Problem Set 2 Solution

All parts are due Thursday, October 13 at 11:59PM.

Name: Jisoo Min

Collaborators:

Part A

Problem 2-1. [10 points] Back-solving recurrences

(a) Solution: \( a = 4 \)

Recurrence: \( T(n) = aT(n/2) + \Theta(n^2) \ (a \in \mathbb{N}); T(n) = \Theta(n^2 \lg n) \).

Only case 2 of the master theorem can produce this asymptotic bound. Case 2 states that if \( f(n) = \Theta(n^\log_a 2) \), then \( T(n) = \Theta(n^\log_a 2 \lg n) \). To match the result, we get \( a = 4 \). We need to make sure that the condition \( f(n) = \Theta(n^\log_a 2) \) can be satisfied with \( a = 4 \). We are given \( f(n) = \Theta(n^2) \) in the problem, so the condition can be satisfied and case 2 works. Therefore, \( a = 4 \) is valid.

(b) Solution: \( a = 9 \)

Recurrence: \( T(n) = aT(n/3) + \Theta(n) \ (a \in \mathbb{N}); T(n) = \Theta(n^2) \).

Only case 1 of the master theorem can produce this asymptotic bound. Case 1 states that if \( f(n) = \Theta(n^\log_3 a - \epsilon) \) for some constant \( \epsilon > 0 \), then \( T(n) = \Theta(n^\log_3 a) \). To match the result, we get \( a = 9 \). We need to make sure that the condition \( f(n) = \Theta(n^\log_3 a - \epsilon) \) can be satisfied with \( a = 9 \). We are given \( f(n) = \Theta(n^2) \) in the problem, so the condition can be satisfied and case 1 works. Therefore, \( a = 9 \) is valid.

(c) Solution: \( b > 2 \)

Recurrence: \( T(n) = 4T(n/b) + \Theta(n^2) \ (b \in \mathbb{R}, b > 0); T(n) = \Theta(n^2) \).

Both case 1 and case 3 of the master theorem might produce this asymptotic bound. First, let’s assume that case 1 produced the bound. Case 1 states that if \( f(n) = \Theta(n^{\log_b 4 - \epsilon}) \) for some constant \( \epsilon > 0 \), then \( T(n) = \Theta(n^{\log_b 4}) \). To match the result, we get \( b = 2 \). We need to make sure that the condition \( f(n) = \Theta(n^{\log_b 4 - \epsilon}) \) can be satisfied with \( b = 2 \). We are given \( f(n) = \Theta(n^2) \) in the problem, so \( f(n) = O(n^{2-\epsilon}) \) cannot hold true. Case 1 does not work. Therefore, \( b = 2 \) is invalid.
Now let’s assume that case 3 produced the bound. Case 3 states that if \( f(n) = \Omega(n^{\log_b 4+\epsilon}) \) for some constant \( \epsilon > 0 \), and if \( 4f(n/b) \leq cf(n) \) for some constant \( c < 1 \) and all sufficiently large \( n \), then \( T(n) = \Theta(f(n)) \). The result is already matched. We need to make sure that first part of the condition \( f(n) = \Omega(n^{\log_b 4+\epsilon})(\epsilon > 0) \) can be satisfied. We are given \( f(n) \in \Theta(n^2) \) in the problem, so \( f(n) = \Omega(n^{\log_b 4+\epsilon})(\epsilon > 0) \) when \( b > 2 \). Because \( b > 2 \) and \( 4f(n/b) < 4f(n/2) \), we see that \( 4f(n/2) \leq cf(n) \) is true for some constant \( c < 1 \) and all sufficiently large \( n \). So the condition can be satisfied and case 3 works. Therefore, \( b > 2 \) (all real numbers greater than 2) is invalid.

(d) Solution: \( b = 5^{\frac{1}{\log 5}} \)

Recurrence: \( T(n) = 5T(n/b) + \Theta(n^5) \) \( (b \in \mathbb{R}, b > 0) \); \( T(n) = \Theta(n^{6.006}) \).

