Section 10: Solutions

1. Asymptotic analysis

Consider the following equations:

- $f(n) = 600n + n\log_5(n)$
- $g(n) = 30n^2$
- $h(n) = 5n\log_3(n)$
- $i(n) = 4\log_4(n)$

Use these equations and show the following. Note: you may need to use the change-of-base identity for some of these problems. (See the list of identities located at the end of this handout).

(a) Show that $f(n) \in \mathcal{O}(g(n))$.

Solution:

We wish to show that $f(n) \in \mathcal{O}(g(n))$ – that is, there exists some c and n_0 such that $600n + n\log_5(n) \leq c(30n^2)$ for all values of $n \geq n_0$.

We observe that the following chain of inequalities are true:

$$\begin{aligned} 600n + n\log_5(n) &\leq 600n^2 + n\log_5(n) && \text{for all } n \geq 1 \\ 600n^2 = n\log_5(n) &\leq 600n^2 + n^2 && \text{for all } n \geq 5 \\ 600n^2 + n^2 &\leq 601n^2 \\ 601n^2 &\leq c30n^2 && \text{when } c = \frac{601}{30} \end{aligned}$$

Therefore, we know one possible choice for c and n_0 is $c = \frac{601}{30}$ and $n_0 = 5$.

(b) Show that $h(n) \in \Omega(i(n))$.

Solution:

We wish to show that $h(n) \in \Omega(i(n))$ – that is, that there exists some c and n_0 such that $5n\log_3(n) \geq c4\log_4(n)$ for all $n \geq n_0$.

We observe that the following chain of inequalities are true:

$$\begin{aligned} 5\log_3(n) &\geq 4\log_3(n) \\ 4\log_3(n) &\geq 4\frac{\log_3(n)}{\log_4(3)} && \text{By change of base identity} \\ 4\frac{\log_3(n)}{\log_4(3)} &\geq c4\log_4(n) && \text{When } c = \frac{1}{\log_4(3)} \end{aligned}$$

Therefore, we know one possible choice for c and n_0 is $c = \frac{1}{\log_4(3)}$ and $n_0 = 5$.

(c) Show that $h(n) + i(n) \in \Omega\left(f(n) + i(n)\right)$.

Solution:

We wish to show that $h(n) + i(n) \in \Omega\left(f(n) + i(n)\right)$ – that is, that there exists some c and n_0 such that $5n\log_3(n) + 4\log_4(n) \geq c(600n + n\log_5(n) + 4\log_4(n))$.

We observe that the following chain of inequalities are true:

$$\begin{aligned} 5n\log_3(n) + 4\log_4(n) &\geq 5n\log_3(n) && \text{For } n \geq 4 \\ 5n\log_3(n) &\geq \frac{5}{\log_3(5)} n\log_5(n) && \text{By the change-of-base identity} \end{aligned}$$

Next, we observe that the following chain of inequalities are also true:

$$\begin{aligned} c(600n + n\log_5(n) + 4\log_4(n)) &\leq c(600n\log_5(n) + n\log_5(n) + 4\log_4(n)) && \text{For } n \geq 5 \\ &\leq c(601n\log_5(n) + 4\log_4(n)) \\ &\leq c\left(601n\log_5(n) + \frac{4}{\log_4(5)}\log_5(n)\right) && \text{By the change-of-base identity} \\ &\leq c\left(601 + \frac{4}{\log_4(5)}\right) n\log_5(n) \end{aligned}$$

(To help save on space, we've omitted copying the right side of the previous inequality to the left side of the next line)

So, we have shown that $h(n) + i(n) = 5n\log_3(n) + 4\log_4() \geq \frac{5}{\log_3(5)} n\log_5(n)$ for all $n \geq 4$.

We have also shown that $c\left(601 + \frac{4}{\log_4(5)}\right) \geq c\left(600n + n\log_5(n) + 4\log_4(n)\right) = f(n) + i(n)$ for all $n \geq 5$.

Note that $\frac{5}{\log_3(5)} n\log_5(n) \geq c\left(601 + \frac{4}{\log_4(5)}\right) n\log_5(n)$ when $c = \frac{5}{\log_3(5)(601 + 4/\log_4(5))}$.

So we conclude $h(n) + i(n) \geq c(f(n) + i(n))$ for all $n \geq n_0$ when $c = \frac{5}{\log_3(5)(601 + 4/\log_4(5))}$ and $n_0 = 5$.

(d) Show that $f(n) \in \Theta\left(h(n)\right)$.

Solution:

To show that $f(n) \in \Theta\left(h(n)\right)$, we must show that both $f(n) \in \mathcal{O}\left(h(n)\right)$ and $f(n) \in \Omega\left(h(n)\right)$ are true.

Step 1: Showing $f(n) \in \mathcal{O}\left(h(n)\right)$.

We wish to show that there exists a c and n_0 such that $600n + n\log_5(n) \leq c5n\log_3(n)$ for all $n \geq n_0$.

