

COMMONWEALTH OF AUSTRALIA

Copyright Regulations 1969

WARNING

This material has been reproduced and communicated to you by or on behalf of the University of Sydney pursuant to Part VB of the Copyright Act 1968 (**the Act**). The material in this communication may be subject to copyright under the Act. Any further copying or communication of this material by you may be the subject of copyright protection under the Act.

Do not remove this notice.

COMP2823

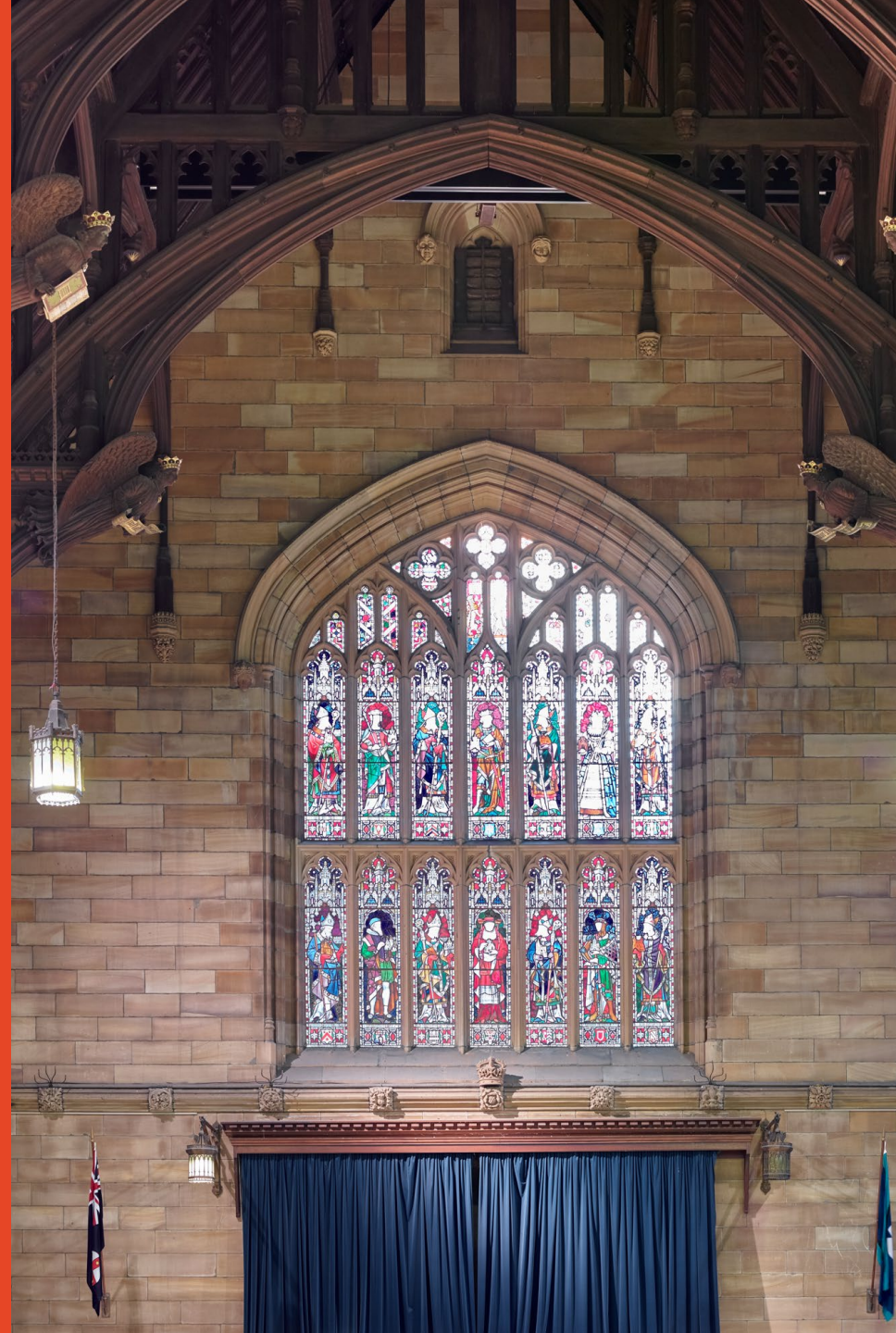
Lecture 9: The greedy method [GT 10]

Joachim Gudmundsson
School of Computer Science

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



THE UNIVERSITY OF
SYDNEY



General techniques

- Greedy algorithms [today]
- Divide & Conquer algorithms [week 10 & 11]
- Dynamic programming algorithms [COMP3027]
- Network flow algorithms [COMP3027]

Greedy algorithms

A greedy algorithm is an algorithm that follows the problem solving approach of making a locally optimal choice at each stage with the hope of finding a global optimum.

Greedy algorithms

Greedy algorithms can be some of the simplest algorithms to implement, but some are among the hardest algorithms to design and analyse.

In this unit we will only consider algorithms that are easy to analyse and leave the harder cases for COMP3027.

