CS 180, Fall 2019
Problem Set 4
Due October 30, 2019

Tian Ye
UID: 704931660

October 29, 2019

Exercise 13 Page 194

The algorithm will sort the jobs by descending weight/time. To prove this algorithm, assume there exists some algorithm that has an optimal solution that until the ith iteration, has the same sum s and time complexity of our algorithm. At this point, suppose our algorithm picks the job j_1 with the highest weight time ratio with a weight of w_1 with a time t_1, while the other algorithm picks a different job, j_2 with a lower weight time ratio with a weight of w_2 and with time t_2. After this point, our algorithm has a sum of $s + (w_1 * (t + t_1))$ while the other algorithm has a sum of $s + (w_2 * (t + t_2))$. Suppose the algorithms now choose the other job that was not chosen prior. Our algorithm has a sum of $s + (w_1 * (t + t_1)) + (w_2 * (t + t_1 + t_2))$, while the other will have a sum of $s + (w_2 * (t + t_2)) + (w_1 * (t + t_1 + t_2))$. Simplifying, we find the difference in time between the two solutions to be $w_2 * t_1 - w_1 * t_2$. However, since $w_1/t_1 > w_2/t_2$ and therefore $w_1 * t_2 > w_2 * t_1$, we know that our solution is better than the potential optimal solution that we proposed earlier. Hence, our solution is the optimal one.

Given a list L of n jobs j

```
Sort the jobs within L by descending weight_i / time_i
For each job j
        Do j
```

Proof. The algorithm runs in $O(n \log n)$ time complexity as it first sorts the n jobs in the list, which takes $O(n \log n)$ with an efficient sort. Afterwards, the algorithm iterates through the n jobs, performing each in turn - a time complexity of $O(n)$. Hence, the final time complexity is $O(n \log n)$. □

Exercise 17 Page 197

Since this problem is cyclical, the original Interval Scheduling Problem cannot be applied directly, as it is impossible to know which time that crosses the 12 interval is the best one. Since we know that the algorithm that solves the Interval Scheduling Problem finds the maximum interval schedule for some interval i that crosses 12, we can find the best one by brute forcing every possible acyclical schedule. We know the algorithm will work as intended as it was discussed extensively both in lecture and in the textbook.

Given a list L of intervals I

```
Sort L by ending time of I
Create an empty set S
For each interval I that crosses 12
        Create a new set of intervals S'
        Add to S' intervals that do not overlap with I
        Create empty set R
        For each interval I in S'
                Add I to R
                Delete intervals overlapping with I in S'
        If S' is larger in size than S, replace S with S'
Return S
```

Proof. The algorithm runs in $O(n^2)$ time complexity. Sorting the n intervals requires $O(n \log n)$ time complexity. Iterating through the nodes that cross 12 requires at worst case $O(n)$ time complexity. Going through the remaining elements inside that loop requires iterating through at worst case $n-1$ elements, leading to a time complexity of $O(n^2)$. Hence, the final time complexity is $O(n^2)$. □

Exercise 3 Page 246

We will use a divide-and-conquer method to determine whether there exists a majority for either side; we know a majority does not exist if there does not exist a majority for either side. Suppose there are some equivalent cards on both sides that do not make up a majority; there will be at most $n/4$ of the cards on each side, leading to a total of $n/2$ cards, which is still not a majority. Hence, unless there is a majority on at least 1 side, there cannot be a majority at all. If there is a majority on one side, a linear scan can be performed on the other side to determine whether there is an overall majority.

