Algorithm Design
First Homework

Students: Simone Petruzzi 1811872 , Domenico Tersigni 1817502

November 24 2022

1 Exercise 1

1.1 a

We've implemented the algorithm using python version 3.10.8, and the code is the next page.

1.2 b

We can estimate the complexity of our algorithm through the dictionary that we use to store the states. In python dictionary is internally implemented as an hash table, and we know that in the average case, using hash table, the search operation costs O(1). All other operations have a total cost that is constant beacuse they are mostly comparison operations and appends. In our Big-O analysis we neglect constants and in order to estimate the complexity of our algorithm we have to estimate the number of recursive calls. A state is a quadruple with this shape $< a, b, c, n >$ where:

- $a = b = c = \frac{\sum_{k=0}^{L} l_k}{3}$
- n = number of kids

Each element of the quadruple is picked from the corresponding set, i.e. :

- $A = \left\{0, ..., \frac{\sum_{k=0}^{L} l_k}{3}\right\}$
- $B = \left\{0, ..., \frac{\sum_{k=0}^{L} l_k}{3}\right\}$
- $C = \left\{0, ..., \frac{\sum_{k=0}^{L} l_k}{3}\right\}$
- $N = \{0, ..., n\}$

In the worst case we have to go over all possible states, and the number of all possibile states is the number of all possible quadruples that we can have in the dictionary. That number is the cardinality of the cartesian product between the sets described above: $A \times B \times C \times N$.

The cardinality of the output set of a Cartesian product is equal to the product of the cardinalities of all the input sets, in our case we obtain (neglecting constants) $\frac{\sum_{k=0}^{L} l_k}{3} \cdot \frac{\sum_{k=0}^{L} l_k}{3} \cdot \frac{\sum_{k=0}^{L} l_k}{3} \cdot n$.

If we neglect $\frac{1}{3}$ which is a constant, the cost of our algorithm is $O(n \cdot (\sum_{k=0}^{L} l_k)^3)$

1.3 c

The algorithm starts by computing the sum of all the elements in the set, i.e, each kid has a level and we compute the sum of these levels. If this sum is not divisible by three we can't create three subsets with an equal sum. Conversely, we check if three subsets with the sum of elements equal to the sum of these levels divided by three exist or not, if so, the algorithm return an array that contains these three sets, if not it returns an empty list. We can do this by considering each item in the given array S one by one, starting from S[n], and for each item there are three possibilities:

- Put the current item in the first subset and recur for the remaining items with the remaining sum
- Put the current item in the second subset and recur for the remaining items with the remaining sum.
- Put the current item in the third subset and recur for the remaining items with the remaining sum.

The base case of the recursion would be when no items are left.

We return true when three subsets, each with $target_i - (sum\_subset_i\_elements) == 0$, are found, where $target_i = \frac{\sum_{k=0}^{L} l_k}{3}$ and $(sum\_subset_i\_elements) = \sum elements \in subset_i$

