Algorithms and Complexity Mastery test 1

Ville Wassberg

February 2024

1 Ad Nauseam

In order to arrange the machines such that they are running at the same time we can analyse it one at the time. In the base case $n = 1$, then obviously all the machines are working at the same time instant, which happens directly after the restart time. If $n = 2$, the earliest possible time for the second machine to begin restarting is at the same time instant as the restart duration for the first machine. Thus the earliest possible time instant for machine 2 to begin functioning is at time $t = 2r$, if the initial time is $t_0 = 0$. This means that in order for the first machine to work at the same time as the second it has to have a function time $f_1 > r$. We can continue with 3 machines and see that the first started machine must have a function time $f_1 > 2r$ and $f_2 > r$. This indicates a pattern to be used. Since the minimum restarting time for n number of machines increases linearly with respect to n, it makes sense to sort the machines by running time in descending order, using merge sort for instance, in $\mathcal{O}(n \log n)$, and then check if the running time of $f_k \in sorted(f_i)_{i=1}^n$ is greater than $(n - k)r$.

Algorithm 1 Ad Nauseam

Require: $n \geq 1$, $f_i \geq 1$
  function MACHINEFUNCTIONING(n, r, $(f_i)_{i=1}^n$):
    if $n = 1$ **then**
      return true
    end if
    merge sort $(f_i)_{i=1}^n$ in descending order
    for $i = 1$ **to** $n - 1$ **do**:
      if $f_i \leq (n - i)r$ **then**
        return false
      end if
    end for
    return true
  end function

The time complexity is then $\mathcal{O}(n \log n) + \mathcal{O}(n) = \mathcal{O}(n \log n) \leq \mathcal{O}(n^{1.5})$ for the

merge sort and the for loop. To prove the correctness of the algorithm we can use a proof by induction.

Proof. Base case: $n = 1$. Always true. For $n = 2$: If $f_1 \leq r$ it outputs false correctly; if not it returns true correctly since $f_2 \geq 1$. Suppose the algorithm is correct for some integer k machines, then it has correctly stated that it can be done by the k first machines or if it can not. If it is true, then for the $(k+1)^{th}$ machine we check that $f_{k+1} > (n-(k+1))r = (n-k)r - r$ which is correct. If it output false for k machines, it makes no difference for the next, since not all machines are running simultaneously. □

2 Caveat Venditor

Given n movies that we want to buy, a constant cost C for each movie and m number of coupons saying "Buy b_i movies, get f_i of them for free", for $i \in \{1, 2, ..., m\}$, we want to desgin an algorithm minimising the cost of buying n movies. Considering this, we can start by supposing $\mathcal{O}$ is an optimal solution. Starting at the m^{th} coupon, we know that either it belong to the optimal solution or it does not. If m does belong to $\mathcal{O}$ then it leaves us with solving the problem of buying $n - b_m$ movies as a low cost as possible and we add the cost of buying the b_m movies including the discount f_m, which then is $(b_m - f_m) \cdot C$. Then we know that it is necessary for the next coupon $m-1$ that $b_{m-1} - f_{m-1} < n - b_m$, otherwise that coupon is discarded. So, if m is in $\mathcal{O}$, then we also must have an optimal solution for the $n - b_m$ movies left that we want to buy. This indicates that the problem could be solved by dividing it up into subproblems, hence, dynamic programming and memoisation can be used to solve this. We begin by defining $Opt(k, j) = 0$ if $k \leq 0$ and $Opt(k, 0) = k \cdot C$ for $k \in \{1, 2, ..., n\}$ and $j \in \{1, ..., m\}$ since the minimum cost of buying 0 or negative amount of movies is 0 and the minimum cost of buying k movies without any discounts is $k \cdot C$. Then for the subsets, $\{1, ..., j\}$, $j \leq m$ we solve the problem as above but for j, i.e. if $j \in \mathcal{O}$ then $Opt(n, j) = (b_j - f_j) \cdot C + Opt(n - b_j, j - 1)$ or if $j \notin \mathcal{O}$ then simply $Opt(n, j) = Opt(n, j - 1)$. This sums up to compute the following:

$$opt(i, j) = \min_{\substack{0 \leq i \leq n \\ 0 \leq j \leq m}} \{(b_j - f_j) \cdot C + Opt(i - b_j, j - 1), Opt(i, j - 1)\}. \quad (1)$$

