Greedy Algorithm

Course Code: CSC2211 Course Title: Algorithms

Dept. of Computer Science
Faculty of Science and Technology

Lecturer No:	**06**	**Week No:**	**06**	**Semester:**	
Lecturer:	*Name & email*				

Lecture Outline

1. Optimization Problem
2. Greedy Algorithm.
3. Coin Changing Problem.
4. Fractional knapsack Problem.
5. Huffman Encoding.
6. Activity Selection Problem.

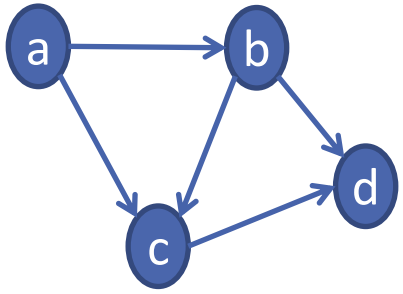
Optimization Problems

- An optimization problem is one in which you want to find, not just *a* solution, but the *best* solution

Example

- Graph coloring *optimization problem*: given an undirected graph, G= (V,E), what is the minimum number of colors required to assign "colors" to each vertex in such a way that no two adjacent vertices have the same color

- Path *optimization problem*: Find the shortest path(s) from u to v in a given graph G.



Find all the **shortest** paths from a to d:
a →b →d
a →c →d

Greedy Algorithm

- Solves an optimization problem
- For many optimization problems, greedy algorithm can be used. (not always)
- Greedy algorithm for optimization problems typically go through a sequence of steps, with a set of choices at each step. Current choice does not depend on evaluating potential future choices or pre-solving repeatedly occurring subproblems (a.k.a., *overlapping subproblems*). With each step, the original problem is reduced to a smaller problem.
- Greedy algorithm always makes the choice that looks best at the moment.
- It makes a locally optimal choice in the hope that this choice will lead to a globally optimal solution.

Optimal Solution

What is an Optimal Solution?

- Given a problem, more than one solution exist
- One of the solution is the best based on some given constraints, that solution is called the optimal solution

What is Global Optimal Solution?

- Optimal Solution to the main problem

What is local Optimal Solution?

- Optimal Solution to the sub-problems

Example of Greedy Algorithms

- Activity Selection Problem
- Coin Changing Problem
- Job Scheduling Problem
- Fractional Knapsac Problem
- Dijkstra's Shortest Path Problem
- Minimum Spanning Tree Problem

Coin changing problem

Definition

Given coin denominations in $\{C\}$, make change for a given amount A with the minimum number of coins.

Example:

Coin denominations, $C = \{25, 10, 5, 1\}$ Amount to change, $A = 73$

Choose 2 25 coins, so remaining is $73 - 2 * 25 = 23$

Choose 2 10 coins, so remaining is $23 - 2 * 10 = 3$

Choose 0 5 coins, so remaining is 3

Choose 3 1 coins, so remaining is $3 - 1 * 3 = 0$

Solution (and it's optimal): $2 \times 25 + 2 \times 10 + 3 \times 1 = 7$ coins

Key Question

Does a greedy approach always produce the optimal solution?

Coin changing problem (continued)

Coin denominations, $C = \{12, 5, 1\}$ Amount to change, $A = 15$

Example (using greedy strategy)

Choose 1 12 coins, so remaining is $15 - 1 * 12 = 3$
Choose 3 1 coins, so remaining is $3 - 1 * 3 = 0$

Solution: 4 coins.

Example (using optimal strategy)

Choose 0 12 coins, so remaining is 15
Choose 3 5 coins, so remaining is $15 - 3 * 5 = 0$

Solution: 3 coins.

Correctness depends on the choice of coins,
so greedy strategy does not provide a general solution to this problem!

Fractional knapsack problem

Definition (fractional knapsack problem)

Given a set S of n items, such that each item i has a positive benefit b_i and a positive weight w_i, the goal is to find the maximum-benefit subset that does not exceed a given weight W, allowing for fractional items.

Key question

- What strategy to use to select the next item (and the amount of it)?

Fractional knapsack problem

Capacity
W=20

Item	Benefit	Weight
A	25	18 kg
B	15	10 kg
C	24	15 kg

First Strategy: Greedy method using **Profit** as it's measure. At each step it will choose an object that increases the profit the **most**.

Fractional knapsack problem

Item	Benefit	Weight
A	25	18 kg
B	15	10 kg
C	24	15 kg

- Note here, If we select C as first element and B as second element then,
- Total benefit= $24 + 15(5/10) = 31.5$
- So, previous solution is not the optimal one.
- Thus, First Strategy fails.

Fractional knapsack problem

Second Strategy: Greedy method using **Capacity** as it's measure will, at each step choose an object that increases the capacity the **least**.

