Section 10: Solutions

1. Finding big-O

For each of the following, find a tight, simplified big-O bound using Master Theorem, or state that Master Theorem doesn't apply to that recurrence.

(a) $A(x) = \begin{cases} 1 & \text{if } x = 1 \\ 3A(\frac{x}{3}) + 3x^2 & \text{otherwise} \end{cases}$

Solution:

Using the master theorem, we know that $\log_b(a) = \log_3(3) = 1 < 2 = c$, so $A(x) \in \mathcal{O}\left(x^2\right)$.

(b) $B(n) = \begin{cases} 1 & \text{if } n = 1 \\ 8B(\frac{n}{3}) + 2n & \text{otherwise} \end{cases}$

Solution:

Using the master theorem, we know that $\log_b(a) = \log_3(8) > 1 = c$, so $B(n) \in \mathcal{O}\left(n^{\log_3(8)}\right)$.

(c) $C(t) = \begin{cases} 3 & \text{if } t = 1 \\ 4C(\frac{t}{5}) + t^3 + 2 & \text{otherwise} \end{cases}$

Solution:

Applying Master Theorem, $\log_b(a) = \log_5(4) < 1 < 3 = c$ so the final answer is $\mathcal{O}\left(n^3\right)$.

(d) $D(m) = \begin{cases} 3 & \text{if } m = 1 \\ 3D(m-1) + 2 & \text{otherwise} \end{cases}$

Solution:

The master theorem does not apply here – the expression $T(m-1)$ in no way resembles $T(m/b)$.

2. Memory, locality, and dictionaries

(a) In lecture, we discussed three different optimizations for disjoint sets: union-by-rank, path compression, and the array representation.

If we implement disjoint sets using `Node` objects with a "data" and "parent" field and implement the first two optimizations, our `find` and `union` methods will have a nearly-constant average-case runtime.

In that case, why do we bother with the array representation?

Solution:

Although the array representation will not help make our data structure more asymptotically optimal, we still use it for several reasons:

(i) It lowers the overall amount of memory we consume, especially in Java. Java adds extra bytes of overhead whenever you create a new object – this means that depending on exactly how your system is configured, we may end up using anywhere from 16 to 32 bytes per each element if we were to use the Node and pointer approach.

In contrast, if we use an array, we use only 4 bytes per each element – just enough to store the int.

(ii) Using an array will likely be faster than a linked list due to space locality. Every time we access something in the array, we will also likely drag in the next several elements.

This may end up not helping if we need to jump to wildly distant locations in the array (e.g. if we were to visit indices 1, 2000, then 300, for example, space locality probably doesn't help). However, it may help if we end up needing to "probe" or jump to nearby locations in the array.

(b) Suppose you implement a dictionary with a sorted array and another with an AVL tree. Consider the time needed to iterate over the key-value pairs of a `SortedArrayDictionary` vs an `AvlDictionary`. It turns out that iterating over the `SortedDictionary` is nearly 10 times faster than iterating over the `AvlDictionary`. Think about why that might be.

Now, suppose we take those same dictionaries and try repeatedly calling the `get(...)` method a few hundred thousand times, picking a different random key each time.

Surprisingly, we no longer see such an extreme difference in performance. The `SortedArrayDictionary` is at most only about twice as fast as the `AvlDictionary`.

Why do you suppose that is? Be sure to discuss both (a) why the difference in performance is much less extreme and (b) why `SortedArrayDictionary` is still a little faster.

Solution:

Iterating over the SortedArrayDictionary was significantly faster than iterating over the AvlDictionary primarily because the iterator for SortedArrayDictionary was able to take full advantage of space locality.

When we visit the initial item in the array, we load in the next several elements, speeding up the time needed to access and return the next batch of element.

In contrast, when we visit random keys, we don't really take advantage of cache locality. The `get(...)` method likely finds the key-value pairs by using binary search, and binary search will, for the most part, probe distant locations in the array. So, if we initially check index 1000, we might check index 500 next, then index 750...

These numbers are far enough apart that we aren't really taking full use of space locality. If we visit index 1000, we might load in the surrounding 64 bytes or so, but that doesn't help us when we look at index 750 next. This problem is further exacerbated by the random keys we pass into `get(...)`.

The AvlDictionary is still likely a little slower than the SortedArrayDictionary possibly because binary search still can take advantage of space locality to at least a limited extent – once the search narrows down to a small range of numbers, we *can* make use of the cache.

In contrast, objects in Java aren't guaranteed to be located anywhere in particular. They might end up being laid out right next to each other, but it's highly unlikely for that to happen, so it's likely that space locality doesn't help us at all.

3. Code Modeling

Consider the following problems:

(a) What is the simplified tight O bound for the runtime of each of the loops below? What is the simplified tight O bound for the runtime of method1?

```
public void method1(int n) {
    for (int i = 10; i < n; i++) {
        System.out.println("373");
    }

    for (int i = n; i >= 1; i /= 2) {
        System.out.println("is");
    }

    for (int i = 0; i < 9999999; i++) {
        System.out.println("cool");
    }

    for (int i = n; i >= n - 4; i-=1) {
        System.out.println("I guess");
    }
}
```

Solution:

Loop 1: $\mathcal{O}(n)$

Loop 2: $\mathcal{O}(\log n)$

Loop 3: $\mathcal{O}(1)$

Loop 4: $\mathcal{O}(1)$

Overall runtime: $\mathcal{O}(n)$

(b) This time, write out the summation of method2 as well as give the simplified tight oh bound for the runtime of this method in terms of n.

```
public void method2(int n) {
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < i; j++) {
            for (int k = 0; k < i; k++) {
                System.out.println("Hi, my name is " + n);
            }
        }
    }
}
```

Solution:

$$\sum_{i=0}^{n-1} \sum_{j=0}^{i-1} \sum_{k=0}^{i-1} 1$$

Overall runtime: Sum of squares -> $\mathcal{O}(n^3)$

(c) What is the simplified tight oh bound for the runtime of each of the loops below? Ignore the context of the if conditions for now. Finally, give the overall simplified runtime of method3.

```java
public void method3(int n) {
    if (n < 10000) {
        for (int i = 0; i < n * n * n; i++) {
            System.out.println(":0");
        }
    } else if (n == 10001) {
        for (int i = 0; i < n * n * n * n; i++) {
            System.out.println(":/");
        }
    } else {
        for (int i = 0; i < 2 * n; i++) {
            System.out.println(":D");
        }
    }
}
```

Solution:

Loop 1: $\mathcal{O}(1)$ if considering if condition, $\mathcal{O}(n^3)$ otherwise

Loop 2: $\mathcal{O}(1)$ if considering if condition, $\mathcal{O}(n^4)$ otherwise

Loop 3: $\mathcal{O}(n)$

Overall runtime: $\mathcal{O}(n)$

(d) What is the simplified tight oh bound for the runtime of each of the loops above? Ignore the context of the if conditions for now. Finally, give the overall simplified runtime of method4.

```java
public void method4(int n) {
    for (int i = 0; i < n; i++) {
        if (i == 0) {
            for (int j = 0; j < n; j++) {
                System.out.println("lol");
            }
        } else if (n < 1000) {
            for (int j = 0; j < n * n; j++) {
                System.out.println("rofl");
            }
        } else {
            for (int j = 0; j < 10000; j++) {
                System.out.println("haha");
            }
        }
    }
}
```

Solution:

Outer Loop: $\mathcal{O}(n)$

Inner Loop 1: $\mathcal{O}(n)$ but will only run once (no matter how many times the outer loop runs) because of the if-condition.

