# Graphs

## Prim’s algorithm

Prim’s algorithm used for optimization problems, this is **Greedy** approach implementation used to find minimum spanning tree in a graph in a weighted graph.

If a graph has **V** vertices and **E** edges. The minimum spanning tree will have **V-1** edges where total cost will be minimum

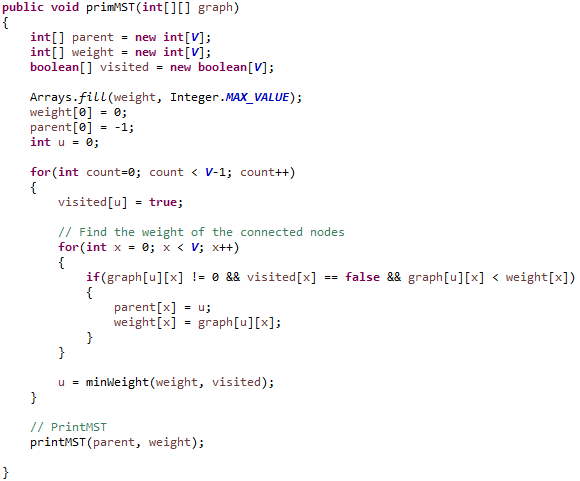
* Select a vertices to start with
* Find all the nodes weight connected with the vertices & update the parent
* Find the minimum cost node from the collected in step-2 & mark it visited
* Repeat step-2 and 3 for V-1 iterations

### Implementation

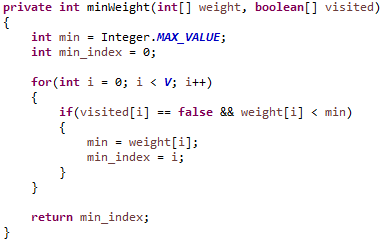
1. Create 3 arrays with size V. parent, weight, visited
2. Set all the weights to Integer.MAX\_VALUE
3. Set weight [0] = 0 and parent [0] = -1 and u=0 which will represent the minimum weight node from all the listed weights**.**
4. Loop through till V-1 times
   1. Set visited[u] = true
   2. Loop for all the vertices
      1. Find the weight of connected nodes which are

* Connected to u
* Not visited
* Weight from current node < stored weight in **weight** array (may be set from other nodes)
  + 1. For match update parent and weight
  1. Update “u” i.e. set minimum weight vertices

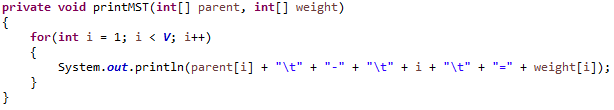
### Program



**To find next minimum weight:**



**Print the MST:**



### Other notes

Prim’s algorithm can’t work for a graph where some of the nodes are not connected.

Cycle formation won’t happen through prim’s algorithm method, which is good.

Time complexity in worst case **O (n2)**

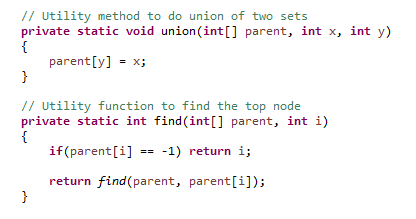
## Disjoin set

Disjoin set is a **union** **find** algorithm which is used to find cycles in Graph and heavily in **Kruskal’s** algorithm.

If an edge connecting vertices found in 2 different sets, it will union the sets to a single set. Which can be represented in form of a tree represented in array by setting parent of the vertices.

The vertices which **rank** will be considered as parent for the other vertices while making **union** in **Path compression** technique.

### Implementation



1. Create a **parent** array and fill with -1 (which says every node is their own parent)
2. Loop through all **Edges**
   1. Find parent of **src**  and **dest**
   2. If both are same then there is a **cycle**
   3. If no then make union, update the parent of one node as other, because they are connected

### Other notes

Time complexity worst case **O (n)**

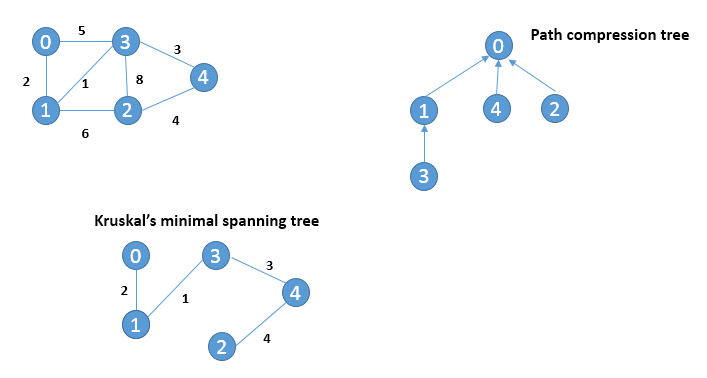
## Kruskal’s algorithm

Kruskal’s algorithm used to find minimum spanning tree in a weighted un-directed graph. Here **Greedy** approach is followed stating with minimum weight edge.

**Disjoin** set is used to avoid cycle while forming spanning tree using Kruskal’s algorithm

Again if graph have **V** vertices then number of edges will be **V-1** in spanning tree.