Algorithm 2 Caveat Venditor

Input: n number of movies, m number of coupons, C cost of one movie, $(b_k, f_k)_{k=1}^{m}$ each coupon
$M \leftarrow$ Array
Require: $j \geq 0$
  function CHEAP:
    if $i \leq 0$ **then**
      return 0
    else if $j = 0$ **then**
      return $i \cdot C$
    else if $M[i] \neq \varnothing$ **then**
      return $M[i]$
    else
      $M[i] \leftarrow \min\{(b_j - f_j) \cdot C + \text{CHEAP}(i - b_j, j - 1), \text{CHEAP}(i, j - 1)\}$
    end if
    return $M[i]$
  end function

The time complexity of this algorithm is $\mathcal{O}(mn)$ since the slowest possible recursive call would decrease by one each time in both m and n direction, calling the two ends up in $m * n$. Next we try to prove the correctness of the algorithm, which then is enough to prove the equation 1.

Proof. Again, proof by induction can be handy. Base case: if $i = 0, j \geq 0$ then the algorithm returns 0 correctly. If $i = 1, j = 0$ it returns C correctly. $i = 1$, $j = 1$ then $\min\{(b_1 - f_1) \cdot C + Opt(1 - b_1, 0), Opt(1, 0)\} = \min\{(b_1 - f_1) \cdot C + 0, C\} = C$ or 0 if $f_1 = b_1$ since $1 \leq f_1 \leq b_1 \leq i$ which is correct. Now suppose that the algorithm works for all i such that $1 \leq i \leq k$ for some k and any $1 \leq j \leq m$. To show that the algortihm works for $i = k$, we have 2 cases.

1. Coupon m is not used in the solution.
   Then the cost of using m coupons is the same as using $m - 1$ coupons, which the algorithm does coerrectly by checking $Opt(k, j - 1)$.

2. Coupon m is used in the optimal solution.
   Then the algorithm uses this coupon and thus a discount of $f_m \cdot C$ is accounted for while we get b_m movies which leaves us with $k - b_m$ movies left to buy and adding a cost of $(b_m - f_m) \cdot C$ and checks $Opt(k - b_m, m - 1)$ as the induction hypothesis.

Therefore, since the minimal solution must either use the m^{th} coupon or it does not, the minimum between these two must ouput the optimal solution. □

3 Coniunctis Viribus

In order to determine the fewest number of trains given the n ordered carriages with corresponding capacities x_i, $i \in \{1, ..., n\}$ with the constraint that the cumulative capacity of a train $\sum_{1 \leq i \leq k \leq j \leq n} x_k \geq (j - i + 1)^2$, one can start by noticing a few facts. The last carriage C_n belongs to some train in the optimal solution that begins at som point $i \leq n$, i.e. we have a train constituted by the ordered partition $(C_i, ..., C_n)$. This means that we have one train and are left with deciding the next $i - 1$ carriages to form a train that fulfills the constraint. We need a boundary constraint $Opt(0) = 0$, since an empty set of carriages can create 0 trains. In other words, if the last segment of carriages of the optimal partition is $C_i, ..., C_n$, then the optimal solution minimising the amount of trains is given by the equation:

$$Opt(n) = 1 + Opt(i - 1). \tag{2}$$

Now, making this a subproblem task, we can for the remaining $C_1, ..., C_j$ carriages create a minimum of $1 + Opt(i - 1)$, $1 \leq i \leq j$. In other words:

$$\begin{aligned} &\text{For the subproblem of the carriages } C_1, ..., C_j, \\ &Opt(j) = \min_{1 \leq i \leq j} \{1 + Opt(i - 1)\}, \\ &\text{and the train } (C_i, ..., C_j) \text{ is used in an optimum solution} \\ &\text{if and only if the minimum is obtained using index } i. \end{aligned} \tag{3}$$

Now, we can develop the algorithm.

Algorithm 3 Coniunctis Viribus

Require: $n \geq 1$

```
cache ← array initialised with ∞
function SegmentedTrains(n, x)
    cache[0] ← 0
    for k = 1 to n do
        weight[k] ← k^2  ▷ Constraint (j − i + 1)^2 for every pair 1 ≤ i ≤ j ≤ n
    end for
    for j = 1 to n do
        for i = 1 to j do
            if Σ_{k=i}^{j} x_k ≥ weight[j − i] then
                cache[j] ← min{1 + cache[i − 1], cache[j]}
            end if
        end for
    end for
    if cache[n] = ∞ then
        return 0
    end if
    return cache[n]
end function
```

This algorithm has a polynomial time complexity $\mathcal{O}(n + n^2) = \mathcal{O}(n^2)$ considering unit cost, since its dynamic approach stores all previously best combinations and does not have to compute them over and over. Next is to prove the correctness of the algorithm.