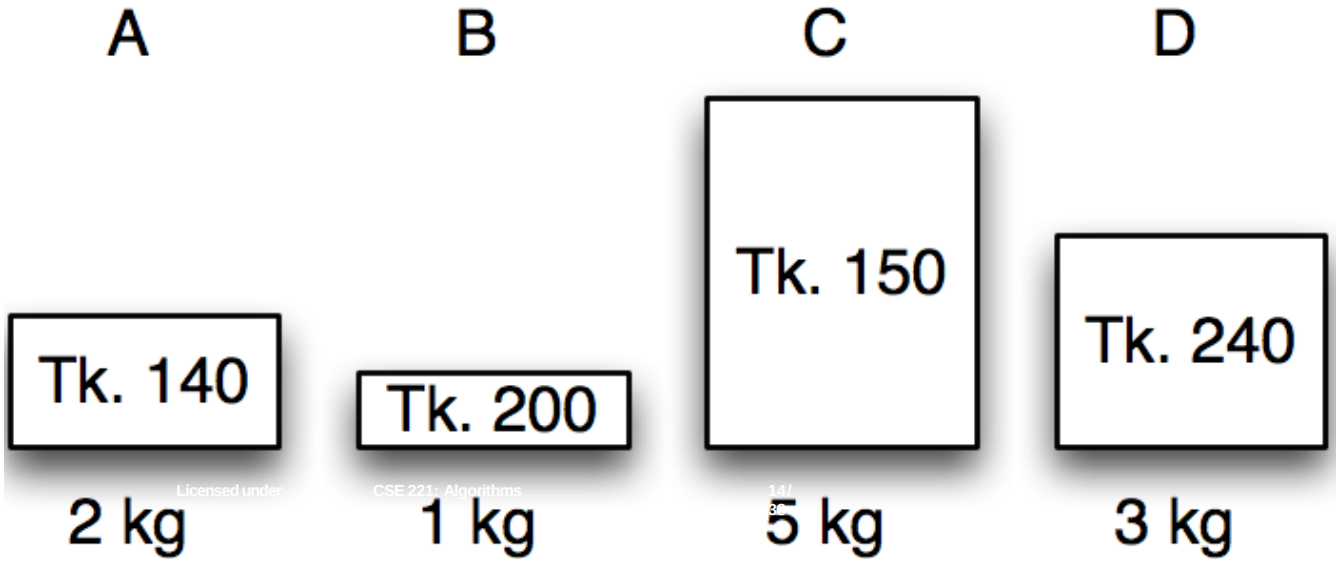
- B has the least weight 10. So select this. Remaining Capacity: $20 - 10 = 10$

Fractional knapsack problem

Third Strategy: Strives to achieve a balance between the rate at which profit increases and the rate at which capacity is used.

- At each step, include the object which has the maximum profit per unit of capacity used.
- That means, objects are considered in order of the ratio b_i/w_i.

Fractional knapsack in action

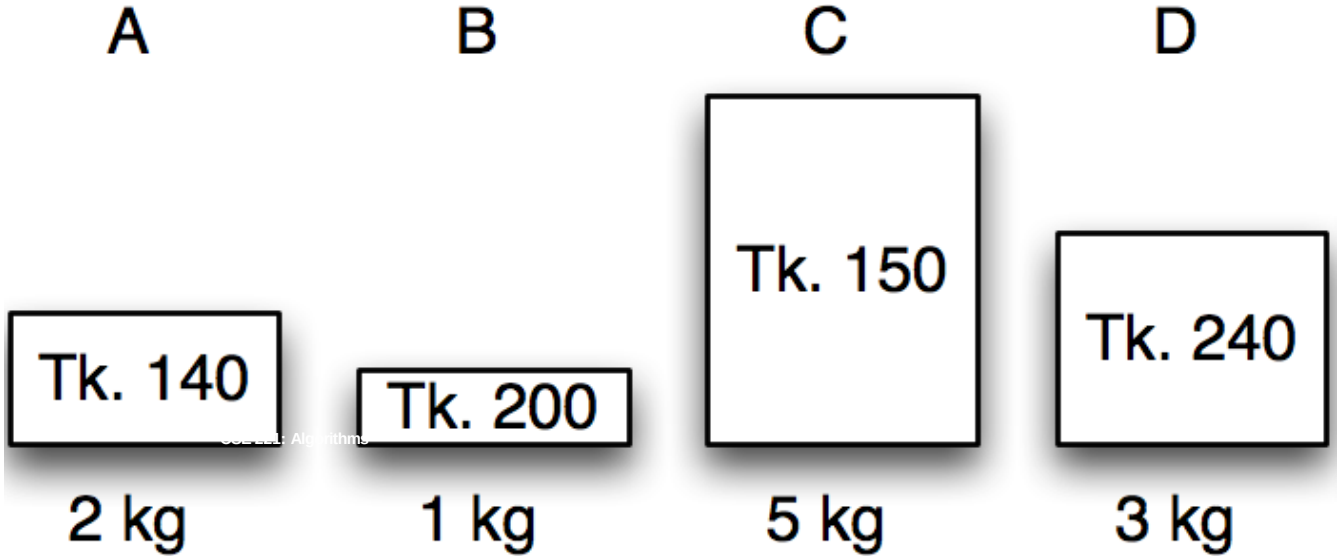


Licensed under

Item	Benefit	Weight
A	140	2 kg
B	200	1 kg
C	150	5 kg
D	240	3 kg

Calculate benefit/kg – the value index.

