Section 07: Solutions

1. Sorting: Mystery

Consider the following sorting algorithm in pseudocode. Note that, in this case, the upper bound of each for loop is *inclusive*, so they run up to and including $i = A.\text{length} - 1$ and $j = i - 1$.

```
1: function MysterySort(A)
2:     for i = 1 to A.length − 1 do
3:         for j = 0 to i − 1 do
4:             if A[j] ≥ A[i] then
5:                 x = A[i]
6:                 shift every item from j to i − 1 right by one
7:                 A[j] = x
8:                 break
```

(a) Is MysterySort most similar to insertion sort, merge sort, quick sort, or selection sort?

Solution:

MysterySort is most similar to insertion sort. After every iteration of the i for-loop, a new item has been *inserted* somewhere in the sorted section of the array. Unlike selection sort, the new item could be inserted anywhere in the sorted section, not just at the end. This may involve shifting items over to the right to make room.

(b) Is MysterySort a stable sorting algorithm? Why or why not?

Solution:

MysterySort is unstable. This is due to a combination of reasons:

(i) The j for-loop goes over the array from left-to-right;

(ii) The if-condition that inserts the item does not use strict inequality.

This means MysterySort will insert each new item *before* the first item that is greater than or *equal* to it. Since each successive new item came *after* the last one in the original array, this reverses the order of equal items, which means MysterySort is unstable.

Note that insertion sort is normally a stable sorting algorithm. This could be considered a bug in MysterySort, since stability is a desirable property among sorting algorithms. You could make MysterySort stable by changing line 4 to check if $A[j] > A[i]$ instead of $A[j] \geq A[i]$.

(c) What is the best-case runtime (as a tight big-$\mathcal{O}$ bound) for MysterySort? Why is this the best case?

Hint: What happens when MysterySort is given an array that is already sorted?

Solution:

The best-case runtime is $\mathcal{O}(n^2)$.

Consider that for an array A that is already sorted, $A[i] \geq A[j]$ for all $i > j$. This means that the if-condition will never be true (ignoring equal items for now). The j for-loop will then always run i times, which is $\mathcal{O}(n^2)$.

For an array A that is sorted in *reverse* order, $A[i] \leq A[j]$ for all $i > j$. This means that the if-condition will always be true, so the loop runs only once. But when it runs, it shifts every item from j to $i-1$ right by one; since $j = 0$, this takes i operations. So it is also $\mathcal{O}(n^2)$.

Any array will be some combination of these two extremes: the i for-loop will run for k iterations before shifting $i-k$ items over. Either way, it has to visit $k + (i-k) = i$ items every time, so it is always $\mathcal{O}(n^2)$.

Note that insertion sort normally runs in $\mathcal{O}(n)$ time in the best case, which is an already sorted array. You could fix MysterySort by reversing the j for-loop so it goes from $j = i-1$ to 0, and changing the if-condition so it checks if $A[i] \geq A[j]$. This is always true if A is already sorted, so the loop runs once; but this time, it tries to shift over $(i-1) - j = (i-1) - (i-1) = 0$ items! This takes constant time, so MysterySort will run in $\mathcal{O}(n)$.

2. Sorting: Design Decisions

For each of the following scenarios, say which sorting algorithm you think you would use and why. There may be more than one right answer.

(a) Suppose we have an array where we expect the majority of elements to be sorted "almost in order". What would be a good sorting algorithm to use?

Solution:

Merge sort and quick sort are always predictable standbys, but we may be able to get better results if we try using something like insertion sort, which is $\mathcal{O}(n)$ in the best case.

(b) You are writing code to run on the next Mars rover to sort the data gathered each night. (Think about sorting with limited memory and computational power.)

Solution:

Since each memory stick costs thousands (millions?) of dollars to send to Mars, an in-place sort is probably your best bet. Among in-place sorts, heap sort is a great choice (since it is guaranteed $\mathcal{O}(n \log n)$ time and doesn't even use much stack memory). Insertion sort meets memory needs, but wouldn't be fast.

(c) You're writing the backend for the website `SortMyNumbers.com`, which sorts numbers given by users.

Solution:

Do you trust your users? I wouldn't. Because of that, I want a worst-case $\mathcal{O}(n \log n)$ sort. Heap sort or Merge sort would be good choices.

(d) Your artist friend says for a piece she wants to make a computer sort every possible ordering of the numbers $1, 2, \ldots, 15$. Your friend says something special will happen after the last ordering is sorted, and you'd like to see that ASAP.

Solution:

Since you're going to sort all the possible lists, you want to optimize for the average case – Quick sort has the best average case behavior, which makes it a really good choice. Merge sort and heapsort also have average speed of $\mathcal{O}(n \log n)$ but they're usually a little slower on average (depending on the exact implementation).

