106119029 , Algos Lab 7(Knapsack Problem)

Dipesh Kafle

1) Problem Statement

The knapsack 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

- 0 1 Knapsack Problem : Given a knapsack which can hold maximum weight of MAX_WT, we are presented with a list of weights and values of n objects. The objective is to choose objects to put in the knapsack that maximises the cost. Here the objects are indivisible and hence can only either be included or excluded.
- Fractional Knapsack Problem : Same as the above, except the objects are divisible, so fraction of objects can also be put in the knapsack.

2) Fractional Knapsack Problem

- We can solve the Fractional Knapsack Problem with Greedy Heuristics.
- We create an array of pairs of value and its corresponding weight and sort(descending) it using the ratio `value / weight` as the comparator. This will give basically give us the cost per unit weight value for every object and we then select the objects greedily and obtain our optimal answer.
- The complexity of this method is O($nlogn$) i.e the time taken to sort the array is the most work we do. Selecting the object given the sorted array is a linear time task.

Code

```
void fractionalKnapsack(vector<pair<int, int>> &v_w, int MAX_WT, double *ans) {
  sort(v_w.begin(), v_w.end(), [](auto x, auto y) {
    return (((double)(x.first)) / x.second > ((double)y.first) / y.second);
  });

  int cur_weight = 0;
  double final_value = 0;
  for (pair<int, int> x : v_w) {
    if (cur_weight + x.second <= MAX_WT) {
      cur_weight += x.second;
      final_value += (x.first);
    } else {
      int remaining_wt = MAX_WT - cur_weight;
      final_value += (((double)(remaining_wt * x.first)) / x.second);
      break;
    }
  }
  *ans = final_value;
}
```

3) 0-1 Knapsack Problem

- The best way of solving this is through dynamic programming. We can also use a brute force method inorder to solve it but that would 2^n possible combinations and would have $O(2^n)$ complexity as well. In brute force method, we would at every element take two things into consideration(Should this be included or not) . We would then compute the value generated by taking or not taking the item and take max of that.

```
[a,b,c], X implies done


             a
      Y/          \N
      b             b
    Y/ \N       Y/ \N
    c    c       c     c
  Y/\N Y/\N    Y/\N Y/\N
  X   X X   X    X   X X   X
```

  - In brute force method, it would come out something like this. We would need to find the cost of every single path from root to X and check its cost and see if it is optimized.

- We can however solve this in much faster time using dynamic programming.

Observation

- The dynamic programming solution is based on the observation that the max value we can obtain is going to be max(value by not including this current object, value by including this object)

- $m[0,\, w] = 0$
- $m[i,\, w] = m[i-1,\, w]$ if $w_i > w$ (the new item is more than the current weight limit)
- $m[i,\, w] = \max(m[i-1,\, w],\, m[i-1, w-w_i] + v_i)$ if $w_i \leqslant w$.

- We keep track of these values using a table of n rows and MAX_WT+1 columns.
- We will then fill the table based on the above recursive definition from left to right and up to down.
- Thus the dynamic programming solution to this problem is a simple O(n*MAX_WT) running time algorithm.

Code

```
void zeroOneKnapsack(const vector<int> &weights, const vector<int> &values,
                     vector<vector<int>> &dp) {
  int MAX_WT = dp[0].size() - 1;
  int ind = 0;
  for (int &x : dp[0]) {
    if (weights[0] > ind) {
      x = 0;
    } else {
      x = weights[0];
    }
    ind++;
  }
  for (size_t i = 1; i < weights.size(); i++) {
    for (int j = 0; j <= MAX_WT; j++) {
      if (weights[i] > j) {
        dp[i][j] = dp[i - 1][j];
      } else {
        dp[i][j] = max(dp[i - 1][j], dp[i - 1][j - weights[i]] + values[i]);
      }
    }
  }
}
```