Note that the following chain of inequalities are true:

$$\begin{aligned} 600n + n\log_5(n) &\leq 600n\log_5(n) + n\log_5(n) && \text{For all } n \geq 5 \\ 600n\log_5(n) + n\log_5(n) &\leq 601n\log_5(n) \\ 601n\log_5(n) &\leq c5n\log_3(n) && \text{When } c = \frac{601\log_3(5)}{5} \end{aligned}$$

So, we know $600n + n\log_5(n) \leq c5n\log_3(n)$ for all $n \geq n_0$ when $c = \frac{601\log_3(5)}{5}$ and $n_0 = 5$.

Step 2: Showing $f(n) \in \Omega\left(h(n)\right)$.

We wish to show that there exists a (possibly different) c and n_0 such that $600n + n\log_5(n) \geq c5n\log_3(n)$ for all $n \geq n_0$.

Note that the following chain of inequalities are true:

$$600n + n\log_5(n) \geq n\log_5(n) \qquad \text{For } n \geq 0$$

$$n\log_5(n) \geq c5n\log_3(n) \qquad \text{When } c = \frac{\log_3(5)}{5}$$

So, we know $600n + n\log_5(n) \geq c5n\log_3(n)$ for all $n \geq n_0$ when $c = \frac{\log_3(5)}{5}$ and $n_0 = 0$.

Step 3: Conclusion.

Since we have shown that both $f(n) \in \mathcal{O}(h(n))$ and $f(n) \in \Omega(h(n))$ are true, we know that by definition $f(n) \in \Theta(h(n))$ is also true.

2. Finding closed forms

For each of the following, find a closed form using the tree method and a big-Theta bound using the master method. If you cannot use the master method on a particular recurrence, briefly explain why.

See the last page of this handout for a list of identities.

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

Solution:

We know there are $\log_3(x)$ recursive levels; each i-th level does $3^i \cdot 3\left(\frac{x}{3^i}\right)^2$ work.

So we have:

$$\left(\sum_{i=0}^{\log_3(x)-1} 3^i \cdot 3\left(\frac{x}{3^i}\right)^2\right) + 3^{\log_3(x)}$$

...which simplifies to:

$$3x\sum_{i=0}^{\log_3(x)-1}\left(\frac{3}{9}\right)^i + 3^{\log_3(x)}$$

...which, by the finite geometric series identity is:

$$3x\frac{\left(\frac{3}{9}\right)^{\log_3(x)} - 1}{\frac{3}{9} - 1} + 3^{\log_3(x)}$$

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

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

Solution:

We know there are $\log_3(x)$ recursive levels; each i-th level does $8^i \cdot 2\left(\frac{x}{3^i}\right)$ work.

So we have:

$$\left(\sum_{i=0}^{\log_3(x)-1} 8^i \cdot 2\frac{x}{3^i}\right) + 8^{\log_3(x)}$$

...which simplifies to:

$$2x \sum_{i=0}^{\log_3(x)-1} \left(\frac{8}{3}\right)^i + 8^{\log_3(x)}$$

...which, by the finite geometric series identity is:

$$2x\frac{\left(\frac{8}{3}\right)^{\log_3(x)} - 1}{\frac{8}{3} - 1} + 8^{\log_3(x)}$$

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

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

Solution:

We know there are $\log_5(x)$ recursive levels; each i-th level does $4^i \cdot \left(\left(\frac{x}{5^i}\right)^3 + 2\right)$ work.

So we have:

$$\left(\sum_{i=0}^{\log_5(x)-1} 4^i \cdot \left(\left(\frac{x}{5^i}\right)^3 + 2\right)\right) + 4^{\log_5(x)}$$

...which, via the "splitting a sum" identity is:

$$\sum_{i=0}^{\log_5(x)-1} 4^i \left(\frac{x}{5^i}\right)^3 + \sum_{i=0}^{\log_5(x)-1} 4^i \cdot 2 + 4^{\log_5(x)}$$

We simplify both summations to get:

$$x \sum_{i=0}^{\log_5(x)-1} \left(\frac{4}{125}\right)^i 2 \sum_{i=0}^{\log_5(x)-1} 4^i + 4^{\log_5(x)}$$

...which, by the finite geometric series identity is:

$$x\frac{\left(\frac{4}{125}\right)^{\log_5(x)} - 1}{\frac{4}{125} - 1} + 2\frac{4^{\log_5(x)} - 1}{4 - 1} + 4^{\log_5(x)}$$

Strictly speaking, the master theorem doesn't apply here since the recurrence isn't exactly of the form n^c.

However, what we can do is observe that the recurrence $C(x) = C_1(x) + C_2(x)$, where:

$$C_1(x) = \begin{cases} 1 & \text{if } n = 1 \\ 4C_1(x/5) + n^2 \end{cases}$$

$$C_2(x) = \begin{cases} 0 & \text{if } n = 1 \\ 4C_2(x/5) + 2 \end{cases}$$

If we apply the master theorem on both of these recurrences, we get $C_1(x) \in \Theta\left(x^2\right)$ and $C_2(x) \in \Theta\left(x^{\log_5(4)}\right)$ respectively.