Only case 1 of the master theorem can produce this asymptotic bound. Case 1 states that if \( f(n) = \Theta(n^{\log_b 5 - \epsilon}) \) for some constant \( \epsilon > 0 \), then \( T(n) = \Theta(n^{\log_b 5}) \). To match the result, we get \( b = 5^{\frac{1}{\log 5}} \). We need to make sure that the condition \( f(n) = \Theta(n^{\log_b 5 - \epsilon})(\epsilon > 0) \) can be satisfied with \( b = 5^{\frac{1}{\log 5}} \). We are given \( f(n) \in \Theta(n^5) \) in the problem, so the condition can be satisfied and case 1 works. Therefore, \( b = 5^{\frac{1}{\log 5}} \) is valid.

(e) Solution: \( f = n^2 \)

Recurrence: \( T(n) = 6T(n/6) + f(n) \); \( T(n) = \Theta(n^2) \).

Only case 3 of the master theorem can produce this asymptotic bound. Case 3 states that if \( f(n) = \Omega(n^{\log_b 6+\epsilon}) \) for some constant \( \epsilon > 0 \), and if \( 6f(n/6) \leq cf(n) \) for some constant \( c < 1 \) and all sufficiently large \( n \), then \( T(n) = \Theta(f(n)) \). To match the result, we get \( \Theta(f(n)) = \Theta(n^2) \). Therefore, \( f = n^2 \) is valid.

**Problem 2-2.** [20 points] Sorting a Rectangle

Solution:
Given a \( n \times m \) 2D array of integers \( A \) \( (m \ll n) \) with all integers sorted within each row and column, we can produce a length-\( nm \) sorted 1D array of \( A \) in \( O(nm \log m) \).

First, we initialize the sorted 1D array output as an empty array. Keeping track of the column and row of origin, we build a min-heap with integers from the first row of \( A \). For \( nm \) iterations, we do the following: 1. Extract minimum from the min-heap and add it to the output array. Say this element was from row \( n' \) and column \( m' \). 2. If \( n' \neq n \), then add \( (A[n'+1][m'], n'+1, m) \) to the heap. If \( n' = n \), we do not add anything to the heap.

Both extracting a minimum from the min-heap and inserting a new item to the min-heap is \( O(\log m) \). We have \( nm \) iterations of this operation. Therefore the asymptotic running time behavior of the algorithm is \( O(nm (\log m+\log m)) \), which simplifies to \( O(nm \log m) \).
Figure 1: Diagram of how sort_rectangle algorithm works

First few iterations are shown in the diagram above. Once all elements from a column is used in the output array, the heap size reduces. This algorithm takes care of all cases without errors, because we go through exactly $nm$ iterations and make sure to add the correct minimum value to the output array. The minimum from the remaining integers in the array is guaranteed to be in the min-heap because all rows and columns are given sorted.

```python
import heapq

# 2D array A with n rows and m columns
def sort_rectangle(A, n, m):
    # start the heap with elements of the first row
    look_up_heap = [(A[1, j], 1, j) for j in range(m)]
    sorted_1d_array = []
    for i in range(n * m):
        element, n0, m0 = heapq.heappop(look_up_heap) # O(lg m)
        sorted_1d_array.append((element, n0, m0))
        if n0 != n:
            heapq.heappush(look_up_heap, (A[n0 + 1][m0], n0 + 1, m0)) # O(lg m)
    return sorted_1d_array
```

Problem 2-3. [25 points] Sorting Venture Capital Offers for 6006LE

(a) Solution:

In the given length-$n$ parking_lot indexed 1, 2, ..., $n$, with values $a_1, a_2, ..., a_n$, we can only compare and swap values within $k$ distance apart in the parking_lot. ($a_1, a_2, ..., a_n$ are constants that do not change.) Under this condition, we are able to switch two arbitrary values $a_i$ and $a_j$, and return all other values to their original positions in $O(n/k)$ swaps. Say $d = |i - j|$ is the distance between the two arbitrary trucks given. (Assume $i < j$)
With $2\lceil \frac{d}{k} \rceil - 1$ swaps, we can switch two arbitrary values and return all other values to their original positions. We start from position $j$. Think as if $a_j$ is walking down some stepping stones. Switch this element with the element $k$ distance to the left. And repeat the process until the distance between $a_j$ and $a_i$ is less than or equal to $k$. Once we are at this point, we are able to swap $a_j$ and $a_i$. We now have $a_j$ in position $i$ and all other swapped elements shifted to the right with order preserved. Similarly we now start from where $a_i$ is and have it switch and walk up the stepping stones to where $a_j$ was originally. We have switched the values in positions $i$ and $j$ and returned any other involved values to their original positions.

**Switch $a_2$ and $a_{12}$ ($k = 3$)**

**Original:**

\[
a_1 \circled{a_2} a_3 a_4 a_5 a_6 a_7 a_8 a_9 a_{10} a_{11} a_{12} a_{13} a_{14} a_{15}
\]

**We bring $a_{12}$ to position 2.**

\[
\begin{align*}
& a_1 a_2 a_3 a_4 a_5 a_6 a_7 a_8 \circled{a_{12}} a_{10} a_{11} a_9 a_{13} a_{14} a_{15} \\
& a_1 a_2 a_3 a_4 a_5 a_{12} a_7 a_8 a_6 a_{10} a_{11} a_9 a_{13} a_{14} a_{15} \\
& a_1 a_2 a_{12} a_4 a_5 a_3 a_7 a_8 a_6 a_{10} a_{11} a_9 a_{13} a_{14} a_{15} \\
& a_1 a_{12} a_2 a_4 a_5 a_3 a_7 a_8 a_6 a_{10} a_{11} a_9 a_{13} a_{14} a_{15}
\end{align*}
\]

\[a_2 \ , \ a_3 \ , \ a_6 \ , \ a_9 \ \text{are moved.}\]

**We now bring $a_2$ to position 12.**

\[
\begin{align*}
& a_1 a_{12} \circled{a_3} a_4 a_5 a_2 a_7 a_8 a_6 a_{10} a_{11} a_9 a_{13} a_{14} a_{15} \\
& a_1 a_{12} a_3 a_4 a_5 a_6 a_7 a_8 a_2 a_{10} a_{11} a_9 a_{13} a_{14} a_{15} \\
& a_1 a_{12} a_3 a_4 a_5 a_6 a_7 a_8 \circled{a_9} a_{10} a_{11} a_2 a_{13} a_{14} a_{15} \\
& a_1 a_{12} a_3 a_4 a_5 a_6 a_7 a_8 \circled{a_9} a_{10} a_{11} a_2 a_{13} a_{14} a_{15}
\end{align*}
\]

\[a_2 \ , \ a_3 \ , \ a_6 \ , \ a_9 \ \text{are back in place.}\]

**Figure 2:** Diagram of how `switch_two` algorithm works

This algorithm takes care of all cases without errors, because we maintain the direction of swapping and go through minimum number of iterations ($2\lceil \frac{d}{k} \rceil - 1$) to switch elements in $i$ and $j$ positions. We are always switching elements exactly $k$ distance apart except for at most once because $d$ may not be divisible by $k$. For distance $d$ apart elements, we need $2\lceil \frac{d}{k} \rceil - 1$ swaps. $\lceil \frac{d}{k} \rceil$ swaps are used to bring $a_j$ to position $i$, and $\lceil \frac{d}{k} \rceil - 1$ swaps are used to bring $a_i$ to position $j$. Because the maximum value of $d$ is $n$, we are bound to $O(n/k)$ swaps for a given $k$ constraint.
# we want parking_lot[i] = a_j, parking_lot[j] = a_i