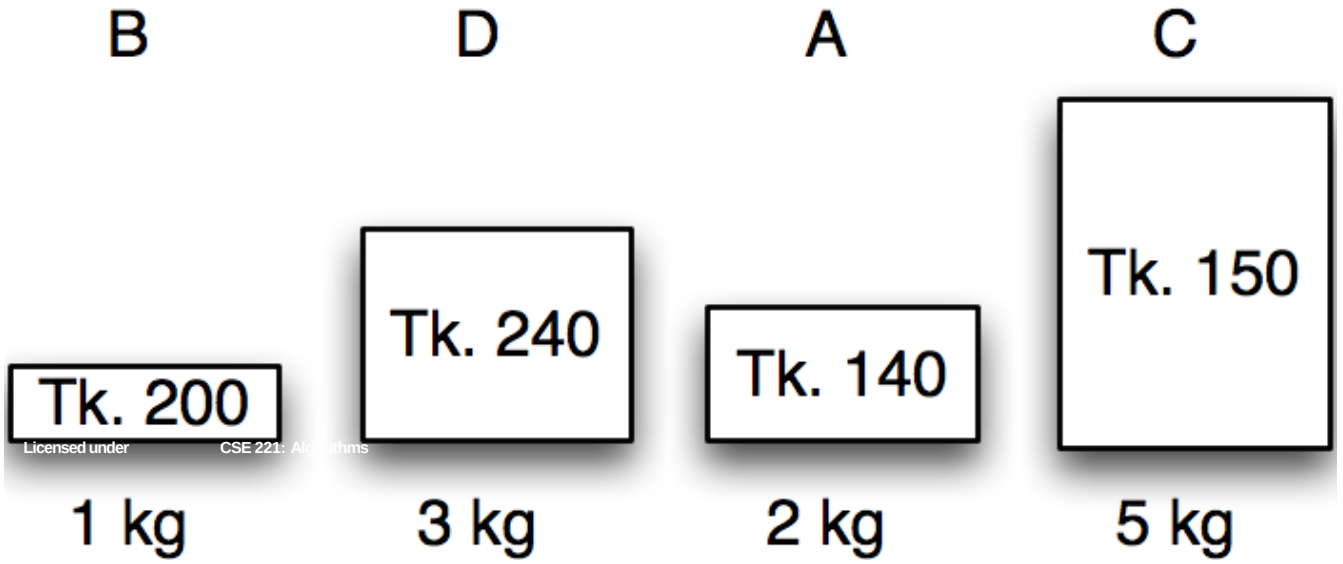
Fractional knapsack in action



Item	Benefit	Weight	Value index
A	140	2 kg	70
B	200	1 kg	200
C	150	5 kg	30
D	240	3 kg	80

Sort by non-increasing value index.

Fractional knapsack in action

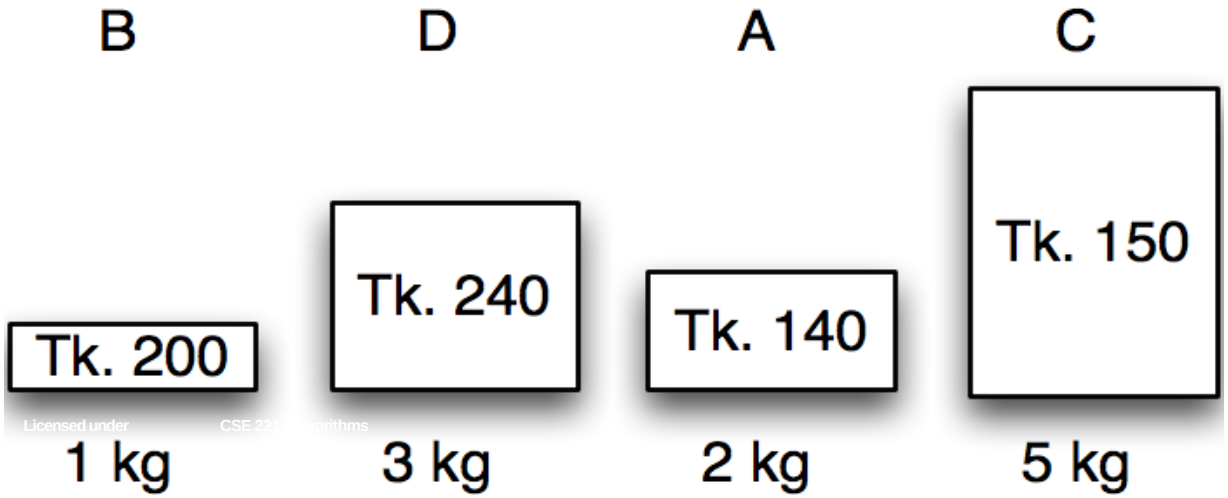


Licensed under CSE 221: Algorithms

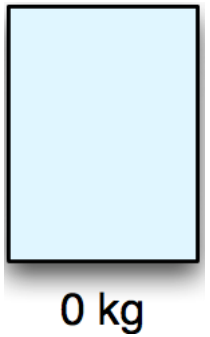
Item	Benefit	Weight	Value index
B	200	1 kg	200
D	240	3 kg	80
A	140	2 kg	70
C	150	5 kg	30