### Graphical representation



### Other notes

**Path compression** technique used on Disjoin set

Time complexity worst case **O (n2).** Time complexity can be improved using **Min Heap** where for every operation time complexity is **O (log n).**

## Dijkstra algorithm

Dijkstra algorithm is used to find **single source** shortest path to all the other nodes in the graph. It works on **Greedy** approach for finding minimum distance node from current node.

This approach is called **relaxation**

If (d[u] + cost(u,v) < d[v]) then d[v] = d[u] + c(v,v)

Dijkstra algorithm works for weighted directed and un-directed graph.

### Implementation

The implementation of Dijkstra is very similar to Prim’s algorithm with a bit twist.

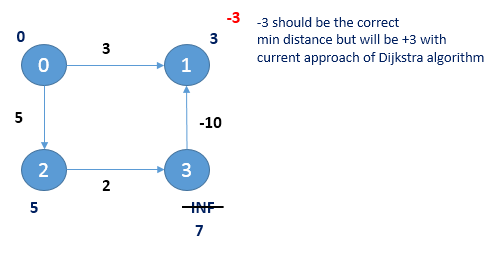
1. Create an array of distance and visited
2. Initialize all the distance value to Integer.MAX\_VALUE
3. Set distance of 0 to 0, considering 0 as the starting node.
4. Loop through all the vertices
   1. Set visited [u] to true (u- minimum weight node)
   2. Loop through all the vertices for a single “u”
      1. If there is an edge between u to v and v is not visited and distance of u + edge weight from u-v is less that current distance/weight of v
      2. Then distance[v] = distance[u] + weight(u,v)
   3. After looped through all the vertices, find the minimum weight node (i.e. u)
5. After loop over, print the solution

### Other notes

Time complexity for Dijkstra algorithm is **O (n2)** worst case.

### Where it won’t work

Dijkstra algorithm may not work properly for the graph having a **negative edge** directed graph.



# Dynamic programming

**Dynamic programming** and **Greedy method** is used for optimization problems.

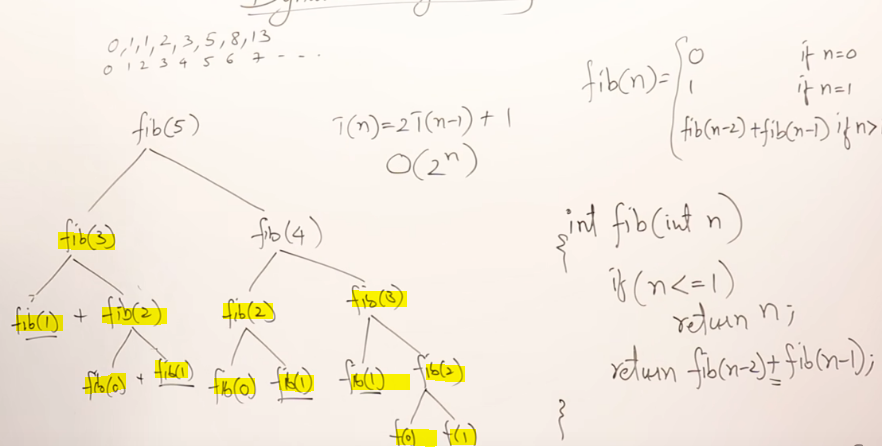
In **Greedy method** the decision is taken once and based on the decision steps would be carried out to find the optimal solution. Which also may not always guarantee the solution is optimal, but the process what got followed is optimal and based on the process that is the optimal solution.

In **Dynamic programing** the solutions will be carried out and from all the solutions the optimal solution will be picked. Here the decision will be picked on every step to approach towards an optimal solution.

The performance/time complexity of the program is improved using **Memoization** or **Tabulation**

## Memoization

Let’s take an example for finding Fibonacci series and will see what the problem was and how time complexity can be improved through Memoization.



Here if we see the highlighted once ex. **fib(3)** or **fib(2)** are computed many times and total time complexity of the program is **O (2n).**

We can improve the performance by using a global **array** where the value for pre computed fib value we will store and will use the already computed value from array rather than calling the function many times.

Using this total call will be **n+1** and hence the complexity will be **O (n+1)** i.e. **O (n)**

Memoization follows top-down approach.