Our code tries to insert one element in the second subset only if inserting it in first subset doesn't result in a solution, and it tries to insert it in the third subset only if the previous cases don't result in any solution. Moreover, we add dictionary to the recursion function so that it can store the results of the search. For example, if we've already seen the state $< 2, 2, 53, 15 >$ and $subProblem(2, 2, 53, 15)$ was called the second time, (some parameters are missing for simplicity) the dictionary can directly return the result of $subProblem(2, 5, 53, 15)$ without searching down the branch of it again. Suppose we have two consecutive elements equal to 3 in the array, we could obtain the state $< 2, 5, 53, 15 >$ from a previous state $< 2, 5, 56, 17 >$ in which we choose to insert the first 3 in the second set and the second 3 in the third one. This state could be also obtained from the same previous state in which we choose to insert the first 3 in the third set and then the second 3 in the second set; in this second case we avoid to check if it is a safe state because we already know that the $subProblem(2, 2, 53, 15)$ is false. For this reason it is surely correct, becuse it splits the problem into simpler subproblems in a recursive way, and if a subproblem occurs many times, then dynamic programming will solve it only one time. Here each subProblem with the same solution correspond to a possible state of our dictionary, and if we go through all these states (in the worst case) we surely obtain the correct solution.

```
def subProblem(S, n, target1, target2, target3, first, second, third, sum_first_elements, sum_second_elements,
    sum_third_elements, states):
  if(target1 - sum_first_elements == 0) and (target2 - sum_second_elements ==0) and (target3 - sum_third_elements ==0):
    return True
  if n<0:
    return False
  state = (target1 - sum_first_elements, target2 - sum_second_elements, target3 - sum_third_elements, n)

  if state not in states:

    A = False
    first_free_space = target1 -((sum_first_elements) + S[n])
    if(sum_first_free_space >=0):
      sum_first_elements = sum_first_elements + S[n]
      A = subProblem(S, n-1, target1, target2, target3, first, second, third, sum_first_elements, sum_second_elements,
    sum_third_elements, states)
      if(A):
        first.append(S[n])
      else:
        sum_first_elements = sum_first_elements - S[n]

    B = False
    second_free_space = target2 -((sum_second_elements) + S[n])
    if(second_free_space >=0):
      sum_second_elements = second_elements + S[n]
      A = subProblem(S, n-1, target1, target2, target3, first, second, third, sum_first_elements, sum_second_elements,
    sum_third_elements, states)
      if(B):
        second.append(S[n])
      else:
        sum_second_elements = sum_second_elements - S[n]

    C = False
    third_free_space = target3-((sum_third_elements) + S[n])
    if(third_free_space >=0):
      sum_first_elements = sum_third_elements + S[n]
      A = subProblem(S, n-1, target1, target2, target3, first, second, third, sum_first_elements ,sum_second_elements,
    sum_third_elements, states)
      if(C):
        third.append(S[n])
      else:
        sum_third_elements = sum_third_elements - S[n]

    states[state] = A or B or C
    return states[state]

def 3partition(S):

  if len(S) < 3:
    return False

  states = {}
  total = sum(S)
  first = []
  second = []
  third = []
  sum_first_elements = 0
  sum_second_elements = 0
  sum_third_elements = 0

  if($total \% 3 ==0) and subProblem(S, len(S) - 1, total//3, total//3, total//3, first, second, third,
     sum_first_elements, sum_second_elements, sum_third_elements, states):
    return  [first,second,third]
  else
    return []


```

2 Exercise 2

In order to show that our problem is NP-Complete, we need to do the following: prove that our problem X is NP and choose a problem Y that is known to be NP-Complete and prove that $Y \leq_p X$
If any problem is in NP, then given a certificate, which is a solution to the problem and an instance of the problem, we will be able to decide whether solution is correct or no in polynomial time. For our problem a certificate is a set of advertisers of size k and we can identify the certificate using algorithm 1, that checks it in polynomial time $O(n^2)$, thus **our problem is NP**.

Algorithm 1

$S \leftarrow$ *set of advertisers of size k*
$F_i \leftarrow$ *set of people associates with* $i, \forall i \in S$
$N \leftarrow$ *set of people in the network*
for all $i \in S$ **do**
  for all $j \in F_i$ **do** $N \backslash$j
  end for
end for
if N *isEmpty* **then**
  $a \leftarrow YES$
else
  $a \leftarrow NO$
end if
return a

We choose as Y the Set Cover problem, that is known to be NP-Complete. Therefore, for any given instance I_y for Set Cover, an instance I_x must be created for our problem, in polynomial time, such that if I_y is a YES instance of Set Cover, then I_x is a YES instance of our problem, and if I_x is a YES instance of our problem, then I_y is a YES instance of Set Cover. The Set Cover problem is: Given a set **U** of **n** elements, a collection $S_i....S_m$ of subsets of U, and a number **k**, does there exists a collection of at most k of these sets, whose union is equal to all of U?
The instance can be created in this way:

Algorithm 2

$U \leftarrow$ *set which we would cover*
$S_i \leftarrow$ **i-th** *subset of U*
$N = \emptyset$
$F_i \leftarrow$ *set of people associates with person* **i** *in the network*
for all $i \in U$ **do**
  $N \leftarrow N \cup \{i\}$
  if $\neg \exists S_i$ **then**
    $S_i \leftarrow S_i = \{i\}$
  end if
  if $\exists S_i \ \wedge i \notin S_i$ **then**
    $S_i \leftarrow S_i \cup \{i\}$
  end if
  for all $j \in S_i$ **do**
    $S_j \leftarrow S_j \cup \{i\}$
  end for
  $F_i \leftarrow S_i$
end for

Now, if an instance I_x of our problem has a valid solution, through it, it's possible to construct a solution for the instance I_y of Set Cover and if I_x doesn't have a valid solution, not even I_y has one. Given a valid solution for I_x, the construction of a valid solution for I_y can be made in this way: $\forall i \in S$, where S is the set of advertisers of size k, which is the solution of our problem, we select subset S_i. This set of k advertisers is able to advertise a product to every person in N, meanwhile the union of these k subsets is equal to all of U.