So, we conclude $C(x) \in \Theta\left(x^2\right)$.

Note: it also would have been acceptable to observe that the $+2$ is likely irrelevant when computing the big-Theta and just ignore altogether when applying the master theorem. This trick we did above where we split the recurrence into two is honestly pretty excessive – we did it mainly to show how we could apply the master theorem without handwaving.

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

Solution:

We know there are $n - 1$ recursive levels; each i-th level does $3^i \cdot 2$ work.

So we have:

$$\left(\sum_{i=0}^{n-1} 3^i \cdot 2\right) + 3^{n-1}$$

...which simplifies to:

$$2\sum_{i=0}^{n-1} 3^i + 3^{n-1}$$

...which, by the finite geometric series identity is:

$$2\frac{3^n - 1}{3 - 1} + 3^{n-1}$$

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

3. 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:

(a) 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.

(b) Using an array will likely improve cache 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, cache 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) Recall that during your midterm, you were asked to consider the time needed to iterate over the key-value pairs of a `SortedArrayDictionary` vs an `AvlDictionary`. It turned out that it iterating over the `SortedDictionary` is nearly 10 times faster then iterating over the `AvlDictionary`. You were then asked to discuss 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 then iterating over the AvlDictionary primarily because the iterator for SortedArrayDictionary was able to take full advantage of cache 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 the cache. 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 then the SortedArrayDictionary possibly because binary

search still can take advantage of cache 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 cache locality doesn't help us at all.

(c) Suppose that we are trying to store an absurd amount of data inside of a dictionary: too large to fit in memory. We discussed in lecture how we might modify tree-based dictionaries to store a large amount of data (B-trees).

How would you modify hash tables to successfully store large amounts of data?

Solution:

The core insight here is that the time it takes to access and load a page into RAM is far far more expensive then the time it takes to actually read everything in that page.

It's in fact often a fair assumption to make that iterating through a page is effectively constant when compared to the time needed to load a page – iterating through and reading an entire page is then effectively "free".

So, what we want to do is to make each page-load "count": make sure each page is stuffed with as much useful information as possible.

One way we could do this is to modify our chaining hash table implementation so that...

(a) Each "chain" or "bucket" is deliberately designed so that they occupy up some multiple of the page size

(b) Rather then computing λ based on the number of elements in the hash table, we compute it based on the number of *pages* used by all the chains in the table.

The net effect is that instead of having each chain only contain a small handful of elements, they instead are now much bigger, and contain enough elements that each chain ends up using almost a full page (or maybe two pages) on average.

So now, when we access a chain, the subsequent page load will drag in more elements then before. We can then iterate through and check everything in the chain effectively for "free", based on the "checking the page is significantly faster then loading it" assumption we made earlier.

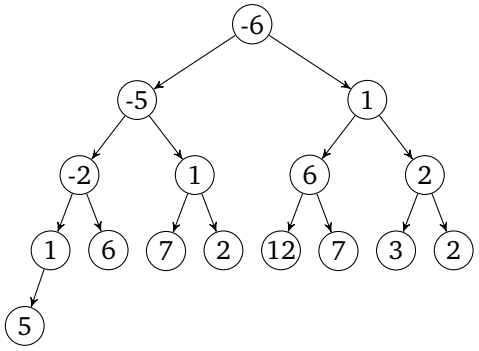
4. Heaps

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:

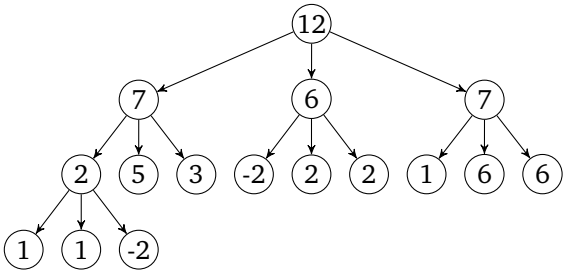


The array representation:

```
[-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 (nto a max-heap with up to three children per node). Show both the final tree and the array representation.

Solution:

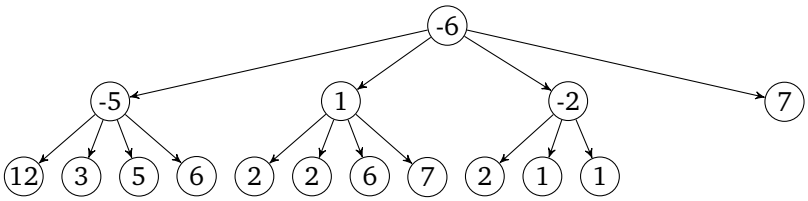


The array representation:

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

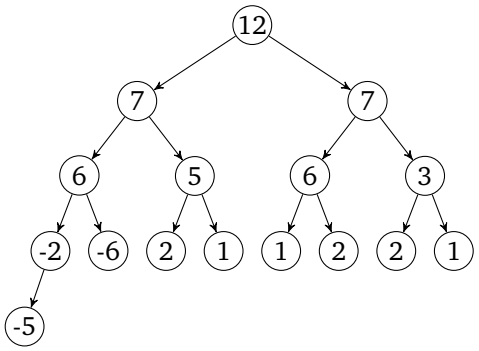


The array representation:

