Practice Problem Set 2

Problem 1:

Consider the following recursive algorithm, which takes as input an array A and start and finish indices, s and f. What is the output when the algorithm is run on array $A = 1, 2, \ldots, 12$ with $s = 1$ and $f = 12$. Next, derive a recurrence for the runtime of the algorithm $T(n)$ and use the master method to determine the runtime in big-Theta notation.

```
Practice(A,s,f)
   if s <  f
      q = round-down (2s+f)/3
      for i = q+1 to f
         print A[i]
      Practice(A,s,q)
   else print A[s]
```

Problem 2:

Let A be an array of n distinct numbers sorted in *increasing* order. Binary search is a technique the searches for a particular number k by comparing k to the item in the middle of A and then either recursing to the left subarray or the right subarray. Let $BSearch(A, s, f, k)$ be the binary search procedure, which returns true if element k is contained in the array A between indices s and f. Write the pseudo-code for $BSearch(A, s, f, k)$. Explain why the worst-case runtime has recurrence $T(n) = T(n/2) + c$. Show that $T(n)$ is $O \log n)$ using the substitution method. Repeat for Master method.

**If you haven't seen binary search before, here is another resource:*

`https://www.youtube.com/watch?v=iP897Z5Nerk`

Problem 3:

Update the pseudo-code above so that if element k is in the array $A[s, \ldots, f]$, then the procedure returns the *index* of the element k. Otherwise the procedure returns NIL

Problem 4:

Below is the pseudo-code for an algorithm that searches for the value k in a sorted array A indexed from s to f. If the value k is in the array A, it returns true. Otherwise, it returns false. The algorithm makes a call to Bsearch(A,s,f,k) which we saw in class.

MySearch(A,s,f, k)
    if $s < f$
        $q = \lfloor (s + f)/2 \rfloor$
        if BSearch$(A, s, q, k) =$ False
            return MySearch(A,q+1,f,k)
        else return True
    else if $k = A[s]$
        return True
    else return False

- Determine the worst-case runtime recurrence for the algorithm: $T(n)$.
- Show that $T(n)$ is $O((\log n)^2)$ using two methods: the recursion tree, and substitution.

Problem 5:

Below are two recursive algorithms for finding the maximum element in an array of size n.

```
Findmax1(A,s,f)
   if (s<f)
      q = round-down((f+s)/2)
      m1 = Findmax1(A,s,q)
      m2 = Findmax1(A,q+1,f)
      if (m1>m2) return m1
      else return m2
   else return A[s]
```

```
Findmax2(A,s,f)
   if (s<f)
      m1 = Findmax2(A,s,f-1)
      if (m1>A[f]) return m1
      else return A[f]
   else return A[s]
```

Briefly explain why each algorithm correctly returns the max. Determine a recurrence for the runtime of each algorithm. Do both algorithms have a runtime of $\Theta(n)$? Justify your answer.

Problem 6:

Below is the pseudo-code for two algorithms: Practice1(A,s,f) and Practice2(A,s,f), which take as input a **sorted** array A, indexed from s to f. The algorithms make a call to *Bsearch(A,s,f,k)* which we saw in class. Determine the worst-case runtime recurrence for each algorithm: $T_1(n)$ and $T_2(n)$. Show that $T_1(n)$ is $O((\log n)^2)$ and $T_2(n)$ is $O(n \log n)$.

Practice1(A,s,f)
  if $s < f$
    $q1 = \lfloor (s+f)/2 \rfloor$
    if BSearch(A,s,q1,1) = true
      return true
    else
      return Practice1(A, q1+1, f)
  else
    return false

Practice2(A,s,f)
  if $s < f$
    if BSearch(A,s+1,f,1) = true
      return true
    else
      return Practice2(A, s, f-1)
  else
    return false

Problem 7:

Rewrite the pseudo-code for bubble-sort as a recursive algorithm. Explain why this new version has the same best and worst case asymptotic runtimes as the original version.

Problem 8:

Rewrite selection-sort as a recursive algorithm. Call your algorithm SelectionSort(A,s,f) where A is the input array and s and f are the start and finish indices of the array. Express the runtime worst-case runtime $T(n)$ using a recurrence. Show that $T(n)$ is $O(n^2)$ using the substitution method. Repeat for Insertion-sort.

Problem 9:

Apply the Master Method to each of :

$T(n) = T(\frac{19n}{20}) + n^3$.

$T(n) = 9 \cdot T(\frac{n}{3}) + n^2 \log^5 n$.

$T(n) = 10 \cdot T(\frac{n}{3}) + n^4 \log n$

$T(n) = 9 \cdot T(\frac{n}{3}) + n^3 \log n$

Problem 10:

- Suppose the runtime of an algorithm has the recurrence $T(n) = T(\sqrt{n}) + \log n$. Show that this is $O(\log n)$ using the recursion tree. Assume $T(1) = c$.

- Suppose the runtime of an algorithm has the recurrence $T(n) = 2T(n/4) + n$. Show that this is $O(n)$ using the recursion tree. Assume $T(1) = c$.
- Suppose an algorithm runs in worst-case time $T(n) = 2T(n/4) + \sqrt{n}$. . Use the recursion tree to determine the runtime of this algorithm in big-Theta notation. Repeat using master method.
- Suppose an algorithm runs in worst-case time $T(n) = 4T(n/2) + 1$. . Use the recursion tree to determine the runtime of this algorithm in big-Theta notation. Repeat using master method.

Problem 11:

Suppose we have a hash table of size 13, where collisions are resolved with chaining. We insert $2, 23, 14, 27, 16, 20, 21, 29, 37, 65, 39$ using the hash function $h(k) = k \mod 13$. Show the result of these insertions. Now repeat the process using linear probing. Repeat for quadratic probing using $a = 1$ and $b = 2$. Repeat for double hashing, where $h_1(k) = k \mod 13$ and $h_2(k) = (k+1)^2 \mod 13$.

Problem 12:

Suppose T is a hash table of size 100. Exactly n keys are hashed into the table using a uniform hash function. Collisions are resolved with chaining. If we carry out 10 inserts, what is the chance that there are no collisions? If we insert all n keys, what is the chance that there is a chain of length n? Do either of these events seem likely? After all n insertions, what is the expected chain length?

Problem 13:

Suppose students A and B each create a hash table of size 10. Both students use the primary hash function $h(k) = k \mod 10$. Person A decides to use linear probing to resolve collisions, and person B decides to use chaining. Give an example of a set of keys and their insertion order demonstrating that person B will have fewer probes when searching for a particular key.

Problem 14:

Suppose we hash keys into a hash table $T[0, \ldots, n-1]$ using a uniform hash function. Collisions are resolved with chaining. If we insert n keys into the table, what is the expected time to search for a random key? Repeat for $2n$ keys and n^2 keys.