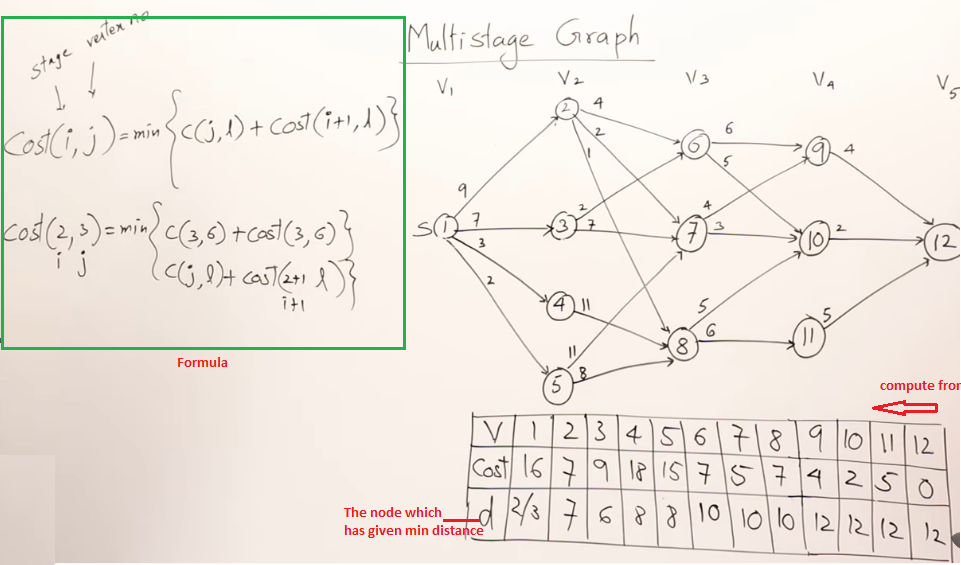
## Multistage Graph

**Multistage** graph is a directed weighted graph where the vertices are divided into stages such that the edges are connecting vertices from once stage to next stage only. First and last stage will have single vertex to represent starting and ending point of the graph.

This graph used for representing resource allocation.

We have to select a path from source to sink which gives minimum cost. As this is a minimization problem which is an optimization problem so problem can be solved using dynamic programming.

### Problem and solution

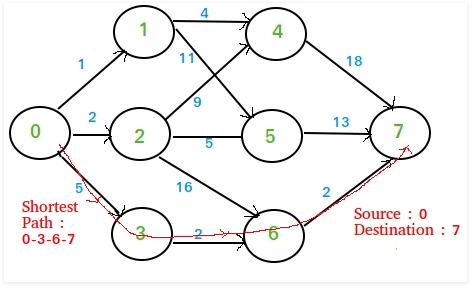


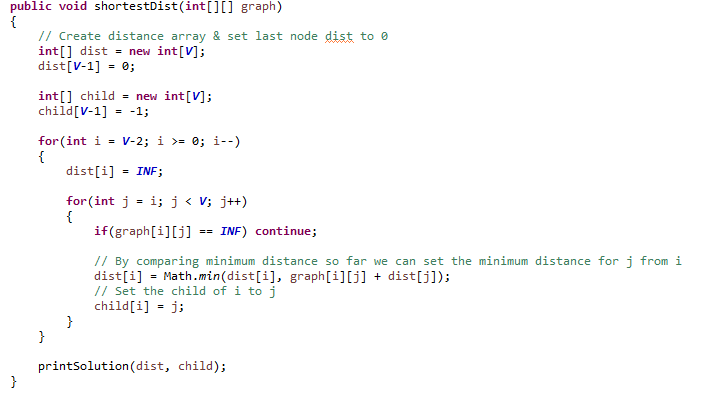
The computation happens from sink to source, but till now dynamic programming is not used. Once this part is formed we will take sequence of decisions from **top – sink** to decide the optimal path. We don’t have to compute any value and it’s already done. This method is called **Tabulation**.

While computing solution will start from **distance** array and based on it the optimal path is

1 – 2 – 7 – 10 – 12 or 1 – 3 – 6 – 10 – 12

1. Create a distance array & child array with size V
2. Initialize the last element of distance array to 0, as we are starting from last node
3. Initialize child array last element to -1, means no further to go.
4. Loop from last before vertex till 0th node
   1. Set the distance of node to INF
   2. Loop through j = i till V
   3. Update distance of ith node to min(dist[i], graph[i][j] + dist[j])
   4. Update child of I to j
5. Now distance [0] will give the minimum distance where from 0th node and child array will give position from 0th node to where to go.





## Floyd Warshall algorithm for all pair shortest path

**Floyd Warshall** algorithm is used to find all pair shortest path in a weighted directed graph.

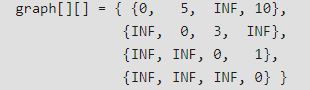
**Dijkstra** algorithm is used for single source shortest path and also can be used to find all pair shortest path by running Dijkstra on all of the nodes. But the time complexity will be **O (n2 \* n) = O (n3).**

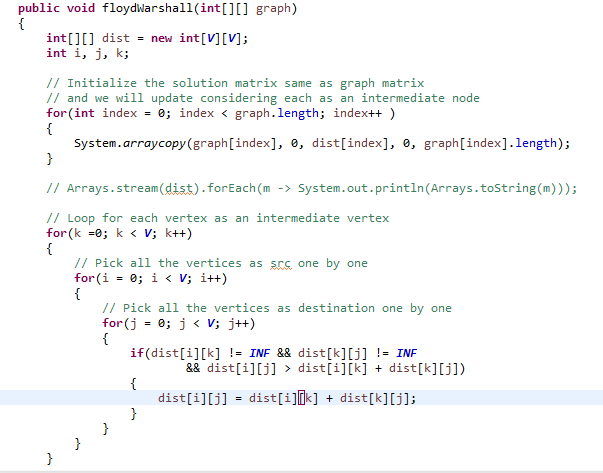
Instead we can use Floyd Warshall algorithm for finding shortest path for **all pair of vertices** using dynamic programming by considering each vertex as an intermediate node on individual step and then decision taken to find the minimum weight by considering that node as an intermediate node.

