Homework 4

Yousef Suleiman | Due: Apr 2

Question 1

- Sally's optimization problem is similar to the 0-1 knapsack problem
- we can solve the problem with dynamic programming

```
W is an array of m widgets being asked for
B is an array of m corresponding bid amount
initialize matrix M of dimensions m by n with nil
/*
    i is the index in the W and P arrays
    r is the number of widgets Sally has remaining
*/
function SallyBids(i, r) {
    /* have we already considered this situation? */
    if M[i, r] is not nil then return M[i, r]
    /* if there are there bids left or Sally ran out of widgets, terminate */
    if i > m or r == 0 then result := 0
    /* if the widgets being asked for as more than what Sally has, skip the
bid */
    else if W[i] > r then result := SallyBids(i - 1, r)
    /*
    otherwise consider the situation where we take the bid and where
    x: Sally skips the bid
    y: Sally takes the bid
    then take the situation that yields the most revenue
    */
    else {
        x := SallyBids(i + 1, r)
        y := B[i] + SallyBids(i + 1, r - W[i])
        result := max(x, y)
    }
    /* memoize */
    M[i, r] = result
    return result
}
```

- dynamic programming in this algorithm works for this problem as it has optimal substructure and overlapping subproblems
    - **optimal substructure**
        - at each step, Sally decides whether to include the current bid or not based on the remaining widgets `r` and bid index `i`
        - by recursively considering all possible combinations of bids, the function ensures that the optimal solution for the entire problem can be constructed from the optimal solutions of its subproblems
    - **overlapping subproblems**
        - solutions to subproblems are memoized in matrix `M`

        - if the solution to a subproblem is already done, it is taken from `M` to avoid redundant computations
        - `M` stores solutions to subproblems indexed by the bid index `i` and the remaining widgets `r` covering all possible combinations of subproblems
- if the bidder accepts partial lots then the problem is similar to the fractional knapsack problem which can be solved greedily
    - iterate through the bids in descending order of d_i/k_i
    - sell the k_i if Sally has k_i still available
    - otherwise sell what is left $< k_i$ at the rate of d_i/k_i and terminate

Question 2

(a)

- given $x_1, x_2, \ldots, x_d$ which is in increasing order
- take x_d which is the furthest building. Place a tower at $t_j = x_d - 1$ where $j = 1$ (i.e. this is our first tower)
- from $i = d - 1$ to 1
    - if $|t_j - x_i| > 1$ such that the building is not in range of the last tower then
        - $j \coloneqq j + 1$
        - place a tower at $t_j = x_i - 1$

(b)

Base Case

Notice that if $d = 1$ such that there is only one building then the algorithm places one tower that covers the building which must an optimal solution as you need at least one tower to cover one building.

Inductive Step

- assume our algorithm gives the optimal solution for $x_2, x_3, \ldots, x_m$ as $T : \{t_1, \ldots, t_j\}$
- given a new problem as $x_1, x_2, x_3, \ldots, x_m$, our algorithm will give the solution T'. There are two cases for what T' looks like
    1. the first case is that $T' \equiv T$. This is when t_j (the last tower placed in T) also covers x_1. This must be an optimal solution as it has the same solution as its subproblem's solution where the subproblem's solution is minimum (as it is optimal by our assumption) such that this new solution T' must also be optimal.
    2. the second case is that $T' = T \cup \{t_{j+1}\}$. This is when x_1 is not covered by t_j so the algorithm adds a new tower
        - since x_1 is not covered by any tower in T, x_1 must be at least 2 units away from the nearest tower t_j in T
        - by placing tower t_{j+1} at $x_1 - 1$, we ensure that x_1 is covered. Since there is no tower within distance 1 from x_1, this tower placement is necessary to cover x_1 or else T wouldn't be optimal and that contradicts our assumption
        - thus, T' is also optimal

Question 3

```
cost is an array of n costs
/*
M will hold the minimum cost to get to the ith step
*/
initialize an array M of dimension n + 1 with nil
/*
starting from steps 0 and 1 are free
*/
M[0] := 0
M[1] := 0
for i from 2 to n {
    /*
    consider taking 1 or 2 steps and
    store the minimum in M
    */
    M[i] = min(M[i - 2] + cost[i - 2], M[i - 1] + cost[i - 1])
}
return M[n]
```