Whole program Code

```cpp
#include <algorithm>
#include <chrono>
#include <cmath>
#include <fstream>
#include <functional>
#include <iomanip>
#include <iostream>
#include <numeric>
#include <string>
#include <unordered_map>
#include <vector>

using namespace std;

template <typename Func, typename... Args>
double timeMyFunction(Func func, Args &&...args) {
  auto start_time = std::chrono::steady_clock::now();
  func(forward<Args>(args)...);
  auto end_time = std::chrono::steady_clock::now();
  std::chrono::duration<double> elapsed_time =
      std::chrono::duration_cast<std::chrono::duration<double>>(end_time -
                                                                start_time);
  return elapsed_time.count();
}
```

```cpp
void zeroOneKnapsack(const vector<int> &weights, const vector<int> &values,
                     vector<vector<int>> &dp) {
  int MAX_WT = dp[0].size() - 1;
  int ind = 0;
  for (int &x : dp[0]) {
    if (weights[0] > ind) {
      x = 0;
    } else {
      x = weights[0];
    }
    ind++;
  }
  for (size_t i = 1; i < weights.size(); i++) {
    for (int j = 0; j <= MAX_WT; j++) {
      if (weights[i] > j) {
        dp[i][j] = dp[i - 1][j];
      } else {
        dp[i][j] = max(dp[i - 1][j], dp[i - 1][j - weights[i]] + values[i]);
      }
    }
  }
}
```

```cpp
void fractionalKnapsack(vector<pair<int, int>> &v_w, int MAX_WT, double *ans) {
  sort(v_w.begin(), v_w.end(), [](auto x, auto y) {
    return (((double)(x.first)) / x.second > ((double)y.first) / y.second);
  });

  int cur_weight = 0;
  double final_value = 0;
  for (pair<int, int> x : v_w) {
    if (cur_weight + x.second <= MAX_WT) {
      cur_weight += x.second;
      final_value += (x.first);
    } else {
      int remaining_wt = MAX_WT - cur_weight;
      final_value += (((double)(remaining_wt * x.first)) / x.second);
      break;
    }
  }
  *ans = final_value;
}
```

```cpp
int main(int argc, char **argv) {
  ofstream zeroOne("ZeroOne.txt");
  ofstream frac("Fractional.txt");
  srand(time(0));
  int size_opt;
  if (argc == 2) {
    size_opt = atoi(argv[1]);
  } else {
    size_opt = 0;
  }
  double time_elapsed;
  for (int size :
       {10,  20,  30,  40,  50,  60,  70,  80,  90,  100, 110, 120, 130,
        140, 150, 160, 170, 180, 190, 200, 210, 220, 230, 240, 250}) {
    int MAX_WT;
    if (size_opt == 0)
      MAX_WT = size;
    else if (size_opt == 1)
      MAX_WT = size * size;
    else if (size_opt == 2)
      MAX_WT = log2(size);
    else
      MAX_WT = size * (rand() % size);
    double fractional_ans;
    vector<int> values(size);
    vector<int> weights(size);
    // for fractional knapsack
    vector<pair<int, int>> v_w;
    // for zeroOneKnapsack
    vector<vector<int>> dp(weights.size(), vector<int>(MAX_WT + 1, INT32_MIN));
    generate(values.begin(), values.end(), []() { return rand() % 1000; });
    generate(weights.begin(), weights.end(),
             [size]() { return rand() % size; });
    transform(values.begin(), values.end(), weights.begin(), back_inserter(v_w),
              [](auto x, auto y) { return make_pair(x, y); });
    time_elapsed =
        timeMyFunction(zeroOneKnapsack, ref(weights), ref(values), ref(dp));
    cout << "DP:" << size << "," << MAX_WT << ":" << fixed << setprecision(20)
         << time_elapsed << ":";
    cout << dp.back().back() << endl;
    zeroOne << size << ":" << MAX_WT << ":" << fixed << setprecision(20)
            << time_elapsed << endl;
    time_elapsed =
        timeMyFunction(fractionalKnapsack, ref(v_w), MAX_WT, &fractional_ans);
    cout << "Fractional:" << size << "," << MAX_WT << ":" << fixed
         << setprecision(20) << time_elapsed << ":";
    cout << fractional_ans << endl;
    frac << size << ":" << MAX_WT << ":" << fixed << setprecision(20)
         << time_elapsed << endl;
    cout << endl << endl;
  }
}
```

