# Tower of Cubes

برای بدست آوردن طول طولانی ترین برج قابل ساخت با مکعب های داده شده و شرایط مسأله ابتدا هر مکعب را به شش بلوک تبدیل میکنیم هر بلوک نشان دهنده ی یک مدل قرار دادن مکعب میباشد چون هر مکعب دارای شش وجه میباشد بنابرین به شش بلوک میتوان تبدیل کرد

و به هر بلوک رنگ وجه بالایی و پایینی که عدد میباشند و همچنین وزن آن مکعب را میدهیم و با ساختن نمونه ای از کلاس بلوک ذخیره میکنیم

سپس به کمک کلاس MergeSort که در پروژه قبل (آسمانخراش ها) نوشته بودیم این بلوک ها را بر اساس وزن و به ترتیب نزولی مرتب میکنیم

لیست این بلوک های مرتب شده را به کلاس گراف میدهیم و با صدا زدن متد ساخت گراف یال هایی بین این n * 6 رای میکشیم به گونه ای که جهت دار میباشند و اگر از a به b یال داریم به این معنی می باشد که وزن a کمتر از b است و رنگ وجه بالای a برابر رنگ وجه زیر b میباشد و aوb متعلق به یک مکعب نیستند

به عبارت دیگه میتوان با چرخاندن آن دو مکعب بلوک a و b را ساخت و آنها را روی هم قرار داد

گراف تشکیل شده یک گراف جهت دار بدون دور میباشد چون لیست بر اساس وزن نزولی چیده شده و یال ها فقط از سمت چپ لیست به سمت راست کشیده می شوند پس نمیتوان دور داشت

پس از ساختن گراف با روش برنامه نویسی پویا سعی میکنیم طولانی ترین مسیر ممکن در این گراف را بیابیم با یافتن این مسیر و شماردن و حساب کردن راس های موجود در این مسیر میتوان این مسیر مدل را به ساخت طولانی ترین برج تبدیل کرد.

برای برنامه نویسی پویا مقدار طولانی ترین مسیر موجود شروع شده از راس i را برابر dp[i] میگذاریم و همه dp هارا حساب کرده و بزرگترین آنها را میگیریم و برج را میسازیم

برای بدست آوردن dp ها (مخفف برنامه نویسی پویا ☺ ) از فرمول زیر استفاده میکنیم:

**dp[i] = MAX{ 1 + dp[k] } (for all childs of i)  k is a child Node**

طولانی ترین مسیر i برابر طولانی ترین مسیر فرزندان به اضافه یک مسیر از i به آن فرزند

برای کامل کردنdp و بدست آوردن جواب مسأله الگوریتم dfs را اجرا میکنیم و با جستجوی عمقی راس هایی انتهایی را مقدار دهی اولیه یعنی صفر میدهیم و برای هر راس دیگر dfs فرزندانش را صدا میکنیم اگر فرزندی در مراحل قبلی ملاقات شده بود dp آن حساب شده است درغیر این صورت dfs آنرا صدا میزنیم تا dp فرزندانش سپس خودش حساب شود

و در انتها هم بیشترین مقدار dp فرزندان به اضافه یک را به آن راس میدهیم

بنابر این در این روش لازم نیست تمام مسیر ها چک شوند فقط با پیمایش راس ها به کمک جستجوی عمقی dp هر راس یک بار حساب شده و برای راس های والدین فقط مقایسه بین اینها انجام میشود (استفاده از روش برنامه نویسی پویا)

در کنار بدست آوردن dp هر راس مقداری به نام bestChild برای هر راس محاسبه می شود و در همان مرحله انتخاب بیشترین مقدار dp فرزندان نام آن فرزند هم در bestChild ذخیره میشود که بتوان بعدا بلند ترین مسیر را علاقه بر مقدارش بدست آورد.

**کد :**

برنامه دارای ۴ فایل می باشد

کلاس Block که در آن بلوک ها ساخته می شود و مقادیر گفته شده در آن ذخیره می شود

کلاس MergeSort که با دادن لیست بلوک ها آنها را مرتب میکند (نزولی)

کلاس Graph که گراف جهت دار بدون دور و الگوریتم dfs را اجرا میکند

و کلاس Main که گرافیک و خروجی ورودی برنامه را شامل می شود.

