 = 13$ inversions.

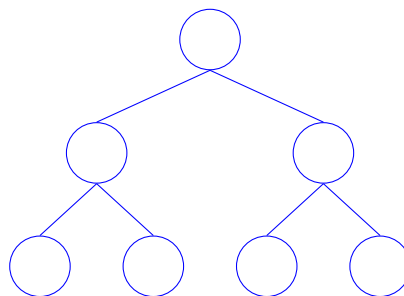
Again, looking at the underlying array, the number 4 being early in the array is costing us 3 inversions. We could try to swap 4 near the end of the array to fix this, but we don't want to swap 4 and 5 (since this would undo the improvement we just made). But if instead, we swapped 4 and 3, we would get this heap:



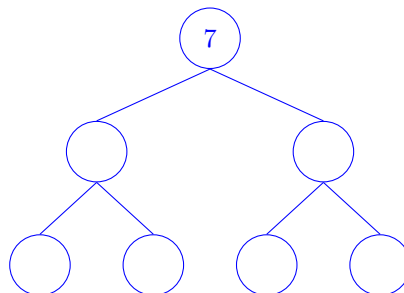
Now, the underlying array is $[7, 3, 6, 1, 2, 4, 5]$, which has $6 + 2 + 4 + 0 + 0 + 0 = 12$ inversions, which meets the requirement in the question, so we are done.

Method 2:

Another way to solve this problem is to start with a 7-item heap with the items not filled in yet, and fill in items (possibly with some trial-and-error, though that's not shown here) to minimize the number of inversions.

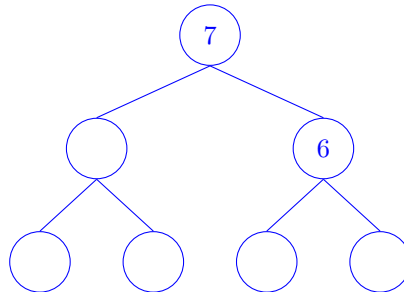


First, we know that the largest item has to be in the root of the max-heap, so we can fill in 7:

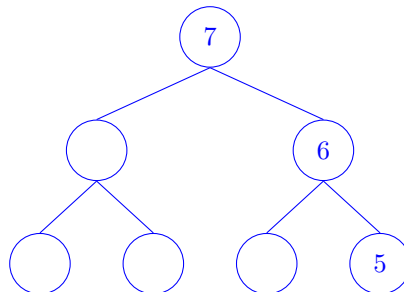


Since our goal is to minimize the number of inversions, we want large numbers to appear near the end of the heap (i.e. bottom-right). In other words, large numbers prefer to be near the bottom of the heap, or near the right of the heap, when possible.

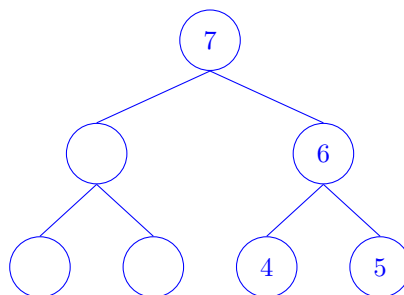
Can we place 6 anywhere at the bottom layer? This would mean that 6's parent has to be greater than 6, but less than 7. We don't have any elements that fit that criteria. Therefore, 6 has to be in the middle layer. To minimize inversions, we'll place 6 in the right-most node of the middle layer:



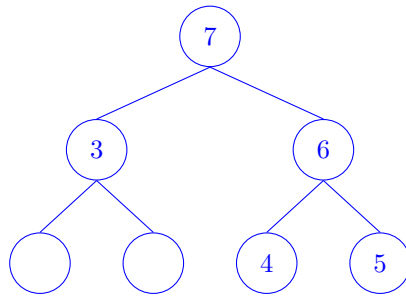
Next, let's consider 5. Putting 5 at the bottom-right node would preserve the max-heap property so far, so we can try that out. (It turns out that putting 5 in the bottom-right is correct, though we haven't proven that fact. If this placement was incorrect, you could always backtrack to this point and start over by placing 5 elsewhere, i.e. trial-and-error.)



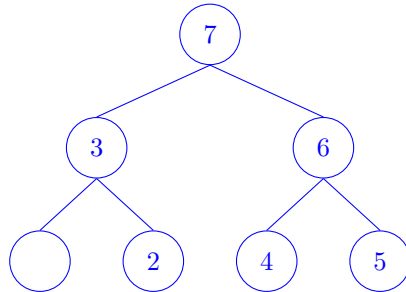
Next, let's place 4 in the bottom-right-most empty spot, since this also preserves the max-heap property (6 is greater than both 5 and 4):



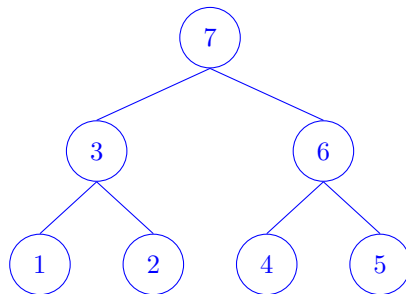
At this point, it might be tempting to put 3 in the bottom layer, but this will not work, because the remaining items (2 and 1) are both ineligible to be parents of 3. Therefore, of the remaining items, we have to put 3 in the middle layer:



Now, we can put 2 in the right-most remaining spot (remembering that we want larger items to appear further to the right to minimize inversions):



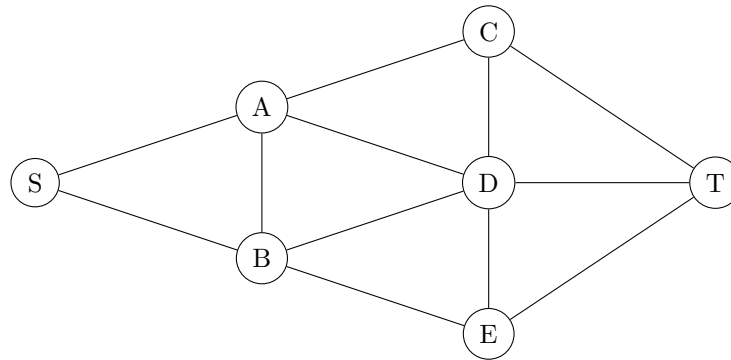
This leaves us with just one place to put 1:



The resulting array has 12 inversions (as seen earlier in this solution), so we are done.

