MPCS 55001 Algorithms Autumn 2023
Homework 3

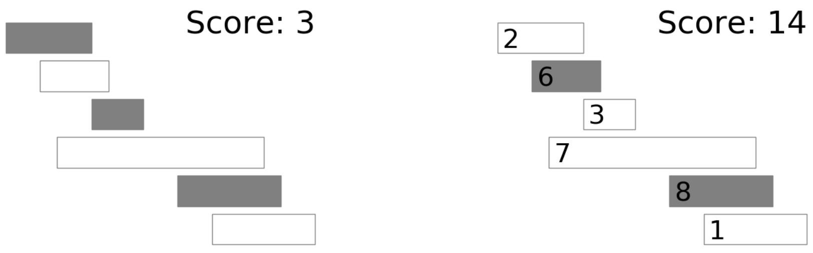
Zhiwei Cao zhiweic

Instructions: Write up your answers in LaTeX and submit them to Gradescope. **Handwritten homework will not be graded.**

Collaboration policy: You may work together with other students. If you do, state it at the beginning of your solution: **give the name(s) of collaborator(s)** and the nature and extent of the collaboration. Giving/receiving a hint counts as collaboration. Note: you must write up solutions **independently without assistance**.

Internet sources: You must include the url of any internet source in your homework submission.

1 Scheduling Problems (13 points)



(a) Interval Scheduling (b) Weighted Interval Scheduling

Figure 1: The best solution for the unweighted scheduling is the three filled jobs in the first image. For the weighted interval scheduling the optimal solution is the two filled jobs in the second image for a total weight of 14.

On a computing system there are N jobs. Each job needs to run from moment s_i to moment t_i. Unfortunately, the system cannot run jobs that overlap in the time they need to run. We want to run the greatest number of jobs possible.

(a) (2 points) Prove that ordering the jobs in increasing finishing time t_i and greedily selecting the first job that has no conflicts achieves the optimum schedule.

Since the schedule is ordered increasingly by the finishing time t_i, it can make sure that the first job with the earliest finish time. Suppose there is an optimal schedule S, but the job_1 is not in the S. This means there is another job in the schedule, we say job_x which overlaps the jab_1. If job_x doesn't overlap job_1, then the schedule is not optimal, since we can still add a job into the schedule. If we remove the job_x from the optimal schedule and replace it with the job_1, then the schedule still remains optimal, since the job_1 has the earliest finish time which doesn't overlap with others as well. It removes one job and adds a new one also reminding the size of S.

However, not all jobs are spawned equal. Each job is assigned a weight w_i and we want to select the subset S of non-overlapping jobs such that $\sum_{i \in S} w_i$ is maximized. Unfortunately, this problem cannot be solved with a greedy algorithm. Give a **dynamic programming** algorithm that takes as input an array of jobs $[(s_1, t_1, w_1), \ldots, (s_n, t_n, w_n)]$ and returns the maximum possible sum of weights w_i of non-overlapping jobs that is achievable. The running time should be $O(n^2)$.

(b) (2 points) Define the subproblem that you will use to solve this problem **precisely**. Define any variables you introduce. What are the dimensions of your dynamic programming table? What is the space complexity?

Sub-problem: Find the maximum weight for the first i jobs. $M[i] = \max(M[i-1], M[x] + w_i)$
M is an array with the length of jobs; i is the index of the job; $M[i]$ means the maximum weight for the first i jobs; x is the biggest index of the job that just finished before the job i.
Dimension: A table with with dimensions of i, which is the number of jobs.
Space Complexity is O(n), the size of table.

(c) (3 points) Give a recurrence which expresses the solution to each subproblem in terms of smaller subproblems. State any base case(s). Justify your recurrence and state the time complexity of your recurrence.

$M[i] = \max(M[i-1], M[x] + w_i)$
Base case: $M[0] = 0$
When we need to consider the current job's weight in the schedule, we need to find the biggest job index that finishes before the current job, which is direct to the job_x. Then we could consider the max weight on the first x jobs and add the current job's weight. On the contrary, if we want to give up the current job, the recurrence could just store the weight for the first $i-1$ jobs to the current job.
In the worst case, each cell is required to check all previous cells. As a result, the time complexity is $O(n^2)$

(d) (3 points) Write **pseudocode** for an algorithm to solve this problem. Argue that your algorithm runs in $O(n^2)$ time.
Since the for loop will iterate n jobs, and it will call the *FindBiggestIndex* function every time. As a result, the time complexity is $O(n^2)$

Algorithm 1 Schedule Problem

```
function WeightedSchedulingProblem(Jobs)
Input: jobs which start time $s$, end time $t$, and weight $w$: $[(s_1, t_1, w_1), ..., (s_n, t_n, w_n)]$
Output: Max weighted Schedule
Sort the Jobs by the finish time $t_i$
create a array$M[0, \dots, n]$
$M[0] \leftarrow 0$
for $i \leftarrow 1$ to $n$ do
    $M[i] = \max(M[i-1], M[FindBiggestIndex(i)] + Jobs_i.w)$
end for
return $M[n]$
```