She didn't appreciate your snarky suggestion to "just print $[1, 2, \ldots, 15]$ $15!$ times." Something about not accurately representing the human struggle.

3. Sorting: Algorithm Practice

(a) Demonstrate how you would use quick sort to sort the following array of integers. Use the first index as the pivot; show each partition and swap.

[6, 3, 2, 5, 1, 7, 4, 0]

Solution:

[solutions omitted]

(b) Show how you would use merge sort to sort the same array of integers.

Solution:

[solutions omitted]

4. Memory: Short Answer

(a) What are the two types of memory locality?

Solution:

Spatial locality is memory that is physically close together in addresses. Temporal locality is the assumption that pages recently accessed will be accessed again.

(b) Does this more benefit arrays or linked lists?

Solution:

This typically benefits arrays. In Java, array elements are forced to be stored together, enforcing spatial locality. Because the elements are stored together, arrays also benefit from temporal locality when iterating over them.

5. Memory: In Context

(a) Based on your understanding of how computers access and store memory, why might it be faster to access all the elements of an array-based queue than to access all the elements of a linked-list-based queue?

Solution:

The internal array within the array-based queue is more likely to be contiguous in memory compared to the linked list implementation of an array. This means that when we access each element in the array, the surrounding parts of the array are going to be loaded into cache, speeding up future accesses.

One thing to note is that the array-based queue won't necessarily automatically be faster then the linked-list-based one, depending on how exactly it's implemented.

A standard queue implementation doesn't support the `iterator()` operation, and a standard array-list based queue implements either $\mathcal{O}(n)$ enqueue or dequeue.

In that case, if we're forced to access every element by progressively dequeueing and re-enqueuing each element, iterating over a standard array-based queue would take $\mathcal{O}(n^2)$ time as opposed to the linked-list-based queue's $\mathcal{O}(n)$ time. In that case, the linked-list version is going to be far faster then the array-list version for even relatively smaller values of n.

The only way we could have the array-based queue be consistently faster is if it supported $\mathcal{O}(1)$ enqueues and dequeues. (Doing this is actually possible, albeit slightly non-trivial.)

(b) Why might `f2` be faster than `f1`?

```
public void f1(String[] strings) {
    for (int i=0; i < strings.length; i++) {
        strings[i] = strings[i].trim();        // omits trailing/leading whitespace
    }
    for (int i=0; i < strings.length; i++) {
        strings[i] = strings[i].toUpperCase();
    }
}

public void f2(String[] strings) {
    for (int i=0; i < strings.length; i++) {
        strings[i] = strings[i].trim(); // omits trailing/leading whitespace
        strings[i] = strings[i].toUppercase();
    }
}
```

Solution:

Temporal Locality. At each iteration, the specific string from the array is already loaded into the cache. When performing the next process `toUppercase()`, the content can just be loaded from cache, instead of disk or RAM.

(c) Consider the following code:

```
public static int sum(IList<Integer> list) {
    int output = 0;
    for (int i = 0; i < 128; i++) {
        // Reminder: foreach loops in Java use the iterator behind-the-scenes
        for (int item : list) {
            output += item;
        }
    }
    return output;
}
```

You try running this method twice: the first time, you pass in an array list, and the second time you pass in a linked list. Both lists are of the same length and contain the exact same values.

You discover that calling `sum` on the array list is consistently 4 to 5 times faster then calling it on the linked list. Why do you suppose that is?

Solution:

This is most likely due to spatial locality. When we iterate through a linked list, accessing the value at one particular index will load the next few elements into the cache, speeding up the overall time needed to access each element.

In contrast, each node in the linked list is likely loaded in a random part of memory – this means we likely must load each node into the cache, which slows down the overall runtime by some constant factor.

(d) Suppose you are writing a program that iterates over an `AvlTreeDictionary` – a dictionary based on an AVL tree. Out of curiosity, you try replacing it with a `SortedArrayDictionary`. You expect this to make no difference since iterating over either dictionary using their iterator takes worst-case $\Theta(n)$ time.

To your surprise, iterating over `SortedArrayDictionary` is consistently almost 10 times faster!

Based on your understanding of how computers organize and access memory, why do you suppose that is? Be sure to be descriptive.

Solution:

This is almost absolutely because the `SortedArrayDictionary` is implemented with an array, which has much better spatial locality, than how the `AvlTreeDictionary` is most likely implemented, with a series of linked AVL tree nodes. Since we know that iterating over an array is faster than iterating over a linked list, the reasoning behind a faster tree is similar and reasonable.

(e) Excited by your success, you next try comparing the performance of the `get(...)` method. You expected to see the same speedup, but to your surprise, both dictionaries' `get(...)` methods seem to consistently perform about the same.

Based on your understanding of how computers organize and access memory, why do you suppose that is?

