Name: Pourna Sengupta

ID: 109086577

CSCI 3104, Algorithms **Escobedo & Jahagirdar**

Mid Term exam Summer 2020 (30 points) **Summer 2020, CU-Boulder**

Advice 1: For every problem in this class, you must justify your answer: show how you arrived at it and why it is correct. If there are assumptions you need to make along the way, state those clearly.

Advice 2: Verbal reasoning is typically insufficient for full credit. Instead, write a logical argument, in the style of a mathematical proof.

Honor code: On my honor as a University of Colorado at Boulder student, I have neither given nor sought unauthorized assistance in this work

Initials PS

Date June 30, 2020

If you violate the CU **Honor Code**, you will receive a 0.

Instructions for submitting your solution:

- The solutions **should be typed**, we cannot accept hand-written solutions. Here's a short intro to **Latex**.
- In this homework we denote the asymptomatic *Big-O* notation by $\mathcal{O}$ and *Small-O* notation is represented as o.
- We recommend using online Latex editor **Overleaf**. Download the **.tex** file from Canvas and upload it on overleaf to edit.
- You should submit your work through **Gradescope** only.
- If you don't have an account on it, sign up for one using your CU email. You should have gotten an email to sign up. If your name based CU email doesn't work, try the identikey@colorado.edu version.
- Gradescope will only accept **.pdf** files (except for code files that should be submitted separately on Canvas if a problem set has them) and **try to fit your work in the box provided**.
- You cannot submit a pdf which has less pages than what we provided you as Gradescope won't allow it.

Name: Pourna Sengupta

ID: 109086577

CSCI 3104, Algorithms **Escobedo & Jahagirdar**
Mid Term exam Summer 2020 (30 points) **Summer 2020, CU-Boulder**

Master Method: Consider a recurrence relation of the form

$$T(n) = aT(n/b) + f(n)$$

where $a \geq 1$ and $b > 1$ are constants, and $T(n) = constant$ for $n \leq 1$.
The asymptotic growth of T(n) is bounded as follows:

- **Case 1** $f(n) = \mathcal{O}(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0 \implies T(n) = \Theta(n^{\log_b a})$
- **Case 2** $f(n) = \Theta(n^{\log_b a}) \implies T(n) = \Theta(n^{\log_b a} \log(n))$
- **Case 3** $f(n) = \Omega(n^{\log_b a + \epsilon})$ for some constant $\epsilon > 0$ and if $af(\frac{n}{b}) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n. $\implies T(n) = \Theta(f(n))$

Formulae

- $1 + 2 + 3 + ... + n = \frac{n(n+1)}{2}$
- $1^2 + 2^2 + ... + n^2 = \frac{n(n+1)(2n+1)}{6}$
- **Sequences** The formulae for the sum of an Arithmetic Progression (AP) and Geometric Progression (GP) are available here

Name: Pourna Sengupta

ID: 109086577

CSCI 3104, Algorithms **Escobedo & Jahagirdar**
Mid Term exam Summer 2020 (30 points) **Summer 2020, CU-Boulder**

1. (3 pts) Which of $\mathcal{O}$, Ω or Θ is the correct asymptotic relationship for $f(n) = n^{\frac{1}{3}} \log_3(n)$ compared to $g(n) = \sqrt{n}$? Write your answer as $f(n) = \tau(g)$ where τ should be replaced by the appropriate symbol $(\mathcal{O}, \Omega, \Theta)$, show the necessary work to justify your answer.

To find τ we must find the asymptotic relationship between f(n) and g(n) to find whether g(n) provides a lower or upper bound to f(n).
$f(n) = n^{1/3} log_3(n)$
$g(n) = \sqrt{n}$
$f(n) > g(n)$
$f(g(n)) = (\sqrt{n})^{1/3} log_3(\sqrt{n})$
$f(g(n)) = n^{1/6} log_3(\sqrt{n})$
$f(g(n)) = \frac{1}{2} n^{1/6} log_3(n)$
$f(g(n)) = \frac{1}{2} n^{1/2} (n^{1/3}) log_3(n)$
$f(g(n)) = \frac{1}{2} n^{1/2} f(n)$
$f(n) > f(g(n))$
$f(n) > \frac{\sqrt{(n)}}{2} f(n)$
Therefore, $f(g(n)) < g(n) < f(n)$.
Since f(g(n)) and g(n) are always smaller than f(n), g(n) proves to be a asymptotically lower bound. Therefore, the two functions have a Big-Ω relationship. τ is therefore Ω.
Answer: f(n) = $\Omega(g)$

