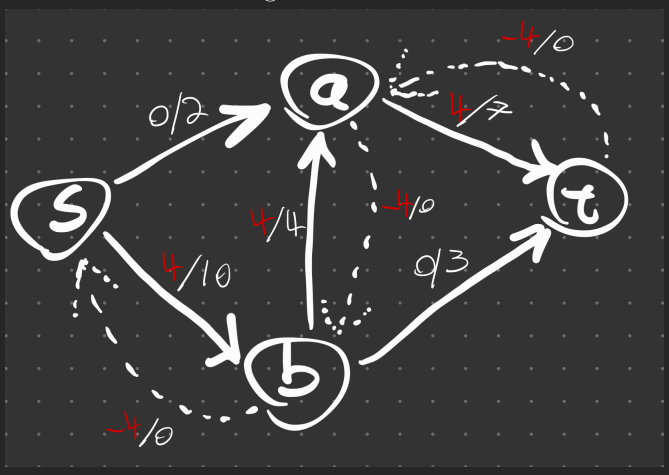
ALGO HW5

Dani Bondar - 206560856

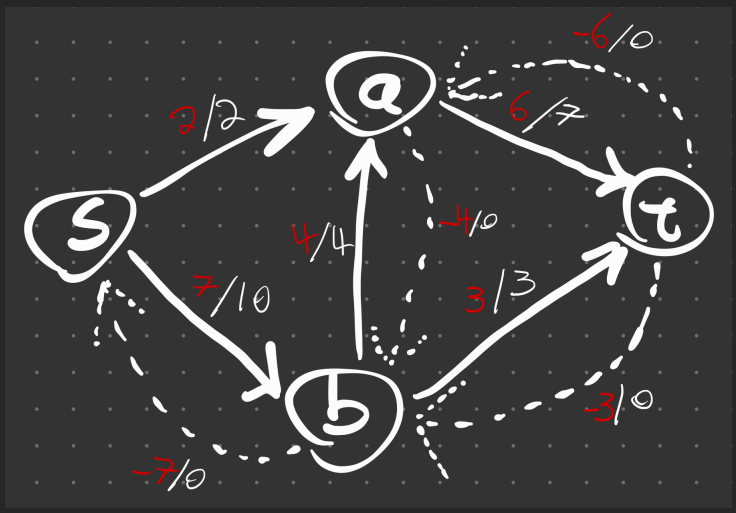
Gur Telem - 206631848

1. Question

1.1. **Section.** Drawing:



1.2. **Section.** Drawing:



1.3. **Section. Yes**.
It's a min cut (although not the only possible min cut).

2. Question

(1) True
(2) False
(3) True

3. Question

Backpack with weight capacity of W. There are n items $\{a_1, \ldots, a_n\}$.

Each item a_i has a weight of $w_i \in N$ and value $p_i \in \{1, 2, \ldots, P\}$.

Given partitioning $G_1, \ldots, G_k$:

$$\dot{\bigcup_{i=1,\ldots,k}} G_i = \{a_1, \ldots, a_n\}$$

Let's use dynamic programming to find the optimal solution (highest total value without exceeding max weight and only a single item at most from each partition):

We'll assume that the set of item a_i can be found in $O(1)$ (if our data structure isn't built in a way to allow that, we can iterate on all the items in each partition, creating a map array s.t. map $[i] = t$ s.t. $a_i \in G_t$). This doesn't increase the total time complexity of our algorithm because it runs in $O(n)$ and only a single time).

We'll note that at most we have n partitions because the partitions can't be empty and there are only n items.

Since there are P values, we can use radix sort to sort the items a_i by value (in each partition as well as in general) in a time complexity of $O(n)$.

The idea:

(1) For each partition, check which item (if any) that you choose, give the best result.
  (a) For each item, calculate the subtask with the next partition and a total max weight of W minus the weight of the item chosen (recursively).
  (b) Calculate also the possibility of not choosing an item (recursively).
  (c) Choose the best solution and save it for future reference

Let's define the recursive formula

$$R(s,t) = \begin{cases} -\infty & t < 0 \\ 0 & t \geq 0 \wedge j \leq 0 \\ \max\{\text{maxValue}(s-1,t), \max_{a_i \in G_s}\{p_i + \text{maxValue}(s-1, t-w_i)\}\} & t \geq 0 j > 0 \end{cases}$$

Let's prove that calculating a matrix s.t. its values are the values of this recursive formula give a matrix of results for n, W (cell index) and the calculation of that matrix is done in $O(nW)$.

The algorithm (we'll mark it's name maxValue):

Done only once:

(1) Initialize a matrix of size $k+1 \times W+1$ (including 0 indexes) with all the values set to 0
(2) Call maxValue(k, W)
(3) Return solutions$[k][W]$

And the recursion

(1) The parameters are s, t respectively
(2) If $s = 0$ then save solutions$[s][t] = 0$ and return
(3) Init max $= 0$
(4) For each $a_i \in G_s$
  (a) Skip if $[s-1][t-w_i]$ are negative indexes (either/both)
  (b) Check if the solution solutions$[s-1][t-w_i]$ doesn't exist, call maxValue$(s-1, t-w_i)$ first
  (c) If that solution is $>$ max then max $=$ solutions$[s-1][t-w_i] + p_i$
(5) If solutions$[s-1][t] >$ max then max $=$ solutions$[s-1][t]$ (again, if it doesn't exist, calculate and come back)
(6) Save solutions$[s][t] =$ max and return

Correctness:

Proof. Before we start, let's assume that $[0][0]$ is top left for convenience.

In induction.

Base: For $k = 0$ ($\forall W \geq 0$):

Always returns 0 because of the second step of the recursion (there's no partition to choose from).