Proof. Using equation 3 as the induction step, we get a base case: $Opt(0) = 0$ which the algorithm saves in cache[0] correctly; $Opt(1) = \min\{1 + 0\} = 1$ if it fulfills the capacity threshold, else 0, which is correct. Suppose the algorithm has correctly determined for $1 \leq j = k < n$, then for $j = k + 1$ the algorithm checks all possible combinations of carriages of carriages $c_1, ..., c_{k+1}$, and if a combination $c_i, ..., c_k + 1$ fulfills the capacity then it adds 1 for the possible train and checks how many trains the optimum combination of carriages $c_1, ..., c_{i-1}$ was and compares this to the current value of cache[k+1] which either was a better combination or not, and records the minimum, correctly. Finally, the algorithm checks if $cache[k+1]$ is infinity which would mean that no combination of carriages fulfilled the capacity threshold and in that case it returns 0 correctly. Thus, the algorithm is correct. □

4 Quid Pro Quo

We are given n resource types, m characters apart from myself (or the player), r_{ij} units of resource type i that character j initially owns, w_{ij} units of resource type i that character j wants and $h_i \geq 0$ units of resource i that the player initially have. The rules are that character j will trade her resource type i' for the resource type i if and only if $r_{i'j} > w_{i'j}$ and $r_{ij} < w_{ij}$, the player can trade as she likes, and all trades must be done with the player as a middleman and that the player wants to end up with a complete collection; at least one of each resource type. This has its traits of a linear programming problem, however, it can also be modeled with a graph and using network flow to solve it. If we begin by model the problem by creating a graph of vertices representing the characters and the resources. Next we connect the characters with the resources by a directed edge e_{ij} between resource i and character j with a capacity of the demand $d = |w_{ij} - r_{ij}|$ with a tail at the resource if $w_{ij} > r_{ij}$, tail at the character if $w_{ij} < r_{ij}$ and no edge if the demand is zero. Next, we can add a sink node with directed edges of capacity 1 from each resource representing the goal of a complete collection (we do not care if we can get more than one of some resource). Finally, we can add a source representing the initial resources the player have drawing an edge from the source to to resource i of capacity h_i for all non-zero h_i's. This would model the problem correctly since the number of resources that can be traded is those the player initially have, and if a character demands a resource and has another in surplus then we can push a flow from the player through the resource to the character who in turn pushes further to the surplus resource which then is connected to the sink and to another character there are one who has a demand for it. This has now become a maximum flow problem where the output will be that it is possible to achieve a complete collection if the maximum flow of the graph equals to n. To check this, one can run the Ford-Fulkerson algortihm on the buildt network in $\mathcal{O}(C(m+n))$, where C is the sum of capacities from the source, i.e. $\sum_{i=1}^{n} h_i$, and m and n here are the number of edges and vertices, respectively.

Algorithm 4 Quid Pro Quo

▷ Let r be the matrix $(r_{ij})_{i,j=1}^{n,m}$, w the matrix $(w_{ij})_{i,j=1}^{n,m}$ and $h = (h_i)_{i=1}^{n}$.
function QUIDPROQUO(n, m, r, w, h)
$E \leftarrow$ set of edges, initially empty
$V \leftarrow$ set of vertices, initially with source s and sink t
for i = 1 **to** n **do**
$V \leftarrow$ add a vertex for resource i
$E \leftarrow$ add a h_i-capacity edge from source s to resource i and an edge with capacity 1 from resource i to the sink t
end for
for j = 1 **to** m **do**
$V \leftarrow$ add a vertex for character j
end for
for i = 1 **to** n **do**
for j = 1 **to** m **do**
if $D[i, j] < 0$ **then**
$E \leftarrow$ edge from resource i to character j with capacity $|D[i, j]|$.
else if $D[i, j] > 0$ **then**
$E \leftarrow$ edge from character j to resource i with capacity $|D[i, j]|$.
end if
end for
end for
$G \leftarrow$ The pair of vertices V and edges E.
return FORD-FULKERSON(G)
end function

The complexity of the algorithm is thus $\mathcal{O}(n+m+nm+C((m+n)+(2n+mn)))$, where $m + n$ is the number of vertices and $2n + mn$ is an upper bound of the number of edges, and since we can set an upper bound on C with the number of resources n, the algorithm is thus running in polynomial time of n and m, $\mathcal{O}(n+m+nm+n((m+n)+(2n+mn))) = \mathcal{O}(n+m+2nm+3n^2+mn^2) = \mathcal{O}(mn^2)$. The correctness proof of this algorithm relies on the correctness of the Ford-Fulkerson algorithm, which is allowed to be assumed in this context.