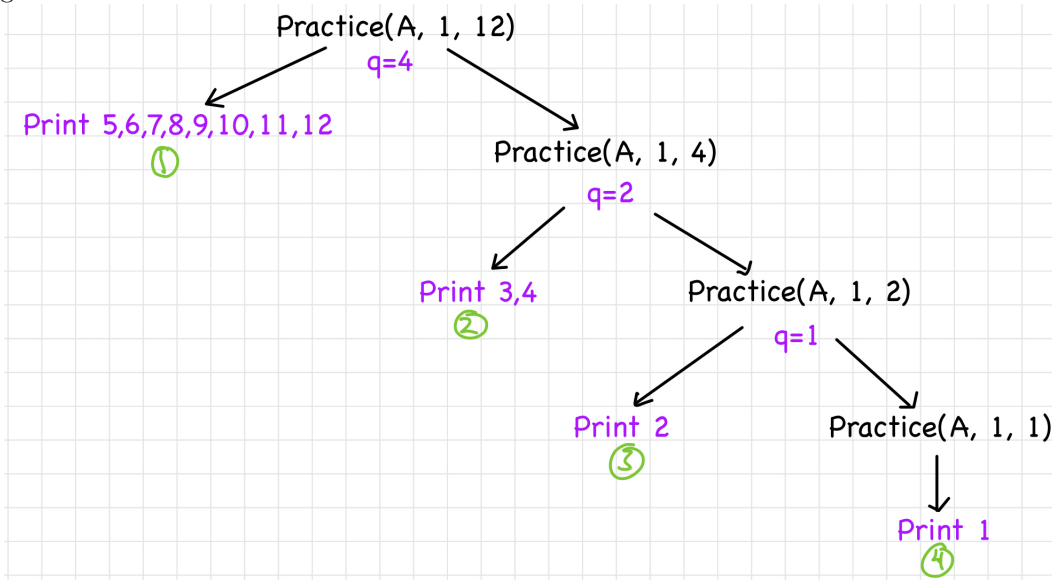# Practice Set 2: Solutions

Problem 1:

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 2:

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 3:

In the worst case, the algorithm performs a call to BSearch(A, s, q, k) and to MySearch$(A, q + 1, f, k)$. The call to BSearch(A, s, q, k) is on an array of size approximately $n/2$, and the call to MySearch$(A, q + 1, f, k)$ is on an array of size approximately $n/2$. The remaining operations run in constant time. Therefore the worst-case runtime recurrence is $T(n) = T(n/2) + c \log n/2 = T(n/2) + c \log n$ for a new constant c
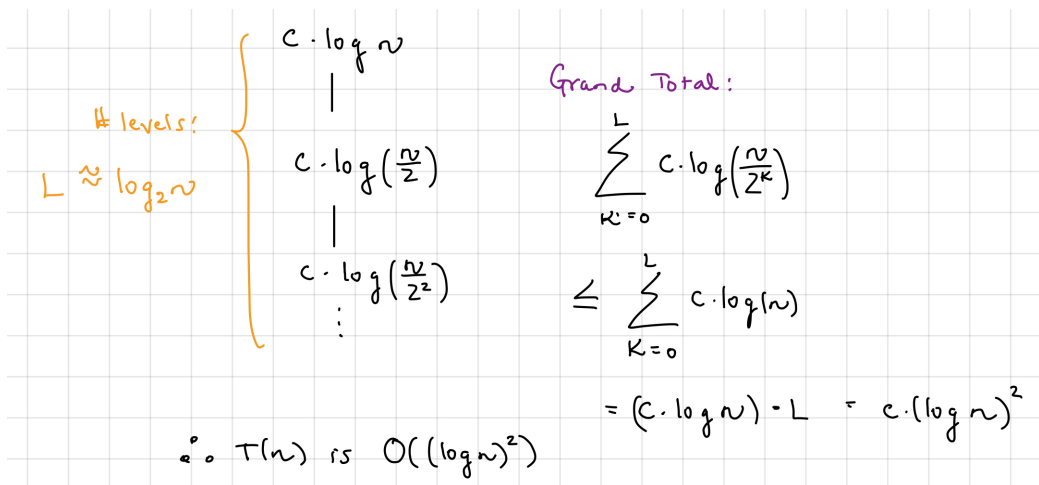
- *Substitution method:* Show that $T(n)$ is $O((\log n)^2)$.

