

An-Najah National University



Dynamic Programming Project – Baymax

Student Name: Yazan Habash

Registration Number: 11819431

Introduction and overview

The problem we're dealing with says that we have a robot called Baymax that can go everywhere as a person goes. However, wireless module does not work properly, as a result, the person and the robot are connected to each other by a wired connection, each one starts his trip independently walking through his path marked by sequence of points. In the beginning of the trip, each one stands at first point on each path, they move to reach final destination point. The main goal is to figure out **what is the minimum possible wire length to use that would allow person and robot to reach destination without being cut!**

At each step, there are only three choices to complete it:

- A) Person moves to next point while Baymax stays.
- B) Person stays at current point while Baymax advances by one step forward.
- C) Both of them moves one step to next point.

First Step on The Road – Inspiration

One random way for solving, is to figure out all possible scenarios and take the best one with least wire length that worked in any explored scenarios, this way is called **Brute-Force** approach. Solving the problem using this technique is very bad regarding with performance because it is very cost if the size of the problem is sufficiently large.

Another Approach – Breaking Down the Problem

What about thinking of dividing the problem into smaller sub-problems and trying to solve each independently and combine them to get final desired result?

Creating the function and define its parameters

Let's create a function called **minLength** that returns the value which we want to optimize:

int minLength(Position[] A, Position[] B, int[] lengths, int i, int j)

where *i* is the **index number** of accessed point by **person** on the path (A) from beginning and *j* is the **index number** of accessed point by **robot** on the path (B) from

beginning. The function **minLength** returns the minimum length of a connected wire between them.

For example, if we called that method like this:

minLength(A, B, lengths, 4, 5)

This means that 'What is the minimum wire length would be used to connect the person and robot when the person reached to the 4th point from starting point in sequence A and the robot reached to the 5th point from the beginning in sequence B?'

Drawing the recursion tree

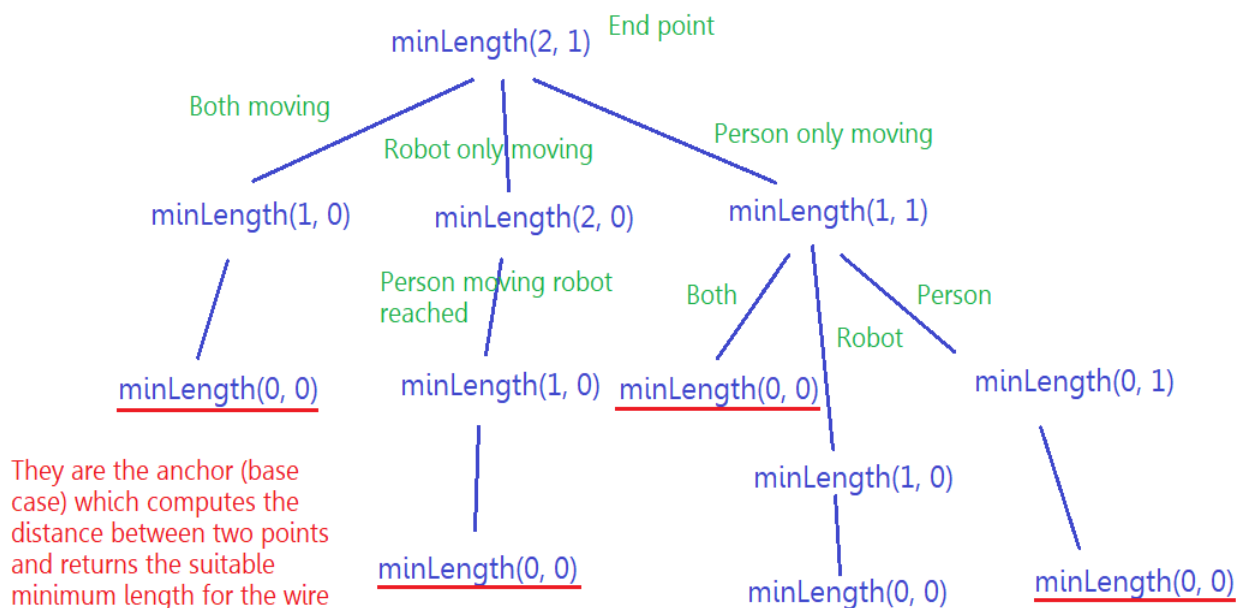
We are going to demonstrate the recursion tree for the function using the values described in the question as requested.

