2 Solve the 0-1 knapsack problem using dynamic programming

pseudocode:

Max (int x int y)
Return the larger of x and y

knapsackProblem(int weight capacity, array of weights, array of values, int length of both the arrays)

Create the table to hold the subproblem solutions T[][] = new int[length + 1][weight capacity + 1]

for(loop through n)
    for(loop through weight capacity)

Set the cells in the first row and column of the table to be zero if either value of the loops are zero

If (the current weight is less than the weight capacity)
    Set the current cell of the table to be the larger of the current value plus the value in the cell that is in the row above and at the column position decided by its weight table [i - 1][weightCapacity - weights[i - 1]]

Else
    Set the value of the current cell to the be value of the cell above it

Return the last cell in the table

Proof:
This algorithm is correct as we have gone through solving all the sub problems in a bottom up manner till we reach the last cell in the table, solving all the possible subproblems before reaching the main problem ensures that this is the optimum solution, showing we have optimal substructure.

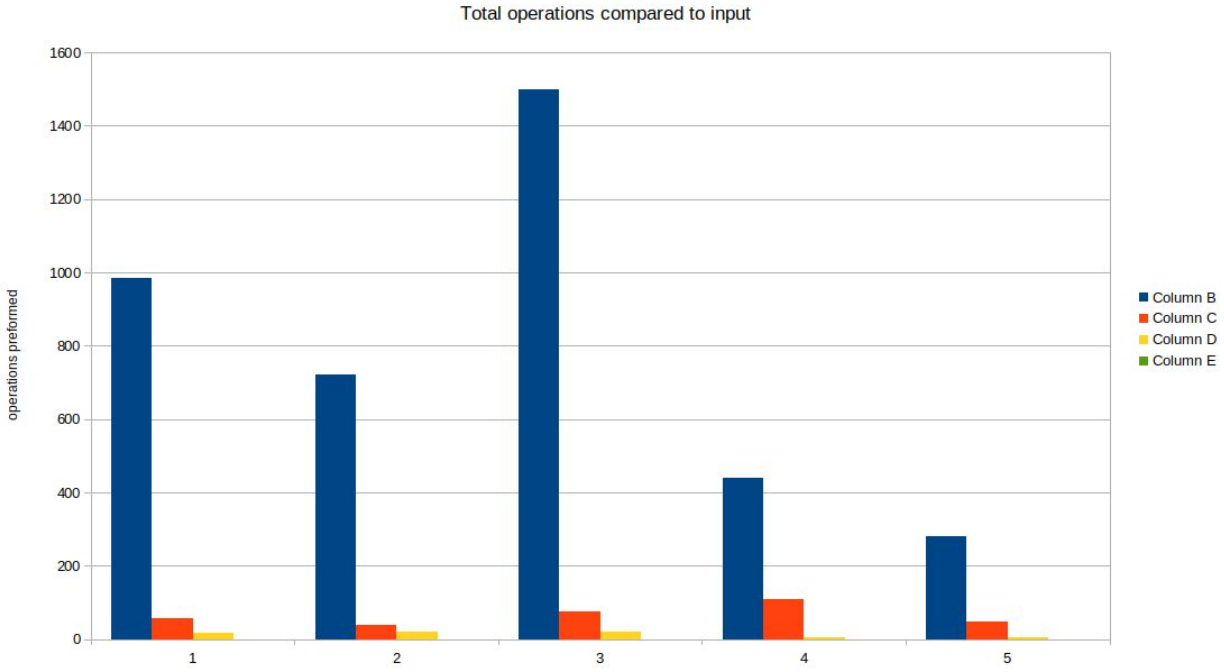
By breaking down the problem into the smallest subproblems possible we reach the base cases:

Case 1:
This instance is included in the optimal subset.

