Andrea Panceri 1884749, Francesco Sudoso 1808353

Exercise 1

Data: K = Group of n kids that want to play football. Each kid k ∈ K has a level l_k ∈ {1, … , L}.

Goal: K kids want to be partitioned into 3 groups, with the same aggregate level.

a. We can solve this problem using dynamic programming through these steps:
Calculate the total sum of the kids' levels and check that it is divisible by 3. If it is not, the algorithm will return False, otherwise proceed with the auxiliary function that performs recursion on each element of the input list corresponding to the levels of each kid.
In order to optimize the performance of the algorithm by means of dynamic programming, it is necessary to define a table that allows us to insert the values of the calculations performed into memory. In this way, we avoid the computational cost of iterations that have already been previously performed, reducing the cost only to the reading of the table. For this reason, the input of the recursive function will be respectively:
- kids: list of kid levels
- n: length of list kids
- rem_sum_1: remaining target sum to be reached in partition 1
- rem_sum_2: remaining target sum to be reached in partition 2
- rem_sum_3: remaining target sum to be reached in partition 3
- table: table in which the performed calculations are stored
- partitioned_list: each element in the list indicates to which partition the element belongs
The recursive function consists of three cases and computes each input in each possible partition. Three cases are therefore required, one for each partition. In the first case, we insert the element in the first partition and perform recursion on the remaining elements until the target sum is reached. In the second case, we insert the element into the second partition and carry out recursion on the remaining elements up to the target sum and finally proceed in the same manner in the third case, with the third partition.
The recursive function will stop when there are no more elements remaining in the list or when each partition reaches zero as the target sum, thus returning True.
b. Complexity is calculated on a worst-case basis. In particular, each recursive call may recursively call three other recursive calls at each iteration, until the target sum reached by each partition is 0, which means until the algorithm terminates by returning True or False, and eventually the partitions. For this reason, the complexity of the algorithm will depend on both the number of kids n and the target sum to be reached. In addition, the computational cost will not be exponential but pseudo-polynomial because we use a lookup table that allows the executions already computed to be placed in memory. If we find a match of the current subproblem with one of the keys in the table, the cost will be O(1) to perform the read. The complexity of the algorithm will therefore be $O(n*(target\_sum)^3)$. However, another technique can be used to obtain the solution to the problem, that is, through an algorithm that performs backtracking. This solution consists of solving a problem by performing an exhaustive search on the entire set of possible options. Using a brute-force approach, one then tries all possible solutions and chooses the desired solution from all possible ones. If we had used this type of approach to solve the problem, the algorithm would have had an exponential cost as a function of n. Instead, we chose to use dynamic programming for solving the exercise because it solves the problem efficiently by breaking the problem down into simpler subproblems and solving each problem exactly once, as it stores the results of a subproblem in a table and reuses them when necessary to avoid solving the same problems over and over again. In fact, when the computation of a subproblem has already been performed, the algorithm will take the result from the table and reduce the cost of the operation to just reading the value.
However, it is also possible to express the complexity of the algorithm according to L (maximum level of a kid). In particular, we can distinguish two cases:
- If L > target_sum => Partitions with the same sum does not exist, consequently the complexity is O(1), as the algorithm terminates without performing any recursion.
- If L is very small, the algorithm will perform more iterations than if L is big. This can be understood from the fact that all other kid levels can be less than or equal to L and consequently at each recursive step a very small subtraction will be made from the target sum. For example, if L = 2, then each $l_i \in [1,2]$. This implies that at each recursive call, a subtraction will be made from the target_sum equal to l_i, and since l_i is very small, multiple recursive calls will be made to make the target_sum reach 0. Consequently, the smaller the value of L,the longer it will take the algorithm to terminate and find, if they exist, the three partitions with the same aggregate sum.