```
[-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]
```

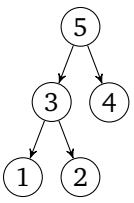
(e) Suppose we modify Floyd's `buildHeap` algorithm so we start from the front of the array, iterate forward, and call `percolateDown(...)` on each element. Why is this a bad idea?

Solution:

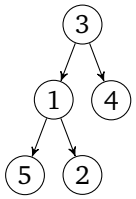
This algorithm would have the net effect of pushing large numbers down, but that doesn't necessarily mean the small numbers will rise.

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.

5. Sorting

(a) During lecture, we focused on four different sorting algorithms: insertion sort, merge sort, quick sort, and counting sort.

For each of these four 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 be used.

Solution:

For insertion sort, the best and worst-case runtimes are $\Theta(n)$ and $\Theta\left(n^2\right)$ respectively. Insertion sort is stable and in-place (assuming it's implemented correctly). Insertion sort is a general-purpose sort.

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.

For quick sort, the best and worst-case runtimes are $\Theta(n \log(n))$ and $\Theta\left(n^2\right)$ respectively. Quick sort is not stable, but is in-place. Quick sort is also a general-purpose sort.

For counting sort, the best and worst-case runtimes are both $\Theta(n + k)$, where n is the number of items in the original list and k is size of the possible range of values in the input list. Counting sort is technically stable (in the sense that counting sort only works on ints or int-like things, and it's unclear how you sort ints in an unstable way). It is not in-place. Counting sort is *not* a general-purpose sort. It can't be used to sort an array of strings, for example.

(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 $\Theta\left(n^2\right)$, but it also has a low constant factor, which may end up making it good enough in this case.

Using either merge sort or quick 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 is 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.