Given a set of cards S:
Function Find_Majority(Set of cards S)

```
If S is empty
        Return (0, NULL)
If S has 1 element e
        Return (1, e)
let l_count,l_card = Find_Majority left half S
let r_count,r_card = Find_Majority right half S
let l_rcard = instances of r_card on left half of S
let r_lcard = instances of l_card on right half of S
If l_count > 0 && r_count > 0
        If l_rcard + r_count > half length of S
                Return (l_rcard+r_count, r_card)
        Else if r_lcard + r_count > half length of S
                Return (r_lcard+l_count, l_card)
Else if l_count > 0
         If r_lcard + r_count > half length of S
                Return (r_lcard+l_count, l_card)
Else if r_count > 0
        If l_rcard + r_count > half length of S
                Return (l_rcard+r_count, r_card)
Return (0, NULL)
```

Proof. The algorithm runs within $O(n \log n)$ time complexity, as with each iteration we divide the problem in half, and at worst case scenario perform a linear scan of both halves. □

Exercise 7 Page 248

We are able to state that a local minima is a node that has a value lower than all of its neighbors. To approach this problem, we will divide the matrix recursively into four subquadrants, dividing along the middle row and column. For each quadrant, we will find which one contains the local minima. We are able to eliminate 3 out of 4 quadrants with every single iteration, since when we find the minimum in the middle column and check its neighbors, we are able to eliminate half the matrix, since we know whichever side contains the smaller node has to contain the local minima. We then check the middle row to eliminate the last quadrant. This will continue until we are left with a 2x2 or 1x1, whereby we brute force it to find the minimum node.

Given a Matrix A Function Local_Minima(Matrix A)

```
If A is 1x1
        Return the only node
If A is 2x2
        Return the smallest node
Let col be the middle column of A
Let row be the middle row of A
Get minimum node n_c in col
If n_c is less than its neighbors
        Return n_c
Else
        Let A' be the half of the matrix
          that contains the smaller neighbor of n_c
        Get minimum node n_r within A'
        If n_r > n_c
                Let Q be the quadrant that contains the
                  smaller neighbor of n_c
        Else
                If n_r is smaller than its neighbors
                        Return n_r
                Let n_m be the smallest neighbor of n_r's column
                Let Q be the quadrant that contains the n_m
        Local_Minima(Q)
```

Proof. With each iteration, we shrink the matrix into a $\frac{n^2}{4}$ sized matrix. At most, we will do $n + n/2 + 4$ probes into the matrix with each iteration, resulting in a time complexity of $O(n)$. $\square$

Exercise 5

We can perform a binary search for the smallest element in the array, assuming there exist no duplicate numbers. If so, the index of the smallest element is exactly K. First, we check to see if the array is not rotated by checking if the first element is less than the last element. If so, return 0. Otherwise, we perform a standard binary search until we are left with two elements, whereby we know the minimum element is the higher index of the two and return its index.

Given a set S with n elements

```
Let min=0
Let max=n-1
If S[min]<S[max]
    Return 0
While max-min > 1
    Let mid=(min+max)/2
    If S[mid]<S[min]
        max=mid
    Else
        min=mid
Return max
```

Proof. The time complexity is $O(\log n)$, as this algorithm is essentially just a binary search algorithm. □

Exercise 6

Assume we have a siftup and siftdown function already implemented for the given heap.

- Extracting the minimum:
  - Remove the top element of the heap $H[0]$
  - Move the last element of the heap into $H[0]$
  - Reduce the size of the heap by 1
  - Call Siftdown(0)
  - Return the removed element
  - Runtime: At worst case scenario, the heap needs to compare from the root all the way to the lowest child. Since the heap is balanced and binary, the time complexity is $O(\log n)$.
- Inserting a new number:
  - Insert item to end of heap
  - Increase size of heap by 1
  - Call Siftup(last element)
  - Runtime: At worst case scenario, the heap needs to compare from the root all the way to the lowest child. Since the heap is balanced and binary, the time complexity is $O(\log n)$.
- Changing a number:
  Assume we're given an index i to change to value x
  - Let $H[i] = x$
  - If $i > 0$ and $H[(i-1)/2] > H[i]$, call Siftup(i)
  - If $2i + 1 <$ length of heap and $H[i] > H[2i+1]$, call Siftdown(i)
  - If $2i + 2 <$ length of heap and $H[i] > H[2i+2]$, call Siftdown(i)
  - Runtime: Since both Siftup and Siftdown take at most $O(\log n)$, and since we perform only one of the two, the maximum time complexity is $O(\log n)$.