Inner Loop 2: $\mathcal{O}(1)$ if considering if condition, $\mathcal{O}(n^2)$ otherwise

Inner Loop 3: $\mathcal{O}(1)$

Overall runtime: $\mathcal{O}(n)$

4. Code Modeling 2: Electric Boogaloo

For each scenario, provide a Θ bound for both the **best** and **worst** case runtimes for the function. Make sure your bounds are as tight/simplified as possible. Give your answers in terms of n, the number of strings in `input`.

You may assume that all strings are of constant length, but can make no other assumptions about the input. You should also assume that all iterators are efficient.

(a) Consider the following snippet of Java code:

```
public static IDictionary<String, Integer> countStrings(ISet<String> input) {
    IDictionary<String, Integer> dict = new BSTDictionary<String, Integer>();

    for (String curString : input)
    {
        dict.put(curString, 1 + dict.getOrDefault(curString, 0));
    }

    return dict;
}
```

(i) Describe the runtime of this function in terms of n, the number of strings contained in `input`. Provide bounds for both the **best** and **worst** case runtimes. Note that `BSTDictionary` uses a BST to implement a dictionary.

Solution:

The best case runtime is $\Theta(n \log n)$. The worst case is $\Theta(n^2)$.

In the best case, the data luckily results in a balanced tree. In the worst case, the tree looks like a LinkedList. We do two operations inside the loop, but this only affects the answer by a constant factor.

Note that using an `ISet` means that each member of `input` has to be unique.

(ii) Do the same thing, but assume now that `dict` is of type `AVLDictionary`. This dictionary is implemented using an AVL Tree, just like in the experiments for HW5.

Solution:

The best case runtime is $\Theta(n \log n)$. The worst case is $\Theta(n \log n)$.

Unlike a BST, an AVL Tree is self-balancing. Asymptotically, this makes the worst case the same as the best case.

(iii) Do the same thing, but assume now that `dict` is of type `ArrayDictionary`. This is the data structure you implemented in HW2. You can ignore the time it takes to resize.

Solution:

The best case runtime is $\Theta(n^2)$. The worst case is $\Theta(n^2)$.

Because all of our elements are unique, we need to iterate through the **entire** `ArrayDictionary` for each item in the loop, both for `getOrDefault` as well as `put`. This means that in both cases the runtime is Gauss's Summation multiplied by a constant of 2!

(iv) Do the same thing, but assume now that `dict` is of type `ChainedHashDictionary`. This is the data structure you implemented in HW4. You can ignore the time it takes to resize. You can also assume that `.hashCode()` takes constant time.

Solution:

The best case runtime is $\Theta(n)$. The worst case is $\Theta\left(n^2\right)$.

In the best case, there is a nice bucket distribution, and we achieve constant time insertion and lookup for each member of `input`. This gives us linear runtime overall.

In the worst case, all items accidentally hash to the same bucket, and the runtime is identical to `ArrayDictionary`.

(b) For the same four `IDictionary` implementations, provide best and worst case runtime bounds for **this** code snippet:

```
public static IDictionary<String, Integer> countStrings(IList<String> input) {
    IDictionary<String, Integer> dict = new BSTDictionary<String, Integer>();

    for (String curString : input)
    {
        dict.put(curString, 1 + dict.getOrDefault(curString, 0));
    }

    return dict;
}
```

Hint 1: You should be able to do this problem very quickly by looking at your answers for the previous snippet.
Hint 2: What's different about this function?

(i) **Solution:**

For `BSTDictionary`, the best case runtime is $\Theta(n)$. The worst case is $\Theta\left(n^2\right)$.

The best case for this code snippet is the same regardless of which data structure is used for `dict`. Because `input` is now an `IList`, we allow duplicates. In the best case, `input` contains n identical strings. This means that only one key/value pair is ever stored in `dict`, where the value is constantly updated. This takes constant work per loop iteration.

In the worst case, all members of `input` are still unique, allowing the same worst case runtime that would occur when using `ISet`.

(ii) **Solution:**

For `AVLDictionary`, the best case runtime is $\Theta(n)$. The worst case is $\Theta(n \log n)$.

The best case for this code snippet is the same regardless of which data structure is used for `dict`. Because `input` is now an `IList`, we allow duplicates. In the best case, `input` contains n identical strings. This means that only one key/value pair is ever stored in `dict`, where the value is constantly updated. This takes constant work per loop iteration.

In the worst case, all members of `input` are still unique, allowing the same worst case runtime that

would occur when using `ISet`.

(iii) **Solution:**

For `ArrayDictionary`, the best case runtime is $\Theta(n)$. The worst case is $\Theta(n^2)$.

The best case for this code snippet is the same regardless of which data structure is used for `dict`. Because `input` is now an `IList`, we allow duplicates. In the best case, `input` contains n identical strings. This means that only one key/value pair is ever stored in `dict`, where the value is constantly updated. This takes constant work per loop iteration.

In the worst case, all members of `input` are still unique, allowing the same worst case runtime that would occur when using `ISet`.

(iv) **Solution:**

For `ChainedHashDictionary`, the best case runtime is $\Theta(n)$. The worst case is $\Theta(n^2)$.

The best case for this code snippet is the same regardless of which data structure is used for `dict`. Because `input` is now an `IList`, we allow duplicates. In the best case, `input` contains n identical strings. This means that only one key/value pair is ever stored in `dict`, where the value is constantly updated. This takes constant work per loop iteration.

In the worst case, all members of `input` are still unique, allowing the same worst case runtime that would occur when using `ISet`.

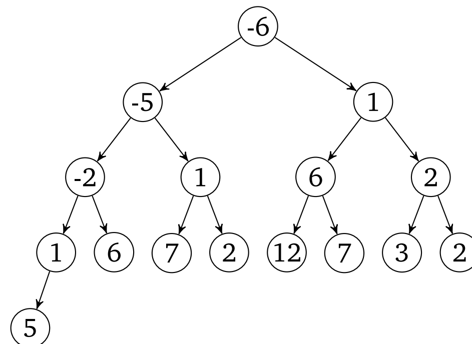
5. Heaps

Consider the following list of numbers:

`[1, 5, 2, -6, 7, 12, 3, -5, 6, 2, 1, 6, 7, 2, 1, -2]`

(a) Insert these numbers into a min 2-heap (into a min-heap with up to two children per node). Show both the final tree and the array representation.

Solution:



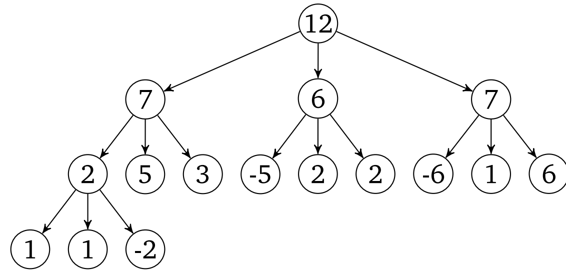
The array representation:

`[-6, -5, 1, -2, 1, 6, 2, 1, 6, 7, 2, 12, 7, 3, 2, 5]`

(b) Insert these numbers into a max 3-heap (a max heap with up to three children per node). Show both the final

tree and the array representation.

