# Homework 1
# class in "Algorithm Design", Fall 2016/17

Marco Favorito
Master of Science in Engineering in Computer Science
Department of Computer, Control, and Management Engineering
University of Rome "La Sapienza'
favorito.1609890@studenti.uniroma1.it

November 18, 2016

## Contents

# 1 Exercise 1

The problem of the Exercise 1 has some elements that suggests us to approach it with dynamic programming principles. The main motivations of this observation are the following:

1. There are some common features between this problem and *knapsack problem*: indeed there are some *items* (*probabilities* $p_1, p_2, \ldots, p_n$) we have to "choose" [1], and each of them having a certain *weight* $w$ (in our case, $w_i = 1 \; \forall i = 1, 2, \ldots, n$) and *value* $v$ ($p_i$ if we choose that $i_{th}$ coin has to be `Head`, $1 - p_i$ otherwise). Moreover, in our choice we make we have to take account on the maximum capacity of the "knapsack" (in our case $k$). For completeness we should notice that, unlike knapsack problem, there is not a *objective function* on values to maximize or minimize, but we have to compute something, i.e. $P(n, k)$. Moreover, we have to check *all* possible combinations of items with cardinality $k$, and not only one.

2. The problem has a recursive nature and can be divided into more subproblems. For each coin we can consider that its toss either turns out `Head` or `Tail` and solve the problem for the rest of coins. We can build a naïve solution of this procedure in the following way:

   $$\texttt{Recursive-P}(n, k, \mathbf{p}) =$$

   $$p_i\texttt{Recursive-P}(n-1, k-1, \mathbf{p}) + (1-p_i)\texttt{Recursive-P}(n-1, k, \mathbf{p})$$

   Where `Recursive-P` is a recursive procedure, $\mathbf{p}$ is the vector containing $p_1, p_2, \ldots, p_n$ and $i$ is the coin that the procedure is considering. Further details will be covered in the following section.

3. The structure of expected result equations suggests that the subproblems just described have some "overlapping expressions". For instance, consider the following expression:

   $$P(3,1) = p_1(1-p_2)(1-p_3) + (1-p_1)p_2(1-p_3) + (1-p_1)(1-p_2)p_3$$

   it is evident that some subgroups of addends are repeated in others: this is a peculiarity of many expression of $P(n, k)$.

4. The running time constraint $\mathcal{O}(nk)$ remainds at the pseudo-polynomial solution of knapsack problem and suggests the same approach, as showed in [1].

[1] "Choose $p_i$" or "Choose coin $c_i$" is the same of saying "the toss of $i_{th}$ coin give us a `Head`.

The following section provide a proposed algorithm for solve the problem and a description about it.

## 1.1 Proposed algorithm and description

Fist we will see the recursive implementation of the algorithm (see motivation #2 in the previous paragraph) and a iterative one that exploits the memoization technique (in oder to avoid computation redundancy explained in motivation #3) and thus uses a bottom-up approach memorizing all partial result.

**Notice** that from now to the end we will use the following notation $\bar{p}$ in order to denote $(1 - p)$.

### 1.1.1 Recursive version of the algorithm

**Algorithm 1** Compute recursively the probability of $k$ `Head` on $n$ coin tosses

**Require:** $n, k \in \mathbb{N}$, $k \leq n$,$n > 0$, $p_i \in [0, 1]\ \forall i \in \{1, \ldots, n\}$
  **function** RECURSIVE-P($n, k, \mathbf{p}$)
    **Inputs:**
      $n$: (residual) number of coins/probabilities to be considered
      $k$: (residual) number of `Head` "to satisfy"
      $\mathbf{p}$: $p_1, \ldots, p_n$, where $p_i$ is the probability of `Head` for the $i_{th}$ coin

    **if** $n = 0$ **then return** $1$
    **else if** $n = k$ **then return** $\mathbf{p}.head$·RECURSIVE-P($n - 1, k - 1, \mathbf{p}.tail$)
    **else if** $k = 0$ **then return** $\bar{\mathbf{p}}.head$·RECURSIVE-P($n - 1, k, \mathbf{p}.tail$)
    **else**
  **return** $\mathbf{p}.head$·RECURSIVE-P($n - 1, k - 1, \mathbf{p}.tail$) + $\bar{\mathbf{p}}.head$·RECURSIVE-P($n - 1, k, \mathbf{p}.tail$)
    **end if**
  **end function**

Now we will try to explain in words how the core function `RECURSIVE-P` works.