5 In-Depth Analysis (5 points)

- (a) (1 point) Select the correct DFS preorder for the graph below, starting at node S. Break ties prioritizing the node that is alphabetically first.



- ☒ S, A, B, D, C, T, E
☐ S, A, B, C, D, E, T
☐ S, A, C, T, D, B, E

Solution: The quickest solution is: at every node, find the neighbor that is alphabetically first and not yet in our traversal, and visit that node. This works here since DFS pre-order corresponds to the order in which nodes are first visited (i.e. when the recursive DFS method is first called on that node).

Start at S. Preorder: [S]

S has neighbors A and B, and A comes alphabetically first. Preorder: [S, A]

A has neighbors B, D, C, S, and B comes alphabetically first. Preorder: [S, A, B]

B has neighbors A, D, E, S, and D comes alphabetically first. (We don't chose A since it's already been visited.) Preorder: [S, A, B, D]

D has neighbors A, B, C, E, T, and C comes alphabetically first. (A and B were already visited.) Preorder: [S, A, B, D, C]

C has neighbors A, D, T, and T is the only unvisited node. Preorder: [S, A, B, C, T]

T has neighbors C, D, E, and E is the only unvisited node. Preorder: [S, A, B, C, T, E]


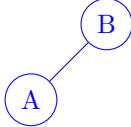
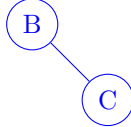
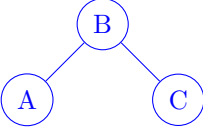
- (b) (2 points) A rooted binary tree of height at most 10 has the same **in-order** and **post-order** traversal.

What is the maximum number of nodes that could be in this tree?

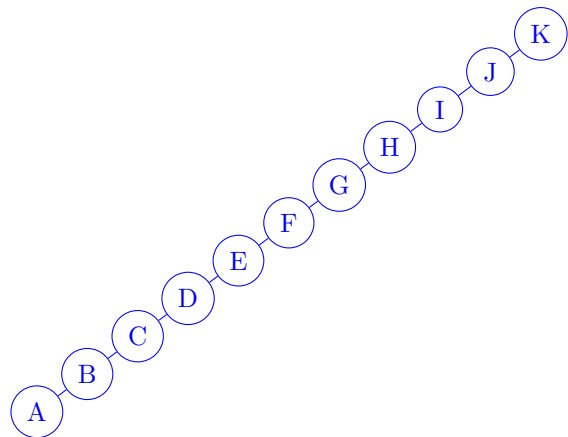
Note: A tree with only the root has height 0.

11

Solution: As a smaller example, let's simplify the problem to a rooted binary tree of height at most 1. We have 4 possible trees that we can build.

label	1	2	3	4
graph				
pre-order	B	B A	B C	B A C
in-order	B	A B	B C	A B C
post-order	B	A B	C B	A C B

Of these, only graphs 1 and 2 have the same in-order and post-order traversal. That is, if any node has a right-child, then the in-order and post-order traversals diverge. Thus, the only valid tree would be a tree with height 10 that has only left-children. Such a graph would look like this:



This tree has 11 nodes, is height 10, and both the in-order and post-order traversals result in: A, B, C, D, E, F, G, H, I, J, K.

Note that the traversal is value-independent. Even if `A.val == B.val`, traversing `A→B` is distinct from traversing `B→A`. Thus, having many duplicate-value nodes does not impact the answer.

- (c) (2 points) A rooted binary tree of height at most 10 has the same **pre-order** and **post-order** traversal.

What is the maximum number of nodes that could be in this tree?

1

Solution: First, see the table in the solutions to part (b). Of the graphs in the table, only graph 1 has the same pre-order and post-order traversal. That is, if any node has a left-child or right-child, then the pre-order and post-order traversals diverge. Thus, the only valid tree would be a tree with height 0, which contains exactly 1 node.

Note that the traversal is value-independent. Even if `A.val == B.val`, traversing `A→B` is distinct from traversing `B→A`. Thus, having many duplicate-value nodes does not impact the answer.

6 Eddie Sort (16 points)

Eddie is sorting a list of exam grades. Halfway through sorting them, he discovers a box of misplaced exams, and needs to integrate their grades into his sort!

Eddie starts off with the following initial list of grades, and decides to run insertion sort to sort them:

[28, 97, 44, 25, 11, 84]

Partway through the sorting algorithm, the list of grades has reached the following state:

[25, 28, 44, 97, 11, 84]

At this point, Eddie receives the following new grades: [32, 84, 28, 61].

- (a) (2 points) To integrate the new grades into his list, Eddie appends the numbers to the end of his list, so that the list is now:

[25, 28, 44, 97, 11, 84, 32, 84, 28, 61]

Eddie then resumes running insertion sort from where he left off. What is the state of the list after **four additional** elements have been correctly inserted into their correct position?

11	25	28	32	44	84	84	97	28	61
----	----	----	----	----	----	----	----	----	----

Solution: The initial list of grades is [28, 97, 44, 25, 11, 84]. Partway through the sorting algorithm, we have reached the following state: [25, 28, 44, 97, 11, 84]. From this, we can observe that the sorted sublist comprises of the first 4 elements, [25, 28, 44, 97], and the unsorted sublist comprises of the last 2 elements, [11, 84]. This means that the last element that we moved into the correct position at this point is 25, since all elements after 25 *in the initial list* are untouched.

After receiving and integrating the new grades into the list, the list is now [25, 28, 44, 97, 11, 84, 32, 84, 28, 61]. Moving 4 additional elements into their correct position means that we will be moving the next 4 unsorted elements (11, 84, 32, 84) into their correct position. This yields a list that looks like [11, 25, 28, 32, 44, 84, 84, 97, 28, 61], where [11, 25, 28, 32, 44, 84, 84, 97] is the sorted sublist and [28, 61] is the unsorted sublist.

- (b) (4 points) Suppose that Eddie has completely sorted a list of N items with insertion sort. At this point, he receives P additional items, appends them to the end of the sorted list, and continues running insertion sort, starting from the first additional item.

What is the **additional** asymptotic runtime needed to sort the entire list of $N + P$ items? Do **not** include the time spent sorting the initial N items.