Andrea Panceri 1884749, Francesco Sudoso 1808353

c. The correctness of the algorithm lies in the fact that it performs all possible combinations of sums for each partition. Consequently, the algorithm will cover through its execution, all possible ways of getting each partition to the target sum, without skipping any cases. More specifically, for each new subproblem solved, the value of the computation will be saved in memory, while for each previously solved subproblem, the algorithm will retrieve the already computed result from memory and therefore skip the execution of that subproblem. In this way, each subproblem will be examined and if necessary computed. Performing all possible sums, the algorithm will return True if and only if exist three partitions with the same aggregate sum.

```
function Partition(kids):
  if len(kids) < 3: return False
  if sum(kids)%3 != 0: return False
  target_sum: sum(kids)/3
  table: {}
  partitioned_list: []
  partition_exist = getPartition(kids, len(kids), target_sum, target_sum, target_sum, table, partitioned_list)
  if partition_exist == True:
          for each (elem,index) ∋ partitioned_list:
                  if elem == 1:
                          partition_1.append(kids[index])
                  if elem == 2:
                          partition_2.append(kids[index])
                  if elem == 3:
                          partition_3.append(kids[index])
          return partition_exist, partition_1, partition_2, partition_3
  else
          return False

function getPartition(kids, n, rem_sum_1, rem_sum_2, rem_sum_3, table, partitioned_list):
  if rem_sum_1 == 0 and rem_sum_2 == 0 and rem_sum_3 == 0: return True
  if n < 0: return False

  key: (rem_sum_1, rem_sum_2, rem_sum_3, n)
  if key ∄ table:
          curr_kid = kids[n]
          if rem_sum_1 - curr_kid ≥ 0:
                  partitioned_list[n] = 1
                  partition_1_exist: getPartition(kids,n - 1,sum_1 - curr_kid,sum_2, sum_3,table,partitioned_list)
          else partition_1_exist: False
          if rem_sum_2 - curr_kid ≥ 0 and partition_1_exist is False:
                  partitioned_list[n] = 2
                  partition_2_exist: getPartition(kids,n - 1,sum_1,sum_2 - curr_kid,sum_3,table,partitioned_list)
          else partition_2_exist = False
          if rem_sum_3 - curr_kid ≥ 0 and partition_1_exist is False and partition_2_exist is False:
                  partitioned_list[n] = 3
                  partition_3_exist: getPartition(kids,n - 1,sum_1, sum_2,sum_3 - curr_kid,table,partitioned_list)
          else partition_3_exist = False
  table[key]: partition_1_exist || partition_2_exist || partition_3_exist
  return table[key]
```

Andrea Panceri 1884749, Francesco Sudoso 1808353

Exercise 2

Data: N = {1, ...,n} number of people. Each person $i \in N$ is associated with a subset of people $F_i \subseteq N$. For every $i \in N$ is true that if person $j \in F_i$ must be that $i \in F_j$.

Goal: Advertise a product to every person in N, choosing a set of advertisers among N, each will advertise it among his associates.

Prove: Finding, if exist, a set of size at most K of advertisers that can promote your product, is NP-Complete.

1. The problem is NP because given a possible solution of K advertisers, it is possible, in polynomial time, to check if the union of subset of people associated with each advertisers is equal to N, meaning that there exists a 'polynomial certificate' for our problem. Example: if we have a solution $K \subseteq N$ of advertisers we iterate for all $k \in K$ and check if the union of related F_k is equal to N. More formally we can give this simple algorithm:

input: $S \subseteq N$ a possible solution of chosen advertisers and initialize $N'=\varnothing$

For each Advertisers - $s \in S$:

$N' = N' \cup F_s$

return $N = N'$

This certificate is obviously polynomial: it is a simple cycle and an union operation in each iteration.

2. The problem is NP-Hard because we can reduce a problem that is NP-complete to our problem, in particular we reduce Dominant Set, that is been demonstrated to be NP-Complete using Vertex Cover, to this problem:

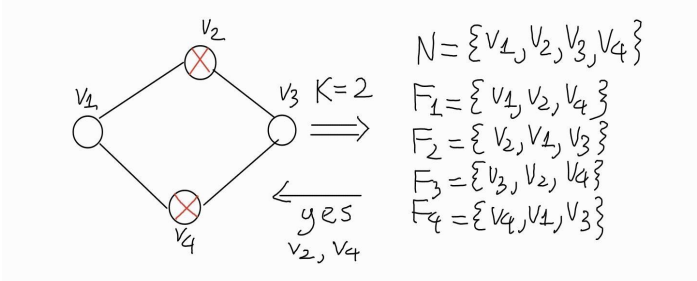
Transformation: Given any instance of Dominant Set that has as input a graph G = (V,E) where V is a set of vertices and E a set of edges, and K is the size of the dominating set that we want to find that is a subset of vertices D such that the vertices not belonging in D are adjacent to some vertex in D.

