CSOR4231 Algorithm
HW3

Jialin Zhao — jz2862

March 5, 2018

1 Problem I

Find the longest monotonically increasing subsequence of a sequence.
Give a dynamic programming algorithm that solves this problem in time $O(n^2)$.

Subproblem OPT(i) = L[i] is the length of the LIS subsequence ending at a_i.
The recurrence relationship is $OPT(i) = \max_{1 \le k < i, a_k \le a_i} (OPT(k) + 1, 1)$.
The problem equals to finding max(OPT(i)), i.e. the max length of LIS subsequnece ending at any i-th element,$i \in [1, n]$.

Pseudocode:

Represent the input sequence as A = $[a_1, a_2 ... a_n]$.
Here I use an array L[n] where L[i] = OPT(i) to denote the length of the LIS subsequence ends at a_i.
And I use an array F[n] to denote the position number of the former element in the LIS subsequence ends at a_i, aiming to backtrack the subsequence.
MaxValue is the length of LIS; Result is the LIS.

Algorithm 1 Function LIS(A)

```
L ⟸ an array of the length n while filled with element 1
F ⟸ an array of the length n while filled with element 0
MaxValue ⟸ 0
Pos ⟸ 0
for i from 2 to n do
  for j from 1 to i-1 do
    if A[j − 1] ≤ A[i − 1] then
      if L[i-1] < L[j-1]+1 then
        L[i-1] = L[j-1]+1
        F[i-1] = j-1
      end if
    end if
  end for
  if MaxValue < L[i-1] then
    MaxValue = L[i-1]
    Pos = i-1
  end if
end for
Result ⟸ an array of the length MaxValue and filled with element Pos
for i from 2 to Maxvalue do
  Result[MaxValue-i]=F[Pos]
  Pos = F[Pos]
end for
Result ⟸ convert the element position stored in Result to the corresponding value in A
return MaxValue, Result
```

Correctness:

Base case: OPT(0) = 0;

Induction:

So if we suppose the case $OPT(i-1), i \geq 1$ is correct for the length of longest monotonically increasing subsequence ebds at a_{i-1};

$OPT(i) = \max_{1 \leq k < i, a_k \leq a_i} (OPT(k) + 1, 1)$.

As we know, for the LIS subsequence ends at a_i, a_i must following a LIS subsequence ends at a_j where $j < i$ and $a_j \leq a_i$; if there is no former element smaller than a_i, the LIS subsequence ends at a_i is just a_i itself with length 1.

Since max($(OPT(k)+1)$ $(for\ 1 \leq k < i, a_k \leq a_i)$, 1) covers every case that may satisfy the above statement. Then OPT(i) is also the length of the longest monotonically increasing subsequence.

Conclusion:

It follows that the statement is true for all $1 \leq i \leq n$ since we can apply the inductive step for every odd or even number i = 2; 3; : : :

Runnig time:

Outside the DP loop: Running time is O(n)+O(M) = O(n) where M is the length of the LIS sequence, $M \leq n$.

Inside the DP loop: The total number of the iterations is $\sum_{i=1}^{n} i = O(n^2)$;The time for every iteration is O(1). So the running time costed inside the loop is $O(n^2)$.

The running time is $O(n^2)$.

Space complexity:

The additional space in this algorithm is the space of L,F,Result,MaxValue,Pos.

The space complexity is $O(n)$

2 Problem II

Give an efficient algorithm that determines:
At which libraries to place copies of the books so that the total cost is minimized; The minimum total cost.
OPT(i) is the minimum cost for the subproblem that considers libraries from 1 to i.
The recurrence relationship is $OPT(j) = \min_{1 \le k < i} (OPT(k) + Cost(k+1, j))$,the Cost(k,j) = 0.5*(j-k)*(j-k+1) is the total cost from k to j when only j store a book.
The problem equals to finding $OPT(n)$.

