CS222 Algorithm Design and Analysis Homework 2

Zhou Litao 518030910407 F1803016

September 23, 2020, Fall Semester

Exercise 1 Consider the following job scheduling problem. We are given all at once N jobs with positive lengths l_1,l_2,...,l_n. We can schedule only one job at a time and once we start a job we must run it to completion. A schedule for a set of jobs is then the starting time for each job s_1,s_2,...,s_n. Find an efficient algorithm to schedule the jobs to minimize the total wait time, where the wait time for a job is the difference between the time the job arrived and when it finished. Since all jobs arrive at the same time $t = 0$, the wait time for job i simplifies to $l_i + s_i$ and so the problem is to schedule the jobs to minimize $\sum_{i=1}^{i=n}(l_i + s_i)$. Please design an algorithm based on greedy strategy to solve the above problem(Write a pseudocode) and prove that your algorithm is correct.

Algorithm 1: Job Scheduling

Input: N jobs with positive lengths $l_1, l_2, \ldots, l_n$
Output: A schedule represented by the starting time for each job $s_1, s_2, \ldots, s_n$

```
Sort the jobs in ascending order of l_i;
time ← 0 ;
foreach Job_i in the sorted result do
    s_i ← time;
    time = time + l_i ;
end
return s_1, s_2, ..., s_n
```

Proof. Note that $\sum_{i=1}^{n} l_i$ is a constant for fixed input, we can argue that the algorithm will result in a schedule S that minimizes $\sum_{i=1}^{n} s_i$.

Define an inversion to be a pair of jobs where $s_i < s_j$ but $l_i > l_j$.

Define S^* to be an optimal schedule that has the fewest number of inversions, and let's see what happens.

We can assume that the optimal schedule has no idle time, otherwise we can remove the idle time, making $\sum_{i=1}^{n} s_i$ no greater than the original schedule.

If S^* has no inversisons, then $S = S^*$.

If S^* has an inversion, then we can find a pair of adjacent inversion s_i and s_j. We claim that swapping i and j won't increase $\sum_{i=1}^{n} s_i$.

- For $k \neq i, j$, since all jobs are consecutive, $s_k = s'_k$
- Before swapping, $s'_i + s'_j = \left(\sum_{k=1}^{i-1} l_k + l_j\right) + \sum_{k=1}^{i-1} l_k$
- After swapping, $s_i + s_j = \sum_{k=1}^{i-1} l_k + \left(\sum_{k=1}^{i-1} l_k + l_i\right)$
- Since $l_i \leq l_j$, it follows that $s_i + s_j \leq s'_i + s'_j$

Our claim contradicts the definition of S^*

□

Consider the above problem, but associate with the jobs positive values of importance $w_1, , w_n$, and minimize $\sum_{i=1}^{i=n} w_i(l_i + s_i)$. Find an efficient algorithm to determine an optimal schedule in this case. Please design an algorithm based on greedy strategy to solve the above problem(Write a pseudocode). You need not prove that your algorithm is correct.

Solution.

Algorithm 2: Job Scheduling 2

Input: N jobs with positive lengths $l_1, l_2, \ldots, l_n$ and weights $w_1, w_2, \ldots, w_n$
Output: A schedule represented by the starting time for each job $s_1, s_2, \ldots, s_n$

```
1 Sort the jobs in ascending order of l_i/w_i;
2 time ← 0 ;
3 foreach Job_i in the sorted result do
4 |   s_i ← time;
5 |   time = time + l_i ;
6 end
7 return s_1, s_2, ..., s_n
```

□

Exercise 2 Considering the following Problem, and show your algorithm with pseudocode. Please ensure that the complexity of your algorithm is $O(n)$.

You have an array "prices" for which the *ith* element is the price of a given stock on day i. Design an algorithm to find the maximum profit. You may complete as many transactions as you like (i.e., buy one and sell one share of the stock multiple times).

Note: You may not engage in multiple transactions at the same time (i.e., you must sell the stock before you buy again).

Example1: the input array is : [7,1,5,3,6,4]. The output of the maximum profit is 7.

Solution.

Algorithm 3: Stock Strategy

Input: Array P[n] representing the prices of stock
Output: The maximum profit profit

```
1  buy ← 0, sell ← 0 ;
2  profit ← 0 ;
3  for i = 1 : n − 1 do
4  |   if P[i] < P[i − 1] then
5  |   |   profit ← profit + P[sell] − P[buy] ;
6  |   |   buy ← i ;
7  |   |   sell ← i ;
8  |   else
9  |   |   if P[i] > P[sell] then
10 |   |   |   sell ← i ;
11 |   |   end
12 |   end
13 |   return profit
14 end
```

□

Exercise 3 Consider the following Interval Problem.

INPUT:A set $S = \{(x_i, y_i) | 1 \leq i \leq n\}$ of intervals over the real line. Think of interval (x_i, y_i) as being a request for a room for a class that meets from time x_i to time y_i.

OUTPUT: Find an assignment of classes to rooms that uses the fewest number of rooms.

Note: that every room request must be honored and that no two classes can use a room at the same time.

Consider the following iterative algorithm. Assign as many classes as possible to the first room(sorted by end time), then assign as many classes as possible to the second room, then assign as many classes as possible to the third room, etc. Is this algorithm correct? Prove the correctness of such idea, or else provide a counter-example, and design your algorithm.

Solution. The iterative algorithm is not correct. A counter example of input can be $\{[0,3],[1,5],[6,7],[4,9]\}$, which has been sorted according to the end time. According to the iterative algorithm, **three** classrooms will be assigned for $\{[0,3],[6,7]\}$, $\{[1,5]\}$, $\{[4,9]\}$. But the optimal solution can be $\{[0,3],[4,9]\}$ and $\{[1,5],[6,7]\}$, which only occupies two classrooms.

My algorithm is designed as Algorithm 4 shows.

Algorithm 4: Interval Scheduling

Input: A set $S = \{(x_i, y_i) | 1 \leq i \leq n\}$ of intervals over the real line
Output: An assignment a_i for each interval that use the fewest number of rooms

```
Sort the requests in ascending order of x_i ;
roomCnt ← 1 ;
while There is an element in S unassigned do
    foreach sorted unassigned request (x_i, y_i) ∈ S do
        if (x_i, y_i) is compatible with current schedule then
            a_i ← roomCnt ;
        end
    end
    roomCnt ← roomCnt + 1 ;
end
return
```

□

Exercise 4 Considering the following Problem, and show your algorithm with pseudocode. Please ensure that the complexity of your algorithm is $O(n)$.

Given an array of non-negative integers, you are initially positioned at the first index of the array. Each element in the array represents your maximum jump length at that position. Your goal is to reach the last index in the minimum number of jumps.

Note:You can assume that you can always reach the last index.

Example1: Input: [2,3,1,1,4]; Output: 2 Explanation: The minimum number of jumps to reach the last index is 2. Jump 1 step from index 0 to 1 (in index 0 you can choose to jump to index1 or index2), then 3 steps to the last index(in index 1 you can jump to index2, index3 or index4).

Solution.

Algorithm 5: Minimal Jump

Input: The maximum jump length array S_n
Output: The minimal steps required to reach the end of the array

```
jump ← 0, far ← 0, end ← 0;
for i = 0 : n − 1 do
    far ← max(far, S[i] + i) ;
    if i = end then
        jump ← jump + 1 ;
        end ← far ;
        if end ≥ n − 1 then
            return jump
        end
    end
end
```

□