Knapsack algorithm – Assignment 1

Varun Gattu
A02092613

1. Recursive solution for knapsack

Language: Java.
Code:

```
package knapsack;
import java.io.*;
import java.util.Random;
public class KnapSack {

  private static int[] N;
  private static boolean output = false;
  public static void main(String[] args) throws IOException {

    long initialTime = System.currentTimeMillis();
    Random r = new Random();
    N = new int[50];

    for (int i=0;i<50;i++)
    {
      N[i] = r.nextInt(100);
    }

    Boolean x = knap(50,1000,1000);
    System.out.println("\nX="+x);
    long finalTime = System.currentTimeMillis();
    long timeTaken = finalTime - initialTime;
    System.out.println("\nTime taken = " + timeTaken);

  }
  private static Boolean knap(int X, int L1, int L2) {

    if(L1==0 && L2==0) { output=true; return output; }
    if(X==0) { return false; }
    if(L1<0 || L2<0) { return false; }
    if(output==false) {
      return (knap(X-1,L1-N[X-1],L2)||knap(X-1,L1,L2-N[X-1])||knap(X-1,L1,L2));

    }
```

```
        else return output;

    }
}
```

---

2. Knapsack using memorizing:

Code:

```
package knapsack3;

import java.io.IOException;
import java.util.Random;

public class Knapsack3 {
    private static int[] N;
    private static boolean output = false;
    private static boolean[][][] cacheValid;
    private static boolean[][][] cache;
    public static void main(String[] args) throws IOException {

        long initialTime = System.currentTimeMillis();
        Random r = new Random();
        N = new int[50];

        for (int i=0;i<50;i++)
        {
            N[i] = r.nextInt(100);
        }

        cache = new boolean[51][1001][1001];
        cacheValid = new boolean[51][1001][1001];
        Boolean x = knapMemo(50,1000,1000);
        System.out.println("\nX="+x);
        long finalTime = System.currentTimeMillis();
        long timeTaken = finalTime - initialTime;
        System.out.println("\nTime taken = " + timeTaken);
    }
    private static Boolean knapMemo(int X, int L1, int L2) {

        if(L1==0 && L2==0) { output=true; return output; }
        if(X==0) { return false; }
        if(L1<0 || L2<0) { return false; }

        if(cacheValid[X][L1][L2] != true)
        {
           Boolean abcd = (knapMemo(X-1,L1-N[X-1],L2)||knapMemo(X-1,L1,L2-N[X-
1])||knapMemo(X-1,L1,L2));
           cacheValid[X][L1][L2]=true;
           cache[X][L1][L2]=abcd;
        }

        return cache[X][L1][L2];

    }
}
```

---

3. **Knapsack using dynamic programming:**

Code:

```
package knapsack55;

import java.util.Random;

public class Knapsack55 {

    private static int[] N;
    private static boolean[][][] cache;

             static boolean[][] cacheL1;
        static boolean[][]cacheL2;

        public static void main(String[] args) {

        long initialTime = System.currentTimeMillis();
        Random r = new Random();
        N = new int[50];

        for (int i=0;i<50;i++)
        {
           N[i] = r.nextInt(100);
        }
```

```
cache = new boolean[51][1001][1001];
cacheL1 = new boolean[51][1001];
cacheL2 = new boolean[51][1001];
Boolean x = knapDP(50,1000,1000);
System.out.println("\nX="+x);
long finalTime = System.currentTimeMillis();
long timeTaken = finalTime - initialTime;
System.out.println("\nTime taken = " + timeTaken);

      knapDP(50,1000,1000);
    }

    public static boolean knapDP(int X,int L1,int L2)
    {
   for (int i = 0; i <= X; i++) {
                        cache[i][0][0] =true;
                                    cacheL1[i][0] =true;
                        cacheL2[i][0] =true;
                              }
                              for (int i = 1;  i <= L1; i++) {
                     cache[0][i][0] =false;
                                    cacheL1[0][i] =false;
                              }
                              for (int i = 1; i <= L2; i++) {
                                    cache[0][0][i] =false;
                        cacheL2[0][i] =false;
                              }

              //Recusrion case
      for (int i=1 ; i <= X; i++){
     for (int j=1; j<=L1 ; j++){
        if(j-N[i-1]>=0){

                                cacheL1[i][j] = cacheL1[i-1][j-N[i-1]] || cacheL1[i-1][j];
        }
     }
     for (int j=1; j<=L2 ; j++){
        if(j-N[i-1]>=0){

                                cacheL2[i][j] = cacheL2[i-1][j-N[i-1]] || cacheL2[i-1][j];
        }
     }
      }

  for (int i=1 ; i <= X; i++){
     for (int j=1; j<=L1 ; j++){
            for (int k=1; k<=L2 ; k++){
                cache[i][j][k] = cacheL1[i][j] && cacheL2[i][k];
            }
        }
         }
        return cache[X][L1][L2];

      }
}
```

---

106. Compare the run-time performance of knapDP() and knapMemo() for a single large problem size, but under different random object size distributions. Set the number of objects to be 50 and L1 and L2 to 1000, making the cache size 50 times 106. If your language, machine or OS will not allow allocation of this much memory, then reduce the size to the maximum possible.

Answer:

<u>Output of KnapDP() :-</u>

X=true

Time taken = 370

<u>Output of KnapMemo() :-</u>

X=true

Time taken = 248

---

107. The idea behind this experiment is to understand how the distribution of object sizes effects which algorithm is more efficient. Because DP computes every possible subsolution, we would expect its performance to be consistent over all distributions. However, memoizing only computes those solutions that are created by the sizes of objects. When the objects are mostly small, we would expect memoizing to take about the same or more time than DP. When the objects are mostly big, we would expect memoizing to take less time than DP. (Why? Explain this in your brief report)

Answer:

Memorizing is a top down approach to solve the top problem first and then the sub problems. But, dynamic programming is a bottom up approach to solve the problems.