Solution:

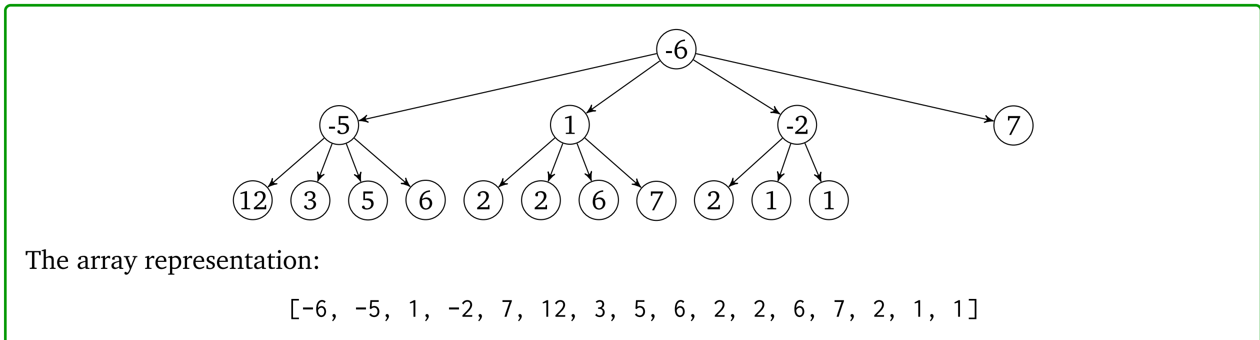


The array representation:

```
[12, 7, 6, 7, 2, 5, 3, -5, 2, 2, -6, 1, 6, 1, 1, -2]
```

(c) Insert these numbers into a min 4-heap using Floyd's `buildHeap` algorithm. Show both the final tree and the array representation.

Solution:

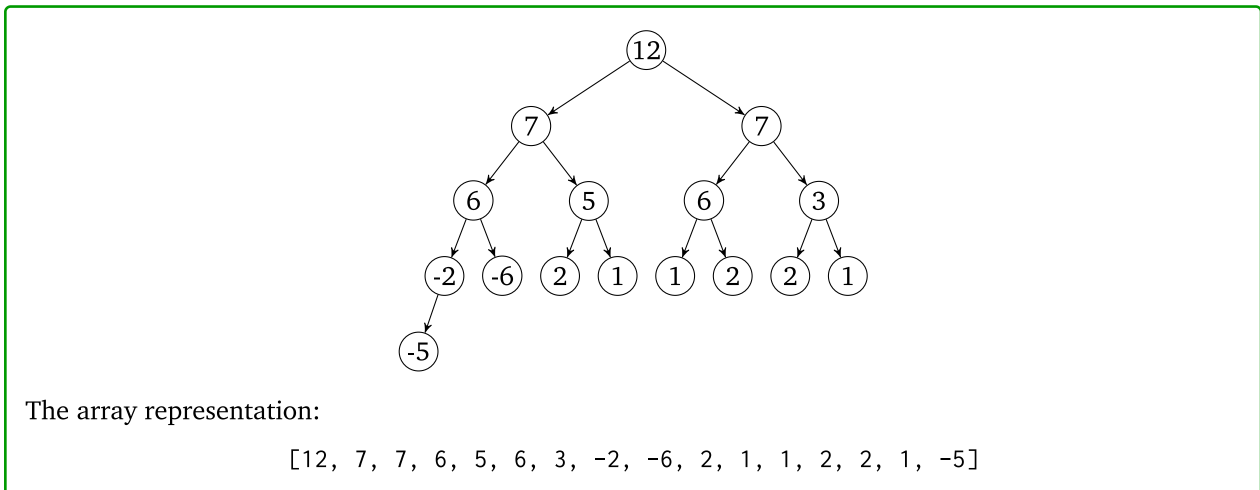


The array representation:

`[-6, -5, 1, -2, 7, 12, 3, 5, 6, 2, 2, 6, 7, 2, 1, 1]`

(d) Insert these numbers into a max 2-heap using Floyd's `buildHeap` algorithm. Show both the final tree and the array representation.

Solution:



The array representation:

`[12, 7, 7, 6, 5, 6, 3, -2, -6, 2, 1, 1, 2, 2, 1, -5]`

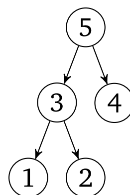
(e) Suppose we modify Floyd's `buildHeap` algorithm so we start from the front of the array, iterate forward, and call `percolateDown(...)` on each element. Why is this a bad idea?

Solution:

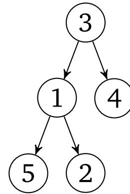
This algorithm would have the net effect of pushing large numbers down, but that doesn't necessarily mean the small numbers will rise.

Specifically, the percolate-down algorithm will visit the root node exactly once, and (potentially) swap it with one of its two children. If we haven't yet visited those two children, there's no guarantee they'll happen to be exactly the min element.

For example, suppose we ran this modified algorithm on the following tree:



After running the proposed algorithm, we would have:



...which is not a valid min-heap.

6. Sorting

(a) During lecture, we focused on five different sorting algorithms: insertion sort, merge sort, quick sort, selection sort, and heap sort.

For each of these five algorithms, state:

- The best and worst-case runtimes
- Whether the sorting algorithm is stable
- Whether the sorting algorithm is in-place
- Whether the sorting algorithm is a general-purpose one, or if there are any restrictions on how it can or should be used.

Solution:

For insertion sort, the best and worst-case runtimes are $\Theta(n)$ and $\Theta(n^2)$ respectively. Insertion sort is stable and in-place (assuming it's implemented correctly). Insertion sort works best when the list is already almost sorted (that's when you can achieve the $\Theta(n)$ running time.

For merge sort, the best and worst-case runtimes are both $\Theta(n \log(n))$. Merge sort is stable, but not in-place – the merge step forces us to output a new array. Merge sort is a general-purpose sort. It is an extremely consistent sort regardless of what is input.

For quick sort, the best and worst-case runtimes are $\Theta(n \log(n))$ and $\Theta(n^2)$ respectively. Quick sort is not stable, but is in-place. Quick sort is *usually* faster than mergesort or heap sort, but can perform badly if the input leads to bad pivot choices.

For selection sort, the best and worst-case runtimes are both $\Theta(n^2)$. Selection sort is unstable and in-place. It is a very simple sort to implement and understand but is not very efficient.

For heap sort, the best and worst-case running times are both $\Theta(n \log n)$. Heap sort is in-place but not stable. It is a very consistent sort and is an excellent choice if you need an in-place sort. The constant factors hidden by the big-Theta notation usually makes it slightly slower than quick sort.

(b) Suppose we want to sort an array containing 50 strings. Which of the above four algorithms is the best choice?

Solution:

Given the small size, it's possible insertion sort will be fast enough to be optimal. It may be worst-case $\Theta(n^2)$, but it also has a low constant factor, which may end up making it good enough in this case.

Using any of merge sort, quick sort, or heap sort instead might also be reasonable choices.

(c) Suppose we have an array containing a few hundred elements that is almost entirely sorted, apart from a one or two elements that were swapped with the previous item in the list. Which of the algorithms is the best way of sorting this data?

Solution:

Here, insertion sort is definitely the best choice – it'll run in $\Theta(n)$ time here.

(d) Suppose we are writing a website named "sort-my-numbers.com" which accepts and sorts numbers uploaded by the user. Keeping in mind that users can be malicious, which of the above algorithms is the best choice?

Solution:

Merge sort or Heap sort are likely the best choice. If we're worried about malicious users overworking our website, we'd want to avoid any algorithms that could potentially have quadratic behavior, which rules out insertion sort and quick sort.