Pseudocode:

Denote the cost of storing the book in each library as array L = $[c_1, c_2 ... c_n]$, use the similiar DP strategy as matrix chain multiplication.
Use array C[i] of size n, C[i] = OPT(i).
Use array F[i] of size n to denote the former library storing the book before library i in the best strategy.
The algorithm iterate on the DP recurrence relationship mentioned above and get the minimum cost and distribution.

Algorithm 2 Function MininumCost(L)

```
C ⟸ an array of size n while filled with ∞
F ⟸ an array of size n while filled with -1
for i from 1 to n do
  for k from 1 to i do
    q = C[k-1] + 0.5 * (i-k)(i-k-1) + L[i-1]
    if q < C[i − 1] then
      C[i-1]=q
      F[i-1]=k
    end if
  end for
end for
return C[0][n − 1], the order by backtrack through F starting at F[n-1] until finding -1
```

Correctness:

Base case: for i = 1, $OPT(1) = c_1$ is the correct cost
Induction:
Suppose the case $OPT(x)$ *for every* $x \in [1, i], i \ge 1$ is true that it represents the minimum cost of the book storage distribution from 1 to i.
For $OPT(i+1)$:
Since we know that the user delay of a library is merely decided by the nearest following library storing a book, so the cost for library i to j can be divided into two seperate part if there is a last but not one library k storing a book: Cost(i+1) = Cost(i,k)+Cost(k+1,j). So the $OPT(i) = \min_{1 \le k \le i} (OPT(k) + Cost(k+1, j))$ considers every possible distribution, and is also true for being the minimum cost of the book storage distribution from i to j.
Conclusion::
It follows that the statement is true for all $i \in [1, n]$ since we can apply the inductive step for every odd or even number i = 2; 3; : : :

Runnig time:

Outside the DP loop: Since the running time of backtrak(F) is O(n), the total time cost is O(n)
Inside the DP loop: The total number of the iterations is $O(n^2)$;
The time for every iteration is O(1). So the running time costed inside the loop is $O(n^2)$.

The running time is $O(n^2)$.

Space complexity:
The additional space in this algorithm is the $O(n)$ space for C,F.
The space complexity is $O(n)$

3 Problem III

Give an efficient algorithm that determines an ordering of the tasks for the supercomputer that minimizes the duration of the process, as well as the minimum duration.
A greedy algorithm is needed for this problem.
To minimize the end time of the processors, we need to schedule the jobs in decreasing order of f_i.

Pseudocode:

Here I use P=$[p_1, p_2, p_3...p_n]$ to denote the supercomputer time for each task and F=$[f_1, f_2, f_3...f_n]$ to denote the processor time for each task.

Algorithm 3 Function Greedy(P, F)

```
//Keep track of the index while we do the sort on the descending order of f_i:
for i from 1 to n do
   F[i-1]=(F[i-1],i)
end for
MergeSort(F) on the descending order of f_i //i.e. F[i-1][0]
Order ⟸ the order of index in F //i.e. F[i-1][1]
Duration = 0, Start = 0
for i from 1 to n do
   Start += P[F[i-1][1]-1]
   Duration = max(Duration,Start+F[i-1][0])
end for
return Order, Duration
```

Correctness:

Note that no matter what the order is, the duration ends with the last task finished by a processor.
The endtime is the maximum of the endtime of each processor:
= whole time spent by supercomputer + max(the remaining time of the processor)
We know that the whole time spent by supercomputer is always tha same $\sum_1^n p_i$. So we need to put the most processor-time-consuming task at the first place followed by the second most processor-time-consuming task.....
To put it another way, longer-on-processor-time tasks need to be arranged before shorter-on-processor-time task.
By contradiction, if we swap the order of two tasks in our result, the finishing time for those two task will be determined by later one which means the processing time will be longer or un-changed.
Conclusion:
So the order in the result is optimal.

Runnig time:

The MergeSort takes $O(n \log n)$ and other parts take $O(n)$ time.

The running time is $O(n \log n)$.

Space complexity:

The additional space in this algorithm is:
The MergeSort uses space $O(n)$;
Order uses $O(n)$ space; other elements use $O(1)$ space.

The space complexity is $O(n)$

4 Problem IV

Design and analyze a dynamic programming algorithm for this problem that runs in time polynomial in n and $\sum_{i=1}^{n} a_i$

Subproblem boolean OPT(i,j,k) represents if there are two disjoint subsets I,J $\subseteq [a_1...a_k]$ satisfying the sum of I is x and the sum of J is y.

The recurrence relationship is $OPT(i,j,k) = OPT(i,j,k-1) \cup OPT(i-a_i,j,k) \cup OPT(i,j-a_i,k)$

Set $A = 1/3 \sum_{i=1}^{n} a_i$, if A is an integer and we can find two disjoint subset of L whose sum = A, then the left elements should also sum to A.

The 3-partition problem equals to find true or false in OPT(n,A,A)

Define a boolean matrix OPT(i,j,k)=M[i][j][k] in size [A+1][A+1][n+1], with the meaning that M[x, y, k] where $x, y \in [0, A], k \in [0, n]$.

As we know that if the M[i][j][k]=1, M[i][j][k+1] should also be 1, the true value can be inherited from former matrix, so we only need to maintain matrix M[A+1][A+1] and update it for n+1 iterations.

Pseudocode:

Algorithm 4 Function 3-Partition(L)

```
n = length of L
A = 1/3 ∑_{i=1}^{n} L[i − 1]
if A is not an integer or n is less than 3 then
  return false
end if
M ⟸ a matrix of size (A+1,A+1) and filled with false
M[0][0]=true
for k from 1 to n do
  a = L[k-1]
  for i from 0 to A do
    for j from 0 to A do
      M[i][j] = M[i][j] or M[i-a][j] or M[i][j-a]
    end for
  end for
end for
return M[A][A]
```

Correctness:

Base case:

$$OPT(i,j,k) = \begin{cases} 1(when\ i=0, j=0) \\ 0(else) \end{cases}$$

is the correct representation for k = 0 whether there are two disjoint subsets I,J $\subseteq [a_1...a_k]$ satisfying the sum of I is x and the sum of J is y.

Induction: Suppose OPT(i,j,k-1) is correct for $k \geq 1$

$OPT(i,j,k) = OPT(i,j,k-1) \cup OPT(i-a_k,j,k) \cup OPT(i,j-a_k,k)$

We know that if we can find two subsets meeting the requirment taking the 1 to k-th elements into consideratoin, when we remove the k th element, we can also find two valid subsets for 1 to k-1th elements;otherwise there should be two valid subsets sum to $i-a_k$,j or $i, j-a_k$ for 1 to k-1 th elements. And those information is correctly known in our asumption.

Then the representation is also correct for OPT(i,j,k).

Conclusion:

The OPT(i,j) in the algorithm is true for every $i \in [1, n], j \in [1, D]$

Runnig time:

Outside the DP loop: O(n)

Inside the DP loop: The total number of the iterations is $\sum_{k=1}^{n} A^2 = O(nA^2)$;
The time for every iteration is O(1). So the running time costed inside the loop is $O(nA^2)$.

The running time is $O(nA^2)$.

Space complexity:

The additional space in this algorithm is the $O(A^2)$ space for M.

The space complexity is $O(A^2)$

5 Problem V

You are given some data to analyze. You can spend D dollars to perform the analysis. You have organized the process of analyzing the data so that it consists of n tasks that have to be performed sequentially by using dedicated hardware: you will use a processor P_i to perform task i, for every i. Each processor is relatively cheap but may fail to complete its task with some probability, independently of the other processors. Specifically, P_i costs c_i dollars and succeeds to complete its task with probability s_i, while it fails with probability $1 - s_i$.

5.1 What is the probability that the process of analyzing the data will be completed successfully?

According to chain rules: the probability that the process of analyzing the data will be com- pleted successfully $P_a = s_1 * s_2 * ...s_n = \prod_{i=1}^{n} s_i$

5.2 Note that you can improve this success probability by using p_i identical processors P_i for task i instead of just one.

5.2.1 What is the probability that task i will be completed successfully now?

$P_i = 1 - probability\ to\ fail = 1 - (1 - s_i)^{p_i}$

5.2.2 What is the probability that the process of analyzing the data will be completed successfully?

According to chain rules: the probability that the process of analyzing the data will be completed successfully $P = \prod_{i=1}^{n} P_i = \prod_{i=1}^{n} (1 - (1 - s_i)^{p_i})$

5.2.3 Given s_1; : : : ; s_n; c_1; : : : ; c_n and D, compute p_1; : : : ; p_n such that the success probability of the entire process is maximized while you do not spend more than D dollars.

Subproblem $OPT(i, j)$ denotes the maximum success probability for task 1 to i if we spend at most j money on them.
The recurrence relationship is:

$$OPT(i,j) = \max \begin{Bmatrix} OPT(i, j - c_i) * (1 - (1 - s_i)^1) \\ ..... \\ OPT(i, j - (k-1)c_i) * (1 - (1 - s_i)^{k-1}) \\ OPT(i, j - kc_i) * (1 - (1 - s_i)^k) \end{Bmatrix}$$

where k satisfy that $j - kc_i$ can still allocate enough money for at least 1 processor for each task before i.
So the problem is equal to finding the OPT(1,n)

Pseudocode:

I use K[i][j] of size (n+1,D+1) as OPT(i,j). Basice as the value

Algorithm 5 Function distribution(S, C, D)

```
K ⟸ Matrix of size (n+1,D+1) to record the max probablity from task 1 to task i and is filled with 0
Dist ⟸ Matrix of size (n+1,D+1), the best number of processor i for max probablity from task 1 to task
i, filled with 0
n ⟸ length of S
/* Basic is the least cost for task 1 to i-1,we know that every task need at least 1 processor */
Basic ⟸ array of length n+1
Basic[0] = 0, Basic[1] = 0
for i from 2 to n do
  Basic[i] = Basic[i-1]+C[i-2]
end for
for i from 0 to n do
  for j from 0 to D do
    if i==0 or j==0 then
      K[i][j]=0
      continue
    end if
    for x from 1 to (j-Basic[i])/C[i-1] do
      xProb = 1 − (1 − S[i − 1])^x
      xCost = x*C[x-1]
      prob = K[i-1][j-xCost]*xProb
      if prob>K[i][j] then
        K[i][j]=prob
        Dist[i][j] = x
      end if
    end for
  end for
end for
return K[n][D], backtrack(C,Dist,n,D)
```

Algorithm 6 Function backtrack(C,Dist,n,D)

```
Distribution ⟸ an array of length n
for i from n to 1 do
  Result[i-1] = Dist[i][D]
  D = D - Dist[i][D]*C[i-1]
end for
```

Correctness:

Base case: OPT(i,j)=0 when i=0 or j=0; It's true for being the maximum probability for the first i tasks under j money.

Induction: Suppose the maximum probability is correct for every $OPT(x \leq i, y \leq j)$ except OPT(i,j).

For OPT(i,j), we know that the any valid distribution for i,j must allocate an amount of money on i and remain some money on tasks before i.

So the maximum probablity for task 1 to i must be multiplation of maximum probability for task 1 to i-1 and probability for task i, under some allocation of money.

Since every best distribution for tasks before i with money less than j, i.e. $OPT(x \leq i, y \leq j)$, is correctly known.

Then the maximum probablity is also correct for OPT(i,j).

Conclusion:

The OPT(i,j) in the algorithm is true for every $i \in [1, n], j \in [1, D]$

Runnig time:

Outside the DP loop: O(n)
Inside the DP loop: The total number of the iterations is $\sum_{i=1}^{n}\sum_{j=0}^{D}((j - Basic[i])/C[i-1]) = O(nD^2)$;
The time for every iteration is O(1). So the running time costed inside the loop is $O(nD^2)$.

The running time is $O(nD^2)$.

Space complexity:

The additional space in this algorithm is the $O(nD)$ space for K and $O(n)$ for Basic,Disc.

The space complexity is $O(nD)$