  Goal:. Show that $T(n) \le d(\log n)^2$

  Assume:. $T(n/2) \le d(\log(n/2))^2$

  Substitute:. $T(n) = T(n/2) + c \log n \le d(\log(n/2))^2 + c \log n = d(\log n - 1)^2 + c \log n = d(\log n)^2 - 2d \log n + d + c \log n \le d(\log n)^2 - 2d \log n + d \log n + c \log n \le d(\log n)^2 + (c-d) \log n \le d(\log n)^2$ as long as $d \ge c$. Therefore $T(n)$ is $O((\log n)^2)$.

- The recursion tree:

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 $T(n) \le dn$ for some constant d.

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

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

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

Problem 5:

- Practice1 makes a call to Bsearch on an array of size $n/2$ and then in the worst-case, it also makes a recursive call on an array of size $n/2$. Therefore the recurrence is $T_1(n) = T_1(n/2) + c \log n$. The master method does not apply here, but we can use substitution method to show $T_1(n) \le O((\log n)^2)$.

  Goal: Show that $T_1(n) \le d(\log n)^2$

  Assume: $T_1(n/2) \le d(\log(n/2))^2 = d(\log n - 1)^2 \le d(\log n)^2 - 2d \log n + d \log n = d(\log n)^2 - d \log n$

  Substitue: $T_1(n) = T_1(n/2) + c \log n \le d(\log n)^2 - d \log(n) + c \log n = d(\log n)^2 - (d-c) \log n \le d(\log n)^2$ as long as $d \ge c$ and $n \ge 2$.

- Practice2 makes a call to BSearch on an array of size $n - 1$ and then in the worst-case, it also makes a recursive call to an array of size $n - 1$. Therefore the recurrence is $T_2(n) = T(n-1) + c \log(n-1)$. We cannot use Master's method here, so instead we use the substitution method to show that $T_2(n)$ is $O(n \log n)$. Note that $T_2(n) \le T_2(n-1) + c \log n$.

  Goal: Show that $T_2(n) \le dn \log n$

  Assume: $T_2(n-1) \le d(n-1) \log(n-1) \le d(n-1) \log n$

  Substitue: $T_2(n) \le T_2(n-1) + c \log n \le d(n-1) \log n + c \log n = dn \log n - d \log n + c \log n \le dn \log n$ as long as $d \ge c$.

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:

$T(n) = T(\sqrt{n}) + \log n$

$\log n \qquad = \log n$

$\log(n^{1/2}) \qquad = \frac{1}{2} \log n$

Sum up all levels...

Total: $\displaystyle\sum_{i=0}^{L} \left(\frac{1}{2}\right)^i \log n$

$\qquad \log(n^{1/4}) \qquad = \frac{1}{4} \log n$

$\qquad \le 2$ no matter what L is.

$\qquad \log(n^{1/8}) \qquad = \frac{1}{8} \log n$

$\qquad \le 2 \cdot \log n.$

$\therefore$ $T(n)$ is $O(\log n)$

Part 2:

$T(n) = 2T(n/4) + n$

$n \qquad = n$

Total: $\displaystyle\sum_{i=0}^{\log_4 n} n \cdot \left(\frac{1}{2}\right)^i \qquad \le 2.$

levels $= \log_4 n$

$n/4 \quad n/4 \qquad = 2(n/4) = n/2$

$n/4^2 \quad n/4^2 \quad n/4^2 \quad n/4^2 \qquad = 4(n/4^2) = n/4$

$\therefore$ $T(n) \le 2 \cdot n$

So $T(n)$ is $O(n)$

$n/4^3 \qquad \cdots \qquad n/4^3 \qquad = 8(n/4^3) = n/8$

Part 3:

$T(n) = 2T(n/4) + \sqrt{n}$

$\sqrt{n} \qquad = \sqrt{n}$

Total / level:

levels $\approx \log_4 n$

$\sqrt{n/4} \qquad \sqrt{n/4} \qquad = 2 \cdot \sqrt{n/4} = \sqrt{n}$

Total runtime:

$\approx \sqrt{n} \cdot \log_4 n$

$\therefore$ $T(n)$ is $\Theta(\sqrt{n} \log n)$

$\sqrt{n/16} \quad \sqrt{n/16} \quad \sqrt{n/16} \quad \sqrt{n/16} \qquad = 4 \cdot \sqrt{n/16} = \sqrt{n}$

$= \sqrt{n}$

Problem 10:

Chaining

39 27 29
29

65	14	2	16					20	21		23	37
0	1	2	3	4	5	6	7	8	9	10	11	12

Linear probing

65	14	2	27	16	29	39	20	21			23	37
0	1	2	3	4	5	6	7	8	9	10	11	12

Quadratic probing

65	14	2	16	27		29	20	21		23	37	
0	1	2	3	4	5	6	7	8	9	10	11	12

$h(K,i) = K + i + 2i^2 \mod 13$

$h(27,1) = 27 + 1 + 2 \mod 13 = 4$
$h(29,1) = 29 + 1 + 2 \mod 13 = 6$
$h(39,1) = 39 + 1 + 2 \mod 13 = 3$
$h(39,2) = 39 + 2 + 8 \mod 13 = 10$
$h(39,3) = 39 + 3 + 18 \mod 13 = 8$
$h(39,4) = 39 + 4 + 32 \mod 13 = 10$
$h(39,5) = 39 + 5 + 50 \mod 13 = 3$
$h(39,6) = 39 + 6 + 72 \mod 13 = 0$
$h(39,7) = 39 + 7 + 98 \mod 13 = 1$
$h(39,8) = 39 + 8 + 2 \cdot 8^2 \mod 13 = 0$
$h(39,9) = 39 + 9 + 2 \cdot 9^2 \mod 13 = 2$
$h(39,10) = 39 + 10 + 2 \cdot 10^2 \mod 13 = 2$

$h(39,11) = 39 + 11 + 2 \cdot 11^2 = 6$
$h(39,12) = 39 + 2 \cdot 12^2 = 1$

∴ 39 is not inserted.

Double hashing

65	14	2	16	39	27	29	20	21			23	37
0	1	2	3	4	5	6	7	8	9	10	11	12

$h_1(K) = K \mod 13$
$h_2(K) = K^2 \mod 13$
$h_2(27) = 4$
$h_2(29) = 3$
$h_2(39) = 1$

Problem 11:

After 10 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 $\frac{99}{100}$. The chance that the third gets hashed to a free spot is $\frac{98}{100}$, the chance that the fourth gets hashed to a free spot is $\frac{97}{100}$, etc. Therefore the chance that all 10 keys get hashed to different slots is $99 \cdot 98 \cdot 97 \cdots 91/(100)^9 = 0.63$. 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 10, 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 $\frac{1}{100}$. Similarly for the third, fourth and fifth key. Therefore the chance that all 10 keys all hash to the same slot as the first is $(1/100)^9$, which is very unlikely.

The expected length of each chain is $n/100 = 10/100 = 1/10$, therefore we expect a search to take $O(1)$ 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.

A:

0	1	2	3	4	5	6	7	8	9
			3	4	14	24	34	13	

B:

0	1	2	3	4	5	6	7	8	9
			3	4					

3 → 13
4 → 14 → 24 → 34

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.