Name: Pourna Sengupta

ID: 109086577

CSCI 3104, Algorithms **Escobedo & Jahagirdar**
Mid Term exam Summer 2020 (30 points) **Summer 2020, CU-Boulder**

2. (2 pts) Which of $\mathcal{O}$, Ω or Θ is the correct asymptotic relationship for $f(n) = \frac{n^2-8n+15}{n-5}$ compared to $g(n) = \frac{1^2+2^2+....+n^2}{1+2+...+n}$? Write your answer as $f(n) = \tau(g)$ where τ should be replaced by the appropriate symbol $(\mathcal{O}, \Omega, \Theta)$, show the necessary work to justify your answer

Before we begin looking for τ, we can easily see that both f(n) and g(n) can be reduced.
$f(n) = \frac{n^2-8n+15}{n-5} = \frac{(n-3)(n-5)}{n-5} = (n-3)$
$g(n) = \frac{1^2+2^2+....+n^2}{1+2+...+n} = \frac{n(n+1)(2n+1)}{6} * \frac{2}{n(n+1)} = \frac{2n+1}{3}$
We can then find the asymptotic relationship between f(n) and g(n) to find whether g(n) provides a lower or upper bound to f(n).
$f(n) > g(n)$
$f(g(n)) = \frac{2n+1}{3} - 3$
$f(g(n)) = 2n + 1 - 9 = 2n - 8$
$f(n) < f(g(n))$
Therefore, $g(n) < f(n) < f(g(n))$.
Since f(n) falls between g(n) and f(g(n), g(n) proves to be an aysmptotically tight bound. Therefore, the two functions have a Big-Θ relationship. τ is therefore Θ.
Answer: f(n) = $\Theta(g)$

Name: Pourna Sengupta

ID: 109086577

CSCI 3104, Algorithms **Escobedo & Jahagirdar**
Mid Term exam Summer 2020 (30 points) **Summer 2020, CU-Boulder**

3. (1 pts) State True or false without justification
If the running time of an algorithm satisfies the recurrence relation
$T(n) = T(n/18) + T(17n/18) + cn$, then $T(n) = \mathcal{O}(n \log(n))$.

Answer: True

4. (1 pts) Let H be a hash table with 2020 slots with a hash function $h(x)$ that satisfies **uniform hashing** property. Given two items x_1, x_2, what is the probability that they do not hash to the same location?

The first item hashes to any of the 2020 slots. That leaves the second item with 2019 other slots to hash to. Since the hash function satisfies uniform hashing property, the second item has the same probability of choosing the same slot as item 1, $\frac{1}{2020}$. The probability of the second item hashing to the same location is therefore 1 - $\frac{1}{2020}$.
Answer: $\frac{2019}{2020}$

Name: Pourna Sengupta

ID: 109086577

CSCI 3104, Algorithms **Escobedo & Jahagirdar**
Mid Term exam Summer 2020 (30 points) **Summer 2020, CU-Boulder**

5. (3 pts) Suppose $T(n) = T(n-5) + n$, where $T(n) = \Theta(1)$ if $n \leq 5$. Find a function $g(n)$ such that $T(n) = \Theta(g(n))$. Clearly justify your answer.

Master Theorem for Decreasing Functions
$T(n) = aT(n-b) + O(n^k)$ where a > 0, b > 0 and $f(n) = O(n^k)$ where k ≥ 0.

Three Cases:
Case 1: $\mathbf{a < 1}$ $T(n) = O(n^k)$
Case 2: $\mathbf{a = 1}$ $T(n) = O(n^{k+1})$
Case 3: $\mathbf{a > 1}$ $T(n) = O(n^k a^{\frac{a}{b}})$

T(n) = T(n-5) + n and T(n) = $\Theta(1)$ where n ≤ 5
In this case, a = 1, b = 5, and k = 1. Therefore, we can apply **Case 2** which occurs when a = 1.
$T(n) = O(n^{k+1} = O(n^{1+1} = O(n^2)$
$T(n) = O(n^2)$
Therefore, the function $g(n) = n^2$ satisfies T(n) = Θ(g(n)).
Answer: T(n) = $\Theta(g(n)) = \Theta(n^2)$ where $g(n) = n^2$.

