Group Assignment 2 Solution

February 9, 2016

Definition:

- *tennis*$[i]$: the location of i^{th} tennis ball
- *tennis_in*$[i]$: is $True$ if there is a tennis ball in the i^{th} locker, otherwise its $False$.
- *key*$[i]$: location of i^{th} key.
- *key_in*$[i]$: is $True$ if we are given the i^{th} key as the initially given key, otherwise its $False$.

1 Naive Method (Brute-force) $O(N2^M)$

For each key that you are initially given, you can either use it or ignore it. So, there are a total of 2^M possible ways of using keys. It's clear that if there is a tennis ball between two used keys then the best possible way is either using the left opened locker to reach it or use the right one. We can use a linear time algorithm for deciding the best split possible that minimizes the number of lockers that must be opened to collect all tennis balls between the two used keys. The linear algorithm is straightforward: between two opened lockers with index i and j, we iterate through all $x(i \leq x \leq j)$ where the locker x contains a tennis ball. Then, we simply compute the cost of x^{th} locker to be the last locker which is opened from left and all of the remaining tennis balls that are to the right side of x will be opened from the right side. Among all solutions, we will choose the one that minimizes total opening lockers.

Algorithm 1 Bruteforce algorithm

```
1: ans ← $\infty$
2: **for** all possible subsets S of the keys **do**
3:     temp ← 0
4:     **for** every consecutive pair (i, j) of keys in S **do**
5:         temp ← temp + solve(i, j)
       temp ← $temp + goLeft(S_{first}) + goRight(S_{last})$
6:     ans ← min($ans, temp$)
   return $ans$
```

The $solve(i, j)$ computes the minimum split assuming the key at i^{th} and j^{th} index are used. We define $next[i]$ as the index of the closest locker to the right of the i^{th} locker that contains a tennis ball (You can compute it with a simple

loop in linear time)
goLeft will compute the number of lockers that must be opened to collect all tennis balls between the first locker and the first initially given key S_{first}. *goRight* will compute the number of lockers that must be opened to collect all tennis balls between the last locker (N^{th} locker) and the last initially given key S_{last}. Computing these two arrays is straightforward and you can compute them in linear time. The pseudo-code for $solve(i, j)$ is provided in algo. 2

Algorithm 2 Computing solve(i, j)

```
1:  if there isn't a tennis ball between $i^{th}$ locker and $j^{th}$ locker return 0
2:  ans ← # of lockers that it will be open if we collect all tennis balls with $j^{th}$
    key
3:  x ← $i + 1$
4:  while x ≤ j do
5:      if tennis_in[x] then
6:          ans = min($ans, (x - i)$)
7:          if $next[x] < j$ then
8:              $ans \leftarrow ans + j - next[x]$
9:      $x \leftarrow x + 1$
10: return ans
```

1.1 Time Complexity Analysis

Since we iterate through all possible subsets of the keys and in each iteration we find the best split in linear time, the total running time of the algorithms is $O(N2^M)$.

2 Dynamic Programming $O(NM^2)$

By using dynamic programming, we can improve the time complexity of our algorithm to polynomial time $O(NM^2)$.
Let $d[i]$ be the minimum number of lockers that must be opened to gather all the tennis balls between the first locker and the locker that key i opens assuming that key i is used. We can iterate through all possible choices of the previous used key, and compare them with the possibility of using only the i^{th} key, and choose the one that is smallest. To do this, we can put the initial value of $d[i]$ to be the value where only the i^{th} key is used, that is:

$$d[i] = \begin{cases} key[i] - tennis[1] + 1 & tennis[1] < key[i] \\ 1 & otherwise \end{cases} \qquad \forall 1 \leq i \leq M \tag{1}$$

Then we try to compare it with the rest of the options and pick the minimum:

$$d[i] = \min_{1 \leq j < i} (d[j] + 1 + solve(key[j], key[i]), d[i]) \tag{2}$$

Same as the first algorithm, The $solve(i, j)$ computes the minimum split assuming the key at i^{th} and j^{th} index are used. The pseudocode for this algorithm follows straightfowardly from the recursion defined above, is thus omitted.

2.1 Time Complexity Analysis

To compute $d[i]$ we iterate through all possible value of j from $1, \ldots, i-1$ and each iteration is done in a linear time $O(N)$. Computing $next[i]$ can be done in linear time $O(N)$. So, the total time complexity of the algorithm is $O(N * N * M)$.

3 Efficient Dynamic Programming $O(T^2 + M)$

The $O(T^2 + M)$ algorithm is based on an observation. The optimal solution to this problem simply tries to find some segments with minimum total length that cover all tennis balls and in each segment there exists at least one initially given key. We will define $d[i]$ to be the minimum number of lockers to open to collect all of the tennis balls $1 \ldots i$. The base case is $d[0] = 0$. To compute $d[i]$, we just need to figure out where the last segment starts. It could start from any of the previous tennis balls j. If the starting position of the last segment is j, then the total number of lockers to be opened would include the number required to cover up to ball $j-1$, and then open all the lockers between j and i. We then just need to pick the j that achieves the minimum is captured with the following simple recursion:

$$d[i] = \min_{1 \le j \le i} (d[j-1] + tennis[i] - tennis[j] + 1 + checkIt(tennis[j], tennis[i]))$$

where *checkIt* is a function that returns 0 if there exists at least one key between $tennis[j]$ and $tennis[i]$, otherwise it returns

$$\min(comeFromLeft[tennis[j]], comeFromRight[tennis[i]]).$$

Where $comeFromLeft[i]$ stores the minimum distance from the i^{th} locker to its nearest initially given key on the left and $comeFromRight[i]$ stores the minimum distance from the i^{th} locker to its nearest initially given key on the right. These two arrays can be computed easily in linear time $O(N)$ using the algorithm below:

Algorithm 3 Computing comeFromLeft

```
1: comeFromLeft[0] ← $\infty$
2: for i = 1, i ≤ N, i = i + 1 do
3:     if key_in[i] == true then
4:         comeFromLeft[i] ← 0
5:     else
6:         comeFromLeft[i] ← comeFromLeft[i - 1] + 1
```

Algorithm 4 Computing comeFromRight

```
1: comeFromRight[N + 1] $\leftarrow \infty$
2: for i = N, i $\geq$ 1, i = i - 1 do
3:     if key_in[i] == true then
4:         comeFromRight[i] $\leftarrow$ 0
5:     else
6:         comeFromRight[i] $\leftarrow$ comeFromRight[i + 1] + 1
```

To efficiently decide if there is a key between position j and i, we simply need to compute $sum(i)$ to be the total number of keys in locker $1, 2, ...i$ and check if $sum(i)$ and $sum(j)$ are different. $sum(i)$ can be computed efficiently using the following algorithm in $O(N)$:

Algorithm 5 Computing cumulative sum

```
1: sum[0] $\leftarrow$ 0
2: for i = 1, i $\leq$ N, i = i + 1 do
3:     sum[i] $\leftarrow$ sum[i - 1] + key_in[i]
```

3.1 Time Complexity Analysis

Once we precomputed $sum(i)$, $comFromRight(i)$, $comFromLeft(i)$ in $O(N)$ time, the *checkIt* operation in the recursion becomes constant time. To compute $d[i]$ $(1 \leq i \leq T)$, because each computation depends on j $(1 \leq j < i)$, we have a double loop, leading to $O(T^2)$ time. The overall runtime is thus $O(T^2 + N)$.