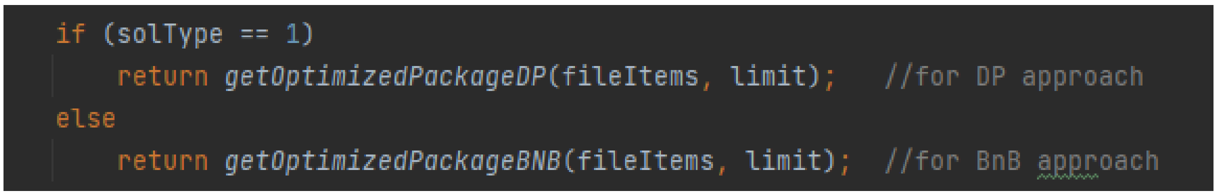
Solving the problem

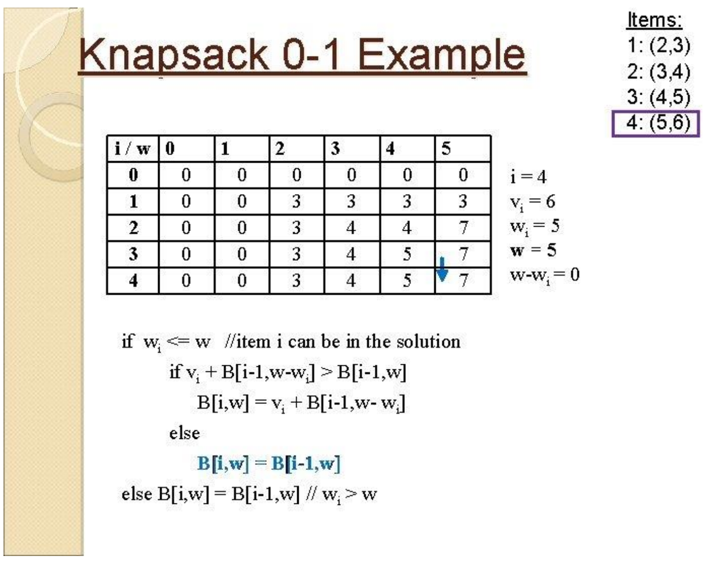
I used 2 different approaches for the solution of knapsack problem. These 2 approaches are **dynamic programming** and **branch and bound.**

In packer.java's solveProblem method, you can select which approach yo want to solve problem by using solType



```java
if (solType == 1)
    return getOptimizedPackageDP(fileItems, limit);   //for DP approach
else
    return getOptimizedPackageBNB(fileItems, limit);  //for BnB approach
```

Dynamic Programming Approach for Knapsack



Knapsack 0-1 Example

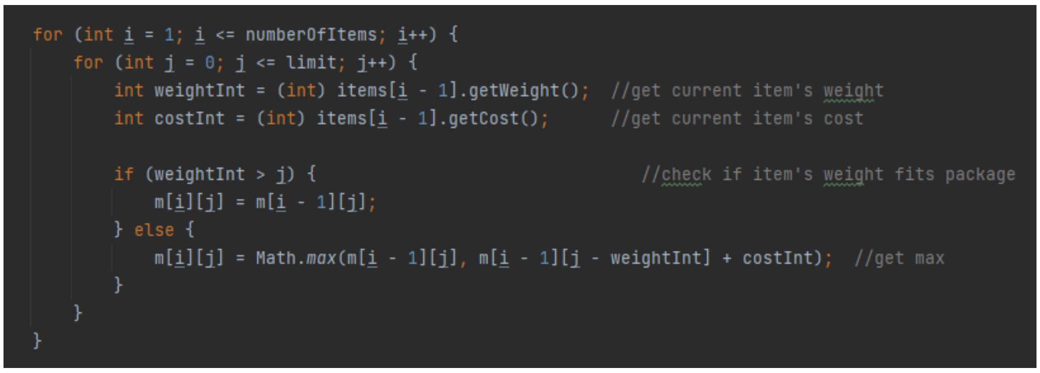
Items:
1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

i / w	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4	5	7
4	0	0	3	4	5	7

$i = 4$
$v_i = 6$
$w_i = 5$
$w = 5$
$w - w_i = 0$

```
if w_i <= w  //item i can be in the solution
    if v_i + B[i-1,w-w_i] > B[i-1,w]
        B[i,w] = v_i + B[i-1,w- w_i]
    else
        B[i,w] = B[i-1,w]
else B[i,w] = B[i-1,w] // w_i > w
```

