Kathmandu University

Department of Computer Science and Engineering

Dhulikhel, Kavre

A Report on '**Lab Work 4'** [COMP 314]

Submitted by:

Chandan Kumar Mahato (31)

III-year, II semester

Submitted to:

Dr. Rajani Chulyadyo

Department of Computer Science and Engineering

Submission Date: May 13, 2023

Qlab4

Solve the Knapsack problem using the following strategies:
1. Brute-force method (Both fractional and 0/1 Knapsack)
2. Greedy method (Fractional Knapsack)
3. [Bonus] Dynamic programming (0/1 Knapsack)

Solution:

Implementing Knapsack problem using the above listed strategies

Source Code:

https://github.com/ChandankMahato/DSA_Lab_6th_Sem

Scripts:

npm run test
npm run BRFK
npm run BR01K
npm run GFK
npm run DP01K

1. Brute-force method (fractional and 0/1 Knapsack)

Algorithm:

- Initialize best combination and best value to empty and 0 respectively.
- Generate all possible combinations of items.
  - For fractional, generate fractional combination.
- For each combination, calculate its total weight and value.
- If the total weight is less than or equal to the capacity and the total value is greater than the current best value, update the best combination and best value
- Return the best combination and best value as the solution.

Time Complexity:

Brute-Force 0/1 Knapsack: **O(2^n)**
Brute-Force Fractional Knapsack: **O(n!)**

Same function is used for both BR01K and BRFK. Only difference is in generating the combination, fractional includes fractional combination and 0/1 does not.

```
function BR01Knapsack(items, capacity) {
  const combinations = generateCombinations(items);
  let bestCombination = [];
  let bestValue = 0;
  combinations.forEach((combination) => {
    let weight = 0;
    let value = 0;
    combination.forEach((item) => {
      weight += item.weight;
      value += item.value;
    });
    if (weight <= capacity && value > bestValue) {
      bestCombination = combination;
      bestValue = value;
    }
  });
  return { combination: bestCombination, value: bestValue };
}
```

1.1 Brute-force method (fractional)

Generating Combination

```
function generateCombinations(items) {
  const combinations = [[]];
  items.forEach((item) => {
    const newCombinations = [];
    combinations.forEach((combination) => {
      const newCombination = [...combination, item];
      newCombinations.push(newCombination);
      for (let i = 1; i < item.weight; i++) {
        const fraction = i / item.weight;
        const newItem = {
          type: item.type,
          weight: i,
          value: fraction * item.value,
        };
        const newCombination = [...combination, newItem];
        newCombinations.push(newCombination);
      }
    });
    combinations.push(...newCombinations);
  });
  return combinations;
}
```

1.2 Brute-force method (0/1 Knapsack)
Generating Combination

```
function generateCombinations(items) {
  const combinations = [[]];
  items.forEach((item) => {
    const newCombinations = [];
    combinations.forEach((combination) => {
      const newCombination = [...combination, item];
      newCombinations.push(newCombination);
    });
    combinations.push(...newCombinations);
  });
  return combinations;
}
```

2. Greedy method (Fractional Knapsack)

Algorithm:

- Sort items by their value-to-weight ratio in descending order
- Initialize an empty array for the best combination and a variable to keep track of the total weight
- Iterate through each item, starting with the one with the highest value-to-weight ratio
- If adding the current item to the combination doesn't exceed the capacity, add it to the combination, update the total weight, and update the best value
- If adding the current item exceeds the capacity, calculate the fraction of the item that can fit into the knapsack and its corresponding value, add it to the combination, update the total weight, and return the best combination and value
- Return the best combination and value

Time Complexity: O(nlogn)

```
function GFKnapsack(items, capacity) {
  const sortedItems = sortItems(items);
  let bestCombination = [];
  let bestValue = 0;
  let weight = 0;
  for (let i = 0; i < sortedItems.length; i++) {
    let item = items[i];
    if (weight + item.weight <= capacity) {
      weight += item.weight;
      bestValue += item.value;
      bestCombination.push(item);
    } else {
      let fractionWeight = capacity - weight;
      let fractionValue = item.value / item.weight;
      weight += fractionWeight;
```

```
      let newValue = fractionValue * fractionWeight;
      bestValue += newValue;
      bestCombination.push({
        type: item.type,
        weight: fractionWeight,
        value: newValue,
      });
      return { combination: bestCombination, value: bestValue };
    }
  }
}
function sortItems(items) {
  items.sort((a, b) => b.value / b.weight - a.value / a.weight);
  return items;
}
```