Best Case:

$\Theta(P)$

Worst Case:

$\Theta(P(N + P))$

Solution:**Best case:**

Method 1: The P additional items are all larger than the original N items, and already in sorted order. When we make a pass over each item for insertion sort, no items are swapped. Thus, it takes $\Theta(P)$ additional time.

Method 2: The best-case runtime for insertion sort on $N + P$ items is $\Theta(N + P)$, and the best-case runtime that for insertion sort on the original list is $\Theta(N)$. Subtracting them results in $\Theta(N + P) - \Theta(N) \sim \Theta(P)$ time.

Worst case:

Method 1: The additional items are all smaller than the original N items, and in reverse sorted order (which is the worst case for insertion sort). The 1st additional item has to make N swaps, the 2nd additional item has to make $N + 1$ swaps, ..., and the P th additional item has to make $N + P - 1$ swaps. Adding up all the work done results in

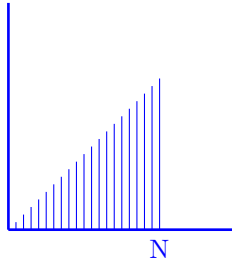
$$\sum_{i=0}^{P-1} (N + i) = NP + \frac{P(P-1)}{2}$$

swaps, which can be simplified (after dropping all asymptotically insignificant terms and constants) to:

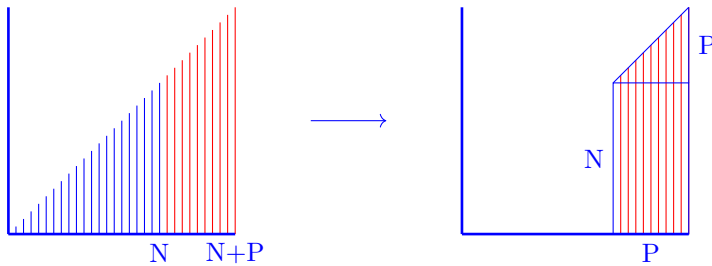
$$\Theta(NP + P^2) = \Theta(P(N + P))$$

Method 2: The worst-case runtime for insertion sort on $N + P$ items is $\Theta((N + P)^2)$, and the worst-case runtime for insertion sort on the original list is $\Theta(N^2)$. Subtracting them results in $\Theta((N + P)^2) - \Theta(N^2) \sim \Theta(NP + P^2) = \Theta(P(N + P))$ time.

Method 3: You may have seen in discussion that a graphical way of justifying insertion sort's $\Theta(N^2)$ worst-case runtime is to characterize the swaps as a bar graph, where the total work done (and thus the runtime) can be interpreted as the “area” of the triangle that is formed (which is proportional to N^2).



By inserting P additional elements, the additional runtime can be characterized as the “area” of the red bars below.



The red bars can be broken up into a rectangle of area NP and a triangle of area $\frac{P^2}{2}$. After adding them up and dropping asymptotically insignificant terms and constants, we will net the final answer $\Theta(NP + P^2) = \Theta(P(N + P))$.

Eddie observes that insertion sort seems to work properly if new data is appended to the end of the list, regardless of when that new data arrives. Eddie is curious to know if any other sorting algorithms can similarly work properly if new data is added partway through the sorting algorithm.

For each of the following sorting algorithms, determine whether the sorting algorithm would still produce a correctly sorted list if new data is added in the specified manner. Unless otherwise specified, the new data may be added at any intermediate step of the sorting algorithm.

(c) (2 points) Selection sort

The new data is appended to the end of the list.

- ☐ Always produces a sorted list ☒ Does not always produce a sorted list

Solution: If we start with a list $[3, 6, 7, 2]$ and perform 2 iterations of selection sort, we will end up with $[2, 3, 7, 6]$. If at this point we receive $[1, 5, 4]$ and perform the rest of selection sort to completion, we will end up with $[2, 3, 1, 4, 5, 6, 7]$, which is incorrectly sorted.

More generally, the problem here is that in selection sort, once an item is swapped to the front of the array, we never move that item again, because we do not expect any smaller items to be added to the array later.

(d) (2 points) Heapsort (without in-place heapification)

The new data is inserted, one element at a time, into the heap.

- ☐ Always produces a sorted list ☒ Does not always produce a sorted list

Solution: Recall that in naive heapsort (not in-place), we insert all items into a heap (represented by a separate heap), and then delete all items from the heap one-by-one.

Suppose we have the items $[3, 4, 5, 6]$, and we insert them into a min-heap. (We'll assume the heap is a min-heap here, though the solution holds if you use a max-heap instead.)

Then, we delete the smallest item (3) from the heap, and place it at the front of the sorted array. Now, the heap contains $[4, 5, 6]$.

At this point, suppose we add $[1, 2]$ into the heap, such that the heap now contains $[1, 2, 4, 5, 6]$.

Next, we delete the smallest item (1) from the heap, and place it in the next index of the sorted array. Our sorted array now starts with $[3, 1, \dots]$, which is incorrect.

More generally, the problem here is that in naive heapsort, once we delete the smallest item and place it at the front of the result array, we will never move that item in the result array, because we do not expect any smaller items to be added to the heap later.

(e) (2 points) Mergesort

The new data is sorted, then merged into any sublist immediately before a merge step.

- ☒ Always produces a sorted list ☐ Does not always produce a sorted list

Solution: Recall that the merge step takes in two sorted sublists as input, and outputs a merged list with the contents of the two sublists, all sorted.

If our new data is sorted, and we perform a merge operation with our new data and a sublist, then the resulting sublist will also be sorted (since the merge operation took in two sorted inputs, it will correctly output a sorted sublist).

Then, the merge step immediately following our addition of new data will take in two sublists (the one we just built with our own merge step, and some other sorted sublist), and create a larger sublist that is sorted.

More generally, the way in which we add new data does not break any of the invariants of mergesort—the inputs to the merge operations are always sorted, so the outputs of the merge operations will also be sorted. Therefore, mergesort will still work as intended.

(f) (2 points) MSD Radix sort

The new data is inserted into its corresponding subproblem list based on each element's most significant digits, immediately before a pass of counting sort.

- ☒ Always produces a sorted list ☐ Does not always produce a sorted list

Solution: Immediately before a pass of counting sort begins, the main invariant is that the data is sorted by all of the most significant digits (up until the digit that's about to be sorted). In other words, inside each subproblem, the most significant digits (up until the digit that's about to be sorted) are the same, and the list is sorted by those most significant digits.

