Algorithm Program Assignment 2

b97502049 電機四 蔡昀達

1. Write a brute force algorithm (just pseudo code) in your report that tried all possible combinations of x_i . What is the time complexity of this algorithm?
   Ans.
   a. To implement the burte force algorithm, I define three function to achieve the goal:

   bruteForce(): is used for the main program call the brute force method

```
If ( allCase.size() == 0) return 0
        else {
                int maxCase = 0
                for  i to allCase.size
                        if   totalValue(allCase[i])  > totalValue(allCase[maxCase])
                                maxCase = I
                report the result
                return the max value
        }
```

   candidate(buffer, n): this subrutine is use for finding all the solution of the knapsack problem in recursive way. Note that "items" is an STL vector to record all the item we can put into our knapsack, and n is the index of items which is being considerated at the current function stact.

```
if n < 0
        allCase.push_back(buffer)
        return
        candidate( buffer+'1', n-1 )
        candidate( buffer+'0', n-1 )
```

   totalSize(vector<Item>): to return the the total size of an "item" vector.
   TotalValue(vector<Item>): to return the the total value of an "item" vector.

   b. Considering my brute force algorithm, we can find out that all the routine is linear but for the **canditate()** is a recursive function. As a result, we only need to take the complexity of **canditate()** function call into account. Because the target of **canditate()** is to find all the combinations of the solution, and the total number of combination of items is 2^n (where n is number of the items), need to using recursive way to generate all the possible combination, and the depth of the recursion tree is n and each level we need to do 2^n times of operation. Hence, the time complexity is $\Theta(n\ 2^n)$.

2. Write a greedy algorithm (just pseudo code) in your report. What is the time complexity of this algorithm? Does your algorithm give the optimal solution?

   a. the pseudo code of greedy algorithm. Note that the items is an STL vector to store all the candidate of given items.

```
Sort the items array
for i  to items.size
        if  items[i].value == items[i+1].value && items[i].size > items[i+1].size
        continue
if remain space >= items[i].size
        remain space = remain space – items[i].size
        max value += items[i].value
return max value
```

   b. The time complexity analysis. We have already know that the complexity the best solution (Qsort)for sorting problem is O(nlgn), and the "for" loop is determinstic with the complexity O(n). Hence, we can conclude that the computing complexity is O(nlgn).

3. Let $\mu(i, s)$ equal to the maximum value that can be obtained by placing up to i items to the knapsack of size less than or equal to s. Show a recursive solution to this problem in your report.

   Ans.

   The recursive solution of optimal solution can be repersent as

$$\mu(i, s) = \max\{\mu(i-1, s) , \mu(i-1, s-w[i]) + v[i]\} \text{ if } s-w[i] \geq 0$$
$$= \mu(i-1, s) \text{ if } s – w[i] < 0$$

4. Write pseudo codes for the recursive algorithm in your report. Analyze the time complexity of this algorithm.

   a. The pseudo codes for the recursive algorithm

```
recursion():
construct the dynamic table
max value = RCFunction( items.size(), pack size)
while remain && x != 0
        while ( dp[x][remain]   == dp[x-1][remain] ) x--
        result[x-1] = 1
        remain = remain – items[x-1].szie
        x--
destruct the dynamic table
```

RCFunction(i , pSize):

```
if i == 0 || pSize == 0
        dp[i][pSize] = 0
if dp[i][pSize] < 0
        if pSize < item[i-1].size
                dp[i][pSize] = RCFunction( i – 1 , pSize )
        else
                dp[i][pSize] = max( RCFunction( i  – 1 , pSize ) ,
                        items[i-1].value + RCFuntion( i -1,  pSize – items[i-1].size )
return dp[i][pSize]
```

max(a, b):

return the larger one between a and b.

b . The time complexity of the recursive algorithm.

We can find out that the calling times of **RCFunction()** is dependent on the space of the knapsack and total number of items. Also, all the items can not be considered more then one times, and so as the space of the knapsack. Hence, we can conclude that the time complexity is $\Theta(nW)$, where the n is number of all items, and the W is the space of knapsack.

