EE4371
Introduction to Data Structures and Algorithms
End Term Report

Author: Rahul Chakwate
AE16B005

10th December 2019

2 Fluid Particles Simulation

2.1 Brute Force Algorithm for all particle's interaction

In this algorithm, every particle interacts with every other particle. While computing the neighbors all particles are considered. The pseudo code for this algorithm is given below.

Pseudo Code

```
define structure Particle {
    double x,y; // particle position
    double vx,vy; // particle velocity
} p[MAX_PARTICLES];
double radius = 0.001 // radius of interacting neighbors
double dt = 0.0001 // smallest time scale of the simulation
Function getneighborindex(integer i, double radius):
    // This function takes in particle index and radius as input and returns indices of all neighbors
     within the given radius.
    integer number_of_neighbors = 0 // store number of neighbors
    integer nbidx[1000] // stores indices of the neighbor particles
    for k in 0 to MAX_PARTICLES do
        double dist = euclidean_distance(i,k)
        if dist = radius then
            nbidx[number_of_neighbors] = k
            number_of_neighbors++
    end
return nbidx[], number_of_neighbors
Function Main():
    set p.x and p.y randomly using builtin rand() function.
    For half the particles, set p.vx and p.vy as zero and for rest half as unit vector in random
     direction.
    for t from 0 to MAX_TIME do
        for i from 0 to MAX_PARTICLES do
            integer nbidx[] = getneighboridx(i,radius) // gives an array of neighbor index and
             number of neighbors found.
            for k from 0 to number_of_neighbors do
                particle pn = p[nbidx[k]] // define pn as neighbor particle
                double denom = pow(euclidean_distance(i,nbidx[k]),3) // computes euclidean
                 distance between p[i] and pn raised to power of 3
                double Fx = (p[i].x - pn.x) / denom
                double Fy = (p[i].y - pn.y) / denom // computes the force by neighbor pn on
                 particle pi
                p[i].vx += Fx*dt
                p[i].vy += Fy*dt // update velocity
            end
        end
        for i from 0 to MAX_PARTICLES do
            p[i].x += p[i].vx*dt
            p[i].y += p[i].vy*dt // update positions
        end
    end
return 0
```

Estimated time requirement:

Number of Particles (p) = 10^8

Calculations to compute neighbors for each particle = p = 10^8

Overall calculations = $p^2 = 10^{16}$

CPU time to compute a floating point operation = 10 nSec = $10^{-8} Sec.$

Let number of floating point operations in a single timestamp = K

Therefore, time of computation of one time stamp in seconds (T_0) is:

$$T_0 = K * 10^{16} * 10^{-8} = K * 10^8 sec \tag{1}$$

Number_of_timestamps = 10^4

$$Totaltimerequirement = K * 10^{12} sec \approx 30,000 years \tag{2}$$

2.2 4-nary Tree based Algorithm

This algorithm uses abstract data structure called trees, specifically 4-nary tree. The **abstract algorithms** is explained in words below.

- The particle's xposition and yposition are converted into a **10 bits** binary number.
- Negetive positions are given by the **2's complement**.
- Starting from the first bit (MSB), each particle is placed into one of the four nodes of the tree by checking MSB of xposition and yposition as **00,01,10 or 11.**
- This process is continued for all bits and a tree with **depth = 10** is created.
- each of the particle forms a leaf node for **one of the 1024 * 1024** tree nodes.
- **Geometrically**, this can be viewed as the $2 * 2$ box divided into $2^{10} * 2^{10}$ grid. The particles lying in the same grid forms neighbors for other particles in the grid.

