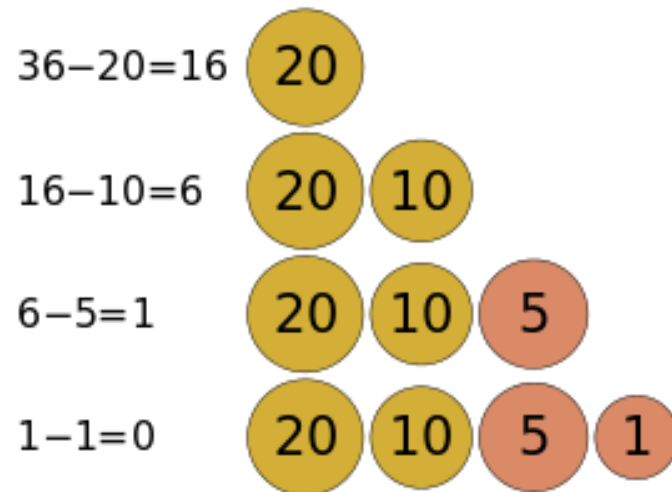# Greedy Algorithm

SWE2016-44

Introduction

How do we represent 36 cents using coins with {1, 5, 10, 20}?

→The coin of the highest value, less than the remaining change owed, is the local optimum.

$36-20=16$ 20

$16-10=6$ 20   10

$6-5=1$ 20   10   5
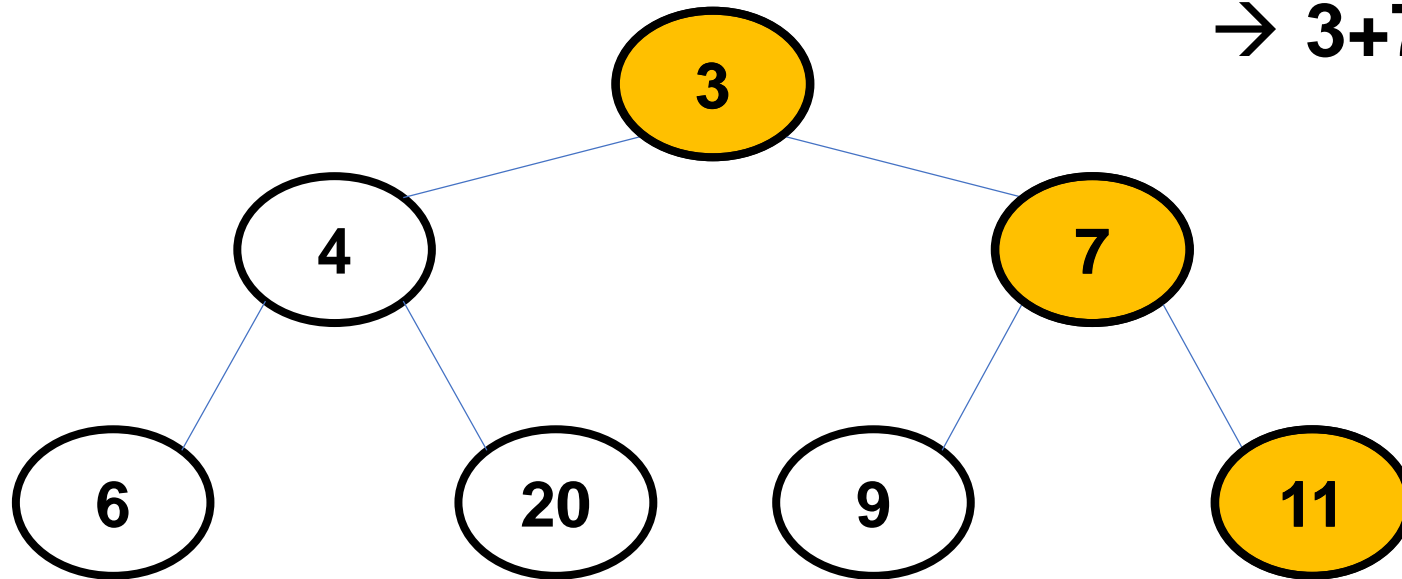
$1-1=0$ 20   10   5   1

Introduction

Greedy Algorithm – an algorithmic paradigm that follows the problem solving approach of making the locally optimal choice at each stage with the hope of finding a global optimum.