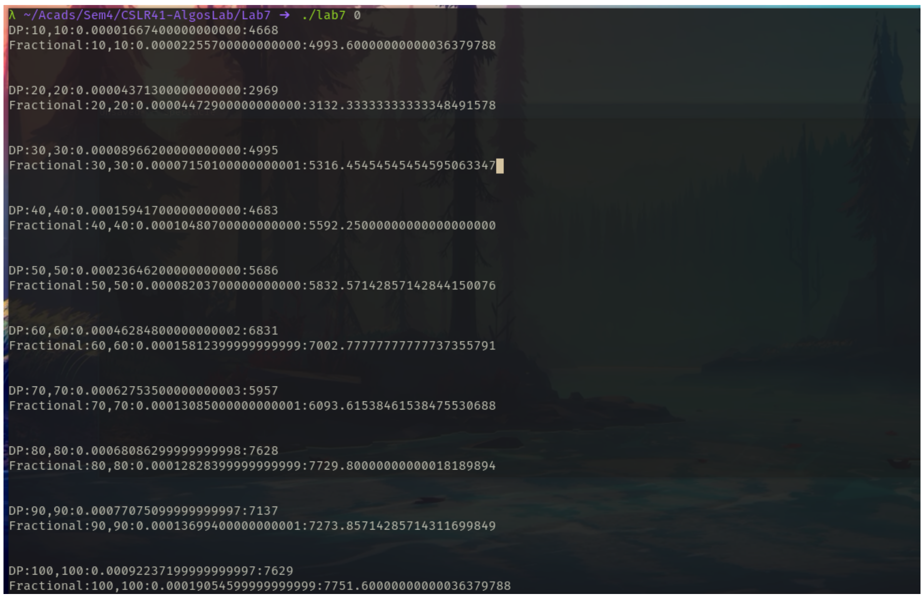
4) Analysis

- Since these two arent fundamentally the same problem(because of one being fractional and another not). We cant compare them directly. So I have made it so that I can give different values of MAX_WT(depending on n) to closely look at their differences.
- The fractional knapsack problem is independent of MAX_WT values but 0 1 knapsack problems isnt.
- Running time of 01 knapsack is O(n*MAX_WT)

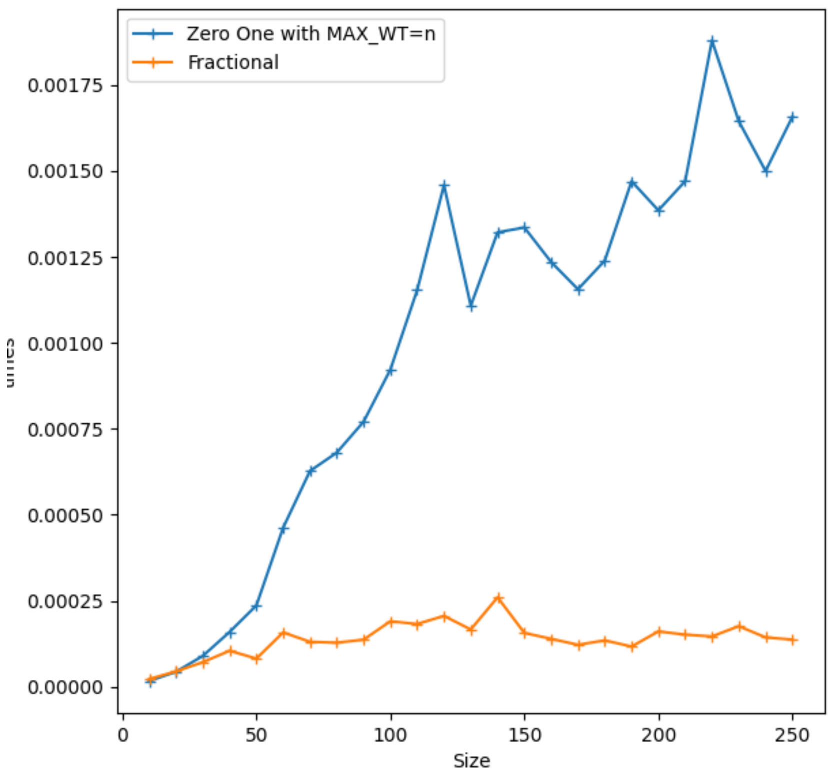
- I have 3 cases of MAX_WT.
  - When MAX_WT = n , will lead to c1*n*n -> c1*n^2 running time
  - When MAX_WT = n^2 , will lead to c2*n*n^2 -> c2* n^3 running time
  - When MAX_WT = logn, will lead to c3*n*logn running time

