Rajshahi University of Engineering and Technology
Department of Computer Science & Engineering

Lab Report 02
4th year Special Short Semester
Course No: CSE 2202

Course Title: Sessional Based on CSE 2201

Submitted To
Biprodip Pal
Assistant Professor
Department of CSE, RUET

Submitted by
Atiqur Rahman
Roll: 143059
Section: A
Department of CSE, RUET

TABLE OF CONTENT

1.1 NAME OF THE EXPERIMENT

Efficiency consideration for Knapsack problem implementation (0/1 and fractional).

1.2 MACHINE CONFIGURATION

OS name	:	Microsoft Windows 10 Pro
Version	:	10.0.18362 Build 18362
System Type	:	x64-based PC
Processor	:	Intel(R) Core(TM) i5-5200U CPU @ 2.20GHz, 2197 Mhz, 2 Core(s), 4 Logical Processor(s)
Installed Physical Memory(RAM)	:	8.00 GB
Total Physical Memory	:	7.90 GB
Total Virtual Memory	:	9.99 GB
Page File Space	:	2.08 GB

1.3 INTRODUCTION TO KNAPSACK PROBLEM

The **knapsack problem** or **rucksack problem** is a problem in combinatorial optimization: Given a set of items, each with a weight and a value, determine the number of each item to include in a collection so that the total weight is less than or equal to a given limit and the total value is as large as possible. It derives its name from the problem faced by someone who is constrained by a fixed-size knapsack and must fill it with the most valuable items.

1.4 IMPLEMENTATION USING BRUTE FORCE APPROACH

The very naive solution is to generate all subsets from the items and calculate the maximum profit. If we have n items than the time complexity will be $O(n * 2^n)$ which is very costly in time. We can compute knapsack for highest **20 to 25** items at most as our regular computer can execute 10^8 instruction in 1 second.

1.4.1 C++ IMPLEMENTATION

```
#include<bits/stdc++.h>
#include <chrono>
using namespace std;
using namespace std::chrono;
#define max_number_of_items 150000

int bruteforce(int profit[], int weight[], int number_of_items, int
knapsack_size)
{
    int max_profit = 0;
    int choice[number_of_items];
    memset(choice,0,sizeof(choice));
    for (int i=0; ; i++)
    {
        int j = number_of_items;
        int tempWeight = 0;
```

```
        int tempValue = 0;
        int k;
        k = 1;
        for (j = 0; j < number_of_items; j++)
        {
            choice[j] += k;
            k = choice[j] / 2;
            choice[j] = choice[j] % 2;
        }
        if (k)
            break;
        for (k = 0; k < number_of_items; k++)
        {
            if (choice[k] == 1)
            {
                tempWeight = tempWeight + weight[k];
                tempValue = tempValue + profit[k];
            }
        }
        if (tempValue > max_profit && tempWeight <= knapsack_size)
        {
            max_profit = tempValue;
        }
    }
    return max_profit;
}

int main()
{
    int knapsack_size,number_of_items;
    int profit[max_number_of_items];
    int weight[max_number_of_items];

    cout<<"Enter the knapsack size : ";
    cin>>knapsack_size;

    cout<<"Enter the number of items : ";
    cin>>number_of_items;

    freopen("dataset_ex_1.txt","r",stdin);
    for(int i = 0; i < number_of_items; i++)
        cin>>profit[i]>>weight[i];

auto start = chrono::high_resolution_clock::now();///Get starting timepoint
int res = bruteforce(profit, weight, number_of_items, knapsack_size);
auto stop = chrono::high_resolution_clock::now();///Get ending timepoint

    cout<<"MaximumPossible profit is :"<<res<<endl;

    double time_taken =  chrono::duration_cast<chrono::nanoseconds>(stop -
start).count();
    time_taken *= 1e-9;
    cout << "Time taken by program is : " << fixed  << time_taken <<
setprecision(9);
    cout << " sec" << endl;
    return 0;
}
```

1.5 IMPLEMENTATION OF 0/1 KNAPSACK USING GREEDY APPROACH

Greedy programming techniques are used in optimization problems. Choosing the items with as high value-to-weight ratios as possible and sorting by any advanced algorithm will take O(N log N) time complexity to compute the highest profit.