Name: Pourna Sengupta

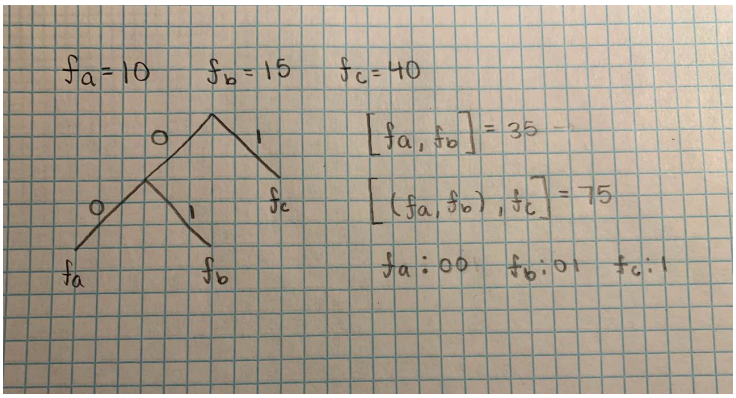
ID: 109086577

CSCI 3104, Algorithms **Escobedo & Jahagirdar**

Mid Term exam Summer 2020 (30 points) **Summer 2020, CU-Boulder**

6. (1 pt) For the set $\Sigma = \{a, b, c\}$, give the set of inequalities on the frequencies f_a, f_b, f_c that would yield the corresponding codewords $00, 01, 1$ respectively under Huffman's algorithm. Assume that while constructing the tree, we merge two nodes such that the node with least frequency is the left child. In your tree branching left corresponds to 0 and branching right corresponds to 1. List the frequencies f_a, f_b, f_c below that produce the specified codewords. Your choice of frequencies should **always** produce the same tree/codes provided.

Answer:
$f_a = 10$
$f_b = 15$
$f_c = 40$



7. (3 pts) For the given algorithm, solve the following.

   You may assume the existence of a `max` function taking $\mathcal{O}(1)$ time, which accepts at most four arguments and returns the highest of the four.

```
FindMax(A[0, ..., n-1]):
     if A.length == 1:
         return A[0]
    count = 0
    for i = 1 to n/2:
        count++

     return max( FindMax(A[0, ..., floor(n/4)],
                 FindMax(A[floor(n/4) + 1, ..., floor(n/2)],
                 FindMax(A[floor(n/2) + 1, ..., floor(3n/4)],
                 FindMax(A[floor(3n/4) + 1, ..., n-1]))
```

   Find a recurrence for the worst-case runtime complexity of this algorithm. You can

assume that n is a multiple of four. Solve the recurrence found to obtain worst-case runtime complexity.

From the code we can see that there is an obvious recurrence formed. a = 4, b = 4, and f(n) = $\Theta(n)$. From this, we can find that f(n) = n, giving us our final recurrence of $T(n) = 4T(\frac{n}{4}) + n$. To find the worst-case runtime complexity, the Masters Theorem can be applied.

(1) $n^{log_b(a)} = n^{log_4(4)} = n$
(2) f(n) = n
(3) $n^{log_b(a)} = f(n) \rightarrow n = n$

From this, we see that **Case 2** of the Masters Theorem can be used to solve the recurrence.
$f(n) = \Theta(n^{log_b(a)}) \rightarrow T(n) = \Theta(n^{log_b(a)} log(n))$.
$T(n) = \Theta(n^{log_4(4)} log(n))$
$T(n) = \Theta(nlog(n))$
Answer: The worst-case runtime complexity for the given recurrence is
$T(n) = \Theta(nlog(n))$

Name: Pourna Sengupta

ID: 109086577

CSCI 3104, Algorithms **Escobedo & Jahagirdar**
Mid Term exam Summer 2020 (30 points) **Summer 2020, CU-Boulder**

8. (4 pts) Assume there are n items $\{item_1, item_2, \ldots item_n\}$, each item has a weight w_i and value v_i associated with it. You have a bag that can carry a maximum load of weight W. Each of the n items can be divided into **fractions** such that the value and weight associated with the item decreases proportionally.
   The inputs to your function will be values v, weights w, number of items n and capacity W.
   Provide well commented pseudo or actual code, that returns the maximum total value of all items that can be carried.
   Also briefly discuss the space and runtime complexity of your pseudo-code.