Pseudo Code:

The ***Main()*** program remains the same as above with calls to the utility functions ***createtreenode(), updatetree()*** and ***getneighborindex()***. The tree data structure and the utility functions are described in the code below.

```
define structure leafnode {
    integer particleidx; // index of the particle in the array.
} leafnode;
define structure treenode {
    integer numleaves; // number of leaves/children the node contains
    struct treenode *child00;
    struct treenode *child01;
    struct treenode *child10;
    struct treenode *child11; //the for possible children of the treenode which are non leaf nodes
    struct leafnode *leaves[1000] // leafnodes containing the particle indices
} treenode;
```

Function `createtreenode()`**:**
- // This function allocates the memory to the new_node equivalent to the size of the treenode struct and initializes the node.
- **struct treenode** **new_node*
- new_node = Assign memory using malloc();
- new_node→ *child*00 = *NULL*
- new_node→ *child*01 = *NULL*
- new_node→ *child*10 = *NULL*
- new_node→ *child*11 = *NULL*
- new_node→ *numleaves* = 0

return *new_node pointer*

Function `gotochild`**(struct treenode** **current_node,* **integer** *bx*,**integer** *by*)**:**
- // This function checks the binary bit of x (bx) and y (by) and points current_node pointer towards the corresponding child. If child is not present, it creates a new child.
- **switch** *2*bx+by* **do**
  - **case** *0* **do**
    - **if** *child00 not present* **then**
      - call createtreenode()
    - current_node = current_node→ *child*00
  - **end**
  - **case** *1* **do**
    - **if** *child01 not present* **then**
      - call createtreenode()
    - current_node = current_node→ *child*01
  - **end**
  - **case** *2* **do**
    - **if** *child10 not present* **then**
      - call createtreenode()
    - current_node = current_node→ *child*10
  - **end**
  - **case** *3* **do**
    - **if** *child11 not present* **then**
      - call createtreenode()
    - current_node = current_node→ *child*11
  - **end**
- **end**

return *current_node pointer*

Function `updatetree`**(struct treenode** **root*)**:**
- // This function builds the tree and add particles as its leaves.
- int bx[BITS], by[BITS]; // stores the binary representation of particle k's position.
- **for** *k from 0 to MAX_PARTICLES* **do**
  - bx = dec_to_bin(p[k].x)
  - by = dec_to_bin(p[k].y) //dec_to_bin converts signed fractional decimal into binary by multiplying the number by 2 and taking out the MSB everytime. For negetive numbers it computes the 2's complement.
  - current_node = root
  - **for** *i form 0 to BITS* **do**
    - current_node = gotochild(current_node, bx[i], by[i]) //gives pointer to appropriate child
  - **end**
  - addleaf(current_node,k) // this function creates leafnode and updates its data, same as createtreenode() function for leaf.
- **end**

return *root*

Function `getneighborindex(integer` k, **struct treenode** *$root$*`):`

```
// This function takes in the root of the tree made by the updatetree() function and the current
  particle index (k), searches the location of k in the tree and computes the number of leaves of
  its parent which is same as its number of neighbors and returns the neighbor indices and the
  number of neighbors.
int bx[BITS], by[BITS]; // stores the binary representation of particle k's position.
bx = dec_to_bin(p[k].x)
by = dec_to_bin(p[k].y) .
current_node = root
for i form 0 to BITS do
    current_node = gotochild(current_node, bx[i], by[i]) //gives pointer to appropriate child
end
int nbidx[] // stores indices of neighbors number_of_children = current_node→ numleaves
for k from 0 to number_of_children do
    nbidx[count] = current_node → leaves → particleidx
    count++
end
```

return *nbidx[], number_of_neighbors*

Time Complexity:

Let number of particles = p, number of boxes = b and the number of bits used be BITS.

Number of boxes b = 4^{BITS}.

In the 4-nary tree for every particle,

- search operation = O($\log b$)
- insert operation = O($\log b$)

Hence, to update the tree with every particle,

update operation = O($p * \log b$)

Number of neighbors = O(p/b),

Again, for every particle,

- To find the neighbours, search operation is required = O($\log b$)$To compute forces from the neighbors = \mathcal{O}$(p/b)

Hence to update all particles forces = O($p * (\log b + p/b)$) = O($p \log b + p^2/b$)

$$TotalComplexity = \mathcal{O}(\frac{p^2}{b} + cp \log b) \tag{3}$$

Let the number of float operation in each iteration be K.