Pros – simple, easy to implement, run fast

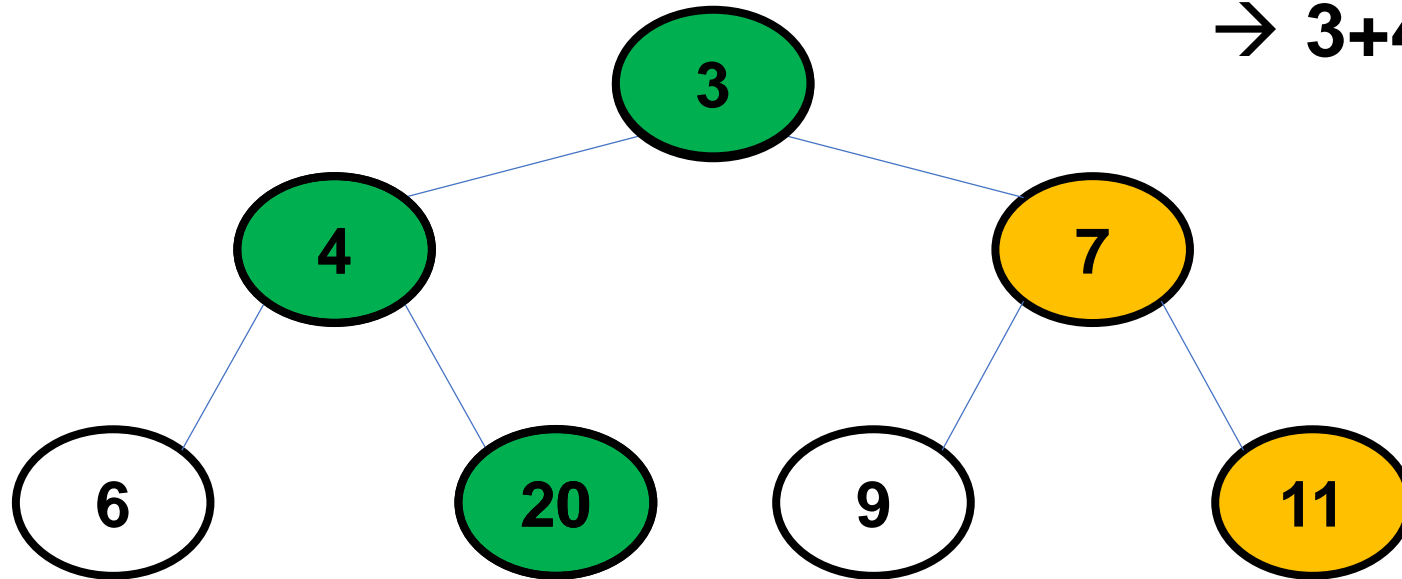Cons – Very often they don't provide a globally solution

Problem

Greedy Answer:
→ 3+7+11=21

Problem

Actual Answer:
→ 3+4+20=27

When to use?

Problems on which greedy approach work has two properties.

1. Greedy – choice property: A global optimum can be arrived at by selecting a local minimum

2. Optimal substructure: An optimum solution to the problem contains an optimal solution to subproblem

Applications

1. Activity Selection Problem

2. Huffman Coding

3. Job Sequencing Problem

4. Fractional Knapsack Problem

5. Prim's Minimum Spanning Tree (in Graph)

Activity Selection Problem

Problem Statement

Problem: Given n activities with their start and finish times. Select the maximum number of activities that can be performed by a single person, assuming that a person can only work on a single activity at a time.

Example:

Activity	A1	A2	A3
Start	12	10	20
Finish	25	20	30

A2 → A3   Total = 2 activities

Solution

<u>Greedy Approach</u>

1. Sort the activities according to their finishing time.

2. Select the first activity from the sorted array and print it.

3. Do following for remaining activities in the sorted array.
   - If the start time of this activity is greater than or equal to the finish time of previously selected activity then select this activity and print it.

Solution

Greedy Approach

1. Sort the activities according to their finishing time.

Activity	A1	A2	A3	A4	A5	A6
Start	0	3	1	5	5	8
Finish	6	4	2	9	7	9

Sorted

Activity	A3	A2	A1	A5	A6	A4
Start	1	3	0	5	8	5
Finish	2	4	6	7	9	9

Solution

Greedy Approach

2. Select the first activity from the sorted array and print it.

Activity	A3	A2	A1	A5	A6	A4
Start	1	3	0	5	8	5
Finish	2	4	6	7	9	9

Answer = A3

Solution

Greedy Approach

3. If the start time of this activity is greater than or equal to the finish time of previously selected activity then select this activity and print it.

Activity	A3	A2	A1	A5	A6	A4
Start	1	3	0	5	8	5
Finish	2	4	6	7	9	9

Answer = A3 ➔ A2 ➔ A5 ➔ A6   Total = 4 activities

Implementation

```cpp
// Returns count of the maximum set of activities that can
// be done by a single person, one at a time.
void printMaxActivities(Activitiy arr[], int n)
{
    // Sort jobs according to finish time
    sort(arr, arr+n, activityCompare);

    cout << "Following activities are selected n";

    // The first activity always gets selected
    int i = 0;
    cout << "(" << arr[i].start << ", " << arr[i].finish << "), ";

    // Consider rest of the activities
    for (int j = 1; j < n; j++)
    {
      // If this activity has start time greater than or
      // equal to the finish time of previously selected
      // activity, then select it
      if (arr[j].start >= arr[i].finish)
      {
          cout << "(" << arr[j].start << ", "
             << arr[j].finish << "), ";
          i = j;
      }
    }
}
```

Complexity

Time Complexity:

For sorted input: O(n)
For unsorted input: O(n log n)

n = number of activities

Huffman Coding

Huffman Coding

- **It is a loseless data compression**

- **We assign variable-length codes to input characters, length of which depends on frequency of characters.**

	Frequency	Code
a	5	1100
f	45	0

Huffman Coding

- **It is a loseless data compression**

- **We assign variable-length codes to input characters, length of which depends on frequency of characters.**