If **k** is an intermediate vertex in the shortest path from **I** to **j.** we will update the value of **dist[i][j]** when **dist[i][j] > dist[i][k] + dist[k][j]**

### Implementation

In the graph we will consider (while forming) for self-loop distance is **0** and for no edge vertices distance is **INF**





## 0/1 Knapsack problem

For given number of items **n. weight** and **profit** will be provided with a knapsack weight **M.** We need to select the items where profit will be maximized. We shouldn’t take a fraction of unit.

This is an optimization problem can be solved using through dynamic programming by taking sequence of decisions.

We need to compute all the possible arrangement and pick the best once, **Tabulation** method will be used to avoid re-computing values for overlapping sub-problems.

### Implementation

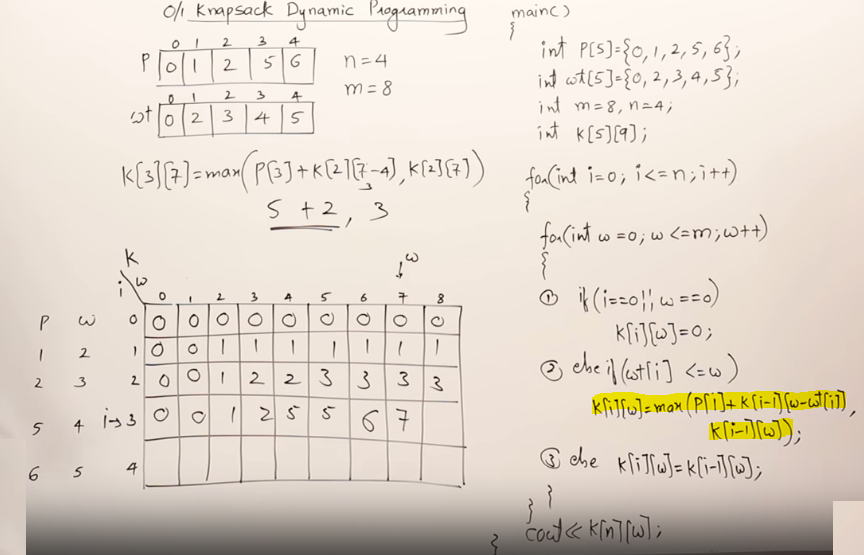
1. Loop for all the items
   1. Loop for weight ranging from 0 – M
      1. For all 0th index fill 0
      2. If weight of the item < current weight in the matrix/capacity (i.e. w)

K[i][w] = Math.max(value in the same w for last item i-1,

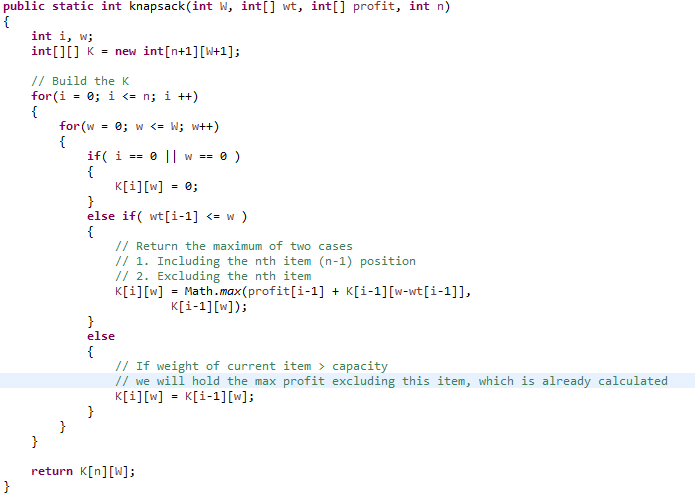
Current item profit + value of last item(i-1) with weight different of current K weight – weight of the item)

* + 1. If weight of the item > current bag weight capacity

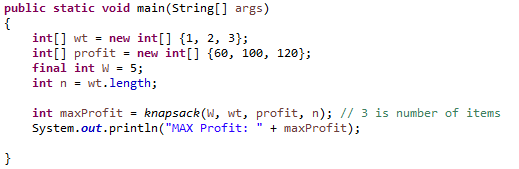
Store the previous item weight in the current node cell



### Program



**To test:**



### Complexity analysis

This is overlapping sub problem. If we solve it through **recursion** the time complexity will be **exponential** i.e. **2n**

Using **dynamic programming** we can bring down the complexity to **O (nW)**

## Coin Change Problem Number of Ways

Given a set of coins number of ways we can make change of another given value.