Until there are still probabilities to consider (i.e. we are not at the end of the vector $\mathbf{p}$, first **if**) and until we can still "choose" positive or negative events on coin tosses (i.e. $k$ is not 0 and the number of remaining "choices" is enough to fill $k$, second and third **if**), the function returns the sum of two recursive call (the **else-return** statement), both of them multiplied by a value:

- The first recursive call represent the positive case of the toss, i.e. "the first coin of the current list turns out `Head`" and it is multiplied by the probability of this event $\mathbf{p}.head$ (i.e. the first element of the list); $k$ is decreased than 1 because for this recursive call the current coin turns out `Head`, so the remaining coins have to turn out $k-1$ positive event, `Head`. Analogously, also $n$ is decreased by 1.

- On the other hand, the second recursive call represent the negative case of the toss, i.e. "the first coin of the current list *does not* turns out `Head`" and it is multiplied by the negation of $\mathbf{p}.head$, i.e. $\mathbf{p}.\bar{head}$; in this case $k$ is not decreased because the current coin *does not* turns out `Head`, so the remaining coins have to turn out still $k$ `Head`. As the former case, $n$ is decreased by 1.

Obviously, we must stop the recursive calls when $n = 0$, because it means that we have already considered all the $n$ coins (notice that $n$ starts from value 1 and it is only incremented by 1 at each recursive call), and we can return a neutral value for the recursively-built products. This instruction ensure that the algorithm terminates since we decrease $n$ by 1 at each recursive call and $n > 0$: it means that there will be a moment in which $n = 0$ and the **return** instruction stops the recursive call stack.
The second and the third **if** checks if we can still "freely" choose $k$ `Head` from remaining coins.
About the former: at the beginning of the function, the number of remaining coins is $n$. If this number is equal to $k$, there are no other ways to choose $k$ `Head` than choose all the remaining $n = k$ coins as `Head`, since choosing them in a different way (i.e. at least one of the $n$ coin turns out "not `Head`") will not be what we are looking for. So the only admissible turn out for the current coin is `Head`. It means that the return of this subproblem will be only the positive case, and this case will be the only one returned for all the following recursive calls in the current recursive call stack.
For the third **if** will be analogous considerations: if $k = 0$, the only way to choose the remaining $n$ coins is `Tail`, because $k$ `Head` has been already considered. It means that the return of this subproblem will be only the negative case, and this case will be the only one returned for all the following recursive calls in the current recursive call stack.

Notice that, if $k > 0$, we are sure that the second and third **if** will be executed at least one, since $n \geq k$ and at each recursive call we decrease $n$ by 1 and we decrease $k$ when consider the positive case. This observations proof immediately that the algorithm terminates.

In order to show the correctness and to provide $\mathcal{O}(nk)$ execution time, we will apply *memoization* technique over Algorithm 1 as shown in [1] for the ***Knapsack problem***.

### 1.1.2 Memoized and iterative version of the proposed algorithm

**Algorithm 2** Compute iteratively the probability of $k$ `Head` on $n$ coin tosses

**Require:** $n, k \in \mathbb{N}$, $k \leq n$,$n > 0$,$p_i \in [0,1]\ \forall i \in \{1,\ldots,n\}$
  **function** P($n, k, \mathbf{p}$)
    **Inputs:**
      $n$: number of coins/probabilities to be considered
      $k$: number of `Head` "to choose"
      $\mathbf{p}$: $p_1, \ldots, p_n$, where $p_i$ is the probability of `Head` for the $i_{th}$ coin
    **local variables:** $M$, a $n \times k$ matrix

    $M[0][0] \leftarrow 1$
    **for** $i$ from 1 to $n$ **do**
      **for** $j$ from $max(0, i-(n-k))$ to $min(i,k)$ **do**
        **if** $j = i$ or $j = k$ **then**
          $M[i,j] \leftarrow \mathbf{p}[i] \cdot M[i-1][j-1]$
        **else if** $j = 0$ **then**
          $M[i,j] \leftarrow \bar{\mathbf{p}}[i] \cdot M[i-1][j]$
        **else**
          $M[i,j] \leftarrow \mathbf{p}[i] \cdot M[i-1][j-1] + \bar{\mathbf{p}}[i] \cdot M[i-1][j]$
        **end if**
      **end for**
    **end for**
    **return** $M[n,k]$
  **end function**