Maximum weight: 5 kg

Fractional knapsack in action

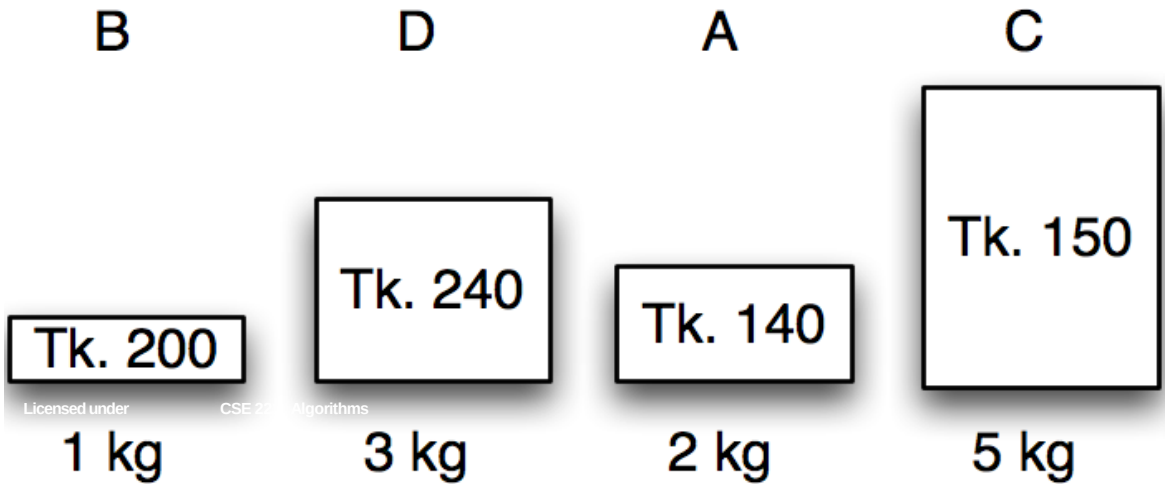


Item	Benefit	Weight	Value index	Chosen
B	200	1 kg	200	0 kg
D	240	3 kg	80	0 kg
A	140	2 kg	70	0 kg
C	150	5 kg	30	0 kg

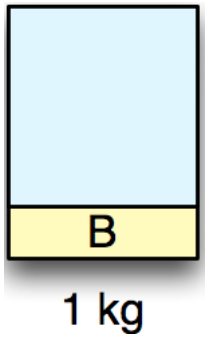


Maximum weight: 5 kg Remaining: 5 kg Benefit: 0 kg

Fractional knapsack in action

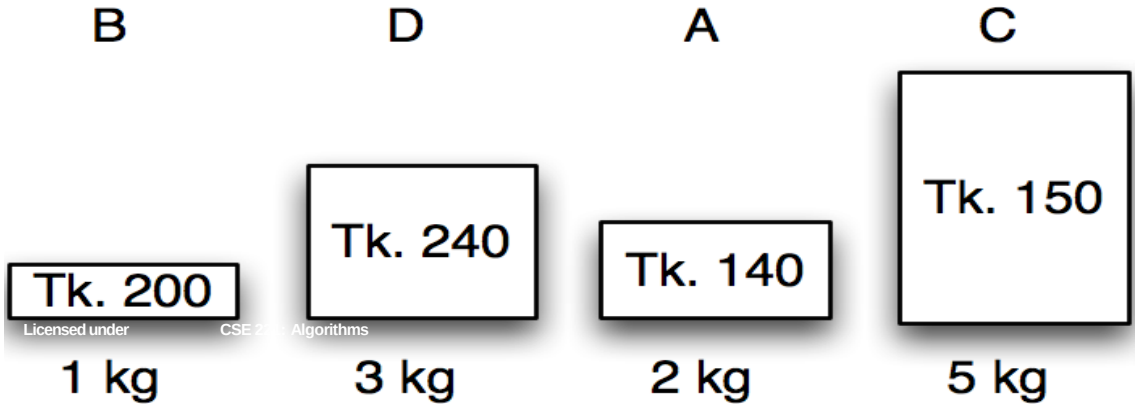


Item	Benefit	Weight	Value index	Chosen
B	200	1 kg	200	1 kg
D	240	3 kg	80	0 kg
A	140	2 kg	70	0 kg
C	150	5 kg	30	0 kg



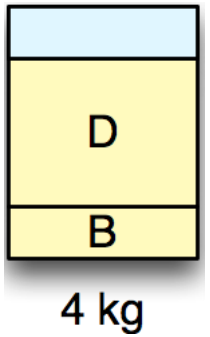
Maximum weight: 5 kg Remaining: 4 kg Benefit: 200 kg

