CSCI 665
HW#2
Aditya Ajit Tirakannavar (at2650@rit.edu)
Aniket Narendra Patil (ap8504@rit.edu)

Problem 1:

a) The recurrence function which captured the running time is:
T = O (1) if (left == right)
T = 2* T (n/2) + O (1) if (left != right)

b)
T (n) <= 2*T (n/2) + d
T (n) <= 2 * (2*T (n/4) + d) + d
T (n) <= 4*T (n/4) + 3d
T (n) = 4 * (2*T (n/8) + d) + 3d
T (n) = 8* T (n/8) + 7d
Stop when n = 2^k
T (n) = n*T (1) + (2^k - 1)*d
T (n) =(n + (n - 1))*d
T (n) <= (2n-1)*d
T (n) = O (n)

c)
This algorithm recursively divides the index values for left indexes till the mid-value is equal to the left value which is 0 for the first recursive call and then it does the same for right indexes, then it recursively keeps calculating the Maxsum, Left aligned max sum, right-aligned max sum, and sum, by iterating through all the elements in the list. For the negative element, it returns 0,0,0 and the negative value and for the positive value, it returns the same positive value eg. 3,3,3,3.

This Algorithm returns 4 values Maxsum, Left aligned max sum, right-aligned max sum, and sum

- Maxsum: This value is Maxmum sum that can be achieved from any of the given below sub-list

  (lmaxsum, llmaxsum, lrmaxsum, lsum) = WHATDOIDO(left, m)
  (rmaxsum, rlmaxsum, rrmaxsum, rsum) = WHATDOIDO(m + 1, right)

- Left aligned max sum: This value is the sum of all elements until a positive number is reached from the left

  leftalignedmaxsum = max(llmaxsum, lsum + rlmaxsum)

- Right aligned max sum: This value is the sum of all elements until a positive number is reached from the right

  rightalignedmaxsum = max(rrmaxsum, lrmaxsum + rsum)

- Sum: This is the sum of all elements of the array.

  sum = lsum + rsum

Problem 3:

Algorithm (Merged sort with counting Inversion):

Merge-Sort Algorithm was taught by Prof. Maria Cepeda in CSCI 603 and our solution heavily relies on the usage of that particular algorithm.

- Take the User input from *std.in* and store it in list in the appropriate datatype which is *int* for age and *float* for height
- Divide the entire list by recursively splitting the list into two equal half.
- Merge two halves according to condition given by the photographer as follows Condition for left and right pointer for merging two halves
  - If both students are 7 years old. Compare their height and place in ascending order of height.
  - If both students are 8 years old. Compare their height and place in descending order of height.
  - If one student is of 7/8 years old and other one is teacher, place the student before the teacher
  - If both students are of different ages, compare their age and place in ascending order of age.

- While merging two halves according to the photographer's requirement, we calculate the number of swaps to place a person in their current position.
- To calculate the swap count we use the Counting inversion algorithm which is as follows:
    - All the points are with reference to left and right pointer in two halves.
    - If the element in the right part is to be placed then we need to calculate swap count as the length of left - current left pointer.
    - If the element in the left part is to be placed then we don't need to calculate the swap count.
- The swap count is combined and the sum of each swap count will give us the entire swaps count performed for placing everyone in right place.

Data Structure Used:
Python Inbuilt List: Complexity and Description of operation performed:

- Access element at particular index: O(1)
- Append Element: O(1)

Overall Complexity of Algorithm:

- Append 'n' people in the list: O(n)
- Divide list recursively in two equal halves: O(logn)
- Merge list and calculate swap count for every merge: O(n)
- Merge List and calculate swap count Log(n) times: O(n*logn)
- Overall complexity for the algorithm: O(n*logn) (Merge-Sort's complexity)

Correctness:

- To Solve the Problem we have used Divide and Conquer approach.
- In order to find the number of swaps to be performed to place students and a teacher according to the photographer's requirement, we are recursively dividing the problem and we then independently solve each problem and merge them.
- Here we divide the problem the same way we divide in merge sort and merge data following the condition by photographer and compute the swap count by not actually swapping but calculating the number of swap counts that will be performed to place an element at the correct position which is nothing but the counting inversion.
- Since we divide the problem recursively, complexity is O(logn), and to merge the element and calculate the swap count it takes O(n), hence the overall complexity turns out to be O(n*logn).

Problem 4:

Algorithm:

- We can solve the problem using a heap sort algorithm has a time complexity of n * log n for both addition and extraction operation.
- We initialize a global time starting at 0 (start of the day) and progressing towards a specified time (end of the day) The task with the earliest arrival time is allowed to be executed first out of the given n tasks. If the arrival time of the task matches the current time, then the task is inserted in the heap.
- Similarly, all tasks having the arrival time equal to the current time, we add those tasks into the heap. Every task is simulated for a predefined unit of time.
- Then we check the below-mentioned scenarios one after the other and perform the appropriate action:
- If for all the tasks in the heap, if the deadline is less than or equal to zero and the running time or time required to complete is greater than 0, then that indicates the deadline for one of the tasks has expired and we cannot complete all tasks within the given deadline, hence we return NO.
- Check if the current task is finished, if task is finished then remove it from the heap, else simulate the same task again.
- Check if any other task has an arrival time equal to the current time, if yes, then add that task into a heap and begin working on that task by simulating it since it has a higher priority.
- Every time a task is simulated, its time required to complete is decremented by 1 and the deadline for all the tasks is also decremented by 1.
- If all tasks are completed within their specified deadline, then we return YES.