- **The variable-length codes assigned to input characters are Prefix Codes.**

{0, 11}	**{0, 1, 11}**
Prefix Code	**Non Prefix Code**

Data

Data Information

Character	Frequency
a	5
b	9
c	12
d	13
e	16
f	45

Approach

Huffman Tree

1. Create a leaf node for each unique character and build a min heap of all leaf nodes.
2. Extract two nodes with the minimum frequency from the min heap.
3. Create a new interval node with frequency equal to the sum of the two nodes frequencies. Make the first extracted node as its left child and the other extracted node as its right child. Add this node to the min heap.
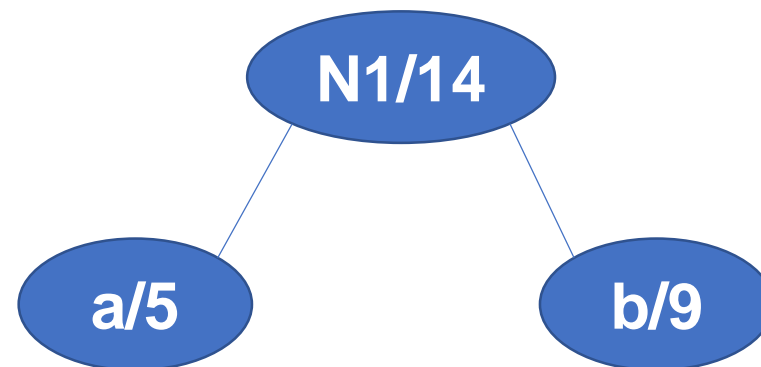4. Repeat Steps 2 and 3 until the heap contains only one node. The remaining tree is the root node and the tree is complete.

Approach

Huffman Tree

1. Create a leaf node for each unique character and build a min heap of all leaf nodes.
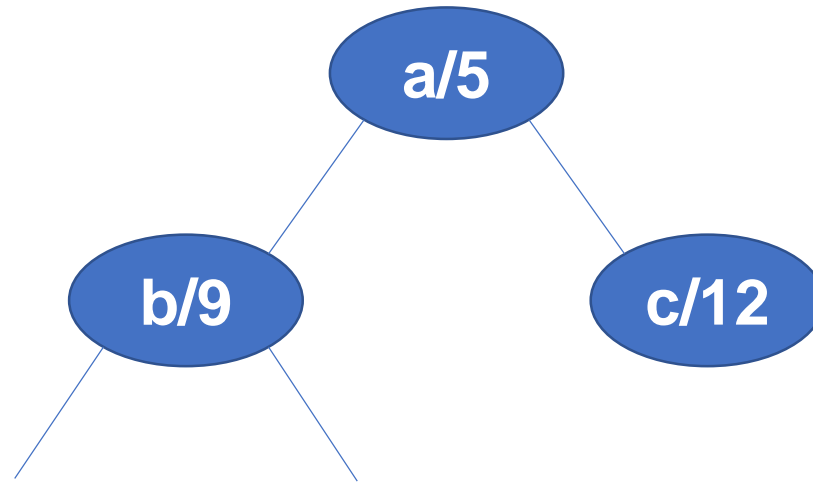
Character	Frequency
a	5
b	9
c	12
d	13
e	16
f	45

Approach

Huffman Tree

2. Extract two nodes with the minimum frequency from the min heap.

Character	Frequency
a	5
b	9
c	12
d	13
e	16
f	45

a/5   b/9

Approach

Huffman Tree

3. Create a new interval node with frequency equal to the sum of the two nodes frequencies. Make the first extracted node as its left child and the other extracted node as its right child. Add this node to the min heap.

Character	Frequency
a	5
b	9
c	12
d	13
e	16
f	45

Approach

Huffman Tree

3. Create a new interval node with frequency equal to the sum of the two nodes frequencies. Make the first extracted node as its left child and the other extracted node as its right child. Add this node to the min heap.

Character	Frequency
N1	14
c	12
d	13
e	16
f	45

N1/14

a/5   b/9

Approach

Huffman Tree

4. Repeat Steps 2 and 3 until the heap contains only one node. The remaining tree is the root node and the tree is complete.