You should choose Heap over merge if you think the extra memory might be an issue (e.g. if you're letting the user input arbitrarily large data sets). On the other hand, you might want Merge sort over heap sort if your users want a stable sort.

(e) Suppose we want to sort an array of ints, but also want to minimize the amount of extra memory we want to use as much as possible. Which of the above algorithms is the best choice?

Solution:

We instead want to use an in-place algorithm. In that case, quicksort is likely the best answer. We could also use heap sort at the cost of (probably) being (slightly) slower if you wanted guaranteed $O(n \log n)$ performance.

(f) Suppose we have a version of quick sort which selects pivots randomly and creates partitions in the manner described in lecture. Explain how you would build an input array that would cause this version of quick sort to always run in $\mathcal{O}(n^2)$ time.

Your answer should explain on a high level what your array would look like and what happens when you try running quick sort on it. You do not need to give a specific example of such a array, though you may if you think it will help make your explanation more clear.

Solution:

One method would be to construct an array containing all identical elements – for example, an array containing only the number 1.

In that case, our pivot strategy doesn't really matter: once we find one, our algorithm will always put every single remaining element into the left-most partition (the ones containing all elements $\leq$ to the pivot) and no elements into the right-most partitionin (the ones containing all elements $>$ than the pivot).

Consequently, we get $\mathcal{O}(n^2)$ behavior.

(g) How can you modify both versions of quicksort so that they no longer display $\mathcal{O}\left(n^2\right)$ behavior given the same inputs?

Solution:

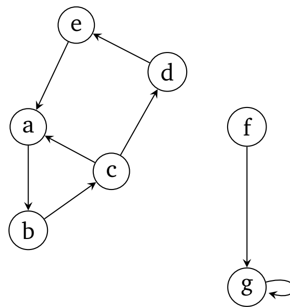
If we want to make both pivot strategies no longer degrade to $\mathcal{O}\left(n^2\right)$ given an array of all duplicates, one strategy might be to partition into three groups, not two.

Previously, we partitioned all elements $\leq$ to the pivot in one group all all elements $>$ into the other; now, we want the partition and get all elements $<$ than the pivot, $=$ to the pivot, and $>$ than the pivot.

In the case of the all-duplicate array, all the elements would fall into the second pivot, and there would be no more work left to do. So our modified algorithm would sort this array in $\mathcal{O}(n)$ time instead of $\mathcal{O}\left(n^2\right)$ time.

7. Graph basics

Consider the following graph:



(a) Draw this graph as an adjacency matrix.

Solution:

Note: to make the solution more readable, we've left any cells that should be false blank. We adopt the convention that the cell located at the i-th row and j-th column is true, there exists an edge from the vertex corresponding to i to the vertex corresponding to j.

	a	b	c	d	e	f	g
a		T					
b			T				
c	T			T			
d					T		
e	T						
f							T
g							T

(b) Draw this graph as an adjacency list.

Solution:

Nodes	List of adjacent nodes
a	b
b	c
c	a, d
d	e
e	a
f	g
g	g

(c) Suppose we run BFS on this list, starting on node a. In what order do we visit each node? Assume we break ties by selecting the node that's alphabetically lower.

Solution:

Answer: a, b, c, d, e

(d) Suppose we run DFS on this list, starting on node a. In order do we visit each node? Assume we break ties in the same way as above.

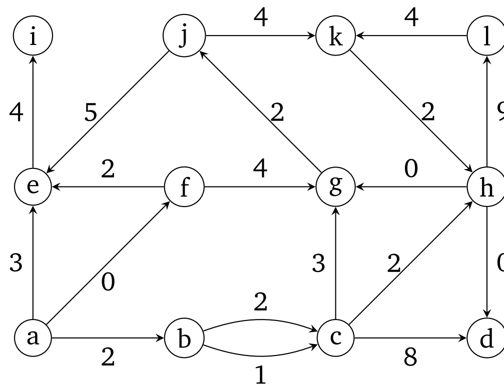
Solution:

Answer: a, b, c, d, e

In retrospect, this graph was not a very good example to showcase the differences between DFS and BFS.

8. Dijkstra's algorithm

(a) Consider the following graph.



(i) Run Dijkstra's algorithm on the following graph starting on node a. List the final costs of each node, the edges selected by Dijkstra's algorithm, and whether or not Dijkstra's algorithm returned the correct result. In the case of ties, select the node that is the smallest alphabetically.

Solution:

Here, we will give the answer just in horizontal tabular format to help save on space.

Vertex	a	b	c	d	e	f	g	h	i	j	k	l
Distance (or cost)	0	2	3	5	2	0	4	5	6	6	10	14
Predecessor	N/A	a	b	h	f	a	f	c	e	g	j	h

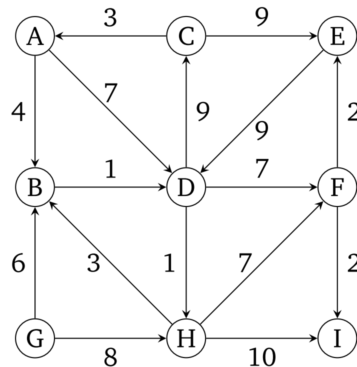
Dijkstra's algorithm returned the correct result. (Dijkstra's algorithm will always return the correct result for graphs that don't have edges with negative weights.)

(ii) In general, is it possible to run Dijkstra's from a single node in a graph in order to recover the shortest path between **any** two pairs of nodes? Why or why not?

Solution:

False. Dijkstra only lets you compute the shortest path from the **specific** start node you select to every other node in the graph.

(b) Consider this following graph.



(i) Run Dijkstra's algorithm on the following graph starting on node G. List the final costs of each node, the edges selected by Dijkstra's algorithm, and whether or not Dijkstra's algorithm returned the correct result. In the case of ties, select the node that is the smallest alphabetically.

Solution:

Vertex	A	B	C	D	E	F	G	H	I
Distance (or cost)	19	6	16	7	16	14	0	8	16
Predecessor	C	G	D	B	F	D	N/A	G	F

Dijkstra's returned the correct result since there are no edges with negative weights.

(ii) When solving this problem, you had a choice when picking the shortest path from G to H. Do ties matter in Dijkstra's? Why or why not? If so, provide another set of shortest paths.

Solution:

No, ties don't matter in Dijkstra's. In a tie, either choice of path has the same cost, so both paths are the "shortest." You can get another set of shortest paths for this problem if you change the predecessor of H to D instead of G. This path would be G $\rightarrow$ B $\rightarrow$ D $\rightarrow$ H = 6 + 1 + 1 = 8, which is the same as G $\rightarrow$ H = 8.

(c) More generally, in any graph, what are some of the reasons why running Dijkstra's might not work correctly?

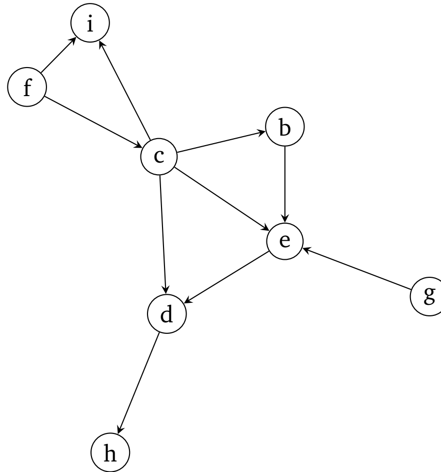
Solution:

If a negative edge exists in the graph, Dijkstra's is not guaranteed to find the correct path. If it does, it's by pure dumb luck. You should use the **Bellman-Ford Algorithm** in this case.

