Practice Set 2: Solutions

Problem 1:

Insertion Sort: In general, this algorithm works well on arrays that are nearly sorted. Suppose the elements that were swapped were at indices k and m. In the array below for example, the element at index $k = 3$ was swapped with the element at index $m = 10$

$$1, 2, 10, 4, 5, 6, 7, 8, 9, 3, 11, 12, 13, 14, 15$$

The outer loop of insertion sort loops from $i = 2$ to $i = n$ and only performs swaps if the element at position $A[i]$ is out of place. Therefore for $i = 2$ to k, no swaps are made. When $i = k + 1$ (element 4) there is exactly **one** swap performed. Similarly, for $i = k + 2$ to $i = m - 1$ only one swap is made for each iteration. At this stage, the array is $1, 2, 4, 5, 6, 7, 8, 9, 10, 3, 11, 12, 13, 14, 15$ and a total of $m - k-1$ swaps have been made. Finally, when $i = m$, element 3 will get swapped all the way to its final position and the array will be sorted: $1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15$. The number of swaps required to place element 3 into position is $m - k$. After that, no swaps are performed. The total number of swaps is at most $2(m - k) \leq 2n$. Similarly, the number of comparisons is $\leq 2n$. Therefore the overall runtime is $O(n)$.

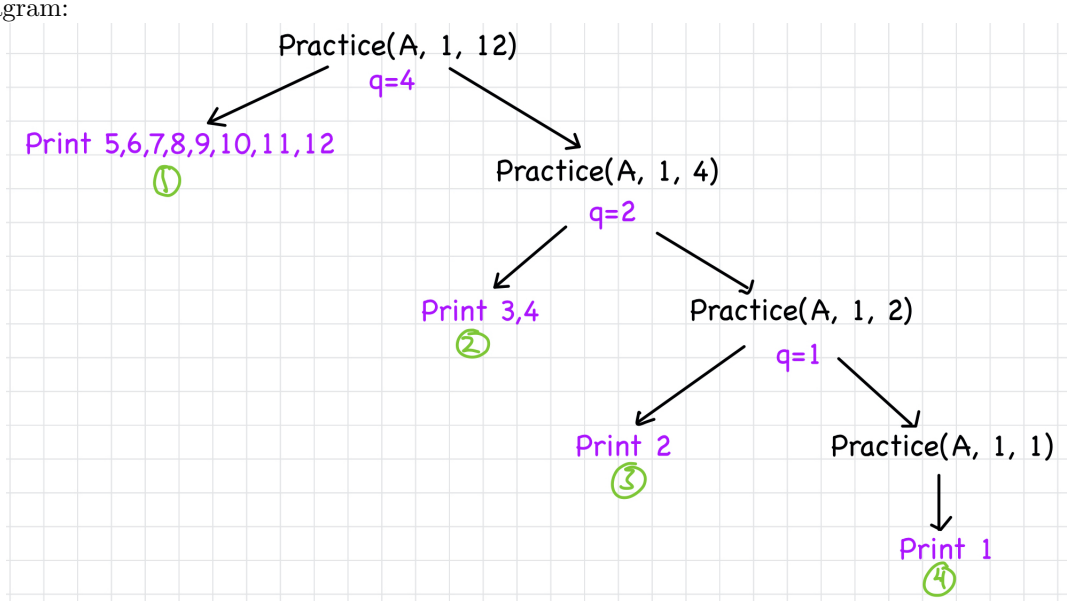
Mergesort: The runtime of Mergesort is unaffected by the sorted order of the array. Mergesort on a nearly sorted array will still run in time $O(n \log n)$.

*Selectionsort:*The runtime of Selection sort is unaffected by the sorted order of the array. Selection sort always runs in time $O(n^2)$.

In this case, the best choice would be Insertion Sort.