For the given problem, p = 10^8, b = $10^6 = 2^{10} * 2^{10}$

Hence, for a single time iteration,

the total time complexity = $K * \mathcal{O}(10^{16}/10^6 + 10^9) = K * 10^{10}$

which is much lesser than the brute force method complexity = $K * 10^{16}$

Neglect of integer operations is warranted because integer operations are carried parallel to the floating point operations which usually take more time than the integer operation.

2.3 Code Implementation and Experiments

Parameters used while implementing are as follows.

- Number of Particles = 10^6
- Number of BITS = 7. Therefore number of grids = $2^7 * 2^7$
- time interval = 10^{-4} secs.
- total number of times = 1000
- Force Multiplication Factor = 10^{-4}
- number of bins = 100
- binsize = 0.1

Reasons for choosing the above values:

Assuming CPU speed to be 10nSec, the original configuration will take about K * 100 secs for a single iteration, which is very large if K is large or number of time stamps is large. Hence, the problem is scaled down to the above configurations.

Number of Particles: Using the default 10^8 particles lead to insufficient RAM issues. Also the time complexity increases. Hence 10^6 is chosen as optimum.

Number of BITS: Since particles are 10^6, using 10 BITS will create 10^6 grid regions which means 1 point will be in each region which makes the problem of interacting particles impractical. To keep about 100 neighbors, $10^4 \approx 2^7 * 2^7$ regions are required. Hence BITS=7 is optimum.

Force Multiplication Factor: The force needs to be scaled down as the magnitude of force blows to large values. The reason is as follows.

$$Fx = \frac{x}{\{x^2 + y^2\}^{3/2}} \approx \frac{1}{x^2} \tag{4}$$

Average distance between particles is

$$x = \frac{2}{10^3} \approx 10^{-3} \tag{5}$$

Therefore, force from each particle is

$$Fx \approx 10^6 \tag{6}$$

But if the particle comes closer, x can be even lesser causes.

In the experiments, it is observed the force to have larger values.

This results in large velocities and displacements which causes the particle to rebound many times from the wall.

Solution to this? Either decrease the force by a constant factor or reduce the time interval sufficiently to capture the intermediate interactions.

binsize: In experiments, it is observed that particle velocities are concentrated near zero but some particles have very large velocities.

If

$$binsize = \frac{max(velocities) - min(velocities)}{100} \tag{7}$$

is used, then no noticeable segregation of particles is observed. Hence the bins are formed near zero to capture the fine variation of most particle velocities near the origin.

2.4 Results

Variation of execution time with parameters

Final Implementation has the above chosen optimum parameters.

Its **Time of Execution = 4.4 hours for 1000 iterations.**

Note: Final code (mentioned above parameters) of 1000 iterations is run on a server with **220GB RAM, i7 processor**. Rest of the experiments are run on laptop (**13 GB available RAM**) for 100 iterations which require lesser time and memory. **OS: Linux 16.04 LTS, compilation command "gcc AE16B005.c -lm".**

Note2: The final algorithm is not memory optimised as a new tree is created for each iteration and all the memory form the old tree is not freed. However, a laptop with 13GB available RAM should be able to run 90-100 iterations of the submitted code.

(a) Table 1. shows the variation of time of execution with number of iterations performed.

Number of iterations	1	10	100	1000
Time of execution (sec)	15.06	166.6	1508.9	15804.6

Table 1: Variation with iterations

Conclusion: Time of execution scales **linearly** with number of iterations. Hence other experiments can be conducted for small number of iterations and can be scaled linearly for large iterations.

$$T \propto \#iterations \tag{8}$$

(b) Execution time variation with number of particles is given below for 100 iterations. Keeping the regions constant (7 bits).

