HW1 - Algorithm Design

Federico Mascoma

May 20, 2018

Exercise1: You are given an array A of n positive real numbers. The task is to find a subarray $A[i, j]$, i.e. starting at $A[i]$ and ending at $A[j]$ such that $\prod_{k=i}^{j} A[i]$ is maximized.
Your algorithm only has to output the value of $\max_{1 \leq i \leq j \leq N} \prod_{k=i}^{j} A[i]$.

Algorithm: Below is reported the code of the main algorithm, the other part is written in the file.
Is possible to run the algorithm with random input to check if the algorithm works, is all already setted in the main.

```
def esercizio1(A):    #returns a subArrai with indexes i,j
    prodotto = 0
    i = j = k = 0
    sol = [0,0,0]
    itemporaneo = 0
    jtemporaneo = 0
    while k < len(A):
        if prodotto <= 1 and A[k] >= prodotto:
            i=k
            j=i
            prodotto = A[k]
        else:
            if prodotto <= float(prodotto)*A[k]:
                j+=1
                prodotto = float(prodotto)*A[k]
            else:
                if float(prodotto)*A[k] >= 1:
                    if prodotto > sol[0]:
                        sol = [prodotto,i,j]
                    j+=1
                    prodotto = float(prodotto)*A[k]
                else:
                    if prodotto > sol[0]:
                        sol = [prodotto,i,j]
                    j+=1
                    prodotto = 0
        k+=1
    if prodotto <= sol[0]:
        prodotto,i,j = sol
    return A[i:j+1] , prodotto ,i,j
```

Claim: The algorithm finds $A[i,j]$ s.t $result = \prod_{k=i}^{j} A[i]$ is maximized

Notation:

- $product_k = \prod_{m=i}^{k} A[m]$ where A[i] is the first element taken into account in the product at step k and A[k] is the last element taken into account in the product at step k.
- $newProduct_k = A[k] * product_{k-1}$
- $sol = \max(sol, product)$ variable that stores the best solution step by step (contains also i,j)

Rules of the algorithm:

1. $product_{k=0}$ is initialized = A[0] in the base step, with $i, j = 0$
2. sol= (is initialized to 0)
3. start to scan A from A[0] until A[N-1] where N = length of A
4. Compute product between $product_{k-1}$ and the current element $A[k]$.
5. in every single step the algorithm checks the condition to decide "i" (the index of the first element taken into account in the variable "$product_k$") and compute the optimal solution "sol".
   Conditions:
   (a) if $product_k \leq 1$; $product_k = A[k]$ and $i = j = k$
      i. if $A[k] \leq product$: $sol = \max(sol, product)$ saving i, j into sol
   (b) if $\neg a$ &$product_k \geq product_{k-1}$ $sol = \max(sol, product_k)$
   (c) if $\neg b$ &$1 < product_k < product_{k-1}$: $sol = \max(sol, product_{k-1})$

Proof:

Step1: $sol = 0$

1. for $k = 0$ we get :
   - $product_k = A[0]$
   - $sol = \max(sol, product_k) = product_k = A[0]$

Step k=i: Assuming at step $k-1$ the algorithm finds sol = maximum solution for any sub-sequence in A[0,k-1], we must check the 3 conditions for $step_k$:

a) if $product_k \leq 1 \Rightarrow product_{k-1} * A[k] \leq 1$: this means that :

- if we take into account $product_k \leq 1$ (without update i) , $product_{k+1} = product_k * A[k+1] \leq A[k+1]$. (because $product_k \leq 1$)
  That's why the algorithm will take into account directly $product_k = A[k]$(and updates indexes i=j=k)
- We need to compute $sol = \max(sol, product_k)$
  Note that sol must be always checked, (it can also doesn't change) but it is important because sol stored the optimal solution step by step

b) if we are in case b $\Rightarrow$ a is not true $\Rightarrow product_k > 1$:

- $\neg a$ & $product_k \geq product_{k-1} \Rightarrow$ I can take into account in the next step $product_k$ without update the index 'i' because $product_k \geq product_{k-1}$.
- We need to compute $sol = \max(sol, product_k)$

c) if we are in (c) $\Rightarrow \neg a$ & $\neg b = true \Rightarrow 1 < product_k < product_{k-1}$:

- $1 < product_k < product_{k-1} \Rightarrow$ is correct to take into account $product_k$ without update index 'i' because $product_k * A[k+1]$ could be $\geq A[k+1]$
- We need to compute $sol = \max(sol, product_k)$

We can conclude that the maximum at the $Step_k = sol$.
In the $Step_{k+1}$ we got all the variables we need to compute the new sol, (We'll compute the new sol for induction).

Exercise2: Assume we are given a power network G = (V, E). We assume that G is a tree, i.e. a connected graph with no cycles. We say a hub conductor is a node v, such that upon removing v, the induced subgraph G_2 consists of connected components of size at most $\frac{1}{2} * |V|$. The task is to find a hub conductor. If multiple hub conductors exist, the algorithm has to find only one. In addition to arguing correctness and running time, implement your algorithm in either Java, Python, or C++ and supply us with a link or a print out of your code.