Proof of correctness: U can be covered with at most k subsets, if and only if there is a set of advertisers of size at most k that advertise the product to all N. This can be proved in this way: If $S_i, ..., S_l$ are $l \leq k$ subsets that cover U, then $F_i, ...F_l$ contains all people of the network N and if we choose the set of advertisers $\{1, 2, ..., l\}$ we have a solution. Conversely if $\{1, 2, ..., l\}$ is a set of advertisers of size $l \leq k$ that is a solution for our problem, then $F_i, ...F_l$ cover all the network and then $S_i, ..., S_l$ cover all U.

Example of a satisfiable and unsatisfiable instances:

Instance I_y *of Set Cover* : $U = \{1,2,3,4,5,6,7,8,9\}$ $S_1 = \{1,2\}$ $S_2 = \{4,7\}$ $S_3 = \{5,8\}$ $S_4 = \{9,10\}$ $S_5 = \{2,3,4,5\}$ $S_6 = \{6,7,8,10\}$ $S_7 = \{1,3,6,9\}$
Instance I_x *of our problem* : $N = \{1,2,3,4,5,6,7,8,9\}$ $F_1 = \{1,2,7\}$ $F_2 = \{1,2,4,5,7\}$ $F_3 = \{3,5,7,8\}$ $F_4 = \{2,4,5,9,10\}$ $F_5 = \{2,3,4,5\}$ $F_6 = \{6,7,8,10\}$ $F_7 = \{1,2,3,6,7,9\}$ $F_8 = \{3,6,8\}$ $F_9 = \{4,7,9\}$ $F_{10} = \{4,6,10\}$
If we set $\mathbf{k = 3}$
Solution for I_x : *We choose as advertisers* $5, 6, 7$

$Solution\ for\ I_y : We\ choose\ as\ subsets\ S_5, S_6, S_7$
If we set $\mathbf{k = 2}$
$Solution\ for\ I_x : There\ isn't$
$Solution\ for\ I_y : There\ isn't$

3 Exercise 3

3.1 a

For the dynamic strategy we compute thresholds using the recursive formula presented in the point c (see below in section c). First we analyze the case with three prizes (n=3) assuming that $E_3(P) = \frac{1}{2}$ and $\tau_3 = 0$

$$E_2(P) = (1-\tau_2)(\frac{\tau_2+1}{2}) + \tau_2(1-\tau_3)(\frac{\tau_3+1}{2}) = \frac{1-\tau_2^2+\tau_2}{2} \tag{1}$$

$$\frac{d}{d\tau_2}[E_2(P)] = \frac{1}{2}(-2\tau_2+1) \tag{2}$$

now if we set the equation equal to zero we obtain value $\tau_2 = \frac{1}{2}$.
Let's proceed calculating τ_1:

$$E_1(P) = (1-\tau_1)(\frac{\tau_1+1}{2}) + \tau_1(1-\tau_2)(\frac{\tau_2+1}{2}) + \tau_1\tau_2(1-\tau_3)(\frac{\tau_3+1}{2}) = \frac{4-4\tau_1^2+5\tau_1}{8} \tag{3}$$

as we did for the previous case we obtain $\tau_1 = \frac{5}{8}$.
The case for n=2 is resolved with the same methodology applied above obtaining $\tau_1 = \frac{1}{2}$ and $\tau_2 = 0$

3.2 b

Let's discuss the static threshold for $n = 1, 2$:
let τ denote the static threshold for $n = 1$, here the solution is trivial because having a single extraction we are forced to take whatever it comes, so $\tau = 0$. The expected value here is $\frac{1}{2}$.
Now let's discuss the case for n=2, therefore we have X_1 and $X_2 \sim$ i.i.d of Uniform (0,1). We define the reward Y as:

$$\begin{cases} X_1 & if\ X_1 > \tau \\ X_2 & if\ X_1 \leq \tau \wedge X_2 > \tau \\ 0 & otherwise \end{cases}$$

$$Y = X_1 I(x_1 > \tau) + X_2 I(X_1 \leq \tau \cap X_2 > \tau) \tag{4}$$

$$\mathbb{E}(Y) = \int_\tau^1 x_1 dx_1 + \int_0^\tau \int_\tau^1 x_2 dx_2 dx_1 = \frac{1+\tau}{2}(1-\tau^2) \tag{5}$$