Case 2:
This instance is not included in the optimal set.

test case generator
My test case generator works by running 10 instances of my knapsack algorithm, each time passing it variables of random lengths weights and values to ensure it works on all variable lengths and values, the program then prints the number of operations we would assume the program would take to complete the calculation as well as the accrual number of operations performed and the time in nanoseconds the program completed the calculator in. with this information we can see how the program handles different input

Plots of my algorithm:



As seen from the above graph, as the input is increased the number of operations and the amount of time taken in nanoseconds increases exponentially. The expected time complexity for the 0-1 knapsack problem when using dynamic programming is O(N * W) this being the number of values given (N) and the weight capacity (W), from the output of my algorithm i.e:

```
 1498 ---          operations performed

916               best value achieved
1498    max expected complexity (n * W )

677953.0          time taken in nanoseconds
====================================
```

It shows that the expected number of operations and the actual operations performed are the same, this mean that my algorithm has the expected time complexity of O(N * W).

3 Solve the 0-N knapsack problem using brute force (enumerate all choices?) (15%)

Pseudocode:

The program consists of two classes, BruteForceKnapsack for the algorithm and calculations and the other, Item which represents an item.

class item

```
int weight
int value
int id

item(int id, int weight, int value)

        this.id = id
        this.value = value
        this.weight = weight

getValue()
        return this.value;

getWeight()
        return this.weight;

getID()
        return this.id;
```

Class BruteForceKnapsack

RandomNumber(min, max)
Create a number between min and max

createItems(i)
Create i items with random value and weight

run() {

Create the items to be used

```
numberOfItems = randomNumber(1, 6)
item[] items = createItems(numberOfItems)
```

Passes the item ids to the brute method

```
for(int i =0; i <= r; i++){
    Brute(items, n, i);
```

```
 brutePass (item array int n int rep) {
      int currentIDs[] = new int[reps + 1]

Pass the variables to the recursive method
      BruteRecurvice(current, item array, 0, rep, 0, n - 1);
   }


BruteRecurvice(int current[], item arr[], int current, int rep, int first, int last) {

  if (current == rep) {

          int totalWeight
          int totalValue

          ArrayList  IDlist

          for (int i = 0; i < rep; i++) {

          Add the current item values to the temporary variables above

              totalWeight += arr[current[i]].weight;
              totalValue += arr[current[i]].value;
              IDlist.add(arr[current[i]].getID());

    if(the current values are the best so far add them to the class fields)

   Take the elements one by one and recursively call

   for (int i = first; i <= last; i++) {
          current[index] = i;
          BruteRecurvice(current, arr, index + 1, rep, i, end);
       }
```

Proof:

The proof for this algorithms correctness can be found by simply looking at how it finds the best combination, it creates every possible combination of the items it is passed and remembers the weights and highest value values as it creates them, if the current values are better than the previous best values then it is replaced by the new best values. This way it will always find the

best combination as it looks at every possible combination using brute force to create every possible combination

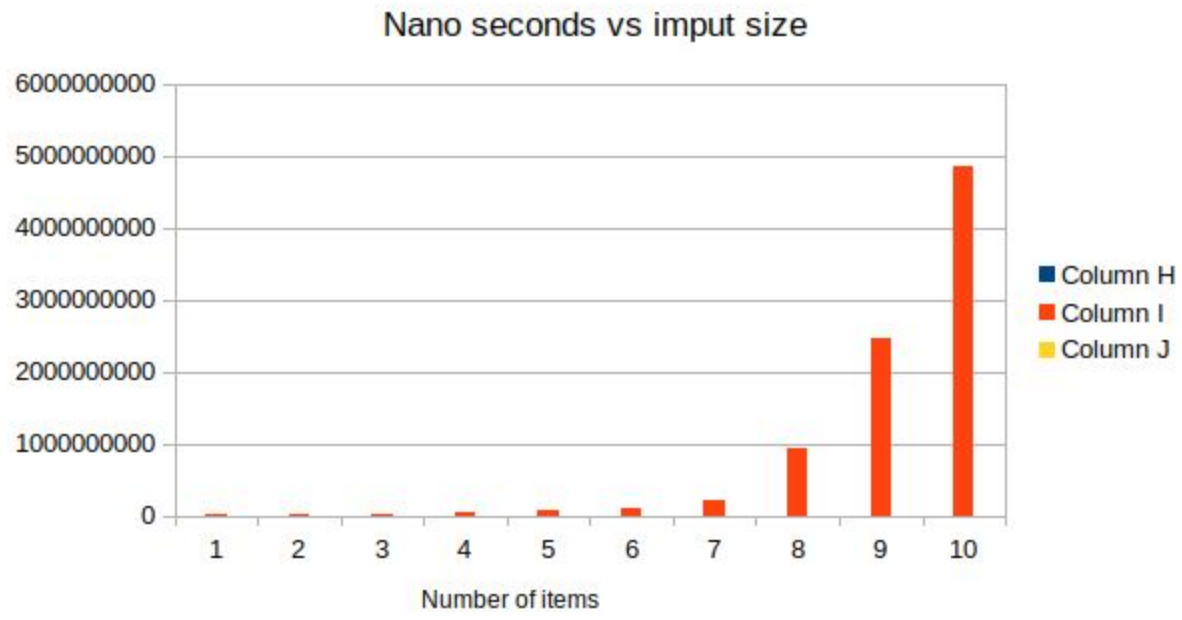
Test case generator:

As the program is already designed to use random values every time it runs the test case generator merely runs the program 10 times, printing the results. The program is set up in a way that everytime it is run it creates a random number of items each with random weights and values with a random weight capacity using the randomNumber() method so the program is always being tested on variables of different sizes and values.

The test case generator was created by adding a for loop around the run() call in the main method to achieve this testing.

Plots of algorithm

The below graph plots the time taken in nanoseconds to complete the program on the Y axis and the number of items it was passed as shown on the X axis.



The above times were found by putting the following code around the run() call

```
long start = System.nanoTime();
        run();

        long end = System.nanoTime();

        double totalTime = end - start;
```

This gives us the time taken in nanoseconds to complete the program's main function of creating all the possible combinations and finding the best one.

As seen from the above graph the time taken in nanoseconds to complete the calculation of creating every combination from the given input increases at an exponential rate as the input size is increased by one starting at the input size of 4 as it would seem the time taken to calculate input sizes 1 - 3 take relatively the same amount of time. With this growth rate i would assume that time complexity to be O(N^2).

4 Solve the 0-N knapsack problem using dynamic programming

pseudocode:

knapSackBounded(int weight capacity, array of weights, array of values, int length of both the arrays)

Create the table to hold the subproblem solutions T[][] = new int[length + 1][weight capacity + 1]

for(loop through n)
        for(loop through weight capacity)

Set the cells in the first row and column of the table to be zero if either value of the loops are zero

If (the current weight is less than the weight capacity)

Set the current cell of the table to be the larger of the current value plus the value in the cell that is in the row above and the item that is currently being used

Else

Set the value of the current cell to the be value of the cell above it

Return the last cell in the table

maxValue (int x int y)
Return the larger of x and y

main ()
Calls the test generator

Proof:

The algorithm differs from the 0-1 version by changing the equation:

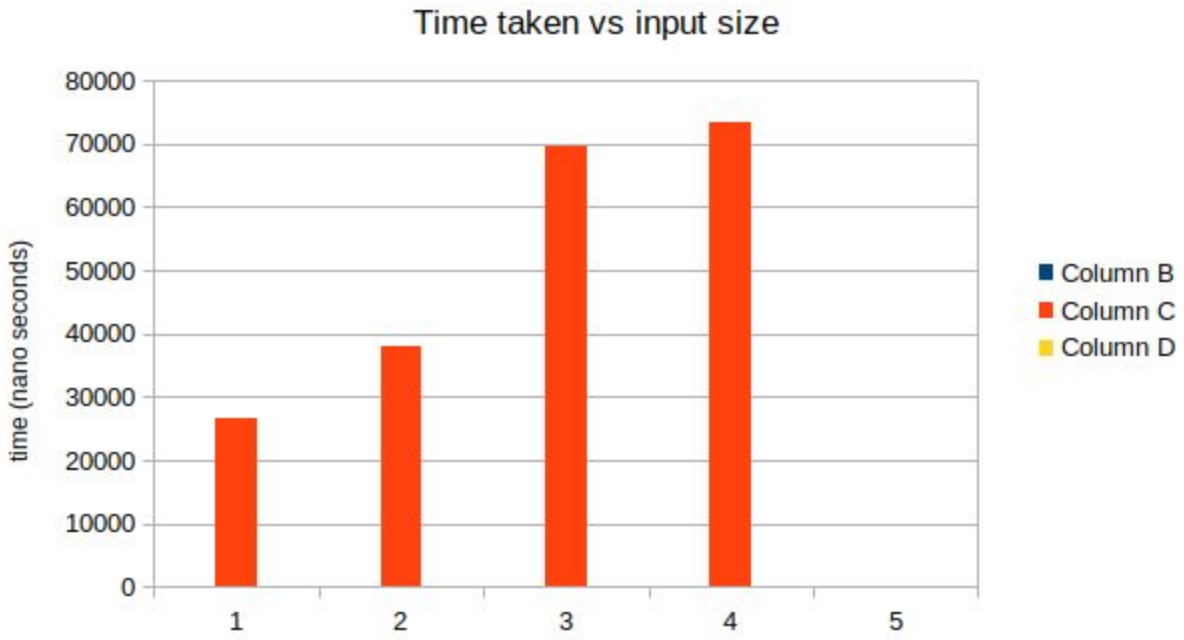
```
valueTable[i][w] = maxValue( values[i - 1] + valueTable[i - 1][w - weights[i - 1]],  (valueTable[i - 1][w]));
```