Fractional knapsack in action



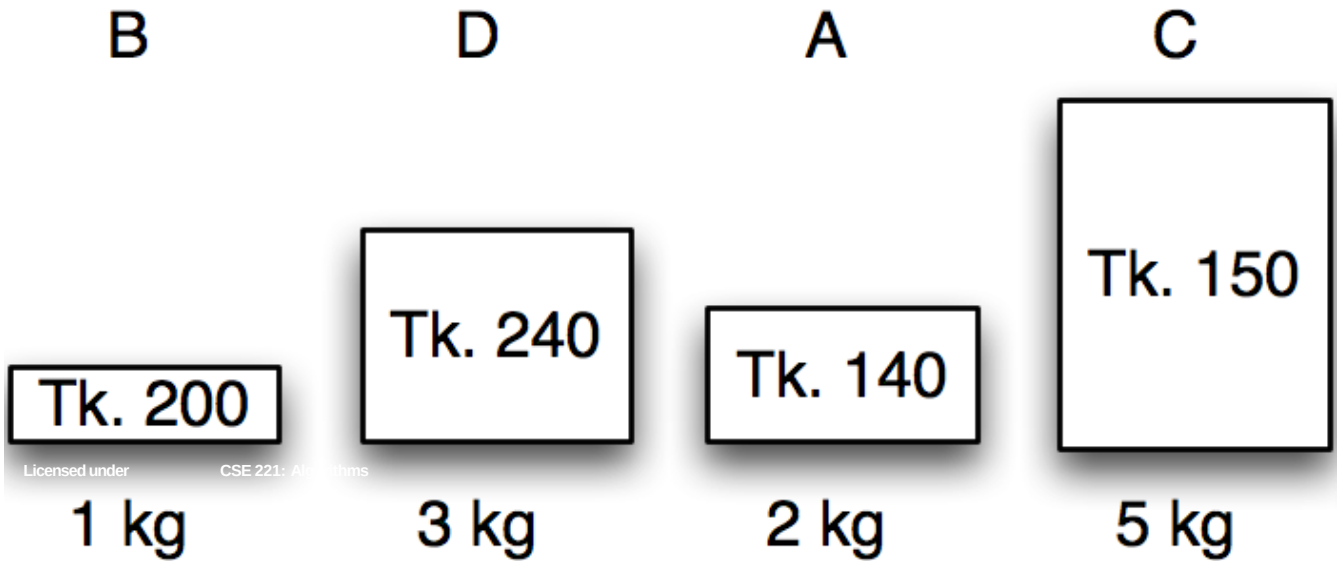
Licensed under CSE 2 Algorithms

Item	Benefit	Weight	Value index	Chosen
B	200	1 kg	200	1 kg
D	240	3 kg	80	3 kg
A	140	2 kg	70	0 kg
C	150	5 kg	30	0 kg

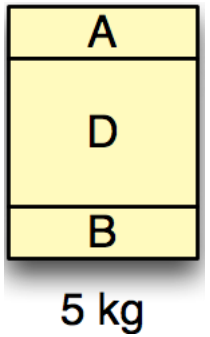


Maximum weight: 5 kg Remaining: 1 kg Benefit: 440 kg

Fractional knapsack in action

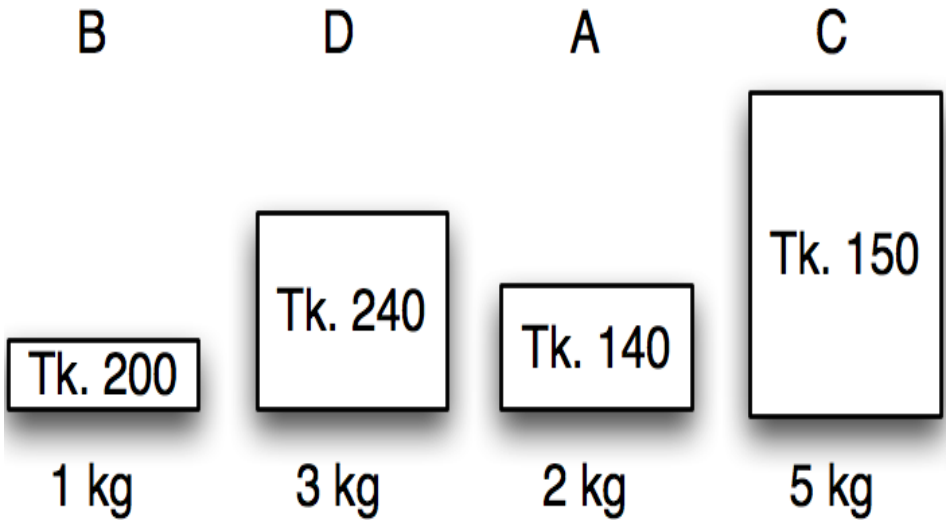


Item	Benefit	Weight	Value index	Chosen
B	200	1 kg	200	1 kg
D	240	3 kg	80	3 kg
A	140	2 kg	70	1 kg
C	150	5 kg	30	0 kg

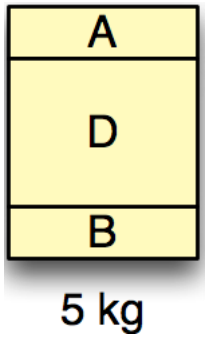


Maximum weight: 5 kg Remaining: 0 kg Benefit: 510 kg

Fractional knapsack in action



Item	Benefit	Weight	Value index	Chosen
B	200	1 kg	200	1 kg
D	240	3 kg	80	3 kg
A	140	2 kg	70	1 kg
C	150	5 kg	30	0 kg



Maximum weight: 5 kg Remaining: 0 kg Benefit: 510 kg

Fractional knapsack greedy algorithm

Algorithm *fractionalKnapsack(S, W)*
Input: set S of items w/ benefit b_i and weight w_i; max. weight W
Output: amount x_i of each item i to maximize benefit w/ weight at most W

```
for each item i in S
    x_i ← 0
    v_i ← b_i / w_i        {value}
w ← 0                      {total weight}
while w < W
    remove item i with highest v_i
    x_i ← min{w_i , W - w}
    w ← w + x_i
```

Fractional knapsack algorithm

Running time:

Given a collection S of n items, such that each item i has a benefit b_i and weight w_i, we can construct a maximum-benefit subset of S, allowing for fractional amounts, that has a total weight W in O(nlogn) time. (how?)

Use heap-based priority queue to store S

Removing the item with the highest value takes O(logn) time

In the worst case, need to remove all items

Exercise

Assume that we have a knapsack with max weight capacity, W = 16. our objective is to fill the knapsack with items such that the benefit (value or profit) is maximum.