Number of Particles	10000	100000	1000000
Time of execution (sec)	4.118	27.72	1508.9

Table 2: Variation with number of particles

Conclusion According to equation (3), if b is constant, then

$$T \propto p^2 + c * p \tag{9}$$

Therefore,

$$\frac{T_2}{T_1} = \frac{100 * p^2 + 10 * cp}{p^2 + cp} = \frac{100p + 10c}{p + c} \tag{10}$$

But this trend is not followed in Table 2. The constant factor or number of floating point operations and memory allocation operation might have caused the deviation in the trend.

(c) Execution time variation with number of BITS (or regions) is given below for 100 iterations. Keeping the particles constant.

Number of BITS (# regions)	7 ($\approx$ 10000)	10 ($\approx$ 1000000)
Time of execution (sec)	1508.9	723.95

Table 3: Variation with number of particles

Conclusion: From equation (3), if p is constant,

$$T_1 \propto \frac{1}{b} + c * \log_4 b \tag{11}$$

$$T_2 = \frac{1}{100b} + c \log_4 100b = \frac{1}{100b} + c * \log 4b + 3.3c \tag{12}$$

Hence the ratio varies with the constant c and the magnitude of b and cannot be determined easily.

(d) Execution time variation with number of particles and regions keeping the number of neighbors (i.e. $\frac{p}{b}$) fixed is given below for 100 iterations.

Number of particles	10000	1000000
Number of regions	100	10000
Time of execution (sec)	23.59	1508.9

Table 4: Variation with particles keeping neighbors fixed

Conclusion From equation (3), if neighbors are fixed, then $\mathrm{p}_{\overline{bisfixed.}}$
Then,

$$T \propto p * \frac{p}{b} + cplog(b) \propto p(1 + c \log b) \tag{13}$$

This trend is observed as scaling p to 100*p causes the time to scale by 60 times.

(e) Comparison of 4-nary Tree algorithm with brute force algorithm.

Approach	**Brute Force**			**4-nary Tree**		
Number of particles	1000 (1000iter)	10000	100000	1000 (1000iter)	10000 (100neighbors)	100000 (1000neighbors)
Execution Time(sec)	33.84	254.5	2323.4	9.013	23.59	344

Table 5: Comparison with Brute Force Method

Conclusion: Brute force time complexity depends on

$$T_{brute} = \mathcal{O}(p^2) \tag{14}$$

Where as 4-nary tree complexity is

$$T_{4nary} = \mathcal{O}(\frac{p^2}{b} + cplog(b)) \tag{15}$$

Assuming first term to be much greater than second term, the ratio

$$\frac{T_{brute}}{T_{4nary}} \propto b \tag{16}$$

The experiments follow this trend as in the experiments, b $\approx 64 and the algorithm scales by that factor.$

Clearly, the 4nary Tree Algorithm is much faster than the brute force algorithm for particle sizes 10^3 and above as tested.

2.5 Velocity Distribution

Partial Quicksort is required. Quicksort sorts the array in place. For each step, it chooses a pivot and sorts the array such that all elements to the left of partition are less than pivot and all elements to the right are greater than pivot.

Here, 100 bins are required with equal pivot intervals. Quicksort is called 99 times with the pivot increasing by constant in each call. Only 99 iterations of quicksort are required instead of a full quicksort.

L2 cache size of CPU = 512kB.

Plotting the velocity distribution.

number of bins = 100

bin width = 0.1

Note: Bin separation given in the question paper is equidistant from minimum velocity to maximum velocity. However almost 98% of the particles have velocities in the first bin. Hence, to get better insight bin width is reduced to 0.1 m/s and the histogram spans upto v = 10 m/s. This captures 95% of the data.

Final distribution is plotted as follows.

