**Chapter 2 and 4 – Practice Exercises**

**For D&C:**

* Is the input sorted? Then you likely need *binary search*.
* Is it unsorted, but you need to find some number of kth items? Then you likely need *fast select*.
* Is it unsorted, but you need to find an arbitrary number of things? Then you likely want to sort it first with *mergesort*.
* Is it unsorted, but you want to eliminate elements in some comparative way? Then you likely want to *modify* the merge of *mergesort* which is guaranteed to compare elements of the same weights.
* Is it some weird way to describe math? Try *Fast Fourier Transform*.

**Exercise 4.11**

**Input:** Directed graph with positive edges

**Goal:** Length of shortest cycle, should also include if graph is acyclic

1.) **Solution**

Since this is a directed graph with only positive edges, a negative cycle is not possible. The shortest cycle is identified when taking the distance, *d*,between vertices *u* and *v*, *d(u, v) + d(v, u)*. So our algorithm is *min{d(u, v) + d(v, u)}*. If quantity equals infinity, then the graph is acyclic and is reported.

Using Floyd-Warshall on the graph gives us an array of *{d(u, v)}* for each V of all the distances between any pair of points. At the beginning of any *(u,v)* pair not represented by an edge, E, is set to: *d(u, v) = infinity*. By inspecting the diagonal in the array, our shortest path is the *min{d(v, v)}* across all *v*’s of V.

3.) **Why it works**

Every cycle visiting vertices *u* and *v* are broken into two directed paths: *u* to *v* and *v* back to *u*. Each path achieves the minimum length at *d(u, v)* and *d(v, u)* respectively, and the cycle of minimal length is given by this sum. Floyd-Warshall take *O(|V|3)* and the minimum can be found in *O(|V|)*, making the overall runtime *O(|V|3)*.

**Exercise 4.21**

**Input:** currencies ranging from c[1] to c[i]

**Goal:** Find the shortest path to convert the desired two currencies

**Part A:**

1.) **Solution**

This is a directed graph problem where we need to recognize that the currency calculation is a product, and the product needs to be maximized. We solve the currency calculation by converting the exchange rates to logs (*log(a) + log(b) = log(a \* b))*. This is then solved by using the negative of the log for our edge weight, flipping maximization to minimization.

We created a graph where vertices represent the currency a country uses, and the edges between them have a weight of *w[i, j] = -log r[i, j]*. We then run Bellman-Ford’s algorithm from vertex *s* to vertex *t*, and the minimal weight path represents the best sequence of currency trades. This would run in *O(|V||E|)* time.

2.) **Why it works**

Converting the exchange rates to logs, then negating that value, the weight of the shortest path (sum of the negated logs) will represent the series of exchanges which maximizes the product of the exchange rates. Since edge weights are negative, we cannot use Dijkstra’s algorithm.

**Part B:**

1.) **Solution**

This anomaly can be solved by running one more iteration of Bellman-Ford, and if any weights between two points changes, we’ve detected a negative cycle, where the cycle represents the trading anomaly allowing infinite profit.

**Exercise 2.5**

**Solutions:**

1. O(nlog32) - Rule #3 in Master theorem and “log change of base rule”: *a = 2, b = 3, d = 0*
2. O(nlog45) - Rule #3 in Master theorem and “log change of base rule”: *a = 5, b = 4, d = 1*
3. O(n log n) - Rule #2 in Master theorem and “log change of base rule”: *a = 7, b = 7, d = 1*
4. O(n2 log n) - Rule #2 in Master theorem and “log change of base rule”: *a = 9, b =3, d = 2*
5. O(n3 log n) - Rule #2 in Master theorem and “log change of base rule”: *a = 8, b =2, d = 3*
6. O(n3/2 log n) – Page 55 in Chapter 2 (ratio of a/bd < 1): a *= 49, b =25, d = 3/2*
7. O(n) – Substitution, refer to answer key
8. O(nc + 1) – Substitution, refer to answer key
9. O(cn) = Substitution, refer to answer key and Geometric Series
10. O(2n) – Substitution, refer to answer key
11. O(log log n) - Substitution, refer to answer key and Ed Discussion

**Binary search modified**

**My approach:** *Binary Search Modified*

* First, check if the *start* input (initialized at 1) is greater than the *end* (initialized at *n – 1*). If true, then return *end* *+ 1*.
* Next, check if the *start* input is not equal to the number at *start* index. If true, then return *start*.
* If neither is true, calculate the *mid* value by adding *start* to *end* and diving the sum by 2. *mid = (start + end)/2*.
  + Next, check if the number at index *mid* equals the value of *mid*. If true, recursively call the function with inputs *A, mid + 1,* and *end*.
  + If the first condition is false, recursively call the function with inputs *A, start,* and *mid*.

**Why it works:**

The input array is in a sorted and non-decreasing order. If the number sequence beings at 1, then the missing value is the lowest index, this is where our *start* does not equal *A[start]*. If we get down to a single element, the *start* index is greater than the *end* index and we return *end + 1*. Once we check if *A[mid] == mid* is true, then we know all numbers in the array up to *mid* are present, and we need to check the right-hand side of the array. If the condition is false, we know to check the left-hand side for our missing number.

**Runtime:** *O(log n)*

The binary search algorithm finds the repeated numbers within the sorted array by essentially diving the array into smaller subarrays at each run. The recurrence for this evaluates to T(n) = T(n/2) + O(1), where a = 1, b = 2, and d = 0. Using the master theorem, we get an overall runtime of O(log n), matching the typical runtime of the binary search algorithm.