Contents

Introduction

We want to help Iron Man gather energy cells to defeat Doctor Doom. The problem constraints specify that:

- Iron Man must visit a series of **N energy cells** located on a 2D Cartesian plane.
- **Each energy cell** provides a specific amount of energy but traveling between them decreases his energy based on the distance between the cells.
- Iron Man **must start** at the first energy cell and **end** at the final energy cell.
- **The goal** is to maximize the remaining energy Iron Man has when he arrives at the last cell.

For example, consider this input where the lines represent the coordinates of each energy cell and the amount of energy gained and with the help of distance between two points formula:

Sample Input 0

```
3
1 2 2
4 2 3
5 5 8
```

$$d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

Starting at cell (1,2) we have 2 energy, and the distance between the first cell and the second cell (4,2) is 3 and we get 3 energy, so now we have a total of 3 energy, then we move from the second cell (4,2) to third cell (5,5) at a cost of 3.1623 and get 8 energy…
So: 2 (current energy) + 8 (third cell energy) – 3.1623 (cost) = 6.8377

Unfortunately, we can't use a greedy algorithm because the optimal solution may require skipping some cells, which a greedy approach wouldn't consider. Also, the best overall path isn't always composed of the best individual steps, as the energy gained at later cells could outweigh short term losses.

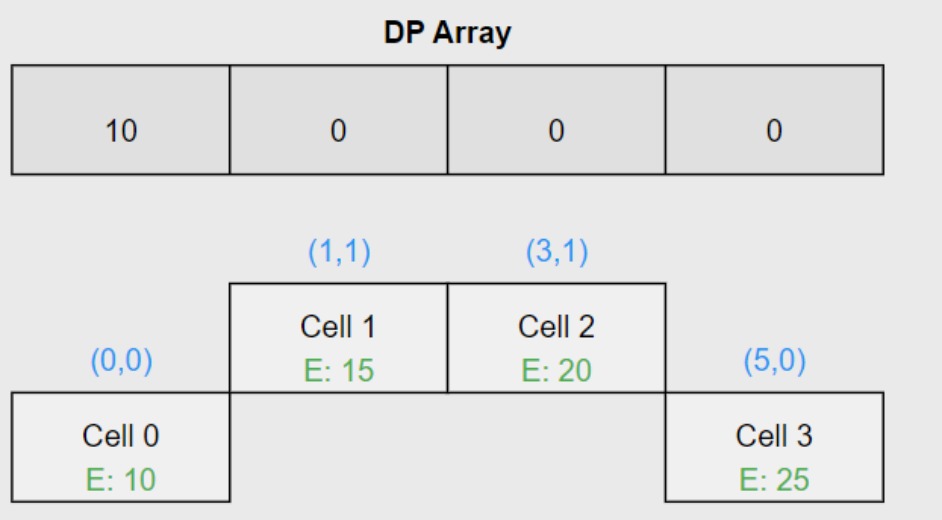
Dynamic Programming Solution

Let’s get a bit creative, and solve another example but now with visualization to make things more exciting:

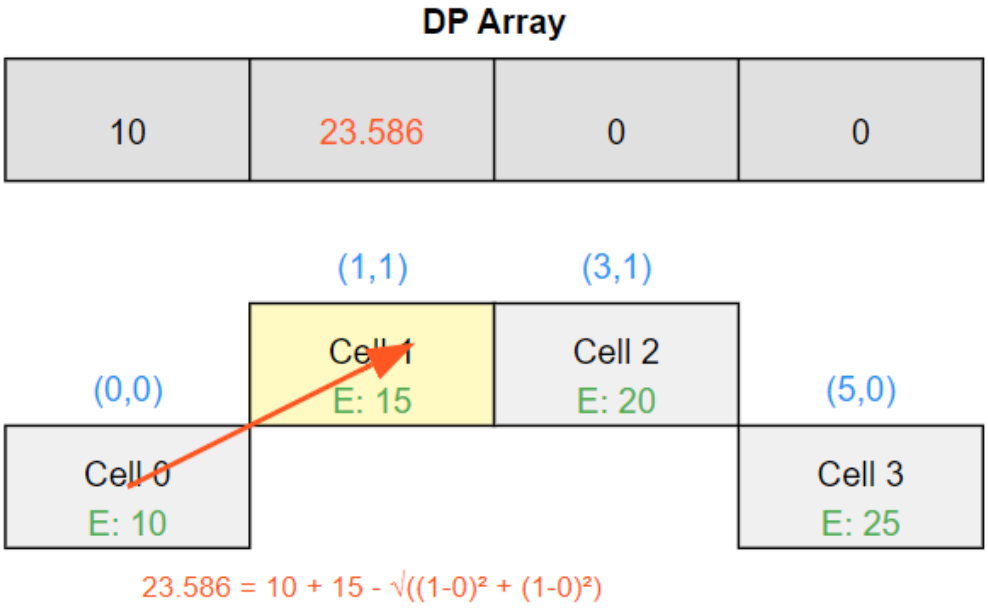
Input sample:

```
4
0 0 10
1 1 15
3 1 20
5 0 25
```