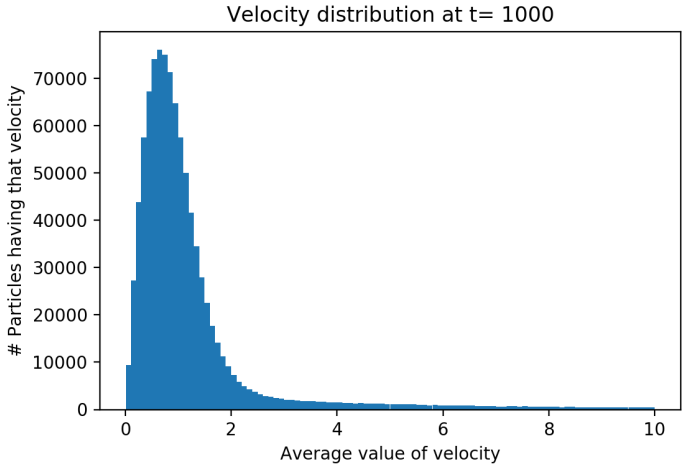

Figure 1

The progression of the particle velocities with respect to time can be seen below in Figure 2.

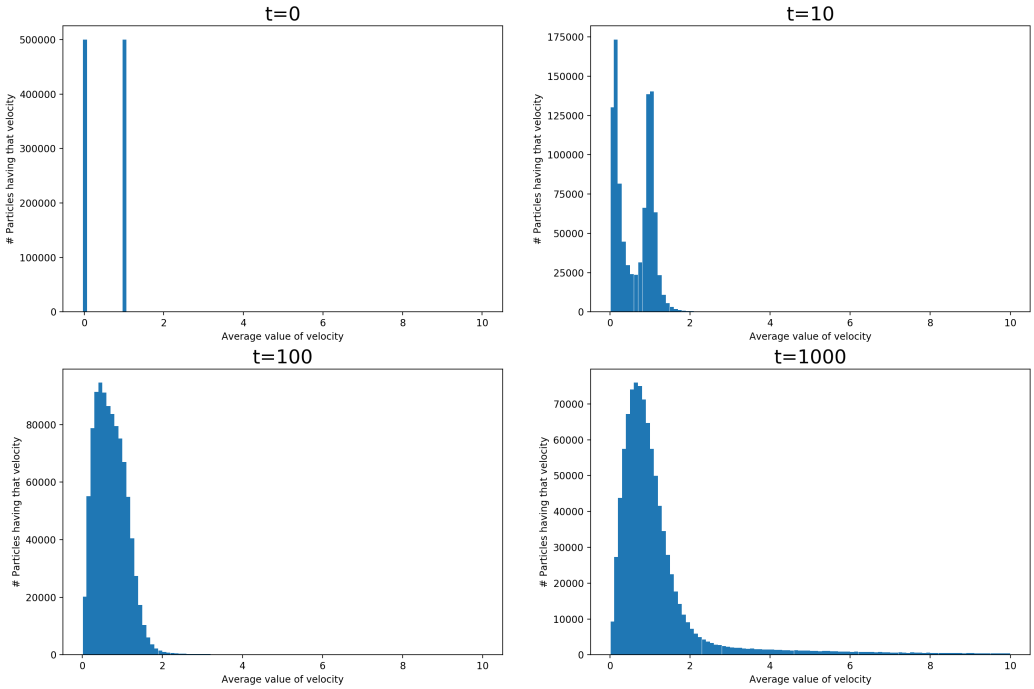

Figure 2

A spline curve is fitted to the velocity distribution passing through the average velocities of the bins. This can be seen in Figure 3.

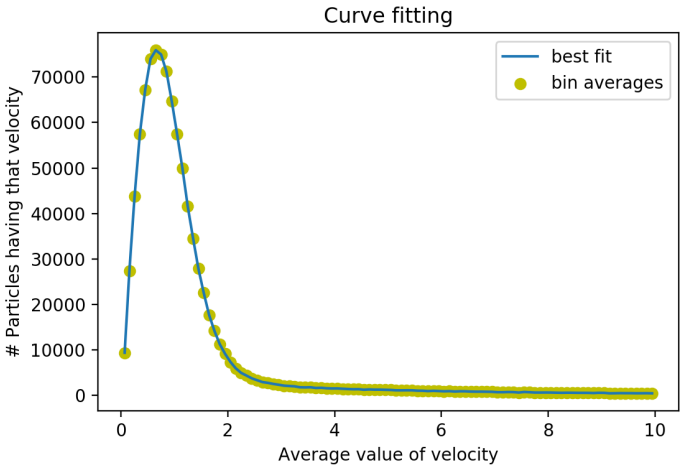

Figure 3

Conclusion:

- Initially, half the particles have zero velocities and other half have unit velocities.
- As the time progresses, the moving particles interact with stationary particles and exchange their momentum resulting in intermediate velocity values.
- Forces on some particles aggregate to large values causing large accelerations of the particles thus increasing their velocities.
- Finally, at thermal equilibrium, the particle velocities attend **Maxwell Boltzmann distribution**.

The fitting of the Maxwell Boltzmann distribution is given in Figure 4. As observed, the curve almost matches the distribution. If the simulation is run for long enough time, both the curves will exactly match.

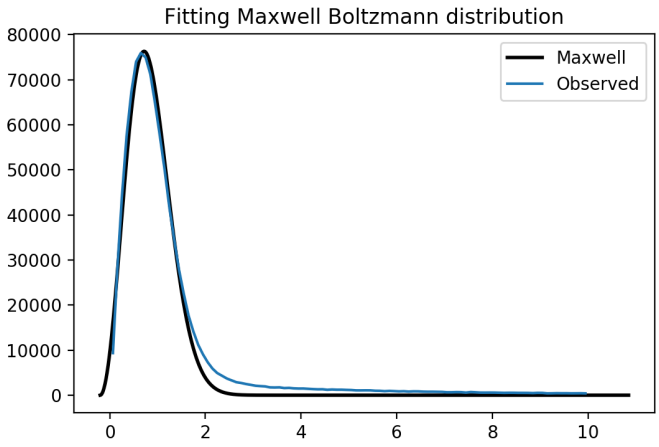

Figure 4

EE4371
Introduction to Data Structures and Algorithms
End Term Report

Author: Rahul Chakwate
AE16B005

10th December 2019

1 Modified Knapsack Problem

1.1 Problem Formulation

The ordinary Knapsack Problem is formulated as
Maximize

$$\sum_{i=1}^{n} v_i x_i \tag{1}$$

subject to

$$\sum_{i=1}^{n} w_i x_i \leq W \tag{2}$$

where $x_i \in \{0, 1\}$ is a binary indicating if weight w_i is present in the summation or not.
This problem can be solved by **greedy** as well as **dynamic programming** based algorithm with **recursion**.
However, the given problem is a modification of the general knapsack problem where

$$v_i = \log_{100} w_i \tag{3}$$

Hence the objective now becomes
Maximize

$$\sum_{i=1}^{n} x_i * \log_{100} w_i \tag{4}$$

and the constraints have an additional multiplying term

$$\sum_{i=1}^{n} w_i x_i \leq 10000 * \sum_{i=1}^{n} x_i \tag{5}$$

1.2 Pseudo Code

The **abstract algorithm** is listed below.

- Solving the ordinary Knapsack Problem using dynamic programming, the m(n,W) matrix stores the best solution so far with first n weights and weights summation constraint to W.
- the $(i, j)^{th}$ entry in the matrix come from one of the following.

$$m(0, w) = m(i, 0) = 0 \tag{6}$$

$$if w_i > w, m(i, w) = m(i - 1, w) \tag{7}$$

$$if w_i \leq w, m(i, w) = max(m(i - 1, w), m(i - 1, w - w_i) + w_i \tag{8}$$

  If there is an improvement, it is buit on the current optimal solution.

- This value increases monotonically from left to right i.e from lower W to higher W.
- Also, the last row uses the optimal weights out of all the available weights to reach the solution.
- Hence, if we start from the last row, last column of m(n,$W\sqrt{n}$), we have the maximum possible value with optimal combination of weights.

- This can be checked against $w\sqrt{k}$ constraint.
- If satisfied, this is our optimal solution.
- If violated, then check for previous entry m(n,$W\sqrt{n}-1$).