Algorithm: Below is reported the code of the main algorithm, the other part is written in the file.
(In the file there are implemented the main s.t is possible to run the algorithm with random input to check if it is working correctly)
GraphNode hub = current possible hub that can change in every step of the algorithm.
hub cardinality is initialized to infinitive. hub is the node root of the subTree which has the smallest cardinality $\geq |V|$, where $|V|$ = cardinality of the whole tree.

```
1  private static int DFS(GraphNode tree,int N,GraphNode hub) {
2    System.out.println("nodo corrente = "+tree.getNode());
3    tree.setVisited(true);
4    LinkedList<Edge> E = tree.getEdges();
5    Iterator<Edge> i = E.iterator();
6    int card = 0;
7    while(i.hasNext()){
8      Edge edge = i.next();
9      GraphNode U = edge.getU();
10     GraphNode V = edge.getV();
11     GraphNode nodo = U;
12     if (nodo.equals(tree))
13       nodo = V;
14     if (!nodo.getVisited()){
15       card = DFS(nodo, N,hub);
16       tree.setcardinality(tree.getcardinality()+card);
17     }
18   }
19   int c = tree.getcardinality();
20   System.out.println("il nodo "+tree.getNode()+" ha cardinality =
      "+tree.getcardinality());
21   if(c>N && c<hub.getcardinality()){
22     hub.setcardinality(c);
23     hub.setNode(tree.getNode());
24   }
25   return tree.getcardinality();
26 }
```

Proof that the algorithm find an Hub: By Contradiction:

Notation:

- $|V|$ =number of nodes in the Graph
- T =Tree that represents $G(V, E)$
- $|v|$ =cardinality of the subtree that has v as root (number of nodes counting v)
- $ch(v)$ =number of direct children of the node $v \in T$
- G_2 =$(V \backslash \{v\}$,$E \bigcap (V \backslash \{v\}$ x $V \backslash \{v\})$
- C = connected component
- $|G_2|$ = number of connected components in G_2

Rules:

1. The algorithm chooses hub v iff $|v| \geq \frac{1}{2}|V| \geq |v| - ch(v)$ (algorithm rule)
2. G has no cycle
3. $C \subset G_2 \forall C$
4. $C_1 \bigcap C_2 \bigcap .... \bigcap C_N = \emptyset$
5. $C_1 \bigcup C_2 \bigcup .... \bigcup C_N = G_2$

Claim: v is the root of the smallest subtree in T that has $|v| \geq \frac{1}{2}|V| \Rightarrow$v is an hub in G

Claim Proof:

1. suppose By contradiction that v is the root of the smallest subtree in T s.t $|v| \geq \frac{1}{2}|V|$ but is not an hub in G
2. :[1] $\Rightarrow \exists C_i$ s.t $|C_i| > \frac{1}{2}|V|$

Case a: C_i is located above v. $\Rightarrow$ C and v have no common node

3. :[2] $\Rightarrow |C_i| > \frac{1}{2}|V| \leq |v| \Rightarrow |C_i| + |v| > |V|$

Contradiction: Since we take into account the condition $|v| \geq \frac{1}{2}|V|$, C_i above v cannot get $|C_i| > \frac{1}{2}|V|$ because the sum of the nodes $\in C_i, v > |V|$ = number of total nodes

Case b: C_i is located below v. $\Rightarrow$ v contains all the nodes $\in C_i$.

4. in b case means that $\exists$ a node $u \in C_i$ s.t u is one of the children of v $\Rightarrow |u| < |v|$
5. Since u is the root of the subTree that represents $C_i \Rightarrow |u| = |C_i| \geq \frac{1}{2}|V|$

Contradiction: v is not the smallest subtree in T that has $|v| \geq \frac{1}{2}|V|$ because the subTree u has $|u| < |v|$ and $|u| \geq \frac{1}{2}|V|$

Lemma: if a node v has subTree with cardinality $= |v| \geq \frac{1}{2}|V| \geq |v| - ch(v) \Rightarrow$ v is an hub in G

To prove the correctness of the Lemma just prove that $|v| > \frac{1}{2}|V| \geq |v| - ch(v) \Rightarrow$ v is an hub in G
Thus $|C_i| \leq \frac{1}{2}|V|$ $\forall i \in (1..|G_2|)$ ($|G_2|$ = number of connected component)

Lemma Proof: By Contradiction
v is an hub ($|v| \geq \frac{1}{2}|V| \geq |v| - ch(v)$)but $\exists$ $C_i \subset G_2$ s.t $|C_i| > \frac{1}{2}|V|$

1. We can write that: by removing v from T you create such C_i located above or below v
   Particularly, knowing the Tree structure, can exists at most one C above v.

   We have to considerate 2 cases:

   (a) C_i is located above v
   (b) C_i is located below v

Case a (same to the claim proof): C_i is located above v. $\Rightarrow$ C and v have no node in common.

2. From Rule [1] : $|C_i| > \frac{1}{2}|V| \leq |v| \Rightarrow |C_i| + |v| > |V|$
3. To note that $|v| = 1 + \sum_i |C_i|$ whereC_i is located below v

CONTRADICTION: $|C_i| + |v| > |V|$ is impossible because C have to be a separate component from the components below v.
Therefore the sum of the nodes cannot be greater than cardinality of V

Case (b): C_i is located below v.

4. This means that a node of C_i is directed child of v $\Rightarrow$ v contains at least all the nodes in C +1(itself) $\Rightarrow |v| > |C_i|$
5. From [3] and Rule [1] : $|v| > |C_i| > \frac{1}{2}|V| \geq |v| - ch(v)$
6. From [4] : $ch(v) \leq |v| - |C_i| - 1$
7. Therefore Assuming $ch(v) = |v| - |C_i| - n$ $(n \geq 1)$: Where $(|v| - |C_i| - 1)$ is the max number of children that v can get knowing the size of $|C_i|$
   [5] $\Rightarrow |v| > |C_i| > \frac{1}{2}|V| \geq |v| - (|v| - |C_i| - n) \Rightarrow |v| > |C_i| > \frac{1}{2}|V| \geq |C_i| + n$

   CONTRADICTION: The value $|C_i|$ cannot be $> \frac{1}{2}|V|$ as long as $|v| \geq \frac{1}{2}|V| \geq |v| - ch(v)$ and C_i is below v ($|C_i| < |v|$).
   In fact, the formula above is inconsistent for each value of n

Lemma2: There exists always at least one hub conductor.

Proof: reviewing the condition to find the hub conductor, we can write that exists no hub conductor in a graph G (which we can see as Tree) iff there are no nodes that have cardinality $\geq \frac{1}{2}|V|$ that is impossible because at least the root of the Tree has cardinality $= |V| \geq \frac{1}{2}|V|$

Exercise3: Let G = (V, E) be a graph. Show that the follwing two problems are NP-hard.