Consider the following items and their associated weight and value

ITEM	WEIGHT	VALUE
i1	6	6
i2	10	2
i3	3	1
i4	5	8
i5	1	3
i6	3	5

Huffman Codes

Widely used technique for data compression

Assume the data to be a sequence of characters

Looking for an effective way of storing the data

Binary character code

Uniquely represents a character by a binary string

Fixed-Length Codes

E.g.: Data file containing 100,000 characters

	a	b	c	d	e	f
Frequency (thousands)	45	13	12	16	9	5

3 bits needed

a = 000, b = 001, c = 010, d = 011, e = 100, f = 101

Requires: 100,000 · 3 = 300,000 bits

Idea: ***Use the frequencies of occurrence of characters to build a optimal way of representing each character***

	a	b	c	d	e	f
Frequency (thousands)	45	13	12	16	9	5

Variable-Length Codes

E.g.: Data file containing 100,000 characters

	a	b	c	d	e	f
Frequency (thousands)	45	13	12	16	9	5

Assign short codewords to frequent characters and long codewords to infrequent characters

a = 0, b = 101, c = 100, d = 111, e = 1101, f = 1100

$(45 \cdot 1 + 13 \cdot 3 + 12 \cdot 3 + 16 \cdot 3 + 9 \cdot 4 + 5 \cdot 4) \cdot 1{,}000$

$= 224{,}000$ bits

Prefix Codes

Prefix codes:

Codes for which no codeword is also a prefix of some other codeword

Better name would be “prefix-free codes”

We can achieve optimal data compression using prefix codes

We will restrict our attention to prefix codes

Encoding with Binary Character Codes

Encoding

Concatenate the codewords representing each character in the file

E.g.:

a = 0, b = 101, c = 100, d = 111, e = 1101, f = 1100

abc = 0 · 101 · 100 = 0101100

Decoding with Binary Character Codes

Prefix codes simplify decoding

- No codeword is a prefix of another $\Rightarrow$ the codeword that begins an encoded file is unambiguous

Approach

- Identify the initial codeword
- Translate it back to the original character
- Repeat the process on the remainder of the file

E.g.:

a = 0, b = 101, c = 100, d = 111, e = 1101, f = 1100

001011101 =

$$0 \cdot 0 \cdot 101 \cdot 1101 = \text{aabe}$$

Constructing a Huffman Code

A greedy algorithm that constructs an optimal prefix code called a **Huffman code**

Assume that:

- C is a set of n characters
- Each character has a frequency $f(c)$
- The tree T is built in a bottom up manner
- Left means '0', right means '1'
- More frequent characters will be higher in the tree

Idea:

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

- Start with a set of $|C|$ leaves
- At each step, merge the two least frequent objects: the frequency of the new node = sum of two frequencies
- Use a min-priority queue Q, keyed on f to identify the two least frequent objects

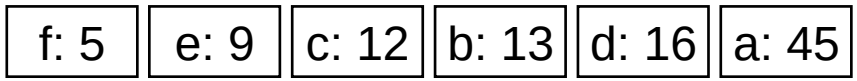
Constructing a Huffman tree

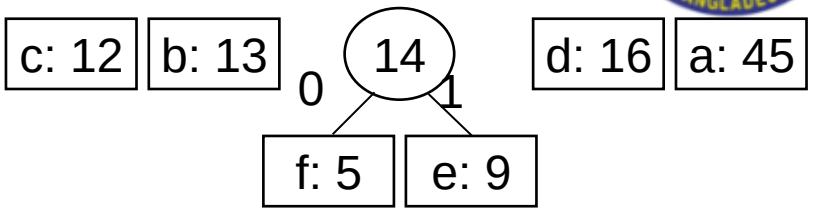
Alg.: HUFFMAN(C)

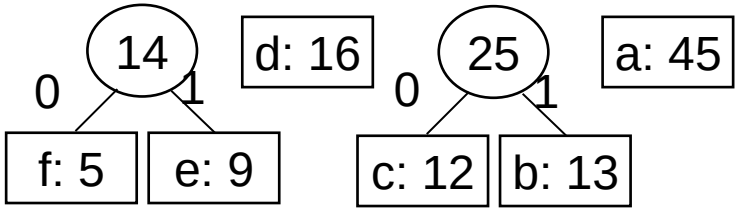
Running time: $O(n \lg n)$

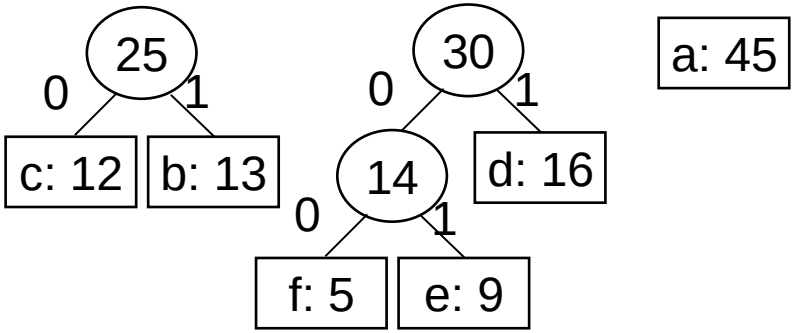
```
1.  n ← |C|
2.  Q ← C
3.  for i ← 1 to n – 1                          ← O(n)
4.      do allocate a new node z
5.          left[z] ← x ← EXTRACT-MIN(Q)
6.          right[z] ← y ← EXTRACT-MIN(Q)       } O(lgn)
7.          f[z] ← f[x] + f[y]
8.          INSERT (Q, z)
9.  return EXTRACT-MIN(Q)
```

