Algorithm Theory - Assignment 3

Téo Bouvard

March 1, 2020

Problem 1

a) If we can prove that E has majority e_m implies that $E - \{e_i, e_j\}$ also has e_m as majority, we can devise a simple algorithm that just removes pairs of distinct 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 that condition is checked, in the first line of MAJORITY.
- 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 having the highest 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 using the same rule.

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 a greedy algorithm solving 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 $b = x_0 + R$ or $b = x_0 - R$
- 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. The function STATION-COORDINATES returns a sequence representing the coordinates of the base stations. The number of base stations to be placed can be determined by checking the size of the sequence returned by the function. If we were only interested in the number of base stations to place, we could replace the sequence returned by additions. This is demonstrated in the STATIONS-COUNTER function. 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 a pointer to the first house not covered, for example.

STATIONS-COORDINATES(X, R)
    if $X = \emptyset$
        return NIL
    $firstHouse = X[0]$
    while $X \neq \emptyset \wedge X[0] \leq firstHouse + R$
        $X = X \setminus X[0]$
    return STATIONS-COORDINATES$(X, R) \cup \{firstHouse + R\}$

STATIONS-COUNTER(X, R)
    if $X = \emptyset$
        return 0
    $firstHouse = X[0]$
    while $X \neq \emptyset \wedge X[0] \leq firstHouse + R$
        $X = X \setminus X[0]$
    return STATIONS-COUNTER$(X, R) + 1$

Problem 4

I am not certain that the algorithm I found is really greedy, as it performs a linear scan of the sequence, but this is the only efficient way I found to solve this problem. The idea is to scan the sequence and keep track of the largest set at each time, comparing each new set with the largest one found until now.

```
LARGEST-SET(X)
    maxlen = 0
    maxseq = 0
    for i = 0 to X.length − 1
        j = i + 1
        n = 2
        while S[j] − S[i] ≤ 1
            j = j + 1
            n = n + 1
        if n − 1 ≥ maxlen
            maxlen = n − 1
            maxseq = i
        return maxseq, maxlen
```

This function returns the index of the first element in the largest set, and the length of this set. The outer loop is limited to the number of elements in the sequence, but the inner loop has a variable size. However, the inner loop is at most as long as the largest interval returned, so it loops for at most $maxlen$ iterations. As such, for a sequence of length n having a largest interval of size l, the time complexity of this algorithm is $\mathcal{O}(n \times l)$.