CSOR W4231: Analysis of Algorithms (sec.001) - Problem Set #3

Jiawen Huang (jh4179). Collaborators: A. Turing. - jh4179@columbia.edu

March 9, 2020

Problem 1

We can use DP to solve the problem. Instead of directly computing the probability, we sum up the probabilities of all the conditions that will lead to k heads in the end.
Pseudocode:

Prob(k, n)

---

if $n = k = 0$ **then**
  return 1
else if $n = k > 0$ **then**
  return $\prod_{i=1}^{n} p_i$
else if $n > k = 0$ **then**
  return $\prod_{i=1}^{n}(1 - p_i)$
else
  return $Prob(k-1, n-1) * p_n + Prob(k, n-1) * (1 - p_n)$
end if

Correctness:
Let $P(k, n)$ be the probability that we get k heads for n biased coins, then P must be the summation of $P(k-1, n-1) * p_n + P(k, n-1) * (1 - p_n)$.
***Proof*:**
Since tossing a coin only has two results: head or tail, the result of the last toss which is tossing the n-th coin must fall in head or tail. The probability that we have k heads after this toss is the summation of probability that we have k-1 heads before and have a head for the n-th toss and the probability that we have k heads before and have a tail for the n-th toss.
Subproblem:
Let $P(i, j)$ $(j \geq i)$ be the probability that we get i heads for j biased coins, then there could be three conditions:
(1) i=0, which means no heads, the probability is $\prod_{i=1}^{j}(1 - p_i)$

(2) i=j, which means all heads, the probability is $\prod_{i=1}^{j} p_i$
(3) otherwise $P(i,j) = P(i-1,j-1) * p_j + P(i,j-1) * (1-p_j)$
Recurrences:

$$P(i,j) = \begin{cases} 1 & \text{if } j = i = 0 \\ \prod_{i=1}^{j} p_i & \text{if } j = i > 0 \\ \prod_{i=1}^{j} (1-p_i) & \text{if } j > i = 0 \\ P(i-1,j-1) * p_j + P(i,j-1) * (1-p_j) & \text{if } j > i > 0 \end{cases}$$

Boundary Conditions: $P(0,0) = 1$, $P(0,j) = \prod_{i=1}^{j}(1-p_i)$, $P(j,j) = \prod_{i=1}^{j} p_i$
Computation Order:
Since we assume $j \geq i$, we only consider the upper triangle. Let M be the triangle whose two sides are $n+1$ and $k+1$, we need to consider the boundary first, so we start with the diagonal and then fill the first row. Next we can fill in each entry of the rest part column by column to compute $P(k,n)$.
Running Time:
The algorithm needs to compute a triangle part of subproblems and for each subproblem we need constant time, so the running complexity is $O(kn)$.
Space:
The space complexity is $O(kn)$ since we need to store all the entries in the triangle.

Problem 2

If $s[1, ..., n]$ is a sequence of valid words, the last valid words must start from k, where $1 \leq k \leq n$, so we can design a DP to find if the string is a sequence of valid words.
Pseudocode:

Val(n)

Initialize $V[0] = 1$
for $i = 2$ to n **do**
$V[i] = \max_{0 \leq k \leq i-1} \{min\{V[i-k-1], dict(s[i-k:i]\}\}$
end for
return $V[n]$

Correctness:
If $s(1, n)$ is a sequence of valid words which means $Val(n) = 1$, then $Val(n) = \max_{0 \leq k \leq n-1} \{min\{Val(n-k-1), dict(s[n-k:n]\}\}$.
***Proof*:**
It is obvious that if the string is a sequence of valid words, then it can be extracted the last valid words and the left part will get 1 if put into the Val function or the string itself is a valid word.
Subproblem:
Let $Val(i)$ represents if a string of $s[1, ..., i]$ is a sequence of valid words. It returns 1 if $s[1, ..., i]$ is a valid word itself or it can be divided into two part that both return 1 when calling Val function and dict function, otherwise it is not a sequence of valid words.

$$Val(i) = \max_{0 \leq k \leq i-1} \{min\{Val(i-k-1), dict(s[i-k:i]\}\}$$

Recurrences:

$$Val(i) = \begin{cases} 1 & \text{if } i = 0 \\ \max_{0 \leq k \leq i-1} \{min\{Val(i-k-1), dict(s[i-k:i]\}\} & \text{if } i > 0 \end{cases}$$

Boundary Conditions: $Val(0) = 1$
Computation Order: We compute $V[i]$ from $i = 0$ to n.
Running Time:
We only have a for loop in the algorithm and in the loop we need to $O(n)$ time to compare items. To conclude, we fill in the row to compute $Val(n)$ which takes $O(n^2)$ time.
Space:
The space complexity is O(n) since we need to store all the entries in V.
Pseudocode:

Traceback(n)

```
if n = 0 then
  return
else
  for k = 0 to n − 1 do
    if V[n − k − 1] = 1 and dict(s[n − k : n]) = 1 then
      Traceback(n-k-1)
      Output s[n − k : n]
    end if
  end for
end if
```

Problem 3

We should put tasks with higher f_i in the supercomputer before tasks with lower f_i.
Pseudocode:

Process(P, F)