The instructions in the inner **for** are very similar to instructions in the body of the Algorithm 1. To understand how it works, it will be enlightening to look at the following figure:

| | 0 | 1 | 2 | ⋯ | j | ⋯ | k |
|---|---|---|---|---|---|---|---|
| 0 | 1 | 0 | ⋯ | | | | |
| | $\overline{p_1}$ | $p_1$ | 0 | ⋯ | | | |
| 2 | $\overline{p_1}\overline{p_2}$ | $p_2\overline{p_1} + \overline{p_2}p_1$ | $p_2p_1$ | 0 | ⋯ | | |
| ⋮ | ⋮ | ⋮ | ⋱ | ⋱ | 0 | ⋯ | |
| $i-1$ | ⋮ | ⋮ | | ⋱ | $\prod_{h=0}^{j} p_h$ | 0 | ⋯ |
| i | $\prod_{h=0}^{i} \overline{p_h}$ | $p_iM[i-1][0] + \overline{p_i}M[i-1][1]$ | ⋯ | ⋯ | $p_iM[i-1][j-1] + \overline{p_i}M[i-1][j]$ | ⋱ | 0 |
| ⋮ | ⋮ | ⋮ | | | | ⋱ | $\prod_{h=0}^{k} p_h$ |
| n | $\prod_{h=0}^{n} \overline{p_h}$ | $p_nM[n-1][0] + \overline{p_n}M[n-1][1]$ | ⋯ | | | ⋯ | $p_nM[n-1][k-1] + \overline{p_n}M[n-1][k]$ |

The figure represents the state of matrix $M$ at the end of the computation. The bottom-up approach is evident: the procedure starts to compute smaller subproblems (it starts from $n = 1$ and $k = 0$) and stores the results in $M[i][j]$, where $i = n$ and $j = k$, in order to be available when, computing the greater subproblems, is required the value of some expressions that have already been faced.

## 1.2 Running time analysis

At the end of the day, the procedure does many sums and products, each of them costing $\mathcal{O}(1)$. An obvious upper-bound on the number of iteration is $\mathcal{O}(nk)$: just looking at indexes in both for-loops.

To go in more detail, it is worth to notice that the start value of $j$ in the inner for-loop is $max(0, i-(n-k))$, and the upper-bound value is $min(i, k)$. It is based on the following observation: since the final result resides in $M[n][k]$, we have to compute only the subresults in $M[n-1][k-1]$ and $M[n-1][k]$ and, recursively speaking, $M[i-1][j-1]$ and $M[i-1][j]$. In other words, we need to compute only $n-k+1$ diagonals of values: the main diagonal and the "lower subdiagonal" until the last subdiagonal that covers $M[n, k]$, i.e. $M[i, j]|0 \leq i-j \leq n-k$ (reasoning with indexes). The previous disequalities are rewritable as the following:

$$i \geq j \geq i - (n - k)$$

Obviously, $i-(n-k)$ could be lower than 0, which if it is the case it doesn't make much sense, so we consider the maximum between 0 and that value. Analogously, $i$ could be greater than $k$, and when $i > k$ we must stop to $k$, and so we have explained also $min(i, k)$.
Now, the total time cost is:

$$1 \cdot (k) + 1 \cdot (n - k) + 3 \cdot (k - 1)(n - k)$$

where the first addend is due to the main diagonal elements times the cost for compute these elements (only a multiplication); the second element is due to the $n - k$ cells on the first column times the cost of a multiplication (like the former); the third element is due to $n - k$ diagonals of $k - 1$ elements (minus 1 because the first-column elements have been already considered) times 2 multiplication and 1 sum. After doing algebriac calculations, we can state that:

$$3nk - 3k^2 - 2n + 3k = \mathcal{O}(nk)$$

since $k \leq n \Rightarrow k^2 \leq nk$.

## 1.3 Formal proof of correctness

We will prove the correctness of iterative algorithm by *loop invariants*.

*Proof.* The invariant is: at iteration $i_{th}$, the result of computation $P(i,k)$ is stored in $M[i,k]$.
**Initialization:** the first iteration is at indexes $i = 1$; $k$ can be 0 or 1. In the former, the result is in $M[1,0]$ where is stored $\bar{p}_1$, which is actually the correct computation. Also for $k = 1$ we have $M[1,1] = p_1 = P(1,1)$.
**Maintenance:** Given the result of $P(i,k)$ correctly computed and stored in $M[i,k]$ for all integers smaller than or equal $i$ and $k$ (*). We want to prove both the following propositions:
(1): for $i+1$ and $k$ the result of $P(i+1,k)$ is in $M[i+1,k]$;
(2): for $i$ and $k+1$ the result of $P(i,k+1)$ is in $M[i,k+1]$;

- (*)$\Rightarrow$ (1)
  Since $k \leq i$, we have $k < i+1$.

  - If $k = 0$, $M[i+1,k] = \bar{p}_{i+1}M[i,k]$, where $M[i,k]$ for assumption is computed correctly and the multiplication by $\bar{p}_{i+1}$ represents the possible choice of the $(i+1)_{th}$ toss turning up `Tail`.
  - Otherwise, $k > 0$ and, since $k < i+1$ the only possibility is $M[i+1,k] = p_{i+1}M[i,k-1] + \bar{p}_{i+1}M[i,k]$. $M[i,k-1] = P(i,k-1)$ and $M[i,k] = P(i,k)$ are correct for assumption. The first element is muiltiplied by $p_{i+1}$ because the addend $p_{i+1}M[i,k-1]$ represents the event "the $(i+1)_{th}$ coin turns up `Head`" times the possible results of the remaining events, $P(i,k-1)$. The second element is multiplied by $\bar{p}_{i+1}$, since represents the opposite choice.