5. Write pseudo codes for the DP algorithm in your report. Analyze the time complexity of this algorithm. Hint: You can use a two dimensional table: one is item, the other is space.

   a. The pseudo codes for the dynamic programming algorithm.

```
construct DP table  dp
construct trace back table  traceBack
for i to items.size
        for j to space of package
                if i == 0 || j == 0
                        dp[i][j] = 0
                else if items[i-1].size <= j
                        dp[i][j] = max( items[i-1].value+dp[i-1][j-items[i-1].size]
                                        > dp[i-1][j] )
                        if the items is put in to dp[i][j], record in the trace back table
                else
                        dp[i][j] = dp[i-1][j]

        let remain = space of knapsack
        let x = items.size()
        while ( dp[items.size()][remain] – dp[items.size()][remain-1] == 0 ) remain—;
        while ( remain )
                while ( traceBack[x][remain] == 0 ) x--
                result[x-1] = 1;
                remain = remain – traceBack[x][remain]
                x--
        destruct the trace back table
        destruct the DP table
```

   b. The time complexity of dynamic programming algortihm.

   The dynaminc programming algorithm of solving the knapsack problem uses the iterative way to fill the table until we achieve the space of knapsack which is given. As a result, to fill up a 2D array, there are two nested loops in my implementation, the first nested loop is to fill the DP table and calculate the maxmium value of the problem.

   Considerating the index of this nested loop, we can conclude that the time complexity is $\Theta(nW)$ because the loop indexes is constrained by n and W, where n is number of items and W is the size of the knapsack.

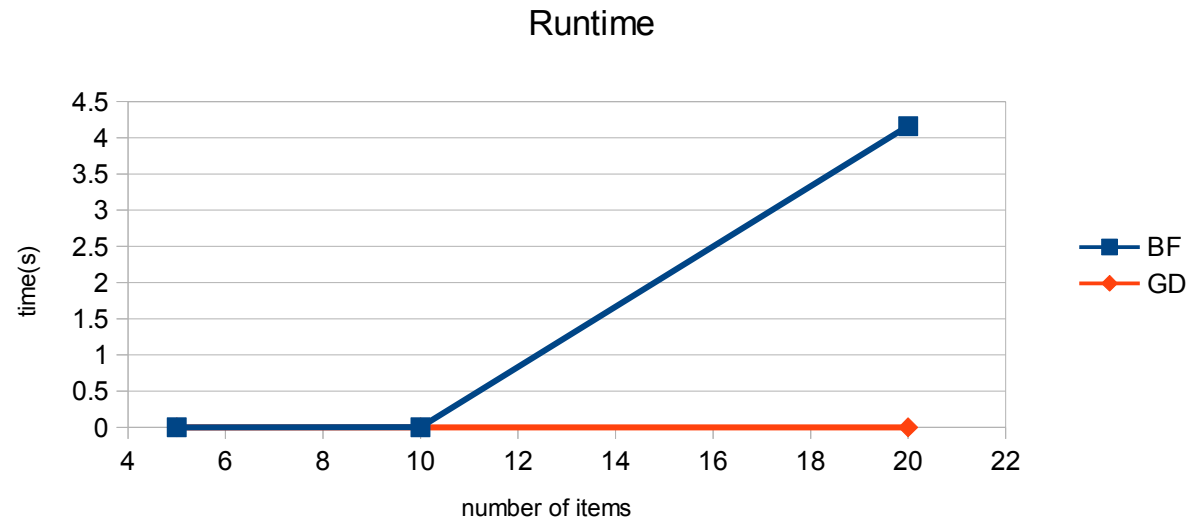
   Also, the second nested loop, which is used to trace back and find the combination of the optimized solution, is constrained by the size of knapsack and the number of all the items, because in my implementation, there is no entry in the **traceBack** table can be visited more then one time through out the trace procedure. From another point of view, we can not get the combination of optimized result until this procedure is been done. Hence, I can conclude that the time complexity of this part is $\Theta(nW)$.