To

```
valueTable[i][w] = maxValue( values[i - 1] + valueTable[i - 1][w - weights[i - 1]],  (valueTable[i][w]));
```

The difference in (valueTable[i - 1][w])); to (valueTable[i][w])); this changes the recursion to remove the element or to not removing the element, this way we can look at an item multiple times giving us more possibilities for an optimal filling of the knapsack.

Plots of algorithm



The above algorithm shows the time taken in nano seconds to complete the programs computation, the first bar of the graph is and input size of 3 for both values and weights, the second bar is an input size of 6 for both values and weights, the third bar is an input size of 9 for both values and weights, the fourth bar is an input size of 12 for both values and weights. As shows in the graph the time taken increases in a linear mannar, due to this i would assume that the time complexity is of O(N * W) time.

5 Solve the 0-N knapsack problem using graph search

The program consists of two classes, Graph Search which holds the main algorithm for finding the best solution and the node class that is used to hold the partial solution.

Pseudocode:

Graph Search class:

```
node root;
weightCapasity = 200;
int bestValue;
int       bestWeight;
String best;

build(node n) {
        Method to build the graph

        if(n.getWeight() >= weightCapasity) {

        Stop the recursion

        else

        node left = new node nS+1, n.R(), n.P
        node middle = new node n.S, n.R+1, n.P
        node right = new node(n.S, n.R, n.P+1)

        n.setLeft(left);
        n.setMiddle(middle);
        n.setRight(right);

        build(left);
        build(middle);
        build(right);



   search(node root){
   Method to find the best solution

Queue<node> queue = new LinkedList<node>();
queue.add(root);
while !queue.isEmpty()

node tempNode = queue.poll();
```

```
if(tempNode.getWeight() < weightCapasity && tempNode.getValue() > bestValue)
```

If the current node is better than the previous best solution then add its values to the best fields

```
        bestWeight = tempNode.getWeight()
        bestValue = tempNode.getValue()
        best = tempNode.toString()
    }

    if(tempNode.getLeft() != null){
        queue.add(tempNode.getLeft())

    }
    if(tempNode.getMiddle() != null){
        queue.add(tempNode.getMiddle())

    }
    if(tempNode.getRight() != null){
        queue.add(tempNode.getRight())
```

Node class:

Variables for the partial solution:
Int Sapphires
Int pearls
Int rubys

Node left
Node middle
Node right

```
int getValue()
```

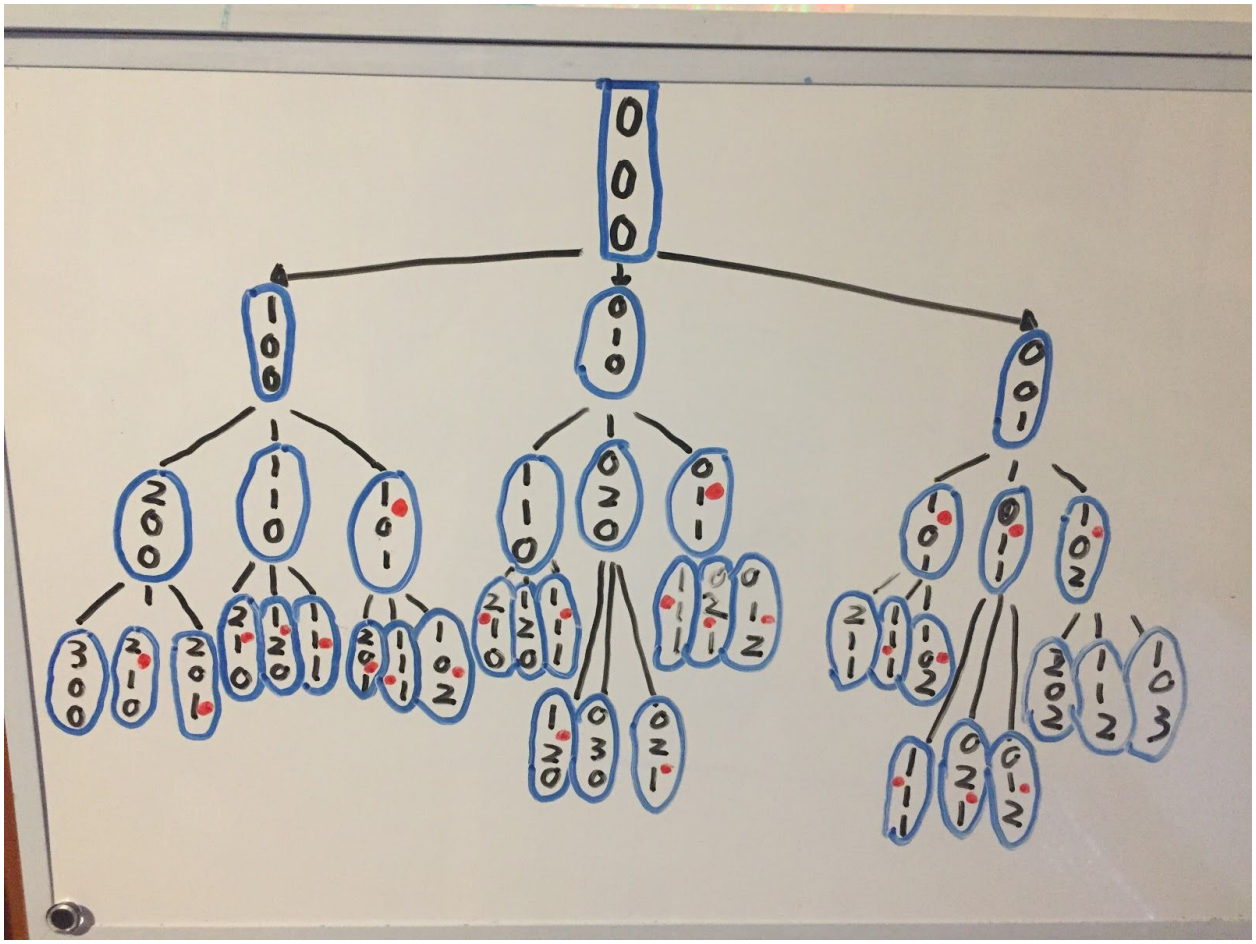
```
        return value;

int getWeight()
        return weight;

 public int getS()
        return s;

 int getR()
        Return r

 int getP()
        return p

 setters for children

setLeft(node ln)
        left = ln;

 setMiddle(node mn)
        middle = mn;

 setRight(node rn)
        right = rn;


getters for children

        node getLeft()
                return this.left;

        node getMiddle()
                return this.middle

        node getRight()
                 return this.right

        toString()
                 String a = s +" "+ r +" " + p
                 return a;
```

Proof:

The algorithm finds the best solution by creating every possible combination of the items while being under the weight capacity, in this case the rubies sapphires and pears as described in the lecture material. To do this the algorithm uses a node object to represent a possible solution, each node has three children. When adding each of the three children to the node we give its left child node the values of its parent but add one to the number of sapphires it holds we then do this with the middle node giving it the same values of its parent node but adding one to the number of rubys it holds then similarly to the right node we give it the values of its parent but adding one to the number of pearls. We then recursively call the build method on these three nodes, this eventually creates every possible combination of sapphires rubies and pearls while still being under the weight constraint.