- (*) $\Rightarrow$ (2).
  Since $k \geq 0$, we have $k+1 > 0$.

  - If $k+1 = i$, $M[i,k+1] = p_i M[i-1,k]$. This expression is correctly computed, because if we have $P(i,i)$, the only possibility is to choose that the $i_{th}$ toss turns up `Head` and multiply the probability of the choosen event ($p_i$) with the subproblem $P(i-1,k=i-1) = M[i,k]$, correct for assumption.

  - Otherwise $k+1 < i$, and surely $k+1 \neq 0$. It follows that, for $k+1$, the only possibility is $M[i,k+1] = p_i M[i-1,k] + \bar{p}_i M[i-1,k+1]$.

The first addend is correct, since $M[i-1,k] = P(i-1,k)$ by assumption, and the multiplication for $p_i$ represents the possible choose of the $i_{th}$ toss turning up `Head` (as said before).

The second addend represents the opposite choice. Since $i-1 < i$, until the first index $i-1$ is different from $k+1$, $M[i-1,k+1]$ is splitted in two addends, in which the first one, as we have just seen, is always computed correctly, and the second one is in the form $p_{\hat{i}}M[\hat{i}-1,k+1]$, where the generic index $\hat{i} < i$. There must exists a point in which $\hat{i} = k+1$ (since we start from $i > k+1$ and we decrease $i$ by one at each "recursion"); at this point, $M[\hat{i},k+1] = p_{\hat{i}}M[\hat{i}-1,k]$, which is correctly computed by assumption (as we seen in previous case) and so also all the greater subresults $M[i,k+1], M[i-1,k+1], \ldots M[\hat{i}+1,k+1]$.

Therefore the invariant is preserved both for the $(i+1)_{th}$ iteration of the outer for-cycle and for the $(k+1)_{th}$ iteration of the inner for-cycle.

**Termination:** At the last iteration the result is stored in $M[n,k]$, which is the returned value. □

# 2 Exercise 2

From now, we will use the following notation: $m = |E|$ and $n = |V|$.

## 2.1 Proposed algorithm

**Algorithm 3** Compute a max flow $f'$ from $f$ with new capacities $\hat{c}$ in $G$

**function** MAX-FLOW-$\hat{c}$($G, s, t, \hat{c}$)
    **Inputs:**
    $G$, a graph $G = (V, E)$
    $s$ and $t$, source and sink node
    $\hat{c} : E \to \mathbb{N}$, new capacity function
    $f : E \to \mathbb{N}$, current max-flow
    **Output:** $f'$, new max-flow
    **local variables:**
    $\hat{e}$, edge $\in E$ $s.t.$ $\hat{c}(\hat{e}) = c(\hat{e}) - 1$
    $\widehat{G_f}$, the residual graph of $G$ with capacity function $\hat{c}$
    $p_f$, path on $G$ from $s$ to $t$ to be decreased by one

    $f' \leftarrow f$
    $\hat{e} \leftarrow$ The edge with decreased capacity (by one)
    **if** $f(\hat{e}) \leq \hat{c}(\hat{e})$ **then**
        **do nothing**
    **else**
        $p_f \leftarrow$ Find a path $s \to t$ in $G$ with a nonzero flow such that $\hat{e} \in p_f$
        $f' \leftarrow \forall e \in p_f, f'(e) = f'(e) - 1$
        **if** There exists an augmenting path $p$ in $\widehat{G_f}$ **then**
            $f' \leftarrow$ AUGMENT($f', \hat{c}, p$)  ▷ See cap. 7.1 in [1]
        **end if**
    **end if**
    **return** $f'$
**end function**

## 2.2 Proof of correctness

In the following we will provide a constructive proof of correctness for the proposed algorithm. We simply follows the steps of the procedure and see that, at the end, it returns a solution for the problem.

**Proposition 1.** *Possible values of flow $v(f')$.*
*Let $f'$ the new flow function returned by the algorithm. Then $v(f')$ (considering $\hat{c}$) is either $v(f)$ or $v(f) - 1$, and it will be the maximum flow feasible by the flow network.*

*Proof.* If $f(\hat{e}) < c(\hat{e})$, decreasing $c(\hat{e})$ by one implies $f(\hat{e}) \leq \hat{c}(\hat{e})$, which means that $f' = f$ is still feasible by the flow network $(G, s, t, \hat{c})$, and we already know that it is the maximum flow (by assumption).
Otherwise, the only valid case is that $f(\hat{e}) = c(\hat{e})$. Since the graph is *acyclic*, we cannot find a nonzero flow in a cycle with $\hat{e}$ in it; if it would be the case, just reduce the flow by one in each edge of the cycle would be enough, and the flow $f$ would be still feasible.

