CS 180 Homework 4

Prithvi Kannan

405110096

1. Exercise 13 on page 194

The algorithm goes as follows:

```
Compute wi/ti for each task

Sort in non-increasing order to get schedule
```

The logic behind this algorithm is simple – assuming all tasks are equal weight, do the fastest jobs first; assuming all tasks are equal time, do the most important tasks first.

We prove this is the optimal algorithm by inversion. Our algorithm produces a solution of $0,1,\dots,n$. Assume there is an alternate solution with a pair out of order, such that $\frac{w_i}{t_i} < \frac{w_{i+1}}{t_{i+1}}$, meaning the solution is $0,1,\dots i-1, i+1, i, i+2, i+3, \dots, n$. For all jobs that are not the inverted pair, the completion time does not change, so we only need to consider the change in job i and job $i+1$. In the greedy algorithm, job i and job $i+1$ have the following contribution to the weighted sum, assuming job i starts at time T, $w_i(T+t_i) + w_{i+1}(T+t_i+t_{i+1})$. In the alternate algorithm, the contribution to the weighted sum is $w_i(T+t_i+t_{i+1}) + w_{i+1}(T+t_{i+1})$. The contribution to the weighted sum of the alternate algorithm is greater than the greedy algorithm, therefore we must swap job i and job $i+1$ to arrive at the optimal solution (minimize weighted sum). Performing these swaps for each inversion will produce the same solution as the greedy algorithm, hence the optimal solution is the greedy algorithm.

The runtime of this algorithm is $O(nlogn)$. It takes $O(n)$ to perform the initial computation of the ratio of weight to time, and then $O(nlogn)$ to sort in non-increasing order using a divide and conquer sorting algorithm such as merge sort.

2. Exercise 17 on page 197

The following algorithm computes the best scheduling:

```
Keep track of the best so far

For each interval n

    Pick an arbitrary point in n called p

    Remove all other intervals that overlap with p

    "Unwrap" the 24 hour timeline at p

    Run standard interval scheduling algorithm

    Update best so far if this interval schedule is better
```

The runtime of this algorithm is $O(n^2)$. This algorithm will take constant time to pick an arbitrary point, $O(n)$ to remove overlapping intervals, constant time to unwrap, and $O(n)$ to run standard interval scheduling. This process is repeated for each interval, so the total runtime is $O(n^2)$.

The proof is as follows. Consider the optimal solution to the full problem. Suppose this produces a set of intervals. This solution must take one of the intervals in the problem and use it as the "unwrapping point". Since our algorithm goes through all possible unwrapping points, our algorithm would find it.

3. Exercise 3 on Page 246

The divide and conquer algorithm is as follows:

```
If the number of cards is 1, return the card

If the number of cards is 2

    Compare cards and return either card if equal

Partition cards into c1 and c2

If recursive call on c1 returns a card

    Check against all other cards

Else

    If recursive call on c2 returns a card

        Check against all other cards

Return card from majority
```

For there to be a majority equivalence class, then at least one of the sides of the partition must contain a card of that equivalence class. This algorithm will check both halves and look for a majority equivalence class.

We can define the runtime of the algorithm with the recurrence relation $T(n) = T\left(\frac{n}{2}\right) + 2n$, which we can simplify to $O(nlogn)$.

4. Exercise 7 on page 248

Let A be the set of nodes outermost rows and columns of the grid. In a grid where a node v that is not in A is adjacent to a node in A and v is less than A, the global minimum does not occur on the border, so therefore G has at least one local min that is not on the border.

Let G satisfy the above property and let $v \notin A$ be adjacent to a node in A and smaller than all nodes in A. Let C be the union of nodes in the middle row and column of G. Let $S = A \cup C$. Deleting $G - S$ creates 4 subgrids. Let T be all nodes adjacent to S.

Using $O(n)$ searches, we find the node $u \in S \cup T$ that has minimum value. $u \notin A$ since $u \in S \cup T$ and v is less than all elements of A. Therefore we have two cases. In the first case, $u \in C$, so u is an internal local min (u's neighbors $\in S \cup T$ and u is the smallest). In the second case, u is in T. Let's denote G' as the subgrid with u and the parts of S bordering. G' follows the above property, so we run the algorithm recursively until we find the internal local minimum.