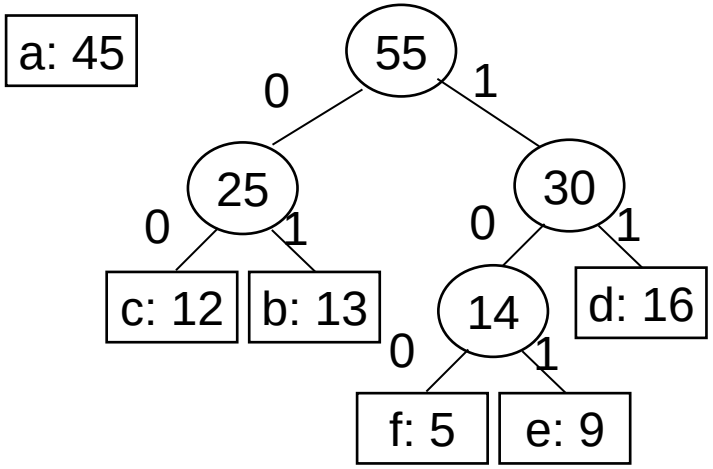
Example

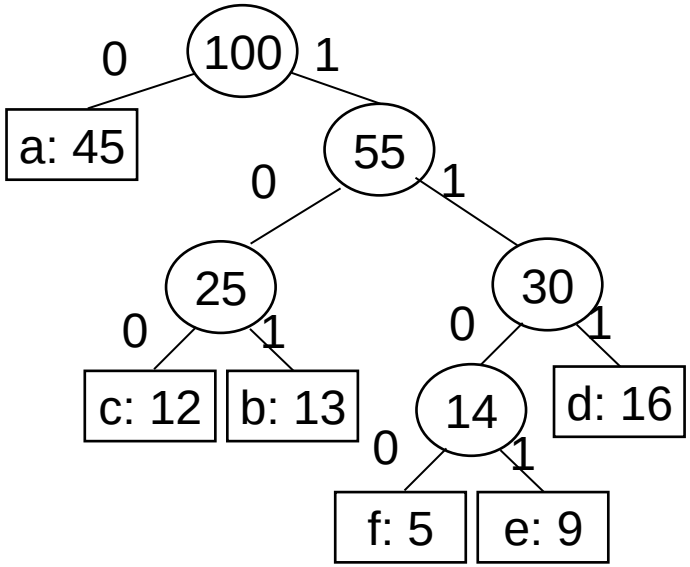

f: 5
e: 9
c: 12
b: 13
d: 16
a: 45


c: 12
b: 13
14
0
1
f: 5
e: 9
d: 16
a: 45


14
0
1
f: 5
e: 9
d: 16
25
0
1
c: 12
b: 13
a: 45


25
0
1
c: 12
b: 13
30
0
1
14
0
1
f: 5
e: 9
d: 16
a: 45


a: 45
55
0
1
25
0
1
c: 12
b: 13
30
0
1
14
0
1
f: 5
e: 9
d: 16


0
100
1
a: 45
55
0
1
25
0
1
c: 12
b: 13
30
0
1
14
0
1
f: 5
e: 9
d: 16

Huffman encoding/decoding using Huffman tree

Encoding:

- Construct Huffman tree using the previous algorithm
- Traverse the Huffman tree to assign a code to each leaf (representing an input character)
- Use these codes to encode the file

Decoding:

- Construct Huffman tree using the previous algorithm
- Traverse the Huffman tree according to the bits you encounter until you reach a leaf node at which point you output the character represented by that leaf node.
- Continue in this fashion until all the bits in the file are read.

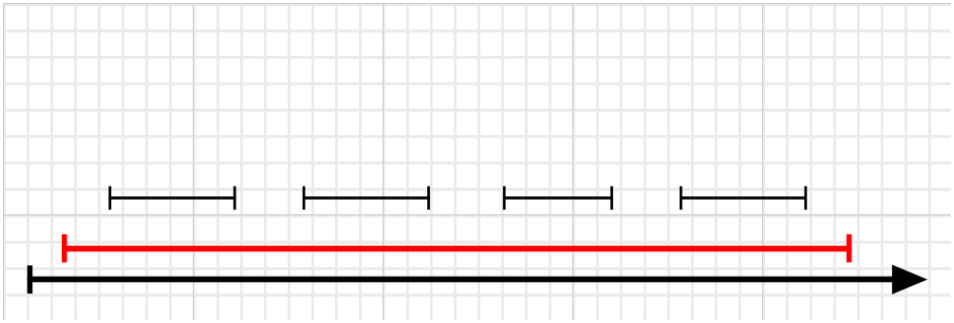
Activity-Selection Problem

- Problem: get your money's worth out of a festival
  - Buy a wristband that lets you onto any ride
  - Lots of rides, each starting and ending at different times
  - Your goal: ride as many rides as possible
    - Another, alternative goal that we don't solve here: maximize time spent on rides
- Welcome to the *activity selection problem*

Strategy 1:

The idea is to start using the resource as early as possible.

1. Sort the activities by starting time, breaking ties arbitrarily.
2. Pick the first one, removing it from the list along with all the activities that conflict with it.
3. Repeat Step 2, until the list is empty.



Number Of Activity = 1

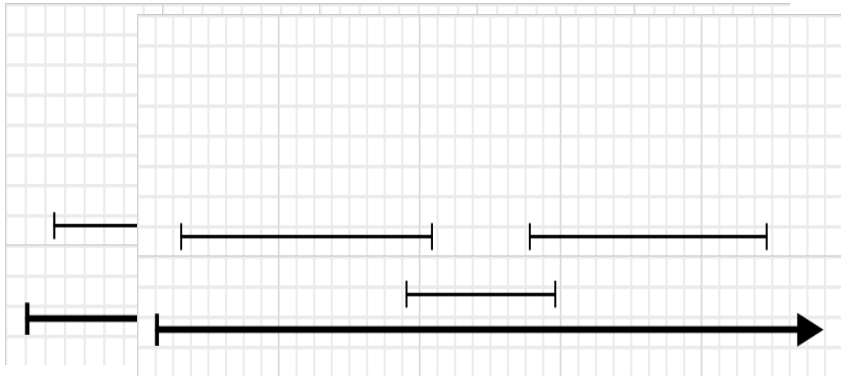
This strategy does not lead to an optimal solution.

Strategy 2:

The idea is to start shortest activity.

1. Sort the activities by length, breaking ties arbitrarily.
2. Pick the first one, removing it from the list along with all the activities that conflict with it.

Repeat Step 2, until the list is empty.



Number Of Activity = ?
Number Of Activity = 1

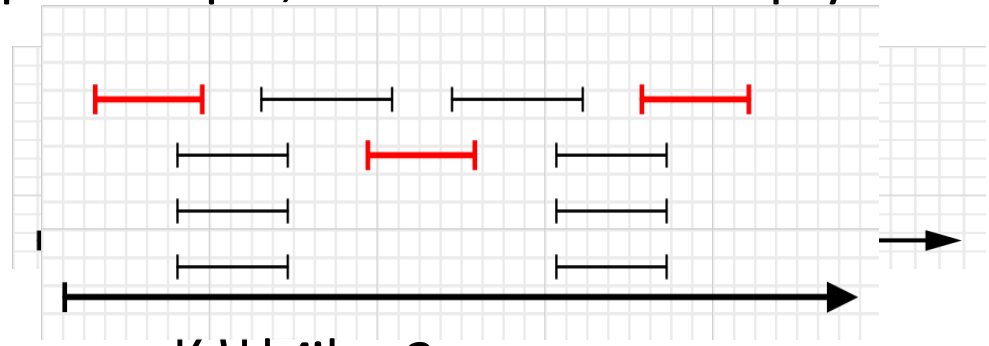
This strategy does not lead to an optimal solution.

Strategy 3:

The idea is to start with least-conflict activity.

The *Shortest First* strategy failed perhaps because the shorter ones had more conflicts, and ruled out too many activities in the process.

1. Sort the activities by the number of other activities which conflict with it.
2. Pick the first one, removing it from the list along with all the activities that conflict with it.
3. Repeat Step 2, until the list is empty.



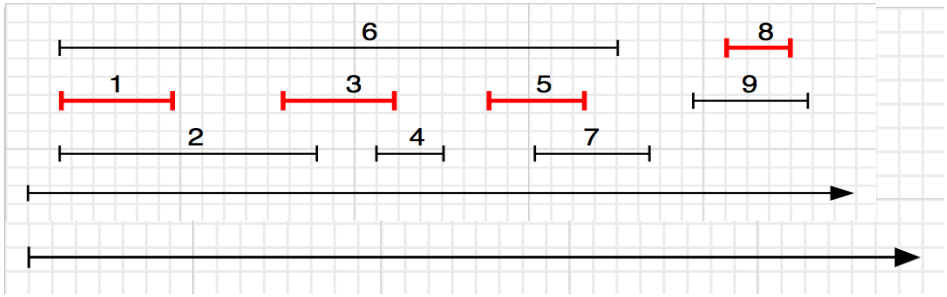
$|A| = 3$.

This strategy does not lead to an optimal solution.

Strategy 4:

The idea is to start with finish-first activity.

1. Sort the activities by finishing time, breaking ties arbitally.
2. Pick the first one, removing it from the list along with all the activities that conflict with it.
3. Repeat Step 2, until the list is empty.



$|A| = 4$.

This strategy is the one that works.

Example

Here are a set of start and finish times

i	1	2	3	4	5	6	7	8	9	10	11
S_j	1	5	0	3	6	5	3	12	8	2	8
f_j	4	7	6	5	10	9	9	16	12	14	11

Sorted according to finishing time

i	1	2	3	4	5	6	7	8	9	10	11
S_j	1	3	0	5	3	5	6	8	8	2	12
f_j	4	5	6	7	9	9	10	11	12	14	16

- {1, 4, 8, 11} which is a larger set (*an optimal solution*)
- Solution is not unique, consider {2, 4, 9, 11} (another optimal solution)

Activity selection Algorithm

```
GREEDY-ACTIVITY-SELECTOR(s, f)
1   n ← length[s]
2   A ← {a_1}
3   i ← 1
4   for m ← 2 to n
5         do if s_m ≥ f_i
6               then A ← A ∪ {a_m}
7                    i ← m
8   return A
```

Exercise

Input: A list of different activities with starting and ending times.
{(5,9), (1,2), (3,4), (0,6), (5,7), (8,9)}

Use greedy algorithm to do maximum number of activities.

Problem types solved by greedy algorithms

- There is no general of knowing whether a problem can be solved by a greedy algorithm.

Conclusion

- Greedy algorithms often lead to polynomial time-solution for an exponential-time problem.

Books

Introduction to Algorithms, Thomas H. Cormen, Charle E. Leiserson, Ronald L. Rivest, Clifford Stein (CLRS).

Fundamental of Computer Algorithms, Ellis Horowitz, Sartaj Sahni, Sanguthevar Rajasekaran (HSR)

References

https://www.tutorialspoint.com/data_structures_algorithms/greedy_algorithms.htm

https://www.geeksforgeeks.org/fractional-knapsack-problem/

https://www.geeksforgeeks.org/activity-selection-problem-greedy-algo-1/

https://www.tutorialspoint.com/Huffman-Coding-Algorithm

https://www.geeksforgeeks.org/huffman-coding-greedy-algo-3/