1. G has a spanning tree where every node has at most k neighbors and k is part of the input.
2. G has a spanning tree where every node has at most 5 neighbors.

Hint: First consider a suitable hard problem for some appropriate choice of k. Then try to generalize your reduction for k = 5.

G has a spanning tree where every node has at most k neighbors and k is part of the input :
To prove the problem is NP-Hard, we choose the problem P1 with k = 2 and we have to find an NP-Complete problem P2 s.t it can be reduced to P1 in polynomial time.
We choose P2 = "Hamiltonian Path problem" (well known NP Complete problem).

$$P2 \leq_p P1$$

Proof P2 $\leq_p$ P1: We can reduce P2 to P1 trivially, just to add to the graph G' (where $\exists$ an HP) k-2 nodes to every single node and one more to the starting node s and ending node t (where s = source and t = sink in the HP).

In other words in P1 we get G = G' U {nodes added with the corresponding edges} Now we can write the follow Claim

1:

Claim: In the problem P1 , $\exists$ a spanning tree T $\subseteq$ G, where every node has at most 2 neighbors $\Leftrightarrow$ In P2 $\exists$ an Hamiltonian Path HP $\subseteq$ G' that visits all the nodes exactly once

Proof ($\Rightarrow$): In G $\exists$ T for problem P1 where every node has at most 2 neighbors.
assume that $|T| > 2$.
every node v $\in$ T has at most 2 neighbors $\Rightarrow$T is formed by only 2 "leafs" (which have only one neighbor each) and $|T| - 2$ nodes in the middle (with at least 2 neighbors to link the previous node to the next one, and at most 2 nodes by the condition k=2).
For k = 2 the nodes added by the reduction are k-2 = 0 for every single node, hence G' = G.
Trivially follows that the existence of an HP in P2 is given by the fact that an

example of an HP is T because every node in T has at most 2 neighbors, just like in an HamiltonianPath.
If $|T| = n \leq 2$ is banal because T contains only n leafs (at most 2).

For k = 2 we don't need to use particular effort to reduce P2 to P1, just considerate hamiltonian Path = T, where s and t in HP are the leafs in T.

Proof ($\Leftarrow$): If $\exists$ HP $\subseteq$ G' in the problem P2 $\Rightarrow$ $\exists$ T $\subseteq$ G in P1 with at most 2 neighbors for each node.
Is the same thing to prove that if $\nexists$ T $\subseteq G$ in P1 with at most 2 neighbors for each node $\Rightarrow$ $\nexists$ HP $\subseteq G'$ in P2
$\nexists$ T$\subseteq$ G in P1 with at most 2 neighbors for each node $\Rightarrow$ in G $\exists$ at least one node $v \in G$ with n $>$ 2 neighbors s.t at least 2 are leafs, hence $v \in T$ will have m $>$ 2 neighbors.
now, we need to apply the De-Reduction from P1 to P2, obtained just cutting the nodes added from the Reduction. But the number of nodes added from the reduction is k-2 = 0 $\Rightarrow$ G' = G .

Trivially follows that also in G' $\exists$ the node v with n $>$ 2 neighbors, (just like T in G), hence v has at least 2 leafs as neighbors $\Rightarrow$ cannot exists HP because v will get at least m $>$ 2 neighbors removing every cycle in G'

2:

Claim: In the problem P1 , G has a spanning tree T where every node has at most 5 neighbors $\Leftrightarrow$ In P2 there exists an Hamiltonian Path HP that visits all the nodes $\in$ G' exactly once

Proof ($\Leftarrow$): $\exists$ HP for P2 in G' $\Rightarrow$ in the problem P1 $\exists$ a spanning tree $T \subseteq G$ with at most 5 neighbors for each node.
G is the graph obtained by the reduction adding 5-2 = 3 nodes for each node in G'.
Since we have added the node to reduce G' to G (and not removed), In the new graph G exists surely a path P that takes the same nodes that takes HP in G'.
we have added 3 nodes for each node in G' $\Rightarrow$ there are exatly 3 leaf nodes for each node $\in$ P.
Now just link these leaf nodes to the nodes in P, 3 for each node in P. linking 3 leaf nodes to each node in the path P, will be created a spanning Tree T which has at most 5 neighbors,(at most 2 $\in$ P plus 3 added after).

Proof ($\Leftarrow$)': is equivalent to prove that : $\not\exists$ T in G, with at most k=5 neighbors, s.t T is a solution for P1$\Rightarrow$ $\nexists$ HP in G', s.t HP is a solution for P2.
$\nexists$ T in G, with at most k=5 neighbors, solution for P1 $\Rightarrow$ $\exists$ at least one node v $\in$ T in the graph G s.t v has n$>$ k = 5 neighbors.
Therefore, if we apply the DeReduction removing k-2 = 3 leaf nodes from T, we

obtained that v $\in$ T' (spanning tree after the deReduction) has n-3 neighbors. since n $>$ 5$\Rightarrow$ v in T' has more than 5-3 = 2 neighbors. Concluding we cannot find an hamiltonian path if $\exists$ v $\in$ T' that has 3 neighbors or more

