Bansilal Ramnath Agarwal Charitable Trust's
Vishwakarma Institute of Information Technology

**Department of**

**Artificial Intelligence and Data Science**

| | | |
|---|---|---|
| Name: Siddhesh Dilip Khairnar | | |
| Class: TY | Division: B | Roll No: 372028 |
| Semester: V | | Academic Year: 2023-24 |
| Subject Name & Code: Design and Analysis of Algorithm: ADUA31202 | | |

Title of Assignment: Implement 0/1 Knapsack problem using Following algorithmic strategies.

1. Dynamic programming
2. Back tracking
3. Branch and bound

# ASSIGNMENT NO. 5

DAA Assignment no 5

Name: Siddhesh Dilip Khairnar
Division: B Batch: B2
Rollno: 372028 PRN no: 22110398

Aim: Implement 0/1 knapsack problem using following algorithm strategies.
(a) Dynamic programming
(b) Back tracking
(c) Branch and bound.

~~The~~
Theory:-

The 0/1 knapsack problem is a classic optimization problem. ~~It will~~ It will explain how to implement it using the three different algorithm strategies. It divide into the theory behind the 0/1 knap-sack problem and the tree algorithm strategies used to solve it.

~~Dynamic Programming~~ → 0/1 knapsack problem:→

- Dynamic Programming is one of the most efficient way to solve the 0/1 knapsack problem.
- The 0/1 knapsack problem is a combinatorial optimization problem where you are given a set of item, each with a weight and a value, and a knapsack with a maximum weight capacity.
- The goal is to select a subset of item to maximize the total value while not exceeding the knapsack's weight capacity
- The "0/1" in the name signifies that you can either include an item (0) or exclude it (1), meaning you cannot take fractional parts of item.

(a) Dynamic programming:
- The DP approach solve the problem by breaking it down into smaller subproblem and storing their solution to avoid redundant calculation
- It uses a 2D table (often called a memoization table) to store intermediate result.
- The key idea is to fill the table iteratively, considering two choices at each step: including the current item or excluding it.

(b) Backtracking:
- Backtracking is a brute-force approach that explores all possible combination of item to find the optimal solution.
- It uses recursion to consider taking or skipping each item at each step.
- It can be inefficient for large problem instances since it explore an exponential no. of possibilities.

(c) Branch and Bound: →
- Branch & Bound is an optimization algorithm that efficiently prunes the search space to find the optimal solution.
- It sort the item by their value-to-weight ratio to consider more promissing branches first.
- The algorithm maintain an upper bound and uses it to eliminate branches where the maximum achievable value cannot exceed the upper bound.
- It explores subproblem in a way that eliminates ~~the~~ unpromissing branches early, making it more efficient than backtracking.

# 0/1 knapsack Problem using Dynamic Programming :→

Consider -

- knapsack weight capacity = W
- No. of item each having some weight & value = n

0/1 knapsack problem is solved using dynamic programming in the following step :→

Step 01 :→

- Draw a table say 'T' with (n+1) no. of rows & (w+1) no. of columns.
- fill all the boxes of $0^{th}$ row & $0^{th}$ column with zeroes as shown :→

Step 02 :→

- Start filling the table row wise top to bottom from left to right.

use the following formula :→

$T(i,j) = \max\{T(i-1,j), value_i + T(i-1, j - weight_i)\}$

Here $T(i,j)$ = max value of the selected item if we can take item 1 to i and have weight restrictions of j.

- This step lead to completely filling the table
- Then, value of the last box represents the maximum possible value that can be put into the knapsack.

Step 03 :→

To identify the item that must be put into the knapsack to obtain that maximum profit.

1) Consider the last column of the table
2) Start scanning the entries from bottom to top.

- On encountering an entry whose value is not same as the value stored in the entry immediately above it, mark the row label of that entry.
- After all the entries are scanned, the marked labels represent the item that must be put into the knapsack:

Time Complexity: →

- Each entry of the table requires constant time O(1) for its computation
- It takes O(nw) time to fill (n+1)(w+1) table entries
- It takes O(n) time for tracing the solution since tracing process traces the n rows:
- Thus, overall O(nw) time is taken to solve 0/1 knapsack problem using dynam programming.

Example

| item | weight | value |
|---|---|---|
| Mirror | 2 | 3 |
| Silver nuggets | 3 | 4 |
| Painting | 4 | 5 |
| Vase | 5 | 6 |

Soln

| | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| ✓ 1 | 0 | 0 | 3 | 3 | 3 | 3 |
| ✓ 2 | 0 | 0 | 3 | 4 | 4 | 7 |
| 3 | 0 | 0 | 3 | 4 | 5 | 7 |
| 4 | 0 | 0 | 3 | 4 | 5 | 7 |

- we mark the rows labelled '1' & '2'.
- Thus, item that must be put into the knapsack to obtain the max value 7 are item-1 & item2.

Conclusion: Thus we successfully performed 0/1 Knapsack Problem.

