Assignment 3

Sanjana S Prabhu

September 15, 2019

1 Introduction

In this assignment, we try to find the weights which sum up to a given total weight, thus resembling a knapsack of weight W and N smaller weights of value w_i ,0<=i<=N. Also we have to determine the scaling and order of the algorithm we use for the same.

2 Part 1

Given N weights w_i and a total weight W, we need to find a subset of these weights such that they add up to W.

2.1 Pseudo Code

The following is the pseudo-code for the knapsack problem:

```
Boolean knapsack(int W,int i){
if W==0
return True
else if W<0 or i>=N
return False
else if knapsack(W-w[i],i+1)
print w[i]
return True
else
return knapsack(W,i+1)
end}
```

2.2 The Code

The following is the C code for the knapsack problem:

```
int knapsack(int *w, int W, int i, int N){
if (W==0)
```

```
return 1;
else if(W<0||i>=N)
return 0;
else if(knapsack(w,W-w[i],i+1,N))//recursive call
{printf("%d\n",w[i]);
 return 1;
}
else
return knapsack(w,W,i+1,N);//recursive call if the
previous combination of weights does not add up
}
```

3 Part 2

The second part of the assignment is to determine the scaling of the algorithm. For this we generate 10^4 sets of random values of N, w_i, W and pass these through a knapsack function which also counts the number of times the function has been called. Now the maximum and minimum values among the counts are determined for each value of N and printed in a table format shown in Figure 2. The large value of sets is to make sure that we actually generate the worst case error.

3.1 The Code

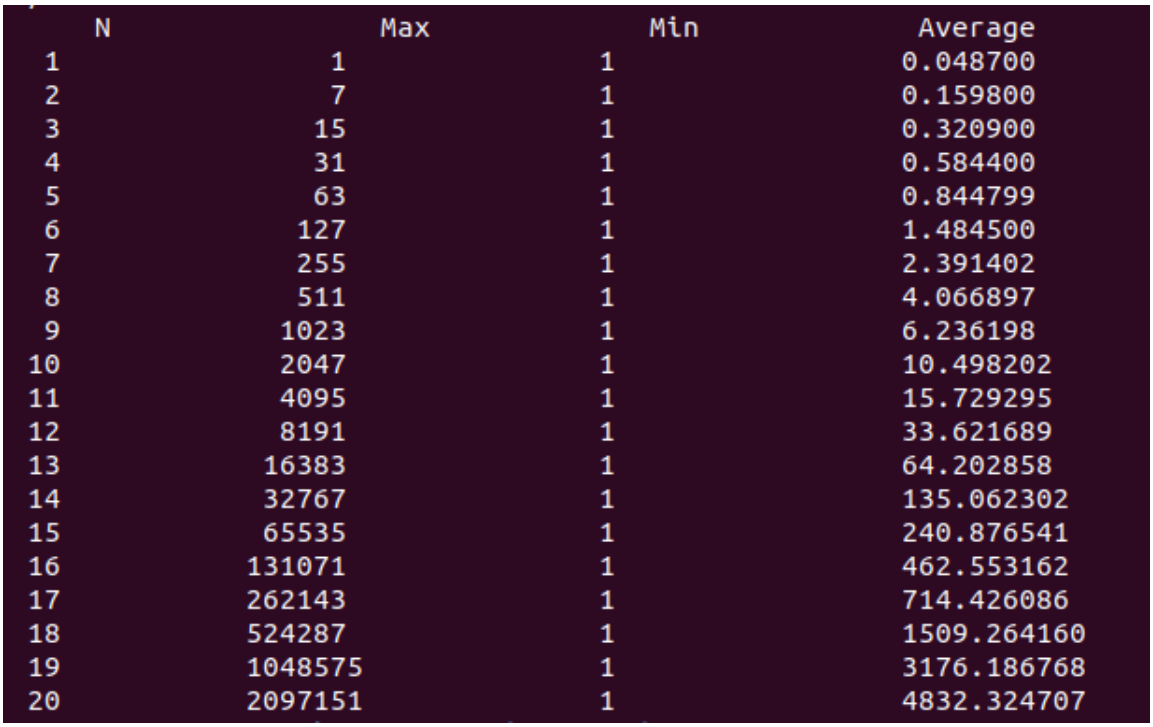
The following code is the knapsack function which also counts the number of times it is called:

```
static long knapsack(int *w, int W, int i, int N, int c){
/*Knapsack function with a count variable
to count the number of times it is being called*/
static long count;//variable to store the number of times the
function is being called
count=c;
count++;
if (W==0)
return count;
else if(W<0||i>N)
return -1*count;//this situation arises when
none of the weights sum up to the total weight
else if(knapsack(w,W-w[i],i+1,N,count)>0)//recursive call
 return count;
else
return knapsack(w,W,i+1,N,count);//recursive call if the
previous combination of weights does not add up
```

```
}
```

3.2 Results

The following table is generated in one of the runs of the code.



N	Max	Min	Average
1	1	1	0.048700
2	7	1	0.159800
3	15	1	0.320900
4	31	1	0.584400
5	63	1	0.844799
6	127	1	1.484500
7	255	1	2.391402
8	511	1	4.066897
9	1023	1	6.236198
10	2047	1	10.498202
11	4095	1	15.729295
12	8191	1	33.621689
13	16383	1	64.202858
14	32767	1	135.062302
15	65535	1	240.876541
16	131071	1	462.553162
17	262143	1	714.426086
18	524287	1	1509.264160
19	1048575	1	3176.186768
20	2097151	1	4832.324707

Figure 1: Table of counts

3.3 Scaling of the algorithm

Using a least squares fit algorithm I found out that if we plot the maximum count on the Y-axis and the N value on the X-axis, it is an exponential curve of the form $2e^{0.693n}$ which can also be written as $2 * 2^n$. Therefore the order of this algorithm is 2^n and T(n)(worst case running time) is $2 * 2^n$.

3.3.1 Justification for the scaling

Since there are N weights, the input is equal to the number of weights , i.e, n=N. Now for each of these weights, there are two possibilities, it can either belong to the subset(of weights adding up to W) or it may not belong to the subset. So for a worst-case scenario where none of the weights add up to W, we have to perform 2 checks for each w_i. Now since there are n weights, the scaling is 2^n.

4 Conclusions

- The knapsack algorithm is a very straight-forward way to determine the subset of weights that can sum up to a given total weight using recursion. The first part of my code does this and prints out the subset of weights.
- The second code analyses this algorithm in terms of the time taken or the worst case running time. We can call it the worst case running time because since the number of iterations is very high(10000) we can expect all possible situations to arise and hence the maximum number of times the function was called is a good measure of the worst case running time and thus we can compute the scaling and the order of the algorithm using this.
- The maximum number of counts can be modelled as $4 * 2^n$. Hence the scaling is said to be exponential of exponent 2.
- The minimum number of counts is always 1 because this arises when W=0 and this can be expected to occur atleast once for each N in our 10^4 iterations.