Algorithm Theory - Assignment 3

Téo Bouvard

February 23, 2020

Problem 1

a) If we can prove that E has majority $e_m \implies E - \{e_i, e_j\}$ also has e_m as majority, we can devise a simple algorithm that just removes pairs of different elements from the original sequence. At some point, one of two things is going to happen.

- We can't pick a pair of different elements because all elements in the sequence are identical. This implies that this repeating element is the majority of E.
- We have an empty sequence. This implies that E has no majority.

Here is the algorithm in pseudo code, which returns TRUE if E has a majority and FALSE otherwise.

```
MAJORITY(E)
    if E = ∅
        return FALSE
    if ALL-EQUAL(E)
        return TRUE
    pair = FIND-PAIR(E)
    return MAJORITY(E \ pair)
```

The above algorithm uses two subroutines.

- ALL-EQUAL returns TRUE if all elements of a sequence are equal, and FALSE otherwise. We do not check for empty sequences as this subroutine is called after this check in the MAJORITY algorithm.
- FIND-PAIR returns a sequence of two distinct elements from the input sequence. We only need to consider sequences who actually have at least two distinct elements, because we checked for empty sets and sets containing identical elements before calling this subroutine.

```
ALL-EQUAL(E)
    elem = E[0]
    for i = 1 to E.length
        if elem ≠ E[i]
            return FALSE
    return TRUE
```

```
FIND-PAIR(E)
    first = E[0]
    for k = 1 to E.length
        if first ≠ E[k]
            return {first, E[k]}
```

b) This problem has optimal substructure because the optimal solution is constructed from optimal solutions of its subproblems. However, there is no overlap among the subproblems. Dynamic programming requires both optimal substructure and overlapping subproblems to be applied, so we can't use it in this case.

Problem 2

This problem is an instance of the fractional knapsack problem. As Maria can get partial points if a problem is solved partially, her best strategy is to solve the problems with hihest point density first. That is, solve the problem with the highest $\frac{p_i}{t_i}$, until either the exam time runs out, in which case the algorithm exits, or the problem is solved, in which case she picks the next problem.

Problem 3

Let $X = \{x_0, \ldots, x_n\}$ be the sequence of coordinates of all houses along the road, sorted in increasing order. Let R be the coverage radius of a base station, in this case $R = 8$. If $X = \emptyset$, we do not need to place any base. If $X \neq \emptyset$, the first base we place should at least include x_0, and maximize the number of houses in its coverage radius in order to minimize the number of bases to place. These two conditions are sufficient to derive an efficient algorithm to solve this problem. Let b be the coordinate of the optimally placed first base.

- The base should at least include $x_0 \implies \|b - x_0\| \leq R$, otherwise the house is not close enough to be included in the coverage radius of the base.
- To maximize the area covered by the base, we should place it as far as possible from x_0. This implies $b = \begin{cases} x_0 + R \\ x_0 - R \end{cases}$
- As the coordinates are sorted in increasing order, we know that there are no houses before x_0 so choosing $b = x_0 - R$ would cover at most one house. However, there are possibly more houses after x_0, so choosing $b = x_0 + R$ is guaranteed to cover at least one house.

From the previous reasoning, we see that the optimal placement of the first base is $b = x_0 + R$. We can then derive a recursive algorithm to solve this problem. We introduce the parameter B which represent the coordinates of the base stations. The first call to this function uses an empty sequence as parameter, and B gets populated as it goes down the recursion calls. The number of base stations necessary can be obtained by checking the size of B after the function call, but it could also be added as parameter incremented at each recursive call. For clarity, this algorithm modifies X in-place by removing all covered houses each time a base station is placed. However, it would be more efficient (implementaion-wise) to not modify the sequence and just keep track of which houses are already covered, by keeping an index for example.

```
STATIONS(X, R, B)
    if X = ∅
        return
    firstHouse = X[0]
    B = B ∪ {firstHouse + R}
    while X ≠ ∅ & X[0] ≤ firstHouse + R
        X = X \ X[0]
    return STATIONS(X, R, B)
```