As a concrete example, after two passes of counting sort, suppose we had the following order (brackets added to denote subproblems): $[[203], [215, 219, 210], [753, 752]]$. The data is sorted by the top two digits, and inside each subproblem, the top two digits are the same. It is okay that within a subproblem, the numbers are not sorted yet (e.g. $[753, 752]$ is not sorted), since we haven't considered digits past the top two digits yet.

At this point, if we insert the new data into the corresponding subproblems, based on the most significant digits, we are preserving all the invariants. The list is still sorted by the most significant digits, and we don't have to worry about the sorting within each subproblem yet. If we continue with more passes of counting sort after inserting data, the resulting sort will be correct.

(g) (2 points) Quicksort with a 3-scan partitioning scheme

The new data is appended to the end of the list immediately before a partitioning step.

- ☐ Always produces a sorted list ☒ Does not always produce a sorted list

Solution: Suppose we had the list $[5, 8, 6, 4, 3, 7]$. After partitioning on 5 with 3-scan partitioning, the resulting list is $[4, 3, 5, 8, 6, 7]$.

At this point, we would recursively quicksort the left partition $[4, 3]$ and the right partition $[8, 6, 7]$ separately. Note that items in the left subproblem will never cross over the pivot (3) into the right subproblem, and vice-versa.

However, if at this point (just before a partitioning step), we decide to add more data like $[1, 2]$ at the end of the list, the right subproblem is now $[8, 6, 7, 1, 2]$. The additional items are on the wrong side of

the pivot, and will never cross over to the correct side, so the sort will no longer be correct.

More generally, the problem here is that in quicksort, once we partition on an item, we are assuming that the items in each subproblem are on the correct side of the pivot. This allows us to solve the subproblems separately, since we know that items in each subproblem will never cross over into the other subproblem. If we add items to the end of the list (right subproblem) that really belong in the left subproblem, those additional items will never cross over into the correct subproblem.

7 Two Seemingly Unrelated Problems (22 points)

Problem I. Consider a connected¹, unweighted, directed acyclic graph with N nodes.

- (a) (4 points) What is the worst-case and best-case asymptotic growth of the longest path in the graph, in terms of N ? Provide a Θ bound.

Best Case:

$$\Theta(1)$$

Worst Case:

$$\Theta(N)$$

Solution:

Best case:

See the example below for how the best case is achieved. Since we found an example where the length of the longest path is $\Theta(1)$, we know that the answer has to be less than or equal to $\Theta(1)$.

$\Theta(1)$ (i.e. length of the longest path does not scale with N) is the lowest possible theta-bound in this context, so we know that the answer has to be greater than or equal to $\Theta(1)$.

Since we proved the answer has to be less than or equal to $\Theta(1)$, and it also has to be greater than or equal to $\Theta(1)$, we found a tight bound, and we're done.

Worst case:

See the example below for how the worst case is achieved. Since we found an example where the length of the longest path is $\Theta(N)$, we know the answer has to be greater than or equal to $\Theta(N)$.

We know that the DAG has no cycles (by definition), so the longest path cannot be any longer than $\Theta(N)$ (which would be a path visiting every node). If we had a longer path, we'd be visiting nodes multiple times, which would create cycles. This tells us that the answer has to be less than or equal to $\Theta(N)$.

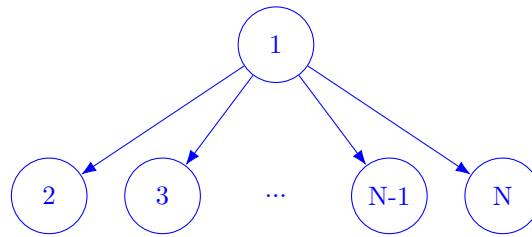
Since we proved the answer has to be less than or equal to $\Theta(N)$, and it also has to be greater than or equal to $\Theta(N)$, we found a tight bound, and we're done.

Note: Technically, these explanations go into more detail than necessary to explain why these bounds are tight. You did not need to go through all of this reasoning to solve these questions on the exam.

- (b) (4 points) Draw graphs that exhibit the best-case and worst-case longest path. Your examples must be clearly extendable to arbitrarily many nodes; for clarity, you may write in English exactly how to extend your example to arbitrarily many nodes.

Best Case:

¹Formally, a *weakly connected* directed graph, which is a directed graph whose underlying undirected graph (i.e. with arrow directions removed) is connected.



Consider this “tree” topology, where all nodes are direct children of node 1.
As the number of nodes N increases, the longest path in the DAG is still length 1.
Other solutions exist.

Worst Case:



In this “linked list” topology, the longest path in a DAG of N nodes has length $N - 1$.
Other solutions exist.

Problem II. Note: This question is inspired by a question written by *Ravi Fernando*.

Consider a new sorting algorithm: *Booksort* takes in a list of N unique numbers from 1 to N inclusive (representing, for example, books numbered 1 through N on a bookshelf), and sorts the list in ascending order by the following process:

Step 1: Select a book that is not in the correct slot in the list; if this is not possible, the list is sorted.

Step 2: Place the book in its correct final location in the list, shifting all intermediate items by one. For this question, we will assume that this step can be accomplished in $\Theta(1)$ time.

Step 3: Repeat Steps 1 and 2 until the list is sorted.

For example, imagine we have five books in the following order:

2	4	3	5	1
---	---	---	---	---

Out of these books, book 3 is in its correct slot (and thus cannot be selected in Step 1), and all other books are in the wrong slot.

Suppose we select book 4. First, we take book 4 out of the bookshelf:

2		3	5	1
---	--	---	---	---

4

Then, we shift books 3 and 5 to the left (in constant time):

2	3	5		1
---	---	---	--	---

4

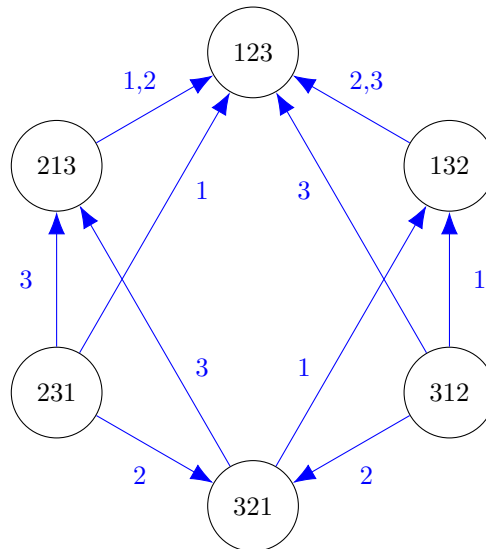
Finally, we insert 4 in its correct slot:

2	3	5	4	1
---	---	---	---	---

Note that by putting book 4 in its correct slot, we have also moved book 3 to an incorrect slot. The bookshelf can be fully sorted if, from the above position, we select book 1, then either book 5 or book 4.