Of course, it's also impossible to compute the shortest path from the start to a goal that can't actually be reached from the start...because that path doesn't exist. The unreachable node just never has its distance updated by the main `while` loop. An example in the first graph is trying to compute a path to any other node starting at node d.

9. Topological sort

(a) List three different topological orderings of the following graph. If no ordering exists, briefly explain why.

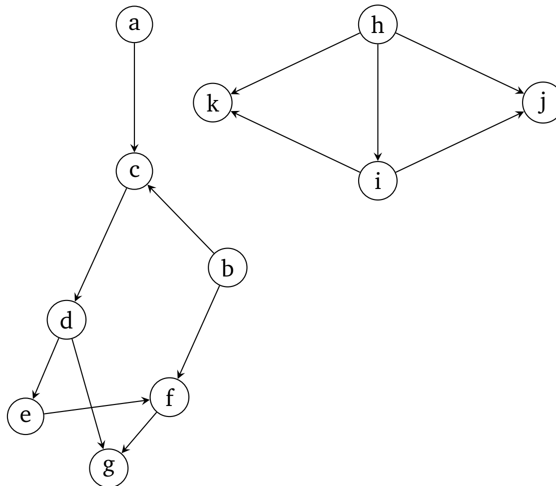


Solution:

Three possible listings:

- g, f, c, i, b, e, d, h
- f, c, i, b, g, e, d, h
- f, g, c, b, e, d, h, i

(b) List three different topological orderings of the following graph. If no ordering exists, briefly explain why.

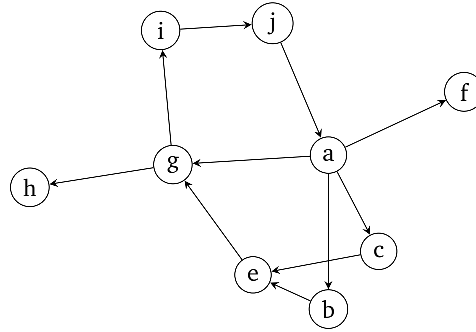


Solution:

Three possible listings:

- h, i, j, k, a, b, c, d, e, f, g
- h, a, b, c, i, d, e, j, f, k, g
- b, h, a, c, i, d, k, e, f, g, j

(c) List three different topological orderings of the following graph. If no ordering exists, briefly explain why.

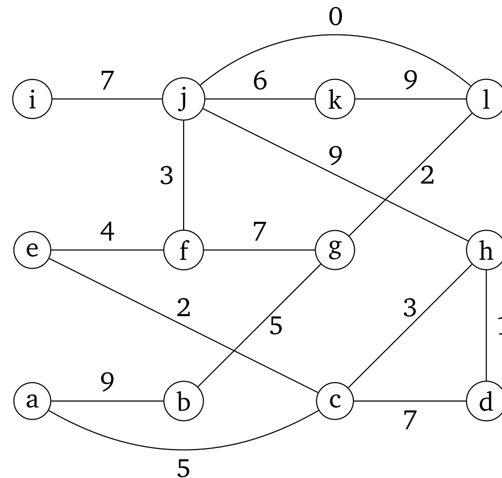


Solution:

There is no valid topological ordering here. The vertices a, g, i, j form a cycle.

10. Minimum spanning trees

Consider the following graph:

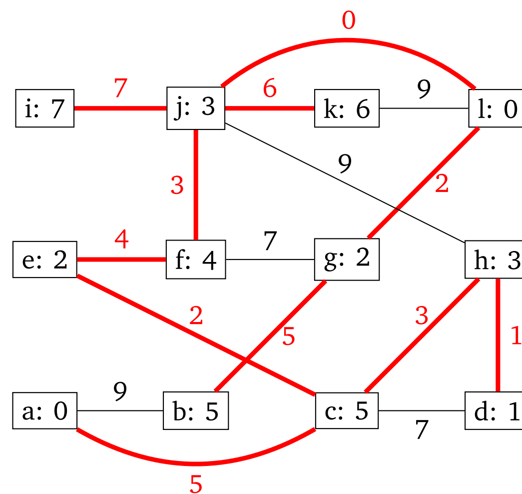


(a) Run Prim's algorithm on the above graph starting on node a to find a minimum spanning tree.

Draw the final MST and the costs per each node. In the case of ties, select the node that is the smallest alphabetically.

Solution:

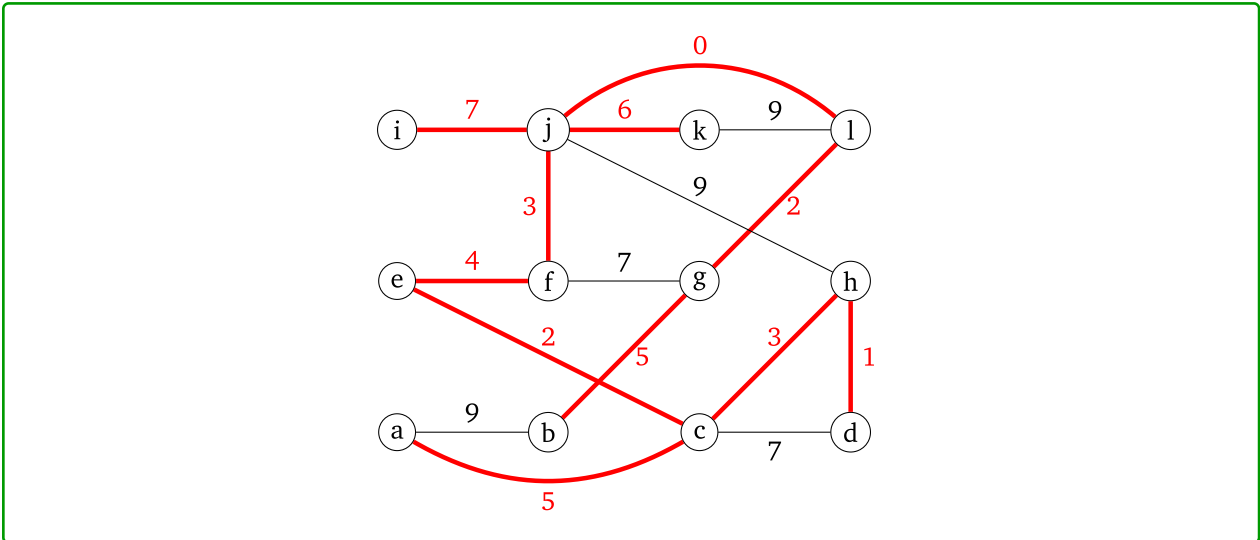
The final MST looks like the following:



(b) Run Kruskal's algorithm on the above graph to find an MST. In the case of ties, select the edge containing the node that is the smallest alphabetically.

Draw the final MST.

Solution:



(c) Suppose we have the following disjoint set. What happens when we run `union(7, 8)`, which uses union-by-rank optimization? Draw both the new trees as well as the array representation of the disjoint set.