Step: Assume for all values left and above. For total max weight, same or lower (i.e. above or equal).

Let's assume that the algorithm calculates all the values left and above are calculated correctly. So for $(k+1, W)$

We iterate over the items in G_{k+1}, for each, we use only values in the matrix that are above and left to $(k+1, W)$, so they're already calculated and are correct (so each is retrieved in $O(1)$).

Proof. Case 1: The optimal solution includes an item from G_{k+1}.
Let's mark the value of the optimal solution B. Let's assume that a_i was the item chosen from G_{k+1}.
$w_i \leq W$ because otherwise we can't choose a_i because alone, it's bringing us to overweight.
BWoC: $B > p_i + \text{solutions}[k][W - w_i]$.

Proof. Thus, $B - p_i$ gives us a better solution for k and $W - w_i$. In contradiction to $\text{solutions}[k][W - w_i]$ being a correct optimal value. □

Thus, $B \leq p_i + \text{solutions}[k][W - w_i]$.
Since $\text{solutions}[k][W - w_i]$ is a correct solution, we can add a_i to the arrangement that gave that solution and we got a solution to $k+1, W - w_i + w_i = W$. Because B is an optimal solution $p_i + \text{solutions}[k][W - w_i]$ can't be bigger so we got

$$B = p_i + \text{solutions}[k][W - w_i]$$

And finally, we proves that the formula is correct in the case that we use an item from G_{k+1}. □

And for the second case

Proof. Case 2: The optimal solution B for $k+1, W$, doesn't include an item from G_{k+1}.
In that case, the algorithm will return $\text{solutions}[k][W]$.

Proof. BWoC: $\text{solutions}[k][W] < B$.
From the case's assumption, B doesn't include any items from G_{k+1}. So it only includes items from G_i for $1 \leq i \leq k$. Meaning that B is a better solution for k, W in contradiction to the induction hypothesis. □

$\text{solutions}[k][W]$ is also a solution for $k+1, W$ (even if not optimal) because it for $k+1$ we have one limit **less**.
So $B = \text{solutions}[k][W]$ because it can't be better than the optimal. □

These two cases cover all the options. So the algorithm is with that we finished the proof for the induction step.
So the algorithm is correct. □

Complexity:

Proof. Each cell is calculated at most once.
BWoC: the complexity is greater that $O(nW)$.
In each recursive call we only check cells from the row adjacent to the left of the current s. So at most $t+1$ for each recursive call ($O(n) = s$ and $O(W) = t$). We do ofc at most $|G_s|$ iterations like this (this sums in total to n ofc).
So the total complexity is

$$\sum_{i=1}^{k} O(|G_i| \cdot t) = \sum_{i=1}^{k} O(|G_i| \cdot O(W)) = O\left(\sum_{i=1}^{k} |G_i| \cdot W\right) = O\left(W \cdot \sum_{i=1}^{k} |G_i|\right) = O(nW)$$

As requested. □

Regarding space complexity, there's the matrix and the recursion, the matrix takes $O(nW)$ (the its size) and the max depth of the recursion is $k+1 = O(n)$ (at most we have n partitions).
So in total space complexity we get $O(nW)$ (each recursive call takes $O(1)$ of extra space).
Q.E.D

4. Question

N types of coins $c_1, \ldots, c_N$. The value of c_i is $w_i \in \mathbb{N}$.

Let's assume that all the w_i are not only natural but also positive (we'll just discard $w_i = 0$ as they only increase the number of coins used without contributing anything).

Recursive algorithm for finding the min amount of coins to make k.

The idea:

(1) If $k = 0$ then return 0.
(2) If $k > 0$ then return $\min \{1 + \text{minCoins}(k - w_i) \mid 1 \leq i \leq N\}$.

To make the solution optimal, we'll use dynamic programming, maintaining an array of solutions for different values of k.

So the formal algorithm:

(1) If $k = 0$ return 0.
(2) Create an array of solutions of size k (values initialized to ∞).
(3) For $i \in \{1, \ldots, k\}$:
    (a) $\min = \infty$.
    (b) For $t \in \{1, \ldots, N\}$:
        (i) if $\text{solutions}[i - w_t] > 0$ and $\text{solutions}[i - w_t] + 1 < \min$ then $\min = \text{solutions}[i - w_t] + 1$.
    (c) $\mathit{solutions}[i] = \min$

The runtime of the algorithm is $O(kN)$ with N being a constant we get $O(k)$ as requested.

Correctness:

Proof. For $k = 0$ and using 0 coins, the answer is correct.

Now let us assume that for $k = 1, \ldots, n$ the algorithm is correct. RTP: for $k = n + 1$.

Using the algorithm for $n + 1$ and n is the same except that for $n + 1$ we have an extra iteration at the end (notice that the inner part of the algorithm, step 3, doesn't refer to k).

So as per the assumption, on the n-th iteration, we got a correct result for all the values from 0 to n.

So assuming we have the result for those values that we need to change, we can check N possibilities of starting from coin c_i and then checking which gives us the minimal result from $\mathit{solutions}[n + 1 - w_i]$ (all of which are already calculated and correct as per the induction assumption).

This covers all the options for a starting coin. Let's mark the min num of coins as a.

BWoC, let's assume there's a better solution, let's assume walog, that the first coin is c_i and a total number of coins $b < a$.

This means that we found a better solution for $k - w_i$ because without one of the coins c_i, we get a solution with $b - 1 < a - 1$ coins for $k - w_i$. □

Complexity:

Proof. We iterate k times (dynamic):

Each iterations does a constant number of operations.

So the total runtime is $O(k)$. □

Q.E.D