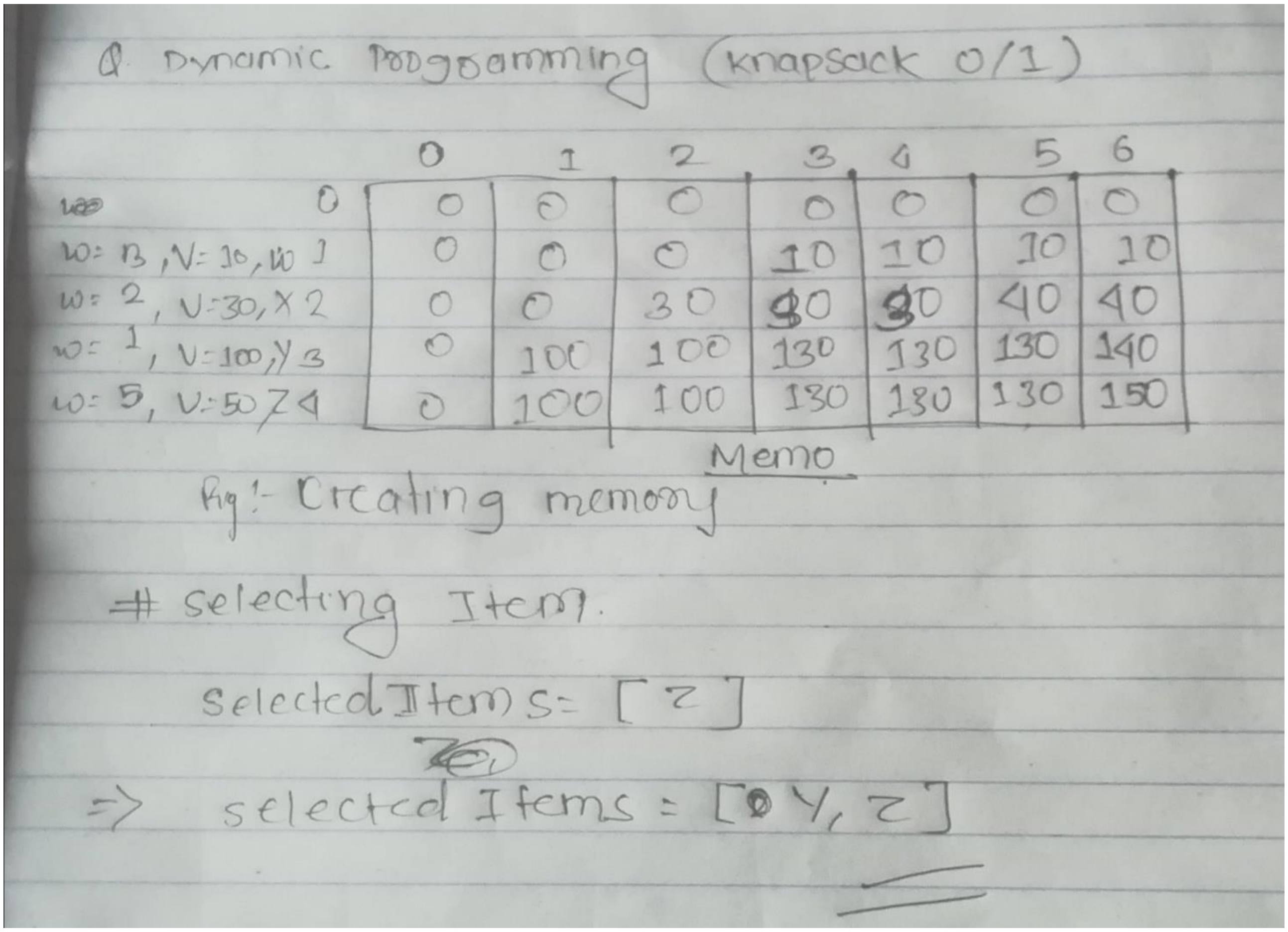
3. [Bonus] Dynamic programming (0/1 Knapsack)