کلاس Main :

در متد run ابتدا تعداد مکعب ها گرفته می شود

سپس برای هر مکعب هفت مقدار صحیح اولی برابر وزن مکعب و شش مقدار بعدی رنگ های آن گرفته می شود (روبه رو پشت چپ راست بالا و پایین)

و شش بلوک برای هر مکعب ساخته می شود و در لیست blocks ذخیره می شود

```
36      public static void run(Group group) {
37          Scanner input = new Scanner(System.in);
38
39          int n = input.nextInt();
40
41          ArrayList<Block> blocks = new ArrayList<>();
42          for (int i = 0; i < n; i++) {
43              int weight = input.nextInt();
44              int fr = input.nextInt();
45              int ba = input.nextInt();
46              int le = input.nextInt();
47              int ri = input.nextInt();
48              int to = input.nextInt();
49              int bo = input.nextInt();
50              blocks.add(new Block(weight, fr, ba));
51              blocks.add(new Block(weight, le, ri));
52              blocks.add(new Block(weight, to, bo));
53              blocks.add(new Block(weight, ba, fr));
54              blocks.add(new Block(weight, ri, le));
55              blocks.add(new Block(weight, bo, to));
56          }
```

سپس لیست بلوک هارا به روش ادغامی مرتب میکنیم و در لیست جدید sortedBlocks ذخیره میکنیم

و گرافی با تعداد بلوک ها می سازیم

سپس این لیست مرتب شده را به گراف میدهیم تا یال های جهت دار را بسازد.

و در انتها از نمونه گراف طولانی ترین مسیر را می گیریم و در list ذخیره میکنیم سپس مکعب های آن لیست را به ترتیب وزن نزولی نمایش میدهیم.

```
58          MergeSort mergeSort = new MergeSort(blocks);
59          ArrayList<Block> sortedBlocks = mergeSort.getBlocks();
60
61          Graph graph = new Graph(sortedBlocks.size());
62
63          graph.create(sortedBlocks);
64
65          ArrayList<Integer> list = graph.longestPath();
66          final double SIZE = Math.min((HEIGHT-100) / list.size(), 100);
67          for (int i = 0; i < list.size(); i++) {
68              StackPane sp = sortedBlocks.get(list.get(i)).getBox(SIZE);
69              sp.setLayoutY(-i * SIZE);
70              group.getChildren().add(sp);
71              System.out.println(sortedBlocks.get(list.get(i)));
72          }
```

کلاس Block :

علاوه بر اطلاعات داده شده این کلاس دارای متد getBox می باشد که یک شکل جعبه چرخش یافته را به همراه اطلاعات متد toString را برای نمایش دادن بر میگرداند

```
@Override
public String toString() {
    return  "{ Weight : " + w +
            "->  Bottom : " + bottom +
            ", Top : " + top + " }";
}

public StackPane getBox(double size){

    Box box = new Box(size, size, size);
    Rotate rxBox = new Rotate( angle: 0,  pivotX: 0,  pivotY: 0,  pivotZ: 0, Rotate.X_AXIS);
    Rotate ryBox = new Rotate( angle: 0,  pivotX: 0,  pivotY: 0,  pivotZ: 0, Rotate.Y_AXIS);
    Rotate rzBox = new Rotate( angle: 0,  pivotX: 0,  pivotY: 0,  pivotZ: 0, Rotate.Z_AXIS);
    rxBox.setAngle(30);
    ryBox.setAngle(30);
    rzBox.setAngle(0);
    box.getTransforms().addAll(rxBox, ryBox, rzBox);

    Text text = new Text(this.toString());
    StackPane stack = new StackPane();
    stack.getChildren().addAll(box, text);

    return stack;
}
```

کلاس Graph :

این گراف دارای آرایه ای از لیست یال جهت دار و تعدادی راس میباشد و با ساخت نمونه از این کلاس تعداد راس هارا که برابر تعداد بلوک ها می باشد قرار میدهیم و لیست متناسب با آن راس را مقدار دهی اولیه میکنیم.