- we associate each $v_i \in V$ to a person $p \in N = \{v_1, \ldots, v_n\}$.
- $\forall\ v_i$ we have that $F_i = \{v_i \cup [\forall\ v_j \text{ s.t } \exists\ e_{ij} \in E]\}$.

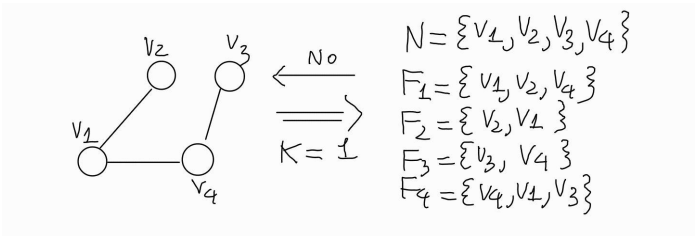
For the construction if $v_i \in F_i \Rightarrow v_j \in F_i$ and vice versa, because we used undirected edges.

We want to find if this instance of our problem has a solution with at most k advertisers.

Example: If the instance of our problem that we construct from Dominant Set has a solution s, exists a polytime function h s.t. h(s) is a solution for Dominant Set.



Given this instance of Dominant set with k=2, we obtain that our problem has a solution in particular we choose as advertisers: V_2,V_4, if we take the same vertices we have a Dominant Set of size = 2. If we have a yes instance in our problem then we have a yes instance for Dominant Set.



Given this instance of Dominant set with k=1, we obtain that our problem has no solution and we observe that also not exist a Dominant Set of size 1. If we have no instance in our problem then we have no instance for Dominant Set.

The constraint on the number of advertisers corresponding to the minimum size of the Dominant Set. In general for any instance, if our problem has a solution we simply take the vertices corresponding to the chosen advertisers and obtain the Dominant Set. This is true because we have chosen the vertices as people and the vertices connected to a vertex via edges as associates of each person, and we obtain a strict relationship between the two problems. We have reduced a problem NP-Complete to our problem and given that our problem is NP we can conclude that also our problem is NP-Complete.

We find also a more complex strategy for satisfying that our problem is NP-Hard, we can do a reduction from the vertex cover, that is a NP-Complete problem, to our problem: we can associate $v_i \in V$ and $e_{ij} \in E$ to a person $p \in N = E \cup V$. $\forall\ v_i$ we have that $F_i = \{v_i \cup [\forall\ v_j \text{ s.t } \exists\ e_{ij} \in E] \cup [\forall\ e_{ij} \in E]\}$ and $\forall\ e_{ij}$ we have that $F_{ij} = \{v_i, e_{ij}, v_j\}$. For the construction if $V_i \in F_{ij}$ is implied that $e_{ij} \in F_i$ and vice versa. We can do like before some examples for demonstrating the symmetry from the two problems.

Andrea Panceri 1884749, Francesco Sudoso 1808353

Exercise 3

Data: At each time t = 1, …, n is associated with a prize x_t and you can choose if to take it or not and continue. x_t are distributed independently from the uniform distribution over [0, 1]. For best performance we mean the best expected value that we can win.