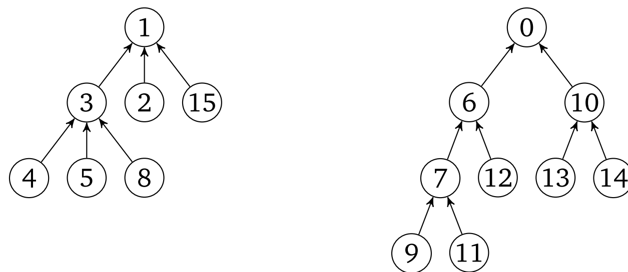
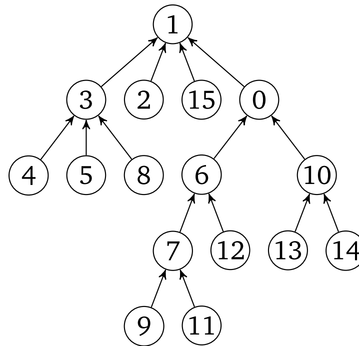


Figure 1: Assume that the rank of the left tree is 4 and the rank of the right tree is 3.

Solution:

Since the left tree has a higher rank, the right tree would get merged in the left one, and the final tree would look like this:



(Your answer may have some of the children arranged in a slightly different order.)

The array representation would then be:

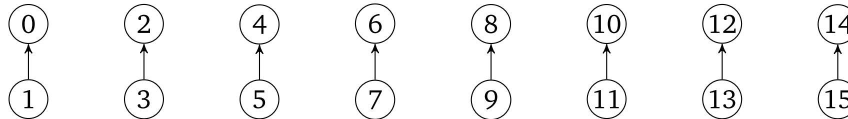
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	-5	1	1	3	3	0	0	1	7	0	7	6	10	10	1

(d) Suppose we have a disjoint set containing 16 elements. Assuming our disjoint set implements the union-by-rank and path-compression algorithm, what is the height of the largest possible internal tree we can construct? Draw what this tree looks like, and what sequence of calls to `union(...)` and `findSet(...)` creates this tree.

Solution:

The trick here is to try and combine trees in such a way that we always trigger ties, forcing the trees to grow by one every time we call `union`. We also take care to make sure to always union in such a way that we avoid triggering the path compression algorithm.

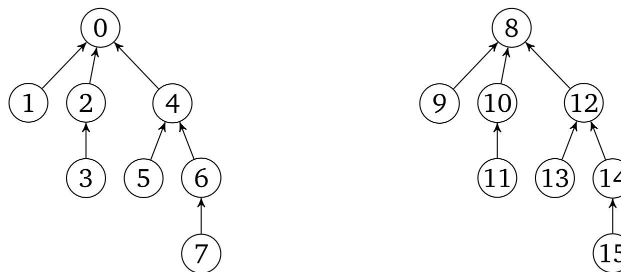
So, we start by calling `makeSet(...)` on the numbers 0 through 15, then call `union(0, 1)`, then `union(2, 3)`, ..., `union(14, 15)`. This produces the following forest:



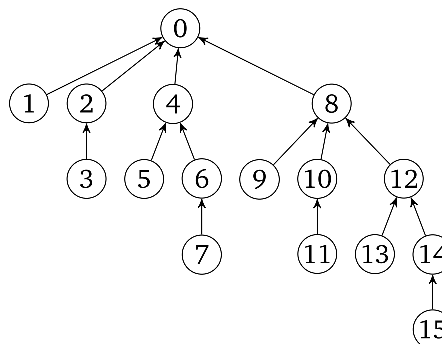
Note that every tree here has a rank of 1. We next call `union(0, 2)`, `union(4, 6)`, `union(8, 10)`, and `union(12, 14)`. Note that we're taking care to only call union on the roots of the trees, to avoid collisions.

Since all the trees have the same rank, we have a tie. Assuming we bias the union algorithm to pick the left one in the case of a tie, we then have the following forest:

We repeat again, calling `union(0, 4)` and `union(8, 12)`. This gives us:



Finally, we call `union(0, 8)`:



This tree ends up having a height of 4, which is the maximum possible height we can obtain using just 16 nodes.

11. Graphs and design

Consider the following problems, which we can all model and solve as a graph problem.

For each problem, describe (i) what your vertices and edges are and (ii) a short (2-3 sentence) description of how to solve the problem.

We will also include more detailed pseudocode to make solutions .

Your description does not need to explain how to implement any of the algorithms we discussed in lecture. However, if you *modify* any of the algorithms we discussed, you must discuss what that modification is.

(a) A popular claim is that if you go to any Wikipedia page and keep clicking on the first link, you will eventually end up at the page about "Philosophy". Suppose you are a given some Wikipedia page as a random starting point. How would you write an algorithm to verify this claim for the given starting point?

Solution:

Here's one possible solution (there are others)

Setup:

Each wikipedia page is a vertex, and every hyperlink is a directed, unweighted edge such that an edge (u, v) means that page u contains a link to page v. Each vertex knows which of its links is the first on the page.

Algorithm description: Initialize an empty hashset. Walk vertex by vertex, going along the edge corresponding to the first link. At each vertex, check if it is "Philosophy" if so return `true` else, check if it is in the hash set, if so return `false`, otherwise add this page to the hash set, and go to the next vertex.

Pseudocode: We store our graph in adjacency list form, where the edges are sorted by the order in which they appear in that corresponding page.

Our algorithm would look roughly like the following:

```
bool everythingGoesToPhilosophy(graph, start):
    encountered = new HashSet()
    encountered.add(start)

    curr = start
    while curr.title != 'Philosophy':
        curr = graph.getFirstLink(curr)
        if curr in encountered:
            return False
        encountered.add(curr)

    return true
```

(b) Suppose you have a bunch of computers networked together (haphazardly) using wires. You want to send a message to every other computer as fast as possible. Unfortunately, some wires are being monitored by some shadowy organization that wants to intercept your messages.

After doing some reconnaissance, you were able to assign each wire a "risk factor" indicating the likelihood that the wire is being monitored. For example, if a wire has a risk factor of zero, it is extremely unlikely to be monitored; if a wire has a risk factor of 10, it is more likely to be monitored. The smallest possible risk factor is 0; there is no largest possible risk factor.

Implement an algorithm that selects wires to send your message such that (a) every computer receives the message and (b) you minimize the total risk factor. The total risk factor is defined as the sum of the risks of all of the wires you use.

Solution:

This problem basically boils down to finding the MST of the graph.

Setup: We make each computer a node and each wire (with the risk factor) a weighted, undirected edge.

Algorithm: Once we form the graph, we can use either Prim's or Kruskal's algorithm as we implemented them in lecture, with no further modifications.

(c) Explain how you would implement an algorithm that uses your predictions to find any computers where sending a message would force you to transmit a message over a wire with a risk factor of k or higher.

Solution:

Setup: We have the same graph as the last part.

Algorithm: Run either DFS or BFS on the graph, but modify it so we no longer traverse down edges that have a risk factor of k or higher. We then return all vertices we were unable to visit.

Pseudocode:

```
Set<Computer> getAllUnreachable(graph, start, k):
    unreachable = copy(graph.vertices)

    stack = new Stack()
    stack.push(start)

    while stack is not empty:
        curr = stack.pop()
        unreachable.remove(curr)

        for edge in graph.getNeighbors(start):
            if edge.dest not in unreachable:
                skip iteration (already visited)

            if edge.weight >= k:
                skip iteration (risk factor too high)

            stack.push(edge.dest)

    return unreachable
```

(d) Suppose you have a graph containing $|V|$ nodes. What is the maximum number of edges you can add while ensuring the graph is always a DAG? Assume you are not permitted to add parallel edges.

Solution:

The first node can have an edge pointing to $|V| - 1$ edges, the second node can have edges pointing to the remaining $|V| - 2$ edges, and so forth.