The algorithm then uses breadth first search to look through the entire graph starting at the root node adding the best solution found so far to the fields to hold the best value, at the end of the search we have found the best solution.

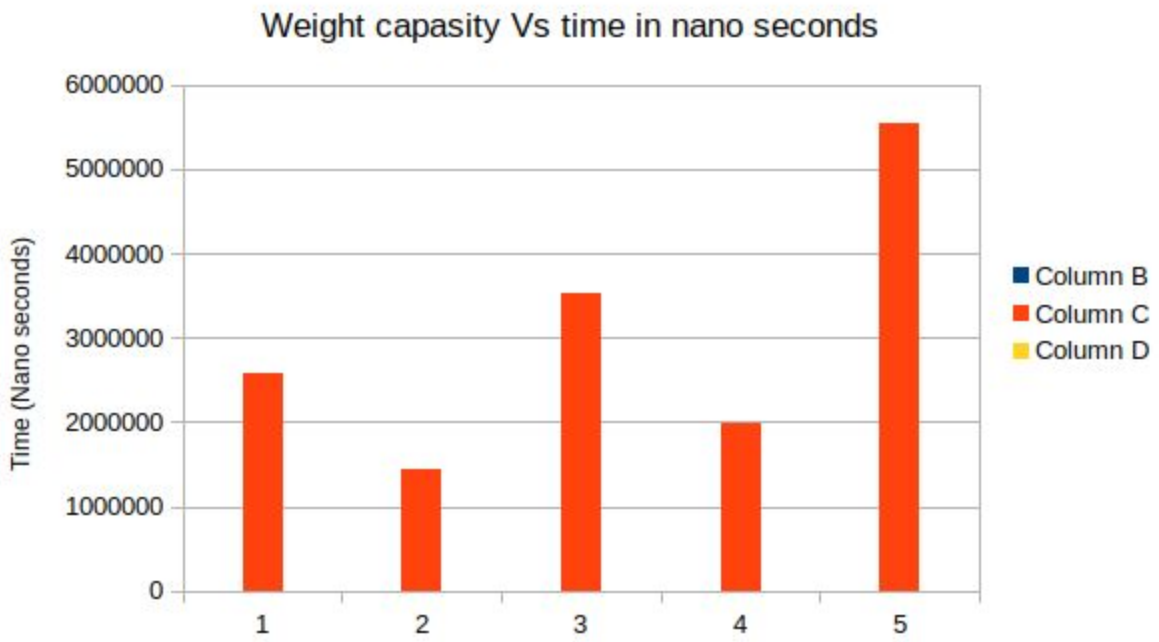
The recursion can be visualized by the photo below



The red dots represent parietal solutions that have duplicates within the graph, this is obviously an issue that affects the computational performance of my algorithm.

• Plots of your algorithm using an appropriate barometer and a conclusion of what your implementation's complexity is.

This graph below shows the running time of the program with different weight capacities, as the weight capacity acts as the stopping criteria for the recursion and is responsible for the depth of the graph.



In the above graph the first bar represents the program's run time in nanoseconds with the weight capacity being 50, the second 100, the third 150, the fourth 200 and the fifth 250.
As you can see from the graph the time taken increases linearly in the first third and fifth bars but drops in the second and fourth but between these two smaller bars the time also increases linearly, i believe this to be due to the weights favoring the left most nodes making the search quicker. The time complexity of this would be $\Theta(n + e)$ with n being the number of vertices and e being the number of edges, this complexity causes the run time to suffer greatly on weight capacity over 400 as the amount of nodes it creates becomes exponentially larger with many duplicates.

6 Challenge: improve your graph search algorithm

• A description of why your improved graph search algorithm can perform faster.

My new graph search algorithm can now perform faster due to the following changes I have made to the graph search method.

Firstly i have implemented pruning by changing the search method to set a node representing a possible solution to be null if it weight is over the weight capacity defined at the top of the class, this means the method will no longer search though any branches that are already undesirable due to their weight as it will only become larger with each recursive call.

I have also changed the queue of the possible solutions to a priority queue using a compare that compass the weights of the solutions within the weight capacity, this means we will look at the best solutions first and will be able to prune the undesirable solutions sooner.

7 Theoretical questions (15%)