متد addEdge (a,b) یالی را از a به سمت b تعریف میکند و در لیست a مقدار b را به این معنی وارد میکند.

```
class Graph {

    int vertices;
    ArrayList[] edge;

    Graph(int vertices) {
        this.vertices = vertices;
        edge = new ArrayList[vertices];
        for (int i = 0; i < vertices; i++) {
            edge[i] = new ArrayList<>();
        }
    }

    void addEdge(int a, int b) { edge[a].add(b); }
```

متد longestPath طولانی ترین مسیر را پیدا میکند و در لیستی بر میگرداند

دو آرایه dp و bestChild را برای هر راس تعریف میکند (در بالا تعریف شده اند)

و تمام مقادیر bestChild برابر منفی یک تعریف میشود تا هنگامی که از طریق بیشترین dp خواستیم برج را بسازیم به صورت لینک لیستی هر کدام بعدی را مشخص کنند تا وقتی به مقدار منفی میرسد جلو برودسپس آرایه ای برای نشان دادن ملاقت شده یا نشده راس ها به اسم visited در نظر گرفته می شود و به متد dfs داده می شود

سپس برای تمام راس هایی که ملاقات نشده اند dfs را صدا میزنیم

در انتهای این کار نیز مقدار dpها تکمیل شده و بشترین مقدار آنها را میابیم

و start را برابر شماره بلوک دارای بزرگترین dp و res را هم برابر dp آن قرار میدهیم سپس به صورت لیست پیوندی از start پیش می رویم و bestChild هر راس را صدا میزنیم و در map ذخیره میکنیم تا وقتی به منفی ۱ برسیم یعنی ته بلند ترین مسیر و mapرا بر میگردانیم تا بلوک ها چاپ شوند

```java
ArrayList<Integer> longestPath() {
    int n = vertices;
    ArrayList[] neighbours = edge;
    ArrayList<Integer> map = new ArrayList<>();

    int[] dp = new int[n];
    int[] bestChild = new int[n];

    for (int i = 0; i < n; i++) {
        bestChild[i] = -1;
    }

    boolean[] visited = new boolean[n];

    for (int i = 0; i < n; i++) {
        if (!visited[i])
            dfs(i, neighbours, dp, visited, bestChild);
    }

    int res = 0;
    int start = 0;
    for (int i = 0; i < n; i++) {
        if (dp[i] > res) {
            res = dp[i];
            start = i;
        }
    }
    while (start != -1) {
        map.add(start);
        start = bestChild[start];
    }

    return map;
}
```

متد dfs :

این متد شماره یک راس و لیست مجاور های آن راس یعنی راس هایی که به آنها یال جهت دار دارد به همراه آرایه dp و bestChild و visited را میگیرد

و برای تمام فرزندان راس گرفته شده اگر در قبل ملاقات شده بودند یعنی مقدار dp آنها وجود دارد پس اگر از مقدار حاضر بزرگتر هستند dp این راس برابر dp آن فرزند + ۱ می شود و آن فرزند به عنوان bestChild انتخاب می شود اگر برای تمام فرزندان این کار را بکنیم مقدار