b. First we calculate the best static threshold-strategies for n = 1, 2 with their expected performances: we define with τ the static threshold that will be the same for all extraction and $P(x_i) = 1 - \tau$ the probability of take the price, because it must be over the threshold we fixed. For n = 1 is very simple our strategy, we take any price because we have not another extraction and any price is good for us, this implies that in this case our $\tau = 0$. The performance is calculated with the probability of taking the current price times the expected value of the price that we win, that in our case is $\tau + (1 - \tau)/2$ (Fig.1) because we take only price over τ and the expected value that we take is at half of this interval. Performance = $P(x_1 > \tau) * EV = (1 - \tau) * (\tau + (1 - \tau)/2) = (1 - 0) * ½ = ½$, that is the best result with only one extraction. For n = 2 is more complex, we consider the probability of take the price in one of the two extraction that statistically is formulate like the probability of take the price in first extraction plus probability of take the price in second extraction minus the intersection of the two probability: $P_{12} = P(x_1 > \tau \cup x_2 > \tau) = (1 - \tau) + (1 - \tau) - (1 - \tau)^2 = 1 - \tau^2$. The expected value (EV) is the same because the τ (that we want to find) is the same in all extraction, performance = $P_{12} * EV = = (1 - \tau^2)*(1+\tau)/2$,we want to maximize this function of τ and for to do that we need to derivate and find the positive value that nullify the follow equation: max(performance) = $(½(1+\tau-\tau^2-\tau^3)' = 1-2\tau-3\tau^2 = 0 \Rightarrow \tau = ⅓ = 0.33$. This τ maximize the performance for n = 2 and we obtain the best performance equal to 0.592.

a. We calculate the best dynamic threshold-strategies for n = 2, 3 with their expected performances: Now we have the problem that each time t we have a different τ_t , we follow the intuition that we use for the static , find the equation of the performance, derivate it but now find the best τ in each time t. For n = 2 we define the probability of win in one of the two extraction slightly different from before, because now the τ_1 and τ_2 are different: $P_{12} = P(x_1 > \tau_1 \cup x_2 > \tau_2) = (1 - \tau_1) + \tau_1(1 - \tau_2) = (1 - \tau_1) + \tau_1 = 1$, we fix $\tau_2 = 0$ because like in the static with n=1 we must take the price in last extraction and the terms in the equation are the probability of win at the first extraction plus the probabilities of not win at first extraction but in the second extraction. For performance, like before we use the expected value that is different in each extraction: $Perf. = (1 - \tau_1)*EV_1 + \tau_1(1 - \tau_2)*EV_2 = (1-\tau_1)(1+\tau_1)/2 + \tau_1/2 \Rightarrow$ max(perf.)$= (((1-\tau_1^2)+\tau_1)/2))' = -2\tau_1+1 = 0 \Rightarrow \tau_1 = 0.5$ and the best performance for n=2 is 5/8=0.625. For n=3 we use the same logic, but we have now the probability of win in the third extraction: $P_{123} = P(x_1 > \tau_1 \cup x_2 > \tau_2 \cup x_3 > \tau_3) = (1 - \tau_1) + \tau_1(1 - \tau_2) + \tau_1\tau_2(1 - \tau_3) = (1 - \tau_1) + \tau_1(1 - \tau_2) + \tau_1\tau_2$,because now the last extraction is the third and we fix $\tau_3 = 0$ because we must take the price. The performance is also changed and it is: Perf.: $(1 - \tau_1)*EV_1 + \tau_1(1 - \tau_2)*EV_2 + \tau_1\tau_2(1 - \tau_3)*EV_3$, part of this equation is been calculated before, in particular $(1 - \tau_2)*EV_2 + \tau_2(1 - \tau_3)*EV_3$ is exactly the best performance for n=2 that is ⅝, now we have a single variable equation we repeat the same step: max(perf.)$= ((1-\tau_1^2)/2 + ⅝*\tau_1)' = -\tau_1+⅝ = 0 \Rightarrow \tau_1 = ⅝$ and the best performance for n=3 is 89/128=0.695.

c. We have illustrated the methodology for n=2,3 but we can find the best dynamic threshold as function of an arbitrary n using the innate recursion of the function that we find:

(1) $Perf_n = (1-\tau_n)*EV_n = ½ \Rightarrow$ because last extraction we fix $\tau_n = 0$ and consequently $EV_n = ½$.