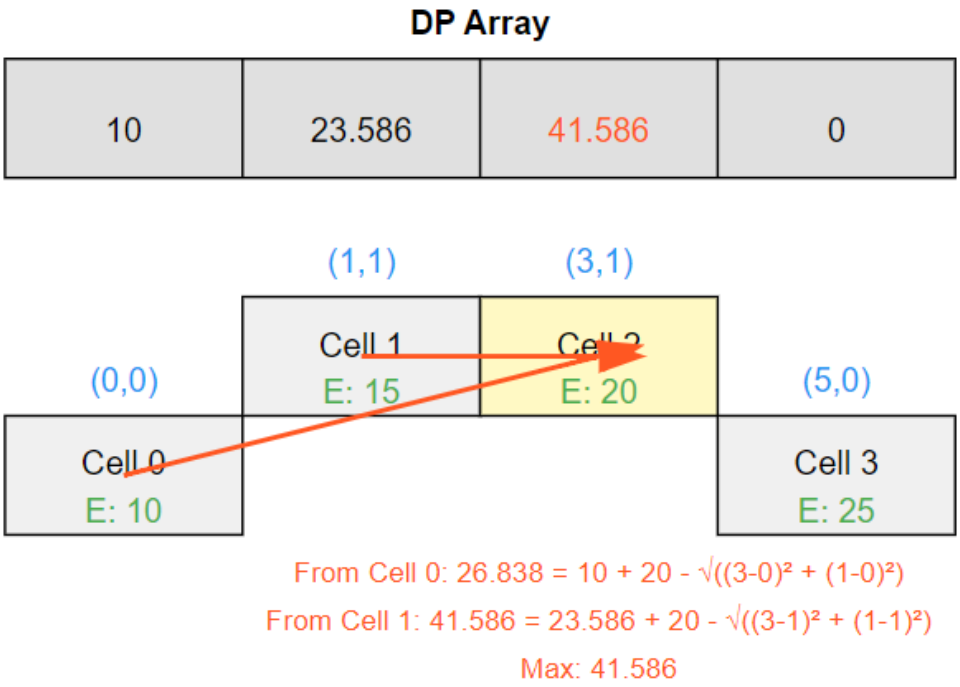
First step:



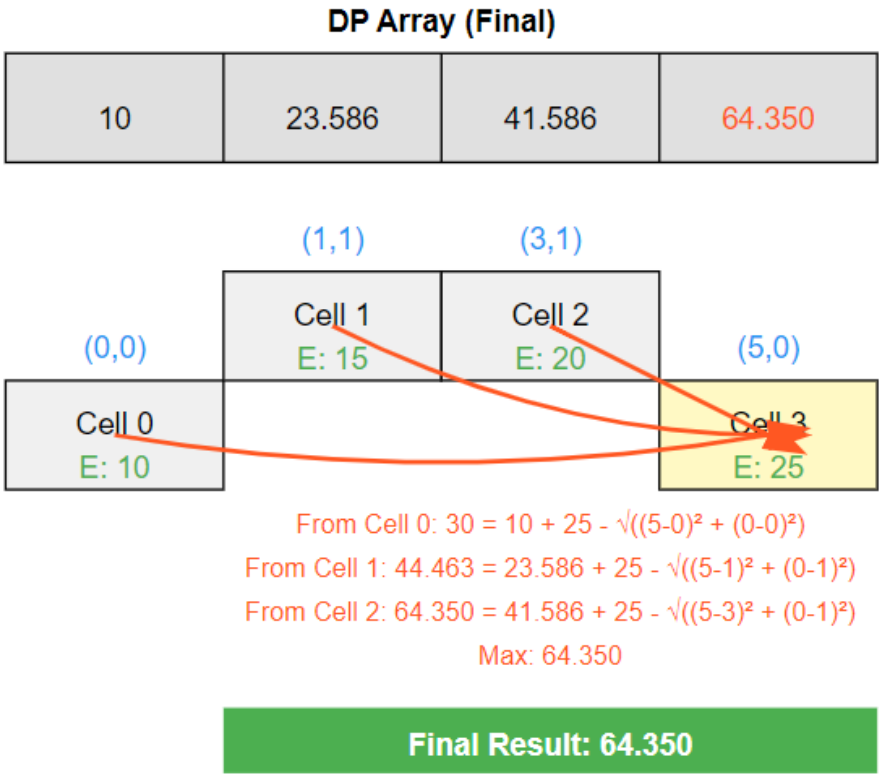
Second step:



Third step:



Fourth and final step:



Isn't that cool?

Now let's see the DP code:

```
double distance(double x1, double y1, double x2, double y2) {
    return sqrt(pow(x1 - x2, 2) + pow(y1 - y2, 2));
}
```

This is the method that calculates the distance between cells (points).

```
vector<double> x(N), y(N), energy(N);
for (int i = 0; i < N; i++) {
    cin >> x[i] >> y[i] >> energy[i];
}
```

First, we get the input from the user, using a vector to be dynamic.

```
vector<double> maxArr(N, 0);
maxArr[0] = energy[0];
```

Initialize maxArr (same as dp array in the previous example), which holds the max energy at each cell.

```
for (int i = 1; i < N; i++) {
    for (int j = 0; j < i; j++) {
        double dst = distance(x[j], y[j], x[i], y[i]);
        maxArr[i] = max(maxArr[i], maxArr[j] + energy[i] - dst);
    }
}
```

This is the core of the dynamic programming solution, to calculate the maxArr values and get the max possible energy, the previous example with visualization explains this process perfectly.

```
printf("%.4f", maxArr[N-1]);
```

Finally, we print the final value.

Here is the full code:

```cpp
// Qais Hweidi - 12029355

#include <iostream>
#include <vector>
#include <cmath>
#include <algorithm>

using namespace std;

double distance(double x1, double y1, double x2, double y2) {
    return sqrt(pow(x1 - x2, 2) + pow(y1 - y2, 2));
}

int main() {

    int N;
    cin >> N;

    vector<double> x(N), y(N), energy(N);
    for (int i = 0; i < N; i++) {
        cin >> x[i] >> y[i] >> energy[i];
    }

    vector<double> maxArr(N, 0);
    maxArr[0] = energy[0];

    for (int i = 1; i < N; i++) {
        for (int j = 0; j < i; j++) {
            double dst = distance(x[j], y[j], x[i], y[i]);
            maxArr[i] = max(maxArr[i], maxArr[j] + energy[i] - dst);
        }
    }

    printf("%.4f", maxArr[N-1]);

    return 0;
}
```

Iron Man called us and said that he doesn't want only the max energy he could get, he wants the path also, so we modify the previous code:

```cpp
double distance(double x1, double y1, double x2, double y2) {
    return sqrt(pow(x1 - x2, 2) + pow(y1 - y2, 2));
}

int main() {
    int N;
    cin >> N;

    vector<double> x(N), y(N), energy(N);
    for (int i = 0; i < N; i++) {
        cin >> x[i] >> y[i] >> energy[i];
    }

    vector<double> maxArr(N, 0);
    vector<int> path(N, -1);
    maxArr[0] = energy[0];

    for (int i = 1; i < N; i++) {
        for (int j = 0; j < i; j++) {
            double dst = distance(x[j], y[j], x[i], y[i]);
            if (maxArr[j] + energy[i] - dst > maxArr[i]) {
                maxArr[i] = maxArr[j] + energy[i] - dst;
                path[i] = j;
            }
        }
    }

    printf("%.4f\n", maxArr[N-1]);

    vector<int> finalPath;
    for (int i = N - 1; i != -1; i = path[i]) {
        finalPath.push_back(i);
    }
    reverse(finalPath.begin(), finalPath.end());

    cout << "Path: ";
    for (int i = 0; i < finalPath.size(); i++) {
        cout << finalPath[i];
        if (i < finalPath.size() - 1) cout << " -> ";
    }
    cout << endl;

    return 0;
}
```