Now for maximizing the value we derive (2) with respect to τ, then setting equal to 0 we obtain $\tau = \frac{1}{3}$

3.3 c

Notation:

- we define $E_n(P) = 1/2$ as the expectation value of our strategy after time n-1 and before time n , infact if we've arrived at time n, it means that there is only one prize left and beacuse of all prizes are distributed independently from the uniform distribution over [0, 1], its expected value is $\frac{1}{2}$ and it makes sense to use $\tau_n = 0$, because there are no more chances.
- $X_t \sim i.i.d\ Uniform(0,1)$

If we want compute recursively the expectation of the best dynamic threshold as a function of n, we have to define:

- $E_{n-1}(P) = P(X_{n-1} > \tau_{n-1})\,\mathbb{E}(X_{n-1} \mid X_{n-1} > \tau_{n-1}) + P(X_{n-1} \leq \tau_{n-1})P(X_n > \tau_n)\,\mathbb{E}(X_n \mid X_n > \tau_n)$

If we compute the probabilities and the conditional expected values in the expression above we obtain:

- $E_{n-1}(P) = (1-\tau_{n-1})(\frac{\tau_{n-1}+1}{2}) + (\tau_{n-1})(1-\tau_n)(\frac{\tau_n+1}{2})$

In order to find the best dynamic threshold for n-1, i.e. τ_{n-1}, we have to maximize the expression above with respect to τ_{n-1}, which is the only variable we have, because as we said before we assume $\tau_n = 0$.

$$\begin{aligned} \frac{d}{d\tau_{n-1}}[E_{n-1}(P)] = & \frac{d}{d\tau_{n-1}}[(1-\tau_{n-1})(\frac{\tau_{n-1}+1}{2}) + (\tau_{n-1})(1-\tau_n)(\frac{\tau_n+1}{2})] = \frac{d}{d\tau_{n-1}}[(1-\tau_{n-1})(\frac{\tau_{n-1}+1}{2}) + (\tau_{n-1})\frac{1}{2}] = \\ & \frac{1}{2}\frac{d}{d\tau_{n-1}}[(1-\tau_{n-1}^2) + (\tau_{n-1})] = \frac{1}{2}(-2\tau_{n-1}+1) \end{aligned} \tag{6}$$

To maximize, we have to set (6) equal to zero and if we do that we obtain $\tau_{n-1} = \frac{1}{2} = E_n(P)$
Recursively, if we want to find the best dynamic threshold for n-2, i.e. τ_{n-2}, we have $E_{n-2}(P)$, which will be a function of τ_{n-2}, τ_{n-1} and τ_n but we know τ_{n-1} and τ_n , then τ_{n-2} will be the only variable of that expression, and if we maximize with respect to τ_{n-2} we obtain the best dynamic threshold for n-2, i.e $\tau_{n-2} = E_{n-1}(P)$ where $E_{n-1}(P)$ is computed using $\tau_{n-1} = \frac{1}{2} = E_n(P)$. And so on.

4 Exercise 4

4.1 a

We are on a line where i^{th} house is placed at a given kilometer L_i ($L_1 < L_2 < .... < L_k$), and the j^{th} hospital is placed at kilometer H_j. Hospital placement must satisfy: $\max_i[\min_j |L_i - H_j|] \leq d$

Algorithm 3

given $L_1 < L_2 < .... < L_k$
if k==0 **then**
  return 0
end if
$h = 0$
$H_1 = L_1 + d$
for $i = 2$ *to* k **do**
  if $|L_i - H_j| > d$ **then**
    $h = h + 1$
    $H_h = L_i + d$
  end if
end for
return $H_1, ..., H_h$

4.2 b

The loop in the code is the main instruction that defines the overall complexity. This loop has complexity depending from k and it's $\Theta(k)$ since we scan all the k houses to produce the solution with a greedy approach.