When MAX_WT = n

- Output

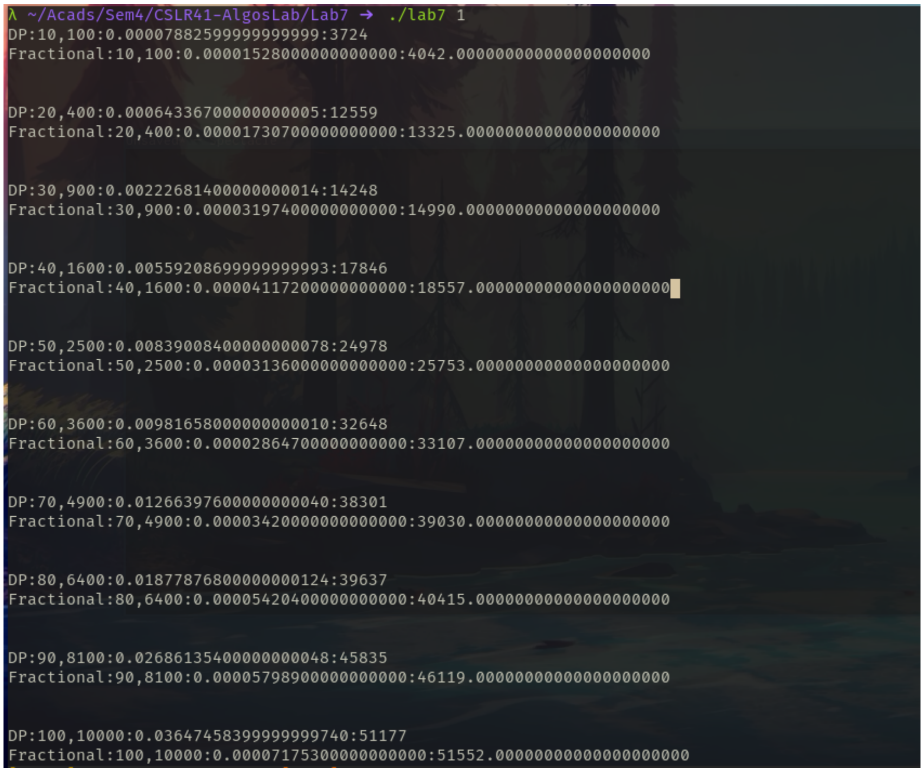


- Comparision plot (c1*nlogn vs c2*n^2)