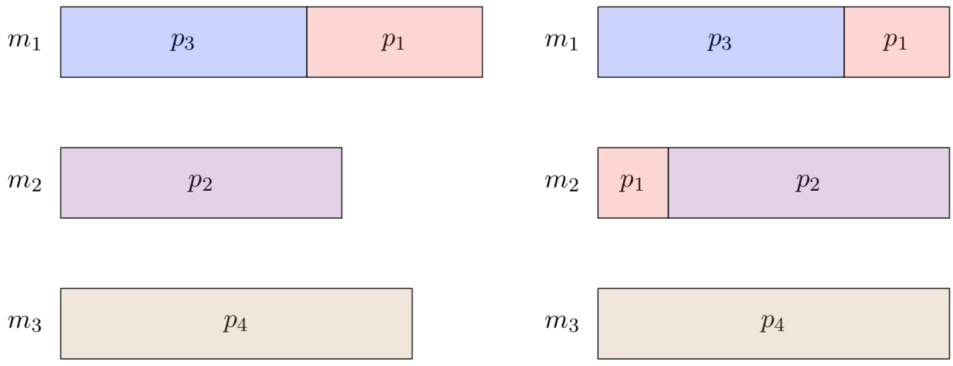
Proof ($\Rightarrow$): $\exists$ T in G, solution for P1, where every node has at most 5 neighbors $\Rightarrow$ in P2 $\exists$ HP that visits all the nodes of G' once.
since P1 is a Reduction of P2, applying the DeReduction by P2 removing 5-2 nodes (added in the Reduction) $\Rightarrow$ we get T' (a spanning Tree) in G' where each node has at most 5-(5-2) = 2 nodes as neighbors. Now, for the proof of the first Claim, T is exactly = a possible HP, and the nodes with only one neighbor(leafs node) are taken as s and t.

Exercise4: We are given n jobs with processing times p1, . . . pn and m machines. We assume that our machines support interruptions, e.g. a job j with pj = 5 may be processed on a machine m1 for 3 units of time, interrupted, and subsequently resumed processing for the remaining 2 units of time on machine m2. A job cannot be executed simultaneously on two different machines.

1. We want to schedule the jobs on the machines such that the completion time Cmax of the final job is minimized. For full marks the algorithm must run in linear time (O(n)). In particular the running time must be independent of the size of the jobs.

2. Show that the problem is NP hard even on 2 machines if interruptions are not allowed.

As an example, consider four jobs with processing times p1 = 5, p2 = 8, p3 = 7, and p4 = 10 to be scheduled on three machines. The schedule on the left is not optimal and has Cmax = Cp1 = 12. The schedule on the right is optimal with a completion time of Cmax = 10.



1 :
Pseudo-code:

```
Algorithm(MachinesSet, JobSet):
  Avarage <— Sum{timeProcesJobs / card(MachinesSet));
  MaxJob <— job with highest process time
  Cmax <— max(Avarage, MaxJob)
  for m in MachinesSet:
    while m time execution < Cmax:
      m.add(JobSet.removeFirst)
    if m time execution > Cmax:
      cut the last job added to get m time execution = Cmax.
      insert to JobSet the remaining part as first element
```

Algorithm: Below is reported the code of the main algorithm, the other part is written in the file.

```
def schedule(jobs,M):
    m = len(M)#m = numero di macchine
    n = len(jobs)
    print "size M = "+str(m)+"\t size jobs = "+str(n)
    Costo = 0
    i=0
    somma = 0
    jobMaggiore = 0
    for job in jobs:
        if jobMaggiore < job[1]:
            jobMaggiore = job[1]
        somma+=job[1]
        Costo+=1
    media = somma/m
    if somma%m > 0:
        media+=1
    print "\nJOBS : "+str(jobs)
    stampaM(M)
    #qui si dividono i due casi, se jobMaggiore > media e se
    JobMaggiore <= media
    if jobMaggiore > media:
        media = jobMaggiore
    print "somma totale : "+str(somma)+"\t media or JobMaggiore : "
    +str(media)+"\n"
    j=0
    for m in M:
        Costo+=1
        while m[0] < media and j < n :
            Costo+=1
            m[1].append(jobs[j])
            m[0]+=jobs[j][1]
            j+=1
        if m[0] > media:
            j-=1
            resto = m[0]-media
            m[0]-=resto
            job2 = m[1].pop()
            job2 = job2[0],job2[1]-resto
            m[1].append(job2)
            jobs[j][1]=resto
    print "COSTO : "+str(Costo)+"\tn :"+str(n)
    return M
```

cost : O(n)

Solution 1:

Notation:

- S = schedule that have to organize jobs with the machines
- machines = set of machines
- jobs = set of jobs
- N = number of jobs
- M = number of machines
- j_i = job number i
- p_i = processing time of job j_i
- m_i = machine number i
- Cp_i = time that m_i has to work
- $Cmax = \max_{\forall i \in [1,M]} Cp_i$
- avarage = $\left\lceil \frac{\sum_{i=1}^{N} p_i}{M} \right\rceil$
- $pmax = \max_{\forall i \in jobs} p_i$

Rules of the algorithm:

1. The algorithm precomputes $Cmax = \max(avarage, pmax)$ (Cmax must be an integer number ≥ 0)
2. S is valid iff organize all jobs on the machines available ($\sum_{i=1}^{M} Cp_i := \sum_{i=1}^{N} p_i$) and $\nexists$ any job that is executed simultaneously by 2 or more machines.
3. The algorithm appends a job j_k in machine m_i iff the previous machine is full ($Cp_{i-1} = Cmax$) or m_i is the first machine (i=1)
4. The algorithm has initially all the machines empty and appends all jobs starting from m_1 to m_M

Claim: The outcome schedule S by the algorithm is valid and gets Cmax which is optimal.
This means that doesn't exist another valid schedule S2 with the same set of jobs and machines s.t has $Cmax_{S2} < Cmax_S$.