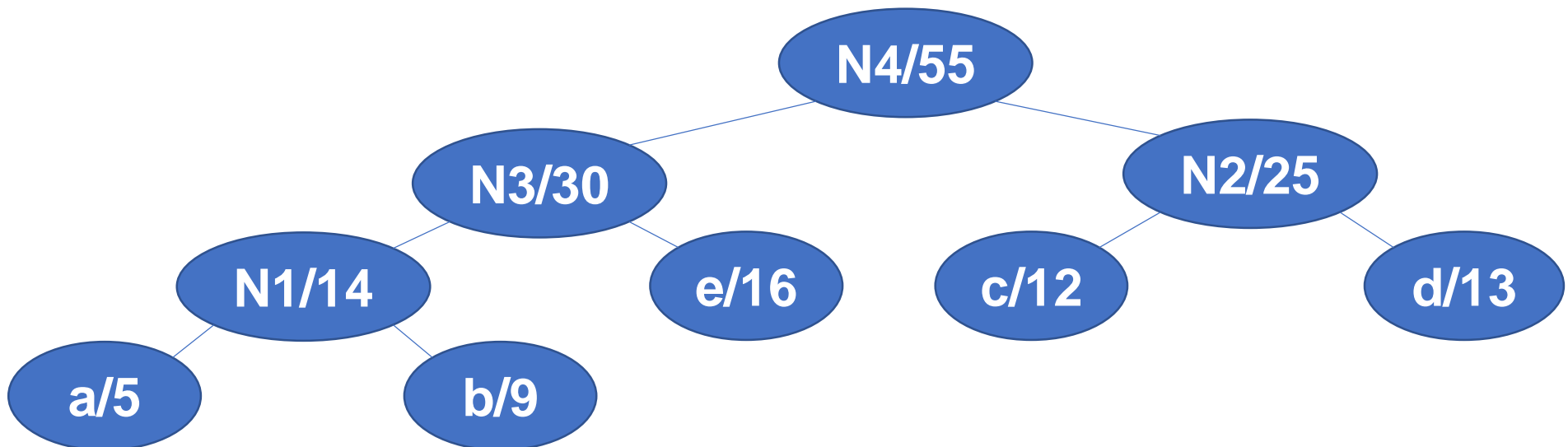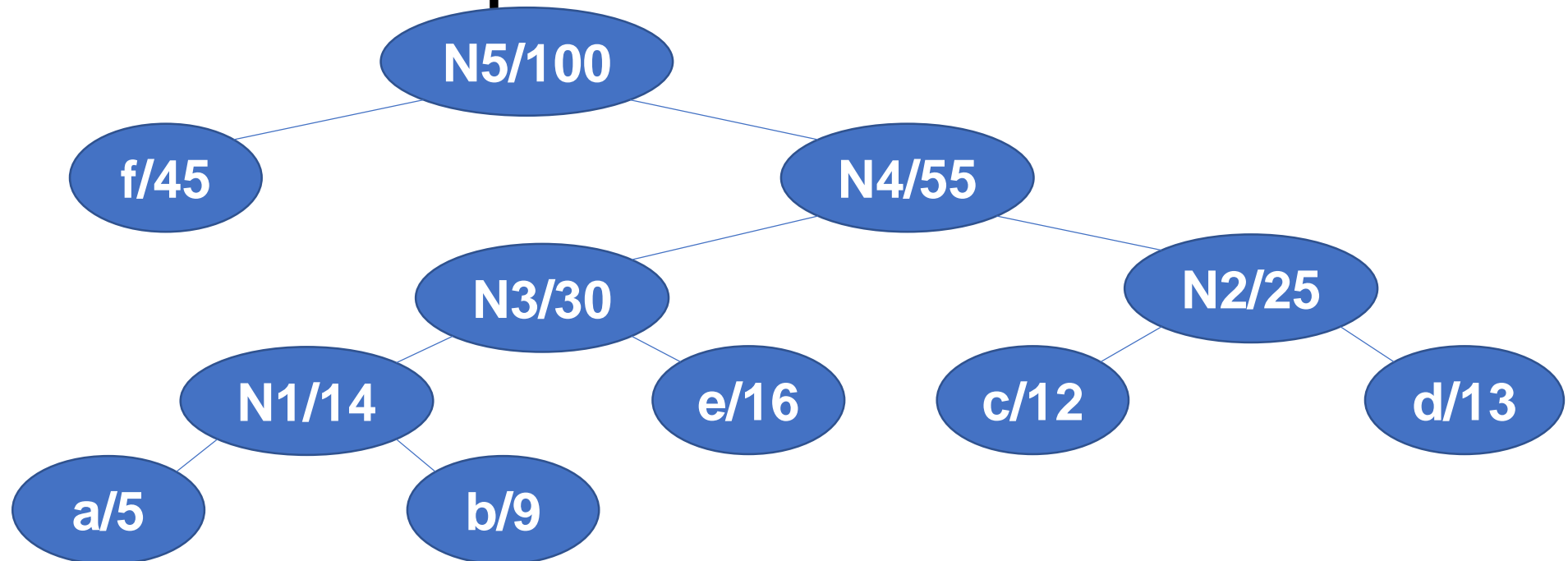| Character	Frequency
N1	14
c	12
d	13
e	16
f	45

Approach

Huffman Tree

4. Repeat Steps 2 and 3 until the heap contains only one node. The remaining tree is the root node and the tree is complete.

Character	Frequency
N1	14
N2	25
e	16
f	45
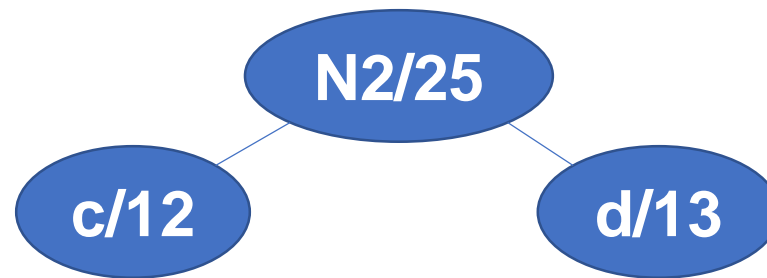
Approach

Huffman Tree

4. Repeat Steps 2 and 3 until the heap contains only one node. The remaining tree is the root node and the tree is complete.

Character	Frequency
N2	25
N3	30
f	45

Approach

Huffman Tree

4. Repeat Steps 2 and 3 until the heap contains only one node. The remaining tree is the root node and the tree is complete.
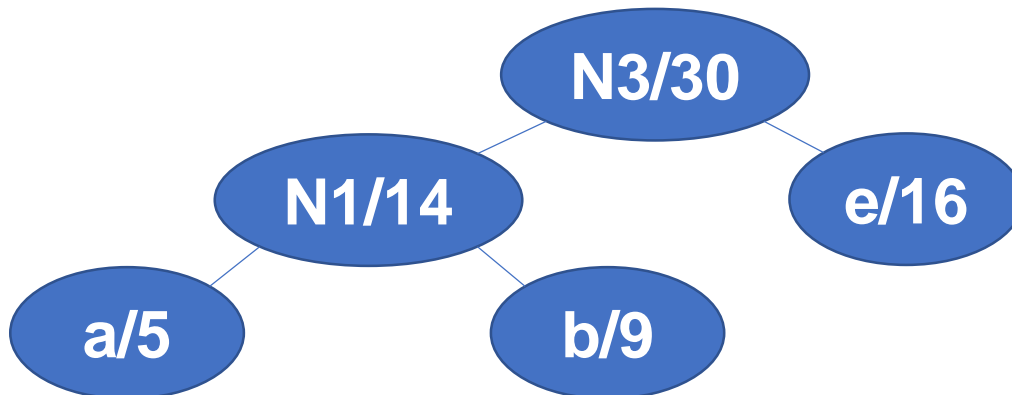
Character	Frequency
N4	55
f	45