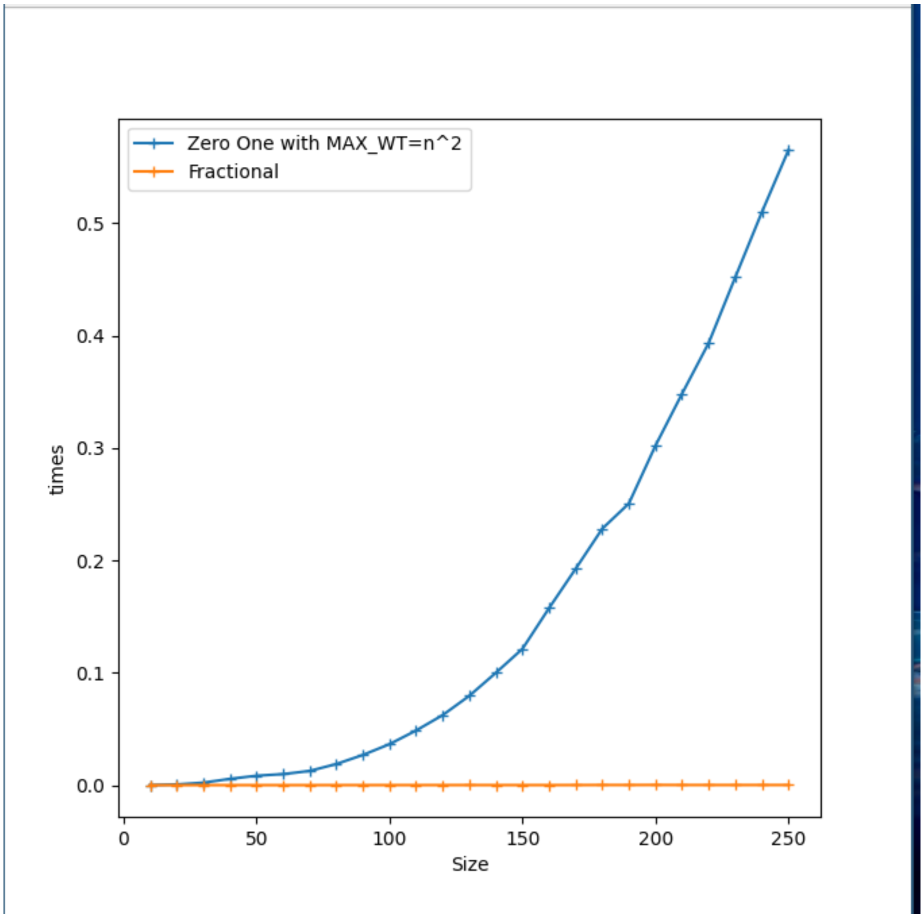
When MAX_WT = n^2

- Output



```
λ ~/Acads/Sem4/CSLR41-AlgosLab/Lab7 → ./lab7 1
DP:10,100:0.00007882599999999999:3724
Fractional:10,100:0.00001528000000000000:4042.00000000000000000000


DP:20,400:0.00064336700000000005:12559
Fractional:20,400:0.00001730700000000000:13325.00000000000000000000


DP:30,900:0.00222681400000000014:14248
Fractional:30,900:0.00003197400000000000:14990.00000000000000000000


DP:40,1600:0.00559208699999999993:17846
Fractional:40,1600:0.00004117200000000000:18557.00000000000000000000


DP:50,2500:0.00839008400000000078:24978
Fractional:50,2500:0.00003136000000000000:25753.00000000000000000000


DP:60,3600:0.00981658000000000010:32648
Fractional:60,3600:0.00002864700000000000:33107.00000000000000000000


DP:70,4900:0.01266397600000000040:38301
Fractional:70,4900:0.00003420000000000000:39030.00000000000000000000


DP:80,6400:0.01877876800000000124:39637
Fractional:80,6400:0.00005420400000000000:40415.00000000000000000000


DP:90,8100:0.02686135400000000048:45835
Fractional:90,8100:0.00005798900000000000:46119.00000000000000000000


DP:100,10000:0.03647458399999999740:51177
Fractional:100,10000:0.00007175300000000000:51552.00000000000000000000
```

- Comparision plot (c*nlogn vs c2*n^3)