We might also want to try counting sort, if the input numbers are guaranteed to be ints. However, this will still be potentially suboptimal, since if the users upload a list of numbers with a very large range (e.g. a list that contains both 0 and 100 million), counting sort would realistically take a long time and allocate an excessive amount of memory.

(We can perhaps mitigate these issues by using something like bucket sort, which is an adaption of counting sort, but bucket sort isn't one of the options above.)

(e) Suppose we want to sort an array of ints, where we know all the ints are between 0 and 1000. Which of the above algorithms is the best choice?

Solution:

Here, counting sort.

(f) 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 might initially think counting sort, but counting sort requires us to allocate an array that is the length of the range of possible input values. If that range is large, we might allocate a large array.

We instead want to use an in-place algorithm. In that case, quicksort is likely the best answer.

(g) Suppose we want to sort an array of chars. Assuming the chars are all lowercase alphabetical letters, which of the above algorithms is the best choice?

Solution:

Note that each char corresponds to an int. And if the array will only contain lowercase letters, we only need to worry about sorting a small range of ints.

So, counting sort is probably best.

(h) On the homework, you were asked to find a way of building an array that would cause a version of quick sort implemented using the median-of-three pivoting strategy to run in $\mathcal{O}\left(n^2\right)$ time.

Now, find a way of making a version of quick sort implemented using the random pivot selection strategy run in $\mathcal{O}\left(n^2\right)$ time.

Solution:

An array of all duplicates would work. (The analysis of why is omitted, since you basically already did the analysis for hw2, even if your answer was different.)

(i) 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:

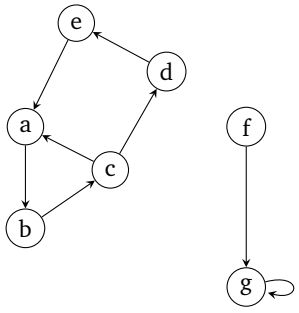
If we want to make both pivot strategies no longer degrade to $\mathcal{O}\left(n^2\right)$ given an array of all duplicates, one strategy might be to partition into three groups, not two.