Problem 2:

The output of the recursive algorithm is $5, 6, 7, 8, 9, 10, 11, 12, 3, 4, 2, 1$. This can been seen from the following recursive diagram:



The algorithm prints approximately $2n/3$ elements, and then makes a recursive call to the first *third* of the subarray. Therefore $T(n) = T(n/3) + c2n/3$. Using master method, $k = log_3(1) = 0$. Therefore $n^k = n^0 = 1$. Function $f(n) = 2cn/3$ and therefore $T(n)$ is $\Theta(n)$.

Problem 3:

The algorithm below returns true if element k is contained in the array $A[s, \ldots, f]$.

```
BSearch(A,s,f,k)
  if (s<f)
     q = round-down((f+s)/2)
     if A[q] = k
       return true
     else if A[q] < k
       return BSearch(A,q+1,f)
     else
       return BSearch(A,s,q-1)
  else if A[s] = k return true
```

```
else return false
```

The algorithm makes only one recursive call, to an array of size $n/2$ approximately. Other than the recursive call, the other steps in the algorithm all all constant time. So the recurrence for the worst-case runtime is $T(n) = T(n/2) + c$. We can show this is $O(\log n)$ using substitution:

-*Goal*: Show $T(n) \leq d \log n$ for some constant d.

-*Assume* for induction that $T(n/2) \leq d \log(n/2)$

-*Substitute* : $T(n) = T(n/2) + c \leq d \log(n/2) + c = d \log n - d + c \leq d \log n$ where the last step is true as long as $d \geq c$.

Therefore $T(n)$ is $O(\log n)$.

If we apply Master method, $b = 2$ and $a = 1$. Then $k = \log_2 1 = 0$, so $n^0 = 1$. Now we compare $f(n) = c$ and $n^0 = 1$. They are asymptotically the same (both are constants). Therefore we are in case 2 of the master method, and $T(n)$ is $\Theta(\log n)$. The worst-case runtime of binary search is $\Theta(\log n)$. The best-case is $O(1)$ since the algorithm may find the element k at the first step and return true immediately.

Problem 4:

The Findmax1 algorithm finds the max of the left subarray, and the max of the right subarray, and returns whichever is bigger. If the array has only one element in it, it returns that value. The recurrence for the runtime is the same in the best and worst case: $T(n) = 2T(n/2) + c$, since it makes two recursive calls to arrays of half the size, and the remaining calls in the algorithm are all constant (comparisons, etc). In this case, $b = 2$ and $a = 2$, so $k = \log_2 2 = 1$. Now we compare $f(n) = c$ to $n^1 = n$. This is $\Theta(n)$ by the master method.

The Findmax2 algorithm finds the maximum of all elements in the array except the last. It then compares that maximum to the last element in the array and returns whichever is bigger. The recurrence for the runtime is the same in the best and worst case: $T(n) = T(n-1) + c$. We can show that this is $O(n)$ using substitution.

-*Goal*: Show $T(n) \leq dn$ for some constant d.

-*Assume* for induction that $T(n-1) \leq d(n-1)$

-*Substitute* : $T(n) = T(n-1) + c \leq d(n-1) + c = dn - d + c \leq dn$ where the last step is true as long as $d \geq c$.

Therefore $T(n)$ is $O(n)$.

Problem 5:

**the solution below finds the index of the first third of the array. This is found using (2s+f)/3. The second third of the array is actually just the median index of the subarray from q1+1 to f.*

```
Tsearch(A,s,f,k)
  if (s = f )
         if k = A[s] return true
         else return false
  else if (f = s+1)
         if  (A[s]=k or A[f] = k) return true
         else return false
  else
         q1=round-down((2s+f)/3)
         q2 = round-down((q1+1+f)/2)
         if k = A[q1]
              return true
         else if k< A[q1]
              return Tsearch(A, s, q1-1,k)
         else if k = A[q2]
              return true
         else if k < A[q2]
              return Tsearch(A, q1+1, q2-1,k)
         else return Tsearch(A,q2+1, f,k)
```

In the worst case, the new algorithm makes exactly one recursive call to input of approximate size $n/3$. The runtime recurrence is now $T(n) = T(n/3) + c$. By the master method, $k = \log_3 1 = 0$ and we compare $f(n) = c$ and $n^0 = 1$, and they are asymptotically the same. This has solution $\Theta(\log n)$ by the master method, which is the same as problem 3.