In order to find the new feasible network flow, first of all we have to find a new temporary flow, $\bar{f}$, such that $\bar{f}(\hat{e}) = \hat{c}(\hat{e}) = c(\hat{e}) - 1$. In order to do it, we should find a path $p$ in $G$ from $s$ to $t$ containing $\hat{e}$ and for each $\bar{e}$ in $p$ should be true that $f(\bar{e}) > 1$.

Once we found such path, we decrease the flow by one for each edge in the path, in symbols: $\forall e \in p, \bar{f}(e) = f(e) - 1$. Now the capacity constraint for $\hat{e}$ is observed, and $\bar{f}$ is surely a feasible flow for the flow network $(G, s, t, \hat{c})$.

At this point, two cases can happen : either a minimum cut with the same minimum-cut capacity equal to $v(f)$ exists, despite the decreased capacity, or there exists a minimum cut with a lower capacity, more precisely $v(f) - 1$. In both the cases, just perform the search for an augmenting path will be enough: in the former case, such augmenting path will be found, and the nework flow value will be restored; in the latter, no augmenting path will be found, and $\bar{f}$ will be the flow function returned by the algorithm. Notice that we already know that only once the AUGMENT function (line line 20) has to be executed, since we know that at most the flow can be $v(f)$, by assumption that $f$ is a max-flow. [2]
So in both cases, in order to find maximum flow, we have to perform an augmenting path search on $G$ and, obviously, the maximum flow $f'$ will be found. □

## 2.3 Running time analysis

Now we list and analyze the critical instructions, in terms of execution time, executed by the algorithm, in order to complete the running time analysis of the entire procedure.

[2] Surely decreasing the capacity on one edge does not increase the minimum cut capacity of the network flow. Indeed, given that $\hat{e} = (u, v)$ considering all cuts, including $u$ without $v$, before and after the decreasing, respectively $(A, B)$ and $(\hat{A}, \hat{B})$. Then $\sum_{eoutof\hat{A}} \hat{c}(e) < \sum_{eoutofA} c(e)$ since for all edges $e \neq \hat{e}$ the capacity is not changed, so we can rewrite the disequality as $\hat{c}(\hat{e}) = c(\hat{e}) - 1 < c(\hat{e})$

- line 18: "Find a path from $s$ to $t$..." performing BFS or DFS, this instruction costs at most $\mathcal{O}(m+n)$;
- line 20: "There exists an augmenting path..." it takes time $\mathcal{O}(m+n)$;
- line 21: "AUGMENT($f', \hat{c}, p$)". The function runs in $\mathcal{O}(n)$ time, since $p$ has at most $n$ nodes.

The most expensive operations run in $\mathcal{O}(m+n)$ time , so all the algorithm runs in that time.

# 3 Exercise 3

## 3.1 Proposed algorithm

**Algorithm 4** Find positions for wireless signal repeaters.

**Require:** Assuming that $\mathbf{c}$ is sorted

**function** FINDREPEATERSPOSITIONS($n, \mathbf{c}, d$)
    **Inputs:**
        $n$: number of homes
        $\mathbf{c}$: $c1, \ldots, c_n$ homes' positions
        $d$: repeater's range
    **Output:**
        $\mathbf{d}$: repeaters' positions that cover all the homes
    **Local variables:**
        $x$: current max position covered by the signal
        $p_{new}$: new repeater position
        $L = c_n$: max position in the road

    $x \leftarrow 0$
    **for** $i$ from 1 to $n$ **do**
        **if** $x < c_i$ **then** ▷ if $c_i$ is not covered by the signal
            $p_{new} \leftarrow$ InstallRepeater$(c_i, d, L)$
            Add $p_{new}$ to $\mathbf{d}$
            $x \leftarrow p_{new} + d$
        **end if**
    **end for**
    **return** $\mathbf{d}$
**end function**
**function** INSTALLREPEATER($c, d, L$)
    **if** $c + d \leq L$ **then return** $c + d$
    **else return** $L$
    **end if**
**end function**

## 3.2 Proof of correctness

The proposed algorithm was designed with a greedy approach. The greedy template is: consider the smallest not-covered home position and cover it with one

repeater at the highest possible position, i.e. $c + d$[3].Intuitively, this approach seems to be the optimal solution for the problem, because maximizing the covered *useful* area of each repeater means minimizing the number of repeaters to be used for the goal. Indeed, the covered area by the repeaters found by the algorithm is entirely *useful*, since:

- at lower position than $c$ there is no uncovered home (due to the greedy template), so covering even a part of that area with the signal would be *useless*, and
- all the area covered by the signal after $c$ is potentially *useful*, because some home could exist at higher position than $c$ (we do not know it *a priori* and, actually, we do not need to know it).