Proof: Assume By Contradiction that $\exists$ a valid schedule S2 that gets $Cmax_{S2} < Cmax_S$

By Notation and assumption we can write that:

$$(*) : Cmax_{S2} = \max_{\forall i \in [1,M]} (Cp_i)_{S2} < Cmax_S = \max(avarage, pmax)_S$$

$$Cmax_{S2} = \max_{\forall i \in [1,M]} (Cp_i)_{S2} \Rightarrow Cmax_{S2} \geq \left\lceil \frac{\sum_{i=1}^{M} Cp_i}{M} \right\rceil_{S2}$$

from above we can follow : $\left\lceil \frac{\sum_{i=1}^{M} Cp_i}{M} \right\rceil_{S2} \leq Cmax_{S2} < Cmax_S = \max(avarage, pmax)_S$

Trivially follows that:

$$(**) : \left\lceil \frac{\sum_{i=1}^{M} Cp_i}{M} \right\rceil_{S2} < \max(avarage, pmax)_S$$

We have to separate 2 Case:

1. $\max(avarage, pmax)_S = avarage$
2. $\max(avarage, pmax)_S = pmax$

Case (1): $\max(avarage, pmax)_S = avarage$

From (**) we write: $\left\lceil \frac{\sum_{i=1}^{M} Cp_i}{M} \right\rceil_{S2} < \left\lceil \frac{\sum_{i=1}^{N} p_i}{M} \right\rceil_{S} = avarage_S$

(Rule 2) Since the schedule must process all jobs with the machines , must be satisfied the condition : $\sum_{i=1}^{M} Cp_i := \sum_{i=1}^{N} p_i$.

Therefore, just change $\sum_{i=1}^{M} Cp_i$ with $\sum_{i=1}^{N} p_i$ we obtain : $\left\lceil \frac{\sum_{i=1}^{N} p_i}{M} \right\rceil_{S2} < \left\lceil \frac{\sum_{i=1}^{N} p_i}{M} \right\rceil_{S}$

CONTRADICTION: To respect the condition, the schedules S2 and S must process the same set of jobs with the same set of machines, so $Cmax_{S2}$ cannot be less than $avarage_S = Cmax_S$.

Case (2): $\max(avarage, pmax)_S = pmax$
From (*) follow : $\max_{\forall i \in [1,M]} (Cp_i)_{S2} < \max_{\forall i \in jobs} (p_i)_S = pmax_S$
Therefore $\exists$ at least a job j_i s.t $p_i = pmax_S > \max_{\forall i \in [1,M]} (Cp_i)_{S2} = Cmax_{S2}$

CONTRADICTION: S2 is not valid because j_i doesn't fit in a whole machine space, hence it will be processed simultaneously by at least 2 different machines.
Therefore in this case Cmax cannot be less than pmax

Proof S is valid: By Contradiction $\exists$ in S $j_i \in jobs$ s.t is executed simultaneously by m_i and m_j with $m_i \neq m_j$ and $i < j$

for the algorithm structure : if m_i and m_j are executing simultaneously $j_i \Rightarrow j_i$ doesn't fit in K = j-i machines $\Rightarrow p_i > K * Cmax$

Case (1): $\max(avarage, pmax) = avarage$

Trivially follows that: j_i has value $p_i > K * avarage = K * \left\lceil \frac{\sum_{i=1}^{N} p_i}{M} \right\rceil \Rightarrow p_i > avarage = \left\lceil \frac{\sum_{i=1}^{N} p_i}{M} \right\rceil$
just because we are in case 1 : $p_i > avarage \geq \max_{\forall i \in jobs} p_i \geq p_i \forall i \in jobs$

CONTRADICTION: if $p_i > avarage$ the algorithm doesn't choose case(1), but chooses case(2). Therefore $p_i \leq avarage$ and j_i will fit in a space $\leq avarage$. Therefore by the structure of the algorithm j_i cannot be executed by 2 or more machines simultaneously

Case (2): $\max(avarage, pmax) = pmax$
In this case by contradiction $\exists$ j_i which has value $p_i > pmax = \max_{\forall i \in jobs} p_i \geq p_i \forall i \in jobs$

CONTRADICTION: p_i can be at most equal to $pmax$ because in the formula $\max_{\forall i \in jobs} p_i$ is also evaluated $p_i \in jobs$.

Solution 2:

Claim: interruptions are not allowed ⇒ problem is NP hard even on 2 machines

P1 is NP-Hard if ∃ a problem NP-Complete P2 s.t P2 $\leq_p$ P1 (we can reduce in polynomial time P2 to P1)

For this type of problem, we can choose P2 = Partition Problem (well known NP-Complete Problem).

Both P1 instances (time of processes p_i) and the instances of P2 (s_i) are positive Integer numbers. Therefore we can transform directly the instances of P2 to the instances of P1 in polynomial time, just apply $f(x) = 1 * x$ choosing f(x)=p and x=s, when s is an elements $\in S$ and p is a time of a process:

Therefore we have to prove the follow Claim:

Claim: ∃ Cmax optimal for P1 ⇔ ∃ f(Cmax) optimal for P2

We can prove taking into account Cmax as lowerBound = $\frac{\sum_{i=1}^{N} p_i}{2}$, hence surely the optimal solution, and M = 2.

∃ Cmax = $\frac{\sum_{i=1}^{N} p_i}{2}$ ⇔ $\exists S_1, S_2$ s.t $|S_1| = |S_2|$ & $S_1 \cup S_2 = S$ & $S_1 \cap S_2 = \emptyset$

Proof (⇒): Cmax = $\frac{\sum_{i=1}^{N} p_i}{2}$ ⇒ $\exists S_1, S_2$ s.t $|S_1| = |S_2|$ & $S_1 \cup S_2 = S$ and $S_1 \cap S_2 = \emptyset$

1. machines didn't allow the interrupt ⇒ m_1 and m_2 have to execute only entire processes.
2. if Cmax = $\frac{\sum_{i=1}^{N} p_i}{2}$ ⇒ m_i (one between m_1,m_2) will execute some processes for Cmax time
3. :[2] ⇒ $m_j \neq m_i$ will be in execution for time = $\sum_{i=1}^{N} p_i - Cmax = \sum_{i=1}^{N} p_i - \frac{\sum_{i=1}^{N} p_i}{2} = \frac{2*\sum_{i=1}^{N} p_i - \sum_{i=1}^{N} p_i}{2} = \frac{\sum_{i=1}^{N} p_i}{2} = Cmax$
4. From [3] we can see that the time of execution of m_i and m_j are equals. (We have only 2 machines , so we can take into account m_i as m_1 and m_j as m_2).
5. Trivially follows that just take the processes time $p_1, , p_k$ ($k < N$), that are in m_1, as integer $s_1, , s_k \in S1$ and $p_{k+1}, , p_N$, that are in m_2, as integer

$s_{k-1}, , s_N \in S2$.