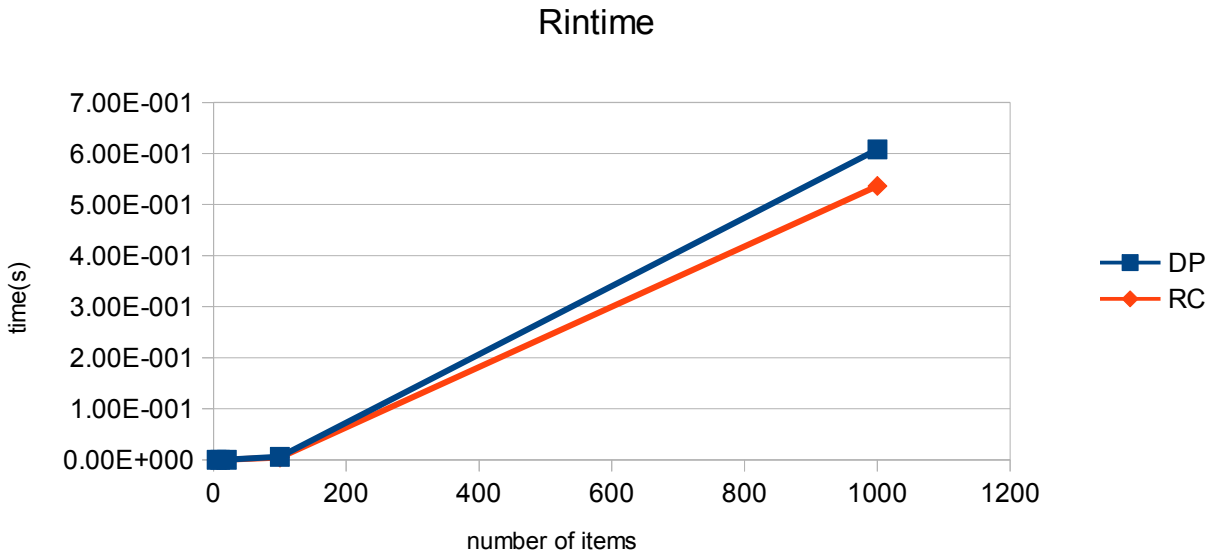
   Now, I can get the time complexity of the whole algorithm, which is $\Theta(nW)$.

6. The result table of testing input

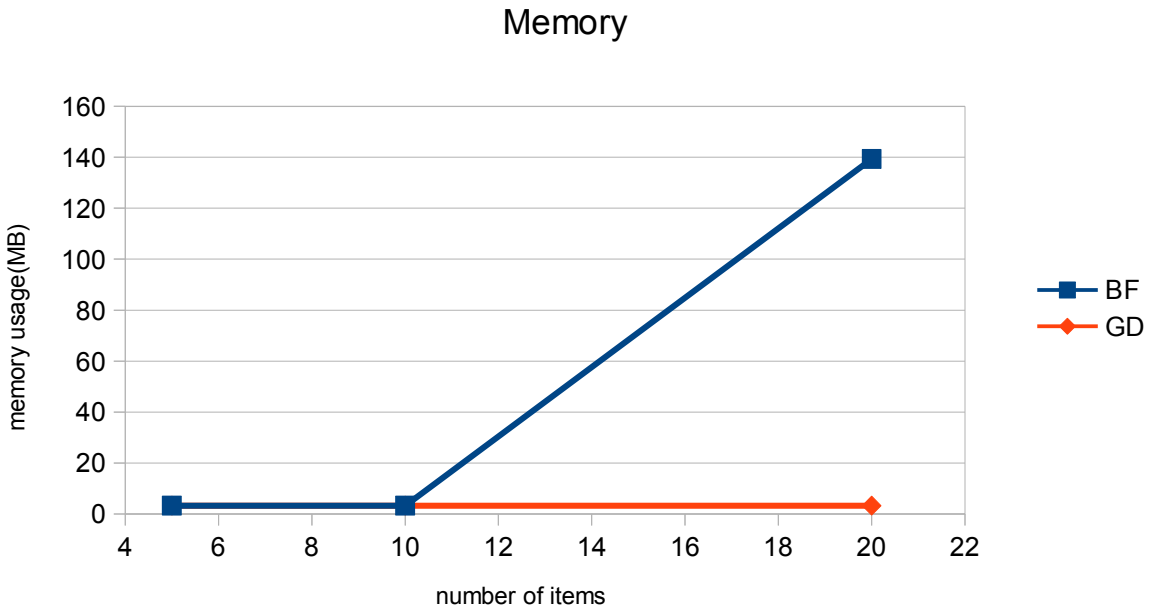
	bruteforce			greedy			dynamic programming			recursion		
input size	Max =	runtime(s)	memory(MB	Max =	runtime(s)	memory(MB	Max =	runtime(s)	memory(MB	Max =	runtime(s)	memory(MB
n5S11	54	0.000175	3.29297	54	7.10E-005	3.29297	54	7.30E-005	3.29297	54	7.10E-005	3.29297
n10S100	366	0.002917	3.29297	366	7.50E-005	3.29297	366	0.000134	3.29297	366	0.000101	3.29297
n20S100	483	4.16123	139.301	483	0.000112	3.29297	483	0.000197	3.29297	483	0.000173	3.29297
n100S1000	--	--	--	--	--	--	4729	0.006055	4.0625	4729	0.004522	3.67578
n1KS10K	--	--	--	--	--	--	45396	0.608071	79.7305	45396	0.53637	41.4258