- (c) (2 points) A *state transition diagram* is a directed graph, where each node represents one of the possible orders that the books can be in. A directed edge from node A to node B is added if we can go from state A to state B in a single iteration of steps 1 and 2.

Draw the edges to complete the state transition diagram for $N = 3$ below. Draw at most one arrow between two states, even if multiple choices in step 1 yield the same transition.



Solution: Edge labels were not required for a correct solution, but are provided here for clarity. The edge labels represent what book(s) you can select to cause that transition to occur. For example, the

edge from 213 to 123 is labeled, 1,2 because given 213, you can select 1 (shifting 2 to the right), or you can select 2 (shifting 1 to the left), and the result would be 123.

Perhaps surprisingly, no matter how we choose the books in step 1, the algorithm is guaranteed to terminate in finite time. However, the way we choose the books in step 1 heavily affects the runtime of this algorithm.

- (d) (2 points) What is the worst-case asymptotic runtime of this algorithm if, at each step 1, we always search for (in $\Theta(N)$ time) and select the **smallest** book that is not in its correct slot? Provide a Θ bound.

$$\Theta(N^2)$$

Solution: The key realization to solve this problem is to note that the algorithm described is very similar to selection sort.

In particular, we are running a $\Theta(N)$ step to find the smallest book. Then, we are running a $\Theta(1)$ step to place that book in its correct slot, at the front of the bookshelf. This is exactly the same runtime analysis that we did with selection sort.

Also, note that just like in selection sort, once we place a book in its correct slot at the front of the bookshelf, we won't be shifting that book again. The book is in its correct slot, so we won't choose to take that book out again. Also, we'll never put a book in front of an already-sorted book (which might cause already-sorted books to slide right). This is because we guaranteed that we're always selecting the smallest books first, so there will be no books "less than" the already-sorted books that would need to be placed in front of the already-sorted books.

Since selection sort has $\Theta(N^2)$ runtime, and we've shown that selection sort and this Booksort variant have the same behavior, we can safely say that this Booksort variant also has $\Theta(N^2)$ runtime.

Or, to be more precise, we can say that each iteration of this Booksort variant takes $\Theta(N)$ time and puts one book into its correct slot. Since there are N books, we have to run N iterations, for a total runtime of $\Theta(N^2)$.

For the next two parts, assume that in step 1, we always select (in $\Theta(1)$ time) the leftmost book that is not in its correct slot. As before, step 2 takes $\Theta(1)$ time.

- (e) (4 points) Find as tight an Ω bound as possible for the worst-case runtime of this sorting algorithm.

Hint: Consider the initial sequence $2, 3, 4, 5, \dots, N-1, N, 1$.

$$\Omega(2^N)$$

Solution:

Following the hint, let's try running this variant of Booksort on the provided example. For the sake of a cleaner solution, we'll set $N = 10$, though the solution works for arbitrary N . We'll also count each iteration in hopes of finding a pattern later.

[2, 3, 4, 5, 6, 7, 8, 9, 10, 1]

Iteration 1: The leftmost book not in its correct slot is 2, so we take out 2, slide 3 to the left, and place 2 in its correct slot:

[3, 2, 4, 5, 6, 7, 8, 9, 10, 1]

Iteration 2: Now, the leftmost book not in its correct slot is 3, so we take out 3, slide [2, 4] to the left, and place 3 in its correct slot:

[2, 4, 3, 5, 6, 7, 8, 9, 10, 1]

Iteration 3: Now, the leftmost book not in its correct slot is 2 again, so we take out 2, slide 4 to the left, and place 2 in its correct slot:

[4, 2, 3, 5, 6, 7, 8, 9, 10, 1]

Iteration 4: Now, the leftmost book not in its correct slot is 4, so we take out 4, slide [2, 3, 5] to the left, and place 4 in its correct slot:

[2, 3, 5, 4, 6, 7, 8, 9, 10, 1]

Iteration 5: The leftmost book not in its correct slot is 2 again, so we take out 2, slide 3 to the left, and place 2 in its correct slot:

[3, 2, 5, 4, 6, 7, 8, 9, 10, 1]

Iteration 6: The leftmost book not in its correct slot is 3 again, so we take out 3, slide [2, 5] to the left, and place 3 in its correct slot:

[2, 5, 3, 4, 6, 7, 8, 9, 10, 1]

Iteration 7: The leftmost book not in its correct slot is 2 again, so we take out 2, slide 5 to the left, and place 2 in its correct slot:

[5, 2, 3, 4, 6, 7, 8, 9, 10, 1]

Iteration 8: The leftmost book not in its correct slot is 5, so we take out 5, slide [2, 3, 4, 6] to the left, and place 5 in its correct slot:

[2, 3, 4, 6, 5, 7, 8, 9, 10, 1]

Iteration 9: Move 2: [3, 2, 4, 6, 5, 7, 8, 9, 10, 1]

Iteration 10: Move 3: [2, 4, 3, 6, 5, 7, 8, 9, 10, 1]

Iteration 11: Move 2: [4, 2, 3, 6, 5, 7, 8, 9, 10, 1]

Iteration 12: Move 4: [2, 3, 6, 4, 5, 7, 8, 9, 10, 1]

Iteration 13: Move 2: [3, 2, 6, 4, 5, 7, 8, 9, 10, 1]

Iteration 14: Move 3: [2, 6, 3, 4, 5, 7, 8, 9, 10, 1]

Iteration 15: Move 2: [6, 2, 3, 4, 5, 7, 8, 9, 10, 1]

Iteration 16: Move 6: [2, 3, 4, 5, 7, 6, 8, 9, 10, 1]

(On an exam, you could continue this until you clearly spot the pattern.)

The key realization is to note the iterations where we move a book for the first time:

Book 2 is chosen for the first time in Iteration 1.