def switch_two(parking_lot, i, j, k):
    d = |i - j|
    if d ≤ k:
        parking_lot[i], parking_lot[j] = parking_lot[j], parking_lot[i]
        return
    current_pos = j
    remaining_dist = d
    while remaining_dist != 0:
        swap_dist = min(remaining_dist, k)
        parking_lot[current_pos], parking_lot[current_pos - swap_dist]
        = parking_lot[current_pos - swap_dist], parking_lot[current_pos]
        current_pos -= swap_dist
        remaining_dist -= swap_dist
        current_pos = i + d\%k
        remaining_dist = d - d\%k
        if d\%k == 0:
            current_pos += k
            remaining_dist -= k
        while remaining_dist != 0:
            swap_dist = min(remaining_dist, k)
            parking_lot[current_pos], parking_lot[current_pos + swap_dist]
            = parking_lot[current_pos + swap_dist], parking_lot[current_pos]
            current_pos += swap_dist
            remaining_dist -= swap_dist

(b) Solution:

In the given length-n parking_lot indexed 1, 2, ..., n, with values a_1, a_2, ..., a_n, we can only compare and swap values within k distance apart in the parking_lot. (a_1, a_2, ..., a_n are constants that do not change.) Under this condition, we are able to compare two arbitrary values a_i and a_j, and return all values to their original positions in O(n/k) swaps. Say d = |i - j| is the distance between the two arbitrary trucks given.

We are guaranteed that we can compare two adjacent values. If d = 1, we compare the values and return. If d ≠ 1, we switch a_{i+1} and a_j so that a_i and a_j are adjacent. (Assume i < j)

With 2\left\lceil \frac{d-1}{k} \right\rceil swaps, we can compare two arbitrary values and return all other values to their original positions. We start from position j. Think as if a_j is walking down some stepping stones. Switch this element with the element k distance to the left. And repeat the process until the distance between a_j and a_{i+1} is less than or equal to
Once we are at this point, we are able to swap \(a_j\) and \(a_{i+1}\). We now have \(a_j\) in position \(i+1\) and all other swapped elements shifted to the right with order preserved. Now that \(a_i\) and \(a_j\) are adjacent to one another, we can compare them. We now make \(a_j\) go back to its original position \(j\). This takes \(\lceil \frac{d-1}{k} \rceil\) swaps again. We will swap the exact same elements we swapped when bringing \(a_j\) to position \(i+1\).

**Compare \(a_1\) and \(a_{12}\) \((k = 3)\)**

**Original:**

\[
\begin{array}{cccccccccccc}
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 \\
\end{array}
\]

We bring \(a_{12}\) to position 2 so that \(a_1\) and \(a_{12}\) are adjacent.

\[
\begin{array}{cccccccccccc}
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 \\
\end{array}
\]

Compare \(a_1\) and \(a_{12}\).

**We now bring \(a_2\) back to position 12.**

\[
\begin{array}{cccccccccccc}
1 & 2 & 12 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 13 & 14 & 15 \\
\end{array}
\]

Everything is back in place.

**Figure 3:** Diagram of how this algorithm of comparing two works

This algorithm takes care of all cases without errors, because we can switch two arbitrary elements in \(O(n/k)\) swaps (see part (a)) and because we can always compare two adjacent elements. We are switching elements exactly \(k\) distance apart except for at most once, because \(d - 1\) may not be divisible by \(k\). For distance \(d\) apart elements, we need \(2\lceil \frac{d-1}{k} \rceil\) swaps to compare them. \(\lceil \frac{d-1}{k} \rceil\) swaps are used to bring \(a_j\) to position \(j\), and another \(\lceil \frac{d-1}{k} \rceil\) swaps are needed to reverse that. Because the maximum value of \(d\) is \(n\), we are again bound to \(O(n/k)\) swaps for a given \(k\) constraint.