Problem 6:

Bubble sort can be written recursively where we simply replace the while-loop with a recursive call that repeats the swapping procedure until no swaps are detected. The number of swaps and comparisons remains identical, and therefore his recursive version is also $O(n^2)$ in the worst case and $O(n)$ in the best case.

```
BubbleSort(A,s,f)
   swapped = false
   for i = s to f-1
       if A[i] > A[i+1]
           Swap A[i] and A[i+1]
           swapped = true
   if swapped = true
       BubbleSort(A,s,f)
```

Problem 7:

```
SelectionSort(A,s,f)
  if s<f
     min_index = s
     for i=s+1 to f
        if A[i] < A[min_index]
           min_index = i
     Swap A[min_index] and A[s]
     SelectionSort(A,s+1,f)
```

For an array of size $n > 1$, the above algorithm makes carries out cn operations to find the maximum, and then makes a recursive call on an array of size $n-1$. The recurrence for the runtime is therefore $T(n) = T(n-1) + cn$. We can show this is $O(n^2)$ using substitution:

Goal: Show $T(n) \leq dn^2$.

Assume: $T(n-1) \leq d(n-1)^2$

Substitute:

$$\begin{aligned} T(n) &= T(n-1) + cn \\ &\leq d(n-1)^2 + cn \\ &= d(n^2 - 2n + 1) + cn \\ &\leq dn^2 - 2dn + d + cn \\ &\leq dn^2 - 2dn + dn + cn \\ &\leq dn^2 - n(d-c) \\ &\leq dn^2 \end{aligned}$$

where the last line is true as long as $d \geq c$.

Insertion Sort is implemented recursively below. The algorithm makes a recursive call to an array of size $n-1$ and then performs cn operations in the worst-case in order to insert element $A[f]$.

```
InsertionSort(A,s,f)
  if s<f
     InsertionSort(A,s,f-1)
     for j=f down to s+1
        if A[j] < A[j-1]
           Swap A[j] and A[j-1]
        else break
```

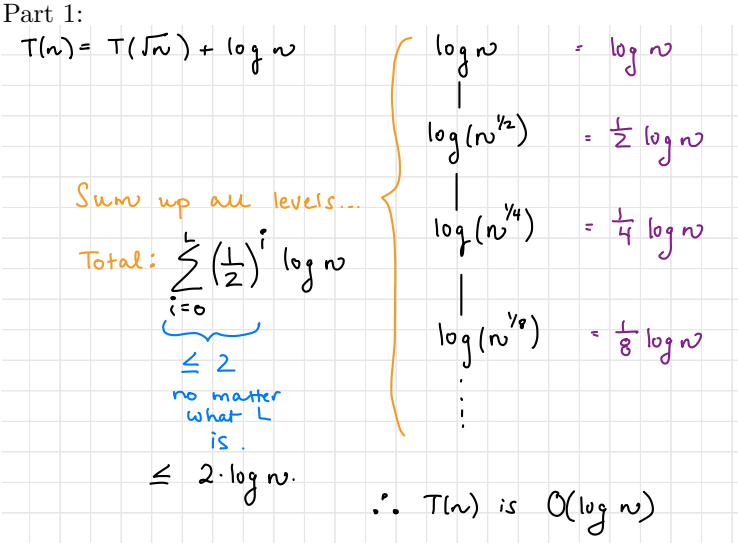
The worst-case runtime has recurrence $T(n) = T(n-1)+cn$. The substitution method is identical to that for SelectionSort, and therefore this algorithm also runs in time $O(n^2)$ in the worst-case.