Book 3 is chosen for the first time in Iteration 2.

Book 4 is chosen for the first time in Iteration 4.

Book 5 is chosen for the first time in Iteration 8.

Book 6 is chosen for the first time in Iteration 16.

In general, Book i will be chosen for the first time in Iteration 2^{i-2} , and Book 1 will be “chosen” (in a sense that it gets placed for the first time in the leftmost spot in the previous iteration) in Iteration 2^{i-1} .

We can prove this rigorously through induction (though this was not necessary during the exam). Let $f(N)$ denote the number of iterations it takes to complete Booksort on the array $[2, 3, 4, \dots, N-1, N, 1]$. Assume that for the $N-1$ case, the following statements are true:

1. $f(i) = 2^{i-1} - 1$
2. The book labeled 1 reaches the leftmost spot for the first time during the final iteration.

This is true for $i = 1$. If there is only one book, then $f(1) = 2^{1-1} - 1 = 0$, i.e. it takes 0 iterations to complete Booksort on a single book. It is also true that the book labeled 1 reaches the leftmost spot for the first time during the 0th and final iteration.

Note that until a book reaches the leftmost spot, its value is never read (and therefore cannot affect any iterations of Booksort), and any book in index i is unaffected by any step prior to the first time Book i (or a higher-numbered book) is moved. In particular, if we start at:

Iteration 0: $[2, 3, 4, 5, 6, 7, \dots, N-1, N, 1]$

Booksort will behave the same as if we had the array $[2, 3, 4, 5, 6, 7, \dots, N-1, N]$ until the N first reaches the leftmost position, which would also behave the same as if we had the array $[2, 3, 4, 5, 6, 7, \dots, N-2, N-1, 1]$ until the 1 first reaches the leftmost position.

By the inductive hypotheses, it takes $2^{N-2} - 1$ iterations of Booksort before the 1 first reaches the leftmost position, starting from the array $[2, 3, 4, 5, 6, 7, \dots, N-2, N-1, 1]$ (the $N-1$ case). Therefore, at Iteration $2^{N-2} - 1$ of the N case, we reach the following state:

Iteration $2^{N-2} - 1$: $[N, 2, 3, 4, 5, \dots, N-2, N-1, 1]$

Iteration 2^{N-2} : $[2, 3, 4, 5, \dots, N-2, N-1, 1, N]$

At this point, book N can no longer move, so this behaves the same as if the array was $[2, 3, 4, 5, \dots, N-2, N-1, 1]$ (the $N-1$ case). It thus takes a further $2^{N-2} - 1$ iterations to reach the final sorted state:

Iteration $2^{N-2} + 2^{N-2} - 1 = 2^{N-1} - 1$: $[1, 2, 3, \dots, N-1, N]$

We further note that the 1 is unmoved until iteration 2^{N-2} , and per the inductive hypothesis does not reach the leftmost position between iterations $2^{N-2} \leq i \leq 2^{N-1} - 2$. Thus, the book labeled 1 reaches the leftmost spot for the first time during the final iteration.

Therefore, both inductive hypotheses hold, and we can conclude that $f(N) = 2^{N-1} - 1 \in \Theta(2^N)$

Therefore, we can say that the worst-case runtime for Booksort is bounded by $\Omega(2^N)$. Note that we are using Ω here in our bound because we have not actually proven that the sequence $2, 3, 4, 5, \dots, N-1, N, 1$ actually results in the worst-case runtime for this Booksort variant. By considering this example, all that we have shown is that the runtime is $\Theta(2^N)$ or worse.

- (f) (6 points) Briefly explain why this sorting algorithm runs in worst-case $O(N!)$ runtime. You may assume (without explanation) that this algorithm is guaranteed to terminate in finite time.

Hint: Apply Problem I to the state transition diagram.

Solution: A fully-correct proof requires stating and justifying three facts.

Fact 1: The state transition diagram in part (c) is a directed acyclic graph, even when extended to arbitrary N .

Proof: There are no cycles in the graph, because the question says that the algorithm is guaranteed to terminate in finite time. If a cycle existed, there would exist a way to run the algorithm forever without terminating, by infinitely repeating the states in the cycle.

Fact 2: The state transition diagram in part (c) has $N!$ nodes when extended to arbitrary N .

Proof: There are N choices for the book in the first slot. For each choice of book in the first slot, there are $N - 1$ choices for the book in the second slot. For each choice of books in the first and second slot, there are $N - 2$ choices of book in the third slot, and so on. In total, the number of ways to arrange N books in some order is: $(N) \cdot (N - 1) \cdot (N - 2) \dots 3 \cdot 2 \cdot 1 = N!$

Fact 3: All paths in the state transition diagram have length that is $O(N!)$.

Proof: From Problem I, we showed that for a DAG with M nodes (using M instead of N to distinguish between the two problems here), the longest path in the DAG is $\Theta(1)$ in the best case and $\Theta(M)$ in the worst case. Therefore, we can say that the longest path in the DAG has length $O(M)$ in all cases.

In Problem II, the state transition diagram has $M = N!$ nodes (per Fact 2), is a DAG (per Fact 1). Therefore, we can say that the longest path in the state transition diagram has length $O(N!)$ in all cases.

Running Booksort on an array involves moving along transitions of the state transition diagram, starting from one state and ending at the fully-sorted state. Since it takes $\Theta(1)$ time to complete one iteration of Booksort, the runtime of Booksort is equivalent to the length of the path taken by Booksort in the state transition diagram, which by the above is $O(N!)$

Note: It is actually not correct to say that the paths in the state transition diagram are $\Theta(N!)$ in the worst case.

Remember that big-O informally means “less than or equal to”, and big-Theta informally means “equal to”, when comparing orders of growth.

In Problem I, we proved that for all DAGs (not just Booksort state transition diagrams, but other DAGs as well), the longest path is $\Theta(1)$ in the best case and $\Theta(M)$ in the worst case, which means that the longest path is $O(M)$ in all cases.

In Problem II, the Booksort state transition diagram is a DAG, so it's safe to say that the longest path is $O(M)$, since we said in Problem I that the longest path in a DAG is $O(M)$ in all cases.