4.3 c

Let OPT be an optimal solution having m hospitals placed at distances $H_1 < ... < H_m$. Let $\{G_1, ..., G_p\}$ be the output of the proposed alogorithm. We want to ensure that for all our algorithm solutions, the number of hospitals given by the optimal solution is major or equal of the number of hospitals that are allocated by the algorithm (i.e m $\geq$ p) and also we must ensure that the hospitall location H_g is not further than G_g given by our algorithm (i.e $H_g \leq G_g$). Let's see by cases:
The case for **0 houses** is trivial since we have no hospitals to allocate and the empty set is returned.
When we have k=1 (**1 house**) our algorithm places $G_1 = L_1 + d$. If this is not optimal one of the two clauses that before we imposed must be unsatisfied. m=0 is not possible since we have one house that otherwise does not receive coverage by any hospital. Having $G_1 < H_1$ is also not possible since this would imply $(H_1 - L_1) > (G_1 - L_1) = d$ implying OPT not being an optimal solution because L_1 has no coverage by H_1.
Let's analyze the **generic case**, thus we must ensure that m $\geq$ p and $H_g \leq G_g$ for all g $\in \{1, .., p\}$. Suppose that g is the smallest index for which the two properties are false, this implies that for $g - 1$ the properties are true (we proved $g > 1$) but not for g, that implies $m \geq g - 1$ and $H_{g-1} \leq G_{g-1}$. Let $k(g-1)$ be the index in which the algorithm places G_{g-1} for covering house $L_{k(g-1)}$ this would imply $G_{g-1} = L_{k(g-1)} + d$. Let $k(g)$ denote the index in which the algorihtm places G_g in $L_{k(g)}$, this implies $G_g = L_{k(g)} + d$, and also $L_{k(g)} > L_{k(g-1)}$, moreover we must have that $|L_{k(g)} - G_{g-1}| > d$ because this is the condition that triggers the placing for G_g. So let's summarize the informations:

$$|L_{k(g)} - G_{g-1}| > d \tag{7}$$

$$L_{k(g-1)} = G_{g-1} - d \tag{8}$$

$$L_{k(g)} > L_{k(g-1)} \tag{9}$$

From here follows that:

$$L_{k(g)} > G_{g-1} + d \tag{10}$$

Moreover we know that for $g - 1$ we have $H_{g-1} \leq G_{g-1}$ and therefore this produces an inequality for $g - 1$ since from (10) we obtain that:

$$L_{k(g)} > H_j + d \; for \; j = 1, ..., g - 1 \tag{11}$$

Since OPT is a solution $L_{k(g)}$ must be in range with at least one hospital and there must be one more tower that is $m > g - 1$. Since we assumed that for g the two conditions are false, combining this with $m > g - 1$ we find that $H_g > G_g$, but we place $G_g = L_{k(g)} + d$ so it results that $H_j > G_g = L_{k(g)} + d$ for $j = s, ..., m$ so H_j is outside the desired range making OPT be not the optimal solution.

5 Excersise 5

5.1 a

By hypotesis we know that we have a Graph G=(V,E) where all edge capacities are integers of even value except for one single edge e^* that has odd value. Furthermore is given a max flow, say $\hat{f}$, of odd value. By the *integrality theorem* we know that if c(e) $\forall$ e $\in$ E are integers, then there exist a max flow f for which every flow value *f(e)* is an integer, thus $\hat{f}$ must be an integer.
If $\hat{f}$ is a maximum flow we know that there exist a *min cut* (A,B) such that c(A,B) = val($\hat{f}$):

$$c(A,B) = \sum_{e \ out \ of \ A} c_e = \ val(\hat{f}) \tag{12}$$

If the value of $\hat{f}$ is odd (and we know this by hypothesis), it means that in $\sum_{e \ out \ of \ A} c_e$ there is a value that is odd (since integer summation between n even numbers and a single odd value is overall odd). Since we have a single edge capacity that has odd value e^* then $f(e^*)$ should be major of 0 (i.e $f(e^*) > 0$) and flow must pass in the edge with odd value, if not the flow will always have an even value.

5.2 b

In addition to the fact that there will always exist flow in e^*, it always will be maximum. The reason for $fe^* = c_{e^*}$ is due to the fact that given the maximum flow $\hat{f}$ it means that there are no s-t paths in the residual graph:

$$val(\hat{f}) = \hat{f}^{out}(A) - \hat{f}^{in}(A) = \sum_{e \ out \ of \ A} \hat{f}(e) - \sum_{e \ into \ A} \hat{f}(e) = \sum_{e \ out \ of \ A} c_e - 0 = c(A,B) \tag{13}$$

since there is no flow that enters into the nodes of A, the capacity is maximum for every edge out of A, and the flow on e^* is full.