Approach

Huffman Tree

Character	Frequency
N5	100

4. **Repeat Steps 2 and 3 until the heap contains only one node. The remaining tree is the root node and the tree is complete.**

Approach

Huffman Tree

Character	Frequency
a	1100
b	1101
c	100
d	101
e	111
f	0
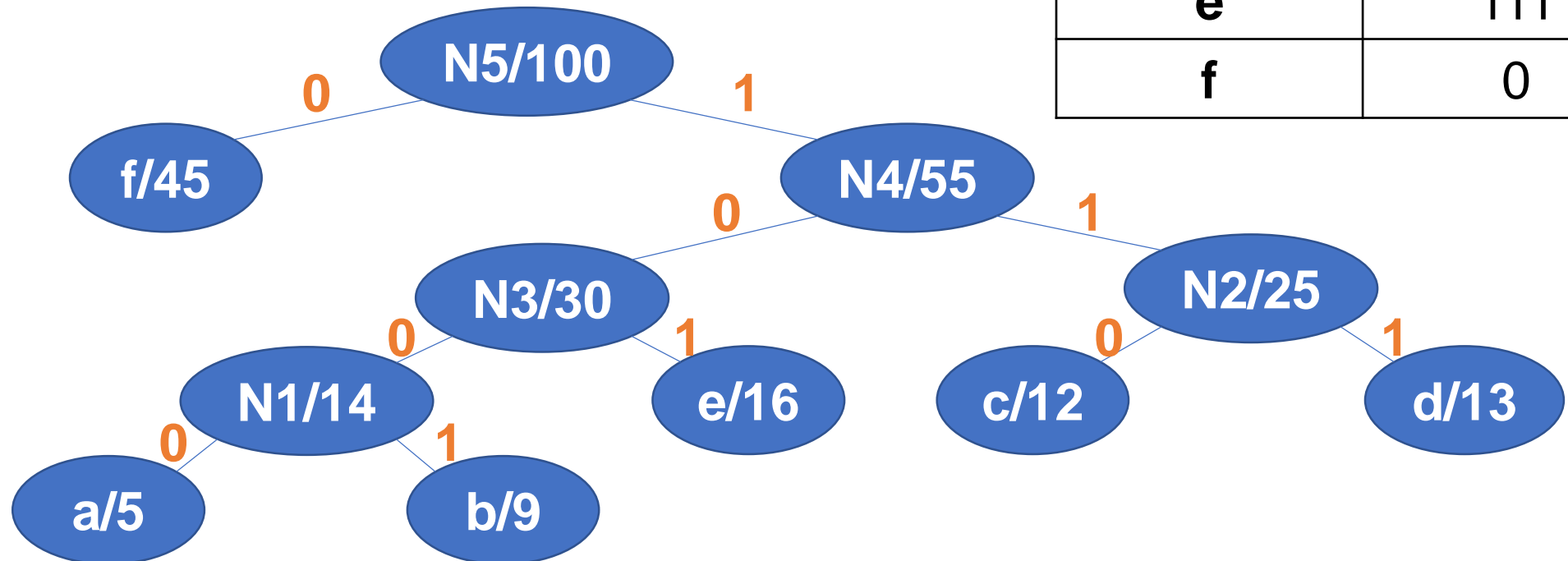
Implementation

```c
struct MinHeapNode* buildHuffmanTree(char data[], int freq[], int size)

{
    struct MinHeapNode *left, *right, *top;

    // Step 1: Create a min heap of capacity
    // equal to size.  Initially, there are
    // modes equal to size.
    struct MinHeap* minHeap = createAndBuildMinHeap(data, freq, size);

    // Iterate while size of heap doesn't become 1
    while (!isSizeOne(minHeap)) {

        // Step 2: Extract the two minimum
        // freq items from min heap
        left = extractMin(minHeap);
        right = extractMin(minHeap);

        // Step 3:  Create a new internal
        // node with frequency equal to the
        // sum of the two nodes frequencies.
        // Make the two extracted node as
        // left and right children of this new node.
        // Add this node to the min heap
        // '$' is a special value for internal nodes, not used
        top = newNode('$', left->freq + right->freq);

        top->left = left;
        top->right = right;

        insertMinHeap(minHeap, top);
    }

    // Step 4: The remaining node is the
    // root node and the tree is complete.
    return extractMin(minHeap);
}
```

Implementation

```c
void printCodes(struct MinHeapNode* root, int arr[], int top)

{

    // Assign 0 to left edge and recur
    if (root->left) {

        arr[top] = 0;
        printCodes(root->left, arr, top + 1);
    }

    // Assign 1 to right edge and recur
    if (root->right) {

        arr[top] = 1;
        printCodes(root->right, arr, top + 1);
    }

    // If this is a leaf node, then
    // it contains one of the input
    // characters, print the character
    // and its code from arr[]
    if (isLeaf(root)) {

        printf("%c: ", root->data);
        printArr(arr, top);
    }
}
```

Complexity

Time Complexity: O(n log n)

n = number of unique characters

If there are n nodes, extractMin() is called 2(n-1) times. extractMin() takes O(log n) time as it calls minHeapify(). So, overall complexity is O(n log n).

Job Sequencing Problem

Problem Statement

Problem: Given an array of jobs where every job has a deadline and associated profit if the job is finished before the deadline. It is also given that every job takes single unit of time, so the minimum possible deadline for any job is 1. How to maximize total profit if only one job can be scheduled at a time.