Algorithm 2 Find Biggest Index of Job that Finishes Before current Job

```
function FindBiggestIndex(Jobs, index)
Input: Jobs which start time $s$, end time $t$, and weight $w$: $[(s_1, t_1, w_1), ..., (s_n, t_n, w_n)]$
Input: Index: the job index we are working with
Output: Max weighted Schedule
for $i \leftarrow index - 1$ to 1 do
    if $Jobs_i.t < Jobs_{index}.s$ then
        return $i$
    end if
end for
return 0
```

(e) (3 points) Modify your algorithm to run in time $O(n \log n)$. If you have already done so above, you do not need to answer this question.
Update the *FindBiggestIndex* function using the binary search method which takes $O(\log n)$ instead of $O(n)$. As a result, it will update the whole algorithm to $O(n \log n)$

Algorithm 3 Find Biggest Index of Job that Finishes Before current Job

```
function FindBiggestIndex(Jobs, index)
Input: Jobs which start time $s$, end time $t$, and weight $w$: $[(s_1, t_1, w_1), ..., (s_n, t_n, w_n)]$
Input: Index: the job index we are working with
Output: Max weighted Schedule
$left \leftarrow 0$
$right \leftarrow index - 1$
while $left <= right$ do
    $mid \leftarrow (left + right)/2$
    if $Jobs_{mid}.t \leq Jobs_{index}.s$ then
        if $Jobs_{mid+1}.t \leq Jobs_{index}.s$ then
            $left \leftarrow mid + 1$
        else
            return mid
        end if
    else
        $right \leftarrow mid - 1$
    end if
end while
return $-1$
```

2 RNA Folding (12 points)

A major difference between RNA and DNA is that RNA exists as a single strand that folds and pairs with itself. RNA consists of 4 different bases: adenine `A`, cytosine `C`, guanine `G`, and uracil `U`. Adenine pairs with uracil and cytosine pairs with guanine. There are many other factors, but generally RNA folds in a way that maximizes the number of pairs formed.

In this problem, we will consider the case where an RNA strand folds on itself in a single location. Consider the RNA strand `AUUGCAGACGCU`. If the strand folds after the 6th base `AUUGCA|GACGCU`, this is a possible result:

```
A U U G C A − − −
U − − C G − C A G
```

If the strand folds after the 7th base `AUUGCAG|ACGCU`, this is a possible result:

```
A U U G C A G −
U − − C G − C A
```

Note that after folding, the second half of the RNA strand is paired backwards and we are able to skip over bases when forming pairs. There are 3 pairs when folding after the 6th base and 4 pairs when folding after the 7th base; thus, we consider folding after the 7th base more optimal.

Design a **dynamic programming** algorithm which, when given a string $S[1 \dots n]$ consisting of the characters $\{\mathtt{A}, \mathtt{C}, \mathtt{G}, \mathtt{U}\}$, returns the maximum number of pairs that can be formed by a single fold. Your algorithm should be as asymptotically efficient as possible.

(a) (4 points) Define the subproblem that you will use to solve this problem **precisely**. Define any variables you introduce.
Find the maximum pairs in $DP[i][j]$ cell, aka, find the maximum pairs from the sub-string $S[i, \dots, j]$
$S[i \dots j]$ means the sub-string of the input from index i to index j.
$DP[i][j]$ is a 2D array where each cells represent the maximum pairs that can be found in the sub-string $S[i \dots j]$.
Reference from: https://www.molgen.mpg.de/3710236/eddy2004.pdf

(b) (4 points) Give a recurrence which expresses the solution to each subproblem in terms of smaller subproblems. State any base case(s).

Base Cases: $DP[i][j] = 0$ where i =j and i = j-1

$$DP[i][j] = max \begin{cases} DP[i][j-1] \\ DP[i+1][j] \\ DP[i+1][j-1]+1, & \text{while S[i] and S[j] matched} \\ max_{i<k<j}(DP[i][k] + DP[k+1][j]) \end{cases} \tag{1}$$

(c) (3 points) Write **pseudocode** for an algorithm to solve this problem.

