COMP2123
Data structures and Algorithms

Lecture 9: The greedy method
[GT 10]

A/Prof Julian Mestre
School of Computer Science

Some content is taken from material provided by the textbook publisher Wiley.

THE UNIVERSITY OF SYDNEY

Greedy algorithms

A class of algorithms where we build a solution one step at a time making locally optimal choices at each stage in the hope of finding a global optimum solution

Some of the most elegant algorithms and the simplest to implement, but often among the hardest to design and analyze

Even when they are not optimal in theory, greedy algorithms can be the basis of a very good heuristic.

Generic form

```
def generic_greedy(input):

  # initialization
  initialize result

  determine order in which to consider input

  # iteratively make greedy choice
  for each element i of the input (in above order) do
    if element i improves result then
      update result with element i

  return result
```

The Fractional Knapsack Problem

Given: A set S of n items, with each item i having

- b_i : a positive benefit
- w_i : a positive weight

Goal: Choose items with maximum total benefit of weight at most W.

Let x_i denote the amount we take of item i

Objective: maximize $\sum_{i \in S} b_i (x_i / w_i)$ [maximize benefit]

Constraint: $\sum_{i \in S} x_i \leq W$ [total weight is bounded]

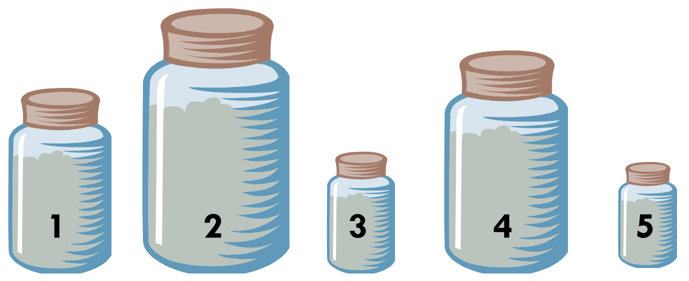
$0 \leq x_i \leq w_i$ [individual weight is bounded]

Example

Given: A set S of n items, with each item i having

- b_i - a positive benefit
- w_i - a positive weight

Goal: Choose items with maximum total benefit of weight at most W.



Items:	1	2	3	4	5
Weight:	4 ml	8 ml	2 ml	6 ml	1 ml
Benefit:	\$12	\$32	\$40	\$30	\$50

"knapsack"

10 ml

Optimal:
- 1 ml of 5
- 2 ml of 3
- 6 ml of 4
- 1 ml of 2

Total value: \$124

The Fractional Knapsack Algorithm

Initial configuration: no items chosen

Each step: identify the "best" item available and add as much as possible (all of it if you can) to the knapsack

What defines "best" choice of item to add next?

```
def fractional_knapsack(b, w, W):

  # initialization
  x ← array of size |b| of zeros
  curr ← 0

  # iteratively make greedy choice
  while curr < W do
    i ← "best" item not yet chosen
    x[i] ← min(w[i], W - curr)
    curr ← curr + x[i]
  return x
```

Different Strategies

A greedy choice: Keep taking as much as possible of the "best" item, where best means:

[highest benefit]: Select items with highest benefit.

[smallest weight]: Select items with smallest weight.

[benefit/weight]: Select items with highest benefit to weight ratio.

Each of these defines a different greedy strategy for this problem.

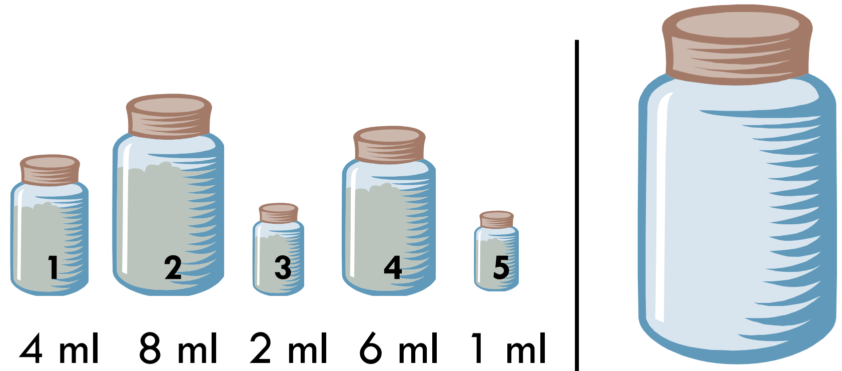
What’s “best”?

Greedy choice: Keep taking the “best” item.

[highest benefit]: Select items with highest benefit.

1 ml of 5 → \$50
2 ml of 3 → \$40
7 ml of 2 → \$28

Total value: \$118



Items:	1	2	3	4	5
Weight:	4 ml	8 ml	2 ml	6 ml	1 ml
Benefit:	\$12	\$32	\$40	\$30	\$50

10 ml

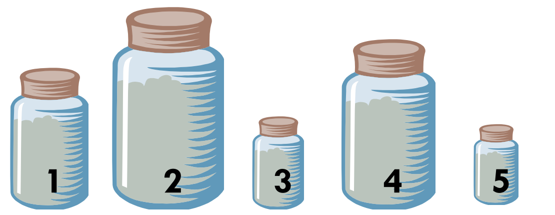
What's "best"?

Greedy choice: Keep taking the "best" item.

[smallest weight]: Select items with smallest weight.

1 ml of 5 → $50
2 ml of 3 → $40
4 ml of 1 → $12
3 ml of 4 → $15

Total value: $117



Items:	1	2	3	4	5
Weight:	4 ml	8 ml	2 ml	6 ml	1 ml
Benefit:	$12	$32	$40	$30	$50

"knapsack"

10 ml

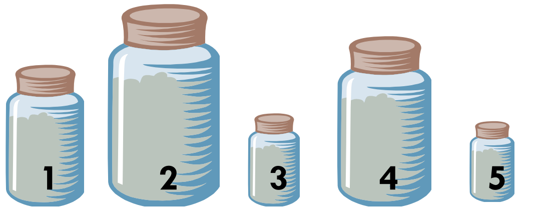
What's "best"?

Greedy choice: Keep taking the "best" item.

[benefit/weight]: Select items with highest benefit to weight ratio.

1 ml of 5 → $50
2 ml of 3 → $40
6 ml of 4 → $30
1 ml of 2 → $4

Total value: $124



Items:	1	2	3	4	5
Weight:	4 ml	8 ml	2 ml	6 ml	1 ml
Benefit:	$12	$32	$40	$30	$50
Benefit/ml:	3	4	20	5	50

"knapsack"

10 ml

The Fractional Knapsack Algorithm: Correctness

Theorem: The greedy strategy of picking item with highest benefit to weight ratio computes an optimal solution.

Proof (sketch):

- Use an exchange argument
- Assume for simplicity that all ratios are different $b_i/w_i \neq b_k/w_k$
- Consider some feasible solution x different than the greedy one
- There must be items i and k s.t. $x_i<w_i$, $x_k>0$ and $b_i/w_i > b_k/w_k$
- If we replace some k with some of i, we get a better solution
- How much? $\min\{w_i-x_i, x_k\}$
- Thus, there is no better solution than the greedy one

The Fractional Knapsack Algorithm: Complexity

Sort items by their benefit-to-weight values, and then process them in this order.

Require O(n log n) time to sort the items and then O(n) time to process them in the for-loop.

```
def fractional_knapsack(b, w, W):

  # initialization
  x ← array of size |b| of zeros
  curr ← 0

  # iteratively do greedy choice
  for i in descending b[i]/w[i] order do
    x[i] ← min(w[i], W - curr)
    curr ← curr + x[i]
  return x
```

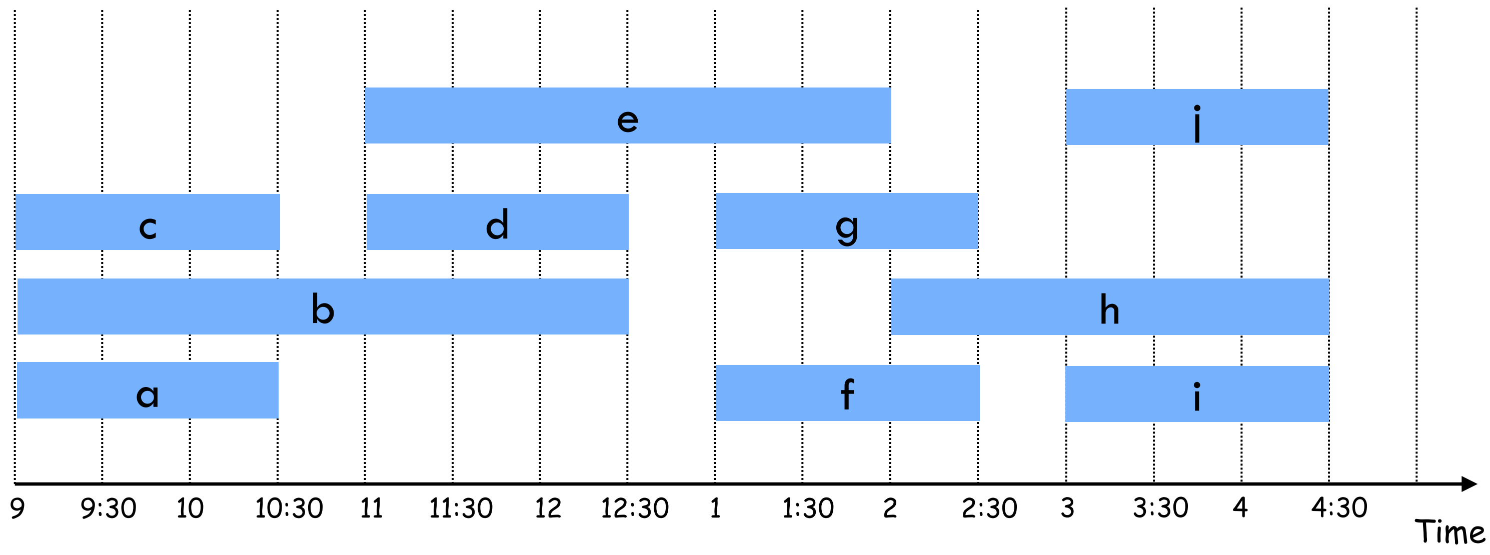
Task scheduling

Given: A set S of n lectures

Lecture i starts at s_i and finishes at f_i.

Goal: Find the minimum number of classrooms to schedule all lectures so that no two occur at the same time in the same room.

Example: This schedule uses 4 classrooms to schedule 10 lectures.



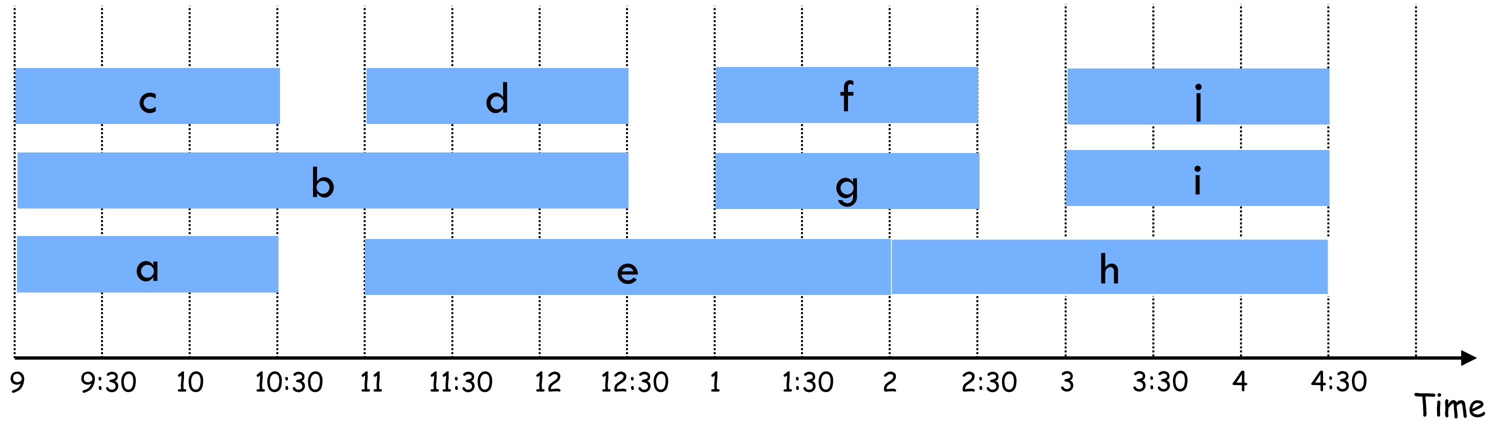
Task scheduling

Given: A set S of n lectures

Lecture i starts at s_i and finishes at f_i.

Goal: Find the minimum number of classrooms to schedule all lectures so that no two occur at the same time in the same room.

Example: This schedule uses only 3!



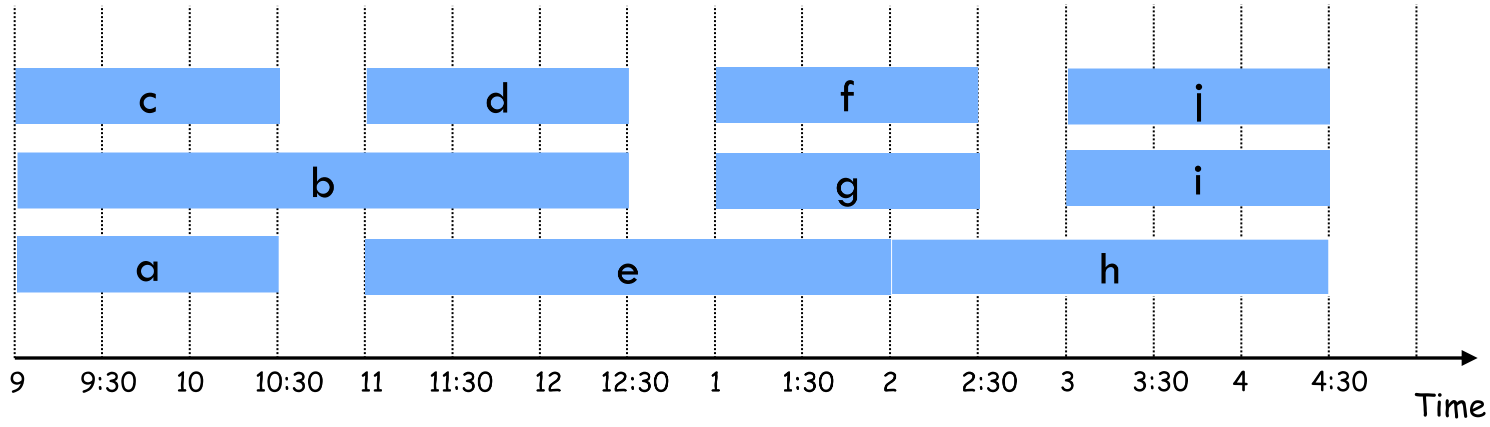
Interval Partitioning: Lower bound

Definition: The depth of a set of open intervals is the maximum number that contain any given time.

Observation: Number of classrooms needed $\geq$ depth. Why?

Example: Depth of schedule below is 3 [a, b, c all contain 9:30]
$\Rightarrow$ schedule below is optimal.

Question: Does there always exist a schedule equal to depth of intervals?



Interval Partitioning: Greedy Algorithm

Greedy algorithm: Consider lectures in increasing order of start time: assign lecture to any compatible classroom.

```
def interval_partition(S):

  # initialization
  sort intervals in increasing starting time order
  d ← 0    # number of allocated classrooms

  # iteratively do greedy choice
  for i in increasing starting time order do
    if lecture i is compatible with some classroom k then
      schedule lecture i in classroom 1 ≤ k ≤ d
    else
      allocate a new classroom d+1
      schedule lecture i in classroom d+1
      d ← d+1
  return d
```

Interval Partitioning: Greedy Algorithm

Greedy algorithm: Consider lectures in increasing order of start time: assign lecture to any compatible classroom.

```
def interval_partition(S):

  # initialization
  sort intervals in increasing starting time order
  d ← 0    # number of allocated classrooms

  # iteratively do greedy choice
  for i in increasing starting time order do
    if lecture i is compatible with some classroom k then
      schedule lecture i in classroom 1 ≤ k ≤ d
    else
      allocate a new classroom d+1
      schedule lecture i in classroom d+1
      d ← d+1
  return d
```

Implementation: O(n log n).

- For each classroom k, maintain the finish time of the last job added.
- Keep the classrooms in a priority queue.

Interval Partitioning: Greedy Analysis

Observation: Greedy algorithm never schedules two incompatible lectures in the same classroom.

Theorem: Greedy algorithm is optimal.

Proof:

- d = number of classrooms that the greedy algorithm allocates.
- Classroom d is opened because we needed to schedule a job, say i, that is incompatible with all d-1 other classrooms.
- Since we sorted by start time, all these incompatibilities are caused by lectures that start no later than s_i.
- Thus, we have d lectures overlapping at time $s_i + \varepsilon$. ← just after time s_i
- Key observation $\Rightarrow$ all schedules use $\geq$ d classrooms.

[Greedy algorithm stays "ahead"]

Text Compression

Given: a string X

Goal: efficiently encode X into a smaller string Y

(saves memory and/or bandwidth)

Input:

WWWWWWWWWWWWBWWWWWWWWWWWWBBBWWWWWW
WWWWWWWWWWWWWWWWWWBWWWWWWWWWWWWWW

Run length encoding (very simple approach):

12W1B12W3B24W1B14W

Text Compression

Given: a string X

Goal: efficiently encode X into a smaller string Y

(saves memory and/or bandwidth)

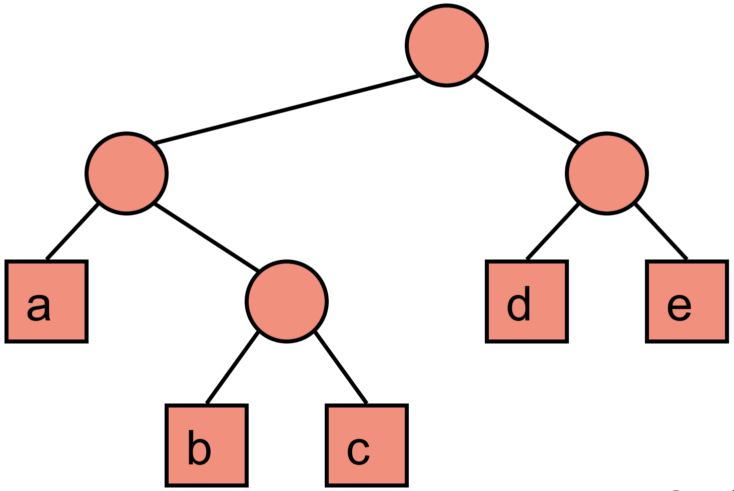
A better approach: Huffman encoding

- Let C be the set of characters in X
- Compute frequency f(c) for each character c in C
- Encode high-frequency characters with short code words
- No code word is a prefix of another code word
- Use an optimal encoding tree to determine the code words

Encoding Tree Example

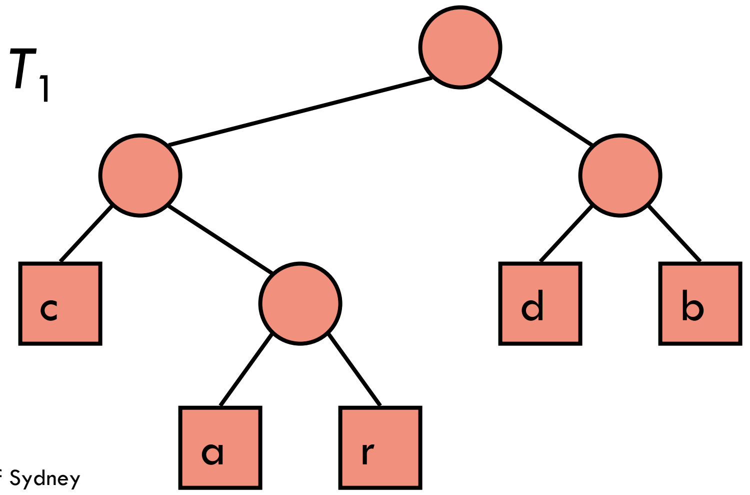
- A **binary code** is a mapping of each character of an alphabet to a binary code-word
- A **prefix code** is a code such that no code-word is the prefix of another code-word
- An **encoding tree** represents a prefix code
  - Each external node stores a character
  - The code-word of a character is given by the path from the root to the external node storing the character (0 for a left child and 1 for a right child)

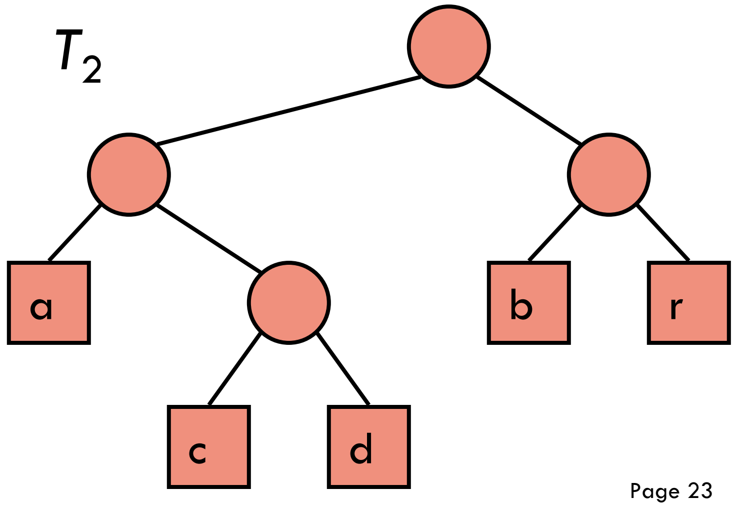
00	010	011	10	11
a	b	c	d	e



Encoding Tree Optimization

- Given a text string X, we want to find a prefix code for the characters of X that yields a small encoding for X
  - Frequent characters should have short code-words
  - Rare characters should have long code-words
- Example
  - X = abracadabra
  - T_1 encodes X into 29 bits
  - T_2 encodes X into 24 bits





Huffman's Algorithm

Given a string X, Huffman's algorithm constructs a prefix code that minimizes the size of the encoding of X

It runs in time O(n + d log d), where n is the size of X and d is the number of distinct characters of X

The algorithm builds the encoding tree from the bottom up, merging trees as it goes along, using a priority queue to guide the process

End result minimizes bits needed to encode X:

$$\sum_{c \text{ in } C} f(c) * depth_T\,(c)$$

Huffman's Algorithm

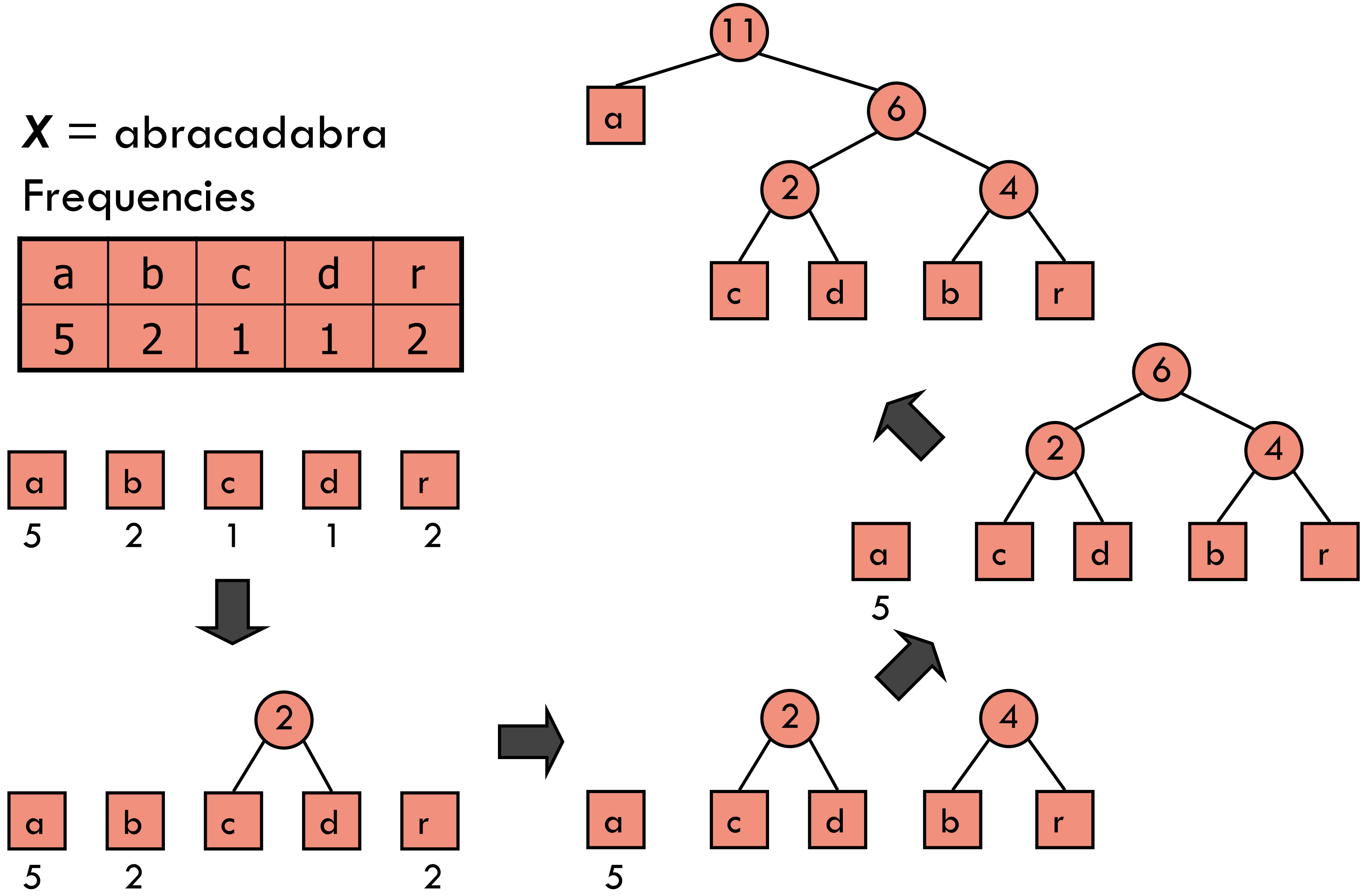
```
def huffman(C, f):

  # initialize priority queue
  Q ← empty priority queue
  for c in C do
    T ← single-node binary tree storing c
    Q.insert(f[c], T)

  # merge trees while at least two trees
  while Q.size() > 1 do
    $f_1$, $T_1$ ← Q.remove_min()
    $f_2$, $T_2$ ← Q.remove_min()
    T ← new binary tree with $T_1$/$T_2$ as left/right subtrees
    f ← $f_1 + f_2$
    Q.insert(f, T)

  # return last tree
  f, T ← Q.remove_min()
  return T
```

Example

X = abracadabra

Frequencies

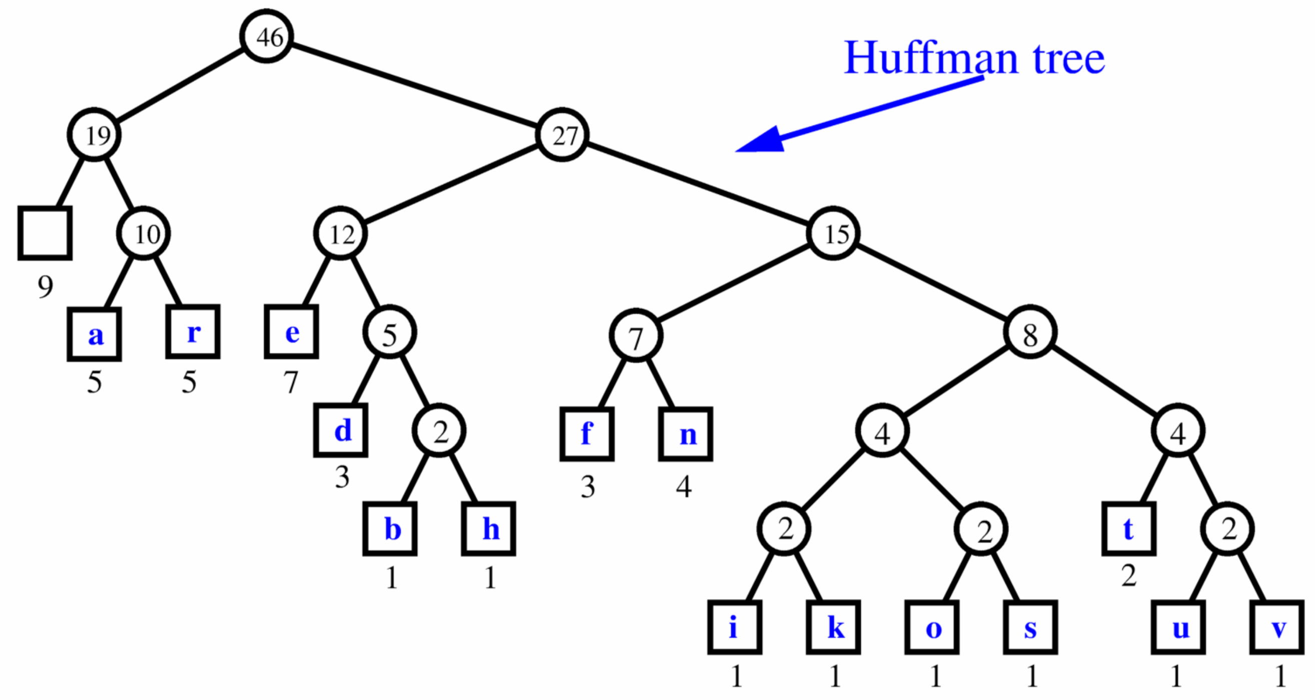
a	b	c	d	r
5	2	1	1	2



Extended Huffman Tree Example

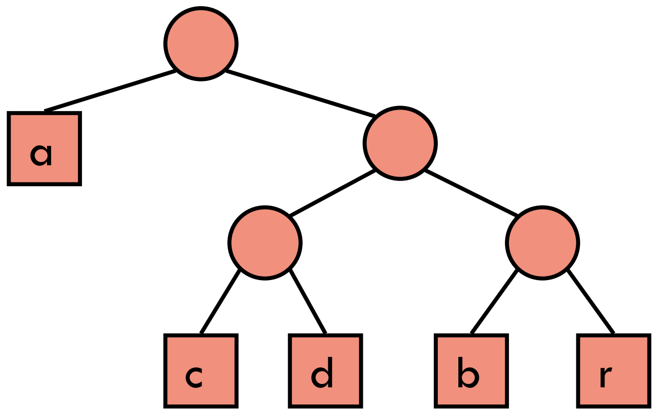
String: **a fast runner need never be afraid of the dark**

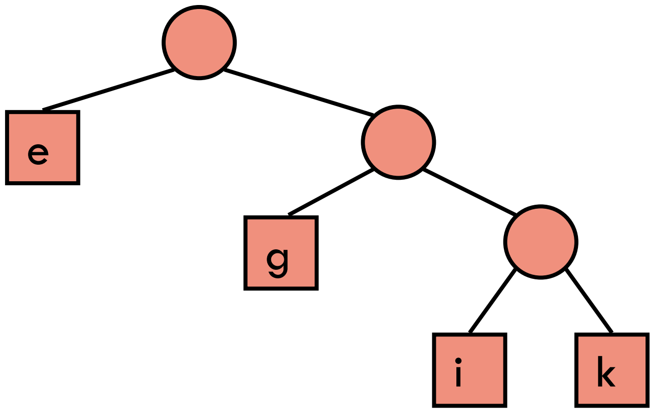
Character		a	b	d	e	f	h	i	k	n	o	r	s	t	u	v
Frequency	9	5	1	3	7	3	1	1	1	4	1	5	1	2	1	1



Huffman's Algorithm Correctness

Obs: Every encoding tree has a pair of leaves that are siblings.

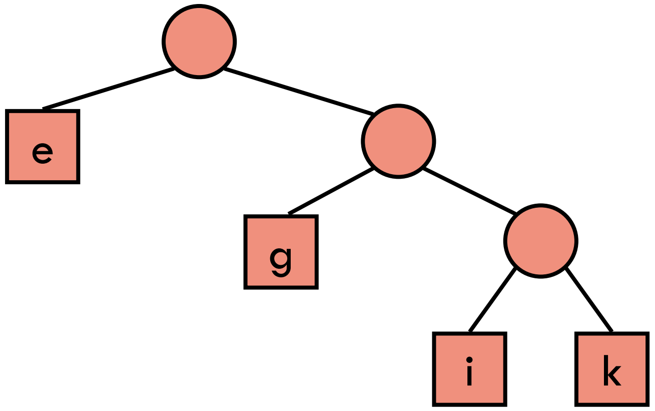




Huffman's Algorithm Correctness

Obs: In an optimal encoding tree T for any a and b in C, if $depth_T(a) < depth_T(b)$ then $f(a) \geq f(b)$.

$$\sum_{c \text{ in } C} f(c) * depth_T(c)$$

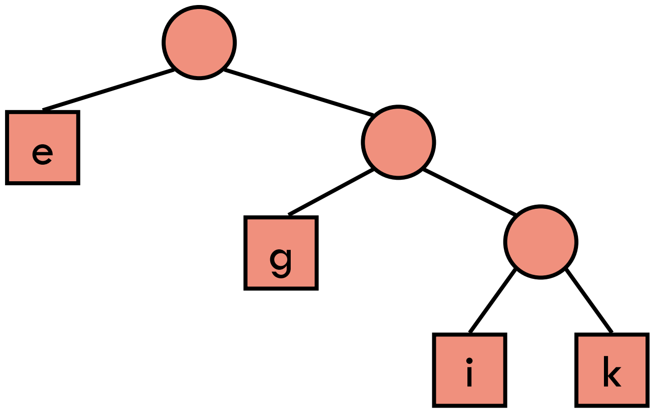


For example, if $f(e) < f(g)$ then swapping them leads to shorter encoding

Huffman's Algorithm Correctness

Obs: There is an optimal encoding tree T where the two sibling leaves furthest from the root have lowest frequency.

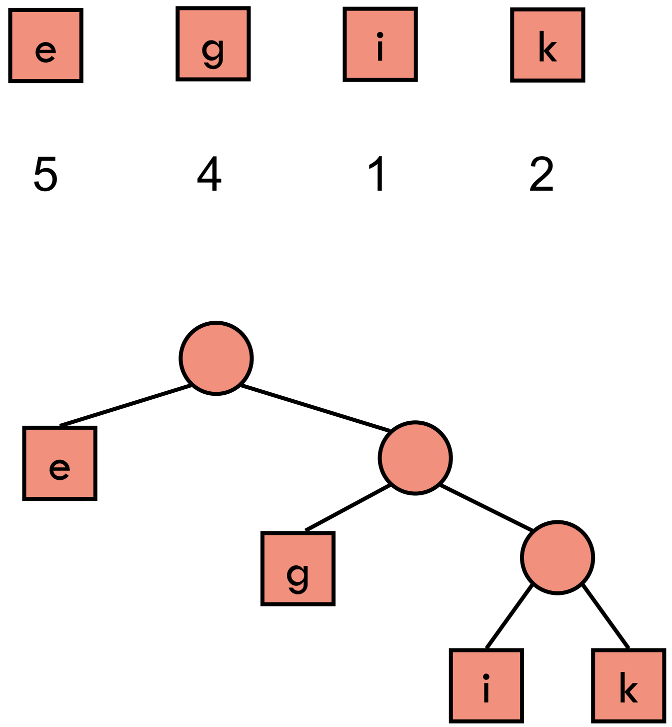
$$\sum_{c \, in \, C} f(c) * depth_T\,(c)$$

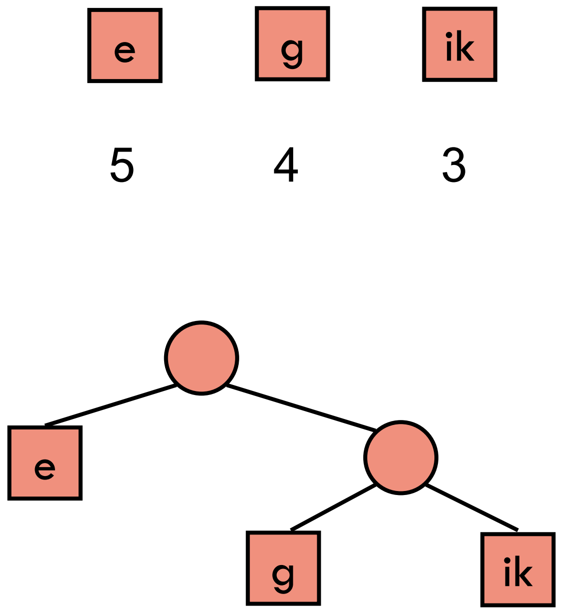


For example, characters i and k have lowest frequency

Huffman's Algorithm Correctness

Obs: If we combine the two lowest frequency characters to get a new instance (C', f'), an optimal encoding tree T' for (C', f') can be expanded to get an optimal encoding tree T for (C, f)

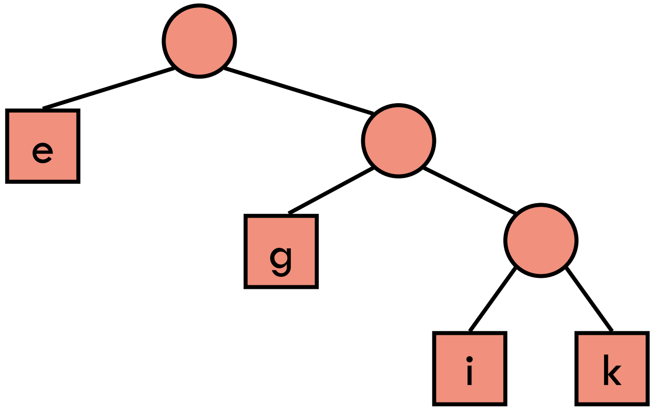


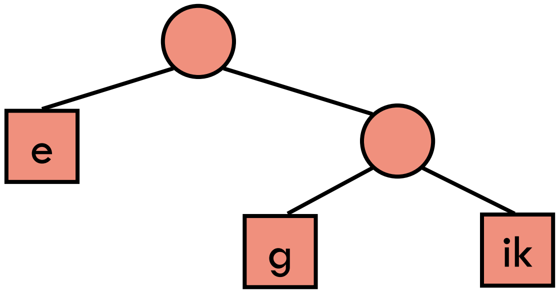


Huffman's Algorithm Correctness

Obs: If we combine the two lowest frequency characters to get a new instance (C', f'), an optimal encoding tree T' for (C', f') can be expanded to get an optimal encoding tree T for (C, f)

$$\sum_{c \text{ in } C} f(c) * depth_T(c) - \sum_{c \text{ in } C'} f'(c) * depth_{T'}(c)$$
$$= f(i) * depth_T(i) + f(k) * depth_T(k) - f'(ik) * depth_{T'}(ik)$$
$$= f(i) + f(k)$$





Huffman's Algorithm Correctness

Thm: Huffman's algorithm computes a minimum length encoding tree of (C, f)

Proof (by induction):

- If |C| = 1 then the encoding is trivially optimal
- If |C| > 1 then let (C', f') be the contracted instance
- By inductive hypothesis, the encoding tree T' constructed for (C', f') is optimal
- Recall that

$$\sum_{c\ in\ C} f(c) * depth_T(c) = \sum_{c\ in\ C'} f'(c) * depth_{T'}(c) + f(i) + f(k)$$

thus, the tree T is optimal for (C, f)

Huffman's Algorithm

```
def huffman(C, f):

  # initialize priority queue
  Q ← empty priority queue
  for c in C do
    T ← single-node binary tree storing c
    Q.insert(f[c], T)

  # merge trees while at least two trees
  while Q.size() > 1 do
    f_1, T_1 ← Q.remove_min()
    f_2, T_2 ← Q.remove_min()
    T ← new binary tree with T_1/T_2 as left/right subtrees
    f ← f_1 + f_2
    Q.insert(f, T)

  # return last tree
  f, T ← Q.remove_min()
  return T
```

Time complexity is dominated by PQ ops, which using heap take $O(|C| \log |C|)$ time

Greedy algorithms recap

Greedy heuristics are easy to design but they are not always optimal. And when they are, small modifications of the problem can render them suboptimal:

- 0-1 knapsack is hard
- if tasks/lectures have special needs the problem is hard
- if we use non-binary encodings, Huffman does not work