Using $O(n)$ searches, we can find any local minimum – not just internal ones, by finding a node v on the border of A. If v is a corner, it must be a local minimum. If v is not a corner, v must have a neighbor $u \notin A$. If v is less than u, then v is the local minimum. Otherwise, G has the property described above, and the algorithm can be run recursively.

This algorithm has runtime $T(n) = O(n) + T(\frac{n}{2})$, since the algorithm reduces the size of the G' by half in each iteration but goes through the entire G' in doing so. We can simply to $O(nlogn)$.

5. Suppose you are given an array of sorted integers that has been circularly shifted k positions to the right. For example taking (1 3 4 5 7) and circularly shifting it 2 position to the right you get (5 7 1 3 4). Design an efficient algorithm for finding K. Note that a linear time algorithm is obvious.

```
if high < low
    return 0
if high == low
    return low
set mid to average of high and low
if mid < high and arr[mid+1] < arr[mid]
    return mid + 1
if mid > low and arr[mid] < arr[mid - 1]
    return mid
if arr[high] > arr[mid]
    return findK(arr, low, mid - 1)
return findK(arr, mid + 1, high)
```

By definition, a sorted array shifted by k must only have one pair of $(i, i+1)$ where $arr[i] > arr[i+1]$, which happens at k. The algorithm checks if the whole input is ascending, and if not, which side the is not ascending. Once the side is determined, the algorithm is recursively called on that half. This ensures that no matter where the pair such that $arr[i] > arr[i+1]$ is within the array, the binary search will find it.

This algorithm can be represented by the recursive relation $T(n) = T\left(\frac{n}{2}\right) + c$, and $T(1) = 1$, which simplifies to time $O(logn)$.

6. Consider a (balanced) heap on n nodes. Show details of how you extract the minimum, insert a new number, and change a number (along with the corresponding post heapify process). Analyze the time complexity of your three algorithms.

Extracting the minimum assuming maxheap:

```
Set current min to heap[0]

Keep track of idx of min

Loop through the heap

        If the current element is smaller than min

                Set to min

                Update idx

Remove the element at idx from the heap

Decrease heap's size by 1

Heapify (see below)
```

The runtime of this algorithm is $O(n)$, finding the minimum value requires traversing the entire heap, which can have n nodes so this step takes $O(n)$. It takes $O(1)$ to remove that element, and then $O(logn)$ to re heapify after removing. The since $n > logn$, we say this runs in $O(n)$.

Extracting minimum assuming minheap:

```
Save the current root node

Copy the last value in the array to the root;

Decrease heap's size by 1

Heapify (see below)
```

Assuming the balanced heap is a minheap, then the minimum element is at the root node of the heap. Extracting the root node takes time $O(1)$. Then we must reheapify, which takes $O(logn)$.

Inserting a new number:

```
Increase heap’s size by 1

Set last value in heap to the new number

Heapify (see below)
```

This algorithm runs in time $O(logn)$. It takes constant time to add a number to the end of the array, and the heapify step takes time $O(log(n+1))$, but we call this $O(logn)$.

Change a number:

```
Loop through the heap
    If the current element is equal to the number
        Remove from heap
        Break
Set last value in heap to the new number
Heapify
```

This algorithm runs in time $O(n)$. It takes $O(n)$ to find a particular value in the heap since it can be anywhere, and we must traverse the entire heap to find it. Removing takes constant time, adding the new number takes constant time, and the heapify step takes time $O(log(n))$.

Heapify:

```
Sift down root's value. Sifting is done as following:
If current node has no children, sifting is over;
If current node has one child
    If heap property is broken
        Swap current node's value and child value
        Sift down the child
If current node has two children
    Find the smallest of them.
    If heap property is broken
        Swap current node's value and selected child value
        Sift down the child
```

The heapify algorithm takes time $O(logn)$. This algorithm can be represented by the recursive relation $T(n) = T\left(\frac{n}{2}\right) + c$, and $T(1) = 1$, which simplifies to time $O(logn)$.