$Cp_1 = Cp_2 \Rightarrow |S_1| = |S_2|$ and $Cp_1 + Cp_2 = \sum_{i=1}^{N} p_i \Rightarrow |S_1 \cup S_2| = |S|$

Proof ($\Leftarrow$): S_1, S_2 s.t $|S_1| = |S_2|$ & $S_1 \cup S_2 = S \Rightarrow \exists$ Cmax $= \frac{\sum_{i=1}^{N} p_i}{2}$

By Contradiction : $\nexists$ Cmax $= \frac{\sum_{i=1}^{N} p_i}{2}$.

1. So we know that the two machines m_1, m_2 are in execution for Cp_1 and Cp_2
2. By definition : $Cp_1 + Cp_2 = \sum_{i=1}^{N} p_i$
3. We know that Cmax is the lower bound, therefore
   if $\nexists$ Cmax $= \frac{\sum_{i=1}^{N} p_i}{2}$ & $\sum_{i=1}^{N} p_i = Cp_1 + Cp_2 \Rightarrow Cp_1 > Cp_2$ or $Cp_1 < Cp_2$

Contradiction: taking into account the consideration of prior(point [5]) We know that $Cp_1 = |S_1|$ and $Cp_2 = |S_2|$. Therefore $|S_1| \neq |S_2|$

Exercise 5: A board game expert is using an algorithm to optimize his playing strategy. The game is based on a set of n challenges. In order to complete a challenge, the board game player has to acquire a set of resources. Challenge Ci, i $\in$ 1, .., n, requires resources $r_1, \ldots, rn_i$.

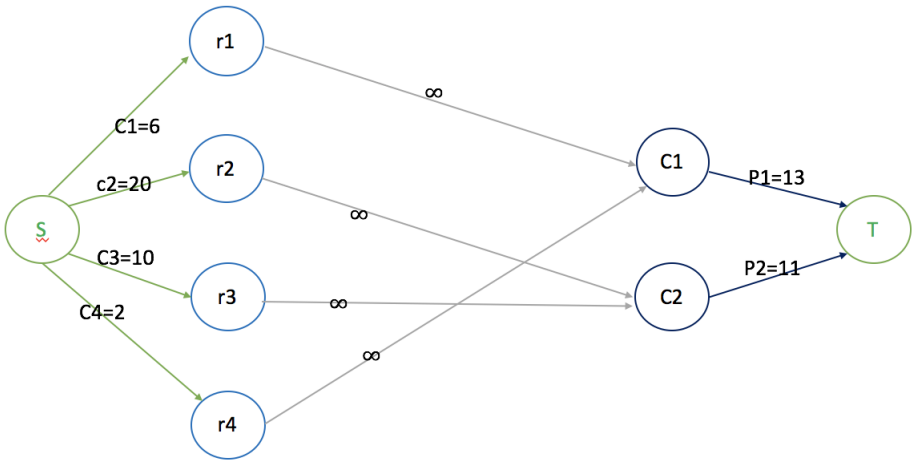
Challenge C_i gives p_i points when the game player has acquired all the resources needed to complete the challenge.
In order to obtain the resources, the board game expert needs to buy resource cards. The resource card r_j , j $\in$ 1, . . . , m, will cost c_j points. Once the game player holds a resource, it can be used for all challenges that require this resource, i.e. resource r_1 is bought only once, and not for every completed challenge. You may assume that resources and points are positive integers.
The net revenue of the game player is equal to the total revenue of the challenges he wins minus the total cost of the cards he buys.

Design an algorithm based on max-flow/min-cut which provides an optimal solution to this problem. Provide a formal proof of the optimality of the algorithm.

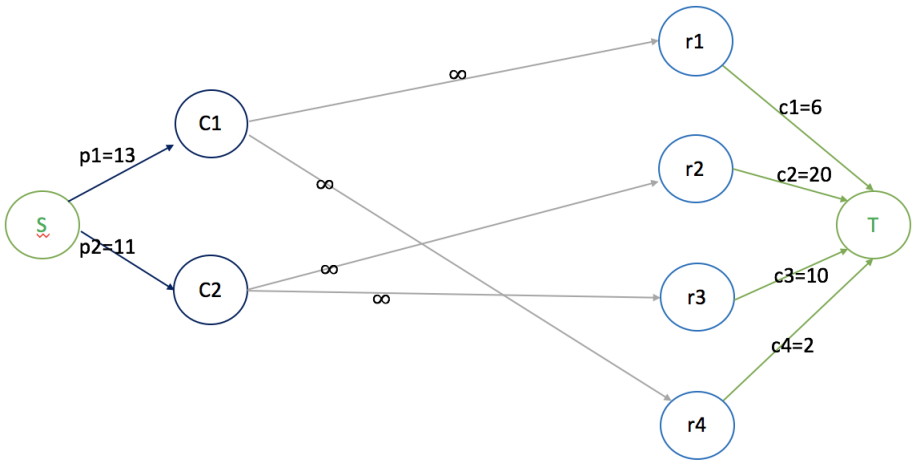
Hint: Design a network with:

- a source node s connected to a node representing the resource card r_j with an edge of cost c_j,
- a node representing challenge C_i connected to sink t with an edge of cost p_i,
- an edge of cost infinity connecting each node with resource r_j to any challenge C_i that requires r_j for completion.

