Helena Bales, Joshua Bowen

Section 1: Introduction

The algorithm we were instructed to solve was as follows. In Dixon there are N number of lockers. With the exception of Locker 1 and Locker N, every locker has a key to it locked into the locker on either side of it, the exception is that Locker 1 and Locker N only have 1 side. Our task is, using a certain number of initially given keys unlock the least amount of lockers necessary to retrieve a number of tennis balls that are locked into the lockers.

Section 2: Algorithm 1 -- Enumeration

Section 2.1: Pseudocode

```
mincost = N


for all keys
    set to not being used
set first key to being used


for i from 0 to (2^M)-1 do
    cost = 0


            for j from 0 to N-1 do
                L[j].open = 0
                L[j].key = 0
                L[j].ball = 0
            end


            for j from 0 to M do
                if ittr[j] == 1 do
                    L[k[j]].key = 1 && L[k[j]].open = 1
                end
            end


            for j from 0 to T do
                        L[b[j]].ball = 1
            end
            j=0
            while j < N do
                        if current is not open do
                    if current locker is last locker
          t = 0
          while adjacent locker not open
               open adjacent lockers
          end
     end
                    if you have the key for the current locker do
                                   locker.open = 1
                         end
                    if the current locker has a ball and an adj. locker is open
                                   locker.open = 1
                    else if the current locker has a ball do
                         t=0
                         while locker+t.open != 1 && locker-t.open != 1
                              t++
                              if far right
                                   only check left
                              if far left
                                   only check right
                         if locker+t.open == 1 do
                              for x from 0 to t { locker+x.open = 1 }
                         else if locker-t.open == 1 do
                              for x from 0 to t { locker-x.open = 1 }
                         end
                         j+=t
                    end
end
               j++
     end
     for total number of lockers
          if locker open
               cost++
          end
     end


     y=M-1
     while(m[y] == 1)
          m[y] = 0
          y--
```

```
        end
        m[y] = 1


        if cost < mincost { mincost = cost }


end
return mincost
```

Section 2.2: Explanation

The idea for this algorithm is to test every possible combination of keys that could be used. If we treat every key as a binary number where it’s either 0 or 1, not being used or being used, then 2^M-1 represents every single case in which the keys are being used. For example. If you have 3 keys then there are 2^3-1 possible combination of keys or 7 possibilities to be used. 000, 001, 011, 100, 101, 110, 111 are the possible cases. Each bit represents which keys in the array of keys are active at any given iteration. After the keys that are being used for the iteration have been determined we then solve for the fewest number of lockers that need to be opened using the keys we are given. We first assume all keys we are using need to be used and as such all of the lockers corresponding to those keys are openned. We then iterate along the row of lockers looking for tennis balls. If we find a ball we check to see if we have the key already, in which case nothing happens, or we then check to see if a locker 1 locker adjacent is open in which case we open the locker and continue forward. If neither of those cases are true we iterate through the array forward and backwards until we find the nearest open locker. From the nearest open locker we then iterate back to the ball openning all of the lockers along the way. After the fewest lockers have been found with a given combination of keys the algorithm counts the final number of open lockers for that iteration and compares that minimum to the current minimum number of lockers needed to be openned. If the new cost is smaller than the old minimum cost then the minimum cost is replaced with the new one. The algorithm goes through every possible combination of keys and states the smallest cost at the end.

Section 2.3: Runtime

This algorithm will run in O(N2^M). The 2^M comes from there being 2^M key combinations that could be the solution. In the worst case, every single combination will be run. The N comes from the iteration within the 2^M loop. It is actually N+M+T+2(N-1)+1, which in big-O notation, simplifies to N since that is the most significant value (M and T are both always smaller than N). In the previous summation, the first N comes from the loop to set all lockers to closed, the M comes from marking each locker whose key we are using, the T comes from marking each locker with a tennis ball, and the 2N comes from the final loop. The best case for that loop would be N, and the worst case would be when the first locker had a ball, and the last locker was the only one with a key. If that happened, then we would look forward to the array until we reached the last element (N-1) then would loop back through those to set each locker to be open (N-1).