With respect to above observations, mathematically speaking, we have the following problem:

$$x = \underset{h}{\operatorname{argmax}}\, h + d, \text{with } |c - h| \leq d$$

With $h$ the position of the repeater to be placed. The constraint can be rewritten like this:

$$c - d \leq h \leq c + d$$

Now it is evident that the value of $h$ that maximize the area next to $c$ (i.e. the useful area) is $c + d$. So $p_{new} = c + d$ and the new maximum position covered by the signal is $x = p_{new} + d = c + 2d$. The next uncovered position we have to consider will be at a position greater than $c + 2d$, since otherwise it would be already covered. Moreover, since we are assuming that **c** is sorted, just iterate over the list of positions will be enough, because higher uncovered positions would be at least after the just considered one.

In order to prove that the proposed algorithm is optimal, suppose by contradiction that exists another algorithm that find a correct solution with a lower number of repeater, say $R' < R$, with $R$ the number of repeater found by our proposed greedy algorithm. Since we have proved that each repeater $r_i$ found by our solution is at the highest possible position, surely $r'_i \leq r_i$ is true for each $r'_i$ repeater of the supposed optimal algorithm, with $i = 1, \ldots, R'$. Comparing one-by-one the repeaters' positions found by both the algorithm, we will arrive at the point in which $r'_{R'} < r_{R'}$ is still true; but this means that the area covered by $r_{R'+1}$ is not covered by $r'_{R'}$, the repeater at the last position of the solution of the supposed optimal algorithm. Moreover, since we know that the covered area by each

[3]Notice that this property is due to line 15, because it means that $c$ is processed only if it is uncovered by the signal and, since we assumed **c** is sorted, it is the earliest one (in terms of position).

repeater "installed" by our algorithm is entirely useful, i.e. there is at least one covered home, we infer that, in general, the supposed optimal algorithm is not correct, instead of the proposed algorithm, that yields a correct and optimal solution.

### 3.3 Running time analysis

Assuming that **c** is sorted, obviously the costs of the whole procedure is $\mathcal{O}(n)$, since the number of instruction is growing with the number $n$ of homes. Notice that if **c** is not sorted, we should sort it before start the algorithm, and the time cost grows as $\mathcal{O}(n \log(n))$.

# 4 Exercise 4

## 4.1 Proposed algorithm

**Algorithm 5** Check if it is possible to meet all demands.

**function** MEETDEMANDSWITHAVAILABILITIES($\mathbf{d}, \mathbf{r}$)
  **Inputs:**
    $\mathbf{d}$: a list containing availabilities $d_0, d_A, d_B, d_{AB}$
    $\mathbf{r}$: a list containing demands $r_0, r_A, r_B, r_{AB}$
  **Output:**
    $\mathbf{b}$: a boolean, it is true if all demands are satisfied
  **Local variables:**
    $(G, s, t, c)$: a special flow network to represent the problem (see the next section)
    $f$: max-flow=min-cut capacity

  $f \leftarrow$ Scaling-Max-Flow$(G, s, t, c)$ ▷ See [1], cap. 7.3
  **if** $v(f) = r_0 + r_A + r_B + r_{AB}$ **then**
    $b \leftarrow$ True
  **else**
    $b \leftarrow$ False
  **end if**
  **return** $b$
**end function**

## 4.2 Proof of correctness

Before proving the correctness of the algorithm, we should make some observations on the ad-hoc flow network build for this problem, shown in figure 1:

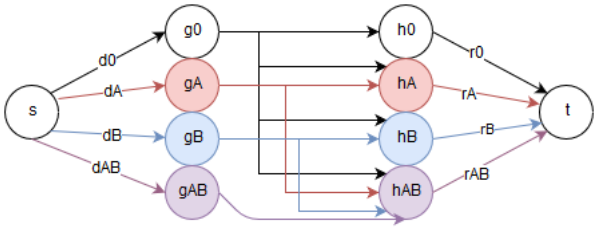


Figure 1: The flow network for Exercise 4

It is easy to see that all the compatibilities between blood types are respected, so it is a consistent model. First of all, some details about the flow network parameters will be provided in the following. We will use $x$ as placeholder for $0, A, B, AB$ when we refer to blood types with generality.

- The edges $e_{s-x} = (s, g_x)$ have capacities $c(e_{s-x}) = d_x$
- The edges $e_{x-t} = (h_x, t)$ have capacities $c(e_{x-t}) = r_x$
- For a fixed $\hat{x}$, the edges $e_{\hat{x}-x} = (g_{\hat{x}}, h_x)$ have capacities $c(e_{\hat{x}-x}) = d_{\hat{x}}$

Now we will prove a Proposition, from which follows the correctness of the algorithm.