Let the sequence **A** = { (0, 1), (1, 1), (1, 2) } and sequence **B** = { (0, 0), (1, 0) }, where A is related to the person and B is related to the robot.

Substituting in the function, gives:

int minLen = minLength(2, 1);

The first parameter = 2 indicates that the person reaches to the point at index 2 in sequence A which is (1, 2) and the second parameter = 1 indicates that the robot reaches to the point at index 1 in sequence B which is (1, 0).



We can see that the function *minLength* can express the size and uniqueness of each sub-problem!

Divide and conquer code (Java Code):

```
private static int minLength(Position [] A, Position [] B, int [] lengths, int i, int j) {
    int sub_sol = MINUS_INFINITY;
    int sub_solA, sub_solB, sub_solC;
    if(i > 0 && j > 0) {
        sub_solA = minLength(A, B, lengths, i-1, j-1);
        sub_solB = minLength(A, B, lengths, i, j-1);
        sub_solC = minLength(A, B, lengths, i-1, j);
        int [] arr = new int[3];
        arr[0] = sub_solA;
        arr[1] = sub_solB;
        arr[2] = sub_solC;
        sort(arr);
        if(arr[2] == -1)
            sub_sol = -1; // All elements are (-1)
        else {
            for(int q=0; q<arr.length; ++q)
                if(arr[q] != -1) {
                    sub_sol = arr[q];
                    break;
                }
        }
    }
    else if(i == 0 && j > 0)
        sub_sol = minLength(A, B, lengths, 0, j-1);
    else if(j == 0 && i > 0)
        sub_sol = minLength(A, B, lengths, i-1, 0);

    double dist = distance(A[i], B[j]);
    //System.out.println("dist = " + dist);

    int len = -1;
    for(int r = 0; r < lengths.length; ++r) {
        //System.out.println(lengths[r]);
        if(lengths[r] >= dist) {
            len = lengths[r];
            break;
        }
    }

    //System.out.println("Suitable length = " + len);
    //System.out.println("Sub-Sol = " + sub_sol);

    if(len != -1 && sub_sol != -1)
        return max(len, sub_sol); // Use a taller wire
    return -1;
}
```