**dp[i] = MAX{ 1 + dp[k] } (for all childs of i)  k is a child Node**

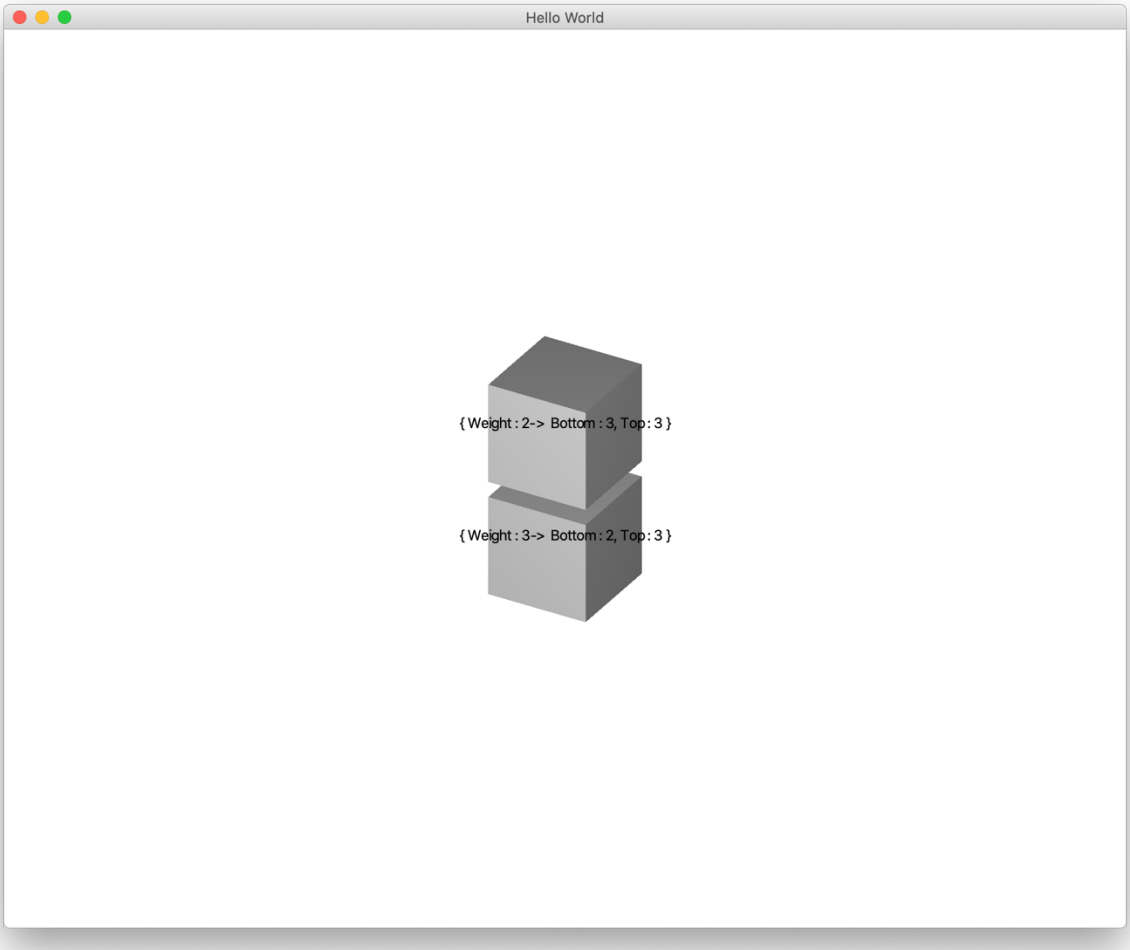
به درستی انتخاب می شود

و اگر راس فرزندش ملاقات نشده باشد dfs آن راس صدا زده می شود تا dp آن فرزند به صورت مشابه محاسبه شود.

```
22 @    void dfs(int node, ArrayList<Integer>[] adj, int[] dp,
23              boolean[] visited, int[] bestChild) {
24          visited[node] = true;
25
26          for (int i = 0; i < adj[node].size(); i++) {
27              if (!visited[adj[node].get(i)])
28                  dfs(adj[node].get(i), adj, dp, visited, bestChild);
29              if (dp[node] < 1 + dp[adj[node].get(i)]) {
30                  dp[node] = 1 + dp[adj[node].get(i)];
31                  bestChild[node] = adj[node].get(i);
32              }
33          }
34      }
```

اجرا :

```
3
1
1 2 2 2 1 2
2
3 3 3 3 3 3
3
3 2 1 1 1 1
{ Weight : 3->  Bottom : 2, Top : 3 }
{ Weight : 2->  Bottom : 3, Top : 3 }
```



```
10
1
1 5 10 3 6 5
2
2 6 7 3 6 9
3
5 7 3 2 1 9
4
1 3 3 5 8 10
5
6 6 2 2 4 4
6
1 2 3 4 5 6
7
10 9 8 7 6 5
8
6 1 2 3 4 7
9
1 2 3 3 2 1
10
3 2 1 1 2 3
{ Weight : 10->  Bottom : 2, Top : 3 }
{ Weight : 9->  Bottom : 3, Top : 3 }
{ Weight : 8->  Bottom : 3, Top : 2 }
{ Weight : 6->  Bottom : 2, Top : 1 }
{ Weight : 4->  Bottom : 1, Top : 3 }
{ Weight : 3->  Bottom : 3, Top : 2 }
{ Weight : 2->  Bottom : 2, Top : 6 }
{ Weight : 1->  Bottom : 6, Top : 5 }
```