**c) Solution:**

In the given length-\(n\) `parking_lot` indexed 1, 2, ..., \(n\), with values \(a_1, a_2, ..., a_n\), we can only compare and swap values within \(k\) distance apart in the `parking_lot`. \((a_1, a_2, ..., a_n\) are constants that do not change.) Under this condition, we are able to sort the values
in $O\left(\frac{n^2}{k} \cdot n \log n\right)$ swaps.

We build a min heap with the values in parking_lot and use heapsort. Heaps require switching values between parent and child. Because we are restricted to compare and swap elements only $k$ distance apart, we need to modify the heap functions.

Building a heap would require $O(n)$ without any constraints. With our $k$ distance constraint, we will have to use $O(n/k)$ swaps (see part(a)) for all switches that are over $k$ distance. Therefore, building the heap requires $O(n \cdot n/k)$. (To be a little more precise, there are approximately $4n$ cases that require $O(n/k)$ swaps.)

While sorting the heap, we need to trickle down items. Again, we face the problem of switching items over $k$ distance. We will have to use $O(n/k)$ swaps (see part(a)) for all switches that are over $k$ distance. Therefore, sorting the heap requires $O((n \log n) \cdot (n/k))$. (To be a little more precise, there are approximately $4n$ cases that require $O(n/k)$ swaps on top of regular trickle downs.) The final asymptotic running time behavior of sorting the values with a min heap is $O((n^2 \log n)/k)$.

We have handled all cases that require multiple swaps and used a min heap to reduce the runtime complexity in finding the minimum. Therefore, this algorithm works.

(d) Solution:

In the given length-$n$ parking_lot indexed 1, 2, ..., $n$, with values $a_1, a_2, ..., a_n$, we can only compare and swap values within $k$ distance apart in the parking_lot. ($a_1, a_2, ..., a_n$ are constants that do not change.) Under this condition, we are able to sort the values in $O\left(\frac{n^2}{k} \cdot n \log k\right)$ swaps.

We sort $k$ elements in index 1, 2, ..., $k$ using heapsort. Then we sort another set of $k$ elements in index $[k/2], [k/2] + 1, ..., k + [k/2]$. We continue this process of sorting $k$ elements as we shift our window over by $k/2$ until we get to the end of the given values. Each of this heapsort takes $O(k \log k)$. Now we are guaranteed to have the last $[k/2]$ to be the greatest $[k/2]$ elements of the parking lot. Only the first $n - [k/2]$ elements are unsorted.

We can write the recurrence as $T(n) = T(n - \frac{k}{2}) + \frac{2n}{k} \cdot O(k \log k)$.

Expanding out and solving the recurrence, we get $T(n) = (2n/k)(2n/k)(1/2) \cdot O(k \log k) = O\left(\frac{n^2}{k} \cdot n \log k\right)$. 

**Problem 2-4.**

(a) **Solution:**

Bowser is in the F1 Kart Racing Finals on a circular track. A competitor loses when another competitor from behind passes by. To get Bowser’s rank in $O(N^2)$, we calculate the losetimes of all combinations of the players. For each player $i$, we calculate the losetimes of $i$ with other competitors. Finding the minimum among these takes $O(N)$. We also check to see if both the loser and the catcher is in the game. If yes, we update `ahead` and `behind` dictionaries accordingly. Checking takes constant time. We may have to repeat this process $O(N)$ times until we find out Bowser’s rank. Therefore, the growth order is $O(N^2)$.  

---

**Figure 4:** Example of using the algorithm on $n=16$, $k=4$
(b) Submitted on alg.csail.mit.edu

(c) Submitted on alg.csail.mit.edu

(d) Submitted on alg.csail.mit.edu

(e) Solution:

- Events to add:
  - If $v_c > v_a$, then add the event of Charlie catching Alice.
- Events to remove:
  - Charlie caught Bowser; remove the event of Charlie catching Bowser.
  - Bowser is no longer in the game; remove the event of Bowser catching Alice.

(f) Submitted on alg.csail.mit.edu