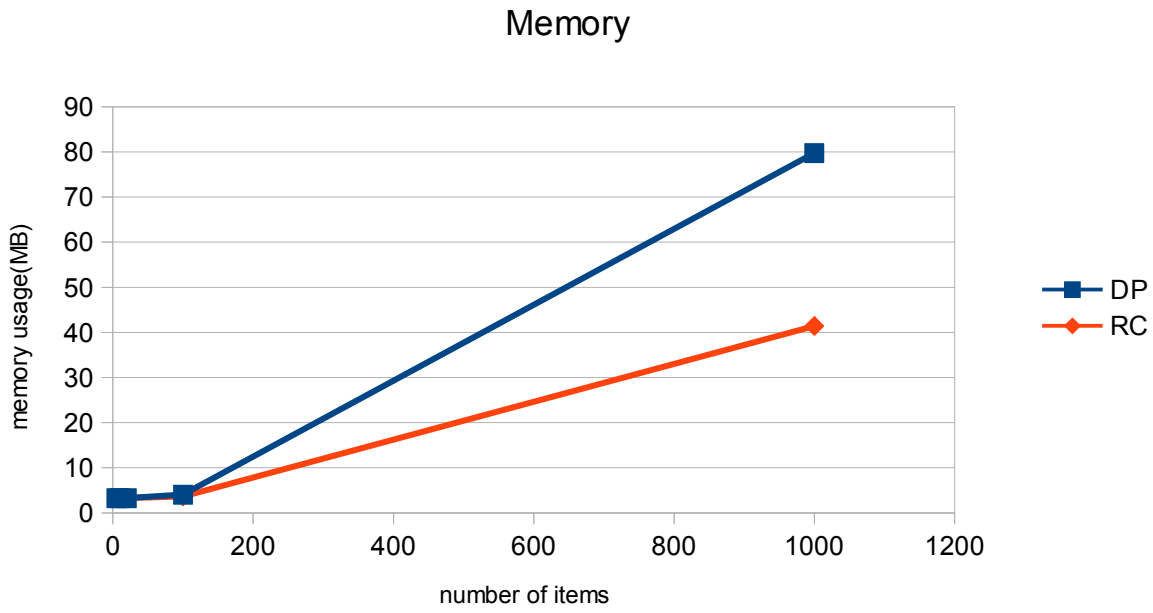
7. Ploting the table
   a. Running time:





b. Memory usage:





8. Analysis the result.
   According the computing result of my implementation, I would like to compare the theoratically prediction of time complexity and real value seperately.

   a. Brute force
   Time complexity: $\Theta(n\ 2^n)$
   Let the n is 5, 10 ,20
   where the ratio between Θ is on the order of 60, and 2000, on the other hand the ratio of experemental result is about 16 and 1426. As a result, the theoratically prediction and experemental result is on the same order, which the Θ is fit to the real implementation.

   Note that in my implementation, I use a vector string to store all the possible combination, but every character in this string can store either 0 or 1, which may wast the memory usage, so if I just use a intger variable and recording 0s and 1s combination by using bitwise operation can speed up the calculation and save the memory.

   b. Greedy
   Time complexity: O(nlgn)
   Let the n is 5, 10 ,20
   And the ratio between O is about 2.6, on the other hand the ration of real value is about 1 and 1.4, which is satisfise the constraing of O function.

   c. Dynamic programming
   Time complexity: $\Theta(nW)$
   Observing the experemental result, the ratio is always in the same order of n' ration, which means the time complexity is exactly linear.

   Note that in my implementation, I use another array to store the path when we applying the dynamic programming procedure, but the path information is not necessory to be stored in another array because we can determine the path we have traverse just by observing the origin dp table. Reduce this redundency, we can save more memory.

   d. Recursive
   Time complexity: $\Theta(nW)$
   Observing the experemental result, the ratio is always in the same order of n' ration, which means the time complexity is exactly linear.