Then we would skip the entire array and set the last locker (the one with the key) to open (1).
The lower bound for the runtime would be omega(N). This would occur if, with an array of length N, there was only one key and one ball. In that case the outer loop would run only once, and we would then just have to iterate through the array (N).

Section 2.4: Solutions

1. 11
2. 14
3. 7
4. 14
5. 18
6. 1
7. 15
8. 8

Section 3: Algorithm 2 -- Dynamic Programming

Section 3.1: Pseudocode

```
int k[M] = [k1, k2, k3, ..., kM];        //given
int b[T] = [b1, b2, b3, ..., bT];        //given
locker L[N];              //locker struct has int key, and int ball
int C[M][M];

for i=0; i<N; i++ {
    init ball and key to 0 at L[i]
}

for i=0; i<M; i++ {
    set key to 1 on the locker indirectly referenced by k[i]
}

for i=0; i<T; i++ {
    set ball to 1 on the locker indirectly referenced by b[i]
}

for i=0; i<N; i++ {
    if L[i].key == 1 {
        Copy i into the list of keys so that they are sorted
    }
    if L[i].ball == 1 {
        Copy i into the list of balls so that they are sorted
    }
}

for i=0; i<M; i++ {
    C[0, i] = C[0][i-1] + the distance from the largest ball before  the k[j-1]-th locker and the largest
                                                            ball before the k[j]-th locker
    C[i, 0] = k[i] - k[0] + 1;
}

for i=1; i<M; i++ {
    for j=1; j<M; j++ {
        C[i][j] = min(
                C[i-1][j-1] + distance from the ball closest to the k[i-1]-th locker that is still before the
                                                            k[j]-th locker,
                C[i-1][j] + number of lockers opened to get from the largest ball before the k[i-1]-th
                                                            locker and the largest ball before the k[i]-th locker,
                C[i][j-1] + the distance from the largest ball before  the k[j-1]-th locker and the largest
                                                            ball before the k[j]-th locker
                )
    }
}

if the last element in array of balls has a ball in a locker after the last key, add last ball position-last key position

return the count
```

Section 3.2: Explanation

This algorithm fills in an MxM dynamic programming table. For some spot (i, j) in the table, the algorithm chooses the lowest cost of the following three options:

1. cost at (i-1, j-1) + the distance from the current key to the first ball between the k[i-1]-th and the k[j]-th locker
2. cost at (i-1, j) + the distance between the largest ball before the k[i-1]-th locker and the largest ball before the k[i]-th locker

3. cost at (i, j-1) + the distance between the largest ball before the k[j-1]-th locker and the largest ball before the k[j]-th locker

Once the whole table has been traversed, the smallest solution for the area between the first key and the last key will be in the bottom right corner of the table. If there are balls after the last key, we should add the distance from the last key to the first ball to the value.

Section 3.3: Runtime

The runtimes for this algorithm are O(NM^2) and omega(M^2). In both cases, the M^2 comes from filling out an M by M grid, so we have two nested loops of length M. The N in the Big-O notation comes from iterating through each element in the array of lockers in the M^2 loops. In the best case, we would not have to loop through the array of lockers, and would get the value that we needed at the first time.

Section 3.4: Solutions

1. 21
2. 23
3. 64
4. 32
5. 107
6. 31
7. 87
8. 83

Section 5: Conclusion

The first algorithm will test every single possibility of key combinations that can be used. Once it has used every combination it will decide what was the fastest way. This of course leads to a longer run time. Every key has exactly two ways that it can go, either to the left or to the right. How far to the left or to the right it needs to go is determined by the other keys. Due to finding all the solutions this is by no means a quick algorithm, but it would find the answer in the end.

The second algorithm involves taking the shortest path over the longer path whenever possible.

The runtime's of the algorithms are similar, but the runtime of the second algorithm is far superior with a runtime of O(NM^2) whereas the first algorithm has O(N2^M)