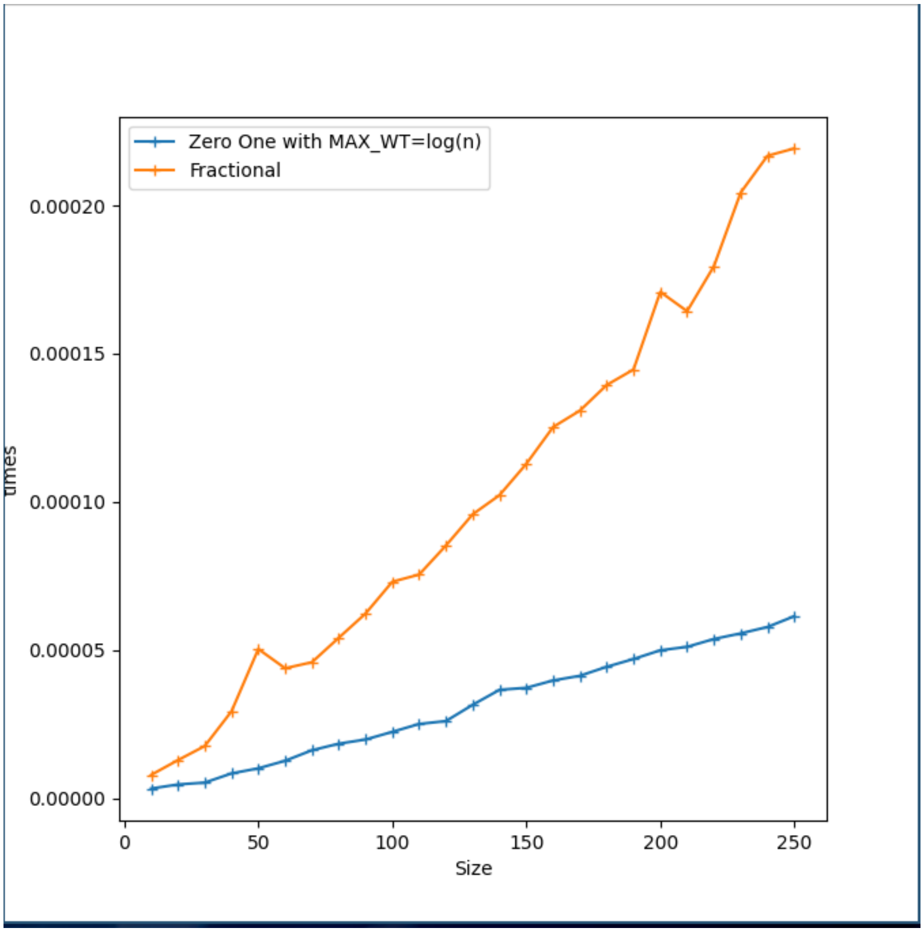


When MAX_WT = logn

- Output

```
λ ~/Acads/Sem4/CSLR41-AlgosLab/Lab7 ➔ ./lab7 2
DP:10,3:0.00000508500000000000:598
Fractional:10,3:0.00001265500000000000:1126.50000000000000000000


DP:20,4:0.00000944900000000000:1670
Fractional:20,4:0.00003497400000000000:1832.00000000000000000000


DP:30,4:0.00001084000000000000:984
Fractional:30,4:0.00003838000000000000:984.00000000000000000000


DP:40,5:0.00001282800000000000:1579
Fractional:40,5:0.00003543400000000000:1579.00000000000000000000


DP:50,5:0.00001289500000000000:634
Fractional:50,5:0.00004653800000000000:925.66666666666674245789


DP:60,5:0.00001495200000000000:1929
Fractional:60,5:0.00005994200000000000:1993.70000000000004547474


DP:70,6:0.00001983700000000000:1363
Fractional:70,6:0.00006741500000000000:1412.00000000000000000000


DP:80,6:0.00002242600000000000:3894
Fractional:80,6:0.00007406400000000000:4112.33333333333303016843


DP:90,6:0.00002432700000000000:1260
Fractional:90,6:0.00008911800000000000:1601.33333333333348491578


DP:100,6:0.00002685600000000000:2979
Fractional:100,6:0.00009137700000000000:3271.33333333333348491578
```

- Comparision plot (c1*nlogn vs c2*nlogn)



Remarks

- Obviously the 0 1 knapsack graph is affected by large amount with changes in MAX_WT values and they are comparable in case when MAX_WT=logn and differ only by some constant factor.
- These two algorithms in all fairness shouldnt be compared as they are fundamentally very different algorithms that do not serve the same purpose(one works in fractional case while another in 0 1 case).