Data Structure:
Max Heap:
Add operation: O(log n)
Add operation for n elements: n * O(log n)

Extract operation: O(log n)
Extract operation for n elements: n * O(log n)

Correctness argument:

- We can solve the above problem using a heap data structure.
- For every task, we can prioritize it based on the time of arrival.
- We use a max heap such that, the task with the largest arrival time is given highest priority.
- A task can be executed by adding into the heap and allowing it to run to for unit time while checking if there exists another task with higher priority which is then executed by pausing the current task we were executing.
- Once the task is completed, it can be removed from the heap and the task having priority just below it is allowed to execute.
- This process can be repeated for all the tasks in the input list.
- If for a task, a deadline has passed while it was waiting to be executed, we can stop simulation of all tasks and return a NO, indicating there exists some task in the heap which was supposed to be executed by now but hasn't indicated a missed deadline.

Problem 5:

Algorithm:

In order to check if the placement of queens is valid or not we need to check whether each queen when placed nxn board does not attack other queens.

- Take the User input from *std.in* and store it in the list of rows and columns' position.
- Initialize 4 python lists with O (N) space complexity to achieve linear time complexity in conflict checking. Four 1-D tables (or arrays) are introduced to store the frequency of queens and also check if there are repeating frequencies to determine the validity of the placements. One for storing the frequency of queens in each row and similarly one for the column. One for storing the frequency of queens in left diagonals and another one for right diagonals. List for the diagonals will be of size 2*N as the sum of the rows and columns can reach a maximum value of 2*N
- Now we check every input row value and store the value in the row frequency list and check if all row values are unique and if any of the row values are the same, which indicates queens are in attacking position.
- Similarly, for column values we can check the validity in the same way.
- Further we check if the sum of the row and column are unique for checking if queens are not attacking diagonally (left).
- And for the right diagonal, we take the sum of column value and board size subtracted by row value (this is done to invert the row which intends to simulate the right diagonal) and check if the values are unique.
- If all values are unique, print 'YES' else print 'NO'

Data Structure Used:
Python Inbuilt List: Complexity and Description of operation performed:

- Access /checking element at particular index: O(1)
- Initialize list of size (N/2N): O(n)

Overall Complexity of Algorithm:

- Initialize list of size 2N and N: O(n)
- Check for unique values in row list: O(n)
- Check for unique values in column list: O(n)
- Check for unique values in the left diagonal list: O(n)
- Check for unique values in the right diagonal list: O(n)
- Overall complexity for the algorithm: O(n)

Correctness:

- To Solve the Problem in O (n) time, we cannot simulate the chessboard using a 2-D list because that would have taken O (n^2) complexity. Instead, we use four 1-D lists which directly simulate a queen's moves in all 8 directions.
- Using these 4 List, one for row check, column check, and diagonal checks, we can check the validity of positions.
- Since we use 1-D lists, the complexity of checking a valid position is just O(l) and as there are N queens, overall complexity is no more than O(n).

Submitted by:
Aniket Narendra Patil (ap8504)
Aditya Ajit Tirakannavar (at2650)

Q.2]

(a) $T(n) = 7T(n/8) + n$

Sol: $f(n) = n$
$a = 7$
$b = 8$

Using Master Theorm:

$\log_b a = \log_8 7 \Rightarrow$ 8 to what power is 7

$8^1 = 8$

$\therefore 8^x ? 7 \quad \therefore x < 1$

$\therefore \log_8 7 < 1$

$\therefore n^x < n^1$

$\log_b a < f(n)$

$\therefore T(n) = \Theta(f(n))$

$\boxed{\therefore T(n) = \Theta(n)}$

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

Sol: $f(n) = \sqrt{n} = n^{\frac{1}{2}}$
$a = 2$
$b = 2$

Using Master Theorm:

$\log_b a = \log_2 2 \Rightarrow$ 2 to what power is 2

$2^1 = 2$

$\therefore \log_2 2 = 1$

$\therefore n^{\log_2 2} = n^1$

Submitted by
Aniket Narendra Patil (ap8504)
Aditya Ajit Tirakannavar (at2650)

$\therefore n^1$ vs $n^{\frac{1}{2}}$

$f(n)$ vs $\log_2 2 \Rightarrow f(n) = \log_2 2$

$\therefore T(n) = \Theta(n^{\log_b a})$

$$\boxed{\therefore T(n) = \Theta(n)}$$

(c) $T(n) = 8T(n/2) + n^2$

Sol: $f(n) = n^2$

$a = 8$

$b = 2$

Using Masters theorem:

$\log_b a = \log_2 8 \Rightarrow$ 2 to what power is 8

$2^3 = 8$

$\therefore \log_2 8 = 3$

$\therefore n^3$ vs n^2

$\log_b a > f(n) \Rightarrow \log_2 8 > f(n)$

$\therefore T(n) = \Theta(n^{\log_b a})$

$$\boxed{\therefore T(n) = \Theta(n^3)}$$

Submitted by
Aniket Narendra Patil (ap8504)
Aditya Ajit Tirakamnavar (at2650)

(d) $T(n) = 3T\left(\frac{n}{9}\right) + \sqrt{n}$.

sol: $f(n) = \sqrt{n} = n^{\frac{1}{2}}$

$a = 3$

$b = 9$

$\log_b a = \log_9 3 \Rightarrow$ 9 to what power is 3

$9^x = 3$

$(3^2)^x = 3^1$

$2x = 1$

$x = \frac{1}{2}$.

$\therefore \log_9 3 = \frac{1}{2}$

$\therefore n^{\log_b a} = n^{\frac{1}{2}}$

$\therefore n^{\frac{1}{2}}$ vs $n^{\frac{1}{2}}$

$\therefore T(n) = \Theta(n^{\log_b a} \log_2 n)$

$\boxed{\therefore T(n) = \Theta(n^{\frac{1}{2}} \log_2 n)}$