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
```

Today's lecture

Consider problems that can be solved using a greedy approach:

- Knapsack
- Text encoding
- Task scheduling
- ...

Greedy: Overview

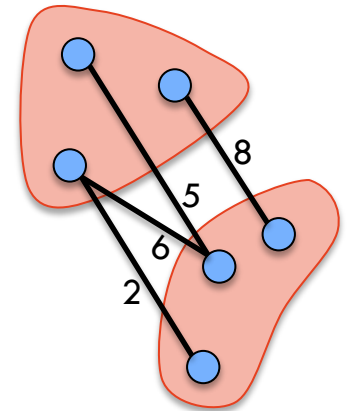
We have already studied three greedy algorithms:

- Prim's algorithm: Pick cheapest edge in the cutset.
- Kruskal's algorithm: Pick cheapest edge that doesn't induce a cycle.
- Dijkstra's algorithm: Pick vertex with shortest path to start vertex.

Greedy: Overview

We have already studied three greedy algorithms:

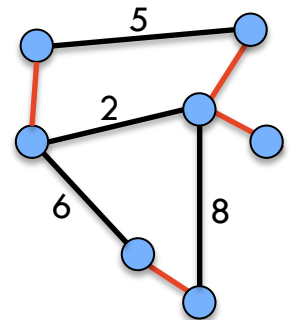
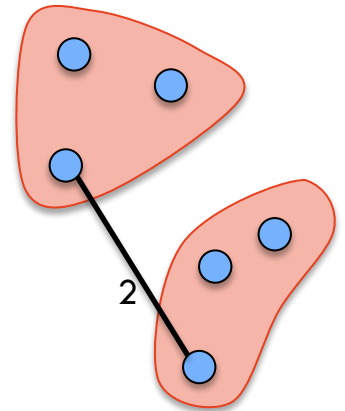
- Prim's algorithm: Pick cheapest edge in the cutset.
- Kruskal's algorithm: Pick cheapest edge that doesn't induce a cycle.
- Dijkstra's algorithm: Pick vertex with shortest path to start vertex.



Greedy: Overview

We have already studied three greedy algorithms:

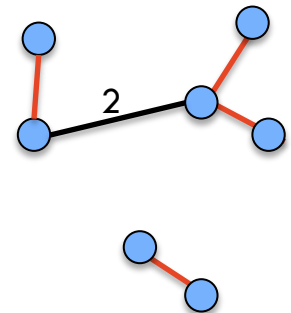
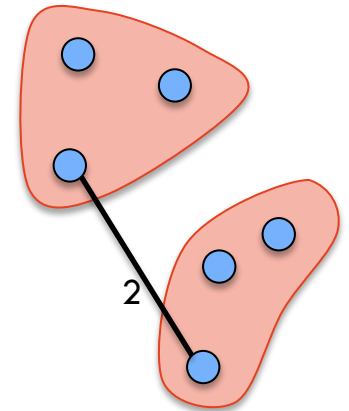
- Prim's algorithm: Pick cheapest edge in the cutset.
- Kruskal's algorithm: Pick cheapest edge that doesn't induce a cycle.
- Dijkstra's algorithm: Pick vertex with shortest path to start vertex.



Greedy: Overview

We have already studied three greedy algorithms:

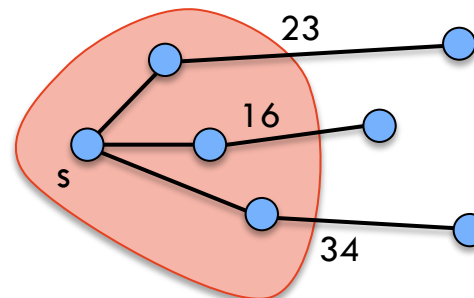
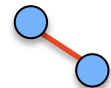
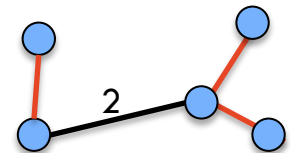
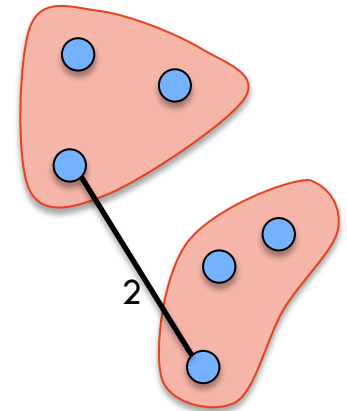
- Prim's algorithm: Pick cheapest edge in the cutset.
- Kruskal's algorithm: Pick cheapest edge that doesn't induce a cycle.
- Dijkstra's algorithm: Pick vertex with shortest path to start vertex.



Greedy: Overview

We have already studied three greedy algorithms:

- Prim's algorithm: Pick cheapest edge in the cutset.
- Kruskal's algorithm: Pick cheapest edge that doesn't induce a cycle.
- Dijkstra's algorithm: Pick vertex with shortest path to start vertex.



The Knapsack Problem



Lots of variants.

Standard model is the 0-1 Knapsack Problem.

Given: A knapsack that can carry W kg
A set S of n items, with each item i having
 w_i : a positive weight
 b_i : a positive benefit (value)

Goal: Choose items with maximum total benefit of weight at most W .
[Can either pick the item (1), or not (0)]



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 .
[Can pick a fraction of each item]

Let x_i denote the amount we take of item i



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 .
[Can pick a fraction of each item]

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”

Optimal:

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

10 ml

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 could mean:

[highest benefit (b_i)]: Select items with highest benefit.

[smallest weight (w_i)]: Select items with smallest weight.

[benefit/weight (b_i/w_i)]: Select items with highest benefit/weight ratio.

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

What's “best”?



Greedy choice: Keep taking the “best” item.

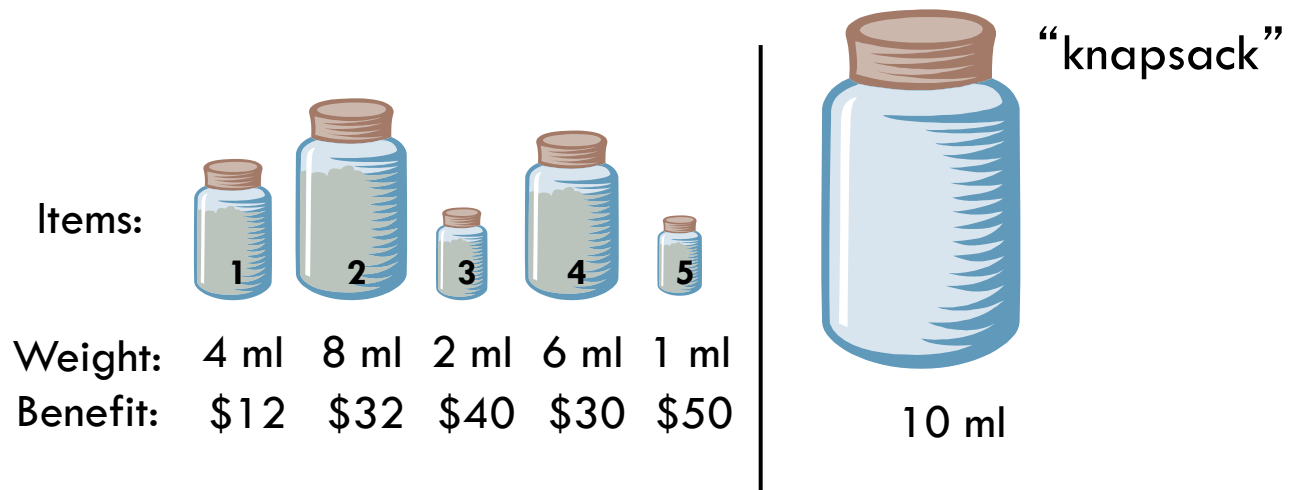
[highest benefit]: Select items with highest benefit.

1 ml of 5 → \$50

2 ml of 3 → \$40

7 ml of 2 → \$28

Total value: \$118



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:					
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

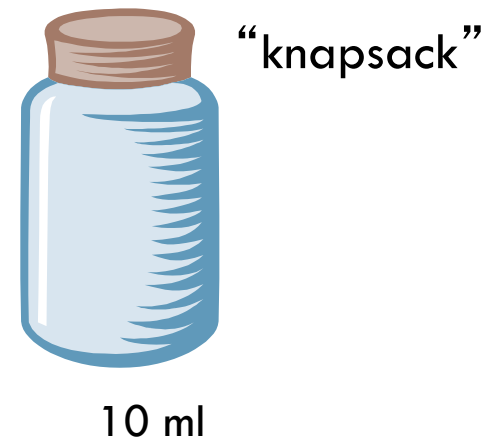
2 ml of 3 → \$40

6 ml of 4 → \$30

1 ml of 2 → \$4

Total value: \$124

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



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 Y different than the greedy one
- There must be items i and k s.t. $y_i < w_i$ and $y_k > 0$, and $b_i/w_i > b_k/w_k$

Why?

Assume w.l.o.g. $(b_1/w_1), \dots, (b_n/w_n)$ sorted in decreasing order:

$$G: w_1 (b_1/w_1) + \dots + w_{m-1} (b_{m-1}/w_{m-1}) + x_m (b_m/w_m) + 0 (b_{m+1}/w_{m+1}) + \dots + 0 (b_n/w_n)$$

$$Y: y_1 (b_1/w_1) + y_2 (b_2/w_2) + \dots + \underbrace{y_i (b_i/w_i)}_{< w_i} + \dots + \underbrace{y_k (b_k/w_k)}_{> 0} + \dots + y_n (b_n/w_n)$$

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 Y different than the greedy one
- There must be items i and k s.t. $y_i < w_i$ and $y_k > 0$, 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? $\min\{w_i - y_i, y_k\}$
- The new solution is closer to a greedy solution
- 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
```

Theorem: There exists a greedy algorithm for the Fractional Knapsack Problem that runs in $O(n \log n)$ time.