(2) $Perf_{n-1} = (1-\tau_{n-1})*EV_{n-1} + \tau_{n-1}*$ (1)

(3) $Perf_{n-2} = (1-\tau_{n-2})*EV_{n-2} + \tau_{n-2}*$ (2)

…

(n) $Perf_1 = (1-\tau_1)*EV_1 + \tau_1*$ (n-1)

(1) is the base condition for the recursive formula as a function of n . For $n \to \infty$ we correctly achieve a performance equal to 1. We obtain this final recursive formula. For each recursive call we must find the τ_i that maximize this formula and find the max performance in each iteration, using the previous result of the recursive call

$$\text{PERFORMANCE}(m) = \begin{cases} 1/2 & , m = 1 \\ (1 - \tau_m) \times EV_m + \tau_m \times \text{PERFORMANCE}(m-1) & , m > 1 \end{cases}$$

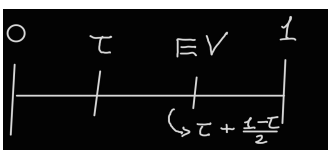


Fig.1

Andrea Panceri 1884749, Francesco Sudoso 1808353

Exercise 4

Data: K houses located on a line in location $L_1<L_2<...<L_k$.

Goal: Build hospitals on the line so that for ∀ i: $|L_i-H_j| <= d$ for some j and d is given.

a. We provide a greedy algorithm that finds the minimal amount m of hospitals that are needed, as well as valid values of $H_1,....,H_m$:

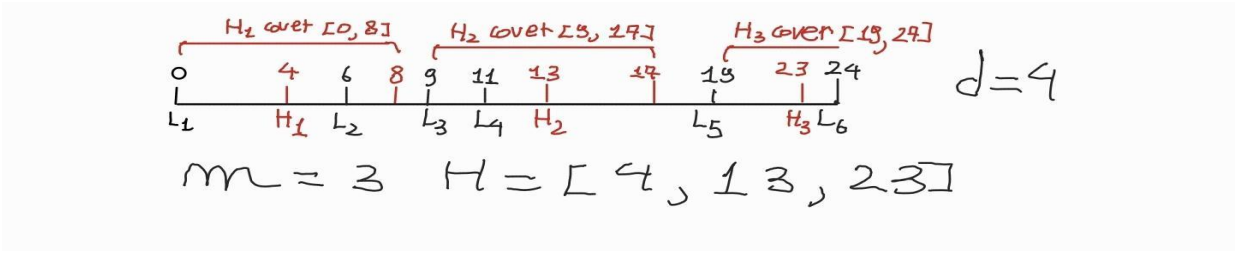
Input: L = $[L_1,...,L_k]$ locations of the houses and the max distance d.

```
def FindMinimumNumberOfHospitals(L, d):
  m = 0                    #minimum number of hospitals needed.
  H = []                   #hospitals
  current_house = 0
  while current_house != len(L):
          new_hospital = L[current_house] + d
          H.append(new_hospital)
          while current_house != len(L) and L[current_house] ≤ H[m] + d and L[current_house] ≥ H[m] - d
                  current_house += 1
          m += 1
  return (m,H)
```

The algorithm takes in input an array L with the locations of the k houses, and returns the minimum number of needed hospitals m and also H=[H_1, …., H_m] valid values for the locations of the hospitals.
We start from the location of the first not covered house and we set the hospital in this location plus d, we cover all houses in the range (H_i-d, H_i+d), iteratively we check how much houses we cover with this $hospital_i$ and in each iteration we take the next not covered house and do the same until we don't cover all houses.
This is a simple greedy algorithm but is very efficient for the given problem.

b. Now we analyze the complexity of the algorithm as a function of K, the number of the houses. We don't have a worst case or a best case, each time the algorithm does exactly k iterations, because though we have two cycles,in each of the two cycles we increment the same variable current_house until we cover all the houses (size of array L). For this motivation, either each house is distant from the next more than 2d and we must do the first cycle exactly k times and we never enter in the second cycle, or we have another instance of the problem in which the k iterations are split between the two cycles, is the same thing.
We achieve a linear complexity O(K), meaning that the complexity grows linear with the number of houses we have. While for the space we have a worst case that is when we need m=K hospitals ,one hospital for each house, because each house is distant from the next more than 2d. Considering this case we need O(K) extra space.

c. For prove that our algorithm find always the minimum number of hospitals m, we proceed for contradiction: absurdly exist an optimal solution m' < m (our solution), it implies that we have at least one less hospital and we must change the precedent configuration, but if we move an $hospital_i$ to L_i+d+ε or to L_i+d-ε at least one house that in precedent configuration was covered, now is not covered anymore. From what has been said it implies that we need more hospitals, violating the assumption that there exists an m' < m.
have demonstrated that the m that we find in our algorithm is the minimum number of hospitals needed.
Excluding what we have said in the demonstration, this implication is clearly given by the formulation of the algorithm that places each hospital to the maximum possible distance from the first not covered house in each iteration.