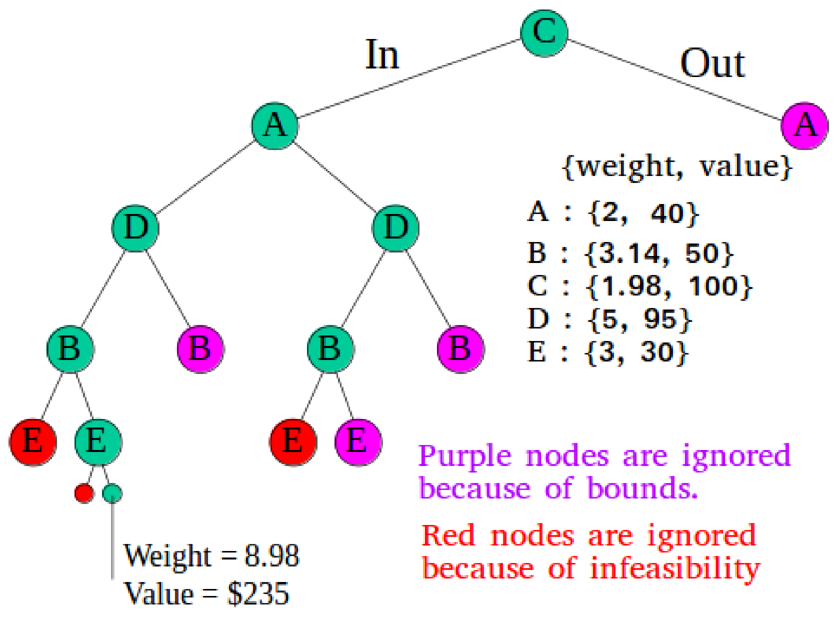
Main idea is, use a matrix to store solutions of solutions of subproblems. If current item's weight is fitting to bag, check if previous weight is higher than previous object plus current item's cost. (table above is more explaining actually =))



```java
for (int i = 1; i <= numberOfItems; i++) {
    for (int j = 0; j <= limit; j++) {
        int weightInt = (int) items[i - 1].getWeight();  //get current item's weight
        int costInt = (int) items[i - 1].getCost();      //get current item's cost

        if (weightInt > j) {                              //check if item's weight fits package
            m[i][j] = m[i - 1][j];
        } else {
            m[i][j] = Math.max(m[i - 1][j], m[i - 1][j - weightInt] + costInt);  //get max
        }
    }
}
```

Branch and Bound Approach for Knapsack

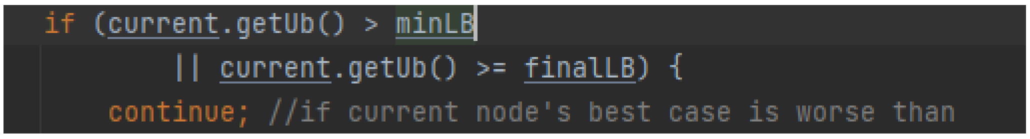


Firstly sort items by most effectively order. Create a dummy node and insert into priority queue.

While queue is not empty;