The 0-1 Knapsack Problem



Given: A knapsack that can carry W kg
A set S of n items, with each item i having
 w_i : a positive weight
 b_i : a positive benefit (value)

Goal: Choose items with maximum total benefit of weight at most W .
[Can either pick the item (1), or not (0)]



The 0-1 Knapsack Algorithm

Initial configuration: no items chosen

Each step: identify the “best” item available that fits in the knapsack

```
def integer_knapsack(b, w, W):  
  
    # initialization  
    x ← array of size |b| of zeros  
    curr ← 0  
  
    # iteratively make greedy choice  
    while exists i : w[i] ≤ W - curr do  
        i ← item maximizing b[i]/w[i]  
        such that w[i] ≤ W - curr    x[i] ← w[i]  
        curr ← curr + x[i]  
    return x
```

The 0-1 Knapsack Algorithm Analysis

Greedy choice: Keep taking the “best” item.

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

Item 1 → \$30

Total value: \$30

Items:



1



2



3

Weight:

2.6 kg

2.5 kg

2.5 kg

Benefit:

\$30

\$25

\$20

B_i/W_i

11.5

10

8



5 kg

The 0-1 Knapsack Algorithm Analysis

Optimal choice

Item 2 → \$25

Item 3 → \$20

Total value: \$45

Items:



1



2



3

Weight:

2.6 kg

2.5 kg

2.5 kg

Benefit:

\$30

\$25

\$20

B_i/W_i

11.5

10

8



5 kg

The 0-1 Knapsack Problem



Known: The 0-1 Knapsack problem belongs to a family of problems known as NP-hard problems [COMP3027].

We don't know how to solve any problem that is NP-hard in polynomial time.

Can be solved in $O(nW)$ time if the weights are positive integers.

Can be approximated in polynomial time (the greedy always finds a solution that is within a factor of 2 from the optimal).

Greedy algorithms: Proving correctness

The most common way of proving the correctness of a greedy algorithm is by using an exchange argument.

They work by showing that you can iteratively transform any optimal solution into the solution produced by the greedy algorithm without changing the cost of the optimal solution, thereby proving that the greedy solution is optimal. Typically, exchange arguments are set up as follows:

1. Define your solutions. You will be comparing your greedy solution X to an optimal solution OPT , so it's best to define these variables explicitly.
2. Compare solutions. Next, show that if X is not equal to OPT , then they must differ in some specific way. This could mean that there's a piece of X that's not in OPT , or that two elements of X that are in a different order in OPT , etc. You might want to give those pieces names.

Greedy algorithms: Proving correctness (cont'd)

3. Exchange Pieces. Show how to transform OPT by exchanging some piece of OPT for some piece of X. You'll typically use the piece you described in the previous step. Then, prove that by doing so, you did not increase the cost of OPT and you therefore have a different optimal solution.
4. Iterate. Argue that you have decreased the number of differences between X and OPT by performing the exchange, and that by iterating this process you can turn OPT into X without impacting the quality of the solution. Therefore, X must be optimal. This last step might require a formal argument using an induction proof. However, in most cases this is not needed.

Typical problem

Mini team triathlon (n members)

- Swim 20 laps
- Cycle 10km
- Run 3km

Only one person in the pool at the time (must complete swimming).




Each contestant i has projected swimming (s_i), cycling (c_i) and running time (r_i).

Decide an order of the team members that minimises the completion time, assuming the projected times.

Typical problem

Mini team triathlon (n members)

- Swim 20 laps
- Cycle 10km
- Run 3km

			
Huey	20	5	4
Dewey	10	10	11
Louie	1	15	15
	s_i	c_i	r_i

Only one person in the pool at the time (must complete swimming).

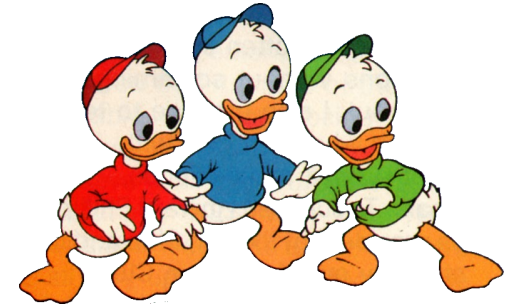
Each contestant i has projected swimming (s_i), cycling (c_i) and running time (r_i).

Decide an order of the team members that minimises the completion time, assuming the projected times.

Use a greedy approach. How do we decide the order?

Typical problem

[Individual completion time]: Fastest first



Example:	Swim	Cycle	Run
Huey	20	5	4
Dewey	10	10	10
Louie	1	15	15

Order: Huey, Dewey and Louie

Total completion time: $20 + 10 + 31 = 61$

Typical problem

Optimal solution

Example:	Swim	Cycle	Run
Huey	20	5	4
Dewey	10	10	10
Louie	1	15	15

Order: Louie, Dewey and Huey

Total completion time: $1 + 10 + 29 = 40$

Typical problem

[cycle+run]: Slowest first \Leftrightarrow Fastest swimmer first

Example:	Swim	Cycle	Run
Huey	20	5	4
Dewey	10	10	10
Louie	1	15	15

Order: Louie, Dewey and Huey

Total completion time: $1 + 10 + 29 = 40$

Proof of correctness (sketch)

X = order produced by the greedy algorithm

OPT = optimal order

Greedy scheduled in order of slowest $c_i + r_i$:

i 's completion time: $s_1 + \dots + s_{i-1} + (s_i + c_i + r_i)$

If $X = OPT$ then we're done.

Otherwise OPT must contain two members k followed by i such that $c_i + r_i > c_k + r_k$. We call such a pair an **inversion**. Pick the first inversion. Look at completion time for i and k in OPT .

OPT

k : $s_1 + \dots + s_{i-1} + (s_k + c_k + r_k)$

i : $s_1 + \dots + s_{i-1} + s_k + \dots + s_p + (s_i + c_i + r_i)$

Consider a solution where i and k swap order in OPT .

i : $s_1 + \dots + s_{i-1} + (s_i + c_i + r_i)$

[i will obviously complete earlier than in OPT]

k : $s_1 + \dots + s_{i-1} + s_i + \dots + s_p + (s_k + c_k + r_k)$

[Note $s_i < s_k$ and $c_i + r_i > c_k + r_k$, hence, k will finish earlier than i did in OPT]

\Rightarrow The swapped order does not have a greater completion time, and must also be optimal.