Example:

Job ID	A	B	C	D	E
Deadline	2	1	2	1	3
Profit	100	19	27	25	15

C → A → E

Solution

Greedy Approach

1. Sort all jobs in decreasing order of profit.

2. Initialize the result sequence as first job in sorted jobs.

3. Do following for remaining n-1 jobs.
   - If the current job can fit in the current result sequence without missing the deadline, add current job to the result, else ignore the current job.

Solution

Greedy Approach

1. Sort all jobs in decreasing order of profit.

Job ID	A	B	C	D	E
Deadline	2	1	2	1	3
Profit	100	19	27	25	15

Sorted

Job ID	A	B	C	D	E
Deadline	2	2	1	1	3
Profit	100	27	25	19	15

Solution

Greedy Approach

2. Initialize the result sequence as first job in sorted jobs.

Job ID	A	B	C	D	E
Deadline	2	2	1	1	3
Profit	100	27	25	19	15

0   1   2   3   4   5

Answer

A

Solution

Greedy Approach

3. Do following for remaining n-1 jobs.
   - If the current job can fit in the current result sequence without missing the deadline, add current job to the result, else ignore the current job.

Job ID	A	B	C	D	E
Deadline	2	2	1	1	3
Profit	100	27	25	19	15

```
        0        1        2        3        4        5
Answer  |   B    |   A    |   E    |        |        |
```

Implementation

```cpp
// A structure to represent a job
struct Job
{
    char id;       // Job Id
    int dead;      // Deadline of job
    int profit;    // Profit if job is over before or on deadline
};

// This function is used for sorting all jobs according to profit
bool comparison(Job a, Job b)
{
    return (a.profit > b.profit);
}
```

```cpp
// Returns minimum number of platforms reqquired
void printJobScheduling(Job arr[], int n)
{
    // Sort all jobs according to decreasing order of prfit
    sort(arr, arr+n, comparison);

    int result[n]; // To store result (Sequence of jobs)
    bool slot[n];  // To keep track of free time slots

    // Initialize all slots to be free
    for (int i=0; i<n; i++)
        slot[i] = false;

    // Iterate through all given jobs
    for (int i=0; i<n; i++)
    {
        // Find a free slot for this job (Note that we start
        // from the last possible slot)
        for (int j=min(n, arr[i].dead)-1; j>=0; j--)
        {
            // Free slot found
            if (slot[j]==false)
            {
                result[j] = i;  // Add this job to result
                slot[j] = true; // Make this slot occupied
                break;
            }
        }
    }

    // Print the result
    for (int i=0; i<n; i++)
        if (slot[i])
            cout << arr[result[i]].id << " ";
}
```

Complexity

Time Complexity: $O(n^2)$

n = number of jobs

Fractional Knapsack Problem

Problem
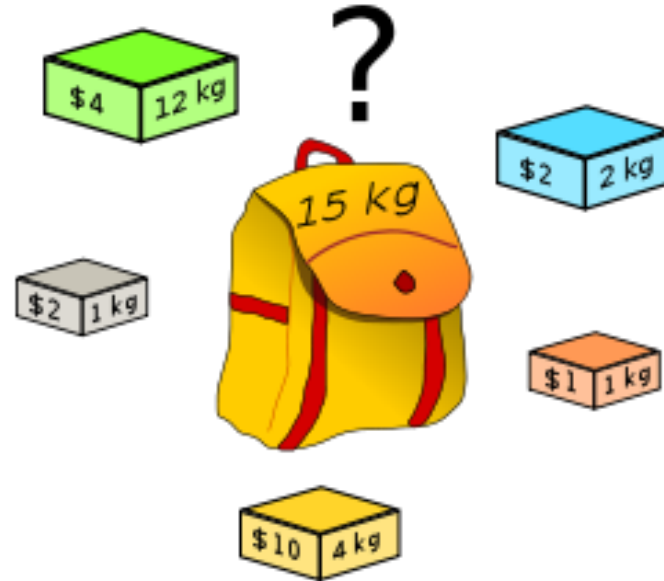
Goal: Fill knapsack such that value is maximum and total weight is at most W. Items can be broken down to maximize the knapsack value.
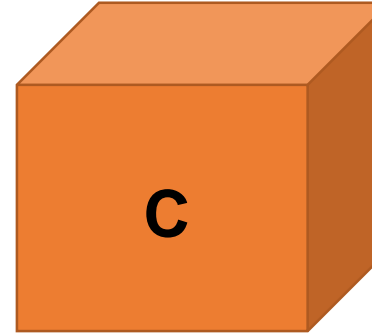
Example

A
Wt = 10
Value = 60

B
Wt = 20
Value = 100
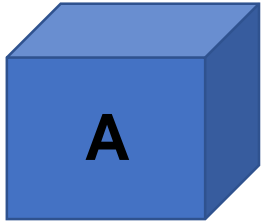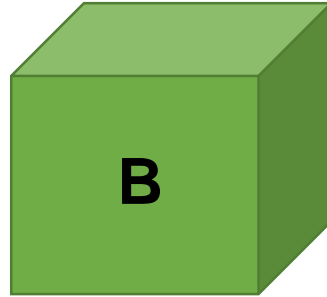
C
Wt = 30
Value = 120

Capacity=50

0-1 Knapsack

- Take B and C
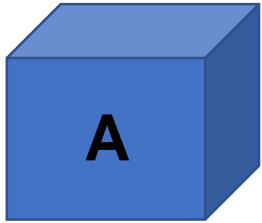- Total Weight = 20+30=50
- Total Value = 100+120=220