Example Graph G(V,E) :



Example Graph G'(V,E) :



Notation

1. G'(V,E) = Graph obtained inverting the towards in the graph G and inverting S with T.
2. MC = MinCut of G'
3. S = Source
4. T = sink
5. C = set of Challenges
6. R = set of resources
7. V = R ∪ C
8. rC_i = resources (r_i, , r_j) that C_i need to get.
9. p_i = edge weight from S to C_i
10. c_i = edge weight from r_i to t
11. P = $\sum_{C_i \in G'} p_i$
12. c(MC):MinCut Capacity = $(P - \sum_{C_i \in MC} p_i) + \sum_{r_i \in MC} c_i = \sum_{C_i \notin MC} p_i + \sum_{r_i \in MC} c_i$
13. revenue = $\sum_{C_i \in MC} p_i - \sum_{r_i \in MC} c_i$
    we can write the revenue as above because we take the resources and nodes which are contained in MC

Claim: if MC is a MinCut of G' $\Leftrightarrow$ revenue is maximized.
By definition c(MC) must be as small as possible.

Rules

1. $rC_i \in$ MC iff $C_i \in$MC since the edges that link r_i to C_i has cost = infinitive (by structure of the graph)

Proof ($\Rightarrow$) :if MC is a MinCut of G' $\Rightarrow$ revenue is maximized.

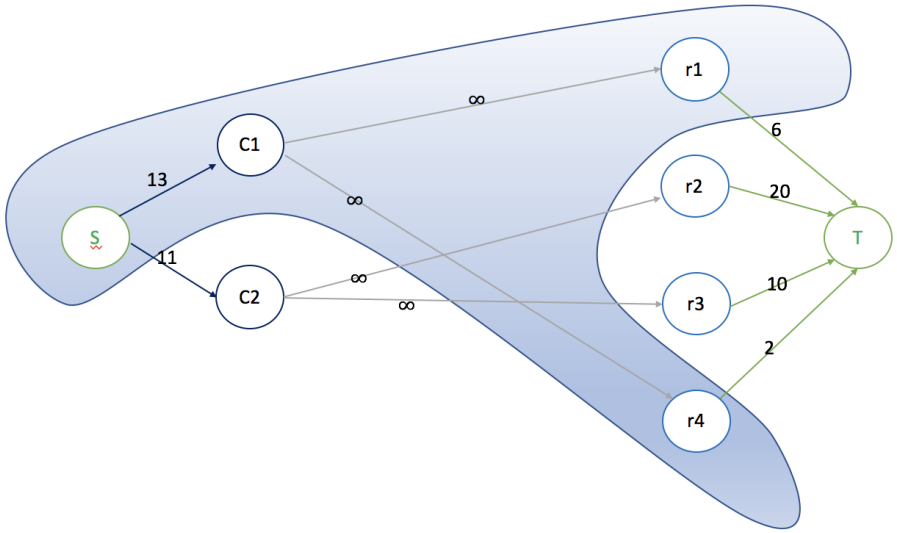
1. MC is a minCut $\Rightarrow c(MC)$ is as small as possible.
2. By the Notation : $c(MC) = P - revenue = (\sum_{C_i \in G'} p_i) - revenue$
3. :[2] $\Rightarrow revenue = \sum_{C_i \in G'} p_i - c(MC)$
   Since revenue is the value that must be maximized, we can get maxRevenue minimizing c(MC).
4. :[1,3] $\Rightarrow$ revenue is maximized.

Proof ($\Leftarrow$) :revenue is maximized $\Rightarrow$ MC is a MinCut of G'
Is the same to prove that : MC is not a MinCut of G' $\Rightarrow$ revenue is not maximized
Assume By Contradiction that MC is not a MinCut of G' but revenue is maximized

1. MC is not a MinCut of G' $\Rightarrow$ there exist a MC2 that is a MinCut of G' $\Rightarrow$ c(MC2) is the smallest capacity of a any cut in G'
2. :[1] $\Rightarrow c(MC2) < c(MC)$.
3. revenue $= \sum_{C_i \in MC} p_i - \sum_{r_i \in MC} c_i$ is maximized $\Rightarrow \nexists$ another subset MC' in G' s.t contains the sets C, R which produce a revenue' > revenue
4. By the Notation : $c(MC) = P - revenue \Rightarrow revenue = \sum_{C_i \in G'} p_i - c(MC)$
5. :[2] $\Rightarrow revenue2 = P - c(MC2) > P - c(MC) = revenue$.

Contradiction: $\exists$ revenue2 > revenue, therefore revenue was not maximized

Example in G' :



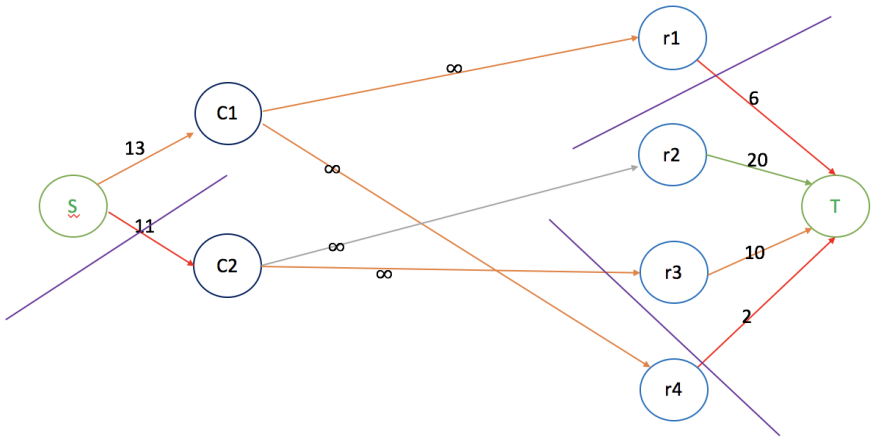
MC = (S,r1,r4,C1)

$$\text{c(MC)} = (P - \sum_{C_i \in MC} p_i) + \sum_{r_i \in MC} c_i = \sum_{C_i \notin MC} p_i + \sum_{r_i \in MC} c_i = 11 + (6+2) = 19$$