Algorithm:

1. Define the function DP01Knapsack with input parameter items and capacity.
2. Initialize the variable n as the length of the items array.
3. Create a memoization array memo with n+1 rows and capacity+1 columns, filled with 0
4. Loop through i from 1 to n, and for each i loop through c from 1 to capacity.
5. Check if the weight of the current item is greater than C, if so set memo[i][C] to the value of memo[i-1][C]
6. otherwise, set memo[i][C] to the maximum value between memo[i-1][C] and ~~memo~~ memo[i-1][C- item.weight] + item.value
7. Create an empty array selected Items and set capacity as C
8. Loop through i from n to 1
9. If memo[i][c] is not equal to memo[i-1][c], add the item at index i-1 to selectedItems and subtract item.weight from C
10. Return an object with properties combination set to selected Items and value set to memo[n] capacity.

*Time Complexity: O(m*n)*

Calculation:



Q. Dynamic Programming (knapsack 0/1)

		0	1	2	3	4	5	6
	0	0	0	0	0	0	0	0
w=3, V=10, W 1		0	0	0	10	10	10	10
w=2, V=30, X 2		0	0	30	30	30	40	40
w=1, V=100, Y 3		0	100	100	130	130	130	140
w=5, V=50, Z 4		0	100	100	130	130	130	150

Memo

Fig 1:- Creating memory

selecting Item.

Selected Items = [Z]

=> selected Items = [Y, Z]

Implementation (Dynamic Programming 0/1 knapsack):

```
function DP01Knapsack(items, capacity) {
  const n = items.length;
  const memo = new Array(n + 1)
    .fill(null)
    .map(() => new Array(capacity + 1).fill(0));
  for (let i = 1; i <= n; i++) {
    const item = items[i - 1];
    for (let C = 1; C <= capacity; C++) {
      if (item.weight > C) {
        memo[i][C] = memo[i - 1][C];
      } else {
        memo[i][C] = Math.max(
          memo[i - 1][C],
          memo[i - 1][C - item.weight] + item.value
        );
      }
    }
  }
```

```
  const selectedItems = [];
  let C = capacity;
  for (let i = n; i >= 1; i--) {
    if (memo[i][C] !== memo[i - 1][C]) {
      const item = items[i - 1];
      selectedItems.unshift(item);
      C -= item.weight;
    }
  }
  return { combination: selectedItems, value: memo[n][capacity] };
}
```

Test Implementation:

```
describe("Knapsack Algorithm Test", () => {
  it("should return best combination using Brute Force 0/1 Knapsack", () => {
    const result = BR01Knapsack(items, capacity);
    expect(result.combination).toEqual([
      { type: "Y", weight: 1, value: 100 },
      { type: "Z", weight: 5, value: 50 },
    ]);
  });
  it("should return best combination using Brute Force Fractional Knapsack", () =>
{
    const result = BRFKnapsack(items, capacity);
    expect(result.combination).toEqual([
      { type: "Y", weight: 1, value: 100 },
      { type: "X", weight: 2, value: 30 },
      { type: "Z", weight: 3, value: 30 },
    ]);
  });
  it("should return best combination using Greedy Fractional Knapsack", () => {
    const result = GFKnapsack(items, capacity);
    expect(result.combination).toEqual([
      { type: "Y", weight: 1, value: 100 },
      { type: "X", weight: 2, value: 30 },
      { type: "Z", weight: 3, value: 30 },
    ]);
  });
  it("should return best combination using Dynamic Programing 0/1 Knapsack", () =>
{
    const result = DP01Knapsack(items, capacity);
    expect(result.combination).toEqual([
      { type: "Y", weight: 1, value: 100 },
      { type: "Z", weight: 5, value: 50 },
    ]);
  });
});
```

Conclusion:

Knapsack problem using Brute-Force strategies for (fractional and 0/1), Greedy strategies for fractional and Dynamic-Programming strategies (0/1) were implemented in the JavaScript. written code was benchmarked, and test cases were written using jest.