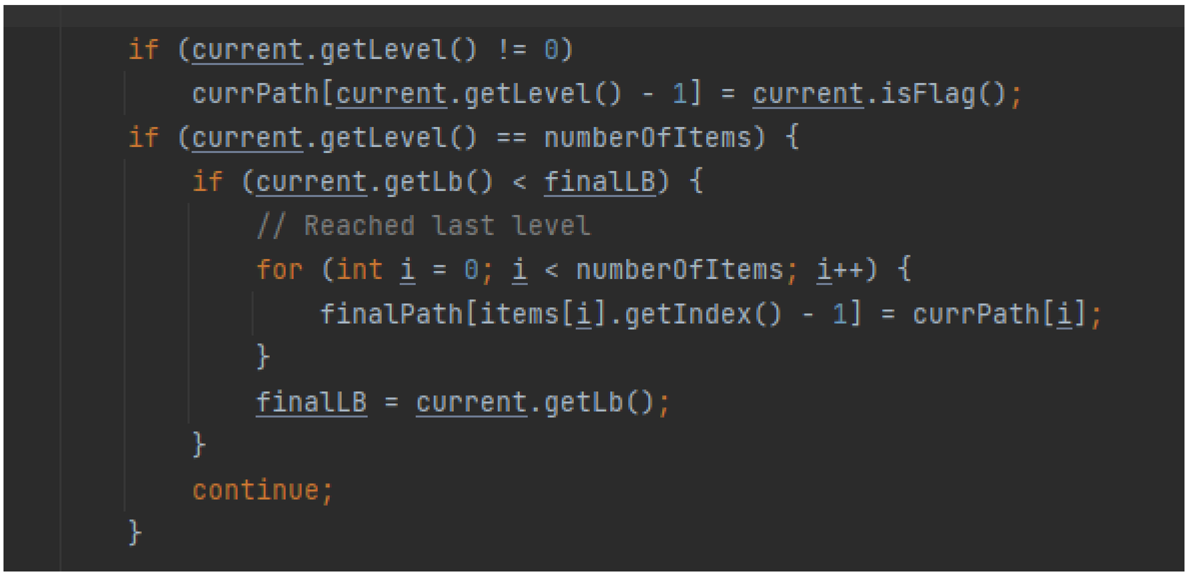
Extract the peek element from the priority queue and assign it to the current node.

If the upper bound of the current node is less than minLB, the minimum lower bound of all the nodes explored, then continue with the next element. If the best value itself is not optimal than **minLB**, then exploring that path is of no use.



```
  if (current.getUb() > minLB
          || current.getUb() >= finalLB) {
      continue; //if current node's best case is worse than
```

If the current node's level is at items.length()(numberOfItems), then check whether the lower bound of the current node is less than finalLB, minimum lower bound of all the paths that reached the final level. If it is true, update the finalPath and finalLB. Otherwise, continue with the next element.

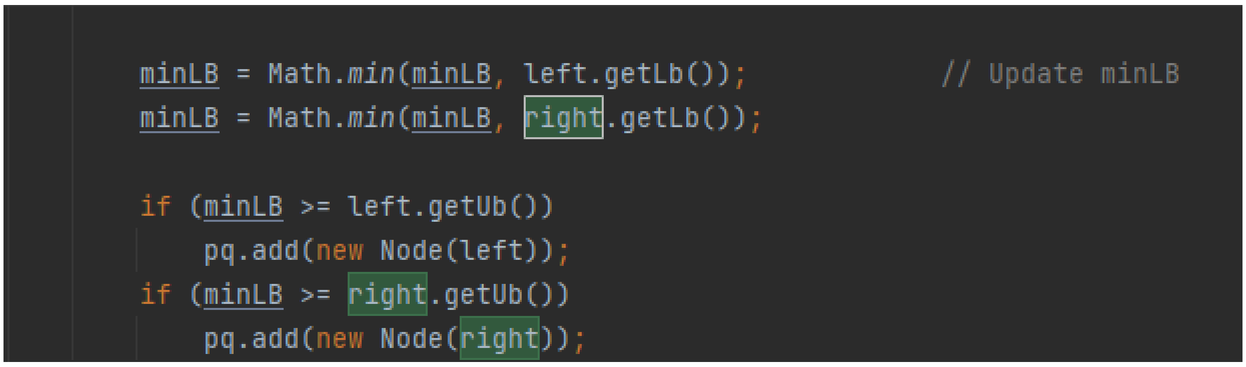


```
if (current.getLevel() != 0)
    currPath[current.getLevel() - 1] = current.isFlag();
if (current.getLevel() == numberOfItems) {
    if (current.getLb() < finalLB) {
        // Reached last level
        for (int i = 0; i < numberOfItems; i++) {
            finalPath[items[i].getIndex() - 1] = currPath[i];
        }
        finalLB = current.getLb();
    }
    continue;
}
```

Calculate the lower and upper bounds of the right child of the current node.

If the current item can be inserted into the bag, then calculate the lower and upper bound of the left child of the current node.

Update the minLB and insert the children if their upper bound is less than minLB.