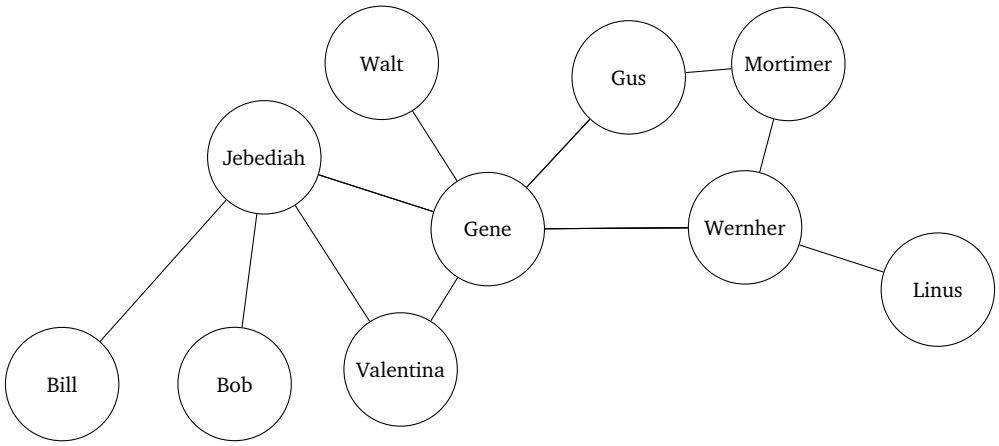
(Note: assume that the `SortedArrayDictionary`'s `get(...)` method is implemented using binary search.)

Solution:

Spatial locality can only be taken advantage of when iterating sequentially. With that, it's not surprising that because we have to jump from `i=100` to `i=50`, etc. until we find what we're looking for. If the array is so big that it spans over multiple pages, the locality that regular iteration takes advantage of is not available to jumping around with `get()`.

6. An Introduction to Graphs

Consider the following graph that models friendships between people on Facebook.



(a) What do the vertices in this graph represent? What do the edges represent?

Solution:

The vertices represent people and the edges represent friendships.

(b) Is this a directed or undirected graph? Why does this kind of graph make sense for Facebook friends?

Solution:

The graph is undirected because friendships on Facebook are mutual: Jebediah can't be friends with Bill unless Bill is also friends with Jebediah. You could still use a directed graph, but you would have to draw twice as many edges to represent both directions of friendship, so the undirected graph is more concise.

(c) Which vertex has the highest degree? What degree does it have?

Solution:

Gene has the highest degree, with a degree of 5.

(d) Gus has an urgent message to send to Bob. What is the shortest path he can use to send his message?

Solution:

The shortest path between Gus and Bob is Gus – Gene – Jebediah – Bob.

(e) Using your data structures from the homework projects, create this graph as an adjacency list in Java.

Hint: The type of the graph should be `IDictionary<String, ISet<String>>`. For convenience, you can assume `ChainedHashSet` has a constructor that can initialize the set with a list of arguments. For example, `new ChainedHashSet<String>("Bill", "Bob", "Valentina")` would create the set {Bill, Bob, Valentina}.

Solution:

```
IDictionary<String, ISet<String>> graph = new ChainedHashDictionary<>();
graph.put("Jebediah", new ChainedHashSet<>("Bill", "Bob", "Valentina", "Gene"));
graph.put("Bill", new ChainedHashSet<>("Jebediah"));
graph.put("Bob", new ChainedHashSet<>("Jebediah"));
graph.put("Valentina", new ChainedHashSet<>("Jebediah", "Gene"));
graph.put("Gene", new ChainedHashSet<>("Jebediah", "Valentina", "Walt", "Gus", "Wernher"));
graph.put("Walt", new ChainedHashSet<>("Gene"));
graph.put("Gus", new ChainedHashSet<>("Gene", "Mortimer"));
graph.put("Wernher", new ChainedHashSet<>("Gene", "Mortimer"));
graph.put("Mortimer", new ChainedHashSet<>("Gus", "Wernher"));
graph.put("Linus", new ChainedHashSet<>("Wernher"));
```

Note that every edge in the original graph is represented twice in the adjacency list. For example, Bill is in Jebediah's set, and Jebediah is also in Bill's set. This is because an undirected graph needs to be converted to a directed graph before it can be stored as an adjacency list, and each edge in an undirected graph is equivalent to two edges going both directions in a directed graph.

(f) Write a method `String maxDegree(IDictionary<String, ISet<String>> graph)` that returns the vertex with the highest degree.

Solution:

```
String maxDegree(IDictionary<String, ISet<String>> graph) {
    String vertex = null;
    int degree = -1;
    for (KVPair<String, ISet<String>> pair : graph) {
        if (pair.getValue().size() > degree) {
            vertex = pair.getKey();
            degree = pair.getValue().size();
        }
    }
    return vertex;
}
```