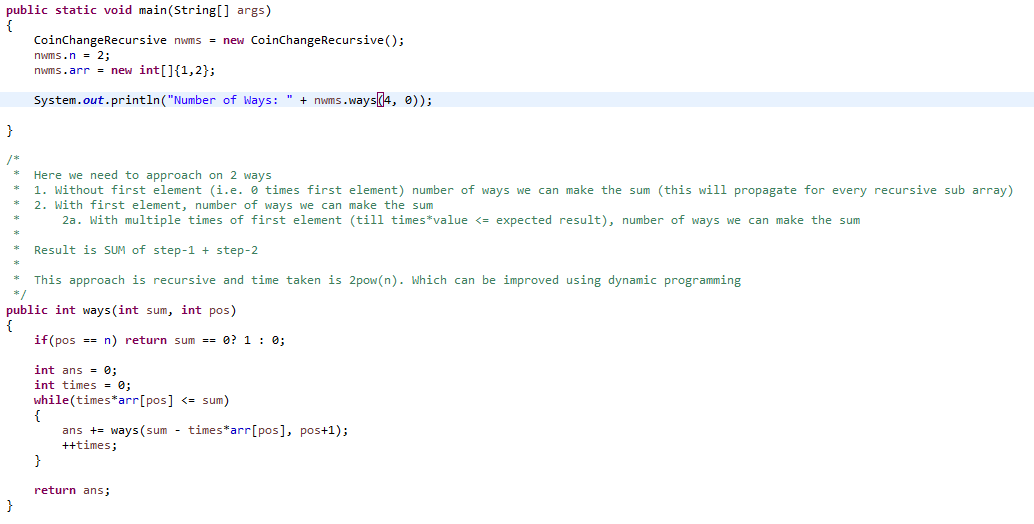
Give coins [1, 3, 5] number of ways we can make change 9

We can approach this problem in 2 ways, as in most dynamic programming case. As these are the overlapping sub problems with some state.

* Recursive approach
* Dynamic programming tabulation approach

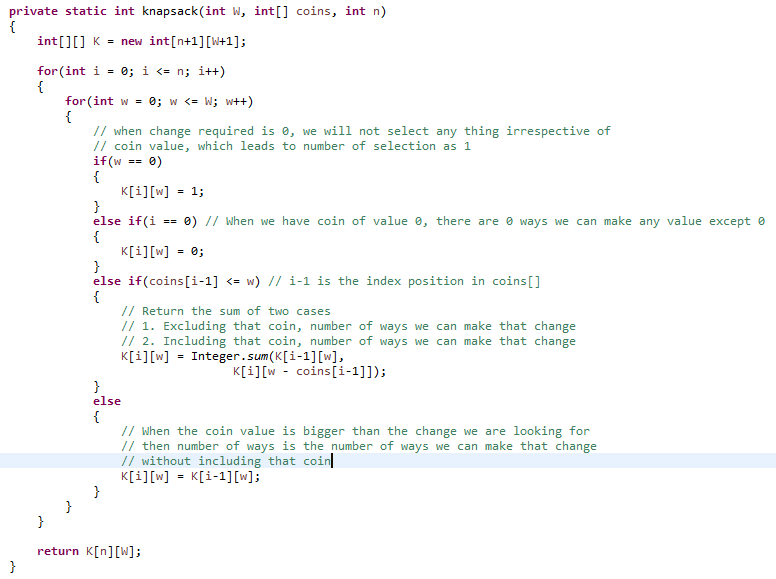
### Recursive approach

Here the time complexity is exponential **O (2n)**

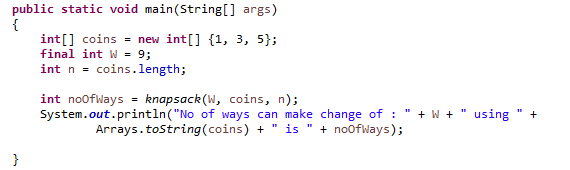


### Dynamic Programming approach

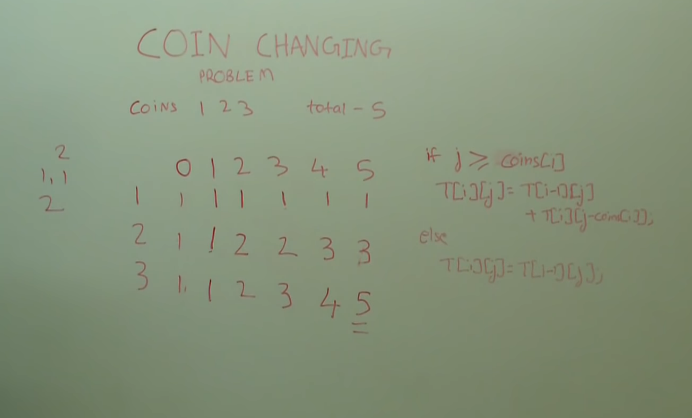
Here time complexity **O (nW)**



**To test:**



### Approach for solving



We need to make 2 considerations and **sum** them.

* Number of ways we can make the change without including the current coin
* Number of ways we can make the sum including current coin (in that case the value we need to look T[i][j – coins[j]] ). Because as we are including that coin we need to deduct that value from the value we are looking for

## Recursive staircase

Given number of steps you can make at once and how many ways you can climb given number of stairs.