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 $<$ then the pivot, $=$ to the pivot, and $>$ then 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 $\mathcal{O}\left(n\right)$ time instead of $\mathcal{O}\left(n^2\right)$ time.

6. 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:

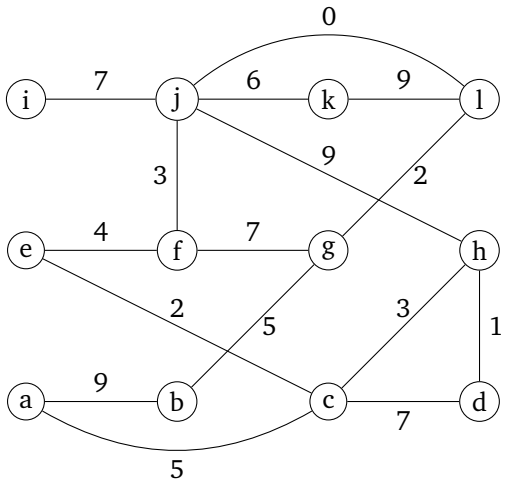
Answer: a, b, c, d, e

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

7. Dijkstra's algorithm

For each of the graphs below, 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.

(a) Run Dijkstra's algorithm starting on node a.

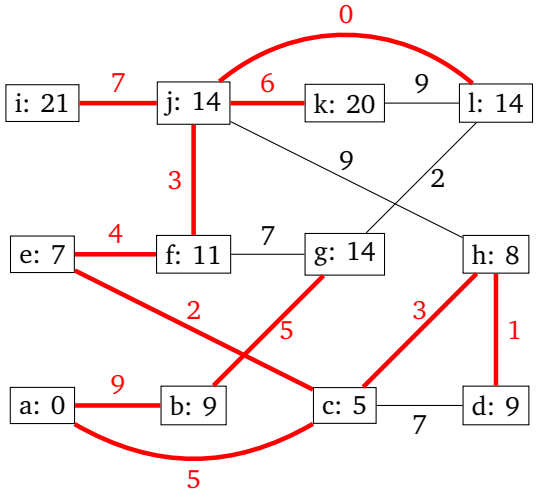


Solution:

We can draw out the answer both pictorally, and in tabular form. In tabular form:

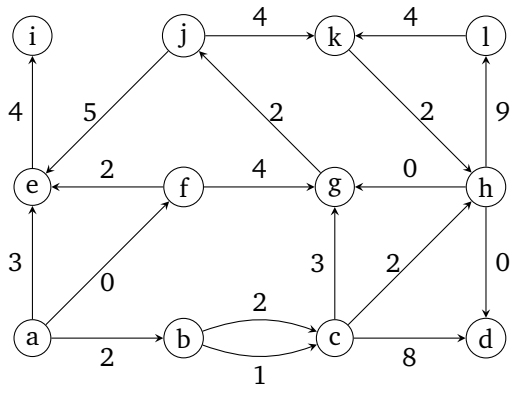
Node name	a	b	c	d	e	f	g	h	i	j	k	l
Cost	0	9	5	9	7	11	14	8	21	14	20	14
Previous node	N/A	a	a	h	c	e	b	c	j	f	j	j

In pictoral form:



Since the graph contains no negative edges, Dijkstra's algorithm will naturally return the correct result.

(b) Run Dijkstra's algorithm starting on node a.



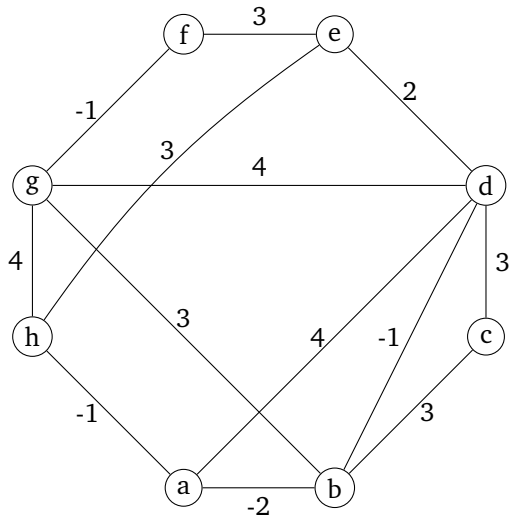
Solution:

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

Node name	a	b	c	d	e	f	g	h	i	j	k	l
Cost	0	2	3	5	2	0	4	5	6	6	7	11
Previous node	N/A	a	b	h	f	a	f	c	e	g	h	h

Since the graph contains no negative edges, Dijkstra's algorithm will naturally return the correct result.

(c) Run Dijkstra's algorithm starting on node a.



Solution:

Again, in tabular format:

Node name	a	b	c	d	e	f	g	h
Cost	0	-2	0	-3	-1	1	1	-1
Previous node	N/A	a	d	b	d	g	b	h

Dijkstra's algorithm will actually return the correct result here. This isn't guaranteed to happen when the graph contains negative edges but sometimes we get lucky.

(d) Now, run Dijkstra's algorithm on the same graph above, but starting on node e.

Solution:

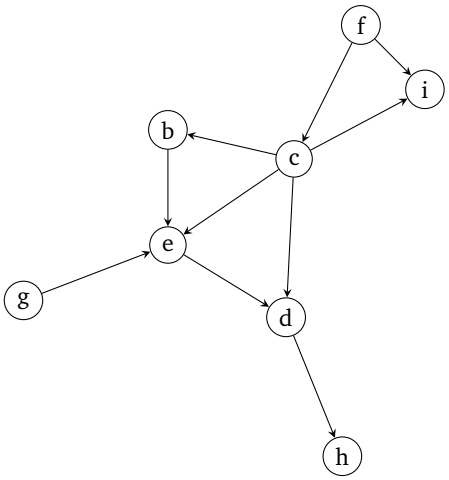
Again, in tabular format:

Node name	a	b	c	d	e	f	g	h
Cost	-1	1	4	2	0	1	2	-2
Previous node	b	d	b	e	N/A	g	h	a

Dijkstra's did not return the correct result – the shortest path from e to b was e, h, a, b for a total cost of 0, for example. However, the algorithm instead selected e, d, b for a total cost of 1.

8. Topological sort

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

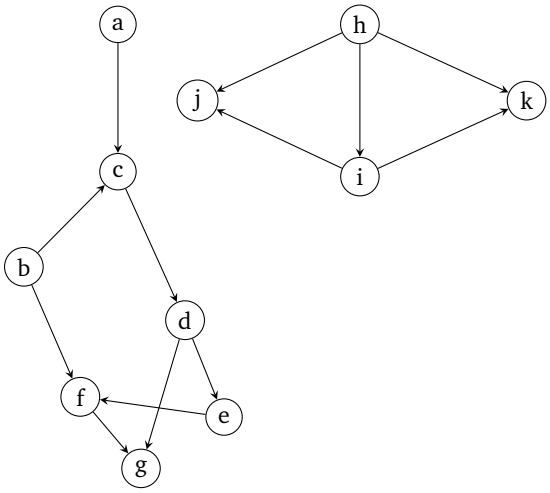


Solution:

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.

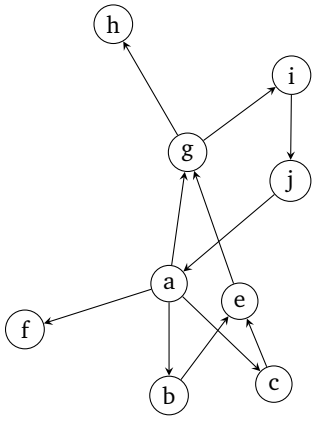


Solution:

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.

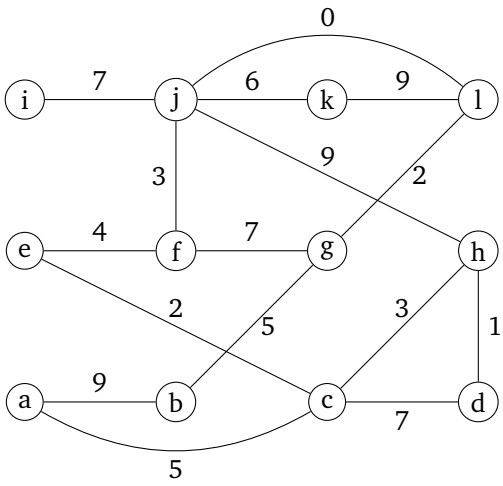


Solution:

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

9. Minimum spanning trees

Consider the following graph:

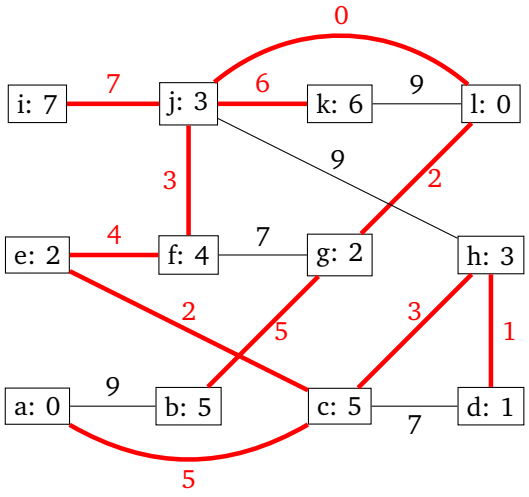


(a) Run Prim's algorithm on the above graph starting on node a to find a minimum spanning tree.

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

Solution:

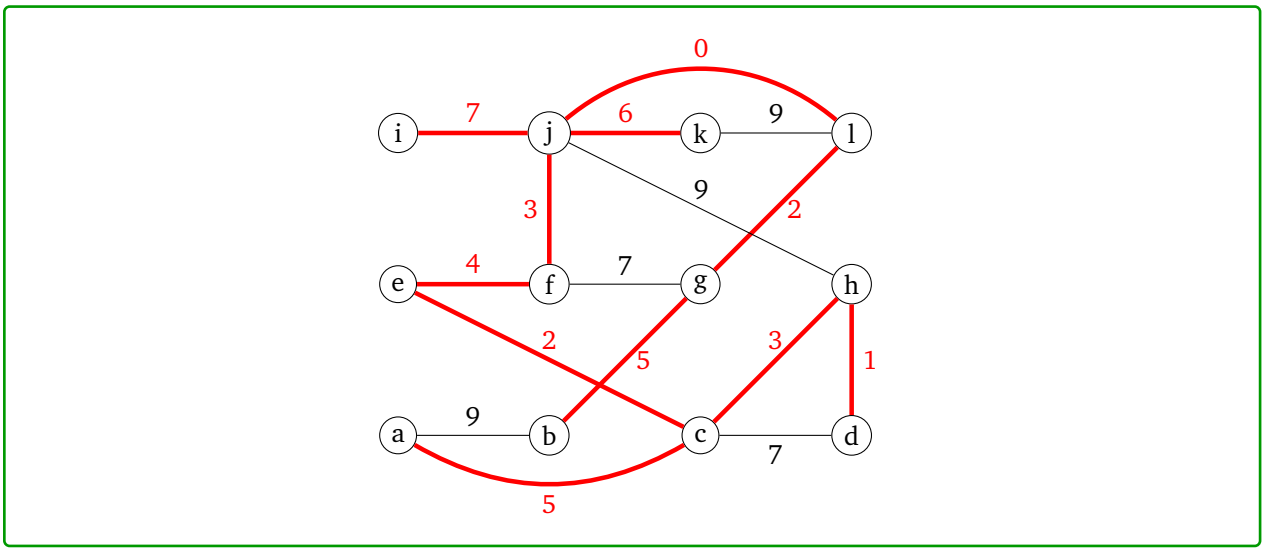
The final MST looks like the following:



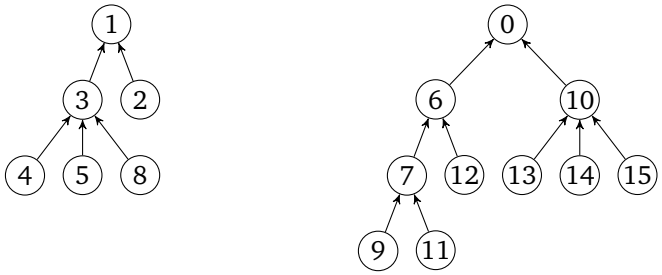
(b) Run Kruskal's algorithm on the above graph to find an MST. In the case of ties, select the edge containing the node that is the smallest alphabetically.

Draw the final MST.

Solution:

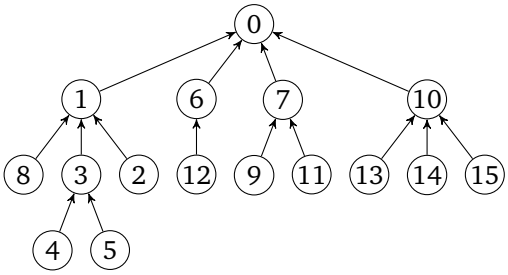


(c) Suppose we have the following disjoint set. What happens when we run `union(7, 8)`? Draw both the new trees as well as the array representation of the disjoint set.



Solution:

Assuming the left tree had an initial rank of 2 and the right tree had an initial rank of 3, the final tree would look like so:



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

The array representation would then be:

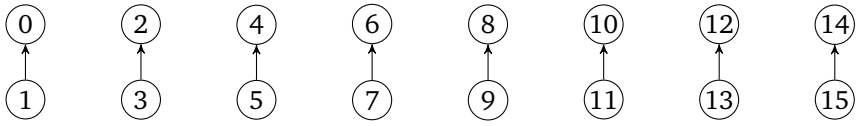
```
[-4, 0, 1, 1, 3, 3, 0, 0, 1, 7, 0, 7, 6, 10, 10, 10]
```

(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.

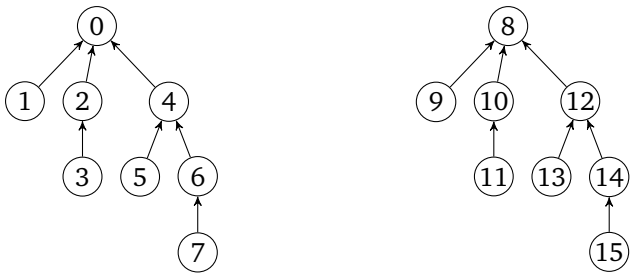
So, we start by calling `makeSet(...)` on the numbers 0 through 15, then call `union(0, 1)`, then `union(2, 3)`, ..., `union(14, 15)`. This produces the following forest:



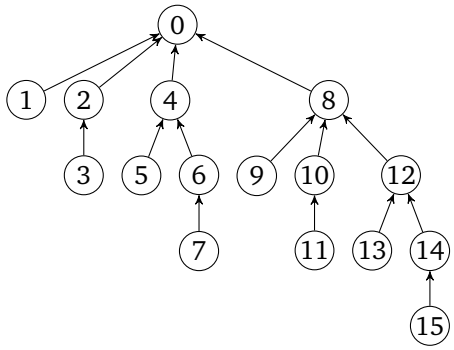
Note that every tree here has a rank of 1. We next call `union(0, 2)`, `union(4, 6)`, `union(8, 10)`, and `union(12, 14)`. Note that we're taking care to only call union on the roots of the trees, to avoid collisions.

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.

10. 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) 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).

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.

(b) 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.