1.5.1 C++ IMPLEMENTATION

```
#include<bits/stdc++.h>
#include <chrono>
using namespace std;
using namespace std::chrono;
#define max_number_of_items 150000

struct item
{
    int item_profit, item_weight;
    item(int p = 0, int w = 0)
    {
        item_profit = p;
        item_weight = w;
    }

    bool operator < (item &ob)const
    {
        double r1 = (double)item_profit / item_weight;
        double r2 = (double)ob.item_profit / ob.item_weight;
        return r1 > r2;
    }
};

int greedy_knapsack(item item_list[], int number_of_items, int knapsack_size)
{
    sort(item_list, item_list + number_of_items); /// main time complexity
here (n log n)

    int current_weight = 0;
    int max_profit = 0;

    for(int i = 0; i<number_of_items; i++)
    {
        if(current_weight + item_list[i].item_weight <= knapsack_size)
        {
            current_weight += item_list[i].item_weight;
            max_profit += item_list[i].item_profit;
        }
    }
    return max_profit;
}

int main()
{
    int knapsack_size, number_of_items, p, w;
    item item_list[max_number_of_items];

    cout<<"Enter the knapsack size : ";
    cin>>knapsack_size;
```

```
    cout<<"Enter the number of items : ";
    cin>>number_of_items;

    freopen("dataset_ex_1.txt","r",stdin);
    for(int i = 0; i < number_of_items; i++)
    {
        cin>>p>>w;
        item ob(p, w);
        item_list[i] = ob;
    }

auto start = chrono::high_resolution_clock::now(); ///Get start timepoint
int res = greedy_knapsack(item_list, number_of_items, knapsack_size);
auto stop = chrono::high_resolution_clock::now();  /// Get ending timepoint

    cout<<"MaximumPossible profit is :"<<res<<endl;

    double time_taken =  chrono::duration_cast<chrono::nanoseconds>(stop -
start).count();
    time_taken *= 1e-9;
    cout << "Time taken by program is : " << fixed  << time_taken <<
setprecision(9);
    cout << " sec" << endl;

    return 0;
}
```

1.6 IMPLEMENTATION OF FRACTIONAL KNAPSACK USING GREEDY APPROACH

Greedy programming techniques are used in optimization problemsChoosing the items with as high value-to-weight ratios as possible and sorting by any advanced algorithm will take O(N log N) time complexity to compute maximum profit.

1.6.1 C++ IMPLEMENTATION

```
#include<bits/stdc++.h>
#include <chrono>
using namespace std;
using namespace std::chrono;
#define max_number_of_items 150000

struct item
{
    int item_profit, item_weight;
    item(int p = 0, int w = 0)
    {
        item_profit = p;
        item_weight = w;
    }

    bool operator < (item &ob)const
    {
        double r1 = (double)item_profit / item_weight;
        double r2 = (double)ob.item_profit / ob.item_weight;
        return r1 > r2;
    }
};
```

```
double greedy_knapsack(item item_list[], int number_of_items, int
knapsack_size)
{
    sort(item_list, item_list + number_of_items); /// main time complexity
here (n log n)

    int current_weight = 0;
    double max_profit = 0.0;

    for(int i = 0; i<number_of_items; i++)
    {
        if(current_weight + item_list[i].item_weight <= knapsack_size)
        {
            current_weight += item_list[i].item_weight;
            max_profit += item_list[i].item_profit;
        }
        else
        {
            int remain = knapsack_size - current_weight;
            max_profit += item_list[i].item_profit * ((double) remain /
item_list[i].item_weight);
            break;
        }
    }
    return max_profit;
}

int main()
{
    int knapsack_size, number_of_items, p, w;
    item item_list[max_number_of_items];

    cout<<"Enter the knapsack size : ";
    cin>>knapsack_size;

    cout<<"Enter the number of items : ";
    cin>>number_of_items;

    freopen("dataset_ex_1.txt","r",stdin);
    for(int i = 0; i < number_of_items; i++)
    {
        cin>>p>>w;
        item ob(p, w);
        item_list[i] = ob;
    }

auto start = chrono::high_resolution_clock::now(); /// Get star timepoint
double res = greedy_knapsack(item_list, number_of_items, knapsack_size);
auto stop = chrono::high_resolution_clock::now();  /// Get ending timepoint

    cout<<"MaximumPossible profit is :"<<res<<endl;

    double time_taken =  chrono::duration_cast<chrono::nanoseconds>(stop -
start).count();
    time_taken *= 1e-9;
```

```
    cout << "Time taken by program is : " << fixed  << time_taken <<
setprecision(9);
    cout << " sec" << endl;

    return 0;
}
```

1.7 EXPERIMENTAL ANALYSIS

I randomly generated up to 150000 data items as the dataset to test the previous implementations time complexities. I used a common main function and the high_resolution_clock of chrono library to compute the execution time.

1.7.1 EXPERIMENTAL RESULTS

The codes have executed with different number of random data and calculated the execution time accordingly. Times are calculated in second unit.

Table 1.1: Experimental Result (Time required(sec)). [Capacity set to 900000000]

Data	Brute Force, O(n*2^n)	Greedy 0/1 knapsack, O(N log N)	Greedy Fractional knapsack, O(N log N)
20000	∞	0.015607 sec	0.015626 sec
30000	∞	0.015624 sec	0.015645 sec
100000	∞	0.046894 sec	0.046876 sec
150000	∞	0.078103 sec	0.062499 sec

1.8 CONCLUSION

The brute force approach has O(n*2^n) time complexity and applicable up to 20-25 data items only. But, greedy approach to choice items with highest profit/weight ratio has O(n log n) time complexity and worked nicely with the datasets.