Either making 1 or 2 step at once in how many ways we can climb 6 steps.

### Approach

This problem looks similar to coin change problem but in coin change problem [1, 1, 2] is same as [2, 1, 1] i.e. both says to make 4 we need two 1 and one 2.

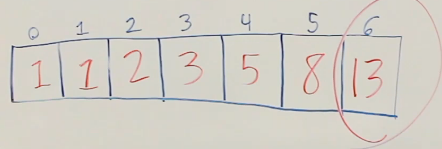
But in stair climbing case both are different. The first says we have taken 1 step at first, next 1 step and then 2 steps at one shot to reach 4.

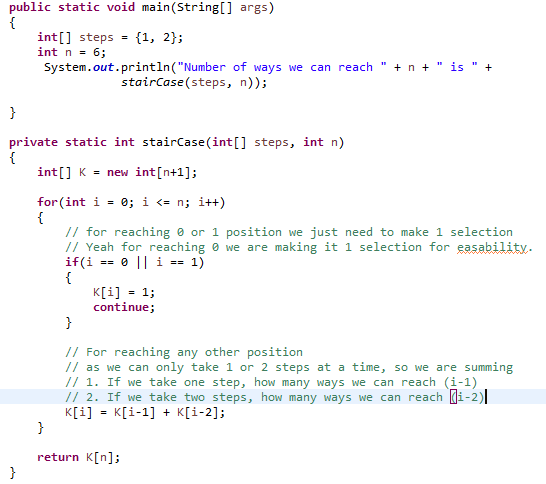
And in second case first 2 steps then 1 and 1 step. So both of the cases are different.

### Recursive solution

Here time complexity will be exponential **O (2n).** Because we will have multiple over lapping sub problems.

### Dynamic programming





# Range DS

## Range minimum segment tree

Range minimum segment tree is used to represent the minimum values of array in a tree format (stored in form of array). Where the cost of maintaining the tree is **O(n)** and cost of finding minimum between a range is **O(log n).**

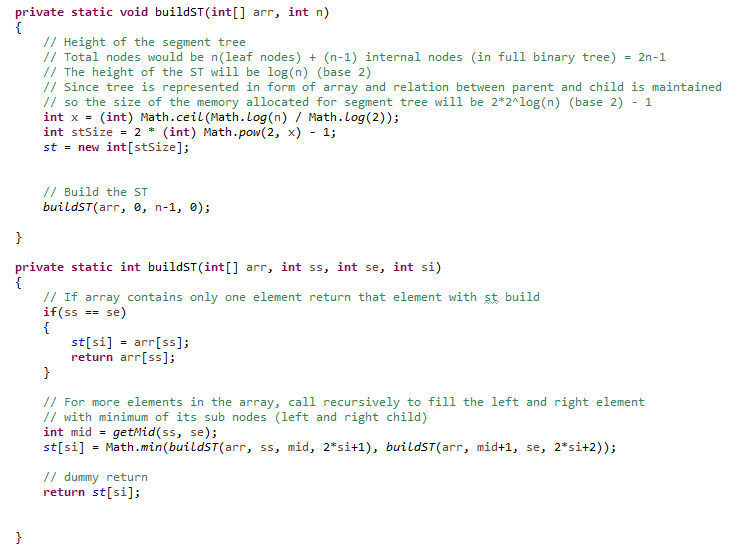
The extra space required is **O(n)** to store the segment tree.

### Construction of segment tree

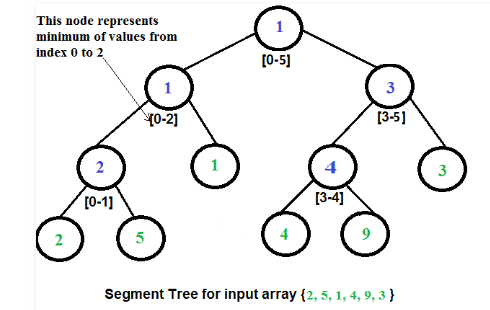
We start with segment [0 … n-1] and every time we divide it into two halves (till the segment length becomes 1).

The tree will be a fully binary tree as we always divide the tree into 2 halves in each level. So the tree will have **n-1** internal nodes for **n leaves.** So the total number of nodes will be **2n -1.**

Height of the segment tree will be **log2 n.** Since the tree is represented using array and relation between parent and child index must be maintained. So the size of the memory allocated for segment tree will be **2 \* 2 log2 n – 1.**