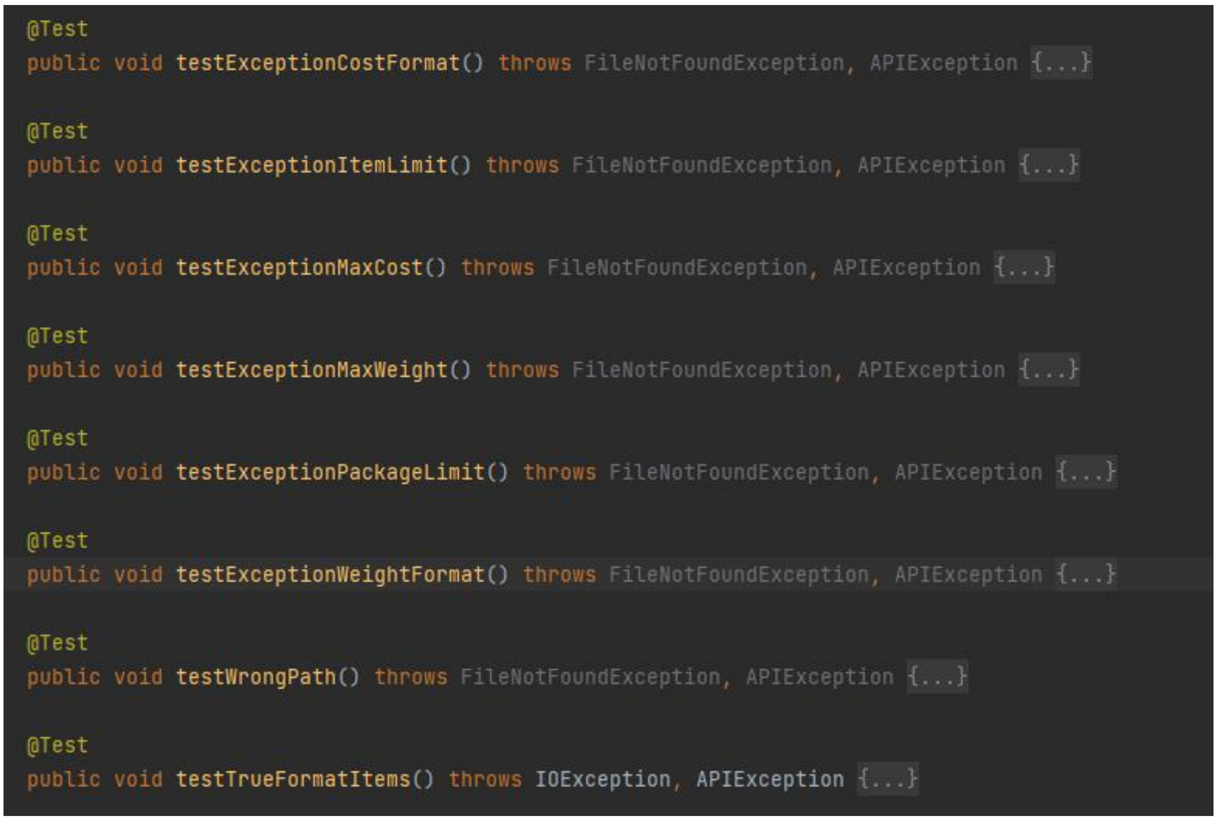


```
minLB = Math.min(minLB, left.getLb());          // Update minLB
minLB = Math.min(minLB, right.getLb());

if (minLB >= left.getUb())
    pq.add(new Node(left));
if (minLB >= right.getUb())
    pq.add(new Node(right));
```

Testing

I created 8 test cases. One is true formatted which returns the solution. Other 7 are returning exceptions for different cases.



```
@Test
public void testExceptionCostFormat() throws FileNotFoundException, APIException {...}

@Test
public void testExceptionItemLimit() throws FileNotFoundException, APIException {...}

@Test
public void testExceptionMaxCost() throws FileNotFoundException, APIException {...}

@Test
public void testExceptionMaxWeight() throws FileNotFoundException, APIException {...}

@Test
public void testExceptionPackageLimit() throws FileNotFoundException, APIException {...}

@Test
public void testExceptionWeightFormat() throws FileNotFoundException, APIException {...}

@Test
public void testWrongPath() throws FileNotFoundException, APIException {...}

@Test
public void testTrueFormatItems() throws IOException, APIException {...}
```