Memorizing uses a cache to store all the previously computed values and hence uses the stored value when needed again. So, for smaller datasets memorization and dynamic programming take almost the same time whereas for bigger problems, dynamic programming computes the same value again and again when encountered.

---

108. Design an experiment where you vary the average size of the objects from small (say 1) to big (say 1000) and for each average size run knapDP() and knapMemo(). Time the two algorithms solving the same problems and keep track of which is best over 10 random runs (for the same average size). While you can change the average size, you must make sure that the actual sizes of the objects is randomly distributed.

Answer:

The experiment results:

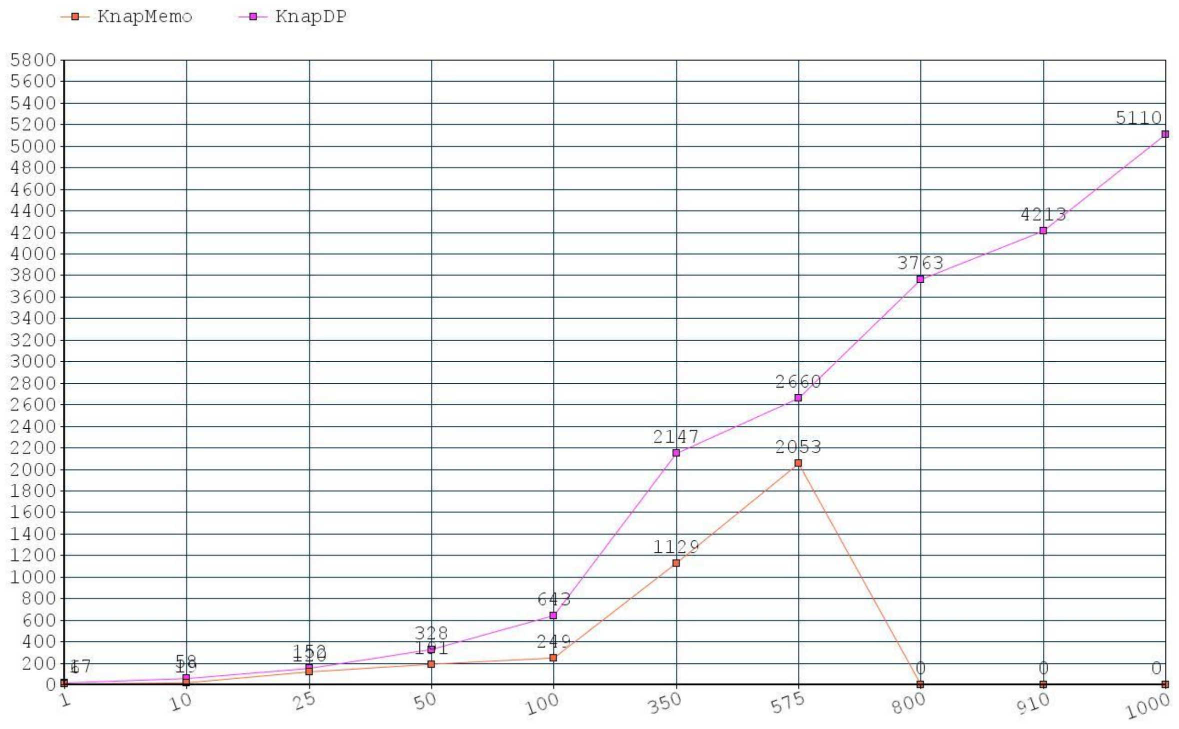
Size of knapsack1 = 1000
Size of knapsack2 = 1000

Number of objects	KnapMemo	KnapDP
1	6	17
10	19	58
25	120	152
50	191	328
100	249	643
350	1129	2147
575	2053	2660
800		3763
910		4213
1000		5110

The last 3 results for KnapMemo is not calculated because it takes a lot of memory to calculate values. Observations state that memorization takes less time than dynamic programming for bigger data sets.

---

109. Generate a single graph where the x axis is the average object size and the y axis is the average time to run the algorithm. There should be two lines on the graph, one for knapDP() and one for knapMemo().Your graph should be clearly captioned with the axis labeled. I recommend using gnuplot or some other professional quality graphing software.



Experiment:

For the recursive algorithm, generate a sequence of small problems of the same size L1 and L2, but with increasing object counts. Generate a table of run time vs. the number of objects. At what object count does it become infeasible to run the algorithm?

Answer:

The object count becomes infeasible for sample space:
Number of objects: 25
Size of knapsack 1: 1000

Size of knapsack 2: 1000

Number of objects	Recursive
1	2
10	4
25	
50	123
100	1
350	3
575	2
800	2
910	2
1000	2

Write a brief (4-6 sentence) technical explanation of how you setup the random distributions and another paragraph explaining the behavior of the algorithms derived from the graphs.

Observations:

Random distribution of number of objects to fill in the knapsack is implemented using Random function in java.util.Random statement. I used the function with an upper limit of 100 which states that all the random numbers generated are in between 0 and 100 included.

The graph shows that memorizing and dynamic programming give almost the same result for smaller datasets. But for bigger datasets, kanpMemo first increases and then decreases gradually and reaches out of memory at a point of time. KnapDP will keep increasing for the bigger datasets gradually.