```
public class QuadraticProbingHashTable<K, V> {
    private Pair<K, V>[] array;
    private int size;

    private static class Pair<K, V> {
        public K key;
        public V value;
    }

    // ...snip...

    /**
     * Deletes the key-value pair associated with the key, and
     * returns the old value.
     *
     * @throws NoSuchKeyException if the key-value pair does not exist in the method.
     */
    public V delete(K key) {
        int index = key.hashCode() % this.array.length;

        int i = 0;
        while (this.array[index] != null && !this.array[index].key.equals(key)) {
            i += 1;
            index = (index + i * i) % this.array.length;
        }

        if (this.array[index] == null) {
            throw new NoSuchKeyException("Key-value pair not in dictionary");
        }

        this.array[index] = null;

        return this.array[index].value;
    }
}
```

Solution:

The full list of all bugs:

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

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

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

(d) 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, $s+1$, $s+5$, $s+14$, $s+30$...

(e) 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.

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

11. Graphs and design

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

For each problem, describe (a) what your vertices and edges are and (b) pseudocode or an English description of how you would solve the given problem.

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?

Solution:

Setup:

A wikipedia page is a vertex, a link is a directed, unweighted edge.

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

Algorithm:

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 are given a list of statements about how cities are located relative to each other as input. For example, suppose we had the following statements as input:

- Seattle is north of Portland
- Seattle is west of Spokane
- Portland is south-east of Spokane
- Spokane is west of New York
- Seattle is south of Vancouver

These statements are all internally consistent with each other. Now, suppose we add one more statement to the list:

- Portland is north of Vancouver

If we add this statement to our list, we suddenly have an inconsistency! We previously said Seattle was north of Portland, and that Vancouver was north of Seattle. In that case, it's impossible for Portland to also be north of Vancouver.

How would you write an algorithm to determine whether a given list of statements is consistent or not?

Solution:

Setup:

The core insight here is that this problem is actually a cycle-detection problem in disguise.

This is more obvious if we consider a simplified version of the problem where we only add north-south constraints. If we take each statement and add a directed arrow from the north-most city to the south-most city, we can see that introducing a contradictory statement will necessarily introduce a cycle.

So, we build two graphs, one for north-south and one for east-west and run a cycle-detection algorithm on both.

So, we add cities as vertices to both graphs. We then take each statement. If it adds information about a north-south relationship, we add that as a directed unweighted edge in the first graph (and make sure the arrow always points south); if the statement adds information about a east-west relationship, we do the same to the other graph (and make sure the arrow always points west).

Algorithm:

```
bool isContradictory(statements):
    northSouthGraph = buildGraph(statements, 'north', 'south')
    eastWestGraph = buildGraph(statements, 'east', 'west')
    return hasCycle(northSouthGraph) or hasCycle(eastWestGraph)

Graph buildGraph(statements, head, tail):
    vertices = new List()
    edges = new List()

    for (city1, direction, city2) in statements:
        if direction does not contain head or tail:
            skip iteration

        if direction does not contain tail:
            swap the two cities (e.g. make sure city1 points to city2)

        add city1 and city2 to vertices
        add (city1, city2) to edge

    return new Graph(vertices, edges)
```

```
bool hasCycle(graph):
    for vertex in graph.vertices:
        if hasCycleStartingFrom(graph, vertex, new Set()):
            return True
    return False

bool hasCycleStartingFrom(graph, curr, visited):
    if visited.contains(start):
        return True
    else:
        for neighbor in graph.getNeighbors(curr):
            visited.add(curr)
            if hasCycleStartingFrom(graph, curr, visited):
                return True
            visited.remove(curr)
        return False
```

(c) Suppose you have a bunch of computers networked together connected 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 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 long such that (a) every computer receives the message and (b) you minimize the total risk factor. The total risk factor is define 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. We make each computer a node and each wire (with the risk factor) a weighted, undirected edge.

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.

(d) 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:

To solve this algorithm, we run either DFS or BFS on the previous 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.

The 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
```

(e) 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 $\frac{|V|(|V|-1)}{2}$.

(f) 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 "$\beth$" 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:

- apple
- apology
- banana

...we'd know the following facts:

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

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

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 then 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?

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 implement the verifier.

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 hash set), 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 algorithm also happens to be in P? Why or why not?

Solution:

Yes, mainly because this problem is very similar to the 2-COLOR problem. After all, if we think about it, if a graph contains an odd-length cycle, that also means it's impossible to 2-COLOR. And since it's possible to determine if a graph is 2-COLOR-able in polynomial time, that suggests it may also be possible to solve this algorithm in polynomial time as well.

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 instead of $\mathcal{O}(|E|)$ 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) If we have an efficient way of solving some arbitrary NP problem, we have an efficient way of solving ALL NP problems.

Solution:

False. Consider 2-COLOR, which is (technically) in NP. If we have an easy way of solving 2-COLOR, that doesn't help us solve harder problems like 3-COLOR.

(c) 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.

(d) 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.)

(e) 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.

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

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.

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

Solution:

Unknown. Since 3-COLOR is an NP-COMPLETE problem, that means that if we had an efficient (e.g. polynomial) solver for 3-COLOR, 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.

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

Solution:

False. Suppose the list is an arraylist.

(i) 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 then $\Theta(n \log(n))$. 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).

(j) 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 $|V| \in \Theta(|E|)$.

(k) 2-COLOR is in NP.

Solution:

True. 2-COLOR is in P; all problems in P are also in NP.

(l) 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 $\mathcal{O}(|n|)$.

(m) 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.

(n) 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.

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

Solution:

False. Again, consider 2-COLOR.

(p) 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.

Identities

Splitting a sum

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

Gauss's identity

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

Factoring out a constant

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

Finite geometric series

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

Change-of-base identity

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

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 $T(n) \in \Theta\left(n^{\log_b(a)}\right)$