Continuing iteratively, eliminate all inversions without increasing completion time.

The greedy order is no worse than the optimal and must also be optimal.

Exchange argument

1. Define your solutions.
2. Compare solutions. If X is not equal to OPT , then they must differ in some specific way.
3. Exchange Pieces. Prove that this did not increase the cost of OPT .
4. Iterate. Argue that you have decreased the number of differences between X and OPT by performing the exchange, and that by iterating this process you can turn OPT into X without impacting the quality of the solution.

Mini team triathlon

Theorem:

The greedy algorithm (fastest swimmer first) solves the Mini Team Triathlon problem in $O(n \log n)$ time

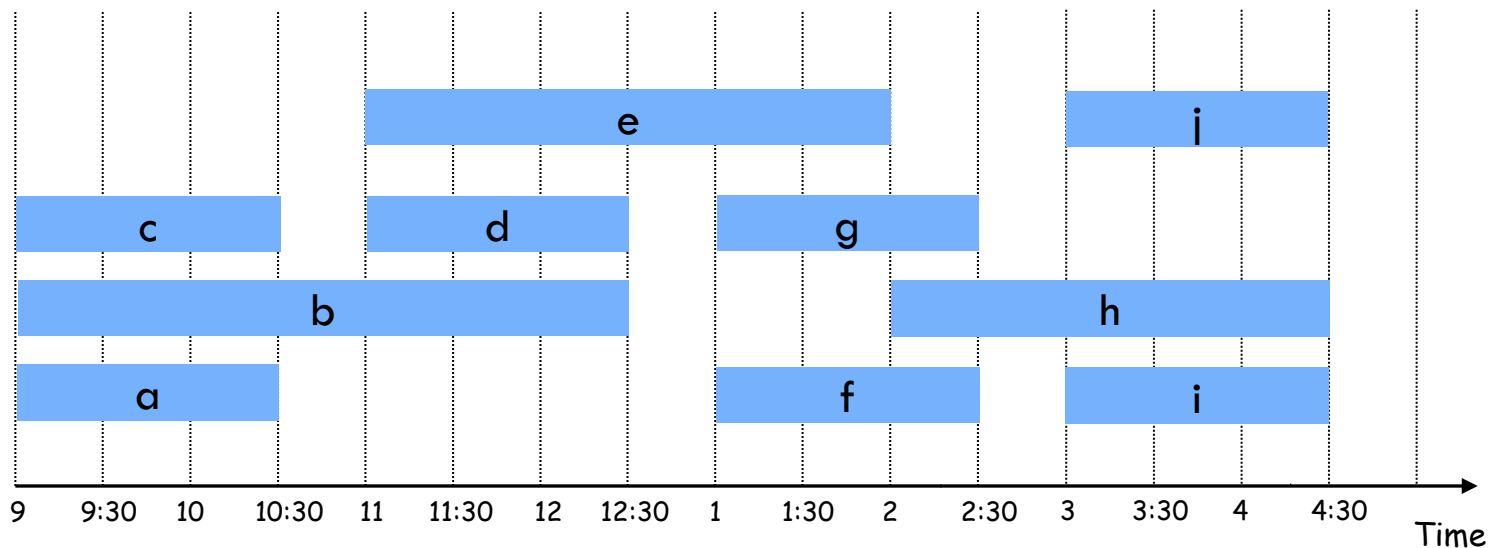
Task scheduling

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.



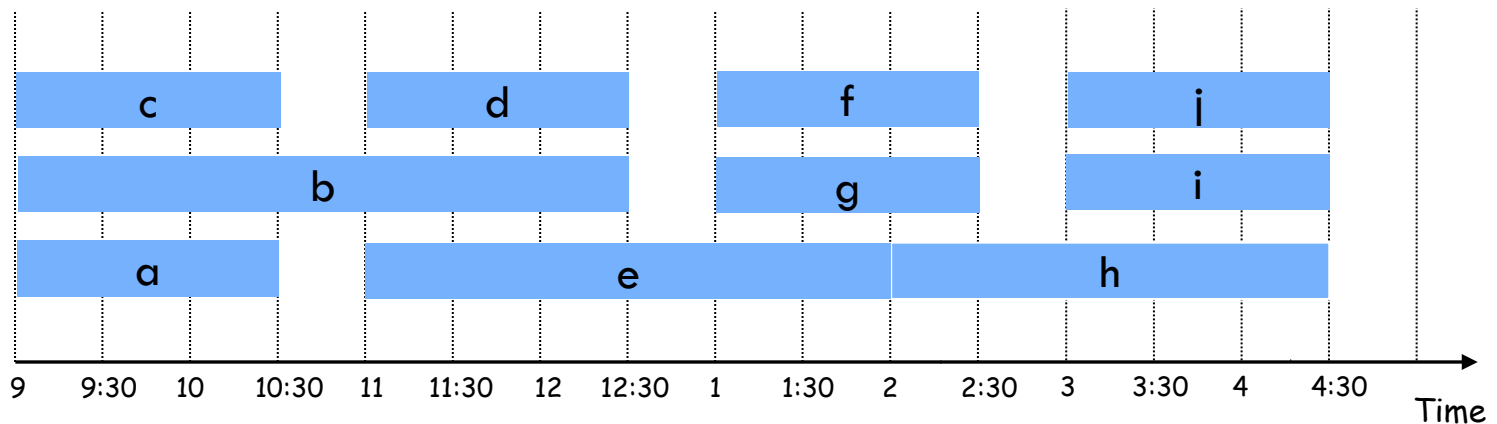
Task scheduling

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.



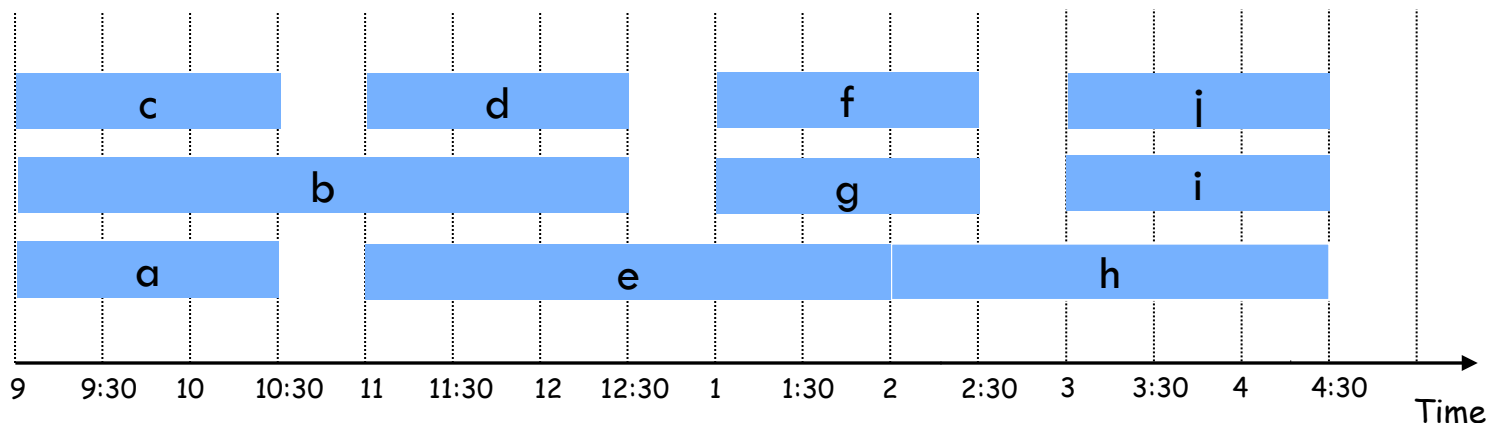
Interval Partitioning: Lower bound

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  $1 \leq k \leq d$   
        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  $1 \leq k \leq d$   
        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.

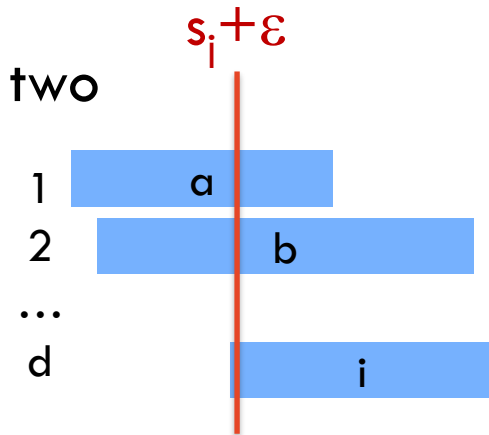
Interval Partitioning: Greedy Analysis

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

Theorem: Greedy algorithm is optimal.

Proof: (sketch)

- d = number of classrooms that the greedy algorithm allocates.
- Classroom d is opened because we needed to schedule a class, 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 $s_i + \epsilon$.
- Key observation \Rightarrow all schedules use $\geq d$ classrooms.



Text Compression

Given: a string X [each character usually needs 8 bits]

Goal: efficiently encode X into a smaller string Y
(saves memory and/or bandwidth)

Input:

WWWWWWWWWWWWWWWWWWBWWWWWWWWWWWWWWWWBBBWWWWWWWW
WWWWWWWWWWWWWWWWWWWWWWWWWWBWWWWWWWWWWWWWWWWWWWW

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**

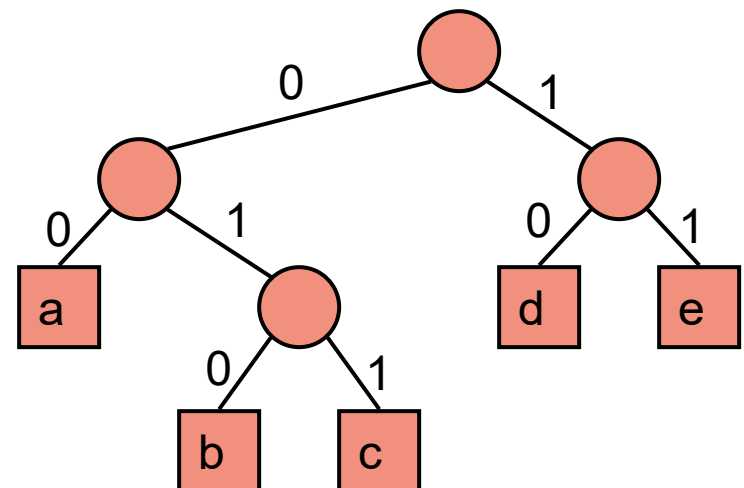
- Let C be the set of characters in X
- Compute frequency $f(c)$ for each character c in C
- Encode high-frequency characters with short code words
- No code word is a prefix for another code
- Use an optimal encoding tree to determine the code words

Encoding Tree Example

[David A. Huffman 1952]

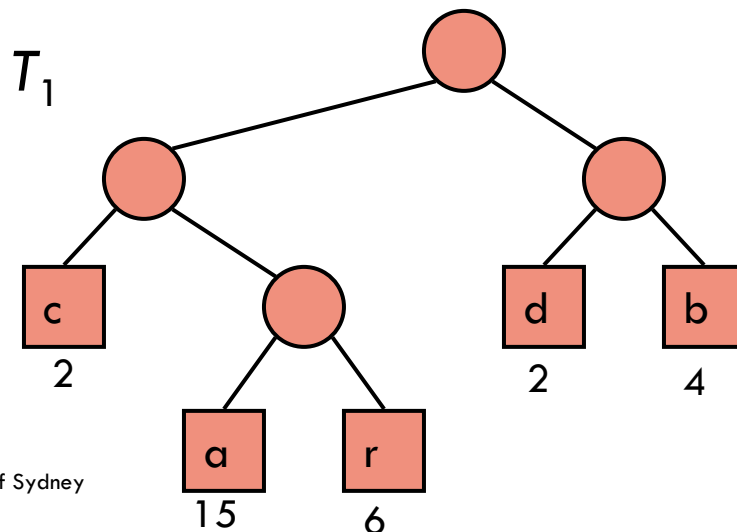
- A **code** is a mapping of each character of an alphabet to a binary code-word
- A **prefix code** is a binary code such that no code-word is the prefix of another code-word
- An **encoding tree** represents a prefix code
 - Each leaf stores a character
 - The code-word of a character is given by the 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



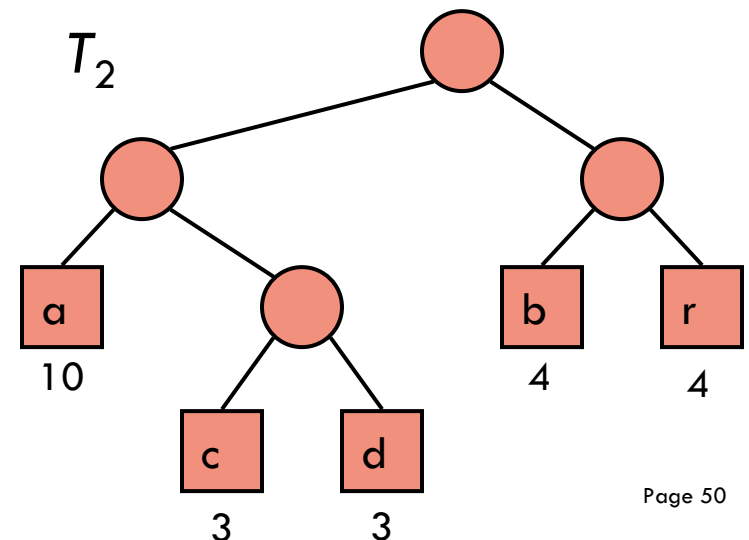
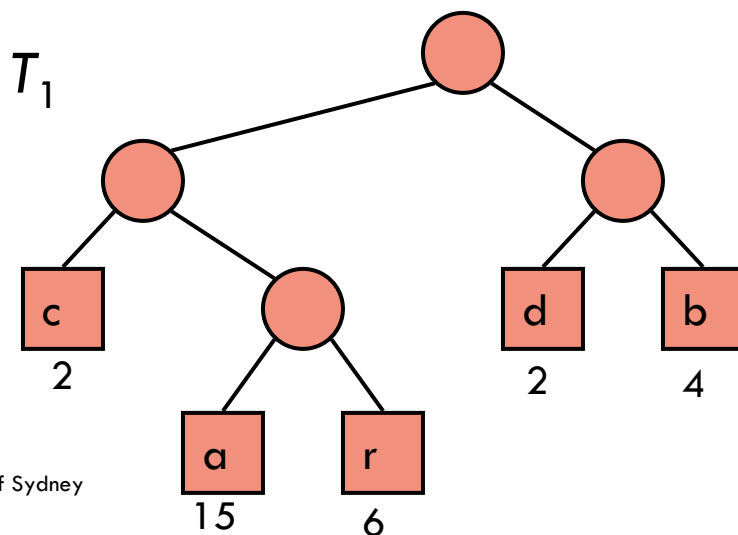
Encoding Tree Optimization

- 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
 - $X = \text{abracadabra}$ (a-5, b-2, r-2, c-1, d-1)
 - T_1 encodes X into 29 bits [01011011...]



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
 - $X = \text{abracadabra}$ (a-5, b-2, r-2, c-1, d-1)
 - T_1 encodes X into 29 bits [01011011...]
 - T_2 encodes X into 24 bits [001011...]



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 :

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

Huffman's Algorithm

def huffman(C, f):

initialize priority queue

Q \leftarrow empty priority queue

for c in C do

 T \leftarrow single-node binary tree storing c

 Q.insert(f[c], T)

merge trees while at least two trees

while Q.size() > 1 do

$f_1, T_1 \leftarrow$ Q.remove_min()

$f_2, T_2 \leftarrow$ Q.remove_min()

 T \leftarrow new binary tree with T_1/T_2 as left/right subtrees

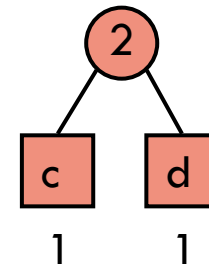
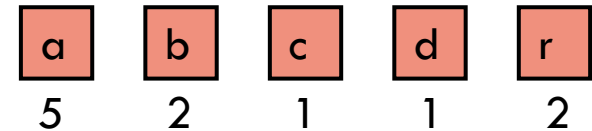
$f \leftarrow f_1 + f_2$

 Q.insert(f, T)

return last tree

f, T \leftarrow Q.remove_min()

return T

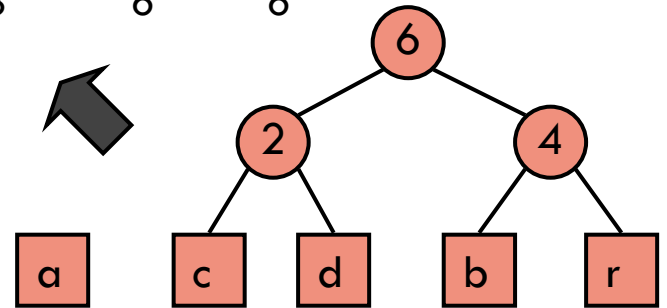
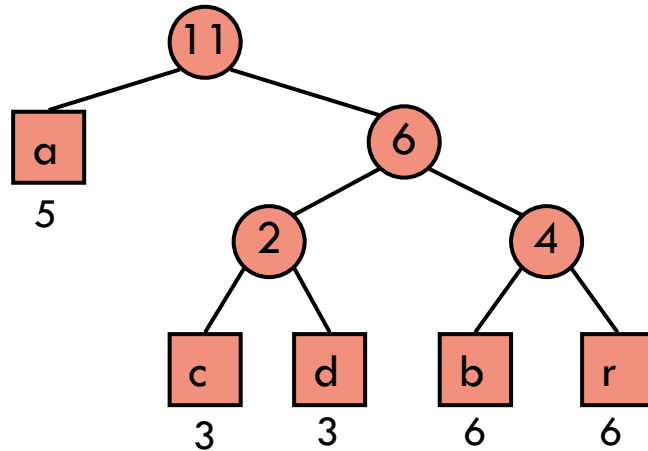
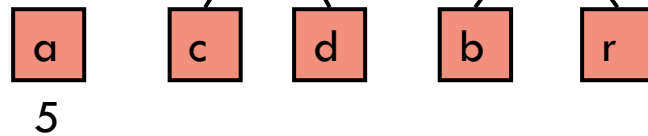
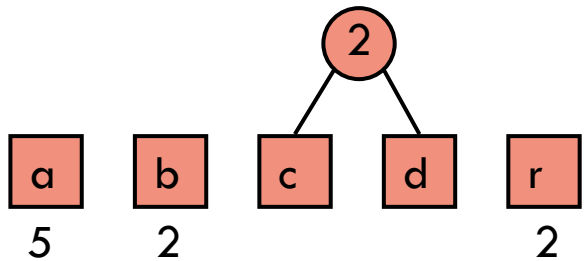
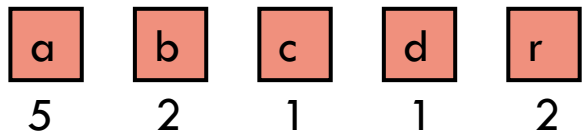


Example

$X = \text{abracadabra}$

Frequencies

a	b	c	d	r
5	2	1	1	2

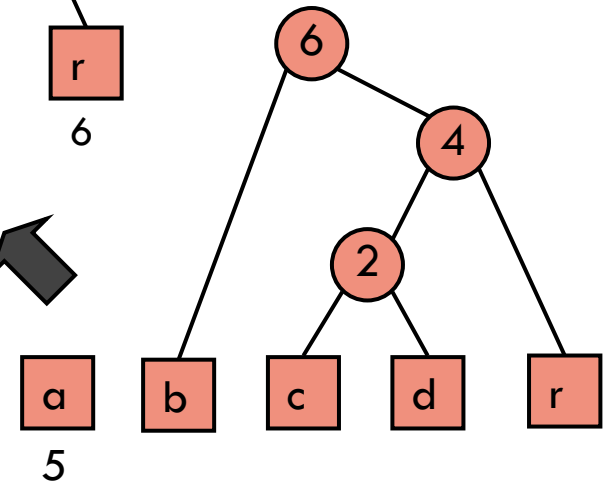
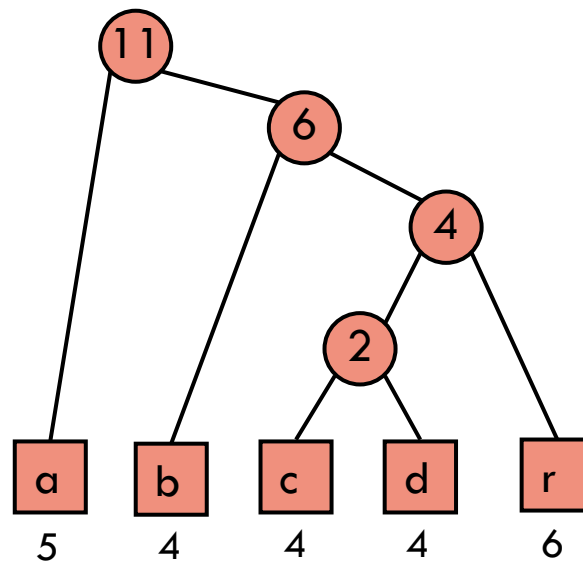
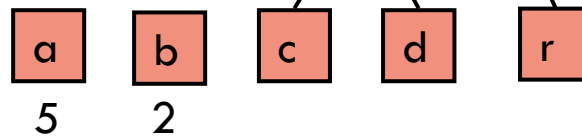
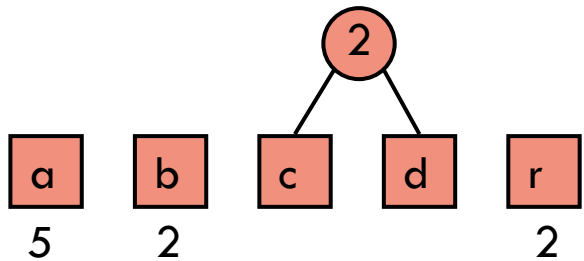
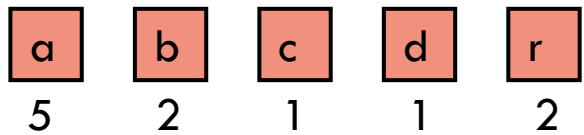


Example

$X = \text{abracadabra}$

Frequencies

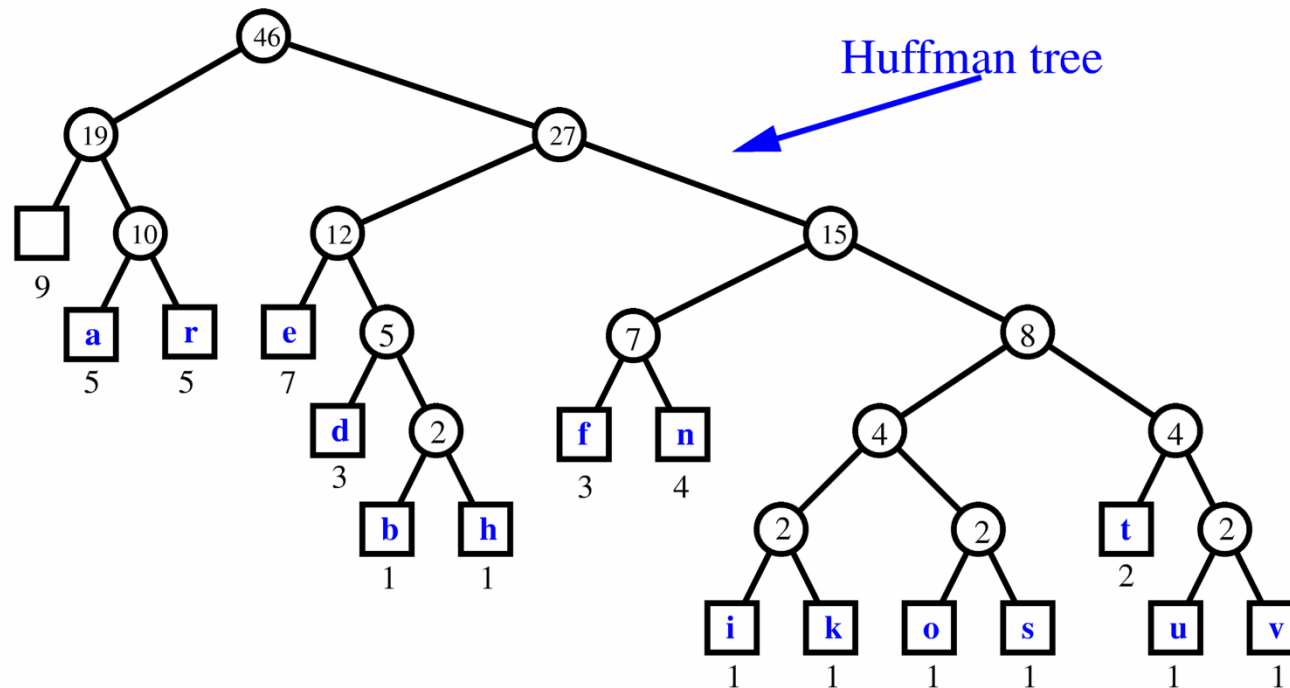
a	b	c	d	r
5	2	1	1	2



Extended Huffman Tree Example

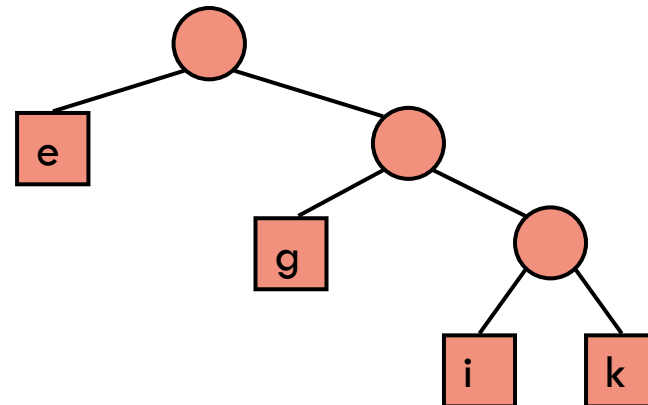
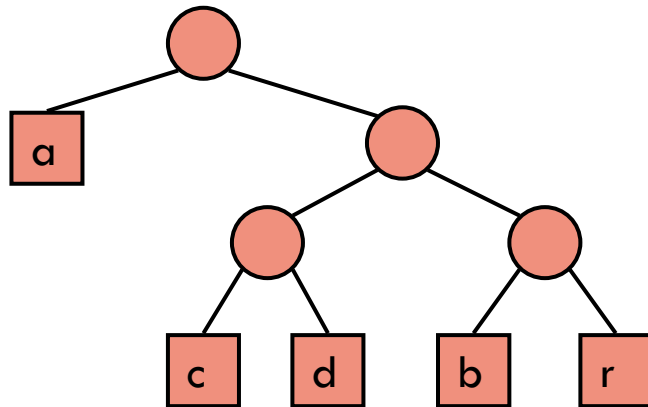
String: **a fast runner need never be afraid of the dark**

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

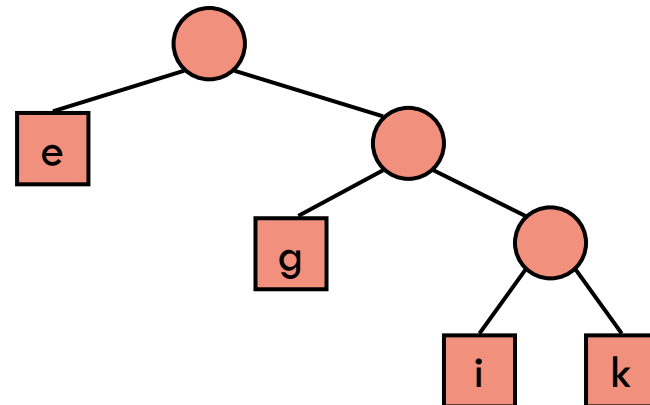
Observation: Every encoding tree has a pair of leaves that are siblings.



Huffman's Algorithm Correctness

Observation: In an optimal encoding tree T for any a and b in C , if $\text{depth}_T(a) < \text{depth}_T(b)$ then $f(a) \geq f(b)$.

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

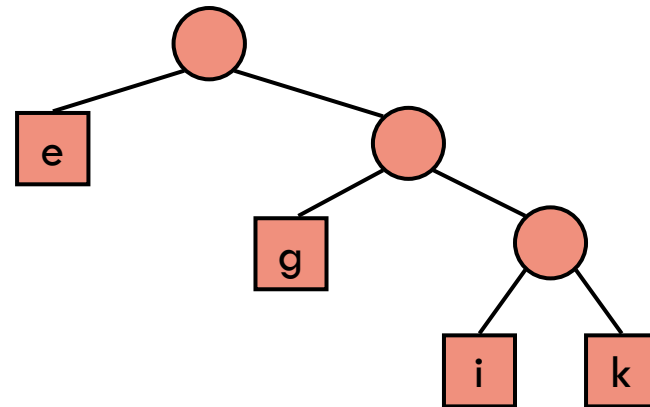


For example, if $f(e) < f(g)$ then swapping them leads to shorter encoding

Huffman's Algorithm Correctness

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

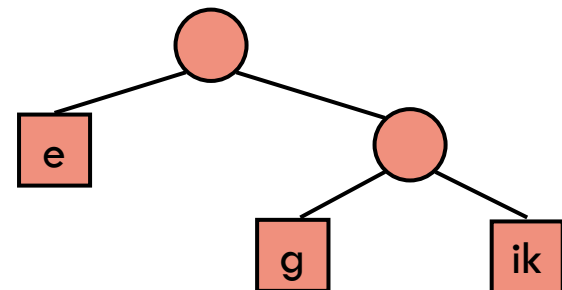
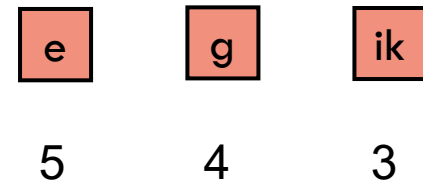
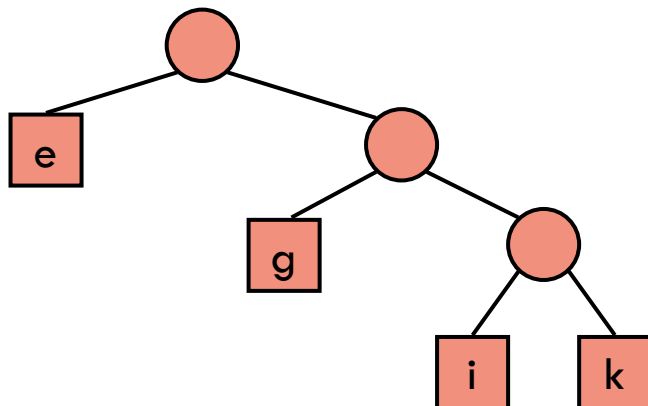
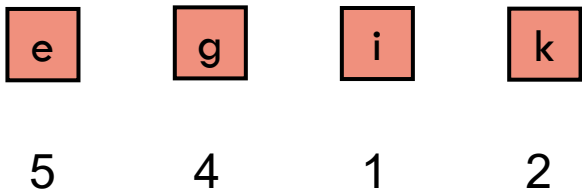
$$\sum_{c \text{ in } C} f(c) * \text{depth}T(c)$$



For example, characters **i** and **k** have lowest frequency

Huffman's Algorithm Correctness

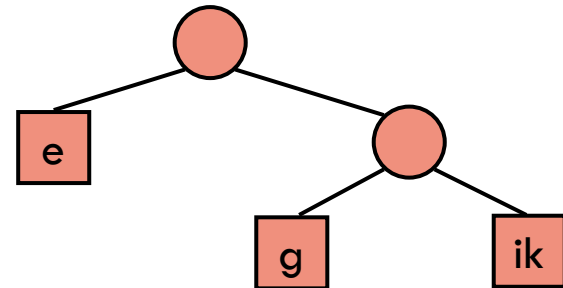
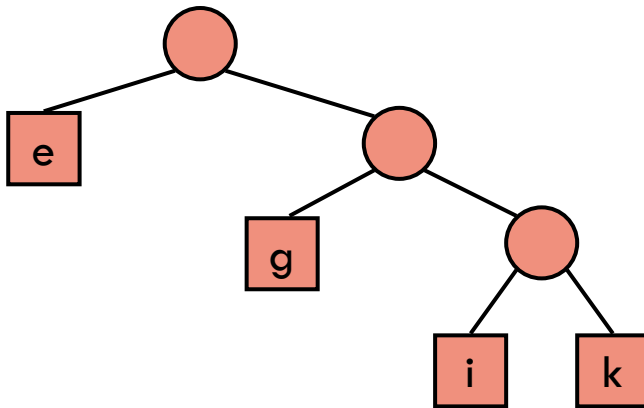
Observation: If we combine the two lowest frequency characters to get a new instance (C', f') , an optimal encoding tree T' for (C', f') can be expanded to get an optimal encoding tree T for (C, f)



Huffman's Algorithm Correctness

Observation: 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)

$$\begin{aligned} \sum_{c \in C} f(c) * \text{depth}_T(c) &- \sum_{c \in C'} f'(c) * \text{depth}_{T'}(c) \\ &= f(i) * \text{depth}_T(i) + f(k) * \text{depth}_T(k) - f'(ik) * \text{depth}_{T'}(ik) \\ &= f(i) + f(k) \end{aligned}$$



Huffman's Algorithm Correctness

Theorem: 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

$$\sum_{c \in C} f(c) * \text{depth}_T(c) = \sum_{c \in C'} f'(c) * \text{depth}_{T'}(c) + f(i) + f(k)$$

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
```

```
         $f_1, T_1 \leftarrow Q.remove\_min()$ 
```

```
         $f_2, T_2 \leftarrow Q.remove\_min()$ 
```

```
        T ← new binary tree with  $T_1/T_2$  as left/right subtrees
```

```
         $f \leftarrow f_1 + f_2$ 
```

```
        Q.insert(f, T)
```

```
    # return last tree
```

```
     $f, T \leftarrow Q.remove\_min()$ 
```

```
    return T
```

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

Next week

- Divide & Conquer
 - Binary search
 - Merge sort
 - Quick sort
- Sorting lower bound