   Input
   v = [20, 27, 18]
   w = [2, 3, 3]
   $W = 3$
   Output
   29
   The Algorithm should pick $item_1$ and $\frac{1}{3}^{rd}$ of $item_2$ leading to a total value of $20+27\frac{1}{3} = 29$.

   The algorithm uses a modified version of insert sort. The algorithm first sorts the arrays in descending order based on their ratio. It will then add the items one by one, checking to make sure it does not exceed the weight limit. If the weight limit is exceeded, the algorithm will go back and add only a fraction of the bags weight in order to meet the weight guideline. The use of insert sort allows for an average run time of $O(n^2)$ with a space complexity of O(1).
   (The psuedocode is below)

```
MaxBag(v, w, W) {
    //declare variables
    weight = 0;
    total = 0;
    //call modified insertion sort to sort values of v and w
    //will sort v and w arrays in descending order of v/w
    sort(v, w);
    //compare current weight to max weight
    for(i = 0; i < w.size(); i++){
        //if the weight can fit in the bag, add it to weight total
        //add v to the total
        if(weight + w[i] <= W){
            weight += w[i];
            total += v[i];
        }
        //if the weight doesn't fit in the bag
        else{
            //how much room is left
            r = W - weight;
            //add a fraction of the weight to the bag and therefore a fract
            total = += v[i] *(r/w[i])
        }
    }
    return total;
}

//modified insertion sort
sort(a, b){
    for(i = 2; i < a.length(); i++){
        //store values at indices
        val1 = a[i];
        val2 = b[i];
        //find and store v/w at index
        r = a[i]/b[i];
        //set index to 1 below i
        index = i - 1;
        //while loop to swap, compares ratios
        while(index > 0 && a[i]/b[i] < r){
```

```
                //swap so that the arrays are in descending order
                a[index + 1] = a[index];
                b[index + 1] = b[index];
                index = index - 1;
            }
            a[index + 1] = val1;
            b[index + 1] = val2;
        }
    }
```

9. (6 pts) Given an array A and a value k, design a divide and conquer algorithm to find the k^{th} smallest element in the array. The algorithm proposed should not use extra space i.e the space complexity of the algorithm should be constant $\Theta(1)$. Your algorithm must have an average case runtime of $\mathcal{O}(n \log(n))$. You can assume the access to a function which returns a random number within a range in constant time. You are allowed to modify the array passed as input.
The inputs to your function will be an array A and value k.
Provide well commented pseudo or actual code for the algorithm.

   Assume that you have access to a function *rand(min, max)* which will return a random integer between *min* and *max* in constant time inclusive of both min and max.

   Input
   $A = [10, 13, 20, 8, 7, 6, 100]$
   $k = 4$
   Output
   10

   10 is the 4th smallest value in the given array.

   The algorithm uses a modified version of random quicksort. The algorithm first finds a new, random pivot to use. It then uses this random pivot to sort through the elements, putting smaller elements to the left and larger elements to the right. Once it has fully sorted the elements, QuickSort calls itself. By doing this, it sorts the element before and after sortArray is called, continuously moving elements until the array is fully sorted. At the end of the algorithm, it returns the kth value in the array, which is now the kth smallest element in the array. The use of quicksort allows for an average run time of $O(nlog(n)$ with a constant time complexity of O(1).
   (The psuedocode is below)

```
//swap function
//swap using temporary variable as placeholder
swap(a, b){
    temp = a;
    a = b;
    b = temp;
}

//function uses pivot to sort elements into correct positions
//places elements smaller than pivot to left and those larger to the right
sortArray(A[], min, max){
    //declare variables
    p = A[min];
    //adds room to the arrays to move elements
    i = min - 1;
    j = max + 1;

    //find furthest element to the left that is greater or equal to the pivot
    if(A[i] < p){
        i++;
    }
    //find further element to the right that is smaller or equal to the pivot
    else if(A[j] > p){
        j--;
    }
    //if the element is the same
    if(i >= j){
        return j;
    }
    //swap arrays
    swap(A[i], A[j]);
}

randomPivot(A[], min, max){
    //generate random pivot
    rP = rand(min, max);

    //swap random pivot with max
```

```
    swap(A[rP], A[max]);

    //call sortArray to use new random pivot
    return sortArray(A, min, max);
}


QuickSort(A, min, max, k){
    //if the min is less than the max
    if(min < max){
        //call randomPivot to set new pivot
        //randomPivot calls sortArray and moves elements into correct order
        pivot = randomPivot(A, min, max);

        //call QuickSort to sort the elements before and after sortArray
        QuickSort(A, min, pivot);
        QuickSort(A, pivot + 1, max);
    }
    //return the value at the kth element
    //which would be the kth smallest element
    return A[k];
}
```

