

General Course Goals

Key Goal

Study algorithms and their analysis in order to **find practical solutions to real-world problems**

Our goals are:

- **Theory:**
 - Learn a variety of important algorithms
 - Understand how to determine algorithm complexity, that is, the estimate running time
- **Practice:**
 - Learn to *implement* various algorithms and data structures to solve a variety of problems
 - Understand how to measure the *actual* running time of an algorithm *empirically*

Why study Algorithms, Data Structures and their evaluation?

In order to **solve problems**, including building computer programs to do so, we need to have (algorithmic) **tools** and **evaluate** how efficient the tools are.

Why do we need an algorithmic toolset and learn how to evaluate their efficiency?

Answer:

- Need to know how the algorithms work and how to measure their efficiency in order to **select** the right one for the right problem/job.
- **Designing** new algorithms or **modify** existing ones to solve your problems.
- **It makes you a better problem solver and programmer!**



Prequisites

- This course is **not** about programming, but we do need a language for the **practical** parts:
 - Python: At a minimum, you should be comfortable with *basic data types and structures, defining new classes and data types.*
 - Warning: The **assignments** and **labs** are all in Python
- Prequisites:
 - *Programming Fundamentals* or *Further Programming*, or *Programming Bootcamp 2 Advanced Programming Techniques*, which teach Python and concepts helpful for this course.

Maths :)

This course has some **maths** in it.

We need to count how many times certain operations occur to analyse algorithms.

- Equations (summations, recursions). $\sum_{i=0}^n 2$.
- Arithmetic and partial sum simplifications. $\sum_{i=0}^n 2 = 2n + 2$.

I understand some of you haven't seen maths for years (or run for nearest train station when it appears), but help is available.



What is an algorithm?

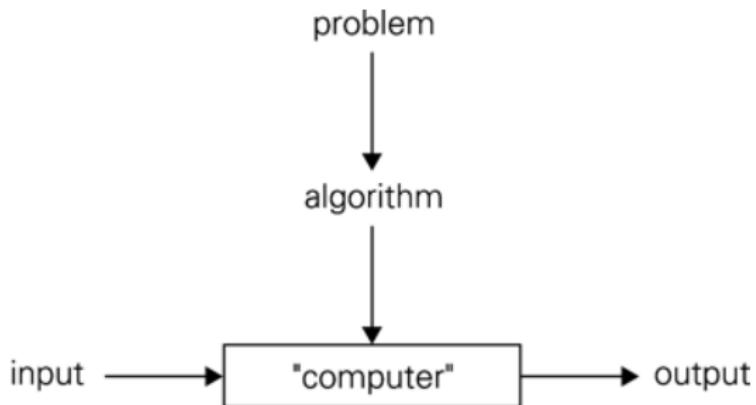
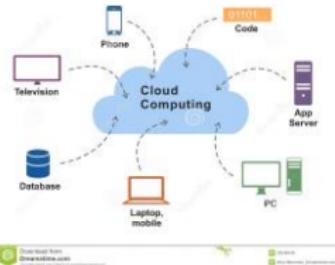


FIGURE 1.1 Notion of algorithm

An **algorithm** is a sequence of (unambiguous) instructions/steps for solving a problem.

Examples of Algorithms ... ?



Example: Euclid's Algorithm

Problem : Find $\text{GCD}(m, n)$, the greatest common divisor of two non-negative integers m and n .

Examples: $\text{GCD}(60, 24) = 12$; $\text{GCD}(60, 0) = 60$; $\text{GCD}(0, 0) = ?$

Solution (*sketch*)

Euclid's algorithm is based on repeated application of the equality $\text{GCD}(m, n) = \text{GCD}(n, m \bmod n)$ until the second number (n) becomes 0. The answer is then the first number (m).

Example : $\text{GCD}(60, 24) = \text{GCD}(24, 12) = \text{GCD}(12, 0) = 12$

Euclid's Algorithm Pseudocode

ALGORITHM **Euclid** (m, n)

//Computes $\text{GCD}(m, n)$ by Euclid's algorithm
//INPUT : Two non-negative integers m and n
//OUTPUT : Greatest Common Divisor of m and n

- 1: **if** $n > 0$ **then**
- 2: **return** $\text{GCD}(n, m \bmod n)$
- 3: **end if**
- 4: **return** m

Euclid's Algorithm

Alternative approaches to this problem?

Is it more “efficient”?

GCD Alternative (Sketch)

Alternative - Common primes:

Step 1 Find the prime factors of m .

Step 2 Find the prime factors of n .

Step 3 Identify all the common factors in the two prime expansions from Step 1 and Step 2.

Step 4 Compute the product of all common factors and return it as the greatest common divisor.

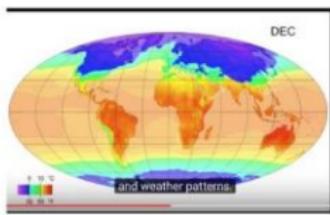
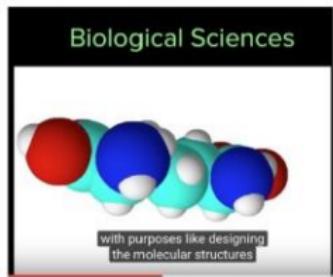
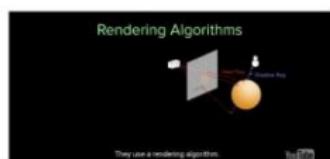
Example:

$$60 = 2 \cdot 2 \cdot 3 \cdot 5$$

$$24 = 2 \cdot 2 \cdot 2 \cdot 3$$

$$\text{GCD}(60, 24) = 2 \cdot 2 \cdot 3 = 12$$

Motivation for studying algorithms



▶ Video

Abstract Data Types and Data structures

Algorithms process input data, produce and process intermediate data, then output some results, which could be stored as data.

But we do not want to consider algorithms for **each type of input data**.

Instead we seek to talk about the input data, the intermediate data, and the output data in terms of **general characteristics and operations** the data should possess.

These data abstractions are called **Abstract data types (ADT)**.

- An ADT can be defined by the collection of common operations that are accessed through an *interface* and its characteristics.

Abstract Data Type

Data structure is the implementation of an abstract data type.

Example:

- Integer (-4, -1, 0, 29) is an abstract data type. Its operations include add, subtract, multiply etc.
- Bit vectors is a data structure. It is an implementation of an integer.
- Hexadecimal vectors is a data structure. It is an implementation of an integer.

Motivation:

- We can specify the input and data forms of our algorithms in terms of ADT, and not worry about the actual implementation until we need to implement it.
- We can change **data structure** implementations to cater for the problem at hand.

Fundamental ADTs

- set
- sequences
- dictionary/map
- stack
- queue
- priority queue
- graph
- tree

Sets

A *set* is a collection of distinguishable objects, often called *members* or *elements*.

- Sets can be finite or infinite, and do not impose any ordering on the members.
- Typical operations: add, remove, search
- Each element may only appear in the set once. If elements may appear multiple times, the resulting collection is referred to as a **bag** or **multiset**.

Example (Sets)

Simple examples of sets include:

- ① binary : $S_1 = \{0, 1\}$,
- ② character : $S_2 = \{c, a, y, s, t\}$,
- ③ word : $S_3 = \{\text{car}, \text{house}, \text{man}, \text{sat}, \text{truck}\}$,
- ④ delimiter : $S_4 = \{\{\!\!\mid\!\!\}, \{\!\!\;\!\!\}, \{\?\!\!\}, \{\.\!\!\}\}$,
- ⑤ variable-length, prefix-free bit sequences:
 $S_5 = \{0, 10, 110, 1110\}$.

Sequences

A *sequence* is a collection of elements in which the **order** of the elements must be maintained, and elements can occur **any number of times**.

- A sequence is simply a multiset in which the **order** is important.
- Typical operations: add, remove, search
- In computer science, can also be referred to as a **list** (ordered collection of elements).

Sequences

Example (Sequences)

Examples of sequences include:

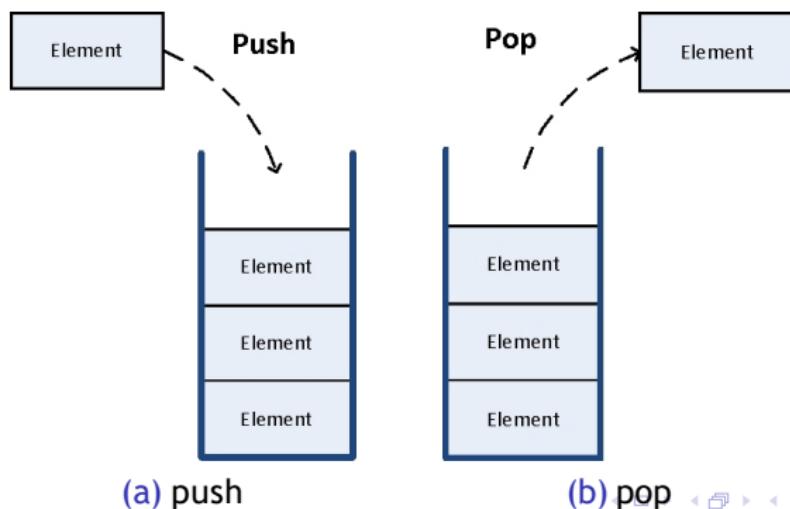
- ① Binary : $T_1 = "101010110001"$,
- ② English : $T_2 = \text{"The red car belongs to me."}$,
- ③ Genomic : $T_3 = \text{"gattcaggaatccgccggtaacgcgcatataattt"}$.

Stack

Stack

A *stack* is a collection with two defining operations, *push* and *pop*. Push adds new element at top of stack, pop removes element at top from stack.

Stack ADT implements LIFO principle (last in, first out).

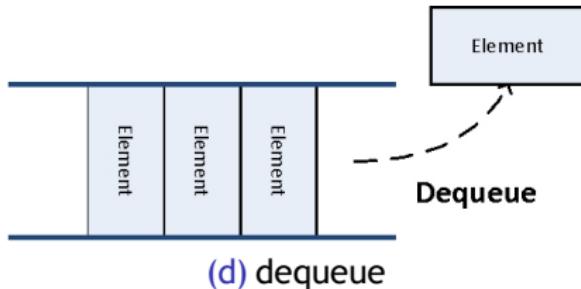
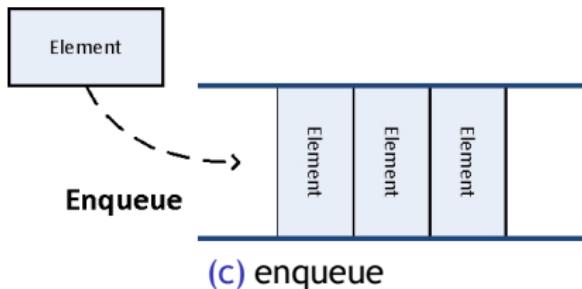


Queue

Queue

A *queue* is a collection with two defining operations, *enqueue* and *dequeue*. Enqueue adds new element at back of queue, dequeue removes element at front from queue.

Queue ADT implements FIFO principle (first in, first out).



Priority Queue

Queue

A **priority queue** is a queue that has a **priority** associated with each element it holds. Enqueue is the same as a (non-priority) queue, but for **dequeue**, the element with the **highest** priority is removed first.

Example: Queue for boarding planes, passengers who are members of the airlines or elderly have priority board plane (dequeue).

Dictionaries/Maps

Dictionary/Maps

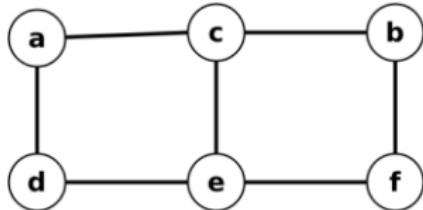
A *dictionary/map* is a collection of *(key,value)* pairs, such that each key can only appear once in the dictionary/map.

Example (Dictionary/map)

Example of a dictionary of current leaders and countries they led:

key	value
Scott Morrison	Australia
Xi Jinping	China
Boris Johnson	UK
Joe Biden	US

Graphs



A **graph** $G = \langle V, E \rangle$ is defined by a pair of two sets: a finite set V of items called **vertices** and a set E called **edges**, representing links/relations/connections between pairs of vertices.

Example:

$$V = \{a, b, c, d, e, f\}$$

$$E = \{(a, d), (a, c), (d, e), (c, e), (c, b), (e, f), (b, f)\}$$

Graphs

- A graph G is **undirected** if the edges do not have a direction (i.e., all of the pairs of vertices in E are unordered).
- A graph G is **directed** if the edges from a direction (i.e., all of the pairs of vertices in E have an ordering imposed).

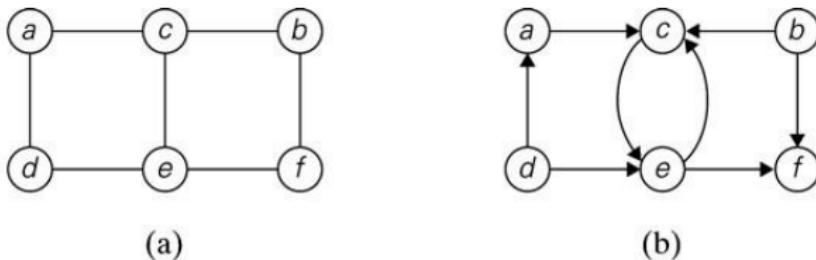
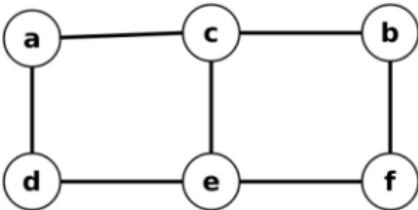


FIGURE 1.6 (a) Undirected graph. (b) Digraph.

Graph Representations



How to represent a graph without drawing it? Vertex and edge lists (previously)? Other possibilities?

	<i>a</i>	<i>b</i>	<i>c</i>	<i>d</i>	<i>e</i>	<i>f</i>
<i>a</i>	0	0	1	1	0	0
<i>b</i>	0	0	1	0	0	1
<i>c</i>	1	1	0	0	1	0
<i>d</i>	1	0	0	0	1	0
<i>e</i>	0	0	1	1	0	1
<i>f</i>	0	1	0	0	1	0

(a)

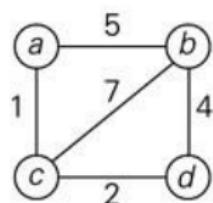
<i>a</i>	→	<i>c</i>	→	<i>d</i>
<i>b</i>	→	<i>c</i>	→	<i>f</i>
<i>c</i>	→	<i>a</i>	→	<i>b</i>
<i>d</i>	→	<i>a</i>	→	<i>e</i>
<i>e</i>	→	<i>c</i>	→	<i>d</i>
<i>f</i>	→	<i>b</i>	→	<i>e</i>

(b)

FIGURE 1.7 (a) Adjacency matrix and (b) adjacency lists of the graph in Figure 1.6a

Weighted Graphs

What if edges have weights associated with them?



(a)

	<i>a</i>	<i>b</i>	<i>c</i>	<i>d</i>
<i>a</i>	∞	5	1	∞
<i>b</i>	5	∞	7	4
<i>c</i>	1	7	∞	2
<i>d</i>	∞	4	2	∞

(b)

<i>a</i>
<i>b</i>
<i>c</i>
<i>d</i>

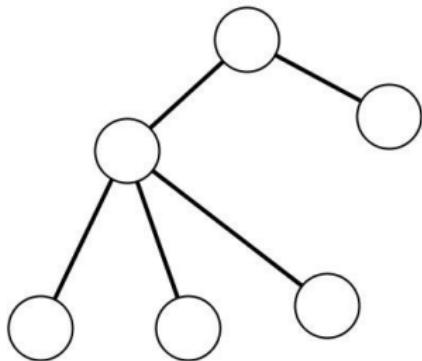
$\rightarrow b, 5 \rightarrow c, 1$
 $\rightarrow a, 5 \rightarrow c, 7 \rightarrow d, 4$
 $\rightarrow a, 1 \rightarrow b, 7 \rightarrow d, 2$
 $\rightarrow b, 4 \rightarrow c, 2$

(c)

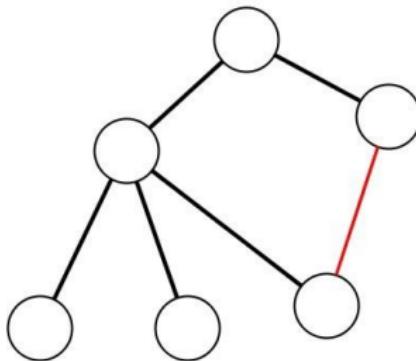
FIGURE 1.8 (a) Weighted graph. (b) Its weight matrix. (c) Its adjacency lists.

Trees

A **tree** is a connected acyclic graph.



(e) tree



(f) Not a tree (but a graph)

Data Structures

Now we describe several data structures that can implement different abstract data types.

We will learn more data structures as we progress through the course.

Two important data structures are [array](#) and [linked-list](#).

Both can implement many ADT, e.g., [sequences](#), [set](#), [dictionary](#).

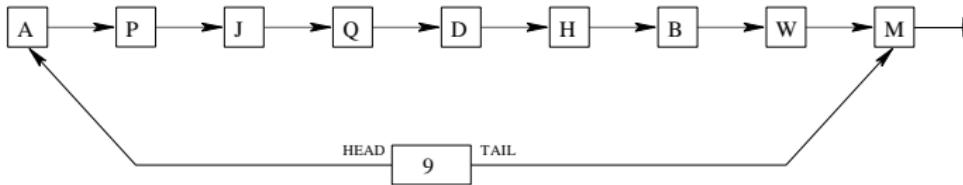
Array

0	1	2	3	4	5	6	7	8
A	P	J	Q	D	H	B	W	M

Characteristics:

- Collection of elements usually stored in a continuous manner, accessed by indices.
- Fast random access.
- Resizing can be costly (may require copying), hence often need to estimate size of array beforehand.

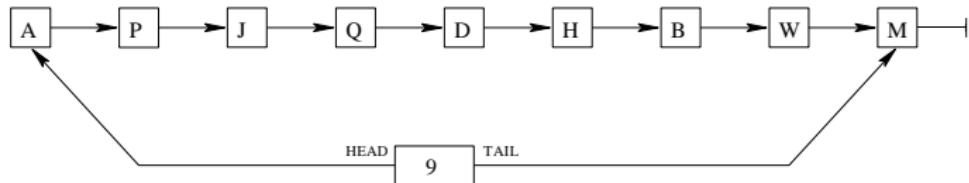
Linked Lists



- Each node stores element and a pointer to next node.
- There is a **head pointer** that points to the start of the list.
- Sometimes there is a **tail pointer** that points to the end of the list.
- Other variants of linked list (e.g., doubly linked lists, go to labs to see!)

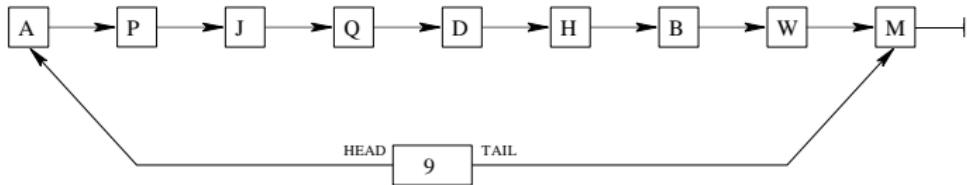
Linked Lists - insertion

Insert the elements of following sequence of letters, one by one
[A,P,J,Q,D,H,B,W,M]:



What if I wanted to insert element C between Q and D?

Linked Lists - deletion



Given linked-list, delete B:

Summary

- What is an algorithm, examples of it, and why the study of algorithms and data structures are important.
- Discussed the GCD problem and how it can be solved.
- Learnt about abstract data structures and two data structures (arrays and linked lists).
- Ultimately, remember the study of algorithms and data structures is for us to become better at solving problem.

Levitin - The design and analysis of algorithms

This week we will be covering the material from Chapter 2.

Learning outcomes:

- Understand why it is important to be able to compare the complexity of algorithms.
- Be able to:
 - measure complexity of algorithms and compare complexity classes.
 - analysis of non-recursive algorithms.
 - analysis of recursive algorithms.
- Be able to perform empirical analysis of algorithms.

Example: Which is faster?

Scenario: You work for a small medicine stocker. Your boss comes and asks your opinion on which of **two approaches is faster** for employees to **search** over an **unordered set of medicines (strings)**.



Question: Which of the following two approaches will you tell your boss is faster?

- Solution A: Sequential search.

Example: Which is faster?

Scenario: You work for a small medicine stocker. Your boss comes and asks your opinion on which of **two approaches is faster** for employees to **search** over an **unordered set of medicines (strings)**.



Question: Which of the following two approaches will you tell your boss is faster?

- Solution A: Sequential search.
- Solution B: Sort then search.

Which is faster?

- In this lecture, we look at the ways of estimating the running time of a program and how to compare the running times of two programs **without ever implementing them**.

Which is faster?

- In this lecture, we look at the ways of estimating the running time of a program and how to compare the running times of two programs **without ever implementing them**.
- It is vital to analyse the resource use of an algorithm, well before it is implemented and deployed.

Which is faster?

- In this lecture, we look at the ways of estimating the running time of a program and how to compare the running times of two programs **without ever implementing them**.
- It is vital to analyse the resource use of an algorithm, well before it is implemented and deployed.

Space is also important but we focus on **time** in this course.

Overview

- 1 Overview
- 2 Fundamentals
- 3 Asymptotic Complexity
- 4 Analysing Non-recursive Algorithms
- 5 Analysing Recursive Algorithms
- 6 Empirical Analysis
- 7 Rule of thumb Estimation of Complexity
- 8 Summary

Theoretical Analysis of Time Efficiency

How to estimate the running time of an algorithm?

Theoretical Analysis of Time Efficiency

How to estimate the running time of an algorithm?

Idea: An algorithm consists of some **operations** executed a number of **times**.

Hence, an **estimate of the running time/time efficiency** of an algorithm can be obtained from determining these **operations**, how long to execute them, and the **number of times** they are executed.

Theoretical Analysis of Time Efficiency

How to estimate the running time of an algorithm?

Idea: An algorithm consists of some **operations** executed a number of **times**.

Hence, an **estimate of the running time/time efficiency** of an algorithm can be obtained from determining these **operations**, how long to execute them, and the **number of times** they are executed.

These operations are called **basic operations** and the number of times is based on the **input size** of the problem.

Running example: a^n

This algorithm computes a^n :

```
// INPUT : a, n
// OUTPUT : s = an
1: set s = 1
2: for i = 1 to n do
3:   s = s * a
4: end for
5: return s
```

Basic Operation

What is a **basic operation**?

Basic Operation

What is a **basic operation**?

- Operation(s) that contribute most towards the **total** running time.

Basic Operation

What is a **basic operation**?

- Operation(s) that contribute most towards the **total** running time.
- Examples: compare, add, multiply, divide or assignment.

Basic Operation

What is a **basic operation**?

- Operation(s) that contribute most towards the **total** running time.
- Examples: compare, add, multiply, divide or assignment.
- Typically the operation most frequently executed, although dependent on the time of each operation.

Basic Operation

What is a **basic operation**?

- Operation(s) that contribute most towards the **total** running time.
- Examples: compare, add, multiply, divide or assignment.
- Typically the operation most frequently executed, although dependent on the time of each operation.

What is the **basic operation** of our example algorithm?

```
// INPUT : a, n  
// OUTPUT : s = an  
1: set s = 1  
2: for i = 1 to n do  
3:   s = s * a  
4: end for  
5: return s
```

Input Size

What is the **input size**?

- More of a characteristic of the problem, “Size of the problem”.

Input Size

What is the **input size**?

- More of a characteristic of the problem, “Size of the problem”.
- E.g., searching through an array of size n , the input size is n .

Input Size

What is the **input size**?

- More of a characteristic of the problem, “Size of the problem”.
- E.g., searching through an array of size n , the input size is n .
- We want to estimate the running time of an algorithm in terms of the problem, not in absolute terms.

Input Size

What is the **input size**?

- More of a characteristic of the problem, “Size of the problem”.
- E.g., searching through an array of size n , the input size is n .
- We want to estimate the running time of an algorithm in terms of the problem, not in absolute terms.
- When analysing algorithms, we state the **number of times that the basic operation is executed in terms of the input size**.

Input Size

What is the **input size**?

- More of a characteristic of the problem, “Size of the problem”.
- E.g., searching through an array of size n , the input size is n .
- We want to estimate the running time of an algorithm in terms of the problem, not in absolute terms.
- When analysing algorithms, we state the **number of times that the basic operation is executed in terms of the input size**.

What is the **input size** of our example algorithm?

```
//INPUT : a, n
//OUTPUT : s = an
1: set s = 1
2: for i = 1 to n do
3:   s = s * a
4: end for
5: return s
```

More examples of basic operation and input size

Problem Type	Basic Operation	Input Size
Iterating through an array, of size n , to print its contents	?	n
Iterating through an $n \times n$ matrix of integers, to find a given integer	Comparison	?

Theoretical Analysis of Time Efficiency

With the basic operation and input size, how do we estimate the running time of an algorithm?

Theoretical Analysis of Time Efficiency

With the basic operation and input size, how do we estimate the running time of an algorithm?

Running time approximately equal to time to execute a basic operation
 × number of basic operations

Theoretical Analysis of Time Efficiency

With the basic operation and input size, how do we estimate the running time of an algorithm?

Running time approximately equal to time to execute a basic operation
 × number of basic operations

$$t(n) \approx c_{op} \times C(n)$$

- $t(n)$ is the running time.
- n is the input size.
- c_{op} is the execution time for a basic operation.
- $C(n)$ is the number of times the basic operation is executed.

Estimating algorithm for a^n

What is the theoretical running time for our algorithm for computing a^n ?

Recall: $t(n) \approx c_{op} \times C(n)$

```
// INPUT : a, n
// OUTPUT : s = an
1: set s = 1
2: for i = 1 to n do
3:   s = s * a
4: end for
5: return s
```

Estimating algorithm for a^n

What is the theoretical running time for our algorithm for computing a^n ?

Recall: $t(n) \approx c_{op} \times C(n)$

```
//INPUT : a, n
//OUTPUT : s = an
1: set s = 1
2: for i = 1 to n do
3:   s = s * a
4: end for
5: return s
```

Answer: $C(n) = n$ (number of times basic op. executed)

Estimating algorithm for a^n

What is the theoretical running time for our algorithm for computing a^n ?

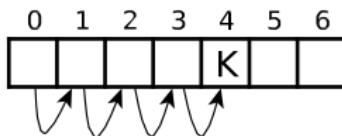
Recall: $t(n) \approx c_{op} \times C(n)$

```
//INPUT : a, n
//OUTPUT : s = an
1: set s = 1
2: for i = 1 to n do
3:   s = s * a
4: end for
5: return s
```

Answer: $C(n) = n$ (number of times basic op. executed)

$t(n) \approx c_{op} \times C(n) = c_{op} \times n$

Example: Searching for a key in n items using Sequential Search



ALGORITHM **SequentialSearch** ($A[0 \dots n - 1], K$)

//INPUT : An array A of length n and a search key K .

// OUTPUT : The index of the first element of A which matches K or n (length of A) otherwise.

- 1: set $i = 0$
- 2: **while** $i < n$ and $A[i] \neq K$ **do**
- 3: set $i = i + 1$
- 4: **end while**
- 5: **return** i

Example: Searching for a key in n items using Sequential Search

```
ALGORITHM SequentialSearch ( $A[0 \dots n - 1], K$ )
//INPUT : An array A of length  $n$  and a search key  $K$ .
//OUTPUT : The index of the first element of A which matches  $K$  or  $n$ 
(length of A) otherwise.
```

- 1: set $i = 0$
- 2: **while** $i < n$ and $A[i] \neq K$ **do**
- 3: set $i = i + 1$
- 4: **end while**
- 5: **return** i

- Basic Operation?

Example: Searching for a key in n items using Sequential Search

```
ALGORITHM SequentialSearch ( $A[0 \dots n - 1], K$ )
//INPUT : An array  $A$  of length  $n$  and a search key  $K$ .
//OUTPUT : The index of the first element of  $A$  which matches  $K$  or  $n$ 
(length of  $A$ ) otherwise.
```

- 1: set $i = 0$
- 2: **while** $i < n$ and $A[i] \neq K$ **do**
- 3: set $i = i + 1$
- 4: **end while**
- 5: **return** i

- Basic Operation? comparison, addition, assignment (any of these are okay)

Example: Searching for a key in n items using Sequential Search

```
ALGORITHM SequentialSearch ( $A[0 \dots n - 1], K$ )
//INPUT : An array  $A$  of length  $n$  and a search key  $K$ .
//OUTPUT : The index of the first element of  $A$  which matches  $K$  or  $n$ 
(length of  $A$ ) otherwise.
```

```
1: set  $i = 0$ 
2: while  $i < n$  and  $A[i] \neq K$  do
3:   set  $i = i + 1$ 
4: end while
5: return  $i$ 
```

- Basic Operation? comparison, addition, assignment (any of these are okay)
- Input size?

Example: Searching for a key in n items using Sequential Search

```
ALGORITHM SequentialSearch ( $A[0 \dots n - 1], K$ )
//INPUT : An array  $A$  of length  $n$  and a search key  $K$ .
//OUTPUT : The index of the first element of  $A$  which matches  $K$  or  $n$ 
(length of  $A$ ) otherwise.
```

```
1: set  $i = 0$ 
2: while  $i < n$  and  $A[i] \neq K$  do
3:   set  $i = i + 1$ 
4: end while
5: return  $i$ 
```

- Basic Operation? comparison, addition, assignment (any of these are okay)
- Input size? n

Example: Searching for a key in n items using Sequential Search

What is the theoretical running time for Sequential Search?

Recall: $t(n) \approx c_{op} \times C(n)$

ALGORITHM SequentialSearch ($A[0 \dots n - 1], K$)

//INPUT : An array A of length n and a search key K .

//OUTPUT : The index of the first element of A which matches K or n (length of A) otherwise.

- 1: set $i = 0$
- 2: **while** $i < n$ and $A[i] \neq K$ **do**
- 3: set $i = i + 1$
- 4: **end while**
- 5: **return** i

What is $C(n)$, the number of times the basic operation is executed?

Runtime Complexity

- **Worst Case** - Given an input of n items, what is the **maximum** running time for any possible input?

Runtime Complexity

- **Worst Case** - Given an input of n items, what is the **maximum** running time for any possible input?
- **Best Case** - Given an input of n items, what is the **minimum** running time for any possible input?

Runtime Complexity

- **Worst Case** - Given an input of n items, what is the **maximum** running time for any possible input?
- **Best Case** - Given an input of n items, what is the **minimum** running time for any possible input?
- **Average Case** - Given an input of n items, what is the **average** running time across all possible inputs?

Runtime Complexity

- **Worst Case** - Given an input of n items, what is the **maximum** running time for any possible input?
- **Best Case** - Given an input of n items, what is the **minimum** running time for any possible input?
- **Average Case** - Given an input of n items, what is the **average** running time across all possible inputs?

NOTE : **Average Case** is *not* the average of the worst and best case. Rather, it is the average performance across all possible inputs.

Sequential Search

ALGORITHM **SequentialSearch** ($A[0 \dots n - 1], K$)

```
1: set  $i = 0$ 
2: while  $i < n$  and  $A[i] \neq K$  do
3:   set  $i = i + 1$ 
4: end while
5: return  $i$ 
```

Best-case : The best case input is when the item being searched for is the **first** item in the list, so $C_b(n) = 1$.

Worst-case : The worst case input is when the item being searched for is **not present** in the list, so $C_w(n) = n$

Sequential Search

Average-case : What does average-case mean?

- Recall: average across all possible inputs - how to analyse this?
- Typically not straight forward.

Sequential Search

Average-case : What does average-case mean?

- Recall: average across all possible inputs - how to analyse this?
- Typically not straight forward.

Sequential Search Example:

- How often do we search for items **in** the array?
 - Where in the array do we find the item? 1st, 2nd, ..., last position?
- How often do we search for items **not** in the array?

Sequential Search

Average-case Analysis : Skipping a few steps (notes of full derivation available on Canvas) and p is the probability of a successful search.

$$C_{avg}(n) = \frac{p(n + 1)}{2} + n(1 - p)$$

If $p = 1$, then $C_{avg}(n) = (n + 1)/2$.

If $p = 0$, then $C_{avg}(n) = n$.

Summary for this part

- Input size, basic operation
- Time complexity estimate using input size and basic operation
- Best, worst, and average cases

Overview

- 1 Overview
- 2 Fundamentals
- 3 Asymptotic Complexity
- 4 Analysing Non-recursive Algorithms
- 5 Analysing Recursive Algorithms
- 6 Empirical Analysis
- 7 Rule of thumb Estimation of Complexity
- 8 Summary

Asymptotic Complexity

Problem:

- We now have a way to analyse the **running time** (aka **time complexity**) of an algorithm, but every algorithms have their own time complexities. How to **compare** in a meaningful way?

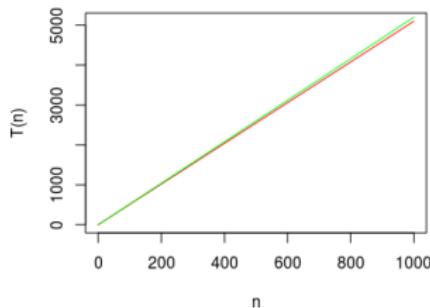
Asymptotic Complexity

Consider the running times estimates of two algorithms:

$$\text{Algorithm 1: } T_1(n) = 5.1n$$

$$\text{Algorithm 2: } T_2(n) = 5.2n$$

Similar timing profiles as n grows?



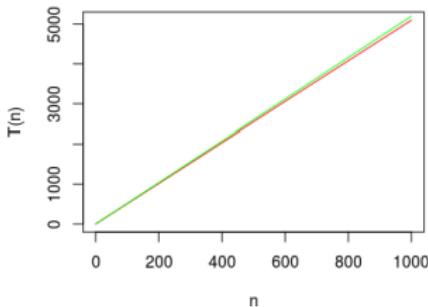
Asymptotic Complexity

Consider the running times estimates of two algorithms:

$$\text{Algorithm 1: } T_1(n) = 5.1n$$

$$\text{Algorithm 2: } T_2(n) = 5.2n$$

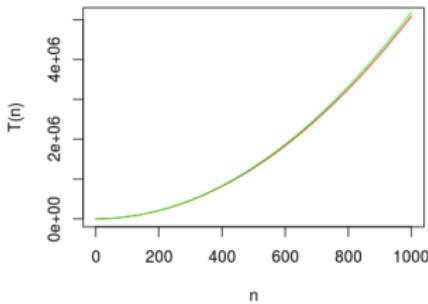
Similar timing profiles as n grows?



What about the following?:

$$\text{Algorithm 3: } T_3(n) = 5.1n^2$$

$$\text{Algorithm 4: } T_4(n) = 5.2n^2$$



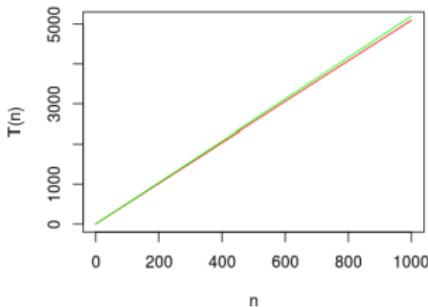
Asymptotic Complexity

Consider the running times estimates of two algorithms:

$$\text{Algorithm 1: } T_1(n) = 5.1n$$

$$\text{Algorithm 2: } T_2(n) = 5.2n$$

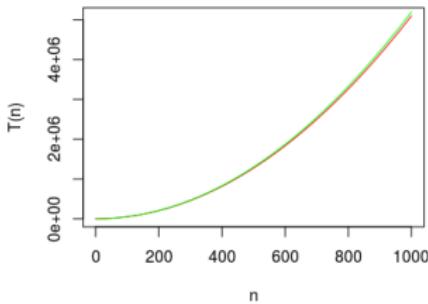
Similar timing profiles as n grows?



What about the following?:

$$\text{Algorithm 3: } T_3(n) = 5.1n^2$$

$$\text{Algorithm 4: } T_4(n) = 5.2n^2$$



Asymptotic Complexity

Problem:

- We now have a way to analyse the **running time** (aka **time complexity**) of an algorithm, but every algorithms have their own time complexities. How to **compare** in a meaningful way?

Solution:

- Group them into **equivalence classes** (for easier comparison and understanding), with respect to the input size.
- Focus of this part, **asymptotic complexity** and **equivalence classes**.

Asymptotic Complexity - bounds

Warning: Some (mathematical) definitions coming up!

Asymptotic Complexity - bounds

Warning: Some (mathematical) definitions coming up!

Idea: Use bounds and asymptotic complexity (as n becomes **large**, what is the **dominant term** contributing to the running time ($t(n)$)?)

Asymptotic Complexity, Upper bounds

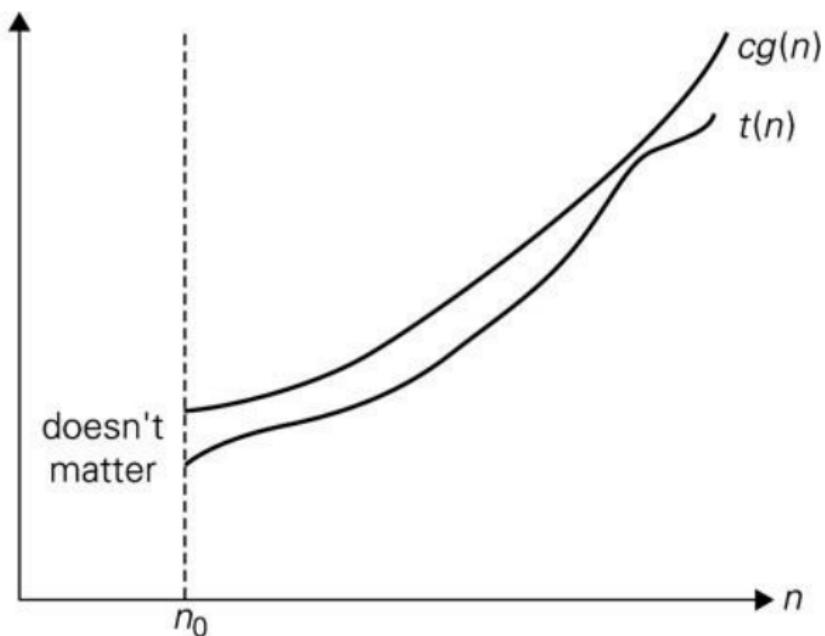
Definition: Given a function $t(n)$ (the running time of an algorithm):

- Let $c \times g(n)$ be a function that is an **upper bound** on $t(n)$ for some $c > 0$ and for “*large*” n .

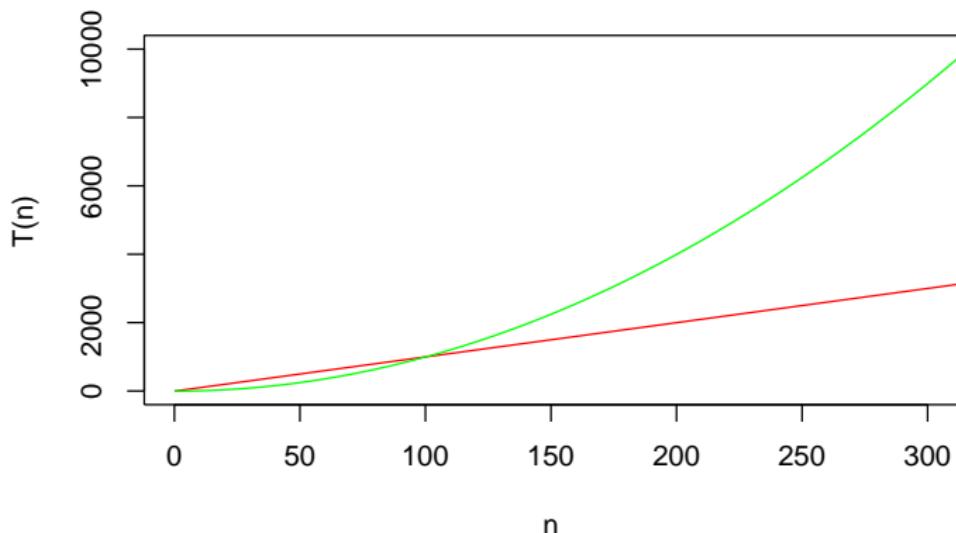
Asymptotic Complexity, Upper bounds

Definition: Given a function $t(n)$ (the running time of an algorithm):

- Let $c \times g(n)$ be a function that is an **upper bound** on $t(n)$ for some $c > 0$ and for “large” n .



Upper bounds Examples



Asymptotic Complexity, $O(n)$

Rather than talking about individual upper bounds for each running time function, we seek to group them into *equivalence classes* that describe their **order of growth**.

Asymptotic Complexity, $O(n)$

Rather than talking about individual upper bounds for each running time function, we seek to group them into *equivalence classes* that describe their **order of growth**.

Big-O notation, $O(n)$: Given a function $t(n)$,

- Informally: $t(n) \in O(g(n))$ means $g(n)$ is a **function** that, as n increases, **provides an upper bound** for $t(n)$.
- Formally: $t(n) \in O(g(n))$, if $g(n)$ is a function and $c \times g(n)$ is an **upper bound** on $t(n)$ for some $c > 0$ and for “*large*” n .

Asymptotic Complexity, $O(n)$

Rather than talking about individual upper bounds for each running time function, we seek to group them into *equivalence classes* that describe their **order of growth**.

Big-O notation, $O(n)$: Given a function $t(n)$,

- Informally: $t(n) \in O(g(n))$ means $g(n)$ is a **function** that, as n increases, **provides an upper bound** for $t(n)$.
- Formally: $t(n) \in O(g(n))$, if $g(n)$ is a function and $c \times g(n)$ is an **upper bound** on $t(n)$ for some $c > 0$ and for “large” n .

E.g., if $t(n) = 5.1n$,

- $g(n) = n$

Asymptotic Complexity, $O(n)$

Rather than talking about individual upper bounds for each running time function, we seek to group them into *equivalence classes* that describe their **order of growth**.

Big-O notation, $O(n)$: Given a function $t(n)$,

- Informally: $t(n) \in O(g(n))$ means $g(n)$ is a **function** that, as n increases, **provides an upper bound** for $t(n)$.
- Formally: $t(n) \in O(g(n))$, if $g(n)$ is a function and $c \times g(n)$ is an **upper bound** on $t(n)$ for some $c > 0$ and for “large” n .

E.g., if $t(n) = 5.1n$,

- $g(n) = n$
- $g(n) = 0.001n - 6$

Asymptotic Complexity, $O(n)$

Rather than talking about individual upper bounds for each running time function, we seek to group them into *equivalence classes* that describe their **order of growth**.

Big-O notation, $O(n)$: Given a function $t(n)$,

- Informally: $t(n) \in O(g(n))$ means $g(n)$ is a **function** that, as n increases, **provides an upper bound** for $t(n)$.
- Formally: $t(n) \in O(g(n))$, if $g(n)$ is a function and $c \times g(n)$ is an **upper bound** on $t(n)$ for some $c > 0$ and for “large” n .

E.g., if $t(n) = 5.1n$,

- $g(n) = n$
- $g(n) = 0.001n - 6$
- $g(n) = n^2$

Asymptotic Complexity, $O(n)$

Rather than talking about individual upper bounds for each running time function, we seek to group them into *equivalence classes* that describe their **order of growth**.

Big-O notation, $O(n)$: Given a function $t(n)$,

- Informally: $t(n) \in O(g(n))$ means $g(n)$ is a **function** that, as n increases, **provides an upper bound** for $t(n)$.
- Formally: $t(n) \in O(g(n))$, if $g(n)$ is a function and $c \times g(n)$ is an **upper bound** on $t(n)$ for some $c > 0$ and for “large” n .

E.g., if $t(n) = 5.1n$,

- $g(n) = n$
- $g(n) = 0.001n - 6$
- $g(n) = n^2$

Asymptotic Complexity, $O(n)$

Rather than talking about individual upper bounds for each running time function, we seek to group them into *equivalence classes* that describe their **order of growth**.

Big-O notation, $O(n)$: Given a function $t(n)$,

- Informally: $t(n) \in O(g(n))$ means $g(n)$ is a **function** that, as n increases, **provides an upper bound** for $t(n)$.
- Formally: $t(n) \in O(g(n))$, if $g(n)$ is a function and $c \times g(n)$ is an **upper bound** on $t(n)$ for some $c > 0$ and for “large” n .

E.g., if $t(n) = 5.1n$,

- $g(n) = n$
- $g(n) = 0.001n - 6$
- $g(n) = n^2$

What $g(n)$ to use if $t(n) = 5.2n$?

Asymptotic Complexity, $O(n)$

Rather than talking about individual upper bounds for each running time function, we seek to group them into *equivalence classes* that describe their **order of growth**.

Big-O notation, $O(n)$: Given a function $t(n)$,

- Informally: $t(n) \in O(g(n))$ means $g(n)$ is a **function** that, as n increases, **provides an upper bound** for $t(n)$.
- Formally: $t(n) \in O(g(n))$, if $g(n)$ is a function and $c \times g(n)$ is an **upper bound** on $t(n)$ for some $c > 0$ and for “large” n .

E.g., if $t(n) = 5.1n$,

- $g(n) = n$
- $g(n) = 0.001n - 6$
- $g(n) = n^2$

What $g(n)$ to use if $t(n) = 5.2n$? Any of the above $g(n)$ functions are possible!

Common Equivalence Classes

Previous slide, we had multiple upper bounds. But we can in fact group them to a smaller number.

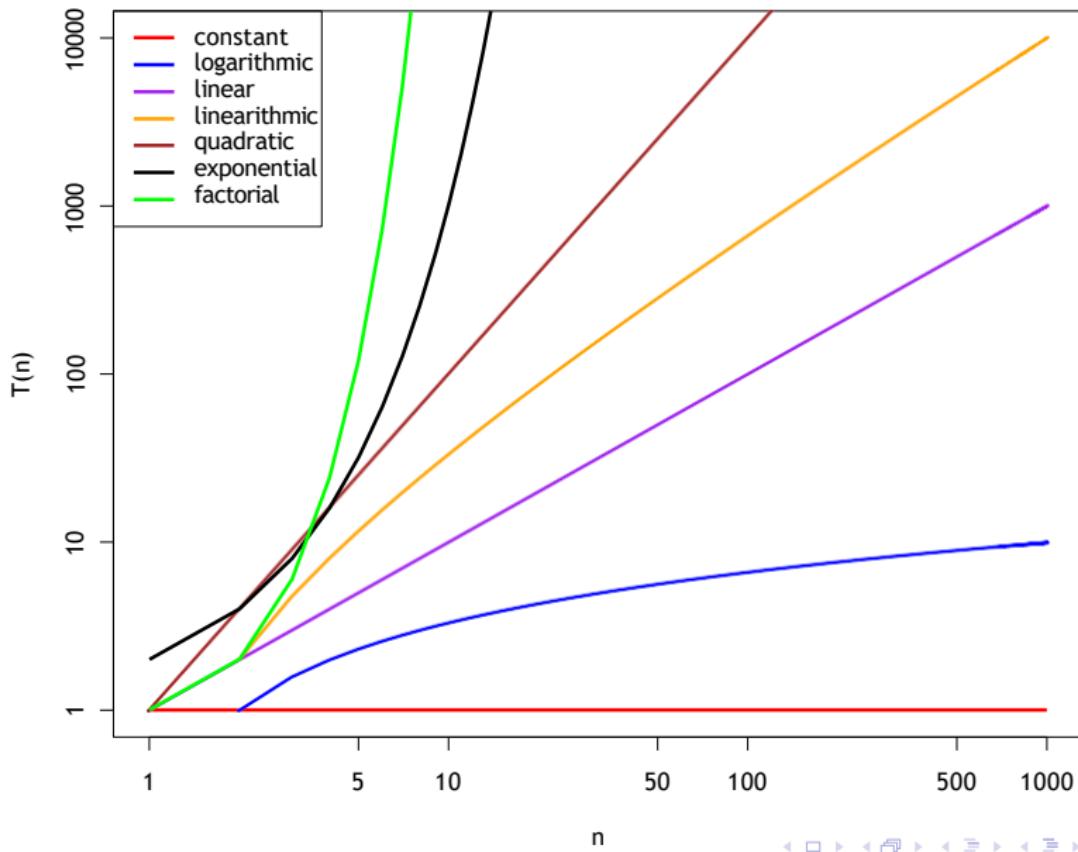
Common Equivalence Classes

Previous slide, we had multiple upper bounds. But we can in fact group them to a smaller number.

First we look at common equivalence classes:

- Constant - $O(1)$: Access array element
- Logarithmic - $O(\log n)$: Binary search
- Linear - $O(n)$: Link list search
- Linearithmic (Supralinear) - $O(n \log n)$: Merge Sorting
- Quadratic - $O(n^2)$: Selection Sorting
- Exponential - $O(2^n)$: Generating all subsets
- Factorial - $O(n!)$: Generating all permutations

Common Complexity Bounds



Asymptotic Complexity

Recall: We want to find equivalence function class that upper bounds different $t(n)$.

Asymptotic Complexity

Recall: We want to find equivalence function class that upper bounds different $t(n)$.

But we might be given an upper bound $g(n)$ that isn't quite in the form of the equivalence classes. How to get the equivalence classes?

Asymptotic Complexity

Recall: We want to find equivalence function class that upper bounds different $t(n)$.

But we might be given an upper bound $g(n)$ that isn't quite in the form of the equivalence classes. How to get the equivalence classes?

Adhere to the following guidelines:

- Do not include constants (e.g., omit '-6' in $0.01n - 6$ to become $0.01n$).
- Omit all the lower order terms (e.g., $n^3 + n^2 + n$, we simplify to n^3).
- Omit the coefficient of the highest-order term (e.g., $0.01n$ becomes n).

Asymptotic Complexity

Recall: We want to find equivalence function class that upper bounds different $t(n)$.

But we might be given an upper bound $g(n)$ that isn't quite in the form of the equivalence classes. How to get the equivalence classes?

Adhere to the following guidelines:

- Do not include constants (e.g., omit '-6' in $0.01n - 6$ to become $0.01n$).
- Omit all the lower order terms (e.g., $n^3 + n^2 + n$, we simplify to n^3).
- Omit the coefficient of the highest-order term (e.g., $0.01n$ becomes n).

Returning to previous example, we can write the two upper bounds $O(0.01n - 6)$ and $O(n)$ as $O(n)$.

Example

Let

$$g(n) = 2n^4 + 43n^3 - n + 50$$

be an **upper** bound for the running time $t(n)$ of an algorithm.

Using the previous guidelines, what equivalence **class** should you use to describe the **order of growth** of the algorithm?

Asymptotic Complexity, Lower Bound ($\Omega(n)$)

Lower Bound Definition: Given a function $t(n)$,

- Informally: $t(n) \in \Omega(g(n))$ means $g(n)$ is a **function** that, as n increases, **is a lower bound** of $t(n)$
- Formally: $t(n) \in \Omega(g(n))$, if $g(n)$ is a function and $c \times g(n)$ is a lower bound on $t(n)$ for some $c > 0$ and for “large” n .

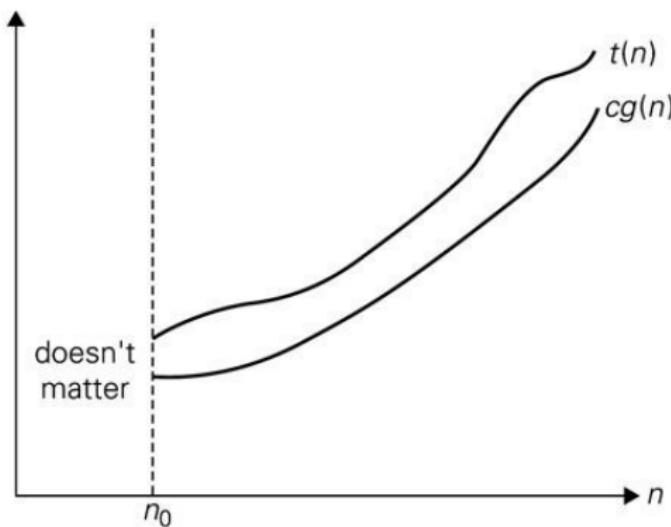
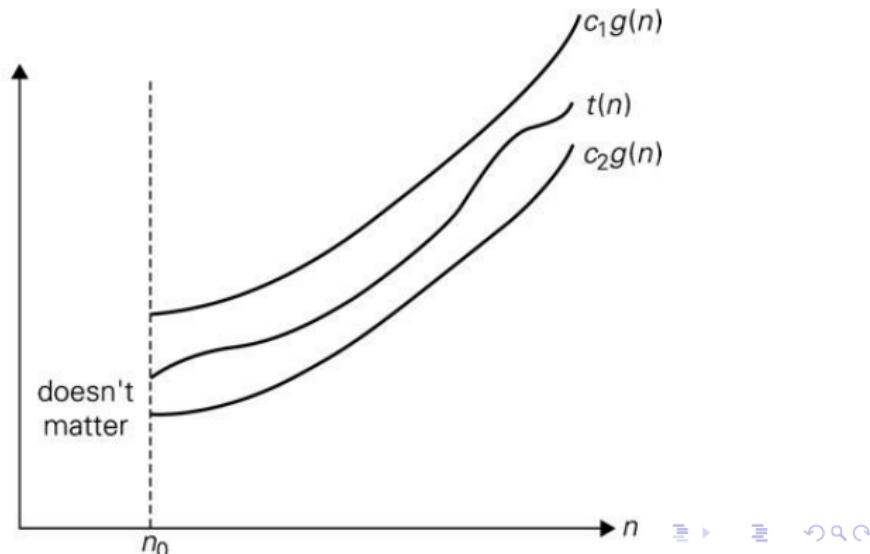


FIGURE 2.2 Big-omega notation: $t(n) \in \Omega(g(n))$

Asymptotic Complexity, Exact Bounds ($\Theta(n)$)

Exact Bound Definition: Given a function $t(n)$,

- Informally: $t(n) \in \Theta(g(n))$ means $g(n)$ is a **function** that, as n increases, is both a **upper and lower bound** of $t(n)$
- Formally: $t(n) \in \Theta(g(n))$, if $g(n)$ is a function and $c_1 \times g(n)$ is an **upper bound** on $t(n)$ and $c_2 \times g(n)$ is an **lower bound** on $t(n)$, for some $c_1 > 0$ and $c_2 > 0$ and for “*large*” n



Examples

$t(n)$	$O(n)$	$O(n^2)$	$O(n^3)$	$\Omega(n)$	$\Omega(n^2)$	$\Omega(n^3)$	Θ
$\log_2 n$	T	T	T	F	F	F	$\Theta(\log_2 n)$
$10n + 5$	T	T	T	T	F	F	$\Theta(n)$
$n(n - 1)/2$	F	T	T	T	T	F	$\Theta(n^2)$
$(n + 1)^3$	F	F	T	T	T	T	$\Theta(n^3)$
2^n	F	F	F	T	T	T	$\Theta(2^n)$

For example, $10n + 5$ is in $O(n)$.

Terminology Clarification

- $O(n)$ is not the same thing as “Worst Case Efficiency”.
- $\Omega(n)$ is not the same thing as “Best Case Efficiency”.
- $\Theta(n)$ is not the same thing as “Average Case Efficiency”.

It is perfectly reasonable to want the $\Omega(n)$ *and* $O(n)$ worst case efficiency bounds for a class of algorithms.

Terminology Clarification

- $O(n)$ is not the same thing as “Worst Case Efficiency”.
- $\Omega(n)$ is not the same thing as “Best Case Efficiency”.
- $\Theta(n)$ is not the same thing as “Average Case Efficiency”.

It is perfectly reasonable to want the $\Omega(n)$ *and* $O(n)$ worst case efficiency bounds for a class of algorithms.

Why all the bounds?

- Generally $O(n)$ is **most commonly** used , but **exact bounds** tell us the bounds are tight and the algorithm doesn't have anything outside what we expect.
- **Lower bounds** are useful to describe the (theoretical) limits of whole classes of algorithms, and also sometimes useful to state how fast can the best case reach.

Overview

- 1 Overview
- 2 Fundamentals
- 3 Asymptotic Complexity
- 4 Analysing Non-recursive Algorithms
- 5 Analysing Recursive Algorithms
- 6 Empirical Analysis
- 7 Rule of thumb Estimation of Complexity
- 8 Summary

Time Efficiency of Algorithms

Typically, we are given the pseudo code of an algorithm, not a nice $t(n)$ function.

Question

So how do we determine **bounds** on the **order of growth** of an algorithm?

- Next two parts, we focus on answering this question.

Time Efficiency of Algorithms

Typically, we are given the pseudo code of an algorithm, not a nice $t(n)$ function.

Question

So how do we determine **bounds** on the **order of growth** of an algorithm?

- Next two parts, we focus on answering this question.
- We first study how to do this for non-recursive algorithms, then recursive ones.

Time Efficiency of Algorithms

Typically, we are given the pseudo code of an algorithm, not a nice $t(n)$ function.

Question

So how do we determine **bounds** on the **order of growth** of an algorithm?

- Next two parts, we focus on answering this question.
- We first study how to do this for non-recursive algorithms, then recursive ones.
- Keep in mind: Generally, we first need to estimate the running time $t(n)$, then determine a bound and order of growth.

Time Efficiency of Non-Recursive Algorithms

Question

So how do we determine **bounds** on the **order of growth** of an algorithm?

Recall $t(n) = c_{op}C(n)$.

Time Efficiency of Non-Recursive Algorithms

Question

So how do we determine **bounds** on the **order of growth** of an algorithm?

Recall $t(n) = c_{op}C(n)$.

- ① Determine what is used to measure the **input size**.
- ② Identify the algorithm's **basic operation**

Time Efficiency of Non-Recursive Algorithms

Question

So how do we determine **bounds** on the **order of growth** of an algorithm?

Recall $t(n) = c_{op}C(n)$.

- ① Determine what is used to measure the **input size**.
- ② Identify the algorithm's **basic operation**
- ③ Determine the **number of basic operation executions** in terms of the input size? (i.e., How to determine $C(n)$?)
 - Setup a **summation** for $C(n)$ reflecting the algorithm's loop structure.
 - **Simplify** the summation by using standard formulas in Appendix A of textbook.

Time Efficiency of Non-Recursive Algorithms

Question

So how do we determine **bounds** on the **order of growth** of an algorithm?

Recall $t(n) = c_{op}C(n)$.

- ① Determine what is used to measure the **input size**.
- ② Identify the algorithm's **basic operation**
- ③ Determine the **number of basic operation executions** in terms of the input size? (i.e., How to determine $C(n)$)?
 - Setup a **summation** for $C(n)$ reflecting the algorithm's loop structure.
 - **Simplify** the summation by using standard formulas in Appendix A of textbook.
- ④ Determine a **equivalence class** $g(n)$ that **bounds** $t(n)$. Recall we generally want the tightest bound possible.

Non Recursive Example: a^n

```
//INPUT : a, n  
//OUTPUT : s = an  
1: set s = 1  
2: for i = 1 to n do  
3:     s = s * a  
4: end for  
5: return
```

Non Recursive Example: a^n

```
//INPUT : a, n  
//OUTPUT : s = an  
1: set s = 1  
2: for i = 1 to n do  
3:   s = s * a  
4: end for  
5: return
```

Input size: n

Basic operation: multiplication

$C(n)$: ?

Non Recursive Example: a^n

```
//INPUT : a, n  
//OUTPUT : s = an  
1: set s = 1  
2: for i = 1 to n do  
3:   s = s * a  
4: end for  
5: return
```

Input size: n

Basic operation: multiplication

$C(n)$: ? $\underbrace{1 + 1 + 1 + \dots + 1}_{n}$ x

Non Recursive Example: a^n

```
//INPUT : a, n  
//OUTPUT : s = an  
1: set s = 1  
2: for i = 1 to n do  
3:   s = s * a  
4: end for  
5: return
```

Input size: n

Basic operation: multiplication

$$C(n) : ? \underbrace{1 + 1 + 1 + \dots + 1}_{n} = \sum_{i=1}^n 1$$

Non-Recursive Examples (continued): Adding two matrices

Given two (square) matrices A and B , both of dimensions n by n , the following algorithm computes $C = A + B$.

```
for (int i = 0; i <= n-1; i++) {  
    for (int j = 0; j <= n-1; j++) {  
        C[i, j] = A[i, j] + B[i, j];  
    }  
}
```

Non-Recursive Examples (continued): Adding two matrices

Given two (square) matrices A and B , both of dimensions n by n , the following algorithm computes $C = A + B$.

```
for (int i = 0; i <= n-1; i++) {  
    for (int j = 0; j <= n-1; j++) {  
        C[i, j] = A[i, j] + B[i, j];  
    }  
}
```

Input size: ?

Non-Recursive Examples (continued): Adding two matrices

Given two (square) matrices A and B , both of dimensions n by n , the following algorithm computes $C = A + B$.

```
for (int i = 0; i <= n-1; i++) {  
    for (int j = 0; j <= n-1; j++) {  
        C[i, j] = A[i, j] + B[i, j];  
    }  
}
```

Input size: ? ***n***

Basic operation: ?

Non-Recursive Examples (continued): Adding two matrices

Given two (square) matrices A and B , both of dimensions n by n , the following algorithm computes $C = A + B$.

```
for (int i = 0; i <= n-1; i++) {  
    for (int j = 0; j <= n-1; j++) {  
        C[i, j] = A[i, j] + B[i, j];  
    }  
}
```

Input size: ? *n*

Basic operation: ? addition

$C(n)$: ?

Non-Recursive Examples (continued): Adding two matrices

Given two (square) matrices A and B , both of dimensions n by n , the following algorithm computes $C = A + B$.

```
for (int i = 0; i <= n-1; i++) {  
    for (int j = 0; j <= n-1; j++) {  
        C[i, j] = A[i, j] + B[i, j];  
    }  
}
```

Input size: ? *n*

Basic operation: ? addition

$C(n)$: ?

$$\sum_{i=0}^{n-1} \sum_{j=0}^{n-1} 1$$

Time Efficiency of Non-Recursive Algorithms

Question

So how do we determine **bounds** on the **order of growth** of an algorithm?

Recall $t(n) = c_{op}C(n)$.

- ① Determine what is used to measure the input size.
- ② Identify the algorithm's basic operation.
- ③ Determine the number of basic operation executions in terms of the input size? (i.e., How to determine $C(n)$)?
 - Setup a summation for $C(n)$ reflecting the algorithm's loop structure.
 - **Simplify the summation by using standard formulas in Appendix A of textbook.**
- ④ Determine a function family $g(n)$ that bounds $t(n)$. Recall we generally want the tightest bound possible.

Useful series (from Appendix A)

- R1 (Sum): $\sum_{i=l}^n 1 = n - l + 1.$
- R2 (Geometric): $\sum_{i=1}^n i = \frac{n(n+1)}{2}.$
- R3 (Distributive): $\sum_{i=1}^n c * a_i = c \sum_{i=1}^n a_i.$
- R4 (Associative): $\sum_{i=1}^n (a_i \pm b_i) = \sum_{i=1}^n a_i \pm \sum_{i=1}^n b_i.$

Non Recursive Example a^n : Simplification

```
// INPUT : a, n
// OUTPUT : s = an
1: set s = 1
2: for i = 1 : n do
3:   s = s * a
4: end for
5: return
```

$$C(n) = \sum_{i=1}^n 1$$

Non Recursive Example a^n : Simplification

$$C(n) = \sum_{i=1}^n 1 \quad (1)$$

(2)

(3)

Non Recursive Example a^n : Simplification

$$C(n) = \sum_{i=1}^n 1 \quad (1)$$

$$= n - 1 + 1 \text{ (Use R1)} \quad (2)$$

(3)

Non Recursive Example a^n : Simplification

$$C(n) = \sum_{i=1}^{\Sigma n} 1 \quad (1)$$

$$= n - 1 + 1 \text{ (Use R1)} \quad (2)$$

$$= n \quad (3)$$

Non-Recursive Example Adding two matrices: Simplification

Given two (square) matrices A and B , both of dimensions n by n , the following algorithm computes $C = A + B$. **Example:**

```
for (int i = 0; i <= n-1; i++) {  
    for (int j = 0; j <= n-1; j++) {  
        C[i, j] = A[i, j] + B[i, j];  
    }  
}
```

$$C(n) = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} 1$$

Non-Recursive Example Adding two matrices: Simplification

$$C(n) = \sum_{i=0}^{\Sigma^{-1}} \sum_{j=0}^{\Sigma^{-1}} 1 \quad (1)$$

(2)

(3)

(4)

(5)

(6)

Non-Recursive Example Adding two matrices: Simplification

$$C(n) = \sum_{i=0}^{\Sigma-1} \sum_{j=0}^{\Sigma-1} 1 \quad (1)$$

$$= \sum_{i=0}^{\Sigma-1} (n - 1 + 1) \text{ (Use R1 on inner summation)} \quad (2)$$

(3)

(4)

(5)

(6)

Non-Recursive Example Adding two matrices: Simplification

$$C(n) = \sum_{i=0}^{\infty} \sum_{j=0}^{\infty} 1 \quad (1)$$

$$= \sum_{i=0}^{\infty} (n - 1 + 1) \text{ (Use R1 on inner summation)} \quad (2)$$

$$= \sum_{i=0}^{\infty} (n) \text{ (Simplify)} \quad (3)$$

(4)

(5)

(6)

Non-Recursive Example Adding two matrices: Simplification

$$C(n) = \sum_{i=0}^{\Sigma-1} \sum_{j=0}^{\Sigma-1} 1 \quad (1)$$

$$= \sum_{i=0}^{\Sigma-1} (n - 1 + 1) \text{ (Use R1 on inner summation)} \quad (2)$$

$$= \sum_{i=0}^{\Sigma-1} (n) \text{ (Simplify)} \quad (3)$$

$$= n \sum_{i=0}^{\Sigma-1} 1 \text{ (Use R3 to take } n \text{ out of summation)} \quad (4)$$

(5)

(6)

Non-Recursive Example Adding two matrices: Simplification

$$C(n) = \sum_{i=0}^{\Sigma-1} \sum_{j=0}^{\Sigma-1} 1 \quad (1)$$

$$= \sum_{i=0}^{\Sigma-1} (n - 1 + 1) \text{ (Use R1 on inner summation)} \quad (2)$$

$$= \sum_{i=0}^{\Sigma-1} (n) \text{ (Simplify)} \quad (3)$$

$$= n \sum_{i=0}^{\Sigma-1} 1 \text{ (Use R3 to take } n \text{ out of summation)} \quad (4)$$

$$= n * (n - 1 + 1) \text{ (Use R1 on remaining summation)} \quad (5)$$

(6)

Non-Recursive Example Adding two matrices: Simplification

$$C(n) = \sum_{i=0}^{\Sigma-1} \sum_{j=0}^{\Sigma-1} 1 \quad (1)$$

$$= \sum_{i=0}^{\Sigma-1} (n - 1 + 1) \text{ (Use R1 on inner summation)} \quad (2)$$

$$= \sum_{i=0}^{\Sigma-1} (n) \text{ (Simplify)} \quad (3)$$

$$= n \sum_{i=0}^{\Sigma-1} 1 \text{ (Use R3 to take } n \text{ out of summation)} \quad (4)$$

$$= n * (n - 1 + 1) \text{ (Use R1 on remaining summation)} \quad (5)$$

$$= n * n \text{ (Simplify)} \quad (6)$$

Overview

- 1 Overview
- 2 Fundamentals
- 3 Asymptotic Complexity
- 4 Analysing Non-recursive Algorithms
- 5 Analysing Recursive Algorithms
- 6 Empirical Analysis
- 7 Rule of thumb Estimation of Complexity
- 8 Summary

Recursion

- Recursion is fundamental tool in computer science.
- A recursive program (or function) is one that calls itself.
- It must have a **termination condition** defined.

Recursion

- Recursion is fundamental tool in computer science.
- A recursive program (or function) is one that calls itself.
- It must have a **termination condition** defined.
- Many interesting algorithms are simply expressed with a recursive approach.

Recursive Algorithm Example: Factorial

Factorial: $F(n) = n * (n - 1) * (n - 2) * (n - 3) * \dots * 3 * 2 * 1$

ALGORITHM $F(n)$

```
1: if  $n = 1$  then
2:   return 1
3: else
4:   return  $F(n - 1) * n$ 
5: end if
```

Time Efficiency of Recursive Algorithms

- ① Determine what is used to measure the **input size**.
- ② Identify the algorithm's **basic operation**

Time Efficiency of Recursive Algorithms

- ① Determine what is used to measure the **input size**.
- ② Identify the algorithm's **basic operation**
- ③ Setup a **recurrence relation** for $C(n)$, including termination condition(s).

Time Efficiency of Recursive Algorithms

- ① Determine what is used to measure the **input size**.
- ② Identify the algorithm's **basic operation**
- ③ Setup a **recurrence relation** for $C(n)$, including termination condition(s).
- ④ **Simplify** the recurrence relation using methods in Appendix B of textbook. (**Backward substitution**)
- ⑤ Determine an **equivalence class** $g(n)$ that **bounds** $t(n) = c_{op}C(n)$. Recall we generally want the tightest bound possible.

Recurrence Relations

A **recurrence relation** represents a sequence of terms, that results from a recursion on one or more of the previous terms.

The recurrence relation include the termination condition.

Recurrence Relations

A **recurrence relation** represents a sequence of terms, that results from a recursion on one or more of the previous terms.

The recurrence relation include the termination condition.

For example, the sequence for a factorial of n ,

$F(n) = n * (n - 1) * (n - 2) * (n - 3) * \dots * 3 * 2 * 1$, can be represented as the recurrence relation:

Recurrence Relations

A **recurrence relation** represents a sequence of terms, that results from a recursion on one or more of the previous terms.

The recurrence relation include the termination condition.

For example, the sequence for a factorial of n ,

$F(n) = n * (n - 1) * (n - 2) * (n - 3) * \dots * 3 * 2 * 1$, can be represented as the recurrence relation:

$$F(n) = F(n - 1) * n, F(1) = 1$$

Time Efficiency of Recursive Algorithms

- ① Determine what is used to measure the input size.
- ② Identify the algorithm's basic operation.
- ③ Setup a **recurrence relation** for $C(n)$, including termination condition(s).
- ④ Simplify the recurrence relation using methods in Appendix B of textbook. (Backward substitution)
- ⑤ Determine a function family $g(n)$ that bounds $t(n) = c_{op}C(n)$.
Recall we generally want the tightest bound possible.

Recursive Algorithm Example: Factorial

ALGORITHM $F(n)$

```
1: if  $n = 1$  then
2:   return 1
3: else
4:   return  $F(n - 1) * n$ 
5: end if
```

Recursive Algorithm Example: Factorial

ALGORITHM F(n)

```
1: if  $n = 1$  then
2:   return 1
3: else
4:   return F ( $n - 1$ ) *  $n$ 
5: end if
```

Basic operation: ?

Recursive Algorithm Example: Factorial

ALGORITHM F(n)

```
1: if  $n = 1$  then  
2:   return 1  
3: else  
4:   return F ( $n - 1$ ) *  $n$   
5: end if
```

Basic operation: ? multiplication Input size: ?

Recursive Algorithm Example: Factorial

ALGORITHM F(n)

```
1: if  $n = 1$  then  
2:   return 1  
3: else  
4:   return F ( $n - 1$ ) *  $n$   
5: end if
```

Basic operation: ? multiplication Input size: ? n ,

Recursive Algorithm Example: Factorial

ALGORITHM F(n)

```
1: if  $n = 1$  then  
2:   return 1  
3: else  
4:   return F ( $n - 1$ ) *  $n$   
5: end if
```

Basic operation: ? multiplication Input size: ? n,

Recurrence relation and conditions: The number of multiplications is $C(n) = C(n - 1) + 1$ for $n > 1$, and $C(1) = 0$.

Recursive Algorithm Example: Factorial

ALGORITHM $F(n)$

```
1: if  $n = 1$  then  
2:   return 1  
3: else  
4:   return  $F(n - 1) * n$   
5: end if
```

Basic operation: ? multiplication Input size: ? n ,

Recurrence relation and conditions: The number of multiplications is $C(n) = C(n - 1) + 1$ for $n > 1$, and $C(1) = 0$.

- “+1” is the number of multiplication operations at each recursive step.

Recursive Algorithm Example: Factorial

ALGORITHM $F(n)$

```
1: if  $n = 1$  then  
2:   return 1  
3: else  
4:   return  $F(n - 1) * n$   
5: end if
```

Basic operation: ? multiplication Input size: ? n ,

Recurrence relation and conditions: The number of multiplications is $C(n) = C(n - 1) + 1$ for $n > 1$, and $C(1) = 0$.

- “+1” is the number of multiplication operations at each recursive step.
- When $n = 1$, we have our termination/base case, where we stop the recursion. When we reach this base case, the number of multiplications is 0, hence $C(1) = 0$.

Time Efficiency of Recursive Algorithms

- ① Determine what is used to measure the input size.
- ② Identify the algorithm's basic operation.
- ③ Setup a recurrence relation for $C(n)$, including termination condition(s).
- ④ Simplify the recurrence relation using methods in Appendix B of textbook. (**Backward substitution**)
- ⑤ Determine a function family $g(n)$ that bounds $t(n) = c_{op}C(n)$.
Recall we generally want the tightest bound possible.

Backward Substitution: Factorial Example

Recurrence: $C(n) = C(n - 1) + 1$ for $n > 1$, and $C(1) = 0$

Aim of simplification and backward substitution: Convert $C(n) = C(n - 1) + 1$ to $C(n) = \text{function}(n)$, e.g., $C(n) = n + 1$.

Backward Substitution: Factorial Example

Recurrence: $C(n) = C(n - 1) + 1$ for $n > 1$, and $C(1) = 0$

Aim of simplification and backward substitution: Convert $C(n) = C(n - 1) + 1$ to $C(n) = \text{function}(n)$, e.g., $C(n) = n + 1$.

- ① Start with the recurrence relation: $C(n) = C(n - 1) + \dots$

Backward Substitution: Factorial Example

Recurrence: $C(n) = C(n - 1) + 1$ for $n > 1$, and $C(1) = 0$

Aim of simplification and backward substitution: Convert $C(n) = C(n - 1) + 1$ to $C(n) = \text{function}(n)$, e.g., $C(n) = n + 1$.

- ① Start with the recurrence relation: $C(n) = C(n - 1) + \dots$
- ② Substitute $C(n - 1)$ with its RHS ($C(n - 1) = C(n - 2) + \dots$) in original equation $C(n) = C(n - 2) + \dots$

Backward Substitution: Factorial Example

Recurrence: $C(n) = C(n - 1) + 1$ for $n > 1$, and $C(1) = 0$

Aim of simplification and backward substitution: Convert $C(n) = C(n - 1) + 1$ to $C(n) = \text{function}(n)$, e.g., $C(n) = n + 1$.

- ① Start with the recurrence relation: $C(n) = C(n - 1) + \dots$
- ② Substitute $C(n - 1)$ with its RHS ($C(n - 1) = C(n - 2) + \dots$) in original equation $C(n) = C(n - 2) + \dots$
- ③ Substitute $C(n - 2)$ with its RHS ($C(n - 2) = C(n - 3) + \dots$) in original equation $C(n) = C(n - 3) + \dots$

Backward Substitution: Factorial Example

Recurrence: $C(n) = C(n - 1) + 1$ for $n > 1$, and $C(1) = 0$

Aim of simplification and backward substitution: Convert $C(n) = C(n - 1) + 1$ to $C(n) = \text{function}(n)$, e.g., $C(n) = n + 1$.

- ① Start with the recurrence relation: $C(n) = C(n - 1) + \dots$
- ② Substitute $C(n - 1)$ with its RHS ($C(n - 1) = C(n - 2) + \dots$) in original equation $C(n) = C(n - 2) + \dots$
- ③ Substitute $C(n - 2)$ with its RHS ($C(n - 2) = C(n - 3) + \dots$) in original equation $C(n) = C(n - 3) + \dots$
- ④ Spot the pattern of $C(n)$ and introduce a variable to express this pattern.

Backward Substitution: Factorial Example

Recurrence: $C(n) = C(n - 1) + 1$ for $n > 1$, and $C(1) = 0$

Aim of simplification and backward substitution: Convert $C(n) = C(n - 1) + 1$ to $C(n) = \text{function}(n)$, e.g., $C(n) = n + 1$.

- ① Start with the recurrence relation: $C(n) = C(n - 1) + \dots$
- ② Substitute $C(n - 1)$ with its RHS ($C(n - 1) = C(n - 2) + \dots$) in original equation $C(n) = C(n - 2) + \dots$
- ③ Substitute $C(n - 2)$ with its RHS ($C(n - 2) = C(n - 3) + \dots$) in original equation $C(n) = C(n - 3) + \dots$
- ④ Spot the pattern of $C(n)$ and introduce a variable to express this pattern.
- ⑤ Determine when the value of this variable that relates $C(n) = C(1) + \dots$

Backward Substitution: Factorial Example

Recurrence: $C(n) = C(n - 1) + 1$ for $n > 1$, and $C(1) = 0$

Aim of simplification and backward substitution: Convert $C(n) = C(n - 1) + 1$ to $C(n) = \text{function}(n)$, e.g., $C(n) = n + 1$.

- ① Start with the recurrence relation: $C(n) = C(n - 1) + \dots$
- ② Substitute $C(n - 1)$ with its RHS ($C(n - 1) = C(n - 2) + \dots$) in original equation $C(n) = C(n - 2) + \dots$
- ③ Substitute $C(n - 2)$ with its RHS ($C(n - 2) = C(n - 3) + \dots$) in original equation $C(n) = C(n - 3) + \dots$
- ④ Spot the pattern of $C(n)$ and introduce a variable to express this pattern.
- ⑤ Determine when the value of this variable that relates $C(n) = C(1) + \dots$
- ⑥ Substitute the value of $C(1)$ and get $C(n)$ in terms of some expression of n .

Backward Substitution: Factorial Example

Recurrence: $C(n) = C(n - 1) + 1$ for $n > 1$, and $C(1) = 0$

Backward Substitution: Factorial Example

Recurrence: $C(n) = C(n - 1) + 1$ for $n > 1$, and $C(1) = 0$

1. $C(n) = C(n - 1) + 1$

Backward Substitution: Factorial Example

Recurrence: $C(n) = C(n - 1) + 1$ for $n > 1$, and $C(1) = 0$

1. $C(n) = C(n - 1) + 1$
2. Substitute $C(n - 1) = C(n - 2) + 1$ into original equation

Backward Substitution: Factorial Example

Recurrence: $C(n) = C(n - 1) + 1$ for $n > 1$, and $C(1) = 0$

1. $C(n) = C(n - 1) + 1$

2. Substitute $C(n - 1) = C(n - 2) + 1$ into original equation

3. $C(n) = [C(n - 2) + 1] + 1 = C(n - 2) + 2$

Backward Substitution: Factorial Example

Recurrence: $C(n) = C(n - 1) + 1$ for $n > 1$, and $C(1) = 0$

1. $C(n) = C(n - 1) + 1$

2. Substitute $C(n - 1) = C(n - 2) + 1$ into original equation

3. $C(n) = [C(n - 2) + 1] + 1 = C(n - 2) + 2$

4. Substitute $C(n - 2) = C(n - 3) + 1$ into original equation

Backward Substitution: Factorial Example

Recurrence: $C(n) = C(n - 1) + 1$ for $n > 1$, and $C(1) = 0$

1. $C(n) = C(n - 1) + 1$
2. Substitute $C(n - 1) = C(n - 2) + 1$ into original equation
3. $C(n) = [C(n - 2) + 1] + 1 = C(n - 2) + 2$
4. Substitute $C(n - 2) = C(n - 3) + 1$ into original equation
5. $C(n) = [C(n - 3) + 1] + 2 = C(n - 3) + 3$

Backward Substitution: Factorial Example

Recurrence: $C(n) = C(n - 1) + 1$ for $n > 1$, and $C(1) = 0$

1. $C(n) = C(n - 1) + 1$
2. Substitute $C(n - 1) = C(n - 2) + 1$ into original equation
3. $C(n) = [C(n - 2) + 1] + 1 = C(n - 2) + 2$
4. Substitute $C(n - 2) = C(n - 3) + 1$ into original equation
5. $C(n) = [C(n - 3) + 1] + 2 = C(n - 3) + 3$
6. We see the pattern $C(n) = C(n - i) + i$ emerge, where $1 \leq i \leq n$.

Backward Substitution: Factorial Example

Recurrence: $C(n) = C(n - 1) + 1$ for $n > 1$, and $C(1) = 0$

1. $C(n) = C(n - 1) + 1$
2. Substitute $C(n - 1) = C(n - 2) + 1$ into original equation
3. $C(n) = [C(n - 2) + 1] + 1 = C(n - 2) + 2$
4. Substitute $C(n - 2) = C(n - 3) + 1$ into original equation
5. $C(n) = [C(n - 3) + 1] + 2 = C(n - 3) + 3$
6. We see the pattern $C(n) = C(n - i) + i$ emerge, where $1 \leq i \leq n$.
7. Now, we know $C(1)$ and want to determine when $C(n - i) = C(1)$, or when $n - i = 1$. This value is $i = n - 1$.

Backward Substitution: Factorial Example

Recurrence: $C(n) = C(n - 1) + 1$ for $n > 1$, and $C(1) = 0$

1. $C(n) = C(n - 1) + 1$
2. Substitute $C(n - 1) = C(n - 2) + 1$ into original equation
3. $C(n) = [C(n - 2) + 1] + 1 = C(n - 2) + 2$
4. Substitute $C(n - 2) = C(n - 3) + 1$ into original equation
5. $C(n) = [C(n - 3) + 1] + 2 = C(n - 3) + 3$
6. We see the pattern $C(n) = C(n - i) + i$ emerge, where $1 \leq i \leq n$.
7. Now, we know $C(1)$ and want to determine when $C(n - i) = C(1)$, or when $n - i = 1$. This value is $i = n - 1$.
$$C(n) = C(n - (n - 1)) + n - 1 = C(1) + n - 1 = 0 + n - 1 = n - 1.$$

Backward Substitution: Factorial Example

Recurrence: $C(n) = C(n - 1) + 1$ for $n > 1$, and $C(1) = 0$

1. $C(n) = C(n - 1) + 1$
2. Substitute $C(n - 1) = C(n - 2) + 1$ into original equation
3. $C(n) = [C(n - 2) + 1] + 1 = C(n - 2) + 2$
4. Substitute $C(n - 2) = C(n - 3) + 1$ into original equation
5. $C(n) = [C(n - 3) + 1] + 2 = C(n - 3) + 3$
6. We see the pattern $C(n) = C(n - i) + i$ emerge, where $1 \leq i \leq n$.
7. Now, we know $C(1)$ and want to determine when $C(n - i) = C(1)$, or when $n - i = 1$. This value is $i = n - 1$.
$$C(n) = C(n - (n - 1)) + n - 1 = C(1) + n - 1 = 0 + n - 1 = n - 1.$$

Hence $t(n) = c_{op} \cdot (n - 1) \in O(n)$.

Summary of these parts

- (non-recursive algorithm) Discussed how to write down expression for $C(n)$ (number of basic operations executed in algorithm).
- Discussed how to simplify this for non-recursive algorithm.
- (recursive algorithm) Discussed the idea of recurrence and how to write $C(n)$.
- Discussed how to simplify via backward substitution.

Overview

- 1 Overview
- 2 Fundamentals
- 3 Asymptotic Complexity
- 4 Analysing Non-recursive Algorithms
- 5 Analysing Recursive Algorithms
- 6 Empirical Analysis
- 7 Rule of thumb Estimation of Complexity

Summary

Theory vs Practice

Formal Analysis (theoretical):

- **Pro:** No interference from implementation /hardware details.
- **Con:** Hides constant factors. Requires a different mindset.

Empirical Analysis (measure):

- **Pro:** Discovers the true impact of constant factors.
- **Con:** May be running on the “wrong” inputs.

“In theory, theory and practice are the same. In practice, they are not.”

- Albert Einstein

Empirical Analysis

- The complexity of an algorithm gives an estimate of the running time (we discussed this in detail).
- Measuring the *actual* time an implementation takes is also important, especially when comparing two algorithms with the same time complexity.
- When in doubt, **measure!**

Empirical Analysis of Algorithm Time Efficiency

When designing experiments:

- ① Understand the experiment's purpose.

Empirical Analysis of Algorithm Time Efficiency

When designing experiments:

- ① Understand the experiment's purpose.
- ② Decide on the efficiency metric M to be measured and the measurement unit (an operation's count vs. a time unit).

Empirical Analysis of Algorithm Time Efficiency

When designing experiments:

- ① Understand the experiment's purpose.
- ② Decide on the efficiency metric M to be measured and the measurement unit (an operation's count vs. a time unit).
- ③ Decide on **characteristics** of the input sample (its range, size, and so on).

Empirical Analysis of Algorithm Time Efficiency

When designing experiments:

- ① Understand the experiment's purpose.
- ② Decide on the efficiency metric M to be measured and the measurement unit (an operation's count vs. a time unit).
- ③ Decide on **characteristics** of the input sample (its range, size, and so on).
- ④ Prepare a program implementing the algorithm (or algorithms) for the experimentation.

Empirical Analysis of Algorithm Time Efficiency

When designing experiments:

- ① Understand the experiment's purpose.
- ② Decide on the efficiency metric M to be measured and the measurement unit (an operation's count vs. a time unit).
- ③ Decide on **characteristics** of the input sample (its range, size, and so on).
- ④ Prepare a program implementing the algorithm (or algorithms) for the experimentation.
- ⑤ Generate a sample of inputs.

Empirical Analysis of Algorithm Time Efficiency

When designing experiments:

- ① Understand the experiment's purpose.
- ② Decide on the efficiency metric M to be measured and the measurement unit (an operation's count vs. a time unit).
- ③ Decide on **characteristics** of the input sample (its range, size, and so on).
- ④ Prepare a program implementing the algorithm (or algorithms) for the experimentation.
- ⑤ Generate a sample of inputs.
- ⑥ Run the algorithm (or algorithms) on the sample's inputs and record the data observed.

Empirical Analysis of Algorithm Time Efficiency

When designing experiments:

- ① Understand the experiment's purpose.
- ② Decide on the efficiency metric M to be measured and the measurement unit (an operation's count vs. a time unit).
- ③ Decide on **characteristics** of the input sample (its range, size, and so on).
- ④ Prepare a program implementing the algorithm (or algorithms) for the experimentation.
- ⑤ Generate a sample of inputs.
- ⑥ Run the algorithm (or algorithms) on the sample's inputs and record the data observed.
- ⑦ Analyse the data obtained.

Benchmarking Algorithms: Significance of input

The **input** and **algorithm** can be significant determinant of performance.

Benchmarking Algorithms: Significance of input

The **input** and **algorithm** can be significant determinant of performance.

Theoretical: Merge-sort and Quick-sort have $O(n \log(n))$ complexity, while Selection-sort is $O(n^2)$.

Benchmarking Algorithms: Significance of input

The **input** and **algorithm** can be significant determinant of performance.

Theoretical: Merge-sort and Quick-sort have $O(n \log(n))$ complexity, while Selection-sort is $O(n^2)$.

Empirical: Running Times (in seconds) for different sorting algorithms on a **randomised list**:

List size	500	2,500	10,000
Merge-Sort	0.8	8.1	39.8
Quick-Sort	0.3	1.3	5.3
Selection-Sort	1.5	35.0	534.7

Benchmarking Algorithms: Significance of input

The **input** and **algorithm** can be significant determinant of performance.

Theoretical: Merge-sort and Quick-sort have $O(n \log(n))$ complexity, while Selection-sort is $O(n^2)$.

Empirical: Running Times (in seconds) for different sorting algorithms on a **randomised list**:

List size	500	2,500	10,000
Merge-Sort	0.8	8.1	39.8
Quick-Sort	0.3	1.3	5.3
Selection-Sort	1.5	35.0	534.7

For ordered lists for $N = 2,500$:

	Random	In Order	In Reverse Order
Merge-Sort	8.1	7.8	7.9
Quick-Sort	1.3	35.1	37.1
Selection-Sort	35.0	34.4	35.3

Comparing Search Algorithms: Significance of input size

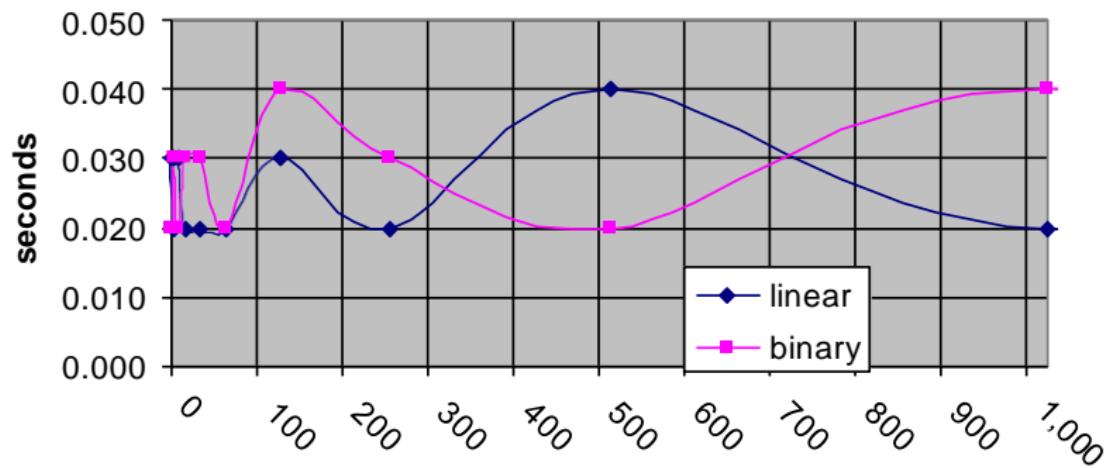
The **input size** can be significant determinant of performance.

Comparing Search Algorithms: Significance of input size

The **input size** can be significant determinant of performance.

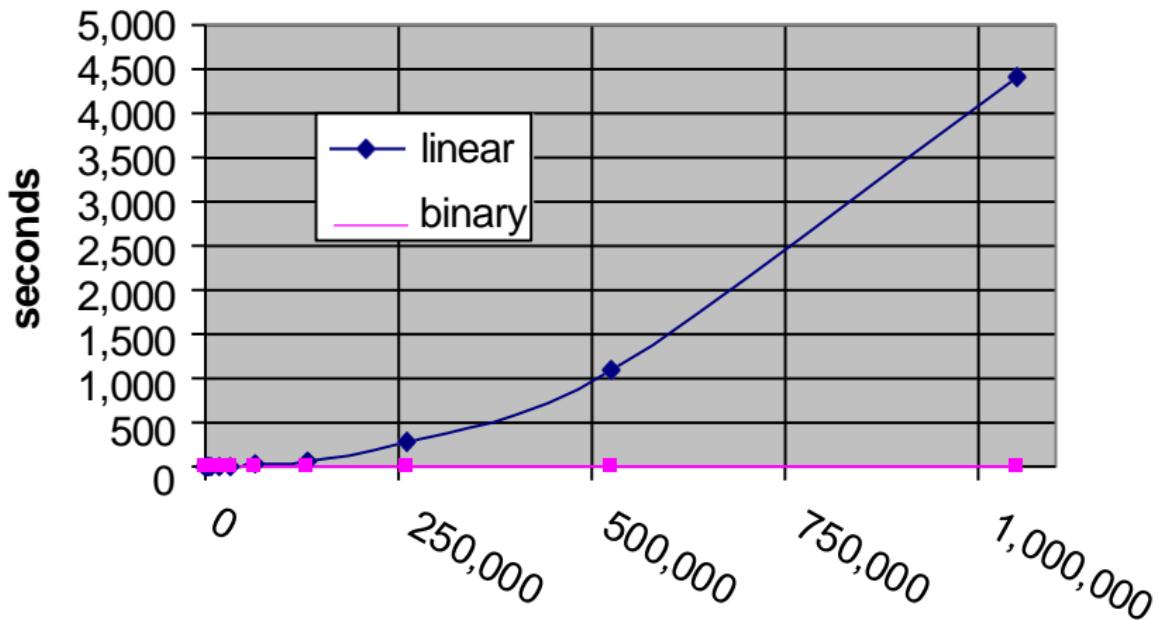
Theoretical: Linear search have $O(n)$ complexity while binary search have $O(\log(n))$ complexity on a sorted list.

linear vs binary search



Comparing Search Algorithms continued

linear vs binary search



Overview

- 1 Overview
- 2 Fundamentals
- 3 Asymptotic Complexity
- 4 Analysing Non-recursive Algorithms
- 5 Analysing Recursive Algorithms
- 6 Empirical Analysis
- 7 Rule of thumb Estimation of Complexity
- 8 Summary

Approximate Estimate of Complexity

- We can sometimes make an educated guess at the complexity, then use theoretical and/or empirical analysis to confirm.

Approximate Estimate of Complexity

- We can sometimes make an educated guess at the complexity, then use theoretical and/or empirical analysis to confirm.
- We discuss a few such guidelines here, but will discuss more as we study further algorithms and data structures.

Approximate Estimate of Complexity

- We can sometimes make an educated guess at the complexity, then use theoretical and/or empirical analysis to confirm.
- We discuss a few such guidelines here, but will discuss more as we study further algorithms and data structures.
- Requires experience.

Constant time $O(1)$

- Typically algorithms or data structures whose operations does not depend on input size.

Constant time $O(1)$

- Typically algorithms or data structures whose operations does not depend on input size.
- E.g., Access to an element in an array.

Linear time $O(n)$

- Typically algorithms or data structures whose operations that needs to evaluate most or all of the elements of a problem (input size).

Linear time $O(n)$

- Typically algorithms or data structures whose operations that needs to evaluate most or all of the elements of a problem (input size).
- E.g., Scan through an array to search for an element.

Linear time $O(n)$

- Typically algorithms or data structures whose operations that needs to evaluate most or all of the elements of a problem (input size).
- E.g., Scan through an array to search for an element.
- E.g., Copy elements from a linked list to another.

Linear time $O(n)$

- Typically algorithms or data structures whose operations that needs to evaluate most or all of the elements of a problem (input size).
- E.g., Scan through an array to search for an element.
- E.g., Copy elements from a linked list to another.
- In terms of pseudo code, it usually involves a single for loop.

Quadratic time $O(n^2)$

- Typically algorithms or data structures whose operations that needs to process/evaluate pairs of elements of a problem.

Quadratic time $O(n^2)$

- Typically algorithms or data structures whose operations that needs to process/evaluate pairs of elements of a problem.
- E.g., Compare each element in an array to all its others to find duplicates (naive + unordered).

Quadratic time $O(n^2)$

- Typically algorithms or data structures whose operations that needs to process/evaluate pairs of elements of a problem.
- E.g., Compare each element in an array to all its others to find duplicates (naive + unordered).
- E.g., Compute all pairwise distance among a set of points.

Quadratic time $O(n^2)$

- Typically algorithms or data structures whose operations that needs to process/evaluate pairs of elements of a problem.
- E.g., Compare each element in an array to all its others to find duplicates (naive + unordered).
- E.g., Compute all pairwise distance among a set of points.
- In terms of pseudo code, it usually involves two nested for loops.

Overview

- 1 Overview
- 2 Fundamentals
- 3 Asymptotic Complexity
- 4 Analysing Non-recursive Algorithms
- 5 Analysing Recursive Algorithms
- 6 Empirical Analysis
- 7 Rule of thumb Estimation of Complexity
- 8 Summary

Summary

- We have covered the core theoretical framework for algorithmic analysis which will be used for the rest of the semester.
 - Problem input size, basic operation, time complexity, asymptotic complexity, worst/best/average cases.
 - Analysis of Non-recursive, recursive algorithms.
 - Empirical analysis.
 - Approximate analysis.

Next week, we look at first of the algorithmic paradigms in this course,
brute force algorithms.

Outline

- ① Overview
- ② Sorting
- ③ Exhaustive Search
- ④ Graph Search
- ⑤ Case Study
- ⑥ Summary

Overview

① [Overview](#)

② [Sorting](#)

③ [Exhaustive Search](#)

④ [Graph Search](#)

⑤ Case Study

⑥ Summary

Levitin - The design and analysis of algorithms

This week we will be covering the material from Chapter 3.

Learning Outcomes:

- Understand the *Brute Force* algorithmic approach.
- Understand and apply:
 - Sorting - selection and bubble sort.
 - Exhaustive search - knapsack.
 - Graph search - DFS, BFS.
- Examine a case study of using a brute force approach to solve a problem.

Brute Force

Brute force is a straightforward approach to solving a problem, usually directly based on the problem statement and definitions of the concepts involved, and can involve enumerating all solutions and selecting the best one.

Examples:

- ① Computing a^n (multiple 'a' n times)
- ② Searching for a key of a given value in an unsorted list.

Overview

① [Overview](#)

② [Sorting](#)

③ [Exhaustive Search](#)

④ [Graph Search](#)

⑤ Case Study

⑥ Summary

Sorting

Examples:

- Telephone book - Sorted by surname.
- Height in class - Tallest to shortest.

Sorting

Examples:

- Telephone book - Sorted by surname.
- Height in class - Tallest to shortest.

Why?

- Important to build efficient searching algorithms and data structures, data compression.
- Heavily studied problem in computer science, with several widely celebrated algorithms.

Sorting

Examples:

- Telephone book - Sorted by surname.
- Height in class - Tallest to shortest.

Why?

- Important to build efficient searching algorithms and data structures, data compression.
- Heavily studied problem in computer science, with several widely celebrated algorithms.

Sorting Problem

Given a **sequence** of n elements $x_1, x_2, \dots, x_n \in S$, **rearrange** the elements according to some **ordering criteria**.

Brute Force Sorting: Selection Sort

Selection sort is a brute force solution to the sorting problem.

Brute Force Sorting: Selection Sort

Selection sort is a brute force solution to the sorting problem.

- ① Scan all n elements of the array to find the **smallest** element, and *swap* it with the *first element*.
- ② Starting with the second element, scan the remaining $n - 1$ elements to find the **smallest** element and *swap* it with the element in the second position.
- ③ Generally, on pass i ($0 \leq i \leq n - 2$), find the **smallest** element in $A[i \dots n - 1]$ and *swap* it with $A[i]$.

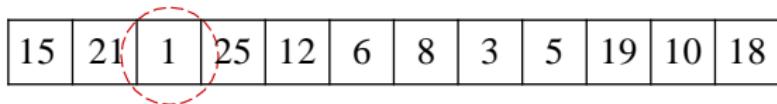
Selection Sort Example

15	21	1	25	12	6	8	3	5	19	10	18
----	----	---	----	----	---	---	---	---	----	----	----

COMPARES

0

Selection Sort Example

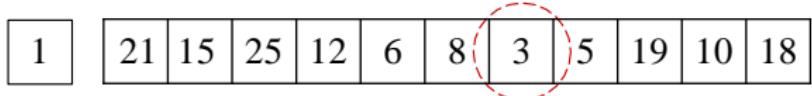


COMPARES

0

 +11

Selection Sort Example



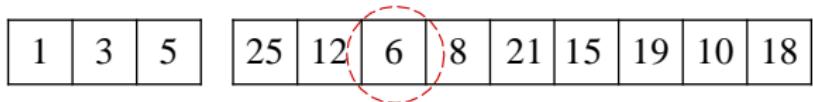
COMPARES 11 +10

Selection Sort Example

1	3	15	25	12	6	8	21	5	19	10	18
---	---	----	----	----	---	---	----	---	----	----	----

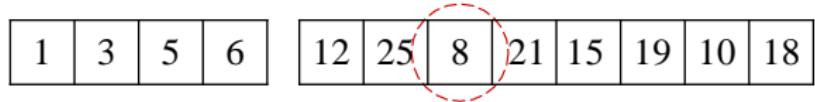
COMPARES 21 +9

Selection Sort Example



COMPARES 30 +8

Selection Sort Example



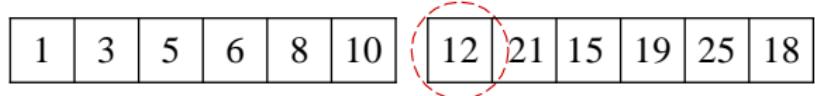
COMPARES 38 +7

Selection Sort Example

1	3	5	6	8	25	12	21	15	19	10	18
---	---	---	---	---	----	----	----	----	----	----	----

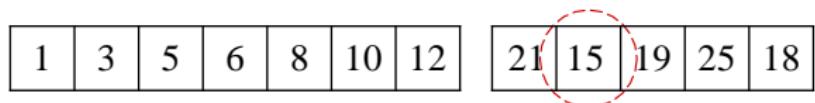
COMPARES 45 +6

Selection Sort Example



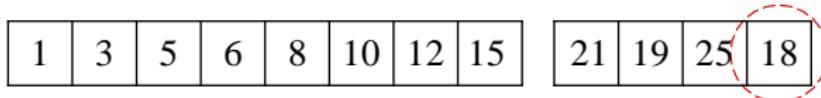
COMPARES 51 +5

Selection Sort Example



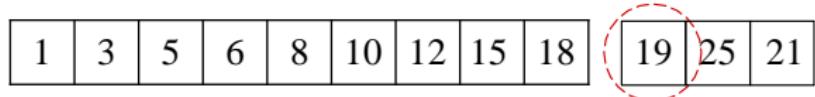
COMPARES 56 +4

Selection Sort Example



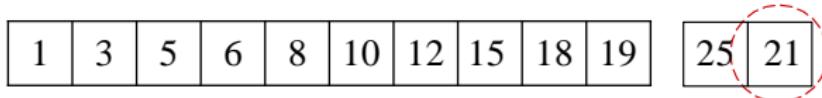
COMPARES 60 +3

Selection Sort Example



COMPARES 63 +2

Selection Sort Example



COMPARES 65 +1

Selection Sort Example

1	3	5	6	8	10	12	15	18	19	21	25
---	---	---	---	---	----	----	----	----	----	----	----

COMPARES

66

Selection Sort

```
ALGORITHM SelectionSort ( $A[0 \dots n - 1]$ )
/* Order an array using a brute-force selection sort. */
/* INPUT : An array  $A[0 \dots n - 1]$  of orderable elements. */
/* OUTPUT : An array  $A[0 \dots n - 1]$  sorted in ascending order. */

1: for  $i = 0$  to  $n - 2$  do
2:    $min = i$ 
3:   for  $j = i + 1$  to  $n - 1$  do
4:     if  $A[j] < A[min]$  then
5:        $min = j$ 
6:     end if
7:   end for
8:   swap  $A[i]$  and  $A[min]$ 
9: end for
```

Selection Sort - Complexity

Selection Sort Complexity?

$$C(n) =$$

Selection Sort - Complexity

Selection Sort Complexity?

$$C(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1 =$$

Selection Sort - Complexity

Selection Sort Complexity?

$$C(n) = \sum_{i=0}^{\frac{n}{2}-1} \sum_{j=i+1}^{\frac{n}{2}} 1 = \sum_{i=0}^{\frac{n}{2}-1} (n - 1 - i) =$$

Selection Sort - Complexity

Selection Sort Complexity?

$$C(n) = \sum_{i=0}^{\frac{n-1}{2}} \sum_{j=i+1}^{\frac{n-1}{2}} 1 = \sum_{i=0}^{\frac{n-1}{2}} (n-1-i) = \frac{(n-1)n}{2}$$

Selection Sort - Complexity

Selection Sort Complexity?

$$C(n) = \sum_{i=0}^{\frac{n}{2}} \sum_{j=i+1}^{\frac{n}{2}-1} 1 = \sum_{i=0}^{\frac{n}{2}} (n - 1 - i) = \frac{(n - 1)n}{2} \in O(n^2)$$

Selection Sort - Complexity

Selection Sort Complexity?

$$C(n) = \sum_{i=0}^{\frac{n}{2}} \sum_{j=i+1}^{\frac{n}{2}-1} 1 = \sum_{i=0}^{\frac{n}{2}} (n-1-i) = \frac{(n-1)n}{2} \in O(n^2)$$

- Needs around $n^2/2$ comparisons and at most $n - 1$ exchanges.
- The running time is insensitive to the input, so the best, average, and worst case are essentially the same (Why?).

Selection Sort - Complexity

Selection Sort Complexity?

$$C(n) = \sum_{i=0}^{\frac{n-2}{2}} \sum_{j=i+1}^{\frac{n-1}{2}} 1 = \sum_{i=0}^{\frac{n-2}{2}} (n-1-i) = \frac{(n-1)n}{2} \in O(n^2)$$

- Needs around $n^2/2$ comparisons and at most $n - 1$ exchanges.
- The running time is insensitive to the input, so the best, average, and worst case are essentially the same (Why?).

Why use it?

- Selection sort only makes $O(n)$ writes but $O(n^2)$ reads.
- When writes (to array) are much more expensive than reads, selection sort may have an advantage, e.g., flash memory.
- Also, for small arrays, selection sort is relatively efficient and simple to implement.

Stable Sorting

- **Definition:** A sorting method is said to be *stable* if it preserves the relative order of duplicate keys in the file.

Before Sorting

1 Adams
2 Black
4 Brown
2 Jackson
4 Jones
1 Smith
4 Thompson
2 Washington
3 White
3 Wilson

After Sorting

1 Adams
1 Smith
2 Black
2 Jackson
2 Washington
3 White
3 Wilson
4 Brown
4 Jones
4 Thompson

Not all sorting methods are stable.

Is Selection Sort Stable?

Question: Is selection sort stable?

Consider the following examples and apply selection sort on them:

5,5,3,2

Brute Force Sorting: Bubble Sort

A **bubble sort** iteratively **compares adjacent** items in a list and **exchanges** them if they are out of order.

Brute Force Sorting: Bubble Sort

A **bubble sort** iteratively **compares adjacent** items in a list and **exchanges** them if they are out of order.

Motivation:

- One of the classic (and elementary) sorting algorithms, originally designed and efficient for tape disks, but with random access memory, it doesn't have much use these days.
- But insightful to study it and to understand why other sorting algorithms are superior in one or more aspects.
- It is simple to code.

Bubble Sort

- ① First iteration, compare each adjacent pair of elements and swap them if they are out of order. Eventually **largest** element gets propagated to the end.
- ② Second iteration, repeat the process, but only from first to 2nd last element (last element is in its correct position). Eventually **second largest** element is at the 2nd last element.
- ③ Repeat until all elements are sorted.

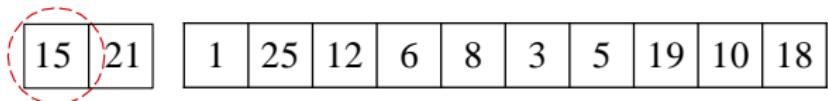
Bubble Sort Example - Round 1

15	21	1	25	12	6	8	3	5	19	10	18
----	----	---	----	----	---	---	---	---	----	----	----

COMPARES

0

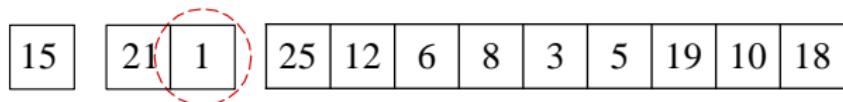
Bubble Sort Example - Round 1



COMPARES

1

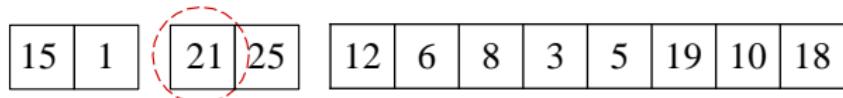
Bubble Sort Example - Round 1



COMPARES

2

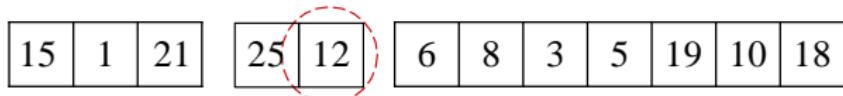
Bubble Sort Example - Round 1



COMPARES

3

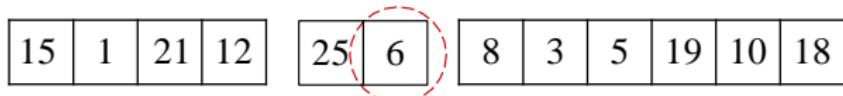
Bubble Sort Example - Round 1



COMPARES

4

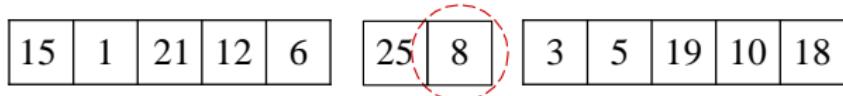
Bubble Sort Example - Round 1



COMPARES

5

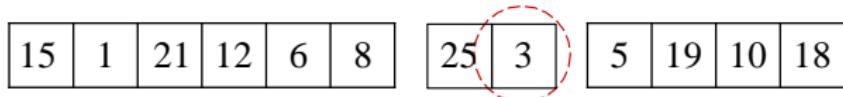
Bubble Sort Example - Round 1



COMPARES

6

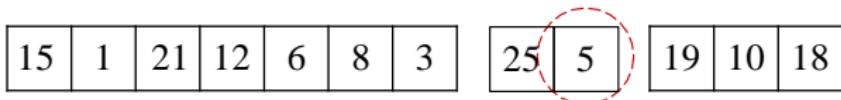
Bubble Sort Example - Round 1



COMPARES

7

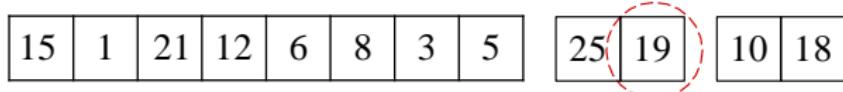
Bubble Sort Example - Round 1



COMPARES

8

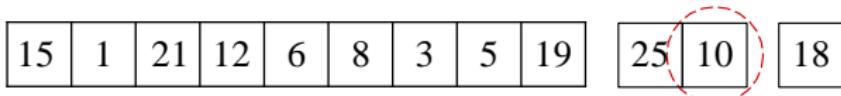
Bubble Sort Example - Round 1



COMPARES

9

Bubble Sort Example - Round 1



COMPARES

10

Bubble Sort Example - Round 1

15	1	21	12	6	8	3	5	19	10	25	18
----	---	----	----	---	---	---	---	----	----	----	----

COMPARES

11

Bubble Sort Example - Round 1

15	1	21	12	6	8	3	5	19	10	18	25
----	---	----	----	---	---	---	---	----	----	----	----



COMPARES

11

Bubble Sort Example - Round 2

1	15	12	6	8	3	5	19	10	18	21	25
---	----	----	---	---	---	---	----	----	----	----	----



COMPARES

21

Bubble Sort Example - Round 3

1	12	6	8	3	5	15	10	18	19	21	25
---	----	---	---	---	---	----	----	----	----	----	----



COMPARES

30

Bubble Sort Example - Round 4

1	6	8	3	5	12	10	15	18	19	21	25
---	---	---	---	---	----	----	----	----	----	----	----

COMPARES

38

Bubble Sort Example - Round 5

1	6	3	5	8	10	12	15	18	19	21	25
---	---	---	---	---	----	----	----	----	----	----	----



COMPARES

45

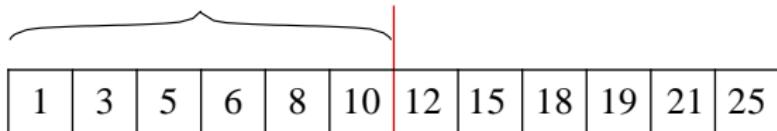
Bubble Sort Example - Round 6

1	3	5	6	8	10	12	15	18	19	21	25
---	---	---	---	---	----	----	----	----	----	----	----

COMPARES

51

Bubble Sort Example - Sorted?



COMPARES

51

Bubble Sort

```
ALGORITHM BubbleSort ( $A[0 \dots n - 1]$ )
/* Order an array using a bubble sort. */
/* INPUT : An array  $A[0 \dots n - 1]$  of orderable elements. */
/* OUTPUT : An array  $A[0 \dots n - 1]$  sorted in ascending order. */

1: for  $i = 0$  to  $n - 2$  do
2:   for  $j = 0$  to  $n - 2 - i$  do
3:     if  $A[j + 1] < A[j]$  then
4:       swap  $A[j]$  and  $A[j + 1]$ 
5:     end if
6:   end for
7: end for
```

Bubble Sort - Complexity

Bubble Sort Complexity?

$$C(n) =$$

Bubble Sort - Complexity

Bubble Sort Complexity?

$$C(n) = \sum_{i=0}^{n-2} n \sum_{j=0}^{2-i} 1$$

Bubble Sort - Complexity

Bubble Sort Complexity?

$$C(n) = \sum_{i=0}^{n-2} \sum_{j=0}^{n-2-i} 1 = \sum_{i=0}^{n-2} [(n-2-i) - 0 + 1] = \sum_{i=0}^{n-2} (n-1-i)$$

Bubble Sort - Complexity

Bubble Sort Complexity?

$$\begin{aligned}C(n) &= \sum_{i=0}^{n-2} \sum_{j=0}^{n-2-i} 1 = \sum_{i=0}^{n-2} [(n-2-i) - 0 + 1] = \sum_{i=0}^{n-2} (n-1-i) \\&= \frac{(n-1)n}{2}\end{aligned}$$

Bubble Sort - Complexity

Bubble Sort Complexity?

$$\begin{aligned} C(n) &= \sum_{i=0}^{n-2} \sum_{j=0}^{n-2-i} 1 = \sum_{i=0}^{n-2} [(n-2-i) - 0 + 1] = \sum_{i=0}^{n-2} (n-1-i) \\ &= \frac{(n-1)n}{2} \in O(n^2) \end{aligned}$$

- **Best case:** if original file is already sorted, about $n^2/2$ comparisons & 0 exchanges - $O(n^2)$.
- **Worst case:** if original file is sorted in reverse order, about $n^2/2$ comparisons & $n^2/2$ exchanges - $O(n^2)$.
- **Average case:** if original file is in random order, about $n^2/2$ comparisons & less than $n^2/2$ exchanges - $O(n^2)$.

Bubble Sort - Complexity

Bubble Sort Complexity?

$$\begin{aligned} C(n) &= \sum_{i=0}^{n-2} \sum_{j=0}^{n-2-i} 1 = \sum_{i=0}^{n-2} [(n-2-i) - 0 + 1] = \sum_{i=0}^{n-2} (n-1-i) \\ &= \frac{(n-1)n}{2} \in O(n^2) \end{aligned}$$

- **Best case:** if original file is already sorted, about $n^2/2$ comparisons & 0 exchanges - $O(n^2)$.
- **Worst case:** if original file is sorted in reverse order, about $n^2/2$ comparisons & $n^2/2$ exchanges - $O(n^2)$.
- **Average case:** if original file is in random order, about $n^2/2$ comparisons & less than $n^2/2$ exchanges - $O(n^2)$.

Is bubble sort **stable**?

Early-Termination Bubble Sort

- This modification attempts to **reduce redundant iterations**, by checking if any exchanges takes place in each pass. If there were **no exchanges** in the current iteration, the sorting is **stopped** after the current iteration.

Early-Termination Bubble Sort

- This modification attempts to **reduce redundant iterations**, by checking if any exchanges takes place in each pass. If there were **no exchanges** in the current iteration, the sorting is **stopped** after the current iteration.
- Why does this work?

Early-Termination Bubble Sort

- This modification attempts to **reduce redundant iterations**, by checking if any exchanges takes place in each pass. If there were **no exchanges** in the current iteration, the sorting is **stopped** after the current iteration.
- Why does this work?
- **Best case** - when the original file is already sorted, only one pass is needed, $n - 1$ comparisons, 0 exchanges - $O(n)$.
- **Worst case** - No improvement over the original implementation - $O(n^2)$.
- **Average case** - Depending on the data set, few iterations can be eliminated at the end of the sort. Therefore, the number of passes is less than $n - 1$, and hence cost is lower than the original implementation. The complexity is still likely to be $O(n^2)$.

Overview

1 [Overview](#)

2 [Sorting](#)

3 [Exhaustive Search](#)

4 [Graph Search](#)

5 Case Study

6 Summary

Exhaustive Search

A brute force approach involving *enumerating/generating all* possible solutions, then *selecting* the “best” one.

Exhaustive Search

A brute force approach involving *enumerating/generating all* possible solutions, then *selecting* the “best” one.

Method:

- Generate a list of **all** potential solutions to the problem in a systematic manner.
- Evaluate potential solutions **one by one**, disqualifying infeasible ones, and keeping track of the best one found so far.
- When all items have been evaluated, announce the **best solution(s)** found.

Exhaustive Search

A brute force approach involving *enumerating/generating all* possible solutions, then *selecting* the “best” one.

Method:

- Generate a list of **all** potential solutions to the problem in a systematic manner.
- Evaluate potential solutions **one by one**, disqualifying infeasible ones, and keeping track of the best one found so far.
- When all items have been evaluated, announce the **best solution(s)** found.

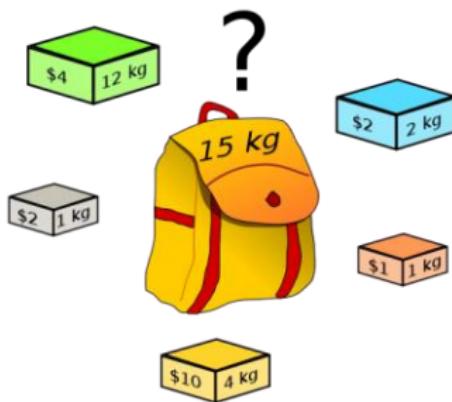
Typically applied to combinatorial problems, and insightful to study brute force solutions to them, as some problems can only be solved optimally by exhaustive search.

Knapsack Problem

Knapsack Problem

Given n items of known weights w_1, \dots, w_n and the values v_1, \dots, v_n and a knapsack of capacity W , find the most valuable subset of the items that fit into the knapsack.^a

^a<http://en.wikipedia.org/wiki/File:Knapsack.svg>



Applications of Knapsack Problem



Brute Force Knapsack Algorithm

Algorithm:

- ① Consider all subsets of the set of n items.

Brute Force Knapsack Algorithm

Algorithm:

- ① Consider all subsets of the set of n items.
- ② Compute the total weight of each subset in order to identify feasible subsets (the ones with the total not exceeding the knapsack's capacity).

Brute Force Knapsack Algorithm

Algorithm:

- ① Consider all subsets of the set of n items.
- ② Compute the total weight of each subset in order to identify feasible subsets (the ones with the total not exceeding the knapsack's capacity).
- ③ Find the subset of the largest value among them.

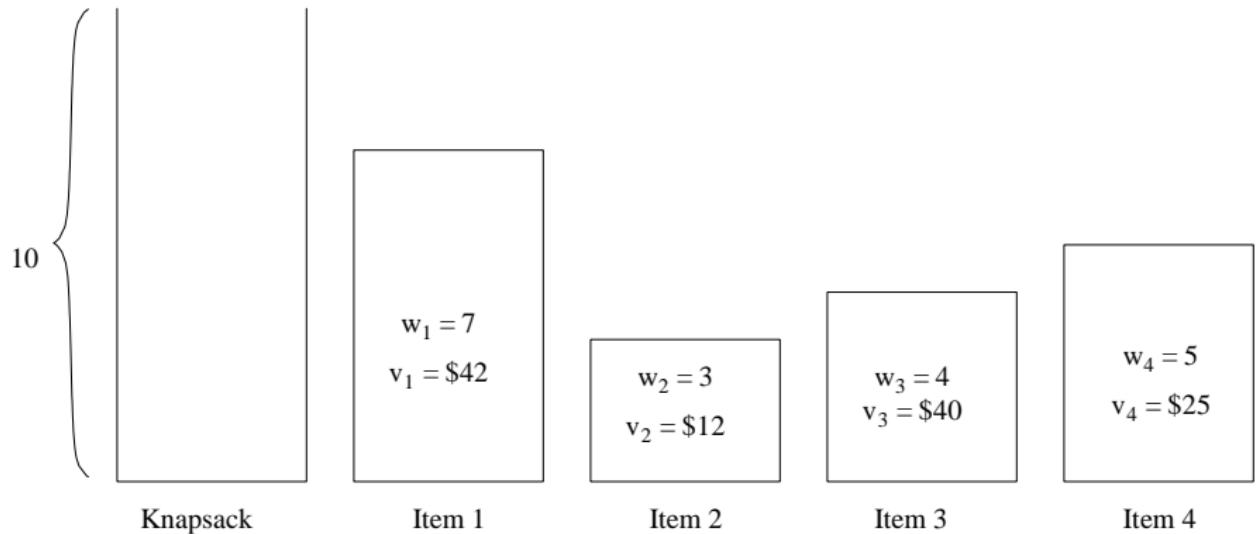
Brute Force Knapsack Algorithm

Algorithm:

- ① Consider all subsets of the set of n items.
- ② Compute the total weight of each subset in order to identify feasible subsets (the ones with the total not exceeding the knapsack's capacity).
- ③ Find the subset of the largest value among them.

Complexity: Since the number of subsets of an n -element set is 2^n , an exhaustive search produces an $O(2^n)$ algorithm.

Knapsack Problem



Knapsack Problem

Subset	Total Weight	Total Value
	0	\$0
{ 1 }	7	\$42
{ 2 }	3	\$12
{ 3 }	4	\$40
{ 4 }	5	\$25
{ 1, 2 }	10	\$36
{ 1, 3 }	11	Not Possible
{ 1, 4 }	12	Not Possible
{ 2, 3 }	7	\$52
{ 2, 4 }	8	\$37
{ 3, 4 }	9	\$65
{ 1, 2, 3 }	14	Not Possible
{ 1, 2, 4 }	15	Not Possible
{ 1, 3, 4 }	16	Not Possible
{ 2, 3, 4 }	12	Not Possible
{ 1, 2, 3, 4 }	19	Not Possible

Overview

1 [Overview](#)

2 [Sorting](#)

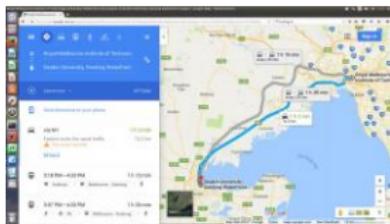
3 [Exhaustive Search](#)

4 [Graph Search](#)

5 Case Study

6 Summary

Searching in Graphs



(a) How to find the shortest path in a road network?

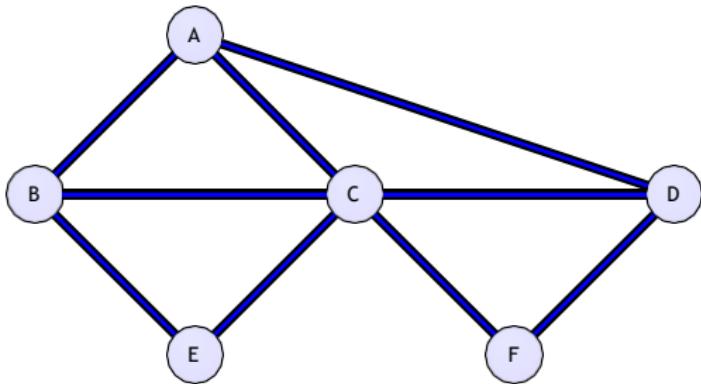


(b) How to find a path through a maze?



(c) How to determine if a power network is connected?

Graph example



Depth-First Search - Overview

Depth-First Search (DFS) - Traversal:

Depth-First Search - Overview

Depth-First Search (DFS) - Traversal:

- 1 Choose an arbitrary vertex and mark it visited.

Depth-First Search - Overview

Depth-First Search (DFS) - Traversal:

- ① Choose an arbitrary vertex and mark it visited.
- ② From the current vertex, proceed to an **unvisited, adjacent** vertex and mark it visited.

Depth-First Search - Overview

Depth-First Search (DFS) - Traversal:

- ① Choose an arbitrary vertex and mark it visited.
- ② From the current vertex, proceed to an **unvisited, adjacent** vertex and mark it visited.
- ③ Repeat 2nd step until a vertex is reached which has no adjacent, unvisited vertices (dead-end).

Depth-First Search - Overview

Depth-First Search (DFS) - Traversal:

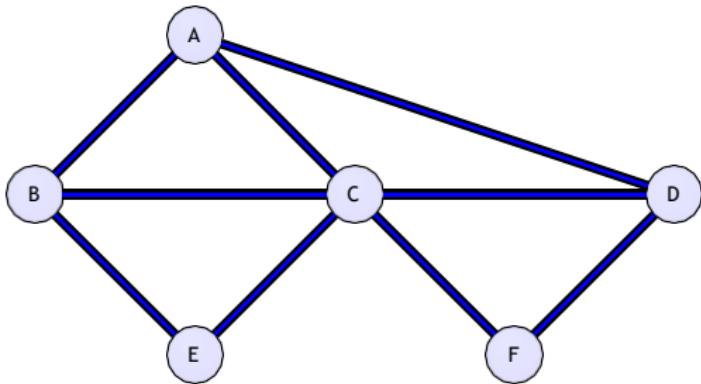
- ① Choose an arbitrary vertex and mark it visited.
- ② From the current vertex, proceed to an **unvisited, adjacent** vertex and mark it visited.
- ③ Repeat 2nd step until a vertex is reached which has no adjacent, unvisited vertices (dead-end).
- ④ At each dead-end, **backtrack** to the last visited vertex and proceed down to the **next unvisited, adjacent** vertex.

Depth-First Search - Overview

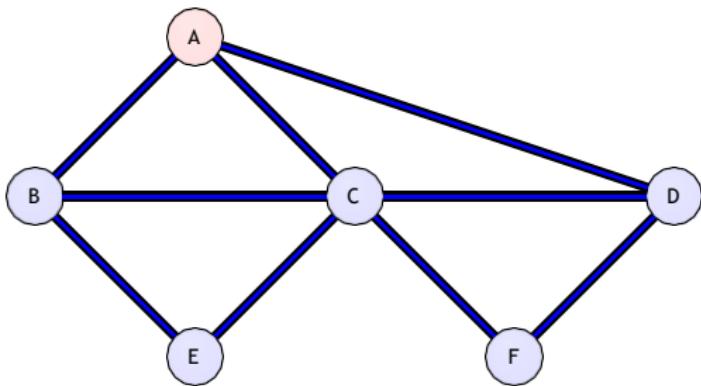
Depth-First Search (DFS) - Traversal:

- ① Choose an arbitrary vertex and mark it visited.
- ② From the current vertex, proceed to an **unvisited, adjacent** vertex and mark it visited.
- ③ Repeat 2nd step until a vertex is reached which has no adjacent, unvisited vertices (dead-end).
- ④ At each dead-end, **backtrack** to the last visited vertex and proceed down to the **next unvisited, adjacent** vertex.
- ⑤ The algorithm halts there are no unvisited vertices.

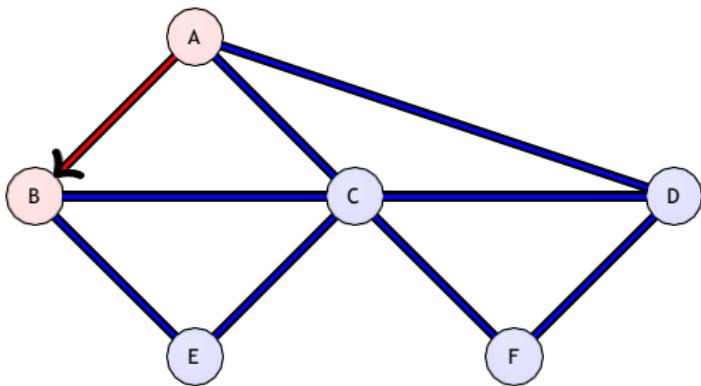
Depth-First Search - Example



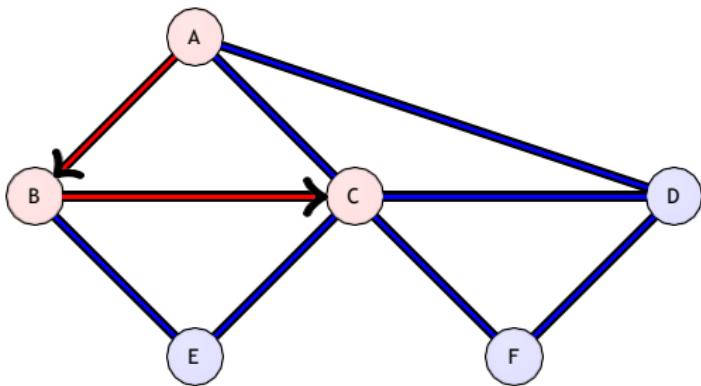
Depth-First Search - Example



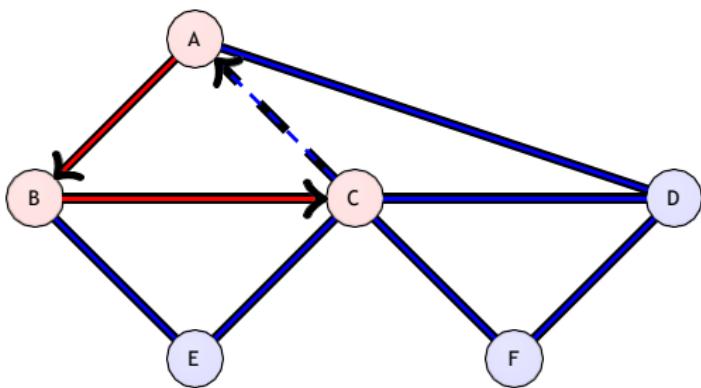
Depth-First Search - Example



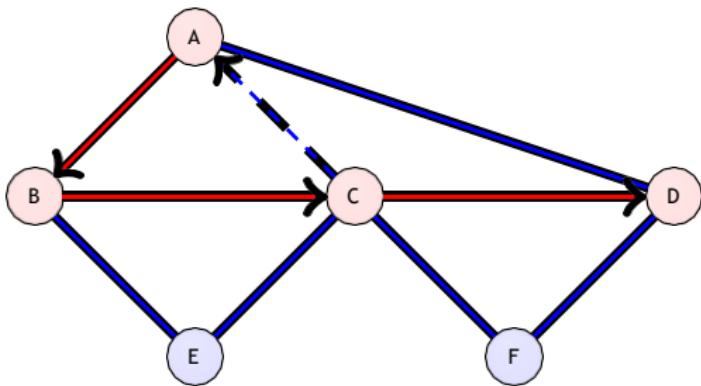
Depth-First Search - Example



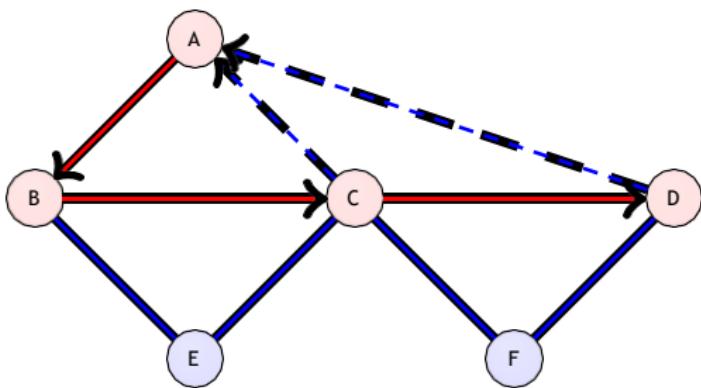
Depth-First Search - Example



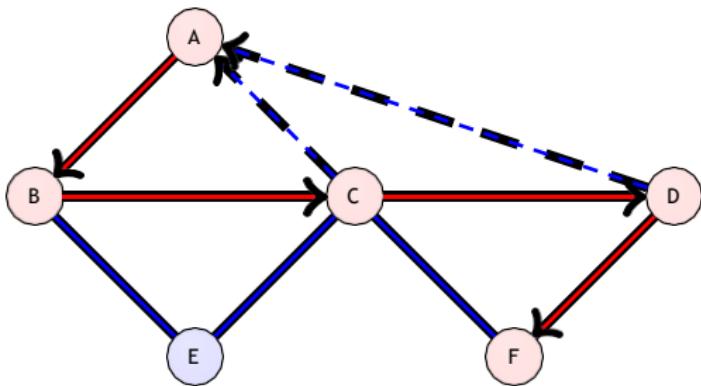
Depth-First Search - Example



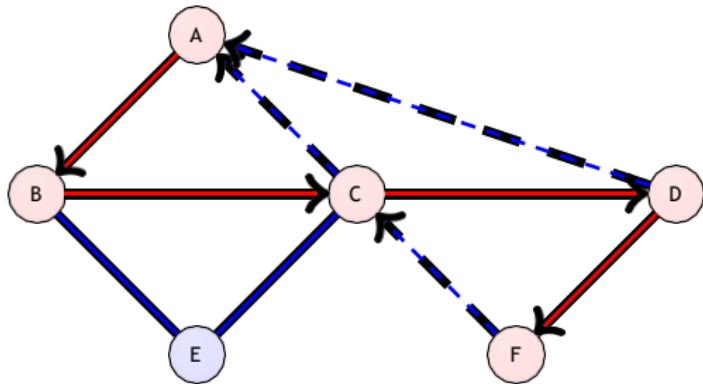
Depth-First Search - Example



Depth-First Search - Example

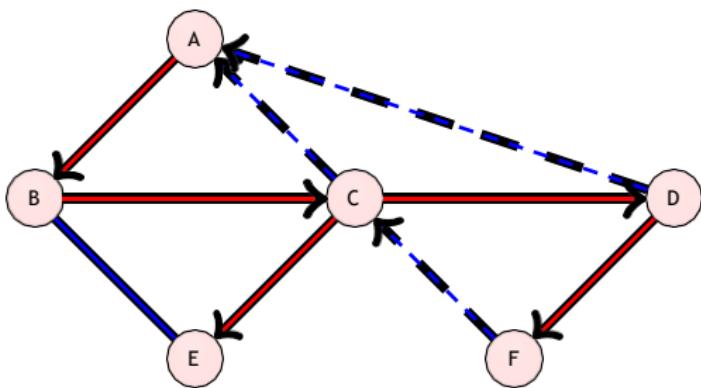


Depth-First Search - Example

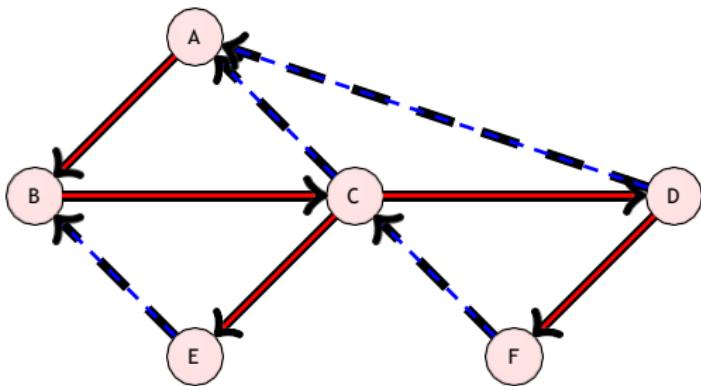


Backtrack.

Depth-First Search - Example

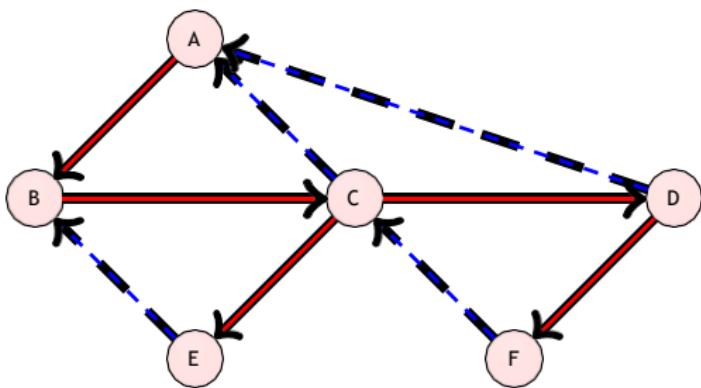


Depth-First Search - Example



Backtrack

Depth-First Search - Example



Depth-First Search - Pseudocode

ALGORITHM **DFS** (G)

/* Implement a Depth First Traversal of a graph. */

/* INPUT : Graph $G = \langle V, E \rangle$ */

/* OUTPUT : Graph G with its vertices marked with consecutive */

/* integers in initial encounter order. */

1: <i>count</i> = 0	<i>d</i> number of nodes visited
2: for v in V do	<i>d</i> mark all nodes unvisited
3: <i>Marked</i> [v] = 0	
4: end for	
5: for v in V do	<i>d</i> visit each unmarked node
6: if not <i>Marked</i> [v] then	
7: DFSR (v)	
8: end if	
9: end for	

Depth-First Search - Pseudocode

ALGORITHM **DFSR** (v)

/* Recursively visit all connected vertices. */

/* INPUT : A starting vertex v */

/* OUTPUT : Graph G with its vertices marked with consecutive */

/* integers in initial encounter order. */

- | | |
|--|---|
| 1: <i>count</i> = <i>count</i> + 1 | <i>d</i> increment the node visited counter |
| 2: <i>Marked</i> [<i>v</i>] = <i>count</i> | <i>d</i> mark node as visited |
| 3: for $v^j \in V$ adjacent to <i>v</i> do | <i>d</i> recursively visit all |
| 4: if not <i>Marked</i> [v^j] then | <i>d</i> unmarked adjacent nodes |
| 5: DFSR (v^j) | |
| 6: end if | |
| 7: end for | |

Depth-First Search - Summary

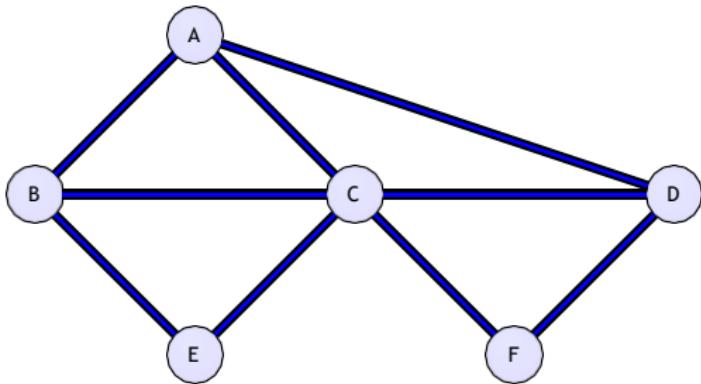
- A DFS search can be implemented with graphs represented as:
 - Adjacency matrices: $O(|V|^2)$
 - It is a graph traversal, so we need to iterate over all vertices ($|V|$ of these). For each vertex, we need to check the neighbours of it. For the matrix representation, the only way we can guarantee to find all neighbours of vertex i is to do a linear scan across its row in the matrix, which has $|V|$ elements. So $|V| * |V|$ gives $O(|V|^2)$ complexity. The traversal also needs to setup visited status, which requires $O|V|$ complexity, but the quadratic term dominates.
 - Adjacency lists: $O(|V| + |E|)$
 - Similarly to the matrix representation, we need to iterate over all the vertices ($|V|$ of these). For each vertex, we need to check the neighbours of it. Different for the adjacency list representation, we only need to scan through the elements in the associated linked list. The total number of elements across all the linked list (and total number of neighbours to consider) is $O(|E|)$. The traversal also needs to setup visited status, which requires $O|V|$ complexity. Hence the total is $O(|V| + |E|)$.

Breadth-First Search - Overview

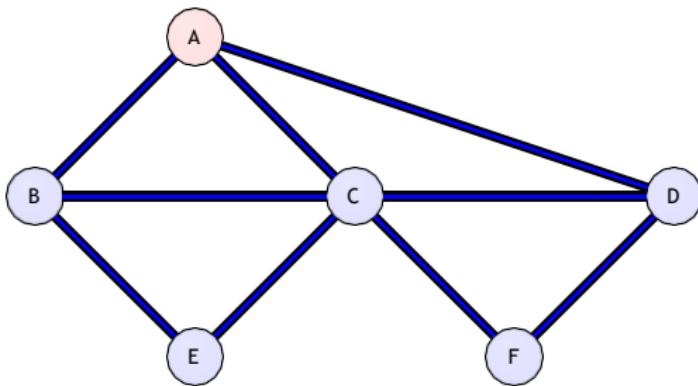
Breadth-First Search (BFS) - Traversal:

- ① Choose an arbitrary vertex v and mark it visited.
- ② Visit and mark (visited) **each of the adjacent** (neighbour) vertices of v in **turn**.
- ③ Once **all** neighbours of v have been visited, select the first neighbour that was visited, and visit all its (unmarked) neighbours.
- ④ Then select the second visited neighbour of v , and visit all its unmarked neighbours.
- ⑤ The algorithm halts when we visited all vertices.

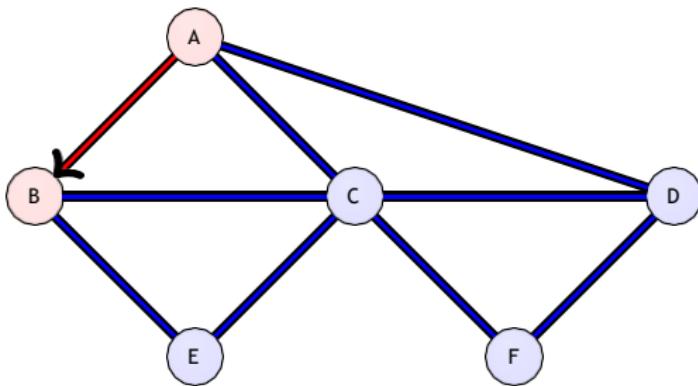
Breadth-First Search - Example



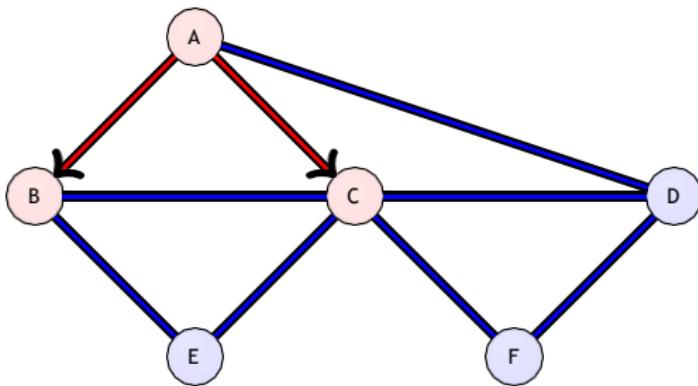
Breadth-First Search - Example



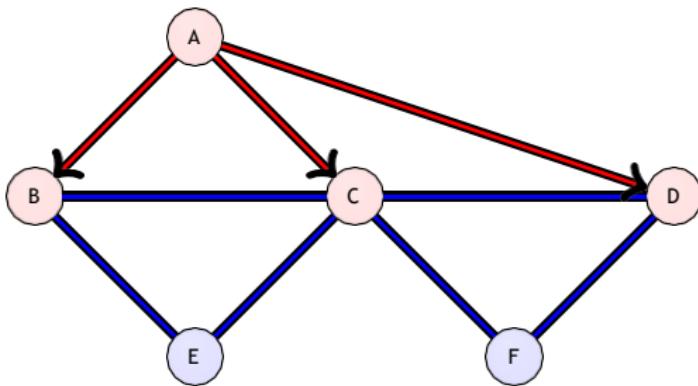
Breadth-First Search - Example



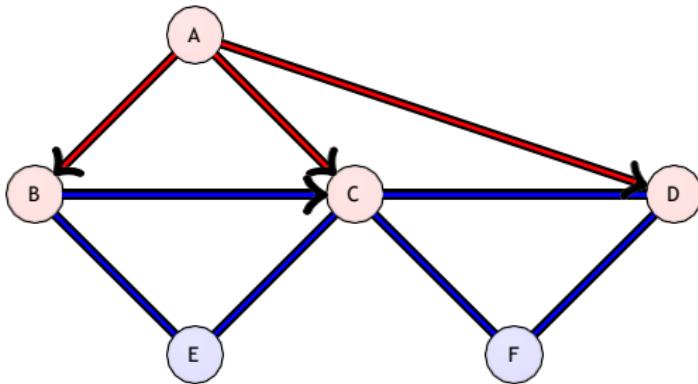
Breadth-First Search - Example



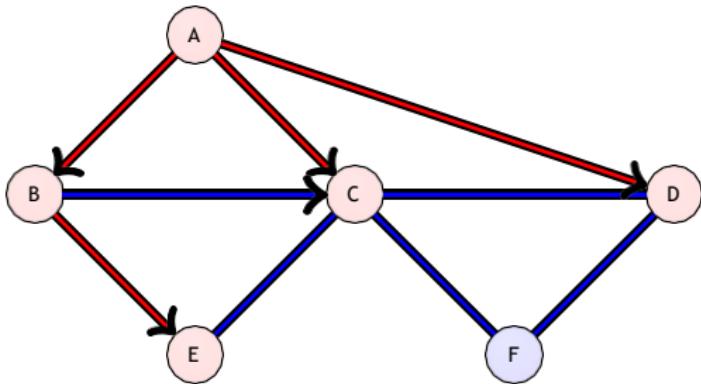
Breadth-First Search - Example



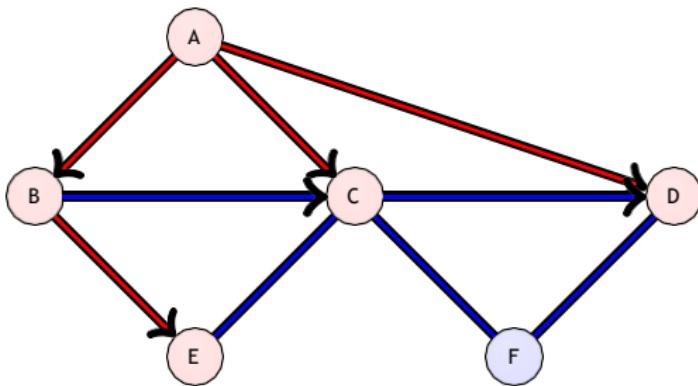
Breadth-First Search - Example



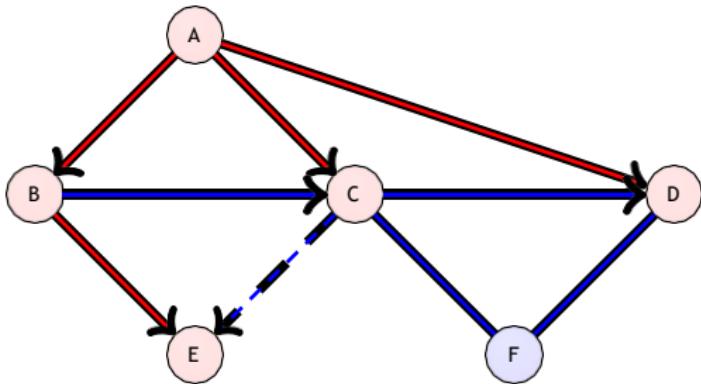
Breadth-First Search - Example



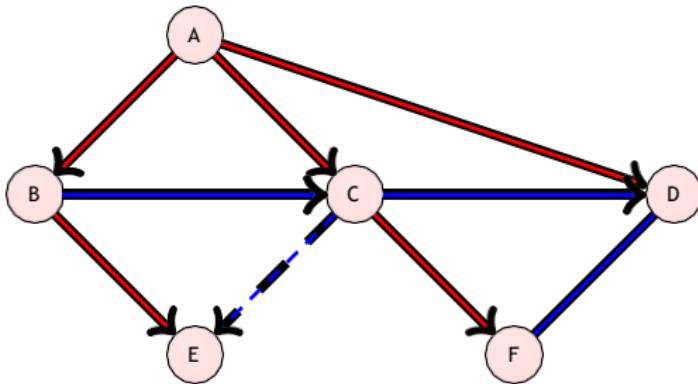
Breadth-First Search - Example



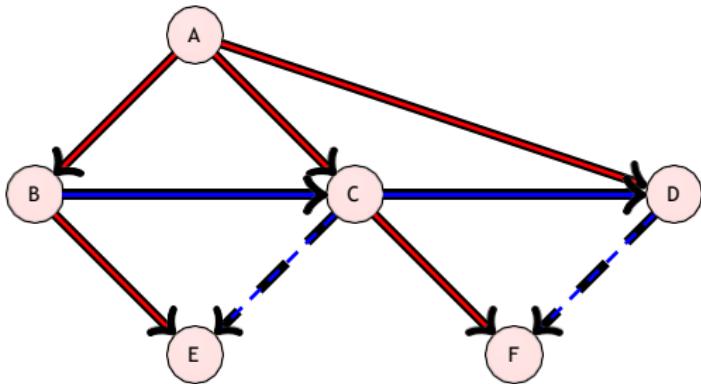
Breadth-First Search - Example



Breadth-First Search - Example



Breadth-First Search - Example



Graph Search - Analysis

	DFS	BFS
Applications	connectivity, acyclicity	connectivity, acyclicity, shortest paths
Efficiency for adjacency matrix	$\Theta(V^2)$	$\Theta(V^2)$
Efficiency for adjacency lists	$\Theta(V + E)$	$\Theta(V + E)$

Overview

1 [Overview](#)

2 [Sorting](#)

3 [Exhaustive Search](#)

4 [Graph Search](#)

5 Case Study

6 Summary

Case Study

From this week onwards, we are going to study a particular problem and how to solve it using one of the algorithms we learn in that week's paradigm.

The structure will be a general problem statement, we then discuss how to map this to a problem we know the algorithm for, then solve it using that algorithm. There maybe more than one algorithm that can solve a problem, then we should evaluate in terms of the problem requirements and characteristics such as time complexity.

Case Study - Problem

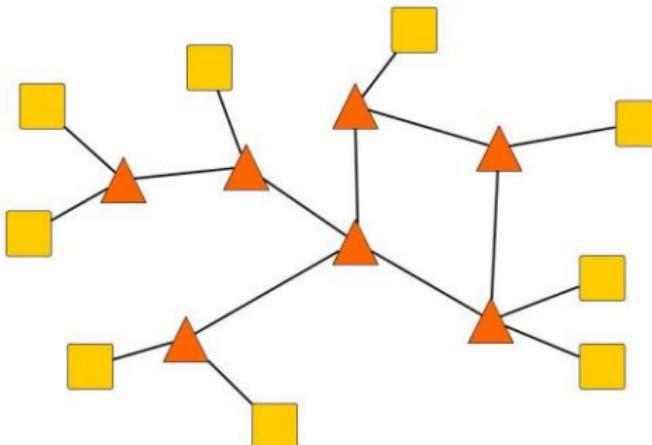
Case Study Problem

ABC Gold Plated Power Company operates a power transmission network and recently had some failures in their lines and shutdown of some substations. They want to determine if all their substations and customers' homes are connected to the network.

They have asked you to help them. How would you approach this problem?

Case Study - Mapping the Problem to a Known Problem

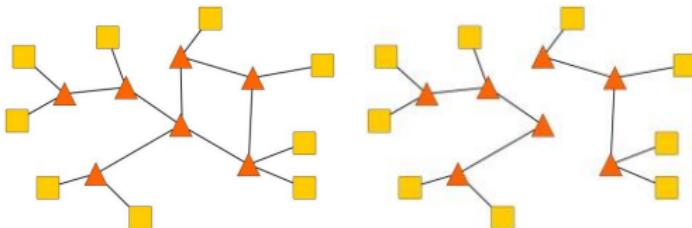
This can be mapped into a graph problem. Each substation and home is a vertex in that graph. Each transmission line between substation and/or home is an edge.



Case Study - Solving the Problem

Finding whether this graph is connected is equivalent to finding if all substations and homes are connected.

We know we can use either DFS and BFS to determine if a graph is connected. A DFS or BFS traversal of this graph is fully connected if it contains all vertices (why is this so?)



(d) Fully connected.

(e) Not fully connected.

Overview

1 [Overview](#)

2 [Sorting](#)

3 [Exhaustive Search](#)

4 [Graph Search](#)

5 Case Study

6 Summary

Summary

- Introduced the *Brute force* algorithmic approach.
- Sorting - selection and bubble sort.
- Exhaustive search (enumeration) - knapsack
- Graph search - DFS, BFS

Next week: Decrease-and-conquer and learn about more algorithms that can be used to solve interesting problems.

Levitin - The design and analysis of algorithms

This week we will be covering the material from Chapter 4.

Learning Outcomes:

- Understand the *Decrease-and-conquer* algorithmic approach.
- Understand, contrast and apply:
 - Decrease-by-a-constant algorithms - insertion and topological sort.
 - Decrease-by-a-constant-factor algorithms - binary search, fake coin problem.
 - Variable-size decrease algorithms - binary search tree.

Outline

- ① Overview
- ② Decrease-by-a-Constant: Insertion Sort
- ③ Decrease-by-a-Constant: Topological Sorting
- ④ Decrease-by-a-Constant-Factor Algorithms
- ⑤ Variable-Size Decrease Algorithms & Binary Search Trees
- ⑥ Case Study
- ⑦ Summary

Overview

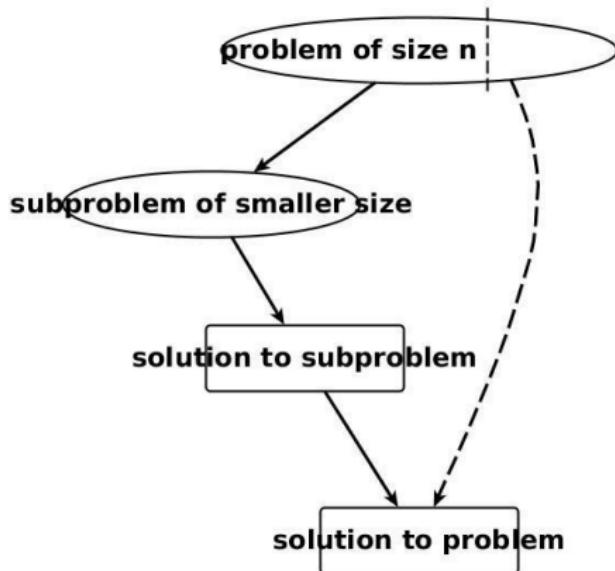
- ① Overview
- ② Decrease-by-a-Constant: Insertion Sort
- ③ Decrease-by-a-Constant: Topological Sorting
- ④ Decrease-by-a-Constant-Factor Algorithms
- ⑤ Variable-Size Decrease Algorithms & Binary Search Trees
- ⑥ Case Study
- ⑦ Summary

Decrease-and-conquer Approach

Process:

- ① Reduce a problem instance to a smaller instance of the same problem.
 - ② Solve the smaller instance.
 - ③ Extend the solution of the smaller instance to obtain the solution to the original instance.
- Sometimes referred to as the **inductive** or **incremental** approach.

Decrease-and-conquer Approach



Decrease-and-Conquer - Examples

1 Decrease-by-a-constant

- **Insertion Sorting**
- **Topological Sorting**
- Algorithms for generating permutations and subsets

2 Decrease-by-a-constant-factor

- **Binary Search**
- **Fake-coin Problem**
- Multiplication à la russe
- Josephus Problem

3 Variable-size-decrease

- **Search, Insert and Delete in a Binary Search Tree**
- Euclid's Algorithm
- Interpolation Search
- Selection by Partitioning
- Game of Nim

Overview

- ① Overview
- ② Decrease-by-a-Constant: Insertion Sort
- ③ Decrease-by-a-Constant: Topological Sorting
- ④ Decrease-by-a-Constant-Factor Algorithms
- ⑤ Variable-Size Decrease Algorithms & Binary Search Trees
- ⑥ Case Study
- ⑦ Summary

Insertion Sort - Sketch

Insertion sort is the method people often use to sort a hand of playing cards. The basic idea:

Insertion Sort - Sketch

Insertion sort is the method people often use to sort a hand of playing cards. The basic idea:

- Consider each element one at a time (left to right).
- Insert each element in its proper place among those already considered. (i.e. insert into a already sorted sub-file). This is a right to left scan.

Insertion Sort - Example

34	53	21	9	12	37	75	4	47	18	27	57	1	68	88	11
----	----	----	---	----	----	----	---	----	----	----	----	---	----	----	----

Insertion Sort - Example

34	53	21	9	12	37	75	4	47	18	27	57	1	68	88	11
----	----	----	---	----	----	----	---	----	----	----	----	---	----	----	----

Insertion Sort - Example

34	53	21	9	12	37	75	4	47	18	27	57	1	68	88	11
----	----	----	---	----	----	----	---	----	----	----	----	---	----	----	----

Insertion Sort - Example

34	53
----	----

21	9	12	37	75	4	47	18	27	57	1	68	88	11
----	---	----	----	----	---	----	----	----	----	---	----	----	----

Insertion Sort - Example

34	53	21	9	12	37	75	4	47	18	27	57	1	68	88	11
----	----	----	---	----	----	----	---	----	----	----	----	---	----	----	----

Insertion Sort - Example

21	34	53
----	----	----

9	12	37	75	4	47	18	27	57	1	68	88	11
---	----	----	----	---	----	----	----	----	---	----	----	----

Insertion Sort - Example

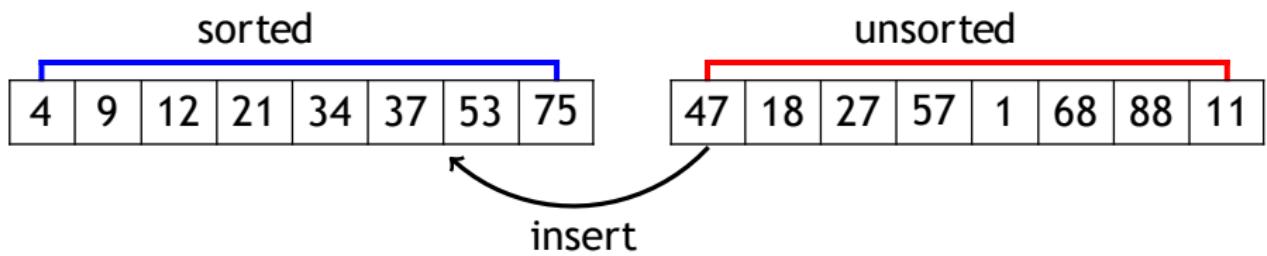
21	34	53	9	12	37	75	4	47	18	27	57	1	68	88	11
----	----	----	---	----	----	----	---	----	----	----	----	---	----	----	----

Insertion Sort - Example

9	21	34	53
---	----	----	----

12	37	75	4	47	18	27	57	1	68	88	11
----	----	----	---	----	----	----	----	---	----	----	----

Insertion Sort - Example



Insertion Sort - Pseudocode

```
ALGORITHM InsertionSort ( $A[0 \dots n - 1]$ )
/* Sort an array using an insertion sort. */
/* INPUT : An array  $A[0 \dots n - 1]$  of orderable elements. */
/* OUTPUT : An array  $A[0 \dots n - 1]$  sorted in order. */

1: for  $i = 1$  to  $n - 1$  do
2:    $v = A[i]$ 
3:    $j = i - 1$ 
4:   while  $j \geq 0$  and  $A[j] > v$  do
5:      $A[j + 1] = A[j]$ 
6:      $j = j - 1$ 
7:   end while
8:    $A[j + 1] = v$ 
9: end for
```

Insertion Sort - Analysis

- **Worst Case :** The input is in reverse order.

$$C_w(n) = \sum_{i=1}^{n-1} \sum_{j=0}^i 1 = \sum_{i=1}^{n-1} i = \frac{(n-1)n}{2} \in O(n^2)$$

Insertion Sort - Analysis

- **Worst Case :** The input is in reverse order.

$$C_w(n) = \sum_{i=1}^{n-1} \sum_{j=0}^{i-1} 1 = \sum_{i=1}^{n-1} i = \frac{(n-1)n}{2} \in O(n^2)$$

- **Average Case :** For randomly ordered data, we expect each item to move halfway back.

$$C_a(n) \approx \frac{n^2}{4} \in O(n^2)$$

- **Best Case :** Next slide

Insertion Sort - Analysis

Best Case :

- ① In terms of the input, what is the scenario/circumstance where we can achieve the best case?
- ② What is the time complexity for the best case?

Useful websites

Interesting and useful sorting visualisation website:

<https://visualgo.net/bn/sorting>

Overview

- ① Overview
- ② Decrease-by-a-Constant: Insertion Sort
- ③ Decrease-by-a-Constant: Topological Sorting
- ④ Decrease-by-a-Constant-Factor Algorithms
- ⑤ Variable-Size Decrease Algorithms & Binary Search Trees
- ⑥ Case Study
- ⑦ Summary

Topological Sort

Imagine we have the following problems:



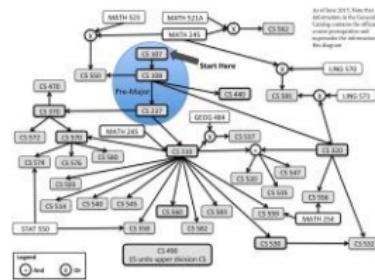
Job scheduling with
order dependencies:
What is the order the
jobs should be
processed to avoid
breaking these
dependencies?

Topological Sort

Imagine we have the following problems:



Job scheduling with order dependencies:
What is the order the jobs should be processed to avoid breaking these dependencies?



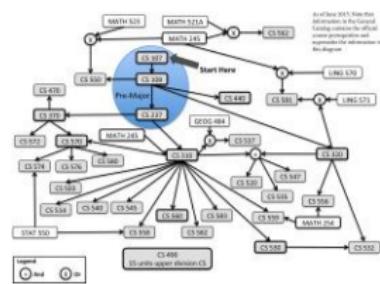
Subject selection with pre-requisites: What is the order the subjects could be taken to ensure we have all the pre-requisites?

Topological Sort

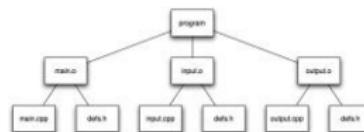
Imagine we have the following problems:



Job scheduling with order dependencies:
What is the order the jobs should be processed to avoid breaking these dependencies?



Subject selection with pre-requisites: What is the order the subjects could be taken to ensure we have all the pre-requisites?



Makefile compilation:
What is the order the source files should be compiled to ensure we can build a working program?

Topological Sort

Topological sort produces a traversal ordering of the vertices/nodes for directed graphs.

Topological Sort

Topological sort produces a traversal ordering of the vertices/nodes for directed graphs.

Topological Sorting Problem

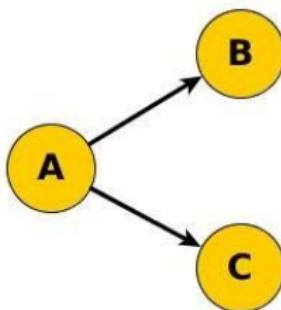
Given a digraph, list its vertices in such an order that for every edge (a, b) in the graph, vertex a must appear before vertex b in the list.

Topological Sort

Topological sort produces a traversal ordering of the vertices/nodes for directed graphs.

Topological Sorting Problem

Given a digraph, list its vertices in such an order that for every edge (a, b) in the graph, vertex a must appear before vertex b in the list.

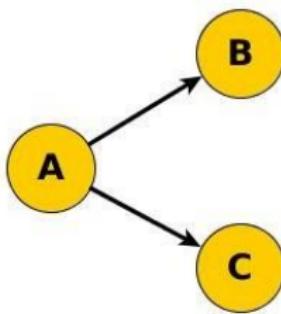


Topological Sort

Topological sort produces a traversal ordering of the vertices/nodes for directed graphs.

Topological Sorting Problem

Given a digraph, list its vertices in such an order that for every edge (a, b) in the graph, vertex a must appear before vertex b in the list.



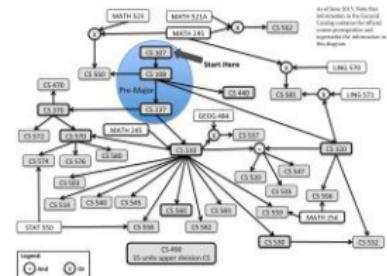
NOTE : If the graph is a **directed acyclic graph (DAG)**, the topological sort has at least one solution.

Topological Sort - Applications

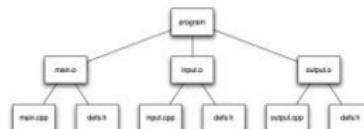
Imagine we have the following problems:



Job scheduling with order dependencies:
What is the order the jobs should be processed to avoid breaking these dependencies?



Subject selection with pre-requisites: What are the order the subjects could be taken to ensure we have all the pre-requisites?



Makefile compilation:
What is the order the source files should be compiled to ensure we can build a working program?

Topological Sort - Approaches

There are two different approaches to solve this problem:

- ① DFS Method
- ② Source Removal Method (focus of this lecture)

Topological Sort - Source Removal Method

Idea: Select **next** vertex in ordering that respect the (a, b) ordering. If we can find a vertex a that does not have any incoming vertex, than it cannot violate the property. Such a vertex is a **source** vertex (one that has no incoming vertices).

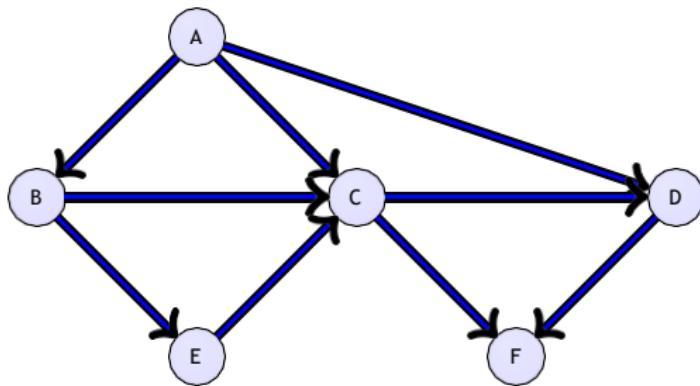
Topological Sort - Source Removal Method

Idea: Select **next** vertex in ordering that respect the (a, b) ordering. If we can find a vertex a that does not have any incoming vertex, than it cannot violate the property. Such a vertex is a **source** vertex (one that has no incoming vertices).

Source Removal Method:

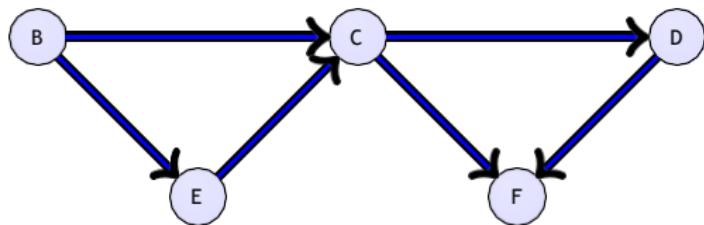
- Choose a **source** vertex.
- **Delete the vertex and all incident edges** and **append** the vertex to topological ordered list.
- Repeat the selection of a source vertex and deletion process for the remaining graph until no vertices are left.

Topological Sort - Source Removal Method



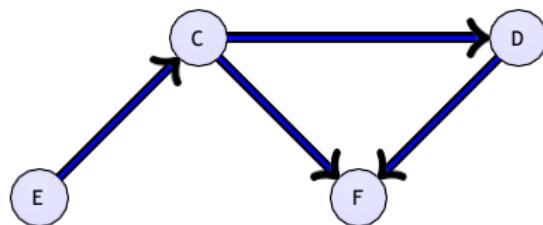
Solution :

Topological Sorting - Source Removal Method



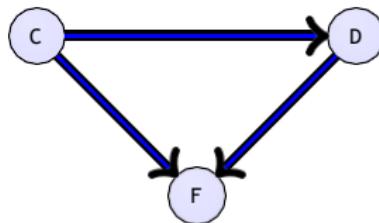
Solution : A

Topological Sorting - Source Removal Method



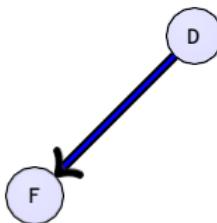
Solution : A B

Topological Sorting - Source Removal Method



Solution : A B E

Topological Sorting - Source Removal Method



Solution : A B E C

Topological Sorting - Source Removal Method

F

Solution : A B E C D

Topological Sorting - Source Removal Method

Solution : A B E C D F

Topological Sorting - Summary & Questions

- Topological sorting can have more than one solution, and often does for very large DAGs.

Topological Sorting - Summary & Questions

- Topological sorting can have more than one solution, and often does for very large DAGS.
- Source removal algorithm:
 - How do you find a **source** (or determine that such a vertex does not exist) in a digraph represented by an **adjacency matrix**? What is the time efficiency? ([Homework](#))

Topological Sorting - Summary & Questions

- Topological sorting can have more than one solution, and often does for very large DAGs.
- Source removal algorithm:
 - How do you find a **source** (or determine that such a vertex does not exist) in a digraph represented by an **adjacency matrix**? What is the time efficiency? ([Homework](#))
 - How do you find a **source** (or determine that such a vertex does not exist) in a digraph represented by an **adjacency list**? What is the time efficiency? ([Homework](#))

Topological Sorting - Summary & Questions

- Topological sorting can have more than one solution, and often does for very large DAGs.
- Source removal algorithm:
 - How do you find a **source** (or determine that such a vertex does not exist) in a digraph represented by an **adjacency matrix**? What is the time efficiency? ([Homework](#))
 - How do you find a **source** (or determine that such a vertex does not exist) in a digraph represented by an **adjacency list**? What is the time efficiency? ([Homework](#))
 - The source removal algorithm for a digraph represented as an adjacency list can be implemented such that the running time is $O(|V| + |E|)$. How? ([Homework](#))

Overview

- ① Overview
- ② Decrease-by-a-Constant: Insertion Sort
- ③ Decrease-by-a-Constant: Topological Sorting
- ④ Decrease-by-a-Constant-Factor Algorithms
- ⑤ Variable-Size Decrease Algorithms & Binary Search Trees
- ⑥ Case Study
- ⑦ Summary

Decrease-by-a-constant-factor Approaches

Algorithms that use this approach divide the problem into parts (half, thirds, etc), and then recursively operate on one of the halves, thirds etc.

Hence, at each iteration, we decrease the problem by a constant (a half, a third etc).

Decrease-by-a-constant-factor Approaches

Algorithms that use this approach divide the problem into parts (half, thirds, etc), and then recursively operate on one of the halves, thirds etc.

Hence, at each iteration, we decrease the problem by a constant (a half, a third etc).

We study two examples:

- Binary search
- Fake coin problem

Binary Search

Binary search is a worst-case optimal algorithm for searching in a sorted sequence of elements.

- Given a sorted sequence, compare the value in the array at position $n/2$ with a key k .
- If $A[n/2] > k$, compare k with the midpoint of the lower half.
- If $A[n/2] < k$, compare k with the midpoint of the upper half.
- If $A[n/2] = k$, return $n/2$ (the index of the position containing k).

Binary Search - Example

1	3	5	9	12	24	29	34	35	37	53	62	74	53	62	74	92
---	---	---	---	----	----	----	----	----	----	----	----	----	----	----	----	----

BINARYSEARCH(5)

Binary Search - Example

1	3	5	9	12	24	29	34	35	37	53	62	74	53	62	74	92
---	---	---	---	----	----	----	----	----	----	----	----	----	----	----	----	----

Binary Search - Example

1	3	5	9	12	24	29	34	35	37	53	62	74	53	62	74	92
---	---	---	---	----	----	----	----	----	----	----	----	----	----	----	----	----

$$5 < 35$$

Binary Search - Example

1	3	5	9	12	24	29	34	35	37	53	62	74	53	62	74	92
---	---	---	---	----	----	----	----	----	----	----	----	----	----	----	----	----

Continue left

Binary Search - Example

1	3	5	9	12	24	29	34	35	37	53	62	74	53	62	74	92
---	---	---	---	----	----	----	----	----	----	----	----	----	----	----	----	----

$$5 < 9$$

Binary Search - Example

1	3	5	9	12	24	29	34	35	37	53	62	74	53	62	74	92
---	---	---	---	----	----	----	----	----	----	----	----	----	----	----	----	----

$$5 > 3$$

Binary Search - Example

1	3	5	9	12	24	29	34	35	37	53	62	74	53	62	74	92
---	---	---	---	----	----	----	----	----	----	----	----	----	----	----	----	----

FOUND!

Binary Search - Recursive

```
ALGORITHM RecursiveBinarySearch ( $A[l \dots r], k$ )
/* A recursive binary search in an ordered array. */
/* INPUT : An array  $A[l \dots r]$  of ordered elements, and a search key  $k$ . */
/* OUTPUT : an index to the position of  $k$  in  $A$  if  $k$  is found or  $-1$  otherwise. */

1: if  $l > r$  then
2:   return  $-1$ 
3: else
4:    $m \leftarrow \lfloor (l + r)/2 \rfloor$ 
5:   if  $k = A[m]$  then
6:     return  $m$ 
7:   else if  $k < A[m]$  then
8:     return RecursiveBinarySearch ( $A[l \dots m - 1], k$ )
9:   else
10:    return RecursiveBinarySearch ( $A[m + 1 \dots r], k$ )
11: end if
12: end if
```

Analysis - Binary Search

1. $C(n) = C([n/2]) + 1$ for $n > 1$ and $C(1) = 1$.
2. Using the smoothness rule, $n = 2^k$.
3. $C(2^k) = C(2^{k-1}) + 1$ for $k > 0$, $C(2^0) = C(1) = 1$.
4. Substitute $C(2^{k-1}) = C(2^{k-2}) + 1$.
5. $C(2^k) = [C(2^{k-2}) + 1] + 1 = C(2^{k-2}) + 2$.
6. Substitute $C(2^{k-2}) = C(2^{k-3}) + 1$.
7. $C(2^k) = [C(2^{k-3}) + 1] + 2 = C(2^{k-3}) + 3$.
8. We see the pattern $C(2^k) = C(2^{k-i}) + i$ emerge.
9. $C(2^k) = C(2^{k-i}) + i$.
10. We want $C(2^{k-i}) = C(2^0)$, or $k - i = 0$ or $i = k$.
11. $C(2^k) = C(2^{k-k}) + k = C(2^0) + k = 1 + k$.
12. If $n = 2^k$ then $k = \log_2 n$:
13. $C(n) = \log_2 n + 1 \in O(\log_2 n)$.

Binary Search Properties

Worst case of $O(\log(n))$

Binary Search Properties

Worst case of $O(\log(n))$

Only achieved if:

- Array is sorted.
- The array has $O(1)$ access to any position.

Fake-Coin Problem

Fake-Coin Problem

Given a **stack** of n identical-looking coins which contains exactly **one fake coin** (which is lighter) and a scale/weigh, devise an efficient algorithm for detecting the fake coin.



Fake-Coin Problem

Fake-Coin Problem

Given a **stack** of n identical-looking coins which contains exactly **one fake coin** (which is lighter) and a scale/weigh, devise an efficient algorithm for detecting the fake coin.

Fake-Coin Problem

Fake-Coin Problem

Given a **stack** of n identical-looking coins which contains exactly **one fake coin** (which is lighter) and a scale/weigh, devise an efficient algorithm for detecting the fake coin.

The solution is to use a decrease by half algorithm:

- Divide into two sub-stacks of $n/2$ coins and weigh on scale.
- The lighter sub-stack contains the fake coin.
- Repeat process with the lighter sub-stack until we have two coins remaining. The fake coin must be one of the two.

Fake-Coin Problem

Fake-Coin Problem

Given a **stack** of n identical-looking coins which contains exactly **one fake coin** (which is lighter) and a scale/weigh, devise an efficient algorithm for detecting the fake coin.

The solution is to use a decrease by half algorithm:

- Divide into two sub-stacks of $n/2$ coins and weigh on scale.
- The lighter sub-stack contains the fake coin.
- Repeat process with the lighter sub-stack until we have two coins remaining. The fake coin must be one of the two.

The recurrence relation for number of weighings: $C(n) = C([n/2]) + 1$ for $n > 1$, $C(1) = 0$. Gives worst case of $O(\log_2 n)$.

Fake-Coin Problem - Example

<http://www.youtube.com/watch?v=wVPCT1VjySA>

Overview

- ① Overview
- ② Decrease-by-a-Constant: Insertion Sort
- ③ Decrease-by-a-Constant: Topological Sorting
- ④ Decrease-by-a-Constant-Factor Algorithms
- ⑤ Variable-Size Decrease Algorithms & Binary Search Trees
- ⑥ Case Study
- ⑦ Summary

Trees

In its most general form, a **tree** is a connected acyclic graph.

There are many different types of trees used in computer science:

- binary trees
- m -ary search trees
- balanced trees (AVL, Red-Black)
- forests
- Kd-tree

Here we focus on **binary search trees**.

Binary Tree

A **binary tree** is:

- a tree (hence has a root node)
- every vertex has no more than two children

Binary Tree

A **binary tree** is:

- a tree (hence has a root node)
- every vertex has no more than two children

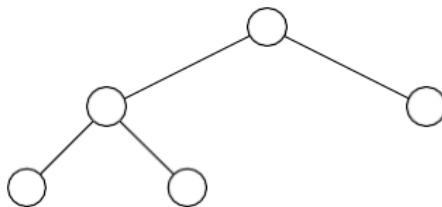
Each child is designated as either a **left** child or a **right** child of its parent

Binary Tree

A **binary tree** is:

- a tree (hence has a root node)
- every vertex has no more than two children

Each child is designated as either a **left** child or a **right** child of its parent

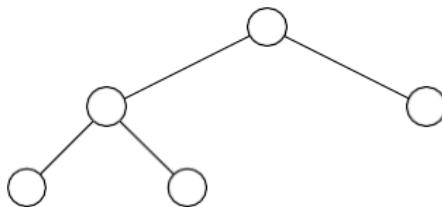


Binary Tree

A **binary tree** is:

- a tree (hence has a root node)
- every vertex has no more than two children

Each child is designated as either a **left** child or a **right** child of its parent



Technical point: If a node **does not have a child**, the pointer/reference to the child is set to **null**.

Binary Search Tree

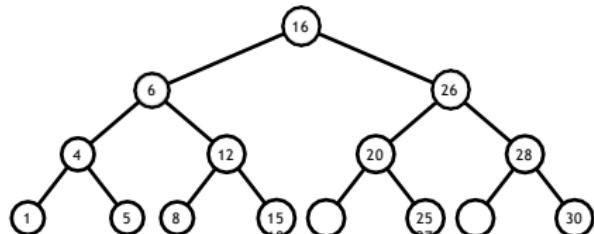
A **binary search tree** is a binary tree that additionally:

- have **values** associated with each node
- ordered such that for each parent vertex:
 - all values in its **left** subtree are **smaller** than the parent's value; and
 - all values in its **right** subtree are **larger** than the parent's value.

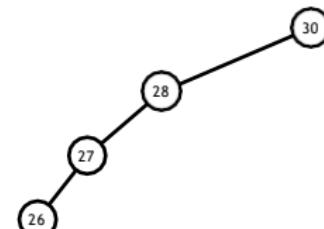
Binary Search Tree

A **binary search tree** is a binary tree that additionally:

- have **values** associated with each node
- ordered such that for each parent vertex:
 - all values in its **left** subtree are **smaller** than the parent's value; and
 - all values in its **right** subtree are **larger** than the parent's value.



Example of a balanced, full binary search tree.

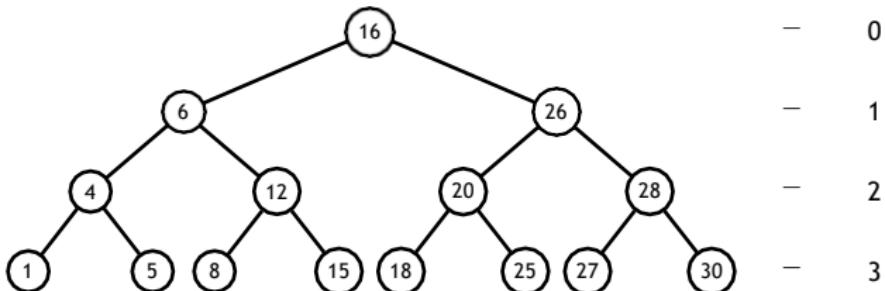


Example of an unbalanced, "stick" binary search tree.

BST Properties

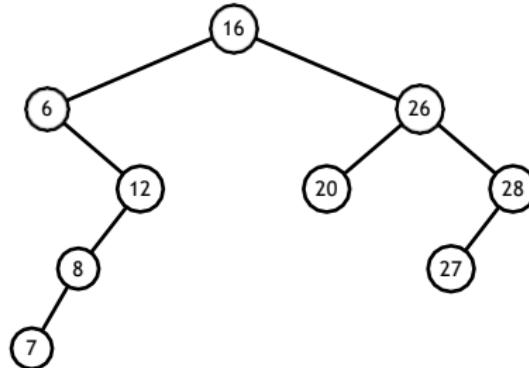
- The **height** of a tree is the length of the path from the root to the deepest node in a tree. A tree with only a root node has a height of 0.
- The **depth** of a node is the length of the path from the root to the node. The root node has a depth 0.
- The **level** of a tree is the set of all nodes at a given depth.

BST Properties - Example



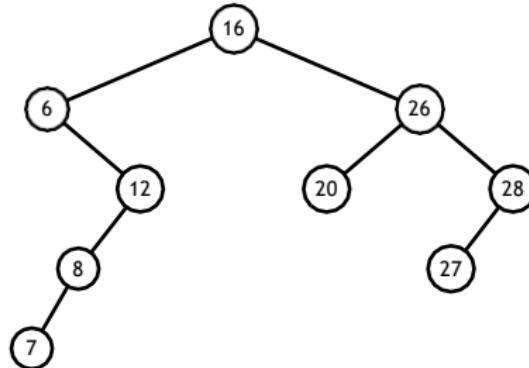
- Height = 3
- Depth of node 12 = 2
- Depth of node 18 = 3
- Nodes in level 1 = {6, 26}

BST Properties - Example



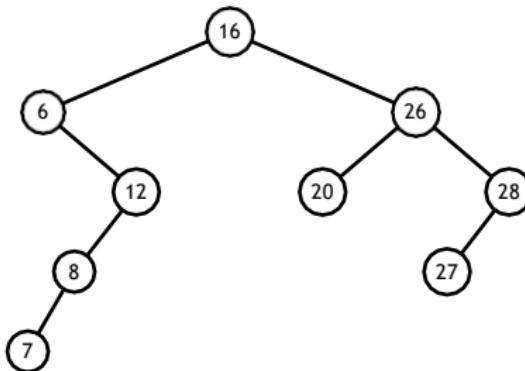
- Root is node 16.
- Height?

BST Properties - Example



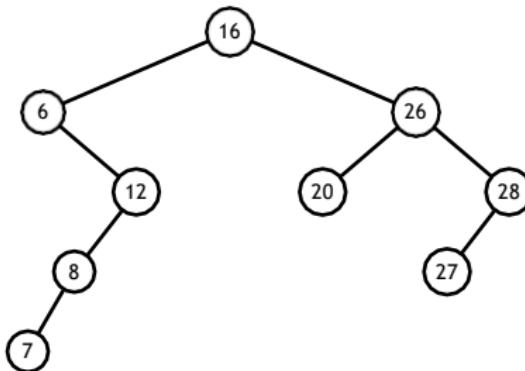
- Root is node 16.
- Height? **4**

BST Properties - Example



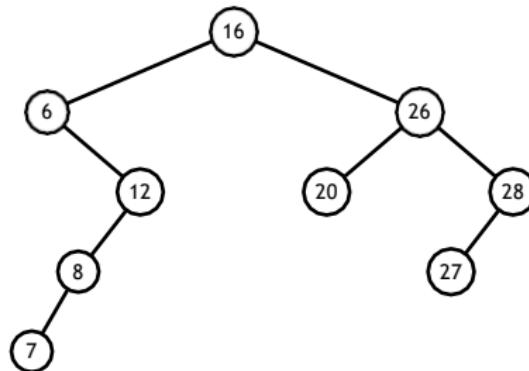
- Root is node 16.
- Height? 4
- Nodes on Level 2?

BST Properties - Example



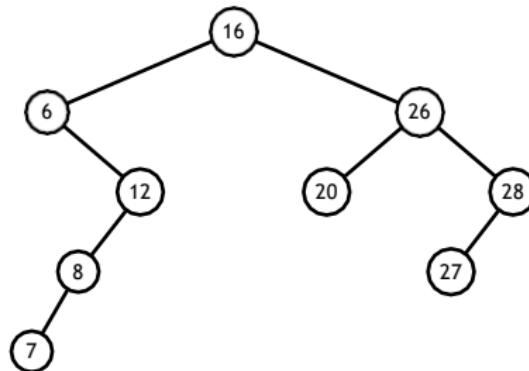
- Root is node 16.
- Height? 4
- Nodes on Level 2? {12, 20, 28}

BST Properties - Example



- Root is node 16.
- Height? 4
- Nodes on Level 2? {12, 20, 28}
- Depth of node 27?

BST Properties - Example



- Root is node 16.
- Height? 4
- Nodes on Level 2? {12, 20, 28}
- Depth of node 27? 3

BST algorithms

- Searching for a value in a BST (variable-size-decrease)
- Inserting a new value into a BST (variable-size-decrease)
- Deleting a value and associated node from a BST

BST Search

Aim: Search for a key k in tree T (represented by its root node).

BST Search

Aim: Search for a key k in tree T (represented by its root node).

Idea: Recursively search for the key k in tree, taking advantage of its structure.

BST Search

Aim: Search for a key k in tree T (represented by its root node).

Idea: Recursively search for the key k in tree, taking advantage of its structure.

Search(T, k):

- ① If T is empty, return **null**.
- ② If value of $k = \text{val}(T)$, return T .
- ③ If value of $k < \text{val}(T)$, search the left subtree (return Search(T_L, k))
- ④ If value of $k > \text{val}(T)$, search the right subtree (return Search(T_R, k))

BST Search

Aim: Search for a key k in tree T (represented by its root node).

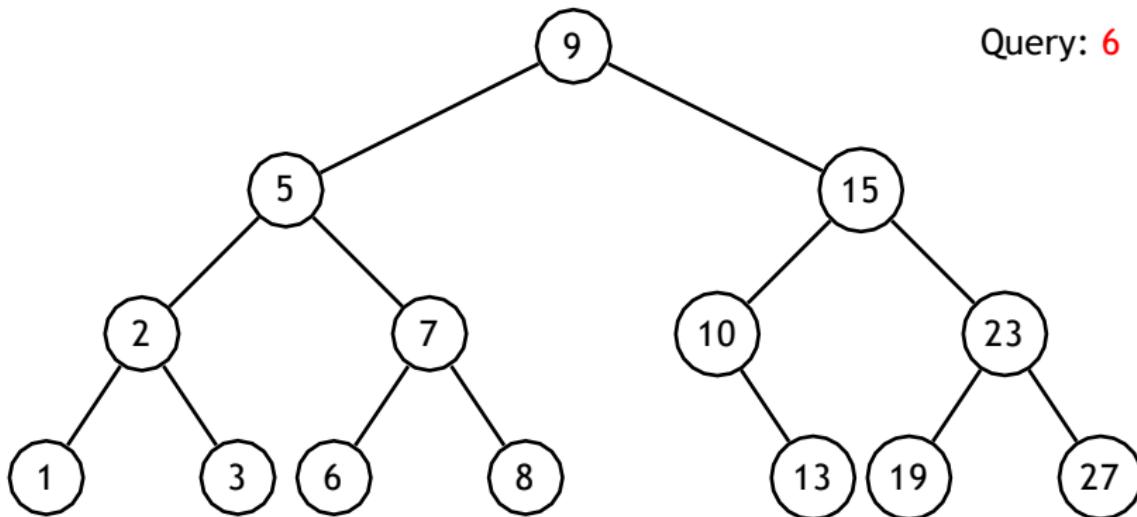
Idea: Recursively search for the key k in tree, taking advantage of its structure.

Search(T, k):

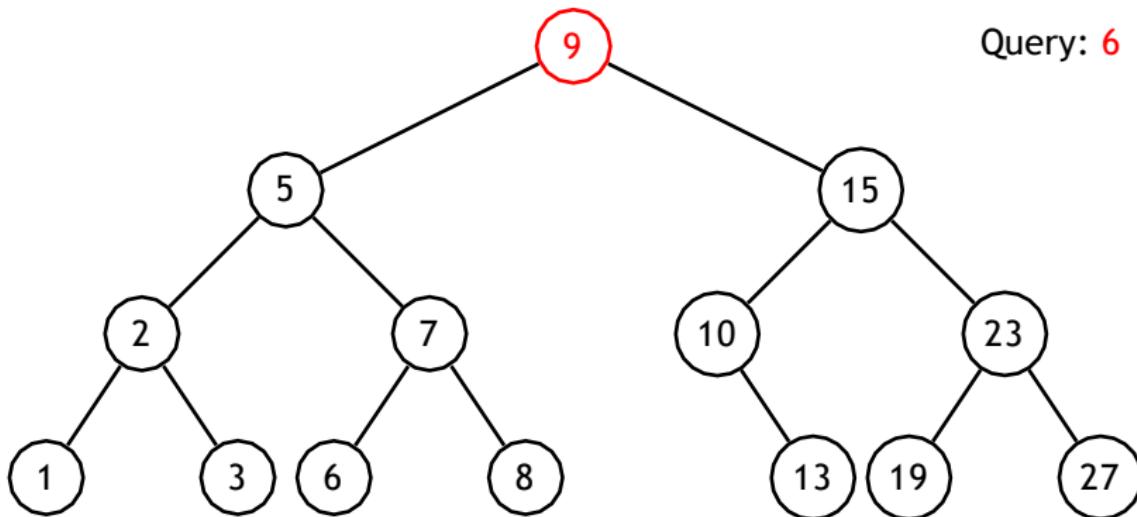
- ① If T is empty, return **null**.
- ② If value of $k = \text{val}(T)$, return T .
- ③ If value of $k < \text{val}(T)$, search the left subtree (return Search(T_L, k))
- ④ If value of $k > \text{val}(T)$, search the right subtree (return Search(T_R, k))

NOTE : This is a variable-size-decrease algorithm, as each iteration we decrease the remaining problem size by a variable amount.

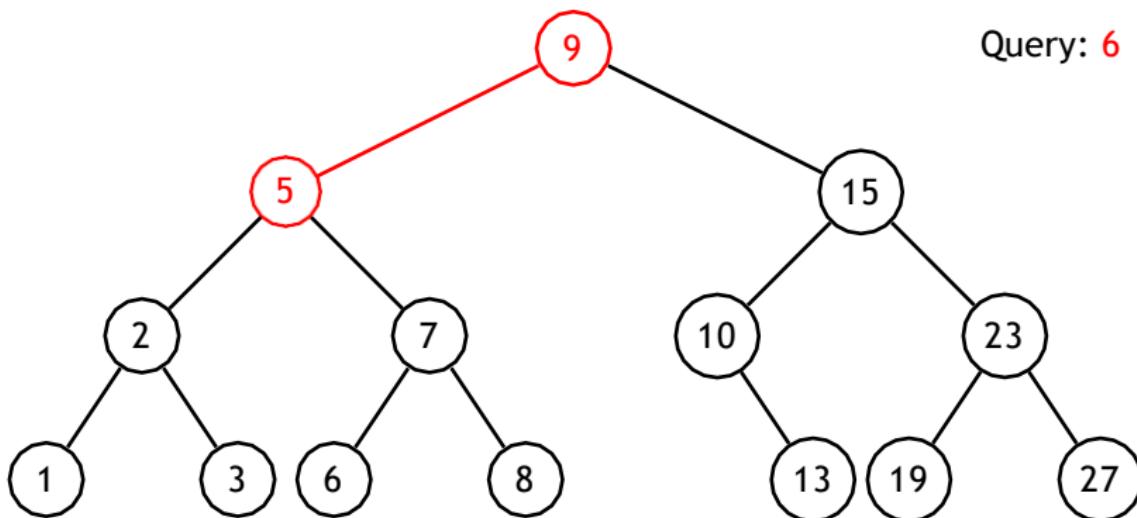
BST Search



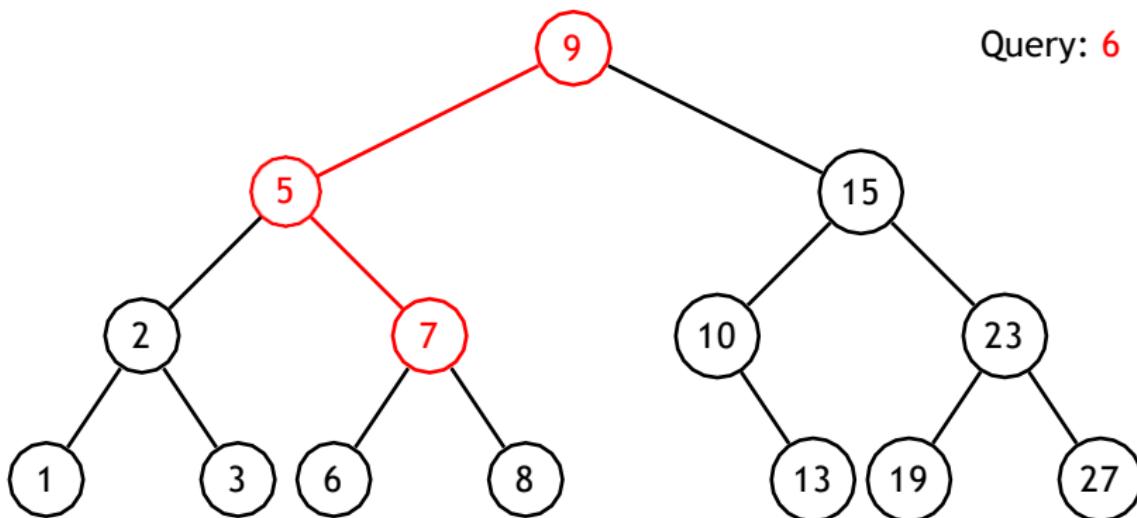
BST Search



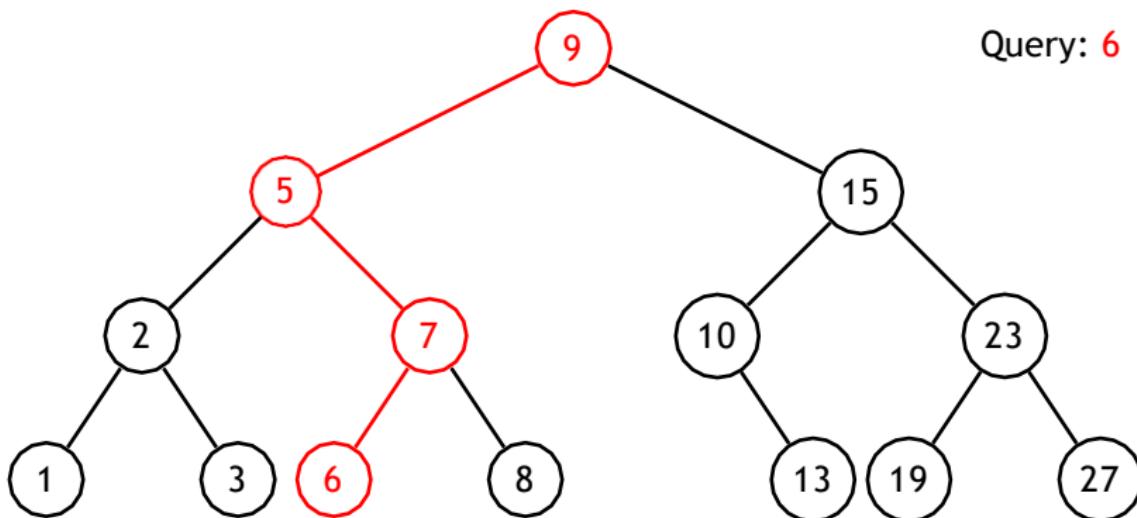
BST Search



BST Search



BST Search



Recursive BST Search

```
ALGORITHM BSTSearch( $T, k$ )
//Recursive search in a binary tree.
//INPUT : Root node  $T$  of a BST and a search key  $k$ .
//OUTPUT : A reference to the node containing  $k$  or null.

1: if  $T = \text{null}$  or  $k = \text{val}(T)$  then
2:     return  $T$ 
3: end if
4: if  $k < \text{val}(T)$  then
5:     return BSTSearch( $T_L, k$ )
6: else
7:     return BSTSearch( $T_R, k$ )
8: end if
```

BST Search - Complexity

Worst case complexity?

BST Search - Complexity

Worst case complexity? $O(n)$

BST Search - Complexity

Worst case complexity? $O(n)$

Can we do better?

BST Search - Complexity

Worst case complexity? $O(n)$

Can we do better? Yes if tree is balanced!

BST Search - Complexity

Worst case complexity? $O(n)$

Can we do better? Yes if tree is balanced!

Worst case complexity is dependent on the height of the tree.

BST Search - Complexity

Worst case complexity? $O(n)$

Can we do better? Yes if tree is balanced!

Worst case complexity is dependent on the height of the tree.

If tree is a “stick”, height = $n - 1$

BST Search - Complexity

Worst case complexity? $O(n)$

Can we do better? Yes if tree is balanced!

Worst case complexity is dependent on the height of the tree.

If tree is a “stick”, height = $n - 1$

If tree is balanced, height = $\log_2 n$

BST Insert

Aim: Insert key/value k into tree T (represented by its root node).

BST Insert

Aim: Insert key/value k into tree T (represented by its root node).

Idea: Recursively traverse the tree to find a position for the key k in T .

BST Insert

Aim: Insert key/value k into tree T (represented by its root node).

Idea: Recursively traverse the tree to find a position for the key k in T .

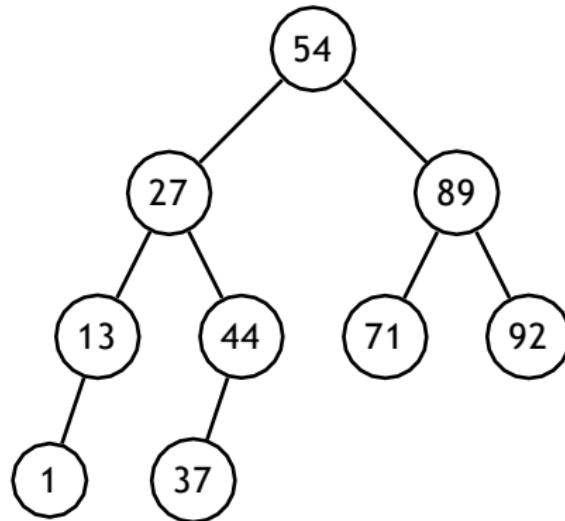
Insert(T, k):

- ① If T is empty, place k at this position.
- ② If $k < \text{val}(T)$, traverse into the left subtree (Insert(T_L, k)).
- ③ If $k > \text{val}(T)$, traverse into the right subtree (Insert(T_R, k)).

NOTE : This is a variable-size-decrease algorithm, as each iteration we decrease the remaining problem size by a variable amount.

BST Insert

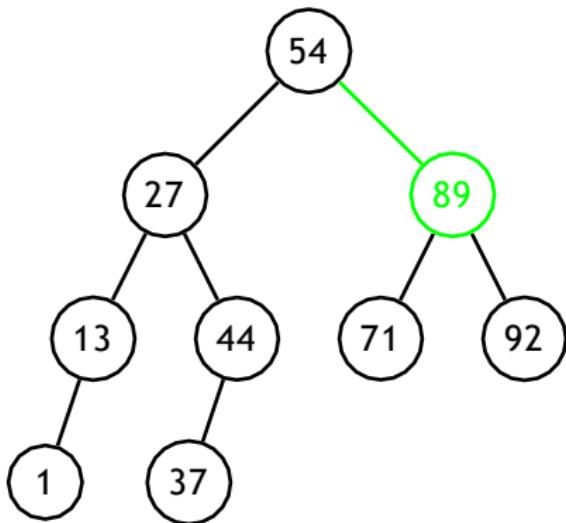
Insert: **64**



BST Insert

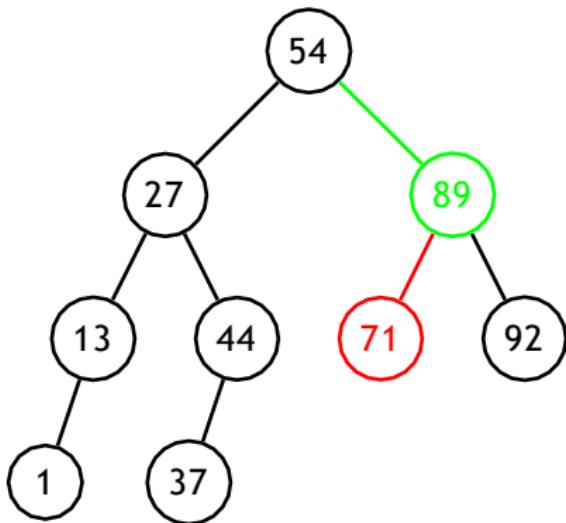
Insert: **64**

$64 > 54 \rightarrow$ right



BST Insert

Insert: **64**

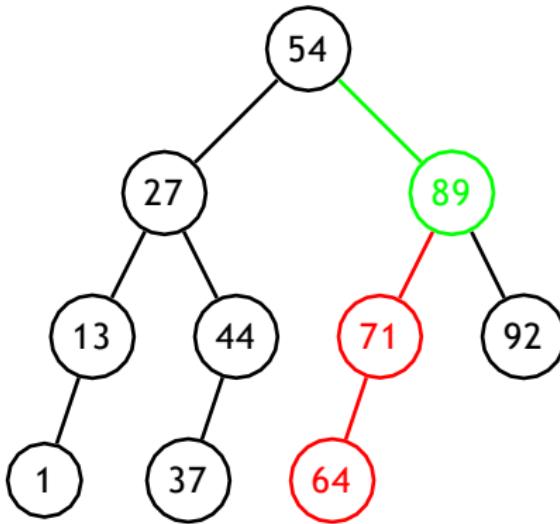


$64 > 54 \rightarrow \text{right}$

$64 < 89 \rightarrow \text{left}$

BST Insert

Insert: 64



$64 > 54 \rightarrow$ right

$64 < 89 \rightarrow$ left

$64 < 71 \rightarrow$ insert left

BST Delete

Aim: Remove or delete a node from a BST.

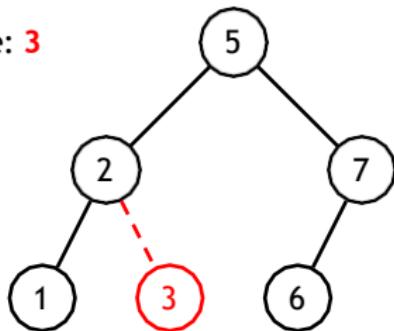
Aim: Remove or delete a node from a BST.

Idea: It can be solved using one of three cases:

- **Case 1 :** The deleted node has **no** children.
- **Case 2 :** The deleted node has **one** child.
- **Case 3 :** The deleted node has **two** children.

BST Delete

Delete: 3

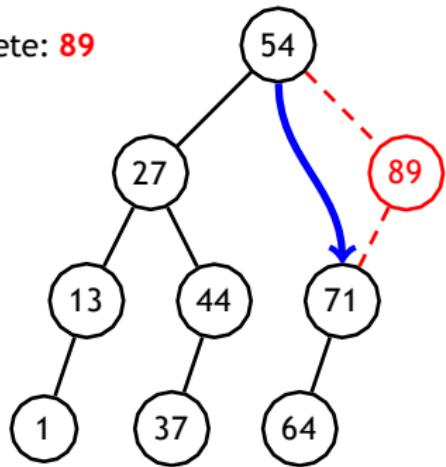


Case 1 : Deleted node has no children

- 1 If the deleted node is the **root** of a **single-node** tree, make the tree empty.
- 2 If the deleted node is a **leaf node**, set the reference from its parent to the **itself** to **null**.

BST Delete

Delete: 89

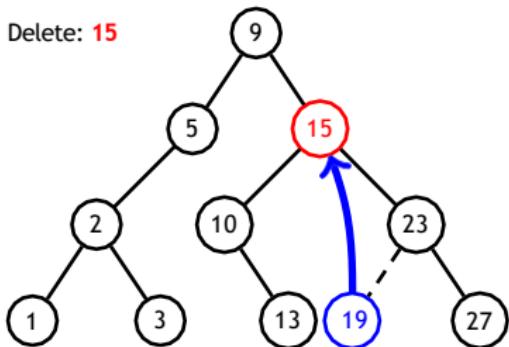


Case 2 : Deleted node has one child

- ① If the deleted node is the **root node with a single child**, make the child the new root.
- ② If the deleted node is **not** the root, make the reference from its parent to point to its single child.

BST Delete

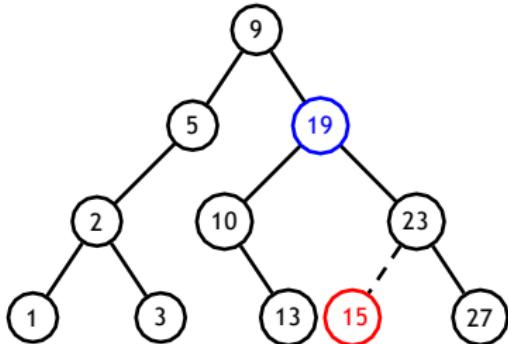
Delete: 15



Case 3: Deleted node has two children

Use the following three-stage procedure:

- ① First, find the node k^l which contains the smallest key in the right subtree.
 - ② Second, exchange the deleted node and node k^l .
 - ③ Third, remove the deleted node from its new position by using either **Case 1** or **Case 2**, depending on whether that node is a leaf (no children) or has a single child.



Overview

- 1 Overview
- 2 Decrease-by-a-Constant: Insertion Sort
- 3 Decrease-by-a-Constant: Topological Sorting
- 4 Decrease-by-a-Constant-Factor Algorithms
- 5 Variable-Size Decrease Algorithms & Binary Search Trees
- 6 Case Study
- 7 Summary

Case Study - Problem

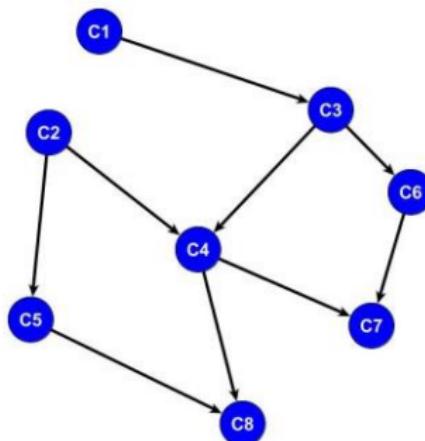
Case Study Problem

Easy-as-123 University has used experienced course advisors to build valid study plans for their students. But they want to explore more automated approaches to help their advisors to quickly devise valid plans. Given courses and their pre-requisites, they want a tool that can produce valid study plans that a student can only study a course if they have satisfy the pre-requisites already.

They asked you to help them. How would you approach this problem?

Case Study - Mapping the Problem to a Known Problem

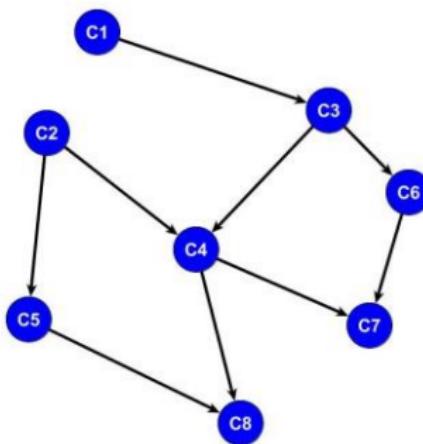
This can be mapped into a graph problem. Each course is a vertex in the graph. Each pre-requisite is a directed edge, from the pre-requisite to the course requiring it.



Case Study - Solving the Problem

A valid plan is a vertex traversal and one that satisfies and respects all the edge directions. This is exactly the properties that a topological sort has.

We can use DFS or source removal algorithm to solve this. Is it possible to have more than one valid plan?



Overview

- 1 Overview
- 2 Decrease-by-a-Constant: Insertion Sort
- 3 Decrease-by-a-Constant: Topological Sorting
- 4 Decrease-by-a-Constant-Factor Algorithms
- 5 Variable-Size Decrease Algorithms & Binary Search Trees
- 6 Case Study
- 7 Summary

Summary

- Introduced the *Decrease-and-conquer* algorithmic approach.
- Decrease-by-a-constant algorithms (Insertion sorting and Topological sort).
- Decrease-by-a-constant-factor algorithms (Binary search, Fake coin).
- Variable-size decrease algorithms (Binary search tree).

Levitin - The design and analysis of algorithms

This week we will be covering the material from Chapter 5.

Learning outcomes:

- Understand the *Divide-and-conquer* algorithmic approach
- Understand and apply merge and quick sort.
- Understand and apply height and traversal operations for BST

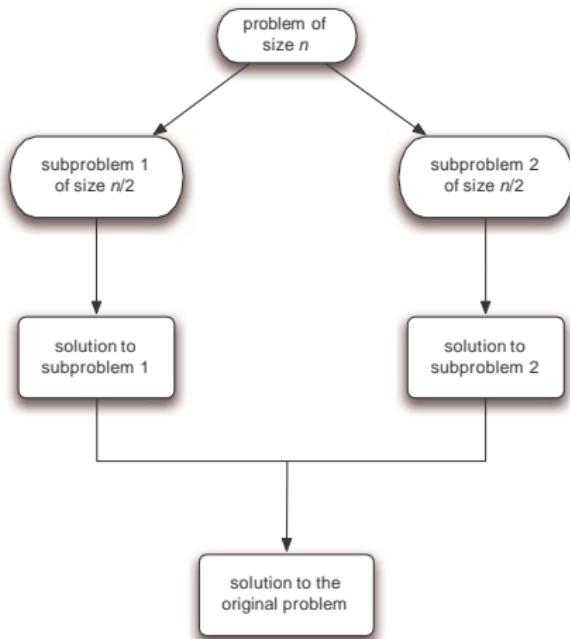
Outline

- ① Problem Overview
- ② Merge Sort
- ③ Quick Sort
- ④ Binary Search Trees
- ⑤ Case Study
- ⑥ Summary

Overview

- ① [Problem Overview](#)
- ② [Merge Sort](#)
- ③ [Quick Sort](#)
- ④ Binary Search Trees
- ⑤ Case Study
- ⑥ Summary

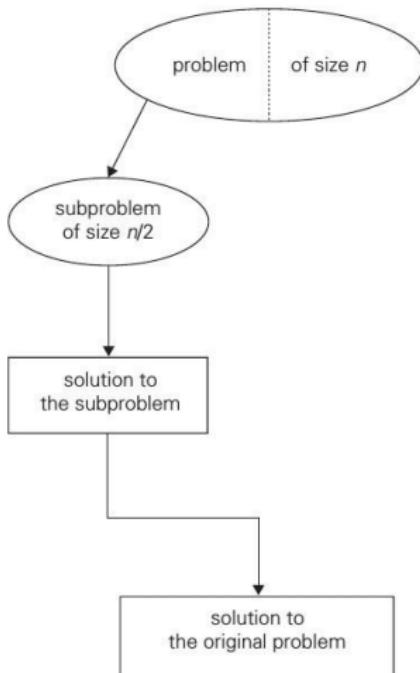
Divide and Conquer



Strategy:

- ➊ Divide the problem instance into smaller subproblems.
- ➋ Solve each subproblem (recursively).
- ➌ Combine smaller solutions to solve the original instance.

Compare with Decrease-by-a-constant-factor



Strategy:

- ① Decrease-by-a-constant-factor algorithms.

Overview

- ① [Problem Overview](#)
- ② [Merge Sort](#)
- ③ [Quick Sort](#)
- ④ Binary Search Trees
- ⑤ Case Study
- ⑥ Summary

Merge Sort

- Idea:
 - Imagine we **recursively divided** an array (we wanted to sort) into halves, until we reach single element partitions.

Merge Sort

- Idea:
 - Imagine we **recursively divided** an array (we wanted to sort) into halves, until we reach single element partitions.
 - We then **recursively merge** the partitions, where we have a process that maintains sorting after partitions are merged.

Merge Sort

- Idea:
 - Imagine we **recursively divided** an array (we wanted to sort) into halves, until we reach single element partitions.
 - We then **recursively merge** the partitions, where we have a process that maintains sorting after partitions are merged.
 - When we finally merge the last two partitions, we have a sorted array.

Merge Sort Example

15	21	1	25	12	6	8	3	5	19	10	18
----	----	---	----	----	---	---	---	---	----	----	----

COMPARES

0

Merge Sort Example

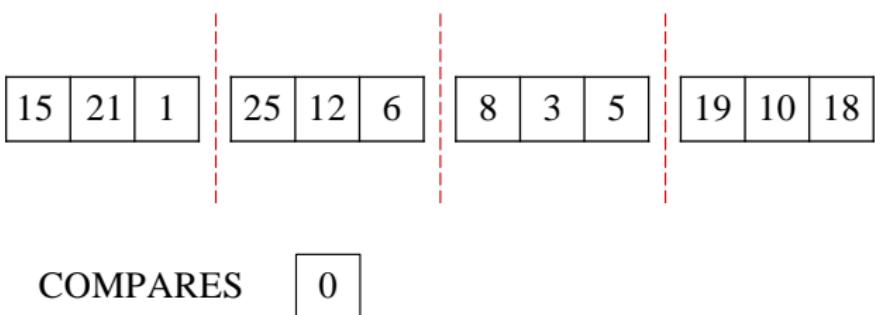
15	21	1	25	12	6
8	3	5	19	10	18



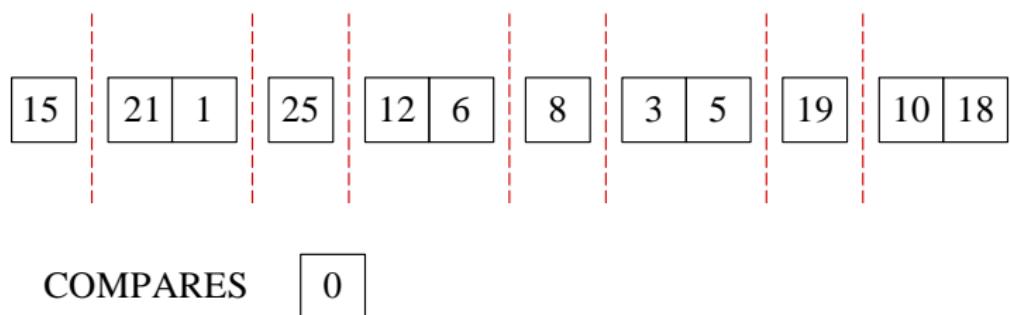
COMPARES

0

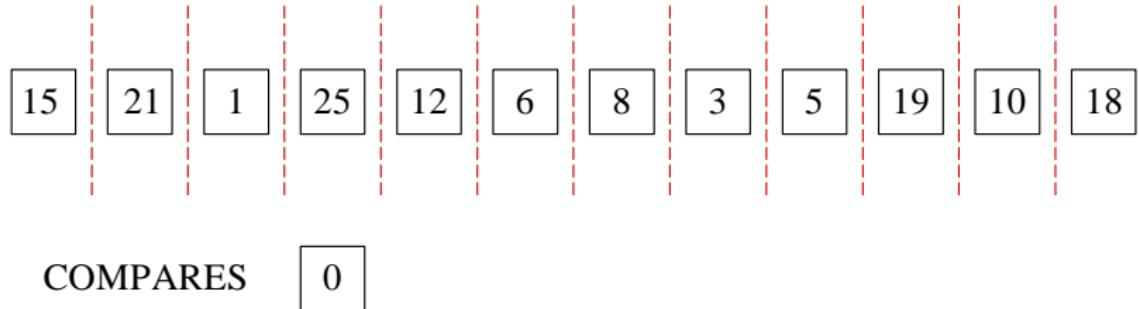
Merge Sort Example



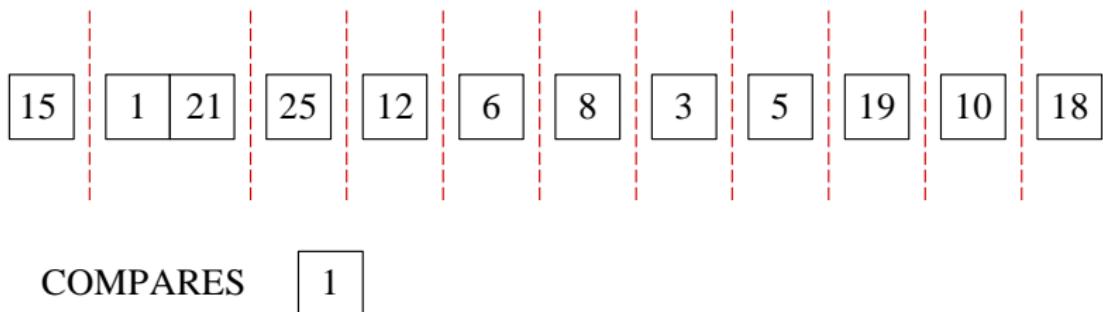
Merge Sort Example



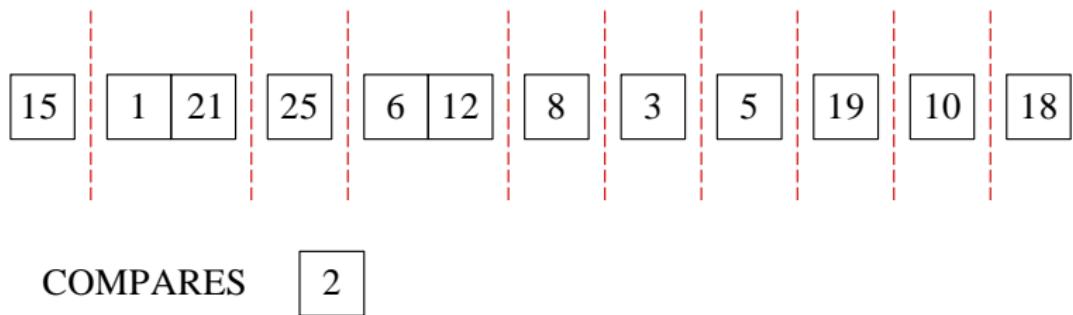
Merge Sort Example



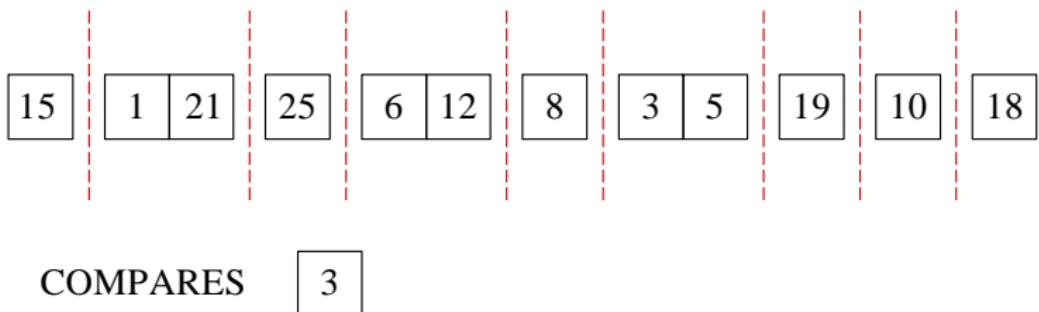
Merge Sort Example



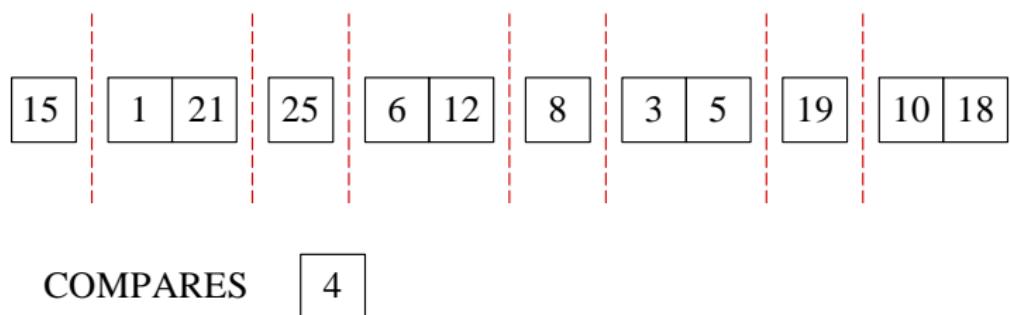
Merge Sort Example



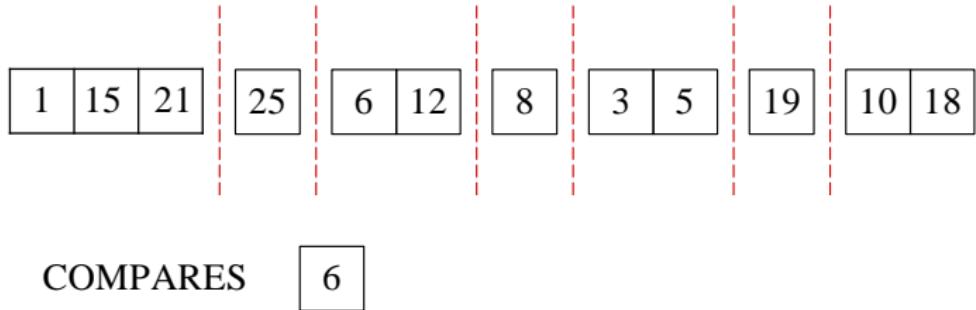
Merge Sort Example



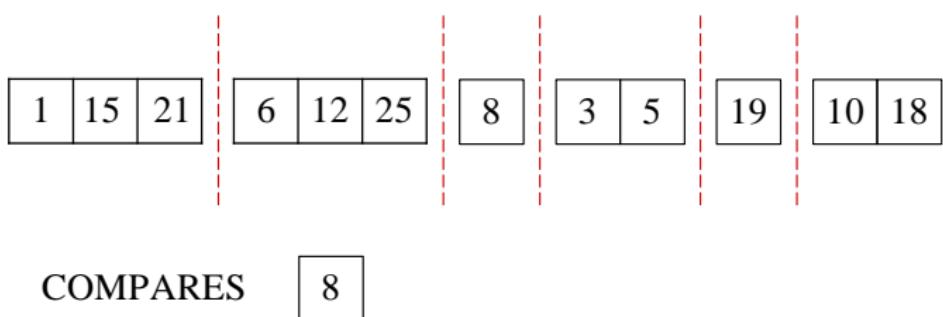
Merge Sort Example



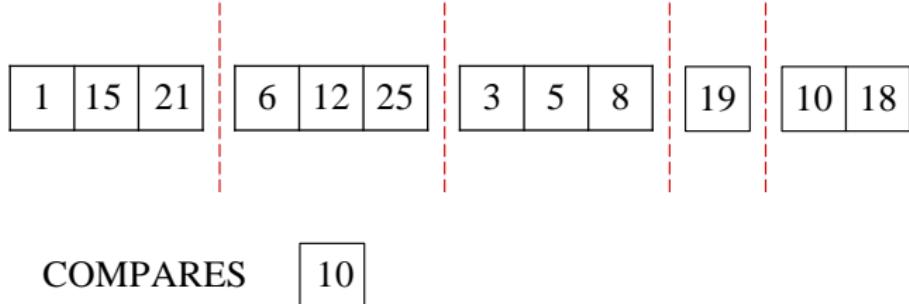
Merge Sort Example



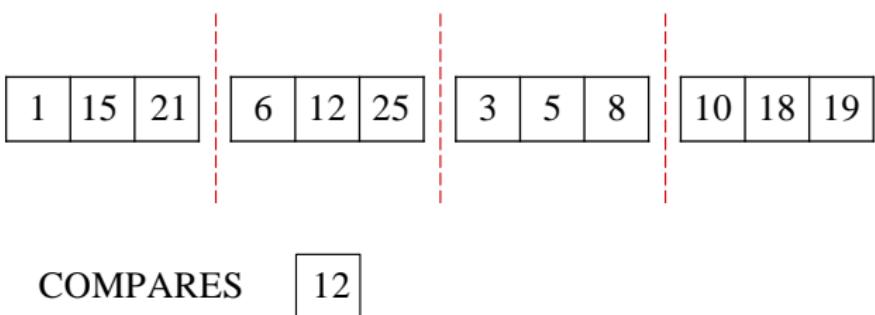
Merge Sort Example



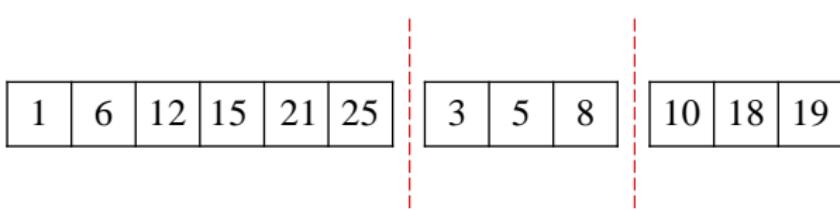
Merge Sort Example



Merge Sort Example



Merge Sort Example



Merge Sort Example

1	6	12	15	21	25
3	5	8	10	18	19



COMPARES

20

Merge Sort Example

1	3	5	6	8	10	12	15	18	19	21	25
---	---	---	---	---	----	----	----	----	----	----	----

COMPARES

30

Merge Sort Example

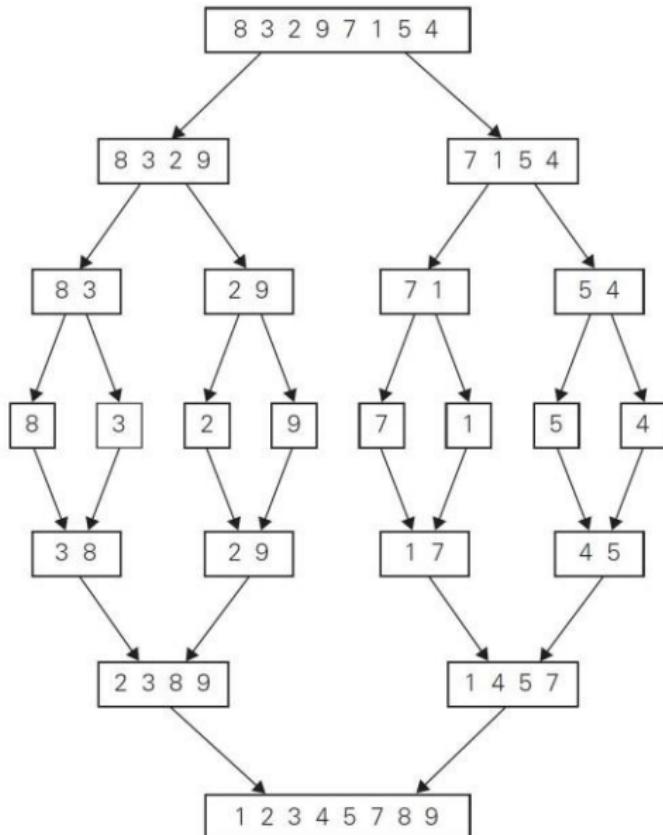


FIGURE 5.2 Example of mergesort operation.

Merge Sort Algorithm

```
ALGORITHM MergeSort (A[0...n - 1])
/*Sort an array using a divide-and-conquer merge sort.*/
/*INPUT : An array A[0 ... n - 1] of orderable elements.*/
/*OUTPUT : An array A[0 ... n - 1] sorted in ascending order.*/
1: if n > 1 then
2:   B = A[0 ... [n/2] - 1] /*B is first half of A */
3:   C = A[[n/2] ... n - 1] /*C is second half of A */
4:   MergeSort (B)
5:   MergeSort (C)
6:   Merge (B, C, A) /*Merge B and C to help sort A */
7: end if
```

Merge in Mergesort

Given two sorted subarrays B, C, want to merge them together to form a sorted A.

Merge in Mergesort

Given two sorted subarrays B, C, want to merge them together to form a sorted A.

Idea:

- ① Consider first element of each subarray, i.e, $B[0]$ and $C[0]$.
Compare them. Whichever one is smaller, copy to $A[0]$, and increment current pointer of subarrays that has smaller element and A.
- ② Repeat until one of subarrays is empty. Then copy the rest of subarray to A.

Merge in Mergesort

```
ALGORITHM Merge ( $B[0 \dots p - 1], C[0 \dots q - 1], A[0 \dots p + q - 1]$ )
/* Merge two sorted arrays into one sorted array. */
/* INPUT : Arrays  $B[0 \dots p - 1]$  and  $C[0 \dots q - 1]$  both sorted. */
/* OUTPUT : Sorted array  $A[0 \dots p + q - 1]$  of the elements of  $B$  and  $C$ . */

1:  $i = 0; j = 0; k = 0$ 
2: while  $i < p$  and  $j < q$  do
3:   if  $B[i] \leq C[j]$  then
4:      $A[k] = B[i]$ 
5:      $i = i + 1$ 
6:   else
7:      $A[k] = C[j]$ 
8:      $j = j + 1$ 
9:   end if
10:   $k = k + 1$ 
11: end while
12: if  $i == p$  then
13:   copy  $C[j \dots q - 1]$  to  $A[k \dots p + q - 1]$ 
14: else
15:   copy  $B[i \dots p - 1]$  to  $A[k \dots p + q - 1]$ 
16: end if
```

Merge Sort Example

Animation of Mergesort

<https://www.youtube.com/watch?v=es2T6KY45cA>

Comments on Merge Sort

- Guarantees $O(n \log n)$ time complexity, regardless of the original distribution of data - this sorting method is **insensitive** to the data distribution.

Comments on Merge Sort

- Guarantees $O(n \log n)$ time complexity, regardless of the original distribution of data - this sorting method is **insensitive** to the data distribution.
- The main drawback in this method is the extra space required for merging two partitions/sub-arrays, e.g. B and C from pseudo-code.

Comments on Merge Sort

- Guarantees $O(n \log n)$ time complexity, regardless of the original distribution of data - this sorting method is **insensitive** to the data distribution.
- The main drawback in this method is the extra space required for merging two partitions/sub-arrays, e.g. B and C from pseudo-code.
- Merge sort is a **stable** sorting method.

Merge Sort Analysis

Lets set up the recurrence relationship.

$$C(n) = 2C(n/2) + C_{merge}(n), \text{ for } n > 1, C(1) = 0.$$

$$C_{merge}(n) = n - 1 \text{ (WHY?)}$$

$$C(n) = 2C(n/2) + n - 1 \text{ for } n > 1, C(1) = 0.$$

Merge Sort Analysis

Lets set up the recurrence relationship.

$$C(n) = 2C(n/2) + C_{\text{merge}}(n), \text{ for } n > 1, C(1) = 0.$$

$$C_{\text{merge}}(n) = n - 1 \text{ (WHY?)}$$

$$C(n) = 2C(n/2) + n - 1 \text{ for } n > 1, C(1) = 0.$$

How do we solve this recursion? There are several ways:

- using backward substitution method.
- using recursion tree (please read textbook, not examinable).
- using the Master Theorem (please read textbook, not examinable).

Merge Sort Analysis

Lets set up the recurrence relationship.

$$C(n) = 2C(n/2) + C_{\text{merge}}(n), \text{ for } n > 1, C(1) = 0.$$

$$C_{\text{merge}}(n) = n - 1 \text{ (WHY?)}$$

$$C(n) = 2C(n/2) + n - 1 \text{ for } n > 1, C(1) = 0.$$

How do we solve this recursion? There are several ways:

- using backward substitution method.
- using recursion tree (please read textbook, not examinable).
- using the Master Theorem (please read textbook, not examinable).

$$C(n) \in O(n \log_2(n))$$

Overview

- ① [Problem Overview](#)
- ② [Merge Sort](#)
- ③ [Quick Sort](#)
- ④ Binary Search Trees
- ⑤ Case Study
- ⑥ Summary

Quick Sort

Motivation:

- Mergesort has consistent behaviour for **all** inputs - what if we seek an algorithm that is fast for the average case?
- Quicksort is such a sorting algorithm, often the best practical choice in terms of efficiency because of its **good performance on the average case**.
- Quick Sort is a *divide and conquer* algorithm.

Quick Sort

Idea:

- ① Select an element from the array for which, we hope, about half the elements will come before and half after in a **sorted** array. Call this element the **pivot**.

Quick Sort

Idea:

- ① Select an element from the array for which, we hope, about half the elements will come before and half after in a **sorted** array. Call this element the **pivot**.
- ② Partition the array so that all elements with a **value** less than the **pivot** are in one subarray, and **larger** elements come in the other subarray.

Quick Sort

Idea:

- ① Select an element from the array for which, we hope, about half the elements will come before and half after in a **sorted** array. Call this element the **pivot**.
- ② Partition the array so that all elements with a **value** less than the **pivot** are in one subarray, and **larger** elements come in the other subarray.
- ③ Swap pivot into position of array that is between the partitions.

Quick Sort

Idea:

- ① Select an element from the array for which, we hope, about half the elements will come before and half after in a **sorted** array. Call this element the **pivot**.
- ② Partition the array so that all elements with a **value** less than the **pivot** are in one subarray, and **larger** elements come in the other subarray.
- ③ Swap pivot into position of array that is between the partitions.
- ④ Recursively apply the same procedure on the two subarrays separately.

Quick Sort

Idea:

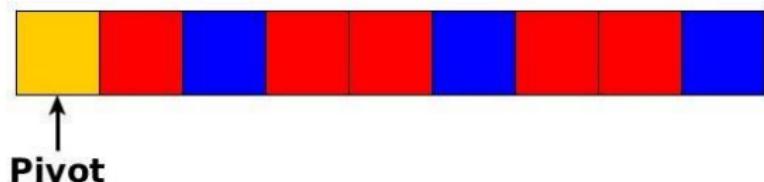
- ① Select an element from the array for which, we hope, about half the elements will come before and half after in a sorted array. Call this element the **pivot**.
- ② Partition the array so that all elements with a **value** less than the **pivot** are in one subarray, and **larger** elements come in the other subarray.
- ③ Swap pivot into position of array that is between the partitions.
- ④ Recursively apply the same procedure on the two subarrays separately.
- ⑤ Terminate when only subarrays are of one element.
- ⑥ When terminate, because we do things in-place, the resulting array is sorted.

Quick Sort

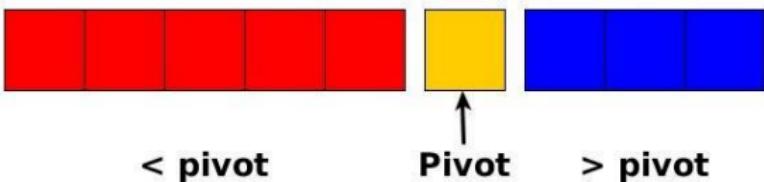
Initial:



Select Pivot:



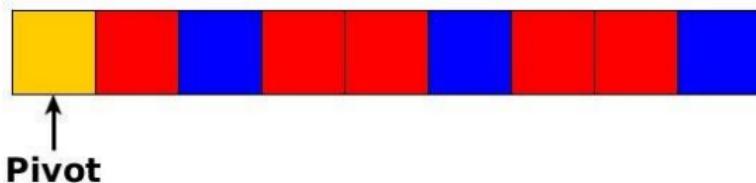
Partition array:



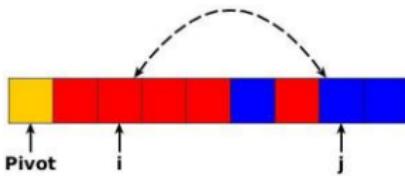
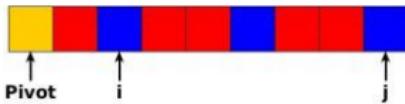
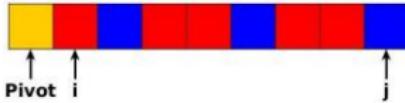
Quick Sort (Hoare partition scheme)

How to efficiently partition the elements less than pivot on one side, greater than partition on the other?

We basically run two linear scans from the front and back of the array, and swap elements that are out of place (i.e., will be on the wrong side of the eventual place of pivot)



Quick Sort (Hoare partition scheme)



Quick Sort

```
ALGORITHM QuickSort ( $A[l \dots r]$ )
/* Sort a subarray using by quicksort. */
/* INPUT : A subarray  $A[l \dots r]$  of  $A[0 \dots n - 1]$ , defined by its left
and right indices  $l$  and  $r$ . */
/* OUTPUT : A subarray  $A[l \dots r]$  sorted in ascending order. */

1: if  $l < r$  then
2:   /*  $s$  is the index to split array. */
3:    $s = QPartition(A[l \dots r])$ 
4:   QuickSort( $A[l \dots s - 1]$ )
5:   QuickSort( $A[s + 1 \dots r]$ )
6: end if
```

Quick Sort

```
ALGORITHM QPartition ( $A[l \dots r]$ )
/* Partition a subarray using the first element as a pivot. */
/* INPUT : A subarray  $A[l \dots r]$  of  $A[0 \dots n - 1]$ , defined by its left
and right indices  $l$  and  $r$ , where ( $l < r$ ). */
/* OUTPUT : A partition of  $A[l \dots r]$ , along with the index of the split position. */

1:  $p = A[l]$ 
2:  $i = l; j = r + 1$ 
3: repeat
4:   do
5:      $i = i + 1$ 
6:     while  $A[i] < p$ 
7:   do
8:      $j = j - 1$ 
9:     while  $A[j] > p$ 
10:    swap( $A[i], A[j]$ )
11: until  $i \geq j$ 
12: swap( $A[i], A[j]$ ) //we need to reverse the last swap, which is incorrect when  $i$  and  $j$  could
   cross each other
13: swap( $A[l], A[j]$ )
14: return  $j$ 
```

Quick Sort Example

15	21	1	25	12	6	8	3	5	19	10	18
----	----	---	----	----	---	---	---	---	----	----	----

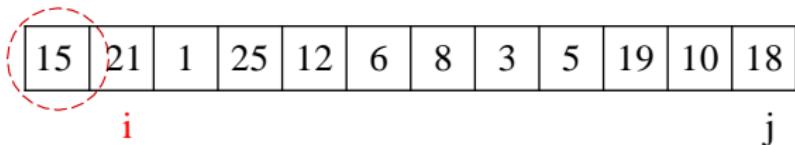
COMPARES

0

SWAPS

0

Quick Sort Example



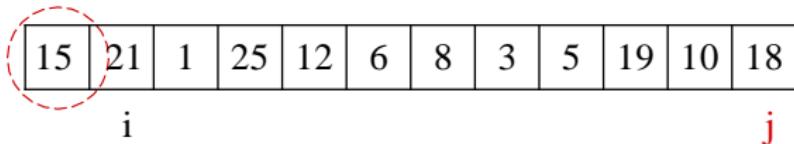
COMPARES

1

SWAPS

0

Quick Sort Example



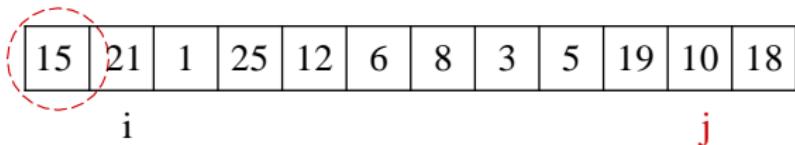
COMPARES

2

SWAPS

0

Quick Sort Example



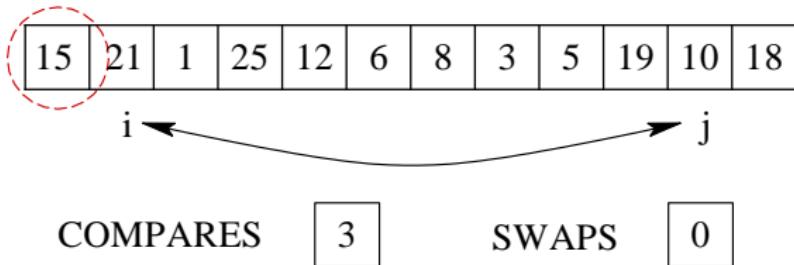
COMPARES

3

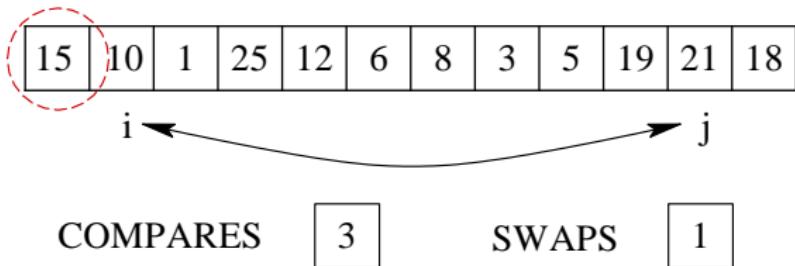
SWAPS

0

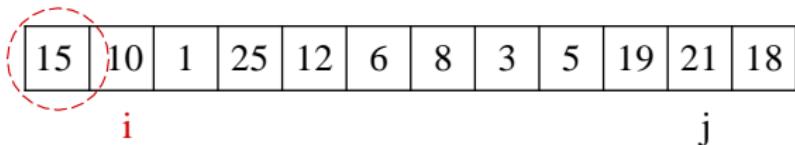
Quick Sort Example



Quick Sort Example



Quick Sort Example



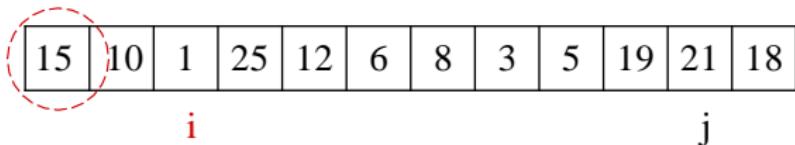
COMPARES

4

SWAPS

1

Quick Sort Example



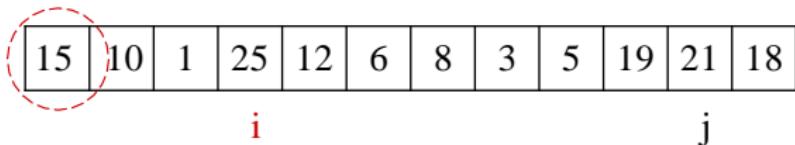
COMPARES

5

SWAPS

1

Quick Sort Example



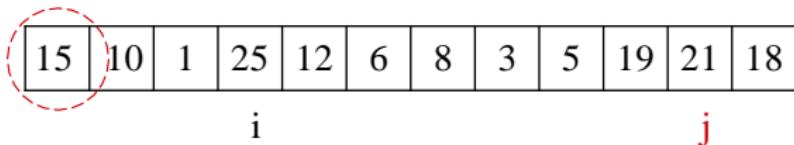
COMPARES

6

SWAPS

1

Quick Sort Example



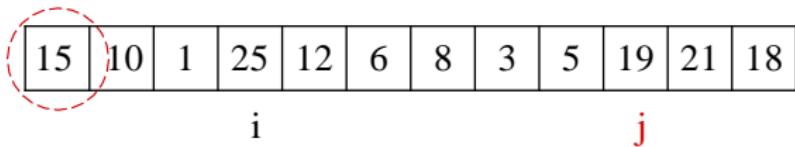
COMPARES

7

SWAPS

1

Quick Sort Example



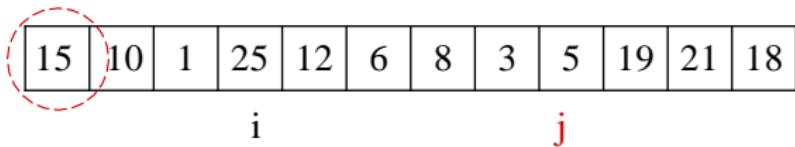
COMPARES

8

SWAPS

1

Quick Sort Example



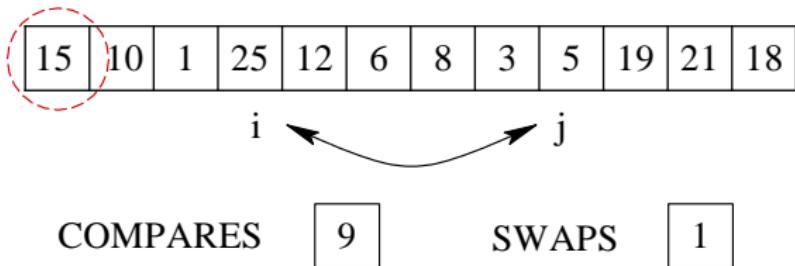
COMPARES

9

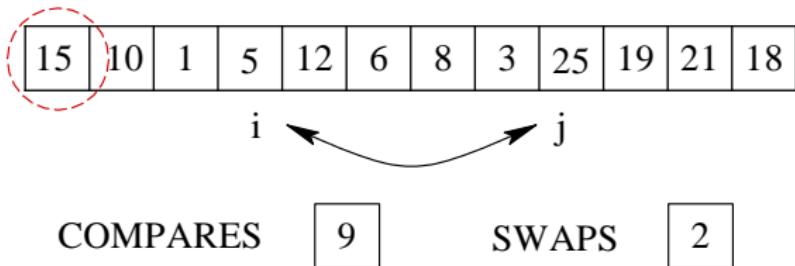
SWAPS

1

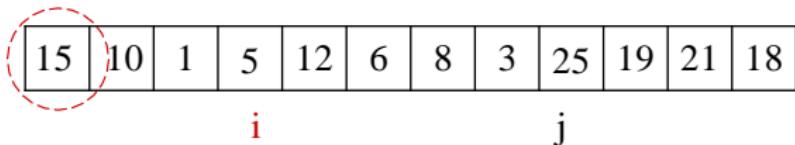
Quick Sort Example



Quick Sort Example



Quick Sort Example



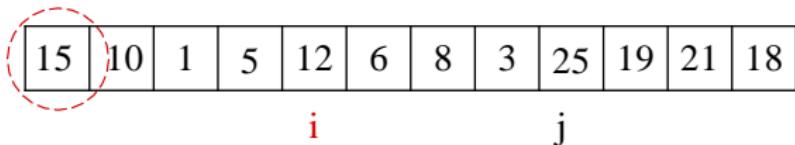
COMPARES

10

SWAPS

2

Quick Sort Example



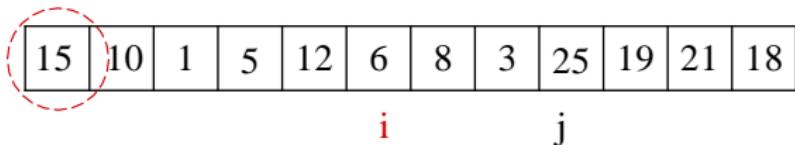
COMPARES

11

SWAPS

2

Quick Sort Example



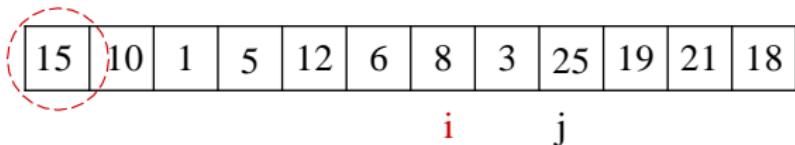
COMPARES

12

SWAPS

2

Quick Sort Example



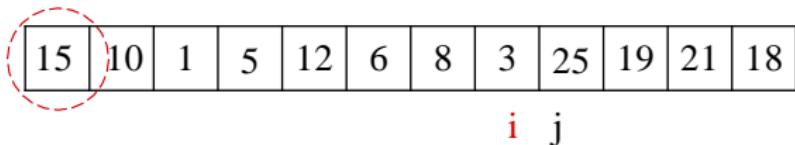
COMPARES

13

SWAPS

2

Quick Sort Example



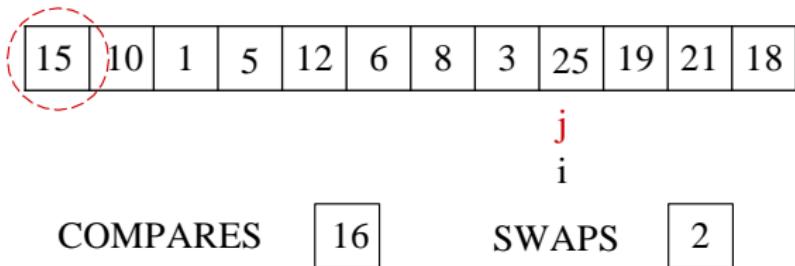
COMPARES

14

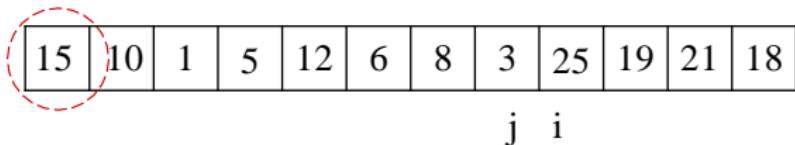
SWAPS

2

Quick Sort Example



Quick Sort Example



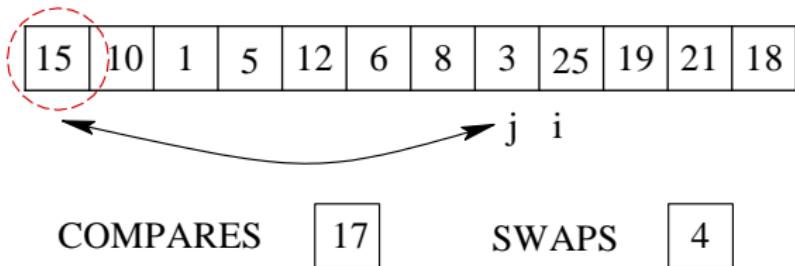
COMPARES

17

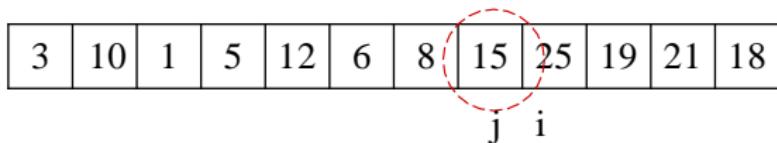
SWAPS

2

Quick Sort Example



Quick Sort Example



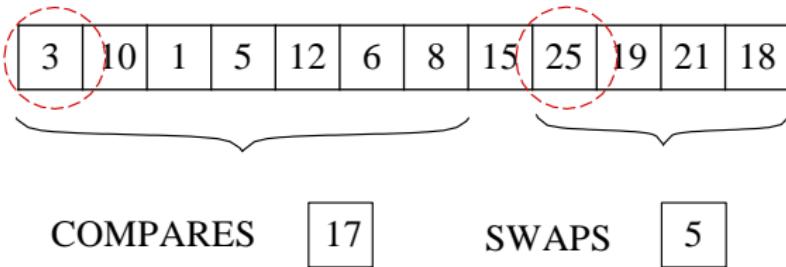
COMPARES

17

SWAPS

5

Quick Sort Example



Quick Sort Complexity

① Best Case:

- Occurs when the pivot repeatedly splits the dataset into **two equal sized** subsets.
- The complexity is $O(n \log_2 n)$.