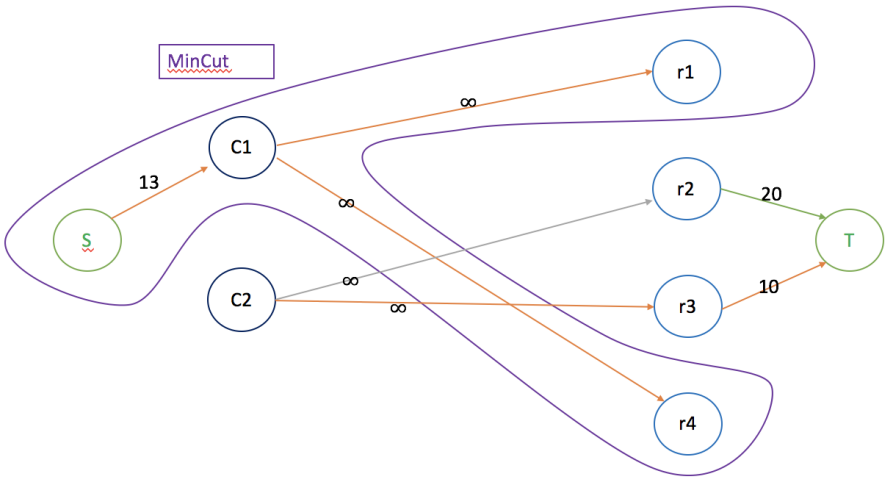
$$\text{revenue} = \sum_{C_i \in MC} p_i - \sum_{r_i \in MC} c_i = 13 - (6+2) = 5$$

$$\text{revenue} = P - c(MC) = \sum_{C_i \in G'} p_i - c(MC) = (13+11) - 19 = 5$$

How we can find the min cut? :
through Ford Fulkerson algorithm for Max Flow, we can find a minimum cut removing from G' one full edge (if there exists) for each path from S to T, choosing the edge as close as possible to S.



The minimum Cut will be the G' separate component that contains S



Algorithm Notation :

- G' = Flow Network: simple Directed graph
  - S = source
  - T = sink
  - Non negative capacity for each edge
  - c(u,v) capacity of the edge u,v in the Graph
  - f(u,v) flow that go through the edge u,v
- Gf = Residual Graph: only edges from G' that can still have more flow
  - cf(u,v) capacity of the edge u,v in the Residual Graph
- AP = Augmenting Paths: Path from S to T
  - Cf(AP) residual capacity = smallest capacity of edges of AP (capacity of the bottle neck of AP)

Algorithm :

```
1  FordFulkerson(G'):
2    Gf = computeGf(G')
3    revenue = 0
4    While there exists AP in Gf:
5      AP = Find augmenting path(Gf)
6      Cf(p) = Find BottleNeckCapacity(AP)
7      for each edge in p:
8        cf(u,v) = cf(u,v) - Cf(p)
9        cf(v,u) = cf(v,u) + cf(p)
10
11   MC = BFS(Gf,G)
12   for each edge e(u,v)in MC:
13     if u is equal to S:
14       revenue = revenue + c(e)
15     else:
16       e2(v,T) = edge (v,T) in G'
17       revenue = revenue - c(e2)
18   return revenue.
```

In line 14 and 17 we just apply the formula that is proved above. revenue = $\sum_{C_i \in MC} p_i - \sum_{r_i \in MC} c_i$
(we can write the revenue as above because we take the resources and nodes which are contained in MC)

- line 14:
  if u $\in e(u,v) \in MC$ is equal to S (source) means, by the graph structure, that v represents a Challenge Node C_i, hence c(e) represent the points that the challenge gives. we have to add those points to respect the formula.
  (note that c(e) = p_i = cost of C_i = points that the challenge i gives)

- line 17: if u $\in e(u,v) \in MC$ is not equal to S (source) means, by the graph structure, that v represents a resource Node R_i, since $R_i \in MC$ means that we must subtract the points that R_i require.
  This cost is represented by the cost of edge e2(v,T)$\in G'$, hence c(e2) represent the points that the resource R_i requires. We must subtract those points to respect the formula.
  (note that c(e2) = c_i = cost of R_i = points that the resources require)

- BFS:(line 11):
  input : Gf,G
  execute BFS in Gf starting from S
  output :sub graph in G' which contains the nodes $\in$ Gf which have no out arrows in Gf.

```
1  BFS(Gf,G):
2    MC = emptySet
3    S = Gf.getSource()
4    A = S.getOutEdges()
5    B = emptySet
6    MC.add(S)
7    while A is not Empty:
8      e(u,v) = A.removeFirst()
9      MC.add(e)
10     B.add(v.getOutEdges())
11     B.remove(e(v,u))
12     e2(u,v) = e(u,v) in G
13     MC.add(e2(u,v))
14   return MC
```

  line 11 we know that there exist always an entry node that returns residual capacity. We have to remove it
  line 12 we have to take into account the capacity of the edge e(u,v) not in Gf but in G

- Consideration : each path from S to T in G' has at most 2 edges with not infinitive capacity, one is a closest edge to S and the other one is a closest edge to T. Therefore at least one of the 2 will be the "Path bottle Neck", and will gets flow = capacity in G' .

- line 11/14 : if u is not equal to S means that u = c_i and v = r_i (is the edge with infinitive capacity).
  Therefore $\exists$ an edge e2(v,T) $\in$ G', this edge has capacity = point that we need to subtract to revenue.

Trivially follows that the revenue respect the formula below:
revenue = $\sum_{e(S,v)\in MC} c(e) - \sum_{e(v,T)\in G', v\in MC} c(e) = \sum_{C_i \in MC} p_i - \sum_{r_i \in MC} c_i$

Algorithm Cost: The cost of the algorithm is equal to the cost of Ford Fulkerson Algorithm = $O(|E| * c(MC))$