**Proposition 1.** *Let $f$ a max flow returned by the execution of the Scaling-Max-Flow algorithm on the flow network above described. Then $v(f) = cap(V \setminus \{t\}, \{t\}) = \sum r_x$ $(1)$ iff* $\mathbf{d}$ *satisfy* $\mathbf{r}$*, i.e. it is possible to meet all demands* $(2)$.

*Proof.* $(1) \Rightarrow (2)$
Since $f$ is a max flow, for the Max-flow min-cut theorem it is equal to the capacity of *every* minimum-cut of the flow network. Given $f = \sum r_x$, we know for sure that *exists* a way to forward from $s$ to $t$ the required resources, due to the correct construction of the network. This means that given $(1)$ there must exist a flow network configuration, after the Scaling-Max-Flow algorithm execution, that makes all edges $(h_x, t)$ saturated, i.e. all demands satisfied.
$(2) \Rightarrow (1)$
If $(2)$ is true it means that all $(h_x, t)$ are saturated. Since $f$ is max flow and $t$ is a sink, follows that $cap(V \setminus \{t\}, t)$ is a min-cut and obviously $v(f) = \sum r_x$. □

The proposed algorithm simply check if the maximum flow $f$ found on the ad-hoc flow network satisfy Proposition 1, and so it provides a solution for the problem.

### 4.3 Running time analysis

The Scaling-Max-Flow algorithm can be implemented in $\mathcal{O}(m^2 \log(C))$ ([1], section 7.3). Since $m$ is constant, actually the running time is $\mathcal{O}(\log(C))$, where $C = \sum d_x$.

# 5 Exercise 5

## 5.1 Proposed algorithm

In order to well-define the problem and its solution, we define the following mathematical elements:

- $p : \{1, 2, \ldots, n\} \to \mathbb{N}$ the function that associates the job $i$ to its processing time;
- $s : \{1, 2, \ldots, n\} \to \{1, 2, \ldots, n\}$ the function that associates the job $i$ to its order in the schedule $s$; if we refer to the $i_{th}$ job of the schedule, we will use $s_i$ (the reader should notice that $s(s_j) = j$);
- $f : (i, s) \to \mathbb{N}$ the function that, given a job $i$ scheduled in at certain position in a schedule $s$, gives the job response time $f_i$ for the schedule $s$;
- $h : s \to \mathbb{N}$ the objective function we want to minimize.

**Algorithm 6** Find a schedule that minimize the average response time.

```
1: function FINDSCHEDULE(n, p)
2:     Inputs:
3:         n: number of jobs
4:         p: processing time function (defined above)
5:     Output:
6:         s: the correct schedule that minimize the average response time.
7:
8:     s ← Sorted list of jobs by processing time such that p(s_1) ≤ ··· ≤ p(s_n)
9:     return s
10: end function
```

The algorithm schedules the jobs by their processing times, from the smallest to the highest one. In the next section we will see why it is correct to minimize the average response time of the jobs.

## 5.2 Proof of correctness

First of all, notice that, given a schedule $s$ on $n$ jobs, minimize the average response time, i.e. $\frac{1}{n}\sum_{j=1}^{n} f(s_j, s)$ is equivalent to minimize the sum of response time, i.e. $\sum_{j=1}^{n} f(s_j, s)$, since the former expression is just the latter one multiplied by $\frac{1}{n}$, that is a rational number and, for one instance of the problem, it can be considered

a constant value that obviously doesn't affect the minimization problem.
Now, notice the mathematical formula for the response time of job $j$ is:

$$f(j,s) = \sum_{i=1}^{s(j)} p(s_i)$$

Putting it in the function that we want to minimize we obtain:

$$h(s) = \sum_{j=1}^{n} f(s_j, s) = \sum_{j=1}^{n} \sum_{i=1}^{j} p(s_i) =$$

$$\sum_{j=1}^{n} (p(s_1) + p(s_2) + \cdots + p(s_j)) = np(s_1) + (n-1)p(s_1) + \ldots 2p(s_{n-1}) + p(s_n)$$

Intuitively, we see that the lower the earlier $p(s_i)$ (with the highest coefficients), the lower the entire sum; and the best sorting template in order to follow this observation is the "smallest processing time first".
Now we state some proposition useful for the last proof. First of all, we define an *inversion*:

**Definition 1.** *Let $s$ a schedule for our problem. An* inversion *is a pair of jobs, $i$ and $j$, such that $p(j) > p(i)$ and $s(j) < s(i)$, i.e. the processing time of job $i$ is lower than the $j$'s one and $j$ is scheduled before $i$ in $s$.*