Relate task numbers i=1,2,...,n to each element in F
Mergesort(-F) and output the related task numbers
$Order \leftarrow$ task numbers after sorted by -F
$Start\_Point = []$, Initialize $Start\_Point[1] = P[Order[1]]$
for $i = 2$ to n **do**
  $Start\_Point[i] = Start\_Point[i-1] + P[Order[i]]$
end for
$Duration = \max_{1 \le i \le n} \{Start\_Point[i] + F[Order[i]]\}$
return Order, Duration

Correctness:
We can divide the total time we use into two parts: time for supercomputer and extra time for processors. No matter what the ordering is like, the time for supercomputer will be equal to $\sum_{1 \le i \le n} p_i$ since it must preprocess all the tasks. Now our goal is to minimize the extra time use for processors, so we put tasks needing more time in the processor to be processed first in order to let them end early. Otherwise, if we put tasks needing more processing time late, it will slow down the whole process.
Running Time:
The algorithm needs to do a merge sort, which takes $O(n \log n)$ time. The for loop, comparison, and other operations take $O(n)$ time, so the running complexity is $O(n \log n)$.
Space:
The space complexity is $O(n)$ since we need to store several variables of length n.

Problem 4

Since we want to find if we can divide $a_1, a_2, ..., a_n$ into three parts that the sum of all element in these part are equal, we can translate the requirement to that three baskets P_1, P_2, P_3 each having a capacity of $\sum_{i=1}^{n} a_i/3$ can contain all the elements. Also, for the last item a_n, it must be in one of the three part, so we can judge if $a_1, a_2, ..., a_n$ satisfies the requirement by checking out if three baskets P_1, P_2, P_3 both having a capacity of $\sum_{i=1}^{n} a_i/3$ and one having a capacity of $\sum_{i=1}^{n} a_i/3 - a_n$ can contain all the elements. To simplify, we only use two baskets for this question

Subproblem

OPT(i,j,k) represents if there are two disjoint subsets I,J satisfying the sum of I is i and the sum of J is j with the k items. OPT(i,j,k)=1 means yes, otherwise no.

Recurrence

The recurrence relationship is

$$OPT(i,j,k) = \max\{OPT(i,j,k-1), OPT(i-a_k,j,k-1), OPT(i,j-a_k,k-1)\}$$

where $i - a_k \geq 0$, $j - a_k \geq 0$.

Base case: OPT(0,0,k)=1, OPT(i,j,0)=0

Pseudocode:

Partition(A)

```
Initialize K[0][0][]=1, K[i][j][0]=0
n← length of A
sum = Σ_{i=1}^{n} a_i/3
for i=0 to sum do
  for j=0 to sum do
    for k=0 to n do
      K[i][j][k] = max{K[i][j][k-1], K[i-a_k][j][k-1](if i-a_k ≥ 0), K[i][j-a_k][k-1](if j-a_k ≥ 0)}
    end for
  end for
end for
return K[sum][sum][n]
```

Computing Order

We should first initialize K[0][0][]=1, K[i][j][0]=0 and then fill the 3-d matrix from the last dimension k to the second dimension j and then to the first dimension i.

Correctness:

If we can find three disjoint subsets as stated in the problem, then the last item must fall in one of the three basket. Equivalently, $OPT(i,j,k) = 1$ which means yes when at least one of $OPT(i,j,k-1), OPT(i-a_k,j,k-1), OPT(i,j-a_k,k-1)$ is equal to

1.

Running time:

The total running time inside the for loop is $O(n(\sum_{i=1}^{n} a_i)^2)$, which will dominate running time outside the for loop, so the running time is $O(n(\sum_{i=1}^{n} a_i)^2)$.

Space complexity:

The additional space in this algorithm is the $O(n(\sum_{i=1}^{n} a_i)^2)$ space for K.

The space complexity is $O(n(\sum_{i=1}^{n} a_i)^2)$

Problem 5

a

The probability that the process of analyzing the data will be completed successfully is $P_a = s_1 * s_2 * ...s_n = \prod_{i=1}^{n} s_i$

b

i.

The probability that task i will succeed is $P_i = 1 - (1 - s_i)^{p_i}$.

ii.

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

iii.

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

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

where k satisfy that $j - kc_i \geq 0$.
So the problem is equal to finding the OPT(n,D)
We use K[i][j] of size (n+1,D+1) to store OPT(i,j).
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), 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 multiplication of maximum probability for task 1 to i-1 and probability for task i, under the specific allocation of money. Since every best distribution for tasks before i with money less than j is correctly known, the maximum probablity is also correct for OPT(i,j).
Conclusion:

The OPT(i,j) in the algorithm is true for every $i \in [0, n], j \in [0, D]$

Computing Order

We should fill the matrix K column by column after initialization.

Runnig time:

The total number of running time in the for loop is $O(nD^2)$, which will dominate running time outside the for loop, so the running time is $O(nD^2)$.

Space complexity:

The additional space in this algorithm is the $O(nD)$ space for K and Dist, so the space complexity is $O(nD)$.

Pseudocode:

Max_Prob(s, c, D)

```
Initialize K[i][j] for all entries
Dist[i][j] ← best p_i with j dollars , initialized to be 0
n ← length of s
for i from 0 to n do
  for j from 0 to D do
    for k from 1 to ⌊j/c[i]⌋ do
      Prob = K[i − 1][j − c[i] ∗ k] ∗ (1 − (1 − s[i])^k))
      if prob>K[i][j] then
        K[i][j]=prob
        Dist[i][j] = k
      end if
    end for
  end for
end for
return Dist[1:n][D]
```