(Visually, pretend the nodes are in a line. To prevent cycles, each node is allowed to point to any node on the right, but not any node on the left).

We can express this as a summation:

$$\sum_{i=0}^{|V|-1} i$$

By Gauss's identity, we know this is equivalent to $\frac{|V|(|V|-1)}{2}$.

(e) Suppose you were walking in a field and unexpectedly ran into an alien. The alien, startled by your presence, dropped a book, ran into their UFO, and flew off.

This book ended up being a dictionary for the alien language – e.g. a book containing a bunch of alien words, with corresponding alien definitions.

You observe that the alien's language appears to be character based. Naturally, the first and burning question you have is what the alphabetical order of these alien characters are.

For example, in English, the character "a" comes before "b". In the alien language, does the character "$\rightmoon$" come come before or after character "ϱ"? The world must know.

Assuming the dictionary is sorted by the alien character ordering, design an algorithm that prints out all plausible alphabetical orderings of the alien characters.

Solution:

Initially, a naive solution might be to take the first characters of each alien word and print out the characters in that order. However, what if there are certain alien words that never start with a particular character? E.g. what if there are no words that start with ϱ?

In this case, a more sophisticated solution is needed.

One way we can do this is to represent each character as a vertex and add an edge whenever we deduce information about which character comes after the next.

For example, if we were looking at an English dictionary and saw the words:

- apology
- apple
- banana

...we'd know the following facts:

(i) The character 'a' comes before the character 'b'

(ii) The character 'o' comes before the character 'p'.

We can add a directed edge for both.

Once we do, we have a DAG. We can then traverse this graph and print out all possible topological orderings.

This problem is admittedly much harder than anything that might show up on the final, so we'll omit the pseudocode.

12. P and NP

Consider the following decision problem:

`ODD-CYCLE`: Given some input graph G, does it contain a cycle of odd length?

You want to show this decision problem is in NP.

(a) What is a convincing certificate a solver could return for this problem? Remember a certificate is a short way of proving that the G really has a cycle of odd length.

Solution:

One convincing certificate might be the list of vertices that make up the odd-length cycle, listed in the order we would encounter them in the graph.

(b) Describe, in pseudocode, how you would check that the certificate is a valid one. Remember that a verifier takes in a supposed certificate and checks (in polynomial time) that the certificate is valid.

Solution:

The core idea is to loop through the list of vertices and double-check and make sure that (a) it's actually odd-length and that (b) it does actually form a cycle.

Of course, the devil is in the details:

```
bool verifyOddCycle(graph, cycle):
    if cycle.length is even:
        return false

    for i from 0 to cycle.length:
        prev, next = cycle[i], cycle[(i + 1)] % cycle.length]
        if next not in graph.getNeighborsOf(prev):
            return false

    return true
```

(c) What is the worst-case runtime of your verifier?

Solution:

The runtime of this algorithm depends on how long it takes to (a) get the neighbors of some node and (b) check and see if `next` is contained within that list.

If we assume that both operations take $\mathcal{O}(1)$ time (e.g. perhaps the graph is some sort of adjacency list, where we store the neighbors in a hashset), the total runtime would be $\mathcal{O}(n)$, where n is the length of the `cycle` list.

The length of the cycle, however, is upper-bounded by $|V|$, so we could also say that the worst-case runtime is $\mathcal{O}(|V|)$.

(d) Do you think it's likely this problem also happens to be in P? Why or why not?

Solution:

It turns out that the answer is yes (though we wouldn't expect you to find us this algorithm under exam conditions). It turns out a graph can be 2-colored if and only if it doesn't have an odd-cycle. So we just try to two color it. If we get stuck, we can backtrack to find the cycle. If we succeed there isn't an odd-cycle to find.

See `https://en.wikipedia.org/wiki/Bipartite_graph`. This algorithm is another application of BFS.

13. Short answer

For each of the following questions, answer "true", "false", or "unknown" and justify your response. Your justification should be short – at most 2 or 3 sentences.

(a) If we implement Kruskal's algorithm using a general-purpose sort, Kruskal's algorithm will run in nearly-constant time.

Solution:

False. Kruskal's algorithm has three stages: we (a) initialize the disjoint set, (b) sort the edges, then (c) find the MST. If we use a general-purpose sorting algorithm, sorting the edges will take $\mathcal{O}(|E| \log(|E|))$ time. This makes Kruskal's algorithm run in worst-case $\mathcal{O}(|E| \log(|E|))$ time, no matter how fast steps (a) and (c) are.

(b) It is possible to reduce all problems in P to some problem in NP.

Solution:

Technically, yes. All problems in P are also in NP, so by definition, any arbitrary problem in P already has been reduced to some problem in NP – itself.

An alternative explanation is that every problem in P can be reduced to (any) NP-complete problem.

(c) Dijkstra's algorithm will always return the incorrect result if the graph contains negative-length edges.

Solution:

False. For example, consider a graph containing two vertices with one edge of negative weight connecting the two. If we run Dijkstra's algorithm, we will definitely get back the correct result.

(Dijkstra's algorithm is not guaranteed to do the right thing if there are negative-cost edges, but sometimes we can get lucky.)

(d) Suppose we want to find a MST of some arbitrary graph. If we run Prim's algorithm on any arbitrary node, we will always get back the same result.

Solution:

False. Suppose we have a graph containing two vertices connected by two parallel edges of the same weight. When we try finding an MST, Prim's algorithm could end up returning one of two solutions, depending on how we break ties.

That said, if we knew that every edge in the graph had a unique weight, then Prim's will indeed be guaranteed to return a unique MST regardless of starting point or other factors.

(e) $\mathcal{O}\left(n^2 \log(3) + 4\right) = \mathcal{O}\left(4n + n^2\right)$

Solution:

True. Both $\mathcal{O}(\ldots)$ families on the left and the right dominate exactly the same functions (the set or family of all functions dominated by n^2). So, the two must be exactly equivalent.

(f) There is an efficient way of solving the `3-SAT` decision problem.

Solution:

Unknown. Since `3-SAT` is an NP-COMPLETE problem, that means that if we had an efficient (i.e. polynomial) solver for `3-SAT`, we'd also have an efficient solver for all problems in NP, which would imply P = NP. However, whether P = NP is currently an open question.

(g) Iterating over a list using the iterator is always faster than iterating by repeatedly calling the `get(...)` method.

Solution:

False. Suppose the list is an arraylist.

(h) We can always sort some list of length n in $\Theta(n)$ time.

Solution:

False. A general-purpose sort, in the worst case, can do no better than $\Theta(n \log(n))$. We can only achieve worst-case $\Theta(n)$ runtimes if we can exploit some property of the list being sorted (which would mean using a non-general sorting algorithm).

(i) In a simple graph, if there are $|E|$ edges, the maximum number of possible vertices is $|V| \in \mathcal{O}\left(|E|^2\right)$.

Solution:

Technically true. If there are some fixed $|E|$ number of edges, the way we can maximize the number of vertices is to give each edge two unique vertices. That would mean that the maximum number of vertices would be $2|E|$. So, $|V| \in \mathcal{O}(|E|)$.

And if $|V| \in \mathcal{O}(|E|)$, that also means $|V| \in \mathcal{O}\left(|E|^2\right)$. It is, however false that $|V| \in \Theta(|E|)$.

(j) Consider the following question:

SHORT-PATH

input: Graph G with non-negative edge weights, vertices u, v and a number k

output: YES if there is a u, v path of length at most k and no otherwise.

Is SHORT-PATH in NP? In P?