Pseudo Code

```
integer w[] //stores the weights
integer v[] //stores the values
double value[i][j] //stores the optimal solution upto (i,j) for dynamic programming
integer numweight[i][j] //stores the number of weights used to check for the modified constraint
Function Knapsack():
    This function calls m(n,W), back tracks the weights and checks for W√k condition.
    value[0,j]=-1 //initialization
    numweight[i][j]=0
    for Wk from W → 0 do
        res = m(n,Wk)
        for i from n → 0 and res ¿ 0 do
            if w[i] > j then
                continue
            //ignore that weight
            if value[i − 1][j] > value[i − 1][j − w[i]] + v[i] then
                continue
            //entry came from the same weights, no new weight added
            else if value[i − 1][j] < value[i − 1][j − w[i]] + v[i] then
                res = res − v[i]
                j = j − w[i]
                print(w[i]) //if the entry came from this option, the a new weight w[i] is added.
                else
                    if numweight[i − 1][j] > numweight[i − 1][j − w[i]] then
                        continue
                    //If values are same from both paths, choose path with more constraints so that the
                      constraint is satisfied.
                    else
                        res− = v[i]
                        j− = w[i]
                        print(w[i]) //if more weights are from above cell, then no weights are added. If
                          not, then weight is added and print it,
        end
        if sum_of_weights ≤W0 * √numweight[n][Wk] then
            Optimal solution found at Wk
        //in the end, check for the modified constraint, if satisfied then this is the optimal solution
          else continue by reducing Wk.
    end
return
```

Function `m(integer` i, **integer** j`):`

This function recursive computes the optimal Knapsack solution upto $(i, j)^{th}$ cell. It computes only required entries and not all the entries.

if $i == 0 or j \leq 0$ **then**
  return 0
//the actual matrix starts from i=1,j=1
if *value[i-1][j] is unassigned* **then**
  call m(i-1,j)
//to compute i-1,j value
if $w[i] > j$ **then**
  value[i][j] = value[i-1][j]
if *value[i-1][j-w[i]] is unassigned* **then**
  call m(i-1,j-w[i])
//to compute i-1,j-w[i] node
else
  if $value[i-1][j] != value[i-1][j-w[i]] + v[i]$ **then**
    value[i][j] = max(value[i-1][j], value[i-1][j-w[i]]=v[i]) numweight[i][j] = corresponding numweight entry.
  else
    if $numweight[i-1][j] > numweight[i-1][j-w[i]] + 1$ **then**
      value[i][j] = value[i-1][j]
      numweight[i][j] = numweight[i-1][j]
    else
      $numweight[i][j] = numweight[i-1][j-w[i]] + 1$
    end
  end
end
return *value[i][j]*

1.3 Code Implementation and Results

Input data: contains 907 weights with values ranging from 1 to 9999.

Array width = $W0 * \sqrt{n} = 10000 * \sqrt{907} = 301165 \approx 310000$

Note: Dynamic memory allocation is required for array of this big size. Hence array of pointers is used with malloc().

Time of Execution: **16.7 sec.**

Optimum solution found at **W = 171466**

Maximum objective function value satisfying all constraints = **369.126**

Number of weights used in optimum solution = **294**

Sum of the weights used = **171464**

Clearly

$$10000 * \sqrt{294} = 171464.28 \tag{9}$$

Hence,

$$\sum_{i=1}^{n} w_i x_i = 171464 < 171464.28 = 10000 * \sum_{i=1}^{n} x_i \tag{10}$$

To arrive at the optimum, $10000\sqrt{970} - 171466 = 129698$ conditions had to be checked for constraint violation.

Time Complexity of Modified Knapsack Problem:

Since full $W\sqrt{n}byn$ matrix needs to be computed in the worst case, the worst case time complexity = O($W * n^{3/2}$).

As compared to ordinary Knapsack complexity of O(Wn), this algorithm is $\sqrt{n}$ times slower.

But we do not compute full table in average case and hence this is a faster approach than iterating over the entire table.