Algorithm 4 RNA Folding

```
function findMaxRNAPairs(S)
Input: a string $S[1 \dots n]$ consisting of the characters $\{A, C, G, U\}$
Output: returns the maximum number of pairs that can be formed by a single fold.
create a 2D array$DP[0, \dots, n][0, \dots, n]$
$DP[i][j] \leftarrow 0$ where i = j
$DP[i][j] \leftarrow 0$ where i = j - 1
for $i \leftarrow$ n-1 to 0 do
    for $j \leftarrow$ n-1 to i-1 do
        $DP[i][j] \leftarrow max(DP[i][j-1], DP[i+1][j])$
        if $S[i]$ matches $S[j]$ then
            $DP[i][j] \leftarrow DP[i+1][j-1] + 1$
        end if
        for $k \leftarrow$ i to j-1 do
            $DP[i][j] \leftarrow max(DP[i][k] + DP[k+1][j])$
        end for
    end for
end for
return DP[0][n]
```

(d) (1 point) Analyze the running time and space complexity of your algorithm.
$O(n^3)$.
Since the 2D array with iterate n^2 cells which take $O(n^2)$. However, the inner for loop which could find the optimal pairs makes the time complexity go up to $O(n^3)$

3 Clustering (13 points)

The problem of **clustering** a sorted sequence of one-dimensional points $x_1, \dots, x_n$ entails splitting the points into k clusters (where $k \leq n$ is an input parameter) such that the sum of the squared distances from each point to its cluster mean is minimized.

For example, consider the following sequence with $n = 5$:

$$3, \; 3, \; 6, \; 16, \; 20$$

Suppose we want to partition it into $k = 2$ clusters. Here is one possible solution:

$$3, 3 \mid 6, 16, 20$$

The mean of the first cluster is $(3+3)/2 = 3$, and the mean of the second cluster is $(6+16+20)/3 = 14$. The cost (total variance) of this clustering is $(3-3)^2 + (3-3)^2 + (6-14)^2 + (16-14)^2 + (20-14)^2 = 104$. This clustering is not optimal because there exists a better one:

$$3, 3, 6 \mid 16, 20$$

The mean of the first cluster is $(3+3+6)/3 = 4$, and the mean of the second cluster is $(16+20)/2 = 18$. The cost of this clustering is $(3-4)^2 + (3-4)^2 + (6-4)^2 + (16-18)^2 + (20-18)^2 = 14$, which is optimal.

Give a **dynamic programming** algorithm that takes as input an array $x[1..n]$ and a positive integer k, and returns the lowest cost of any possible clustering with k or fewer clusters. The running time should be $O(n^3k)$. Note: $O(n^3k)$ is not necessarily the optimal running time!

(a) (4 points) Define the subproblem that you will use to solve this problem **precisely**. Define any variables you introduce. What are the dimensions of your dynamic programming table? What is the space complexity?
Find the minimum cost of clustering in $DP[i][j]$ which puts the first i elements into j clusters.
Input: an array x[1..n] and a positive integer k
$DP[i][j]$ with the dimension of n * k
Space Complexity: $O(n * k)$
Reference: https://journal.r-project.org/archive/2016/RJ-2016-022/RJ-2016-022.pdf

(b) (4 points) Give a recurrence which expresses the solution to each subproblem in terms of smaller subproblems. State any base case(s). Justify your recurrence and state the time complexity of your recurrence.
$DP[i][j] = min_{1 \leq p \leq i}(DP[p-1][j-1] + cost(p,i))$
Base case: DP[i][0] = inf; DP[0][j] = inf
The time complexity should be $O(k * n^3)$, which i takes n values, j takes k value, p takes n values, and the $cost()$ takes n values as well.

(c) (3 points) Write **pseudocode** for an algorithm to solve this problem. Argue that your algorithm runs in $O(n^3 k)$ time.
The time complexity should be $O(k * n^3)$, in which i takes n values, j takes k value, p takes n values, and the $cost$ takes n values as well.

Algorithm 5 Clustering

```
function findLowestCost(x, k)
Input: an array x[1..n]
Input: a positive integer k
Output: the lowest cost of any possible clustering with k or fewer clusters
create a 2D array$DP[0,\dots,n][0,\dots,n]$
$DP[i][0] \leftarrow inf$
$DP[0][j] \leftarrow inf$
for $i \leftarrow$ 1 to n do
    for $j \leftarrow$ 1 to k do
        for $p \leftarrow 1 to i$ do
            $meanVal \leftarrow$ mean value of $x[p \dots i]$
            $cost \leftarrow sum((x[m] - meanVal)^2)$ where m ← p to i // the cost of current clustering
            $DP[i][j] = min(DP[i][j], DP[p-1][j-1] + cost)$
        end for
    end for
end for
return $DP[n][k]$
```

(d) (2 points) Explain in a few sentences how you can improve the running time of your algorithm to $O(n^2 k)$. If your previous algorithm already runs in $O(n^2 k)$ time, you may leave this part blank for full credit.
Could compute the cost for each cell in advance which takes $O(n^2)$, which saves time during the loop iterations. It can just take the cost of the cell. It will update the time complexity of filling the dp table to $O(k * n^2)$. In sum, the time complexity will be $O(k * n^2)$.

4 Programming: Document Reconstruction (20 points)

Follow this GitHub Classroom link to accept the assignment. Your code should be pushed to GitHub; you do not need to include it here.