Solution:

Both are true. SHORT-PATH is in P because you can solve it with Dijkstra's algorithm; all problems in P are also in NP.

(k) The `.get(...)` method of hash tables has a worst-case runtime of $\mathcal{O}(n)$, where n is the number of key-value pairs.

Solution:

True. Even with resizing, prime table sizes, etc, we could still get profoundly unlucky and have all elements hash and collide to the same spot, giving us a worst-case runtime of $\mathcal{O}(|n|)$.

(l) A hash table implemented using open addressing is likely to have suboptimal performance when $\lambda > 0.5$.

Solution:

True. When $\lambda > \frac{1}{2}$, it's likely that we'll end up colliding and have to probe multiple times.

(m) The `peekMin(...)` method in heaps has a worst-case runtime of $\mathcal{O}(\log(n))$.

Solution:

True. The `peekMin(...)` has a worst-case runtime of $\mathcal{O}(1)$. And if some function is upper-bounded by a constant, it certainly must also be upper bounded by the log function.

(n) If a problem is in NP, that means it must take exponential time to solve.

Solution:

False. Remember every decision problem in P is also in NP (by definition) so e.g. SHORT-PATH from a few questions above is in NP but doesn't require exponential time to solve.

(o) For any given graph, there exists exactly one unique MST.

Solution:

False. Consider again the example with two vertices connected by two parallel edges of the same weight.

14. Debugging

Suppose we are in the process of implementing a hash map that uses open addressing and quadratic probing and want to implement the `delete` method.

(a) Consider the following implementation of `delete`. List every bug you can find.

Note: You can assume that the given code compiles. Focus on finding run-time bugs, not compile-time bugs.

```
1     public class QuadraticProbingHashTable<K, V> {
2         private Pair<K, V>[] array;
3         private int size;
4
5         private static class Pair<K, V> {
6             public K key;
7             public V value;
8         }
9
10        // Other methods are omitted, but functional.
11
12        /**
13         * Deletes the key-value pair associated with the key, and
14         * returns the old value.
15         *
16         * @throws NoSuchKeyException if the key-value pair does not exist in the method.
17         */
18        public V delete(K key) {
19            int index = key.hashCode() % this.array.length;
20
21            int i = 0;
22            while (this.array[index] != null && !this.array[index].key.equals(key)) {
23                i += 1;
24                index = (index + i * i) % this.array.length;
25            }
26
27            if (this.array[index] == null) {
28                throw new NoSuchKeyException("Key-value pair not in dictionary");
29            }
30
31            this.array[index] = null;
32
33            return this.array[index].value;
34        }
35    }
```

Solution:

The full list of all bugs:

(i) If the dictionary contains any null keys, this code will crash. (See the call to `.equals(...)` in the while loop condition.)

(ii) If the key parameter is null, the code will crash. (See the call to `.hashCode(...)` at the top of the method.)

(iii) If the key's hashCode is negative, this code will crash. (We try indexing a negative element).

(iv) We probe the array incorrectly. If s is the initial position we check, we ought to be checking s, $s+1$, $s+4$, $s+9$, $s+16$...

Instead, we check s, $s+1$, $s+5$, $s+14$, $s+30$...

(v) Nulling out the array index will break all subsequent deletes. Suppose we have a collision, and our algorithm ends up checking index locations 0, 1, 5, 14, and 30 respectively.

If we null out index 5, then all subsequent probes starting at index 0 will be unable to find whatever's located at 14 or 30.

(vi) The final return has a null pointer exception – we null out that pair before fetching the value.

(b) Let's suppose the `Pair` array has the following elements (pretend the array fit on one line):

["lily", V_2]	["castle", V_6]	["resource", V_1]	["hard", V_9]	["bathtub", V_0]
["wage", V_4]	["refund", V_7]	["satisfied", V_6]	["spring", V_8]	["spill", V_3]

And, that the following keys have the following hash codes:

Key	Hash Code
"bathtub"	9744
"resource"	4452
"lily"	7410
"spill"	2269
"wage"	8714
"castle"	2900
"satisfied"	9251
"refund"	8105
"spring"	6494
"hard"	9821

What happens when we call `delete` with the following inputs? Be sure write out the resultant array, and to do these method calls *in order*. (**Note**: If a call results in an infinite loop or an error, explain what happened, but don't change the array contents for the next question.)

(i) `delete("lily")`

Solution:

Nothing bad happens:

`null`	["castle", V_6]	["resource", V_1]	["hard", V_9]	["bathtub", V_0]
["wage", V_4]	["refund", V_7]	["satisfied", V_6]	["spring", V_8]	["spill", V_3]

(ii) `delete("spring")`

Solution:

We don't probe correctly in general (see bug above), but we *do* happen to loop around eventually to remove "spring". This code is inefficient, and won't always work out like this, but in this case, we managed a successful delete:

`null`	["castle", V_6]	["resource", V_1]	["hard", V_9]	["bathtub", V_0]
["wage", V_4]	["refund", V_7]	["satisfied", V_6]	`null`	["spill", V_3]

(iii) `delete("castle")`

Solution:

We stop after we see `array[0]` is null, and we throw a `NoSuchKeyException`, without continuing to probe.

(iv) `delete("bananas")`

Solution:

`NoSuchKeyException`, but that's what we wanted, so no bugs caught.

(v) `delete(null)`

Solution:

`NullPointerException`. Note, this is *not* what we wanted, because `Pairs` support null keys. This code should have returned a `NoSuchKeyException`.

(c) List four different test cases you would write to test this method. For each test case, be sure to either describe or draw out what the table's internal fields look like, as well as the expected outcome (assuming the `delete` method was implemented correctly). **Hint**: You may use the inputs previously given to help you identify tests, but it's up to you to describe what kind of input they are testing generally.

Solution:

Some test cases include:

- Picking a key not present in the dictionary. This should trigger an exception (and not change the size).
- Picking a key present in the dictionary. This should succeed, and return the old value (and decrease the size by 1).
- Inserting and attempting to delete a null key. This should succeed (and decrease the size by 1).
- Deleting a key that forces us to probe a few times. This should succeed (and decrease the size, etc).
- Deleting a key in the middle of some probe sequence. All subsequent calls to delete/get/etc should correctly.
- Using a key with a negative hashcode should behave as expected.

Identities

Splitting a sum

$$\sum_{i=a}^{b}(x+y) = \sum_{i=a}^{b} x + \sum_{i=a}^{b} y$$

Factoring out a constant

$$\sum_{i=a}^{b} cf(i) = c\sum_{i=a}^{b} f(i)$$

Change-of-base identity

$$\log_a(n) = \frac{\log_b(n)}{\log_a(b)}$$

Gauss's identity

$$\sum_{i=0}^{n-1} i = 0 + 1 + \ldots + n - 1 = \frac{n(n-1)}{2}$$

Finite geometric series

$$\sum_{i=0}^{n-1} r^i = \frac{r^n - 1}{r - 1}$$

Master theorem

Given a recurrence of the form:

$$T(n) = \begin{cases} d & \text{if } n = 1 \\ aT\left(\frac{n}{b}\right) + n^c & \text{otherwise} \end{cases}$$

We know that:

- If $\log_b(a) < c$ then $T(n) \in \Theta\left(n^c\right)$
- If $\log_b(a) = c$ then $T(n) \in \Theta\left(n^c \log(n)\right)$
- If $\log_b(a) > c$ then $T(n) \in \Theta\left(n^{\log_b(a)}\right)$