However, it turns out that the Booksort state transition diagram is a special case of a DAG (with additional restrictions on what edges can and cannot exist). In particular, a “linked list” topology like the DAG shown in part (b) is not actually a valid Booksort state transition diagram. Therefore, it is not correct to say that the longest path in the state transition diagram is $\Theta(N!)$ in the worst case, because this implies that the worst case order-of-growth is “equal” to $\Theta(N!)$. We cannot state this equality because we never actually showed an example of the special-case state transition diagram achieving the $\Theta(N!)$ worst-case bound. Further, the worst-case path taken by Booksort is not necessarily a worst-case path in the original state transition diagram, since this particular Booksort variant is restricted against traversing any edge resulting from taking a non-leftmost book.

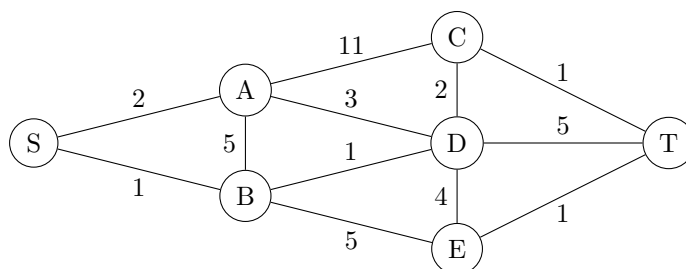
Since we never found a case of the $\Theta(N!)$ bound being achieved, the best we can say is that the length of the longest path in the state transition diagram in all cases is “less than” $\Theta(N!)$, i.e. the order of growth is always $O(N!)$. It turns out that finding the exact tight theta-bound for the worst-case longest path in the state transition diagram is extremely difficult (far beyond the scope of this class), and the tight bound is $\Theta(2^N)$. (Determining this requires proving that the algorithm terminates, which was the original problem written by Ravi Fernando.)

8 Lost in the Ceryneian H.I.N.D. (15 points)

After being repeatedly hacked in past exams, you’ve finally tracked down the hackers responsible as members of the Ceryneian H.I.N.D. With the assistance of an investigator (codenamed Heracles), you have managed to infiltrate their internal network. Help Heracles catch the Ceryneian H.I.N.D.

The internal network is a weighted undirected graph, with start vertex S and target vertex T . You want to **delete one edge** from the graph, such that when deleted, the shortest path length from S to T is **increased** by as much as possible.

For example, consider the graph below:



Before deleting an edge, the shortest path from S to T has length 5. If we delete edge BS , the shortest path from S to T will instead have length 8; deleting any other edge will result in a shortest path from S to T that is shorter than 8. Thus, we should delete edge BS .

Design an algorithm (in English or pseudocode) that, given a graph, returns the edge that should be deleted. If multiple edges yield the same maximal shortest-path length, your algorithm may return any of them. You may assume that:

- All edges have positive integer edge weights.
- $S \neq T$
- The input graph is connected, and cannot be disconnected by removing any one edge.
- You have access to a working black-box implementation of Dijkstra’s algorithm, which runs in $\Theta(E \log(V))$ time.

For credit, your algorithm must run in $O(VE \log(V))$ time, where V and E are the number of vertices and edges in the graph, respectively.

Solution:

First, run Dijkstra’s algorithm on the unmodified graph to get the shortest path from S to T .

Then, for every edge on the shortest path from S to T , run Dijkstra’s algorithm on the graph with just that edge removed, and record the resulting distance from S to T . Return the edge that, when removed, resulted in the largest distance from S to T .

Intuition:

Sometimes, when solving algorithm-design questions, it helps to come up with a naive, slow algorithm first, and then work on optimizing it.

The naive solution to this problem would be to check every edge. In other words, run Dijkstra’s algorithm E times, with a different edge removed each time, and return the edge that, when removed, resulted in the largest distance from S to T .

However, this naive algorithm is too slow. Each run of Dijkstra’s algorithm takes $\Theta(E \log(V))$ time, and we are running Dijkstra’s algorithm E times, for a total runtime of $\Theta(E^2 \log(V))$.

The key realization to optimize this solution is: If an edge is not on the original shortest path from S to T , then removing that edge will not increase the shortest path length at all, since we can just use the same original shortest path. **Therefore, we only have to check edges that were on the original shortest path from S to T .**

Finally, to finish the problem, we can verify that our optimized solution meets the required runtime bound.

Since all edge weights are positive, the shortest path from S to T will not contain cycles. (Cycles would just increase the length of the shortest path.) Therefore, the shortest path has most $V - 1$ edges (which would occur if the shortest path passed through every single vertex). The shortest path would not pass through a vertex twice, since that would create a cycle.

We have to run Dijkstra’s algorithm once on the unmodified graph to get the original shortest path from S to T . Then, we run Dijkstra’s algorithm $V - 1$ more times, to check each edge on the shortest path from S to T . This means we run Dijkstra’s algorithm a total of V times. Each run of Dijkstra’s algorithm takes $\Theta(E \log(V))$, for a total runtime of $\Theta(VE \log(V))$. This meets the $O(E \log(V))$ bound specified in the question, and we are done.

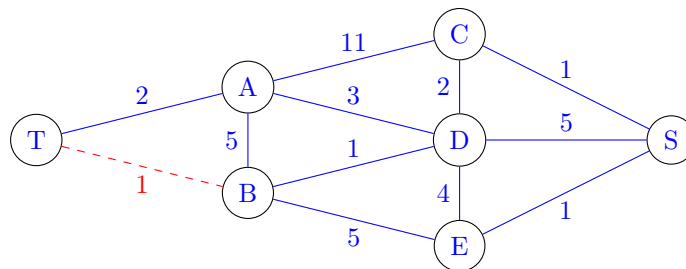
Note: It is possible (and indeed strongly suspected) that a more efficient solution exists. Full credit would be awarded for a fully correct solution that yields equal or higher efficiency, even if it did not match our solution.

Also, we don’t know if the more efficient solution still uses Dijkstra’s algorithm as a black-box. (Black-box means that we are calling Dijkstra’s algorithm as a “helper function” while totally abstracting away its internal implementation.) We suspect that more efficient solutions would no longer black-box Dijkstra’s algorithm, and would need to be designed from scratch (though this would be allowed within the context of the question).