10. (6 pts) Assume there are n carrots and n rabbits along a straight line. Each rabbit needs to eat a carrot. Rabbits can move in either direction, simultaneously along the line and travelling 1 unit of distance takes 1 minute. Design a greedy algorithm that takes $\mathcal{O}(n \log(n))$ to assign carrots to rabbits such that the time taken to eat the last carrot is minimized. The algorithm should return the value of the time taken to eat last carrot.
You will be given the position of rabbits and carrots along the straight line.

Expectations

- You should clearly describe the greedy choice that the algorithm makes in assigning carrots to rabbits.
- Provide well commented pseudo or actual code for the algorithm.
- Discuss the space and runtime complexity of the pseudo or actual code.

Example 1:
rabbits = [7, 3, 2, 13, 2]
carrots = [1, 3, 5, 14, 21]
output : 8
In this example the assignment is as follows.

- The carrot at distance 1 (index 0) is eaten by rabbit at distance 2 (index 4).
- The carrot at distance 3 (index 1) is eaten by rabbit at distance 2 (index 2).
- The carrot at distance 5 (index 2) is eaten by rabbit at distance 3 (index 1).
- The carrot at distance 14 (index 3) is eaten by rabbit at distance 7 (index 0).
- The carrot at distance 21 (index 4) is eaten by rabbit at distance 13 (index 3), with 21-13=8 minutes being the longest time.

Example 2:
rabbits = [84, 15, 15, 161, 187, 9, 66, 1]
carrots = [92, 103, 163, 119, 63, 117, 144, 172]
output : 102

The algorithm first determines where the furthers carrot is located by comparing each carrot distance until it finds the largest. It them finds the distance from each rabbit to carrot to see which rabbit is closest. This is a greedy algorithm because we are sending each rabbit to the closes carrot to minimize the time, therefore 'greedily' choosing the best option. Since there are two loops, each taking $O(n)$ time, and a sorting function that takes $O(nlog(n))$ time, we find the overall time complexity to be $O(nlog(n))$. The space time complexity is O(1).
(The psuedocode is below)

Solving time complexity:
O(2n) + O(nlog(n)) = O(2n + nlog(n)) = O(nlog(n))

```
//greedy algorithm
EatCarrots(r[ ], c[ ]){
    //declare variable
    max = 0;
    //for loop from beginning to end of carrot array
    for(i = 0; i < c.length(); i++){
        //compare each element of carrot array to the max
        //if the carrot distance is max, change the value of max to reflect this
        if (max < c[i]){
            max = c[i];
        }
    }
    //to start, set min = max
    //so that we can continue finding the smallest distance
    min = max;
    //for loop from beginning to end of rabbit array
    for(j = 0; j < r.length(); j++){
        //find the distance between the carrot and rabbit
        temp = max - r[j];
        //find the smallest distance
        if(temp < min){
            //set min to the smallest distance
            min = temp;
        }
    }
    //return min, which is the longest time it'll take to get to the last carrot
    return min;
}
```

11. **For extra credit pick one of the two questions provided. Extra credit will only be considered if your midterm score less than 100% (2 pts)**

    For this extra credit question, please refer the leetcode link provided below. Multiple solutions exist to this question ranging from brute force to the most optimal one. Points will be provided based on Time and Space Complexities relative to that of the most optimal solution.

    Please provide your solution with proper comments, solutions without proper comments will not be considered.

    https://leetcode.com/problems/gas-station/

    OR

    https://leetcode.com/problems/maximal-square/

    ```
    Replace this text with your source code inside of the .tex document
    ```