KØBENHAVNS UNIVERSITET

Assignment 3

Alex (zhc522), Adit (hjg708), Usama (mhw630), Nikolaj (bxz911)

AD 2023

10. april 2023

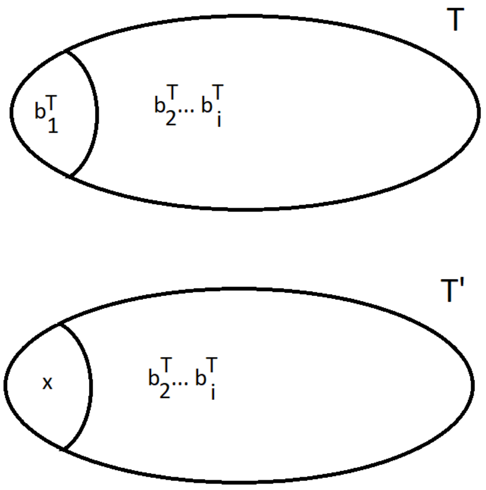
Task 1

In the pseudocode, b[] is an array of how many beers are in each bar, p[] is an array of the distances between the bars and minB is the targeted beers in each bar.

```
beerRun(b[], p[], minB)
    n = b[].length
    //-1 because we compare the ith element and i+1th element,
        //otherwise it would compare with an index out of bounds.
    for i = 1 to n-1
        //if bar i has less than the targeted beer amount
        if (b[i] < minB)
            //Add the nessesary amount of beers to bar i
            b[i] += (minB - b[i])
            //Remove the cost and the beers being transported
            b[i+1] -= (p[i+1] - p[i])*2 + (minB - b[i])
        //If bar i has more beers than the targeted beer amount
        else
            //Move excess beers from bar i to bar i+1
            if ((b[i] - minB > p[i+1] - p[i])*2)
                //Set the beers in bar i to the targeted beer amount
                b[i] = minB
                //Add the beers to bar i+1
                b[i+1] += b[i] - minB - (p[i+1] - p[i])*2

    //If the last bar has less than the minimum amount of beers
        //then it isnt possible and should return false
    if (b[n] < minB)
        return false
    else
        return true
```

Task 2



Figur 1: The solution T and the solution T'

We will using a direct proof to prove the greedy choice and the substructures. Our greedy choice works as by two cases.

1. Either bar b_i has less than $\bar{b}$ beers. Then we take beers from bar b_{i+1} and send it back to bar b_i, even if bar b_{i+1} goes negative.
2. Bar b_i has more than $\bar{b}$ beers and the cost of moving is lower than the surplus in beer in bar b_i. Then we would send the excessive beers to bar b_{i+1}.

The algorithm checks whether it is possible to have at least $\bar{b}$ beers in every bar. Unlike the rod-cut problem, we are not finding the optimal solution. We are just finding whether or not a solution is possible. If not, then are infinite solutions. For a direct proof, let us assume that T is a possible solution, that can be divided by every bar. For the solution for bar 1 called b_1^T. It needs to be at least as big or bigger than $\bar{b}$, because if $b_1^T < \bar{n}$ then the solution is not possible.

We define the greedy choice for b_1 as x_1. Which means that x_1 is bigger or equal to $\bar{b}$. With this we define T' which denotes a solution for our greedy algorithm. T' defines as T without b_1^T, but we added x_1 instead. Figure 1 above shows that the difference between T and T' are b_1^T and x_1. Because b_1^T and x_1 have the same properties of being bigger or equal to $\bar{b}$, can we state that T' is a solution. It is important to mention that x_1 and b_1^T are not necessarily equal in term of values.

Optimal substructure property states that an optimal solution to a problem includes optimal solutions to its subproblems. Meaning that by finding the optimal solutions for all the subproblems can we find the optimal solution for the true problem. Because the cost to transfer beers is linear, we can move on to the next bar/subproblem easily. Our algorithm locks the first solved bar, before moving on to the next one, meaning if the first bar has been solved then we can solve the next bar in the same fashion. By repeating this process, we can traverse through all the subproblem in the given problems, so we know if a solution is possible.

As our algorithm satisfies both the greedy choice property and optimal substructure property, we can conclude that it provides a correct solution to the problem.

Task 3

In this pseudocode, b[] is an array of how many beers are in each bar, p[] is an array of the distances between the bars and minB is the targeted beers in each bar.

```
beerRunBinarySearch(b[], p[], minB)
    lower = 0
    higher = max(b[]) // higher is the amount of beers in the bar with the most beers
    while lower <= higher
        //Set the targeted beer amount to midway between lower and higher
        minB = lower + (higher - lower) / 2
        if beerRun(b[], p[], minB)
            //If the targeted amount of beers is possible
                //we move the lower to the targeted amount
            lower = minB
        else
            //If the targeted amount of beers isnt possible we move
                //the higher to the targeted amount - 1
            higher = minB - 1
    return minB
```

This code works by checking if the midway point between a lower cap and a higher cap is possible, if it is we move the lower cap to the midway point and then call again, and if it is not possible we move the higher cap to the midway point - 1. This checks if the targeted amount of beer is possible, but there is the chance that a higher amount is possible, so we keep checking until we are sure there is not a higher amount of beers that are possible.

This is binary search that calls the pseudocode from task one. The pseudocode from task one runs in $O(n)$ and binary search runs in $O(log\, B)$. As the binary search calls the pseudocode from task one in each iteration, the runtime is $O(n \cdot log\, B)$.