C
23/11/2023

# Using Dynamic Programming approach

## Program Code:

```
#include <bits/stdc++.h>
using namespace std;
 int max(int a, int
b)
{     return (a > b) ? a :
b;
}  int knapSack(int W, int wt[], int val[],
int n)
{     int i, w;     vector<vector<int>> K(n + 1,
vector<int>(W + 1));
     for(i = 0; i <= n;
i++)
    {         for(w = 0; w <=
W; w++)
        {
            if (i == 0 || w == 0)
K[i][w] = 0;
            else if (wt[i - 1] <= w)
                K[i][w] = max(val[i - 1] +
                                K[i - 1][w - wt[i -
1]],                                 K[i - 1][w]);
else
                K[i][w] = K[i - 1][w];
        }
}
    return K[n][W];
}
int main()
{     int val[] = { 60, 100, 120
};     int wt[] = { 10, 20, 30
};     int W = 50;
    int n = sizeof(val) / sizeof(val[0]);
cout << knapSack(W, wt, val, n);     return
0;
}
```

## Output:

```
220

...Program finished with exit code 0
Press ENTER to exit console.
```

## <u>Using Backtracking approach</u> Program Code:

```
#include <bits/stdc++.h> using namespace std;
int knapSack(int W, int wt[], int val[], int
n)
{     int dp[W + 1];
memset(dp, 0, sizeof(dp));
     for (int i = 1; i < n + 1; i++)
{          for (int w = W; w >= 0; w--
) {
             if (wt[i - 1] <= w)                 dp[w] =
max(dp[w],                               dp[w - wt[i - 1]] +
val[i - 1]);
        }
}
    return dp[W];
}
int main()
{
    int val[] = { 60, 100, 120 };
int wt[] = { 10, 20, 30 };     int W =
50;     int n = sizeof(val) /
sizeof(val[0]);     cout << knapSack(W,
wt, val, n);     return 0;
}
```

## Output:

```
220

...Program finished with exit code 0
Press ENTER to exit console.
```

## Using Branch and Bound approach

This algorithm uses the greedy approach to calculate an upper bound on the solution. If a node's upper bound is less than the maxProfit, we don't explore it further because it cannot lead to a better solution. This helps us prune the search space and find the optimal solution more efficiently.

In essence, the algorithm explores different combinations of items, always considering the most promising ones first, and updates the maxProfit when a better solution is found. This way, it finds the best solution for the Fractional Knapsack problem.

### Algorithm:

1. Sort all items in decreasing order of the value-to-weight ratio. This allows us to consider the most valuable items first.
2. Initialize maxProfit to 0. This will keep track of the best solution found so far.
3. Create an empty queue, Q, to explore different possibilities.
4. Start with a dummy node representing the root of a decision tree. This node has zero profit and zero weight.
5. While there are nodes in the queue Q:
    - Take out an item from Q. Let's call it u.
    - Calculate the profit of the next level node. If this profit is greater than maxProfit, update maxProfit.
    - Calculate a bound for the next level node. If this bound is greater than maxProfit, add the next level node to the queue.
    - Consider the scenario where the next level node is not included in the solution and add a new node to the queue with an increased level, but without the weight and profit of the next level nodes.

## Program Code:

```
#include <bits/stdc++.h>
using namespace std;
 struct
Item
{     float
weight;     int
value;
};  struct
Node
{     int level, profit,
bound;     float weight;
};  bool cmp(Item a,
Item b)
{     double r1 = (double)a.value /
a.weight;     double r2 = (double)b.value
/ b.weight;     return r1 > r2;
}  int bound(Node u, int n, int W, Item
arr[])
{     if (u.weight >=
W)         return 0;
     int profit_bound =
u.profit;
     int j = u.level + 1;
int totweight = u.weight;
     while ((j < n) && (totweight + arr[j].weight <=
W))
    {         totweight +=
arr[j].weight;         profit_bound
+= arr[j].value;         j++;
    }
    if (j < n)         profit_bound += (W - totweight)
* arr[j].value /
arr[j].weight;
     return
profit_bound;
}
```

```
 int knapsack(int W, Item arr[], int
n)
{     sort(arr, arr + n,
cmp);

    queue<Node> Q;
    Node u, v;

    u.level = -1;
    u.profit = u.weight = 0;
    Q.push(u);
int maxProfit = 0;
while (!Q.empty())
    {
        u = Q.front();
        Q.pop();
if (u.level == -1)
            v.level = 0;
if (u.level == n-1)
continue;

        v.level = u.level + 1;

        v.weight = u.weight + arr[v.level].weight;
        v.profit = u.profit + arr[v.level].value;
if (v.weight <= W && v.profit > maxProfit)
maxProfit = v.profit;

        v.bound = bound(v, n, W, arr);
        if (v.bound > maxProfit)
            Q.push(v);

        v.weight = u.weight;
        v.profit = u.profit;
        v.bound = bound(v, n, W, arr);
if (v.bound > maxProfit)
            Q.push(v);
    }      return
maxProfit;
}  int
main()
```

```
{
    int W = 10;
    Item arr[] = {{2, 40}, {3.14, 50}, {1.98, 100},
                {5, 95}, {3, 30}};
int n = sizeof(arr) / sizeof(arr[0]);
     cout << "Maximum possible profit =
"
        << knapsack(W, arr, n);
     return
0;
}
```

## Output:

```
Maximum possible profit = 235

...Program finished with exit code 0
Press ENTER to exit console.
```