Output and Screenshots

```
> lab4@1.0.0 BR01K
> node BR01Knapsack.js

Input List of Items
[
  { type: 'W', weight: 3, value: 10 },
  { type: 'X', weight: 2, value: 30 },
  { type: 'Y', weight: 1, value: 100 },
  { type: 'Z', weight: 5, value: 50 }
]
Sack Capacity:  6
Best Combination
{
  combination: [
    { type: 'Y', weight: 1, value: 100 },
    { type: 'Z', weight: 5, value: 50 }
  ],
  value: 150
}
```

Fig 1: Brute-Force (0/1) Knapsack

```
> lab4@1.0.0 BRFK
> node BRFKnapsack.js

Input List of Items
[
  { type: 'W', weight: 3, value: 10 },
  { type: 'X', weight: 2, value: 30 },
  { type: 'Y', weight: 1, value: 100 },
  { type: 'Z', weight: 5, value: 50 }
]
Sack Capacity:  6
Best Combination
{
  combination: [
    { type: 'X', weight: 2, value: 30 },
    { type: 'Y', weight: 1, value: 100 },
    { type: 'Z', weight: 3, value: 30 }
  ],
  value: 160
}
```

Fig 2: Brute-Force (Fractional) Knapsack

```
> lab4@1.0.0 DP01K
> node DP01Knapsack.js

Input List of Items
[
  { type: 'W', weight: 3, value: 10 },
  { type: 'X', weight: 2, value: 30 },
  { type: 'Y', weight: 1, value: 100 },
  { type: 'Z', weight: 5, value: 50 }
]
Sack Capacity:  6
Best Combination
{
  combination: [
    { type: 'Y', weight: 1, value: 100 },
    { type: 'Z', weight: 5, value: 50 }
  ],
  value: 150
}
```

Fig 3: Dynamic Programming (0/1) Knapsack

```
> lab4@1.0.0 GFK
> node GFKnapsack.js

Input List of Items
[
  { type: 'Y', weight: 1, value: 100 },
  { type: 'X', weight: 2, value: 30 },
  { type: 'Z', weight: 5, value: 50 },
  { type: 'W', weight: 3, value: 10 }
]
Sack Capacity:  6
Best Combination
{
  combination: [
    { type: 'Y', weight: 1, value: 100 },
    { type: 'X', weight: 2, value: 30 },
    { type: 'Z', weight: 3, value: 30 }
  ],
  value: 160
}
```

Fig 4: Greedy (Fractional) Knapsack

```
PASS  test/Knapsack.test.js
  Knapsack Algorithm Test
    √ should return best combination using Brute Force 0/1 Knapsack (13 ms)
    √ should return best combination using Brute Force Fractional Knapsack (15 ms)
    √ should return best combination using Greedy Fractional Knapsack (3 ms)
    √ should return best combination using Dynamic Programing 0/1 Knapsack (4 ms)

-----------------|---------|----------|---------|---------|-------------------
File             | % Stmts | % Branch | % Funcs | % Lines | Uncovered Line #s
-----------------|---------|----------|---------|---------|-------------------
All files        |     100 |      100 |     100 |     100 |
 BR01Knapsack.js |     100 |      100 |     100 |     100 |
 BRFKnapsack.js  |     100 |      100 |     100 |     100 |
 DP01Knapsack.js |     100 |      100 |     100 |     100 |
 GFKnapsack.js   |     100 |      100 |     100 |     100 |
 data.js         |     100 |      100 |     100 |     100 |
 output.js       |     100 |      100 |     100 |     100 |
-----------------|---------|----------|---------|---------|-------------------
Test Suites: 1 passed, 1 total
Tests:       4 passed, 4 total
Snapshots:   0 total
Time:        1.394 s, estimated 4 s
Ran all test suites.

Watch Usage: Press w to show more.
```

Fig 5: Test cases