Problem 8:

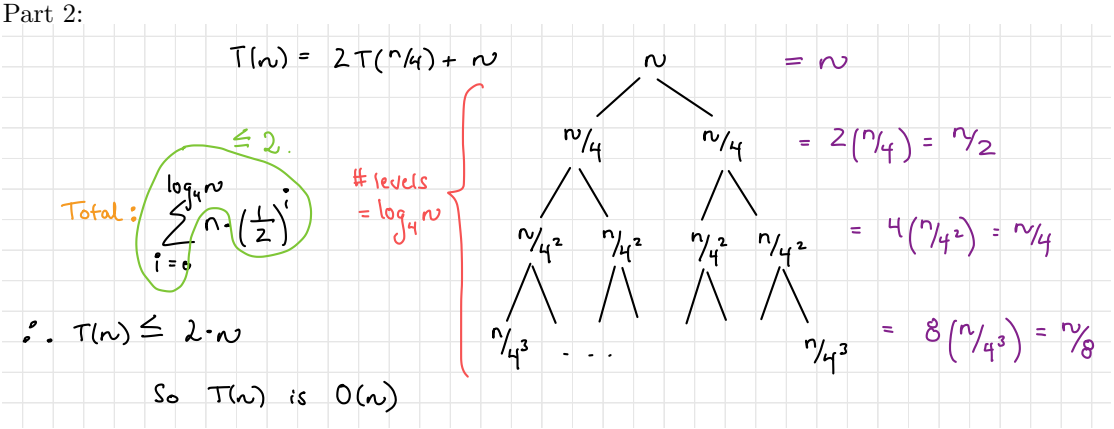
- $k = \log_{20/19} 1 = 0$. Therefore $n^k = 1$. And $f(n) = n^3$. By the master method, the solution is $\Theta(n^3)$.
- $k = \log_3(9) = 2$. Therefore $n^k = n^2$. And $f(n) = n^2 \log^5(n)$. Since $f(n)$ is not $O(n^{2-e})$ nor $\Omega(n^{n+e})$, the master method does not apply.
- $k = \log_3(10) = 2.09$. Therefore $n^k = n^{2.09}$. And $f(n) = n^4 \log(n)$. By the master method, the solution is $O(n^4 \log n)$.
- $k = \log_3(9) = 2$. Therefore $n^k = n^2$. And $f(n) = n^3 \log(n)$. In this case, $f(n)$ is $\Omega(n^{2+e})$, so $T(n)$ is $\Theta(n^3 \log n)$.

Problem 9:

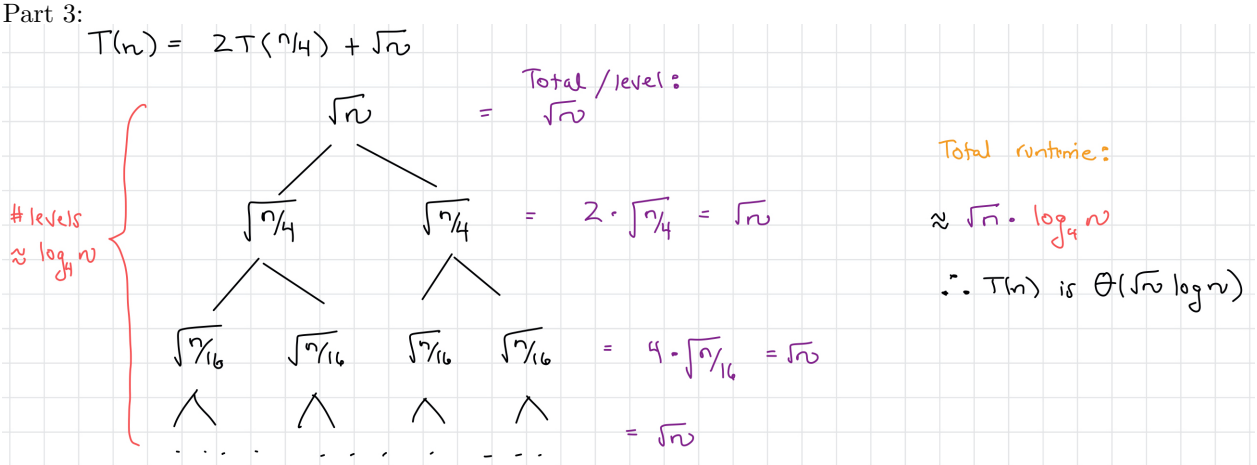
Part 1:



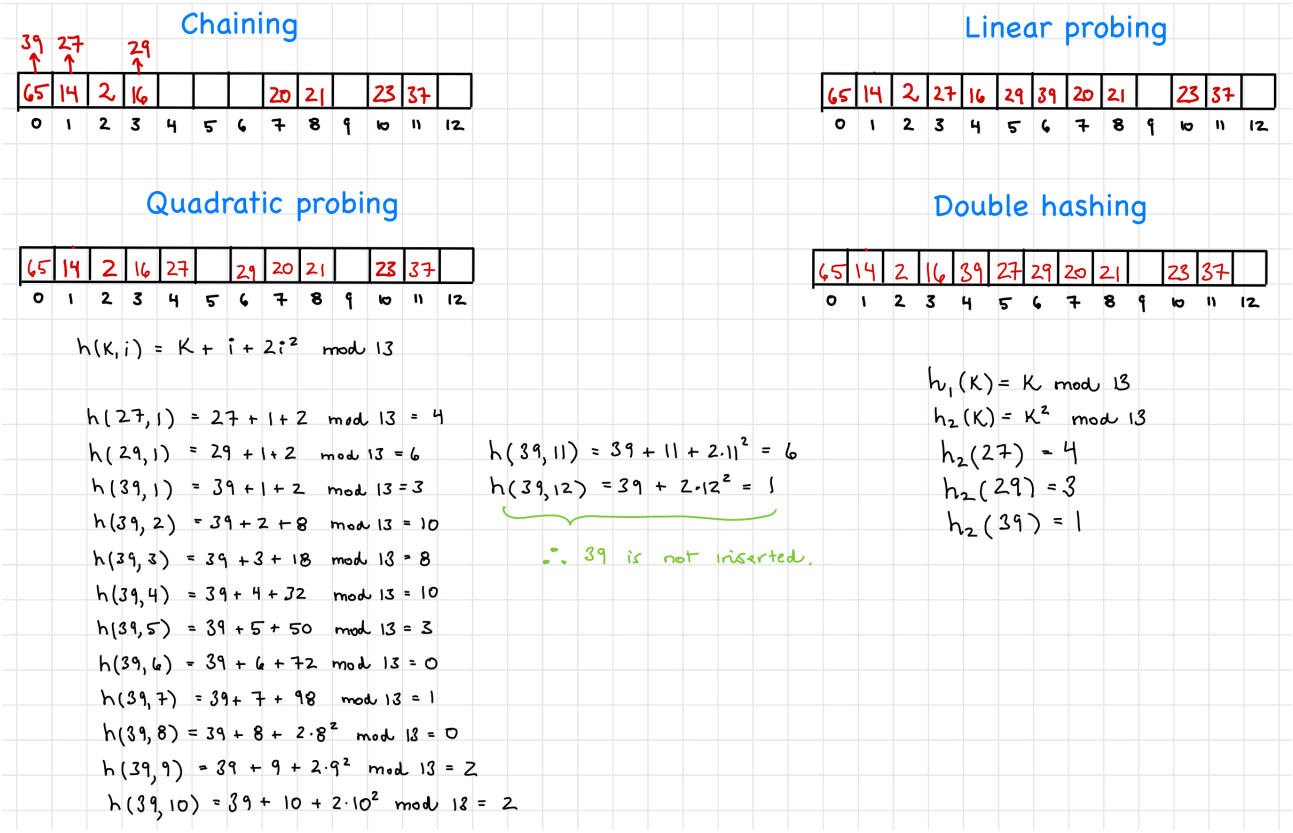
Part 2:



Part 3:



Problem 10:



Problem 11:

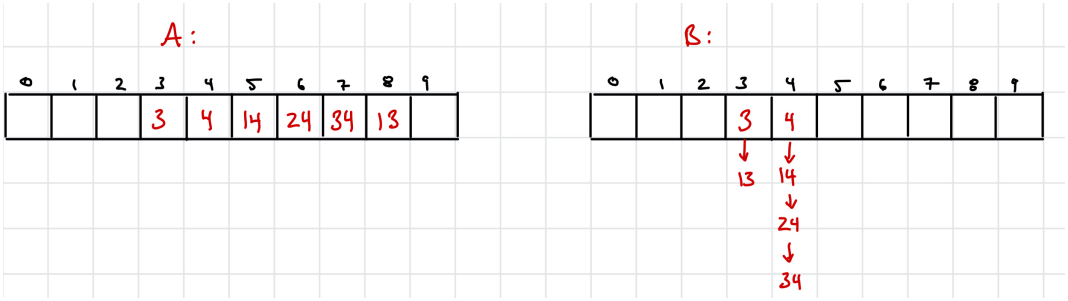
After 5 insertions, if there have been no collisions, it is because each key got hashed to a unique spot. After the first key gets inserted, the chance that the second key gets hashed to a different slot is $\frac{24}{25}$. The chance that the third gets hashed to a free spot is $\frac{23}{25}$, the chance that the fourth gets hashed to a free spot is $\frac{22}{25}$, and finally the chance that the fifth gets hashed to a free spot is $\frac{21}{25}$. Therefore the chance that all 5 keys get hashed to different slots is $(24 \cdot 23 \cdot 22 \cdot 21)/25^4 = 0.65$. Therefore there is a good chance that we have no chains at all in our table.

After all n insertions, we next consider the chance of having a chain of length n, which occurs when *all* keys got hashed to the same slot. We must determine the chance of this happening. After the first key gets inserted, the chance that the second key gets hashed to the exact same slot is $\frac{1}{25}$. Similarly for the third, fourth and fifth key. Therefore the chance that all n keys all hash to the same slot as the first is $(1/25)^{n-1}$, which is very unlikely.

The expected length of each chain is $n/25$, therefore we expect a search to take $O(n)$ time.

Problem 12:

Suppose we insert keys $3, 4, 14, 24, 34, 13$. The resulting table for each student in shown below. Note that the linear probing table ends up with a large cluster. In order to search for key 13, student A requires 6 probes, whereas student B only requires 2.



Problem 13:

The hash table for this question is $T[0, \ldots, n-1]$, and has n slots. If a uniform hash function is used, then keys are hashed randomly to one of the n slots. Therefore if we insert n keys into the table, the load factor is $\alpha = n/n = 1$. The expected length of each chain is $\Theta(1)$. The expected time for a successful/unsuccessful search is $\Theta(1)$, or constant. If we insert

$2n$ keys, $\alpha = 2n/n = 2$, and again, the expected chain length is $\Theta(2)$, results in an expected constant time for searching. If we insert n^2 keys, then $\alpha = n^2/n = n$, in which case the expected chain length is $\Theta(n)$ and therefore the expected search time is $O(n)$.

Next, if we use open addressing with a uniform probe sequence on a table with n slots and $\sqrt{n}$ keys, then the load factor is $\alpha = \sqrt{n}/n = 1/\sqrt{n}$. The expected time for an unsuccessful search (the worst-case) is $1/(1-1/\sqrt{n}) = \sqrt{n}/(\sqrt{n}-1)$, which is approximately 1. On the other hand, if we insert $n/2$ keys, the load factors is $\alpha = (n/2)/n = 1/2$, so the expected time for an unsuccessful search is $1/(1/2) = 2$, which is again, constant.