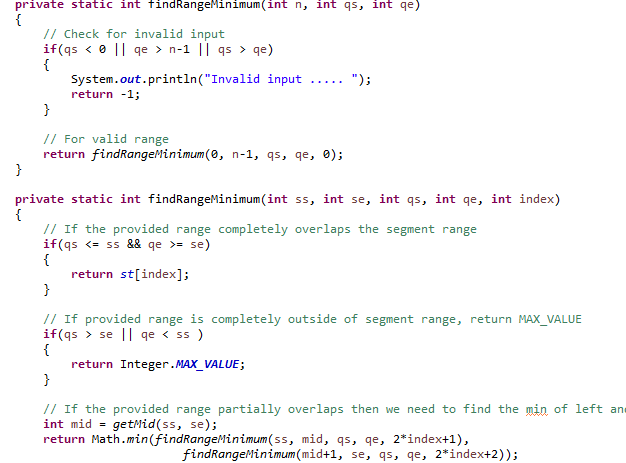


### Example



### To find range minimum

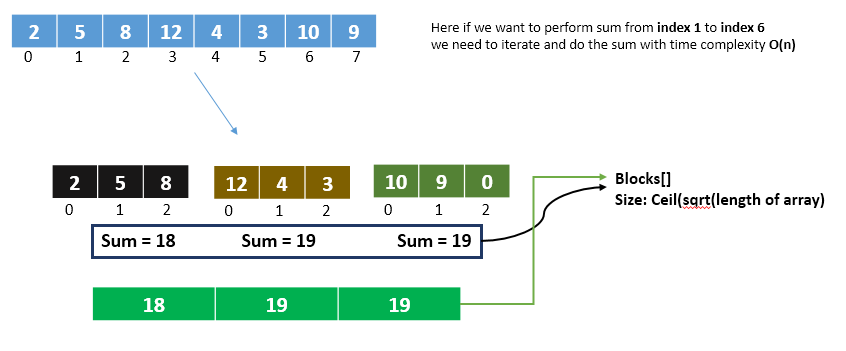
To find range minimum we should start will root of the tree i.e [0 … n-1]



## Sqrt decomposition

This technique is used to perform range operation in **sqrt** size of blocks where the time complexity goes down to **O()** from **O(n)**

### Range sum



Now we can have maximum size of **block array** will be **ceil ()**

When we need to do the range sum ex **index 1** to **index 6**

* Sum all the elements from start index to last index of first block (max elements)
* Sum all the in between **blocks** value (max blocks)
* Sum all the elements from start index of last block to last search index (max elements)

Total time complexity **O( +O(**

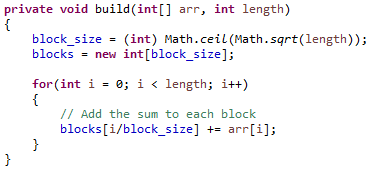
### Source code

We have 3 parts **build** the sqrt decomposition structure of the array

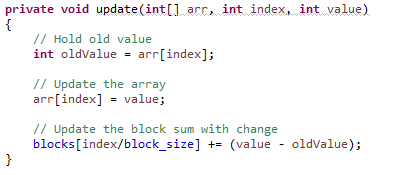
**Update** the element and hence the block sum **O(1)** complexity

**Query the sum** with complexity **O(**

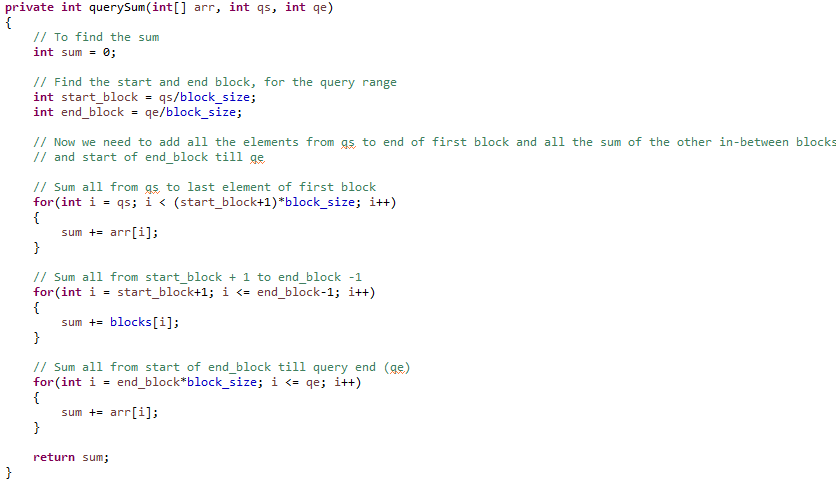
### Build



### Update



### Query the sum

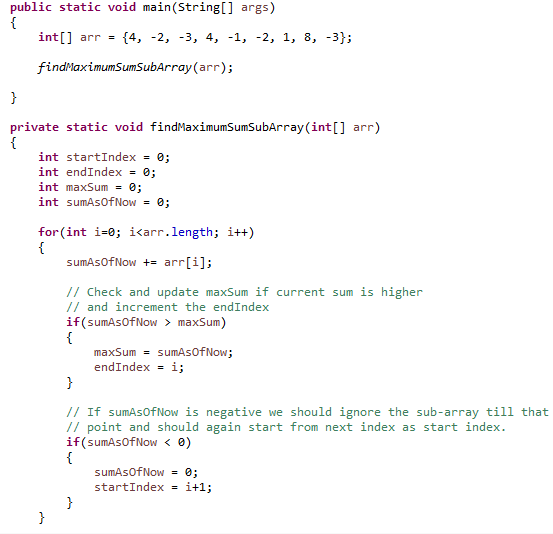


# Array Data Structure

## Kadane’s algorithm

Kadane’s algorithm is used to find maximum sub-array in a **negative** value contained array in complexity of **O(n).**