Fractional Knapsack

- Take A, B and 2/3 of C
- Total Weight = 10 +20+30*2/3=50
- Total Value = 60+100+120*2/3=240

Approach

Greedy Approach

1. Calculate the ratio (value/weight) for each item.

2. Sort the items based on this ratio.

3. Take the item with highest ratio and add them until we can't add the next item as whole.

4. At the end add the next item as much(fraction) as we can.

Approach
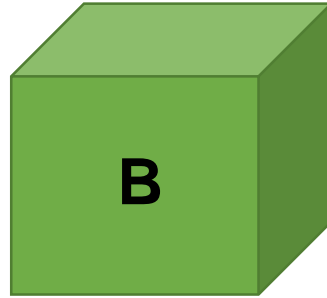
A	**B**	**C**	
Wt = 10	Wt = 20	Wt = 30	Capacity=50
Value = 60	Value = 100	Value = 120	
Ratio = 6	Ratio = 5	Ratio = 4	
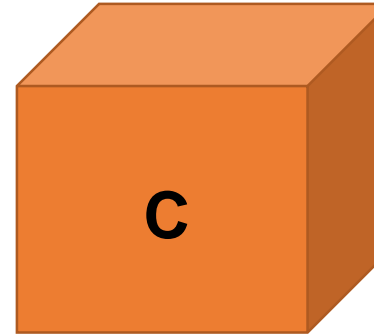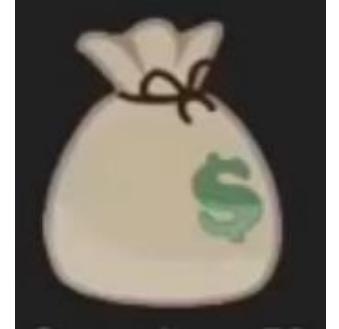
1. Calculate the ratio (value/weight) for each item.

Approach

A

Wt = 10
Value = 60

Ratio = 6

B

Wt = 20
Value = 100

Ratio = 5

C
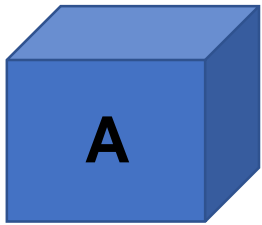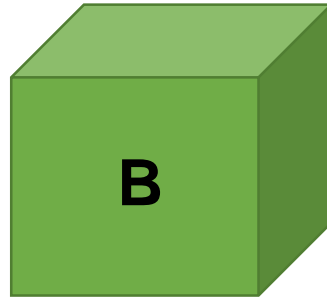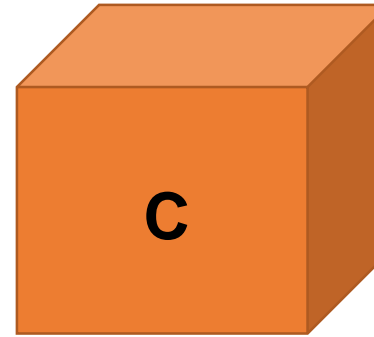
Wt = 30
Value = 120
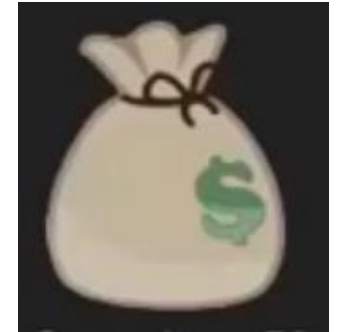
Ratio = 4

Capacity=50

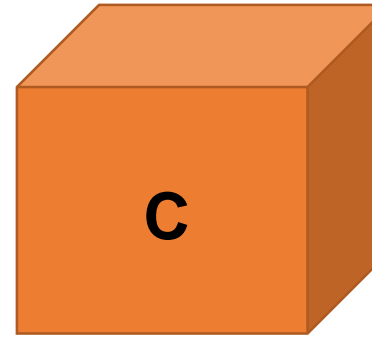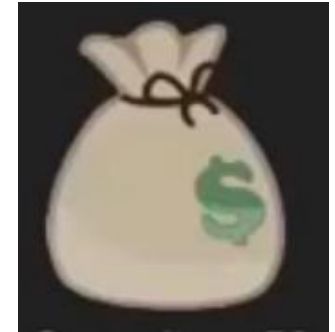2. Sort (descending) the items based on this ratio.

Approach
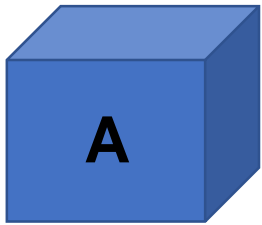
A
Wt = 10
Value = 60

Ratio = 6

B
Wt = 20
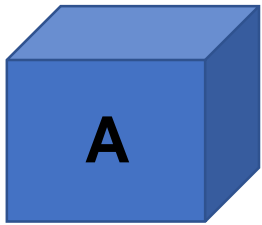Value = 100
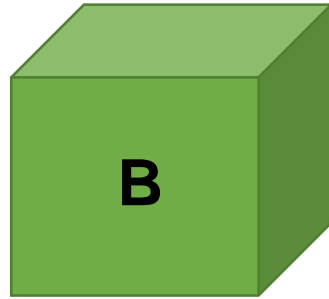
Ratio = 5