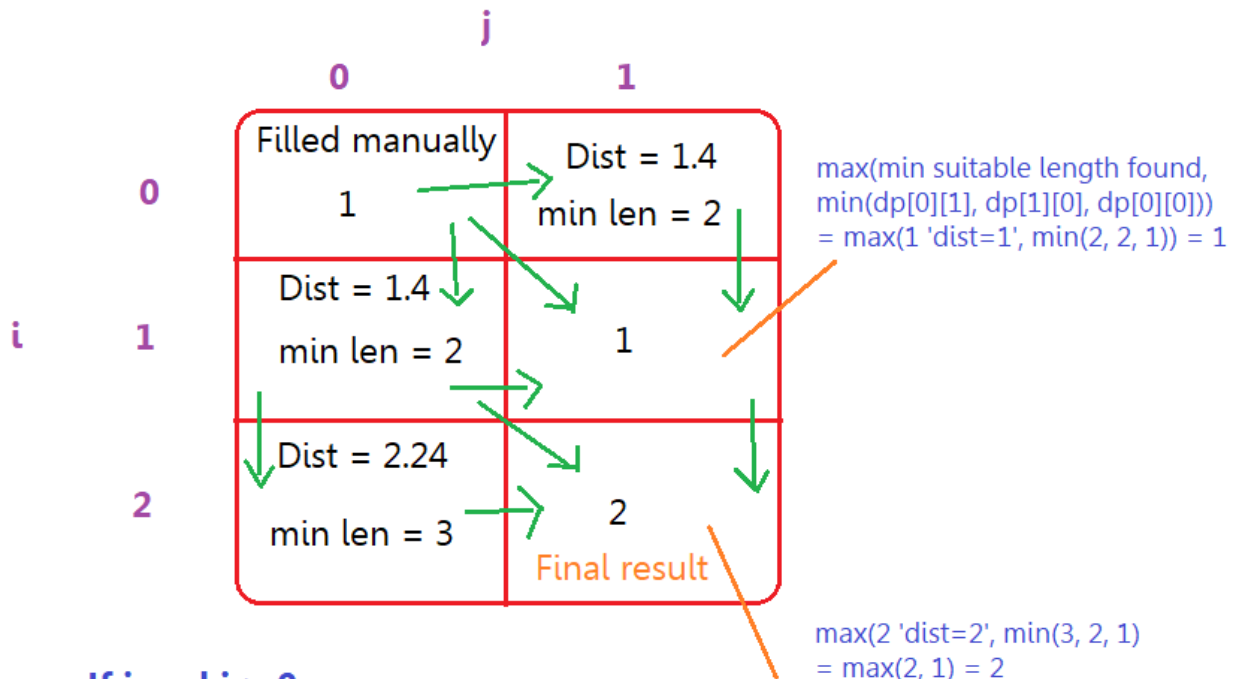
Improving The Solution – Achievement

After using **Divide & Conquer** technique, the main problem can be broken down into subproblems. So, optimal solution can be constructed from optimal solutions of its subproblems. However, the large obstacle to improve the performance is that the recursive algorithm solves the same sub-problems over and over rather than always generating new subproblems. As a result, The **Dynamic Programming** approach can be the best choice!

How do we start using DP?

Any DP problem can be solved by following these steps:

1. Draw the table which we will store the solution of each subproblems and determine the dependencies between the table cells. The dimension of the table is $i \times j$ in our case 3×2 , where 3 is number of points in sequence A and 2 is number of points in sequence B. We fill the first index $dp[0][0]$ by min length suitable for distance between first point in seq. A and corresponding first point in seq. B.



If i and j > 0:

In general at $dp[i][j] = \max(\text{min suitable length found in array according to distance}, \min(dp[i-1][j], dp[i][j-1], dp[i-1][j-1]))$

In other words, we choose the best scenario between previous subproblems, if it is fine, take it. Else, choose another taller wire suitable according to distance.

2. Determine the direction of the table. As we can see, the table can be filled by iterating row by row and take the minimum (best) solution between three previous subproblems and compare it with min wire length obtained according to computed distance. For example, If the distance was 3 and the best scenario gives min length = 2 (dist. > best scenario), then we should choose another taller wire to guarantee the wire would not be cut. So, we should choose 3 as min length. Else, we stay on choice min length = 2.
3. Write the code.

The following code fills the two tables, table **dp** which stores min lengths of each subproblem, and the second table **decision** stores the decision (How were we able to access the optimal solution).

```
private static int minLengthDP(Position [] A, Position [] B, int [] lengths, int i, int j) {
    int sub_sol = MINUS_INFINITY;

    int [][] dp = new int [i+1][j+1];
    decision = new int [i+1][j+1];

    int len = -1;
    double dist = distance(A[0], B[0]);
    for(int r = 0; r < lengths.length; ++r)
        if(lengths[r] >= dist) {
            len = lengths[r];
            break;
        }
    //System.out.println("Suitable length = " + len);

    if(len != -1 && sub_sol != -1)
        dp[0][0] = max(len, sub_sol);
    else
        dp[0][0] = -1;
    decision[0][0] = -1;
    //System.out.println("first element = " + dp[0][0]);

    // After fill first index dp[0][0], then fill the table

    int sub_solA, sub_solB, sub_solC;
    for(int s=0; s<=i; ++s)
        for(int t=0; t<=j; ++t) {
            if(s == 0 && t == 0)
                continue;
            if(s == 0 && t > 0) {
                sub_sol = dp[0][t-1];
                decision[0][t] = 1;        // From left
            }
        }
}
```

```

else if(t == 0 && s > 0) {
    sub_sol = dp[s-1][0];
    decision[s][0] = 3;    // From above
}

else if(s > 0 && t > 0) {
    sub_solA = dp[s-1][t-1];
    sub_solB = dp[s][t-1];
    sub_solC = dp[s-1][t];
    int [] val = new int[3];
    int [] sol = new int[3];
    val[0] = sub_solA;
    val[1] = sub_solB;
    val[2] = sub_solC;
    sol[0] = 2;
    sol[1] = 1;
    sol[2] = 3;
    sortMap(val, sol);
    if(val[2] == -1)
        sub_sol = -1;    // All elements are (-1)
    else {
        int q;
        for(q=0; q<val.length; ++q)
            if(val[q] != -1) {
                sub_sol = val[q];
                break;
            }
        decision[s][t] = sol[q];
    }
}

dist = distance(A[s], B[t]);
len = -1;
for(int r = 0; r < lengths.length; ++r)
    if(lengths[r] >= dist) {
        len = lengths[r];
        break;
    }
//System.out.println("Suitable length = " + len);
//System.out.println("Sub-Sol = " + sub_sol);

if(len != -1 && sub_sol != -1) {
    dp[s][t] = max(len, sub_sol);
    //System.out.println("Saved value dp[" + s + "][" + t + "] = " + dp[s][t]);
}
else {
    dp[s][t] = -1;
    decision[s][t] = -1;
    //System.out.println("Saved value dp[" + s + "][" + t + "] = " + dp[s][t]);
}
}

return dp[i][j];
}