### Program



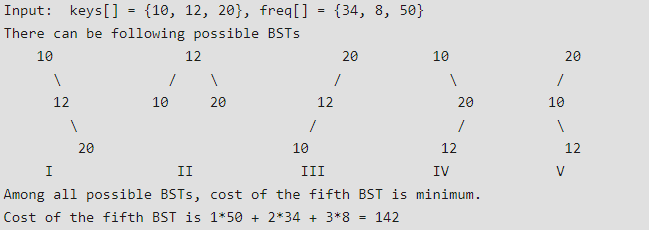


# Tree

Tree is recursion.

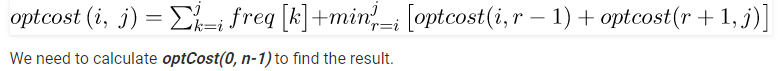
## Optimal binary search tree

If the elements and frequencies of search is provided. Optimal binary search tree is the arrangement of the elements in form of nodes, such that the total cost of the search will be minimum.



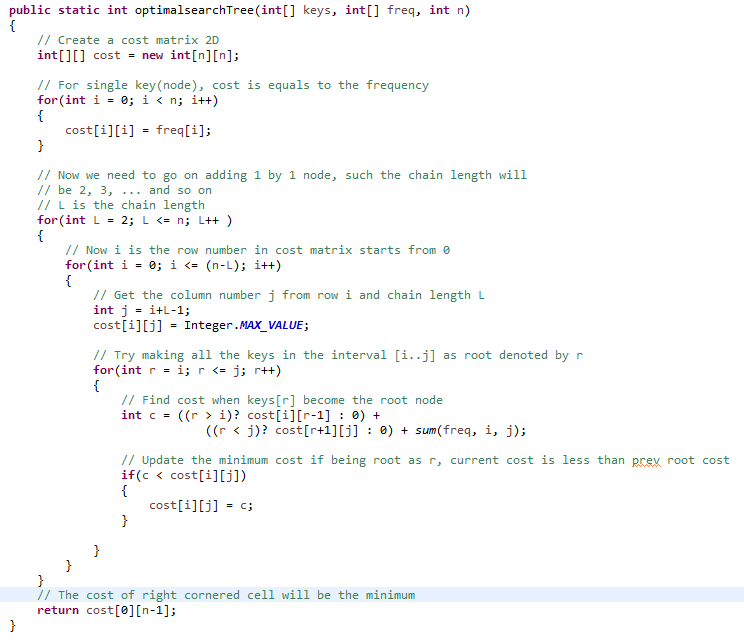
### Solution with optimal substructure using dynamic programming

The optimal cost for frequency[i…j] can be calculated using following formula.



Here we one by one try all the node as root node (r varies from i to j). When we make rth node as root node we recursively calculate (can be stored in a 2D matrix when used dynamic programming) **i** to **r-1** and **r+1** to **j.** we also need to add sum of frequencies from i to j this is added because every search will go through root and one comparison will be done for every search.

### Program



# Heap Data Structure

**Heap** is a tree based data structure where the tree is a complete binary tree. Generally heaps are 2 types **Max-Heap** and **Min-Heap**

## Making a heap

### Max-Heap

In Max-heap the key present in the root node is greater than to its children. The same concept is recursively true for all of the sub-tree in that binary tree. Then that binary tree can be considered as Max-Heap.

### Min-Heap

In Min-heap the key present in the root node is less than to its children. The same concept is recursively true for all of the sub-tree in that binary tree. Then that binary tree can be considered as Min-Heap.

Either a heap will be Max-Heap or Min-Heap

### Heap in a array

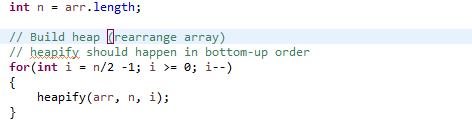
Array is a best representation for array.

* **Left node index:** 2\*index + 1
* **Right node index:** 2\*index + 2
* **Root node index from current index:** index/2

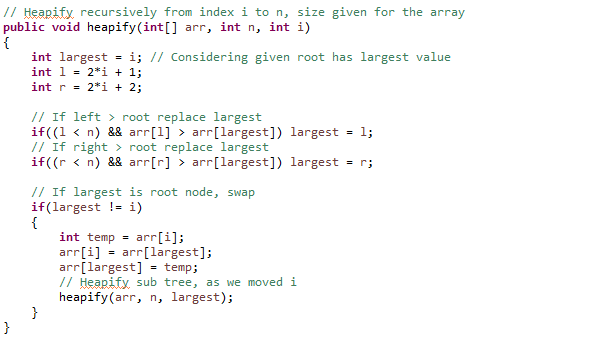
Heapify procedure can be applied to a node only if its children nodes are heapified. So the heapification process works in **bottom-up** order.

## Building a heap code

### Arrange the array in heap



### Heapify (Max Heap)



### Sorting from that heap