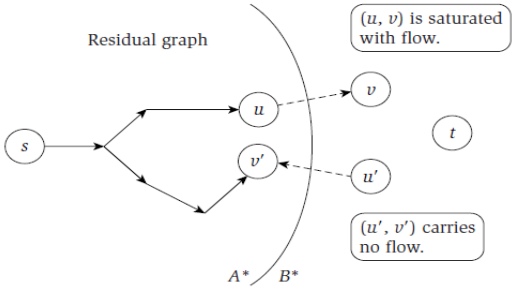


Andrea Panceri 1884749, Francesco Sudoso 1808353

Exercise 5

Data: *Given a graph G = (V,E) with integer capacities on the edges. All edges e have even capacities and e* has odd capacity. The maximum flow is odd.*

b. Prove - For prove the statement of question (b) we use a result that we see at lesson, the Max-flow min-cut theorem that says: Value of the max-flow = capacity of min-cut. From this theorem we add other thing, we also know that all edges out of cut A^* ($s \in A^*$) are completely saturated with flow, while all edges into cut A^* are completely unused. Now the last fundamental statement, the value of the max flow is equal to the capacity of the min cut that is equal to the summation of the capacities associated to the edges exiting the cut A^*.



$$
\begin{aligned}
v(f) &= f^{\text{out}}(A^*) - f^{\text{in}}(A^*) \\
&= \sum_{e \text{ out of } A^*} f(e) - \sum_{e \text{ into } A^*} f(e) \\
&= \sum_{e \text{ out of } A^*} c_e - 0 \\
&= c(A^*, B^*). \ \blacksquare
\end{aligned}
$$

We finally can arrive to a formal prove of question (b), given that our max flow is odd, this implies that the summation of the capacities associated to the edges exiting the cut A^* is odd, but if by contradiction the max flow passes only for edges with even capacity and consequently the capacity of all outgoing edges from A^* is even, then it is impossible that the max-flow is odd, because there must be a full flow in these even edges. Then it must be that the odd edge is one of the outgoing edges from A^* and for what had been said there is a full flow on e^*. Obviously because we prove the question (b) is also proved the question (a), in edge e^* there is always a positive flow but we add that the flow is exactly equal to his capacity.

This is the most formal prove we find, but we also think to another approach:

a. Another Prove - We do a prove using contradiction: let assume that there exists a maximum flow in that there isn't a flow in e. This means that the flow goes through only in edges with even capacities. Using the Ford Fulkerson algorithm we know that in each iteration we search an augmenting path s-t in the residual graph, in particular, we define in the iteration a bottleneck that fixes the maximum flow that can go through that path. Using the assumption we have that all the paths that we use, go through only on even edges, that implies the bottleneck is an edge that has an even capacity, but because we use all the capacity of the bottleneck that are even, we arrive to a contradiction, because the maximum flow will be even, instead by the initial assumption the maximum flow is odd and finally this implies that all the maximum flows go through the odd edge.*

b. Another Prove - Continuing the previous demonstration we arrive at the point in which we say that it is not possible that all augmenting paths have even capacities, but this implies that one of the bottlenecks is our odd edge and for definition of the algorithm in the bottleneck flow a full flow.

We came to this conclusion because we only have an odd edge and the max flow is odd. This implies, considering the demonstration of point (a), we have that not only does the maximum flow pass through that edge e^*, but it must also be one of the bottlenecks and therefore have zero residual capacity. Otherwise, if the odd edge were not the bottleneck, it would have residual capacity which could also be even and result in an even maximum flow, contradicting the assumption of having an odd maximum flow.

Example: