CS 413 Homework 5

Sterling Jeppson

March 14, 2021

Problem 5

To solve this problem start at the western-most endpoint and proceed east. When the first house is encountered place a station 4 miles to the east of the house and eliminate all those houses which are within an inclusive range of 4 miles east or west of the station. Now continue to the next house which has not been eliminated and again place a station 4 miles to the east of this house. Repeat this procedure until every house is exists in a range covered by a station. This will be the minimum number of stations required to cover all the houses. The algorithm is given formally below:

```
int minStations(vector<int> houseLocations, int broadcastDist) {
    int numStations = 0, coveredDist = 0;
    for(const auto& houseDist : houseLocations) {
        if(houseDist > coveredDist) {
            coveredDist = houseDist + broadcastDist;
            numStations++;
        }
    }
    return numStations;
}
```

Since the for loop iterates n times where n is the number of houses and since the body of the loop runs in $O(1)$ time we conclude that the algorithm runs in $O(n)$ time.

In order to prove that the algorithm is optimal we will show that it always stays ahead of an optimal solution. Let s_k be the distance from the western starting point of the kth station and let $S = \{s_1, ..., s_m\}$ be the set of m sorted station positions which are placed by our algorithm to cover some collection of houses. Now define an optimal set of sorted station positions $Q = \{q_1, ..., q_p\}$.

Proposition. *After the kth station is placed, all houses in the range $[0, s_k + 4]$ will be covered and $s_k \geq q_k$ for $1 \leq k \leq p$.*

Proof. Let the property $P(k)$ be the proposition to be proved.

Show that $P(1)$ is true: Clearly $s_1 \geq q_1$ because s_1 is selected by the algorithm to be as far to the east of the first house as possible. Since s_1 has a covering range of 4 miles in both directions and since s_1 is placed 4 miles to the east of the first house, s_1 also covers all houses in the range $[0, s_1 + 4]$

Show that for all integers $k \geq 1$, $P(k) \implies P(k+1)$: Let k be any integer with $1 \leq k < p$. By inductive hypothesis $s_k \geq q_k$ and s_k covers all houses in the range $[0, s_k + 4]$. It follows that $\{s_1, ..., s_k\}$ covers at least all of the houses that $\{q_1, ..., q_k\}$ covers. This means that the next house which is outside of the range $[0, s_k + 4]$ must also be outside of the range of $[0, q_k + 4]$. But when this house is encountered by our algorithm the $k + 1$st station will be placed 4 miles to the east which is the max range of the station. Hence $s_{k+1} \geq q_{k+1}$ and all houses in range $[0, s_{k+1} + 4]$ are covered. □

Suppose our algorithm is not optimal. Then $m > p$ and so stations placed at positions $\{s_1, ..., s_p\}$ do not cover every house. But stations at $\{s_1, ..., s_p\}$ cover every house in range $[0, s_p + 4]$ and $s_p \geq q_p$. Hence Q is not optimal, which is a contradiction.

Problem 6

No matter which order the contestants are sent in, the amount of time needed to complete the swimming portion of the event is constant. This is because only 1 participant can use the pool at a time. Therefore, in order to minimize the completion time, send out participants in order of decreasing bike time plus run time. That is to say send out those participants first who have the slowest combined run and bike times. In this way the overlap between the passage of mandatory swim time and the other events time is maximized(there is no additional cost).

More formally let there be n participants in the event, and let the swim, bike, and run times of the kth participant be given as s_k, b_k, and r_k. Send out the n participants so that $b_1 + r_1 \geq ... \geq b_n + r_n$. In this way the event time will be minimized.

Proposition. *In any event where $b_k + r_k < b_{k+1} + r_{k+1}$ for $1 \leq k \leq n-1$, swapping the order of the kth and $k+1$st participant will not increase the event time.*

Proof. Swap participant k and $k+1$ so that participant $k+1$ starts the event before participant k. It follows that participant $k+1$ will finish the event at a sooner time then in the original ordering. Also, participant k will now complete the swimming portion of the event at the same time as participant $k+1$ would have in the previous ordering. Since $b_k + r_k < b_{k+1} + r_{k+1}$, participant k will complete the event sooner than participant $k+1$ did in the original ordering. It follows that the event time will not increase. □

Finally, suppose that there is an optimal ordering that does not follow the ordering prescribed by our solution. Now find every instance of an ordering violation and correct it. Since there can only be $(n^2 - n)/2$ of such occurrences the process must eventually come to an end. At each swap, the total event time will not have been increased and so in the final ordering where $b_1 + r_1 \geq ... \geq b_n + r_n$, the total event time will not be increased. Hence our ordering produces a minimal event time.

We can write an efficient algorithm which runs in $O(n \log n)$ and returns the minimum event time for a vector of n athletes.

```
int minEventTime(vector<Athlete> athletes) {
    sort(athletes.begin(), athletes.end(),
        [](const Athlete& a1, const Athlete& a2) {
            return a1.bikeRun_T() < a2.bikeRun_T();
    });

    int totalSwim_T = accumulate(athletes.begin(), athletes.end(), 0,
        [](int sum, const Athlete& athlete){return sum + athlete.swim_T;});
    int total_T = totalSwim_T + athletes.front().bikeRun_T();
    int swim_TCurrent = 0;

    for_each(athletes.rbegin(), athletes.rend(),
        [&](const Athlete& athlete) {
        swim_TCurrent += athlete.swim_T;
        total_T = max(total_T, swim_TCurrent + athlete.bikeRun_T());
    });
    return total_T;
}
```

The above algorithm sorts the athletes according to increasing bike time + run time. However this is not contrary to the required ordering because we simply consider the last element in the vector to be the first athlete to start. The total swim time of all

athletes is accumulated because we know that the total run time cannot be less than this. We then take the currently minimum possible total time to be the max of the current minimum possible total time and the sum of the time from the start of the race with the time that will be required for the current athlete to complete the race. The immediate incorrectness of the algorithm can be ruled out with a simple algorithm that models the natural process of running the race and is therefore correct by default. That is each iteration of the loop simulates the events that occur with all the athletes during a single unit of time. The algorithm is very inefficient and runs in time $O(mn)$ where n is the number of athletes and m is the max number of time units required by one athlete in any event. The algorithm is given below:

```
int minEventTime_Method2(vector<Athlete> athletes) {
    sort(athletes.begin(), athletes.end(),
        [](const Athlete& a1, const Athlete& a2) {
            return a1.bikeRun_T() < a2.bikeRun_T();
    });

    int totalTime{};
    vector<Athlete> bikeEvent, runEvent;
    //While loop runs in O(m) where m = max number of time units
    //of all events of all athletes
    while(bikeEvent.size() or runEvent.size() or athletes.size()) {
        //Inner loop runs in time O(n) where n = athletes.size()
        bikeEvent.erase(
            std::remove_if(bikeEvent.begin(), bikeEvent.end(),
                [&](Athlete& athlete) {
                    if(!(--athlete.bike_T)) {
                        athlete.run_T++;
                        runEvent.push_back(athlete);
                        return true;}
                    return false;
            }),
            bikeEvent.end());

        runEvent.erase(
            std::remove_if(runEvent.begin(), runEvent.end(),
                [](Athlete& athlete) { return !(--athlete.run_T); }),
            runEvent.end());

        if(athletes.size() and !(--athletes.back().swim_T)) {
            bikeEvent.push_back(athletes.back());
            athletes.pop_back();
        }
        totalTime++;
    }
    return totalTime;
}
```

Even though we have already proved the correctness of the ordering of the athletes we can achieve an additional verification of correctness with a brute force algorithm that runs in $O(n \cdot n!)$. $n!$ for the number of permutations of a list of n items, and n for the number of comparisons to generate each permutation. This algorithm generates every possible ordering of the athletes and runs them all against algorithm 2, returning the minimum run time. In order for the algorithm to work the sorting portion of algorithm 2 needs to be eliminated. Also the algorithm is not practical for all but the smallest inputs due to the growth rate of $n!$. The algorithm is given below:

```
int minEventTime_Method3(vector<Athlete> athletes) {
    sort(athletes.begin(), athletes.end(),
        [](const Athlete& a1, const Athlete& a2) {
            return a1.bikeRun_T() < a2.bikeRun_T();
    });

    int minTime{numeric_limits<int>::max()};

    do {
        int temp = minEventTime_Method2(athletes);
        cout << temp << endl;
        minTime = min(minTime, temp);
    }
    while(std::next_permutation(athletes.begin(), athletes.end(),
        [](const Athlete& a1, const Athlete& a2) {
            return a1.bikeRun_T() < a2.bikeRun_T();
    }));
    return minTime;
}
```