C
Wt = 30
Value = 120
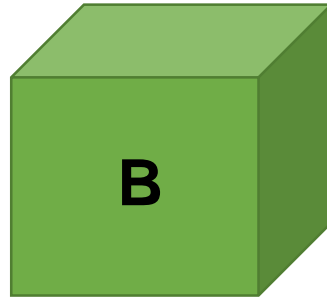
Ratio = 4

Capacity=50

Cap Left = 40
Value = 60

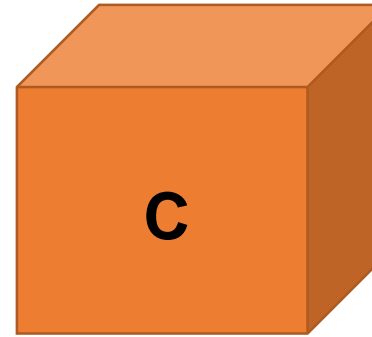3. **Take the item with highest ratio and add them until we can't add the next item as whole.**

Approach

A
Wt = 10
Value = 60

Ratio = 6

B
Wt = 20
Value = 100

Ratio = 5

C
Wt = 30
Value = 120

Ratio = 4

Capacity=50

Cap Left = 40
Value = 60

Cap Left = 20
Value = 160

3. Take the item with highest ratio and add them until we can't add the next item as whole.
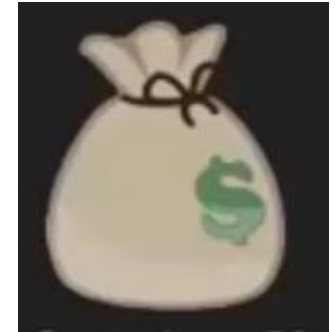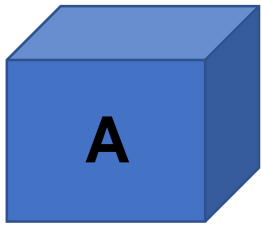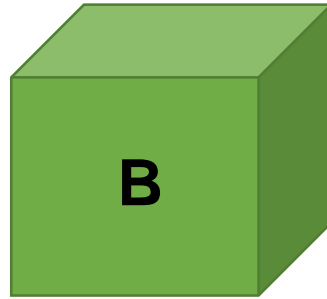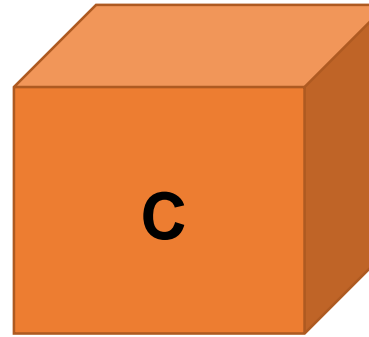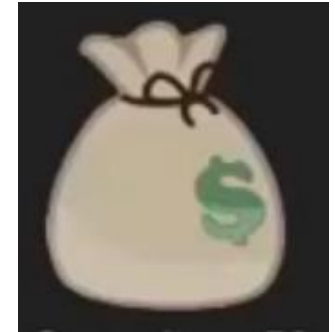
Approach

A
Wt = 10
Value = 60

Ratio = 6

B
Wt = 20
Value = 100

Ratio = 5

C
Wt = 30
Value = 120

Ratio = 4

Capacity=50

Cap Left = 40
Value = 60

Cap Left = 20
Value = 160

4. At the end add the next item as much(fraction) as we can.

Take 2/3 of C
Weight: 2/3 * 30 = 20
Value: 2/3 * 120 = 80

Cap Left = 0
Value = 240

Implementation

```cpp
double fractionalKnapsack(int W, struct Item arr[], int n)
{
    //    sorting Item on basis of ratio
    sort(arr, arr + n, cmp);

    //    Uncomment to see new order of Items with their ratio
    /*
    for (int i = 0; i < n; i++)
    {
        cout << arr[i].value << "  " << arr[i].weight << " : "
             << ((double)arr[i].value / arr[i].weight) << endl;
    }
    */

    int curWeight = 0;  // Current weight in knapsack
    double finalvalue = 0.0; // Result (value in Knapsack)

    // Looping through all Items
    for (int i = 0; i < n; i++)
    {
        // If adding Item won't overflow, add it completely
        if (curWeight + arr[i].weight <= W)
        {
            curWeight += arr[i].weight;
            finalvalue += arr[i].value;
        }

        // If we can't add current Item, add fractional part of it
        else
        {
            int remain = W - curWeight;
            finalvalue += arr[i].value * ((double) remain / arr[i].weight);
            break;
        }
    }

    // Returning final value
    return finalvalue;
}
```

Complexity

Time Complexity: O(n log n)

n = number of items

Greedy Algorithm vs Dynamic Programming

- **<u>Greedy algorithm</u>: Greedy algorithm is one which finds the feasible solution at every stage with the hope of finding global optimum solution.**

- **<u>Dynamic Programming</u>: Dynamic programming is one which breaks up the problem into series of overlapping sub-problems.**

Greedy Algorithm vs Dynamic Programming

1. Greedy algorithm never reconsiders its choices whereas Dynamic programming may consider the previous state.

2. Greedy algorithm have a local choice of the sub-problems whereas Dynamic programming would solve the all sub-problems and then select one that would lead to an optimal solution.

3. Greedy algorithm take decision in one time whereas Dynamic programming take decision at every stage.

Greedy Algorithm vs Dynamic Programming

4. Greedy algorithm work based on choice property whereas Dynamic programming work based on principle of optimality.

5. Greedy algorithm follows the top-down strategy whereas Dynamic programming follows the bottom-up strategy.

Reference

- Charles Leiserson and Piotr Indyk, *"Introduction to Algorithms",* September 29, 2004

- https://www.geeksforgeeks.org