```

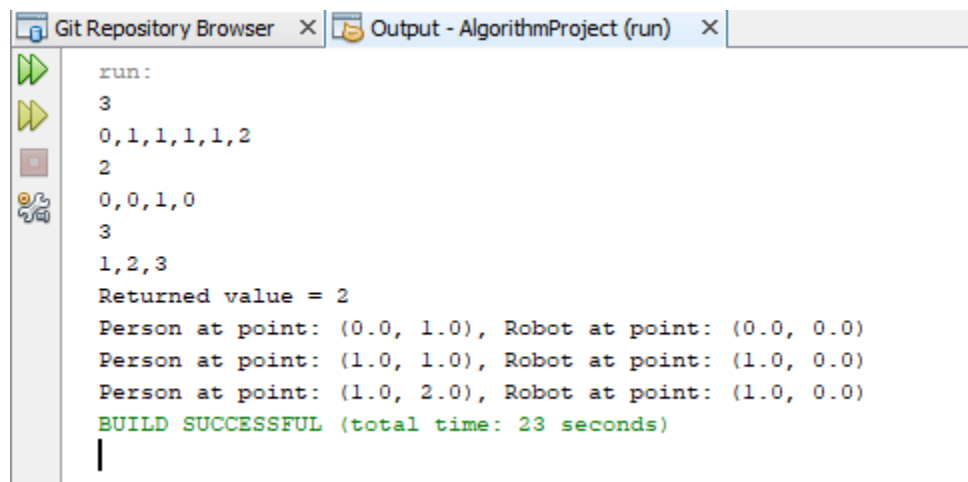
How to obtain optimal solution?

The function in the previous code fills two tables **dp** and **decision** with data. We usually store in the second table the direction of the best and minimal solution, there are four possibilities **-1, 1, 2** and **3**. For example, if **decision[2][1] = -1** that means 'it is impossible to connect person and robot by any wire from given lengths, so, there are not any path to reach this subproblem'. If **decision[2][1] = 1**, this means that we adopted the previous left cell which is **decision[2][0]**. If **decision[2][1] = 2** we adopted **decision[1][0]**. Else, we adopted the above cell **decision[1][1]**.

1 = \leftarrow , 2 = \nwarrow and 3 = \uparrow . So, we store directions that we depend to reach optimal solution. The following code prints the sequence of moves that go us the solution.

```
private static void printPath(Position [] A, Position [] B, int m, int n) {
    int start = decision[m][n];
    if(start == 2) // We will move to decision[m-1][n-1]
        printPath(A, B, m-1, n-1);
    else if(start == 1) // Go to decision[m][n-1]
        printPath(A, B, m, n-1);
    else if(start == 3) // Go to decision[m-1][n]
        printPath(A, B, m-1, n);
    System.out.println("Person at point: " + A[m].toString() + ", Robot at point: " + B[n].toString());
}
```

Output from NetBeans IDE of the example given in the question:



```
run:
3
0,1,1,1,1,2
2
0,0,1,0
3
1,2,3
Returned value = 2
Person at point: (0.0, 1.0), Robot at point: (0.0, 0.0)
Person at point: (1.0, 1.0), Robot at point: (1.0, 0.0)
Person at point: (1.0, 2.0), Robot at point: (1.0, 0.0)
BUILD SUCCESSFUL (total time: 23 seconds)
```