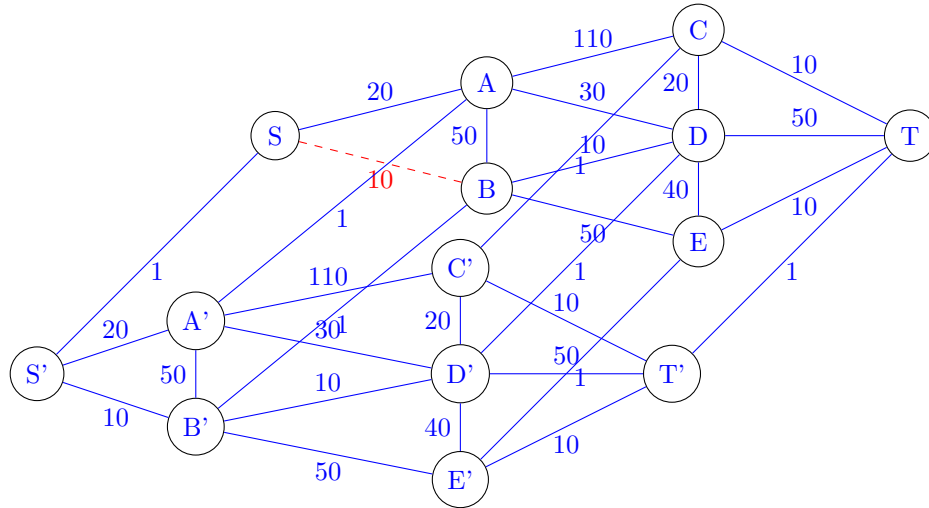
If you have a solution that’s different from our intended solution, and you’d like to verify if it works, consider the following example graphs, which together catch many common erroneous algorithms. This list is not exhaustive, and an algorithm may still be incorrect even if it yields the correct answer on all the below test cases.

Test Cases:

Test Case 1 (Reverse): S and T swapped (which wouldn’t change which edge is optimal to remove). Notably, the removed edge is NOT adjacent to S .

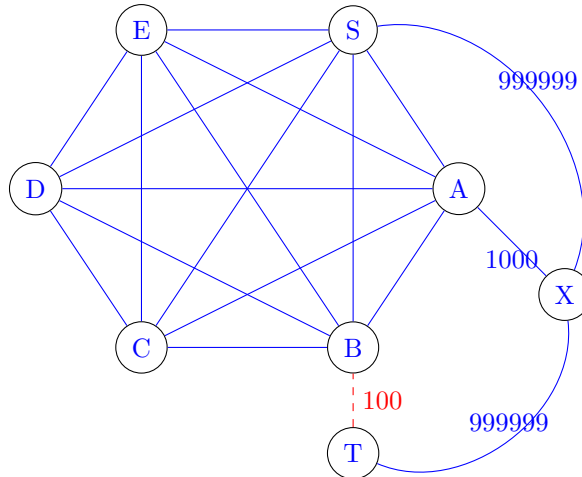


Test Case 2 (Duplicate): Make two copies of the graph, with the edge weights multiplied by 10. Then, add an edge of weight 1 between each node and the corresponding node in the copied graph. The optimal path should never go into the second copy, since the start and target remain in the first graph. Notably, the removed edge is NOT the shortest outgoing edge of any vertex.

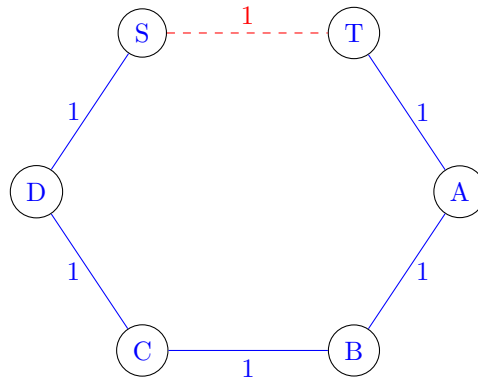


Test Case 3 (Lollipop): N vertices fully connected, plus two additional nodes X and T shown below. $N = 6$ in the graph below, but this can be extended to infinitely many nodes. All unlabeled edges have an arbitrary but small weight (for example, random integers between 1 and 10).

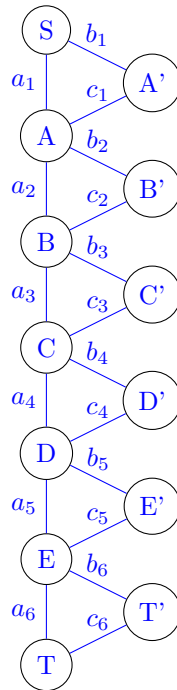
Notably, the path that ends up being taken is the $\Theta(N!)$ th shortest path, and the weights of edges between S, A, \dots, E can be selected so that iterating over all the neighbors of vertices in the original shortest path iterates $\Theta(V^2)$ times. Further, the path that ends up being taken is NOT the combination of the shortest path (in the original graph) from node i to S and the shortest path (in the original graph) from node i to T , for any node i .



Test Case 4 (Closure of Hayward Bridge): N vertices connected in a circle, with S and T initially connected. Notably, no edges in the original shortest path are taken.



Test Case 5 (Sawtooth): A sequence of triangles, with weights such that taking the straight path is optimal. Notably, the values of b_i and c_i can be manipulated to make any edge in the original shortest path be the optimal choice, independent of the weights of the edges along the shortest path. Further, removing any vertex (and the surrounding edges) either disconnects the graph, or does not affect the shortest path. In the example below, the optimal edge to remove is neither the lightest, nor the heaviest edge on the original shortest path.



$$a_i < b_i + c_i$$

Remove a_i that maximizes $(b_i + c_i) - a_i$

Example:

i	a_i	b_i	c_i
1	1	1	1
2	2	3	1
3	60	99999	5
4	100	40	70
5	4	7	4
6	3	2	2

Extra Answer Space for Questions 7-8

Do not tear this page off (even if you leave it blank).

If you need more space for your answers to Question 7(f) and/or Question 8, you may use this page. On this page, you must clearly indicate which parts are associated with 7(f) and/or 8.

If you want us to grade anything on this page, **you must draw an arrow** (\rightarrow) at the end of the original box to tell us that your answer continues on this page.

Nothing on this page is worth any points. Do not tear this page off.

9 Crowdsourced Literature

(0 Points)

As a class, create a meaningful poem. You may write up to three words in the box below. After the exam, course staff will attempt to combine all valid fragments into a single poem.

10 Feedback

(0 Points)

Leave any feedback, comments, concerns, or more drawings below!

Hope you had a great semester!