In other words, an inversion is a different sorting from the "smallest processing time first" sorting template. From this definition, follows that:

**Proposition 1.** *Let $s$ be an output of the proposed algorithm. Then $s$ has no inversions.*

*Proof.* By definition of our algorithm. □

A key observation is the following:

**Proposition 2.** *Let $s$ be a schedule with at least one inversion. Then there exists a consecutive inversion in $s$.*

*Proof.* By definition of inversion, $\exists a, b \in s$ such that $p(a) < p(b)$ and $s(b) < s(a)$. Consider the earliest job, $b$. If we go on higher position than $s(b)$, there must exist a certain job $j$ in which the next job in the schedule, say $i$, has a lower processing time. This means that $s(j) = s(i) - 1 < s(i)$ and $p(j) > p(i)$, which is actually the definition of a consecutive inversion in symbols. □

Now we prove that, in terms of response time, if we have a consecutive inversion, it is better swapping the inverted jobs:

**Proposition 3.** *Let $s'$ be a schedule with a consecutive inversion and let $s$ the same schedule after swapping the inverted jobs. Then the response time strictly decrease, i.e. $h(s') > h(s)$.*

*Proof.* Let the pair of job $(i, j)$ the inversion in $s'$, such that $s(j) < s(i)$ and $p(j) > p(i)$. By contradiction, assume that, after the swap, $h(s') \leq h(s)$.
For the sake of clarity:

$$s = (s_1, s_2, \ldots, i, j, \ldots, n)$$

$$s' = (s_1, s_2, \ldots, j, i, \ldots, n)$$

Now consider the objective function computed with both the schedules, i.e.:

$$h(s) = \sum_{j=1}^{n} f(s_j, s)$$

$$h(s') = \sum_{j=1}^{n} f(s'_j, s')$$

If we expand these sums:

$$h(s) = f(s_1, s) + \cdots + f(i, s) + f(j, s) + \cdots + f(s_n, s)$$

$$h(s') = f(s'_1, s') + \cdots + f(j, s') + f(i, s') + \cdots + f(s'_n, s')$$

It is easy to see that the addends before and after the schedule difference are equal, in symbols:

$$f(k, s) = f(k, s'), k < s(i)$$

and

$$f(l, s) = f(l, s'), l > s(j)$$

This is due to the particular expression of $f(i, s)$ and to the fact that jobs at the left and at the right of the inversion are in the same order: more in detail, it is due to $set(s_{1,i-1})$ = $set(s'_{1,i-1})$ and $set(s_{1,j>})$ = $set(s'_{1,j>})$ and to the commutative property of addition.
Now, by assumption of contradiction:

$$h(s') \leq h(s) \Rightarrow f(j, s') + f(i, s') > f(i, s) + f(j, s)$$

And by some calculations:

$$\sum_{k=1}^{s'(j)} p(s'_k) + \sum_{k=1}^{s'(i)} p(s'_k) \leq \sum_{k=1}^{s(i)} p(s_k) + \sum_{k=1}^{s(j)} p(s_k)$$

$$\Rightarrow$$

$$(p(s_1)+\cdots+p(j))+(p(s_1)+\cdots+p(j)+p(i)) \leq (p(s_1)+\cdots+p(i))+(p(s_1)+\cdots+p(i)+p(j))$$

$$\Rightarrow$$

$$p(j) + p(j) + p(i)) \leq p(i) + p(i) + p(j)$$

$$\Rightarrow$$

$$p(j) \leq p(i)$$

But $p(j) > p(i)$, so it is a contradiction, and then $h(s) < h(s')$ □

Finally, we prove formally that the proposed algorithm finds the optimal solution, i.e. finds a schedule that minimize the average response time.

**Proposition 4.** *Let $s$ be an output of the proposed algorithm. Then $s$ is the optimal solution for the problem*[4].

*Proof.* Suppose by contradiction that exists another solution, say $s'$, that is optimal and has a lower response time, i.e. minimize the objective function $h(s)$. Since $s' \neq s$ and $s$ has no inversion (Proposition 1), this means that $s'$ has at least one inversion, and for Proposition 2 there exists at least one consecutive inversion. By Proposition 3, if we swap one consecutive inversion in $s'$, we have that the objective function strictly decrease. But we assumed that $s'$ already had a minimum response time. This is a contradiction that prove the optimality of our solution. □

### 5.3 Running time analysis

The algorithm runs in $\mathcal{O}(n \log(n))$, since we only need to sort the jobs list. It can be implemented, for example, with *merge sort*.

[4]We implicitly say that the solution of "minimize the total response time" is the same of "minimize the average of response time", as explained at the start of the section.

## References

[1] Jon Kleinberg and Eva Tardos. *Algorithm Design*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2005. ISBN: 0321295358.