Quick Sort Complexity

1 Best Case:

- Occurs when the pivot repeatedly splits the dataset into **two equal sized** subsets.
- The complexity is $O(n \log_2 n)$.

2 Worst Case:

- If the pivot is chosen poorly, one of the partitions may be empty, and the other reduced by only one element.
- Then the quick sort is slower than brute-force sorting (due to partitioning overheads).
- The complexity is $(n + 1) + n + (n - 1) + \dots + 1 \approx n^2/2 \in O(n^2)$.
- Occurs when array is **already sorted** or **reverse order sorted**.

Quick Sort Complexity

1 Best Case:

- Occurs when the pivot repeatedly splits the dataset into **two equal sized** subsets.
- The complexity is $O(n \log_2 n)$.

2 Worst Case:

- If the pivot is chosen poorly, one of the partitions may be empty, and the other reduced by only one element.
- Then the quick sort is slower than brute-force sorting (due to partitioning overheads).
- The complexity is $(n + 1) + n + (n - 1) + \dots + 1 \approx n^2/2 \in O(n^2)$.
- Occurs when array is **already sorted** or **reverse order sorted**.

3 Average case:

- Number of comparisons is $\approx 1.39n \log n$.
- 39% more comparisons than merge sort.
- But faster than merge sort in practice because of lower cost of other high-frequency operations, and uses considerably less space (no need to copy to temporary arrays).

Quick Sort - Pivots

Choosing a pivot:

- First or last element: worst case appears for already sorted or reverse sorted arrays (as we saw last slide).

Quick Sort - Pivots

Choosing a pivot:

- First or last element: worst case appears for already sorted or reverse sorted arrays (as we saw last slide).
- Median of three: requires extra compares but generally avoids worst case.

Quick Sort - Pivots

Choosing a pivot:

- First or last element: worst case appears for already sorted or reverse sorted arrays (as we saw last slide).
- Median of three: requires extra compares but generally avoids worst case.
- Random element: Poor cases are very unlikely, but efficient implementations can be non-trivial.

Quick Sort - Pivots

Choosing a pivot:

- First or last element: worst case appears for already sorted or reverse sorted arrays (as we saw last slide).
- Median of three: requires extra compares but generally avoids worst case.
- Random element: Poor cases are very unlikely, but efficient implementations can be non-trivial.
- As long as selected pivot is not always the worst case, Quick sort on average performs well.

For the curious: Watch the Google Talk: 'Three Beautiful Quicksorts' by Jon Bentley (after lectures)

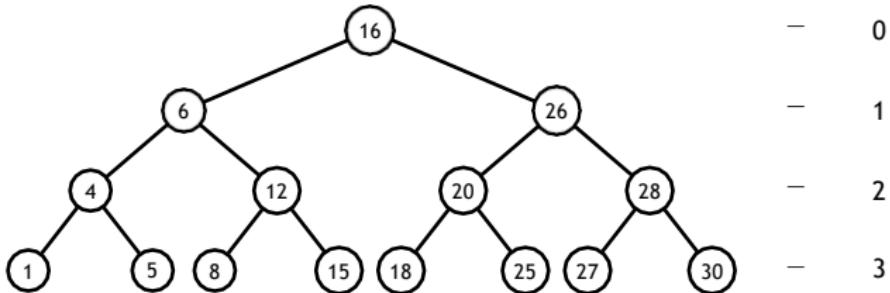
Quick Sort Properties

- Quick sort does sorting in place (don't need additional copying and temporary arrays).
- Quick Sort is *not* a stable sorting method.

Overview

- 1 [Problem Overview](#)
- 2 [Merge Sort](#)
- 3 [Quick Sort](#)
- 4 Binary Search Trees
- 5 Case Study
- 6 Summary

BST Height



The **height** of a tree is the length of the path from the root to the deepest node in the tree. A rooted tree with only 1 node has a height of zero.

BST Height

ALGORITHM **BSTHeight** (T)

//Recursively compute the height of a binary search tree.

//INPUT : A binary tree T .

//OUTPUT : Height of T .

1: if $T = \emptyset$ then

2: return -1

3: else

4: return max (**BSTHeight** (T_L), **BSTHeight** (T_R)) +1

5: end if

BST Height

```
ALGORITHM BSTHeight ( $T$ )
//Recursively compute the height of a binary search tree.
//INPUT : A binary tree  $T$ .
//OUTPUT : Height of  $T$ .
1: if  $T = \emptyset$  then
2:     return -1
3: else
4:     return max (BSTHeight ( $T_L$ ), BSTHeight ( $T_R$ )) + 1
5: end if
```

The worst case complexity to calculate the height is $\Theta(n)$. Why?

BST Height

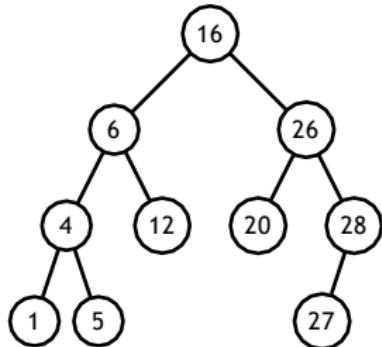
```
ALGORITHM BSTHeight ( $T$ )
//Recursively compute the height of a binary search tree.
//INPUT : A binary tree  $T$ .
//OUTPUT : Height of  $T$ .
1: if  $T = \emptyset$  then
2:     return -1
3: else
4:     return max (BSTHeight ( $T_L$ ), BSTHeight ( $T_R$ )) + 1
5: end if
```

The worst case complexity to calculate the height is $\Theta(n)$. Why?
What is the best case?

BST Traversal

- Given a tree, systematically process every node in that tree.
- For binary trees, we have up to two children for each node, and therefore we have three basic orders in which we can process the tree.
- **preorder**: The root is visited before the left and right subtrees are visited (in that order).
- **inorder**: The root is visited after visiting its left subtree but before visiting the right subtree.
- **postorder**: The root is visited after visiting the left and right subtrees (in that order).
- All of these methods are $\Theta(n)$.

Inorder BST Traversal

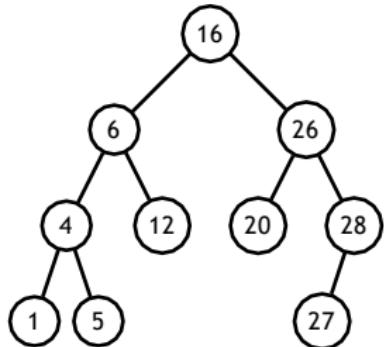


ALGORITHM Inorder(T)

//An In-order traversal of a binary tree.
//INPUT : A binary tree T (with labeled vertices).
//OUTPUT : Node labels listed in-order.

- 1: if $T \neq \emptyset$ then
 - 2: Inorder(T_L) // T_L is the left sub-tree.
 - 3: process node label
 - 4: Inorder(T_R) // T_R is the right sub-tree.
 - 5: end if
-

Inorder BST Traversal



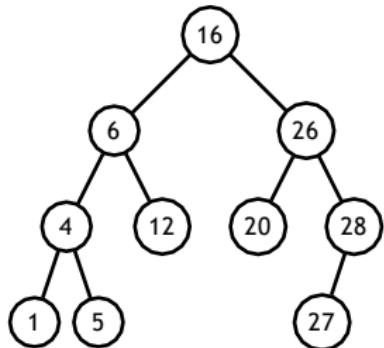
ALGORITHM **Inorder(T)**

//An In-order traversal of a binary tree.
//INPUT : A binary tree T (with labeled vertices).
//OUTPUT : Node labels listed in-order.

- 1: **if** $T \neq \emptyset$ **then**
- 2: **Inorder(T_L)** // T_L is the left sub-tree.
- 3: **process node label**
- 4: **Inorder(T_R)** // T_R is the right sub-tree.
- 5: **end if**

OUTPUT : 1,4,5,6,12,16,20,26,27,28

Pre-order BST Traversal

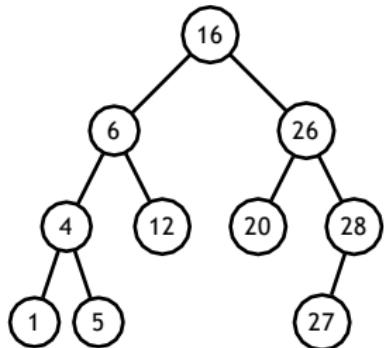


ALGORITHM **Preorder(T)**

//A Pre-order traversal of a binary tree.
//INPUT : A binary tree T (with labeled vertices).
//OUTPUT : Node labels listed in preorder.

- 1: if $T \neq \emptyset$ then
- 2: process *node label*
- 3: Preorder(T_L) // T_L is the left sub-tree.
- 4: Preorder(T_R) // T_R is the right sub-tree.
- 5: end if

Pre-order BST Traversal

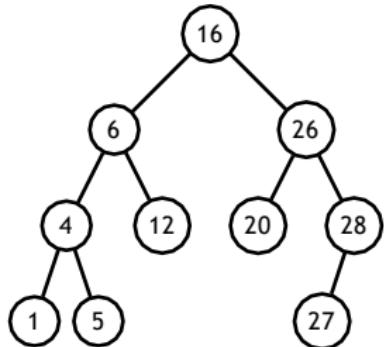


```
ALGORITHM Preorder( $T$ )
//A Pre-order traversal of a binary tree.
//INPUT : A binary tree  $T$  (with labeled vertices).
//OUTPUT : Node labels listed in preorder.

1: if  $T \neq \emptyset$  then
2:   process node label
3:   Preorder( $T_L$ ) //  $T_L$  is the left sub-tree.
4:   Preorder( $T_R$ ) //  $T_R$  is the right sub-tree.
5: end if
```

OUTPUT : 16,6,4,1,5,12,26,20,28,27

Postorder BST Traversal

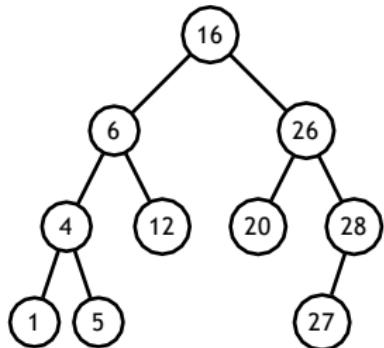


ALGORITHM Postorder(T)

//A Post-order traversal of a binary tree.
//INPUT : A binary tree T (with labeled vertices).
//OUTPUT : Node labels listed in postorder.

- 1: **if** $T \neq \emptyset$ **then**
- 2: **Postorder**(T_L) // T_L is the left sub-tree.
- 3: **Postorder**(T_R) // T_R is the right sub-tree.
- 4: **process node label**
- 5: **end if**

Postorder BST Traversal



ALGORITHM Postorder(T)

//A Post-order traversal of a binary tree.
//INPUT : A binary tree T (with labeled vertices).
//OUTPUT : Node labels listed in postorder.

- 1: **if** $T \neq \emptyset$ **then**
- 2: **Postorder**(T_L) // T_L is the left sub-tree.
- 3: **Postorder**(T_R) // T_R is the right sub-tree.
- 4: **process node label**
- 5: **end if**

OUTPUT : 1,5,4,12,6,20,27,28,26,16

Overview

- 1 [Problem Overview](#)
- 2 [Merge Sort](#)
- 3 [Quick Sort](#)
- 4 Binary Search Trees
- 5 Case Study
- 6 Summary

Case Study - Problem

Case Study Problem

XYZ Pty. Ltd. manufactures and maintains a set of golf buggies around the world. Each buggy is identified by a unique ID that encodes when it was made and what batch it came from, and buggies that have IDs in a particular range are made in the same period and have same features in them. XYZ wants to build a custom database that maintains what is the current set of active buggies they are servicing around the world. They want to quickly retrieve buggies by their IDs and also quickly retrieve all buggies in a particular ID range. They ask for your help.

Case Study - Mapping the Problem to a Known Data Structure

The problem has two requirements:

- Retrieval of buggy by ID
- Retrieval of a set of buggies that have the queries ID range

Which data structure to use?

Case Study - Mapping the Problem to a Known Data Structure

The problem has two requirements:

- Retrieval of buggy by ID
- Retrieval of a set of buggies that have the queries ID range

Which data structure to use?

Binary search tree!

Case Study - Solving the Problem

Binary search tree:

- Use ID as the key.
- Use BST search to locate individual key.

Range queries?

Case Study - Solving the Problem

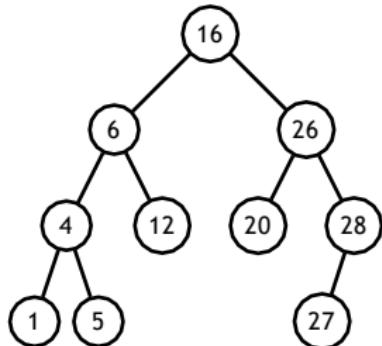
Binary search tree:

- Use ID as the key.
- Use BST search to locate individual key.

Range queries?

Variation of in-order traversal!

Case Study - Solving the Problem



ALGORITHM **RangeQ(T, lr, ur)**

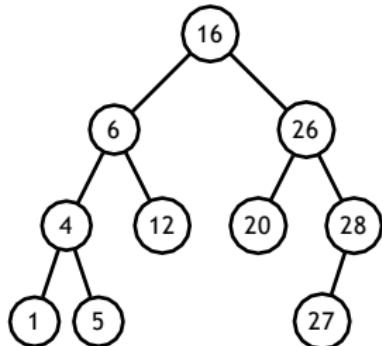
//An range query of a binary search tree.

// INPUT : A search binary tree T (with labeled vertices), range $[lr,ur]$.

//OUTPUT : Node labels that are within the range.

- 1: **if** $T \neq \emptyset$ **then**
 - 2: **If** $node\ label > lr$ **then** **RangeQ**(T_L, lr, ur)
 - 3: **If** $lr < node\ label < ur$ **then print** $node\ label$
 - 4: **If** $node\ label < ur$ **then** **RangeQ**(T_R, lr, ur)
 - 5: **end if**
-

Case Study - Solving the Problem



ALGORITHM **RangeQ(T, lr, ur)**

//An range query of a binary search tree.

// INPUT : A search binary tree T (with labeled vertices), range $[lr,ur]$.

//OUTPUT : Node labels that are within the range.

- 1: **if** $T \neq \emptyset$ **then**
 - 2: **If** $node\ label > lr$ **then** **RangeQ**(T_L, lr, ur)
 - 3: **If** $lr < node\ label < ur$ **then print** $node\ label$
 - 4: **If** $node\ label < ur$ **then** **RangeQ**(T_R, lr, ur)
 - 5: **end if**
-

QUERY : [10,23]

OUTPUT : 12,16,20

Overview

- ① [Problem Overview](#)
- ② [Merge Sort](#)
- ③ [Quick Sort](#)
- ④ Binary Search Trees
- ⑤ Case Study
- ⑥ Summary

Summary

- Introduced the *Divide-and-conquer* algorithmic approach.
- Sorting - merge and quick sort.
- Height and tree traversal for BST.
- Case study.

Levitin - The design and analysis of algorithms

This week we will be covering the material from Chapter 6.

Learning outcomes:

- Understand the *Transform-and-conquer* algorithmic approach.
- Understand and apply:
 - Instance simplification: Pre-sorting & Balanced search trees (AVL trees)
 - Representation change: Balanced search trees (2-3 trees) & heaps and heapsort
 - Problem reduction

Outline

- ① Overview
- ② Presorting
- ③ Balanced Search Trees: AVL Trees
- ④ Balanced Search Trees: 2-3 Trees
- ⑤ Heaps and Heapsort
- ⑥ Problem Reduction
- ⑦ Summary

Overview

① Overview

② Presorting

③ Balanced Search Trees: AVL Trees

④ Balanced Search Trees: 2-3 Trees

⑤ Heaps and Heapsort

⑥ Problem Reduction

⑦ Summary

Transform and Conquer

Idea: Some problems are easier/simpler to solve after they are first transformed to another form.

This group of techniques can be broken into the following classifications:

- **instance simplification** - transform to a simpler or more convenient instance of the same problem.
- **representation change** - transform to a different representation of the same problem instance.
- **problem reduction** - transform to an instance of a different problem for which an algorithm is already available.

Overview

1 Overview

2 Presorting

3 Balanced Search Trees: AVL Trees

4 Balanced Search Trees: 2-3 Trees

5 Heaps and Heapsort

6 Problem Reduction

7 Summary

Instance simplification - Presorting

Presorting : Many problems involving arrays are easier when the array is sorted.

General approach of presorting based algorithms:

- ① *Transform:* Sort the array
- ② *Conquer:* Solve the transformed problem instance, taking advantage of the array being sorted

Instance simplification - Presorting

Presorting : Many problems involving arrays are easier when the array is sorted.

General approach of presorting based algorithms:

- ① *Transform:* Sort the array
- ② *Conquer:* Solve the transformed problem instance, taking advantage of the array being sorted

Examples:

- Searching
- Computing the median (selection problem).
- Checking if all elements are distinct (element uniqueness).

Sort efficiency?

Sorting efficiency

Recall many of the sorting algorithms we have seen so far has $O(n \log_2(n))$ worst cases to sort an array of size n .

- The efficiency for all presorting algorithms is bound by the cost of sorting. If we can **amortise** this cost then it is often worth the effort.

Search in a sorted list

Problem : Search for a key k in a array $A[0 \dots n - 1]$.

Search in a sorted list

Problem : Search for a key k in a array $A[0 \dots n - 1]$.

Presorting-based algorithm :

- ① Sort the array using an efficient sorting algorithm.
- ② Apply binary search.

Search in a sorted list

Problem : Search for a key k in a array $A[0 \dots n - 1]$.

Presorting-based algorithm :

- ① Sort the array using an efficient sorting algorithm.
- ② Apply binary search.

Efficiency : $O(n \log n) + O(\log n) = O(n \log n)$

Search in a sorted list

Problem : Search for a key k in a array $A[0 \dots n - 1]$.

Presorting-based algorithm :

- ① Sort the array using an efficient sorting algorithm.
- ② Apply binary search.

Efficiency : $O(n \log n) + O(\log n) = O(n \log n)$

Worst than brute-force sequential search, but if the array is static and search is performed many times (amortised), then it may be worth the extra effort of presorting.

Presorting - Element Uniqueness

Problem : Given an array of unsorted items, determine if all items in the array are distinct.

Presorting - Element Uniqueness

Problem : Given an array of unsorted items, determine if all items in the array are distinct.

Presorting-based algorithm :

- ① Sort the array using an efficient sorting algorithm.
- ② Scan the array to check all pairs of adjacent elements.

Presorting - Element Uniqueness

Problem : Given an array of unsorted items, determine if all items in the array are distinct.

Presorting-based algorithm :

- ① Sort the array using an efficient sorting algorithm.
- ② Scan the array to check all pairs of adjacent elements.

Efficiency : ?

Presorting - Element Uniqueness

Problem : Given an array of unsorted items, determine if all items in the array are distinct.

Presorting-based algorithm :

- ① Sort the array using an efficient sorting algorithm.
- ② Scan the array to check all pairs of adjacent elements.

Efficiency : ? $\Theta(n \log n) + \Theta(n) = \Theta(n \log n)$

Presorting - Element Uniqueness

Problem : Given an array of unsorted items, determine if all items in the array are distinct.

Presorting-based algorithm :

- ① Sort the array using an efficient sorting algorithm.
- ② Scan the array to check all pairs of adjacent elements.

Efficiency : ? $\Theta(n \log n) + \Theta(n) = \Theta(n \log n)$

The brute force algorithm compares all pairs of unsorted elements for an efficiency of $\Theta(n^2)$.

Is presorting worth the effort?

Problem

How many searches in a list are necessary to justify the time spent on presorting an array of 10^3 elements? What about 10^6 elements?
(Assume merge sort and binary search are used)

Is presorting worth the effort?

Problem

How many searches in a list are necessary to justify the time spent on presorting an array of 10^3 elements? What about 10^6 elements?
(Assume merge sort and binary search are used)

- **Presorting:**

- Merge sort uses $n \log_2 n$ comparisons on average.
- Binary search uses $\log_2 n$ comparisons on average.

Is presorting worth the effort?

Problem

How many searches in a list are necessary to justify the time spent on presorting an array of 10^3 elements? What about 10^6 elements?
(Assume merge sort and binary search are used)

- **Presorting:**
 - Merge sort uses $n \log_2 n$ comparisons on average.
 - Binary search uses $\log_2 n$ comparisons on average.
- **Linear search** in an unsorted array uses $n/2$ comparisons on average.

Is presorting worth the effort?

Problem

How many searches in a list are necessary to justify the time spent on presorting an array of 10^3 elements? What about 10^6 elements?
(Assume merge sort and binary search are used)

- **Presorting:**
 - Merge sort uses $n \log_2 n$ comparisons on average.
 - Binary search uses $\log_2 n$ comparisons on average.
- **Linear search** in an unsorted array uses $n/2$ comparisons on average.
- We want to determine how many searches (k) before the presorting is faster than a non-presort method like linear search.
 - Time for presort $(n, k) \leq$ Time for linear (n, k)

Is presorting worth the effort?

Problem

How many searches in a list are necessary to justify the time spent on presorting an array of 10^3 elements? What about 10^6 elements?
(Assume merge sort and binary search are used)

- **Presorting:**
 - Merge sort uses $n \log_2 n$ comparisons on average.
 - Binary search uses $\log_2 n$ comparisons on average.
- **Linear search** in an unsorted array uses $n/2$ comparisons on average.
- We want to determine how many searches (k) before the presorting is faster than a non-presort method like linear search.
 - Time for presort (n, k) \leq Time for linear (n, k)
 - $n \log_2 n + k \log_2 n \leq kn/2$

Is presorting worth the effort?

Problem

How many searches in a list are necessary to justify the time spent on presorting an array of 10^3 elements? What about 10^6 elements?
(Assume merge sort and binary search are used)

- **Presorting:**
 - Merge sort uses $n \log_2 n$ comparisons on average.
 - Binary search uses $\log_2 n$ comparisons on average.
- **Linear search** in an unsorted array uses $n/2$ comparisons on average.
- We want to determine how many searches (k) before the presorting is faster than a non-presort method like linear search.
 - Time for presort (n, k) \leq Time for linear (n, k)
 - $n \log_2 n + k \log_2 n \leq kn/2$
 - $k \geq \frac{n \log_2 n}{n/2 - \log_2 n}$

Is presorting worth the effort?

Problem

How many searches in a list are necessary to justify the time spent on presorting an array of 10^3 elements? What about 10^6 elements?
(Assume merge sort and binary search are used)

- **Presorting:**
 - Merge sort uses $n \log_2 n$ comparisons on average.
 - Binary search uses $\log_2 n$ comparisons on average.
- **Linear search** in an unsorted array uses $n/2$ comparisons on average.
- We want to determine how many searches (k) before the presorting is faster than a non-presort method like linear search.
 - Time for presort (n, k) \leq Time for linear (n, k)
 - $n \log_2 n + k \log_2 n \leq kn/2$
 - $k \geq \frac{n \log_2 n}{n/2 - \log_2 n}$
- substituting $n = 10^3$, $k = 21$.
- substituting $n = 10^6$, $k = 40$. (Is the computation precise?)

Overview

1 Overview

2 Presorting

3 Balanced Search Trees: AVL Trees

4 Balanced Search Trees: 2-3 Trees

5 Heaps and Heapsort

6 Problem Reduction

7 Summary

Balanced Search Trees

- We study two methods to balance search trees. Why balanced trees are desirable?

Balanced Search Trees

- We study two methods to balance search trees. Why balanced trees are desirable?
- **Recall:** The worst-case performance of simple binary trees for its operations is dependent on the **height** of the tree.

Balanced Search Trees

- We study two methods to balance search trees. Why balanced trees are desirable?
- **Recall:** The worst-case performance of simple binary trees for its operations is dependent on the **height** of the tree.
 - As a result, a great deal of research effort has been invested in keeping binary search trees **balanced** and of **minimum height**.

Balanced Search Trees

- We study two methods to balance search trees. Why balanced trees are desirable?
- **Recall:** The worst-case performance of simple binary trees for its operations is dependent on the **height** of the tree.
 - As a result, a great deal of research effort has been invested in keeping binary search trees **balanced** and of **minimum height**.
- Two approaches:
 - **AVL trees:** An example of the **instance simplification** approaches, whereby rebalancing (transformation) is performed when inserting or deleting elements.
 - **2-3 trees:** An example of the **representation change** approaches, by allowing more than one key per node.

Definition

An **AVL tree** is a:

- binary search tree
- for every node, the **difference** between the **heights** of the left and right subtree can **at most** be 1. This includes the root node, hence AVL trees are guaranteed to be almost balanced.

Definition

An **AVL tree** is a:

- binary search tree
- for every node, the **difference** between the **heights** of the left and right subtree can **at most** be 1. This includes the root node, hence AVL trees are guaranteed to be almost balanced.

The height difference is called the **balance factor**.

$\text{balance factor}(\text{node } v) = \text{height}(\text{left subtree of } v) - \text{height}(\text{right subtree of } v)$

Definition

An **AVL tree** is a:

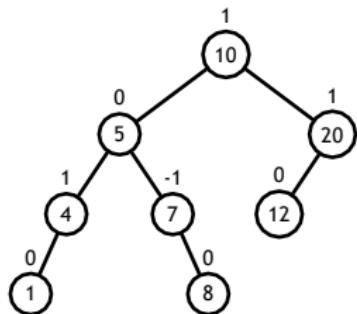
- binary search tree
- for every node, the **difference** between the **heights** of the left and right subtree can **at most** be 1. This includes the root node, hence AVL trees are guaranteed to be almost balanced.

The height difference is called the **balance factor**.

$\text{balance factor}(\text{node } v) = \text{height}(\text{left subtree of } v) - \text{height}(\text{right subtree of } v)$

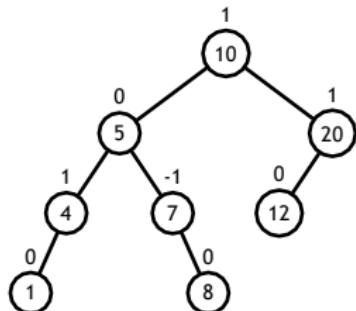
The height of an empty tree is defined as -1 .

AVL Trees

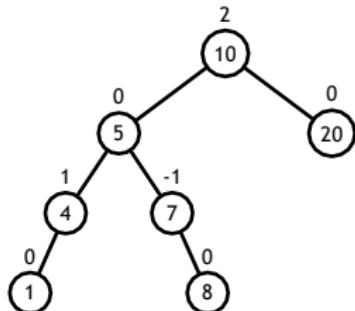


(a) an AVL tree

AVL Trees



(a) an AVL tree



(b) **not** an AVL tree

AVL trees - Insertion

Let w denote the new node to insert.

- ① Insert w using BST insertion.

AVL trees - Insertion

Let w denote the new node to insert.

- ① Insert w using BST insertion.
- ② Update balance factors and check if need to rebalance tree:

AVL trees - Insertion

Let w denote the new node to insert.

- ① Insert w using BST insertion.
- ② Update balance factors and check if need to rebalance tree:
 - From w , updating the balance factor of each node as we traverse back up to root.

AVL trees - Insertion

Let w denote the new node to insert.

- ① Insert w using BST insertion.
- ② Update balance factors and check if need to rebalance tree:
 - From w , updating the balance factor of each node as we traverse back up to root.
- ③ If tree is AVL-tree unbalanced, perform rotation transformations to make avl tree balanced again.

AVL trees - Insertion

Let w denote the new node to insert.

- ① Insert w using BST insertion.
- ② Update balance factors and check if need to rebalance tree:
 - From w , updating the balance factor of each node as we traverse back up to root.
- ③ If tree is AVL-tree unbalanced, perform rotation transformations to make avl tree balanced again.
 - We repair the tree at the node where we first detect imbalance (imbalance node that is closest to the inserted leaf node).

AVL trees - Insertion

Let w denote the new node to insert.

- ① Insert w using BST insertion.
- ② Update balance factors and check if need to rebalance tree:
 - From w , updating the balance factor of each node as we traverse back up to root.
- ③ If tree is AVL-tree unbalanced, perform rotation transformations to make avl tree balanced again.
 - We repair the tree at the node where we first detect imbalance (imbalance node that is closest to the inserted leaf node).
 - May need to update balance factor for the parent node (and their ancestors) of the subtree where the rotations occur.

AVL trees - Insertion Example

Let w denote the new node to insert.

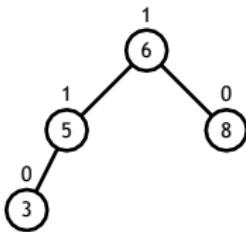
- ① Insert w using BST insertion.
- ② Update balance factors and check if need to rebalance tree:
 - From w , updating the balance factor of each node as we traverse back up to root.

AVL trees - Insertion Example

Let w denote the new node to insert.

- ① Insert w using BST insertion.
- ② Update balance factors and check if need to rebalance tree:
 - From w , updating the balance factor of each node as we traverse back up to root.

Insert 2:

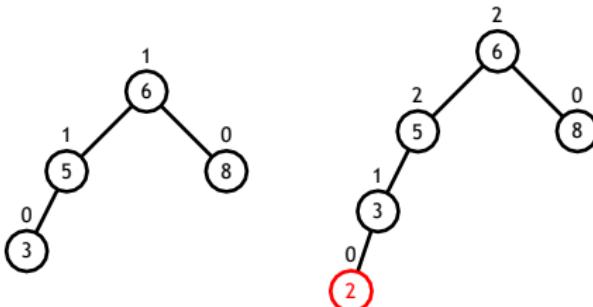


AVL trees - Insertion Example

Let w denote the new node to insert.

- ① Insert w using BST insertion.
- ② Update balance factors and check if need to rebalance tree:
 - From w , updating the balance factor of each node as we traverse back up to root.

Insert 2:

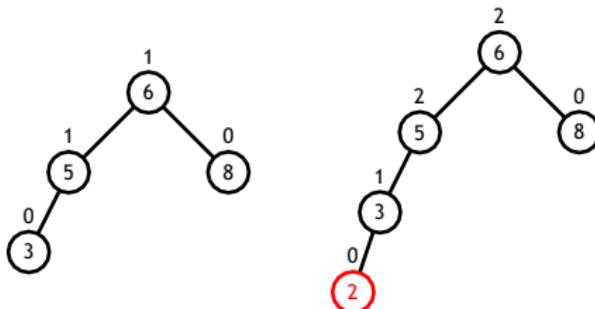


AVL trees - Insertion Example

Let w denote the new node to insert.

- ① Insert w using BST insertion.
- ② Update balance factors and check if need to rebalance tree:
 - From w , updating the balance factor of each node as we traverse back up to root.

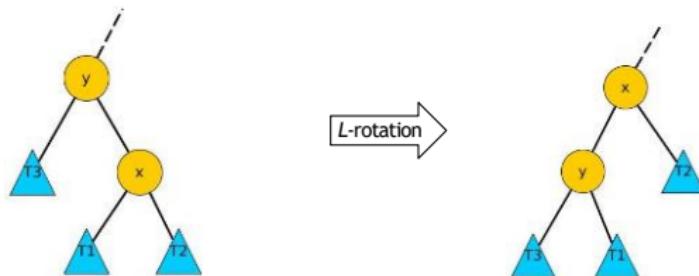
Insert 2:



Next: Introduce the rotation operations, then how to apply them to do rebalancing.

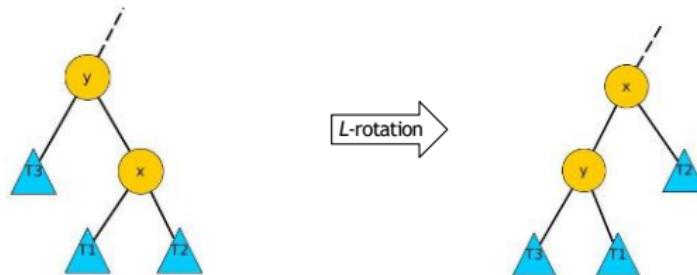
AVL trees - Rotations

Left Rotation:

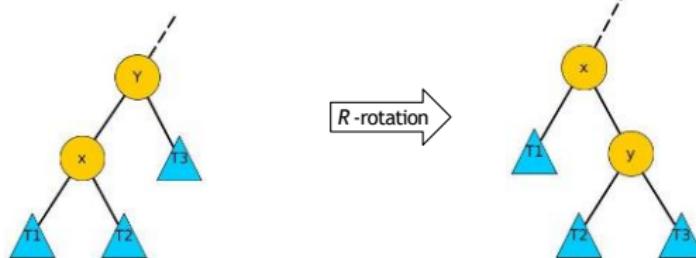


AVL trees - Rotations

Left Rotation:

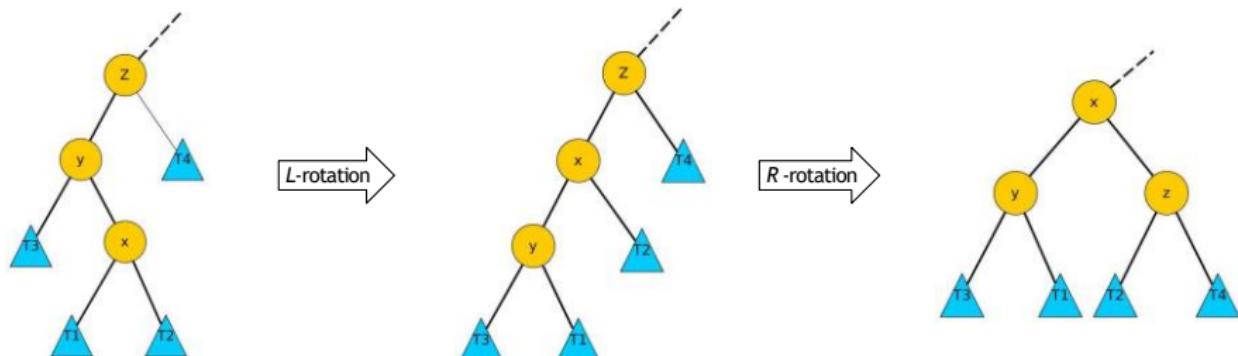


Right Rotation:



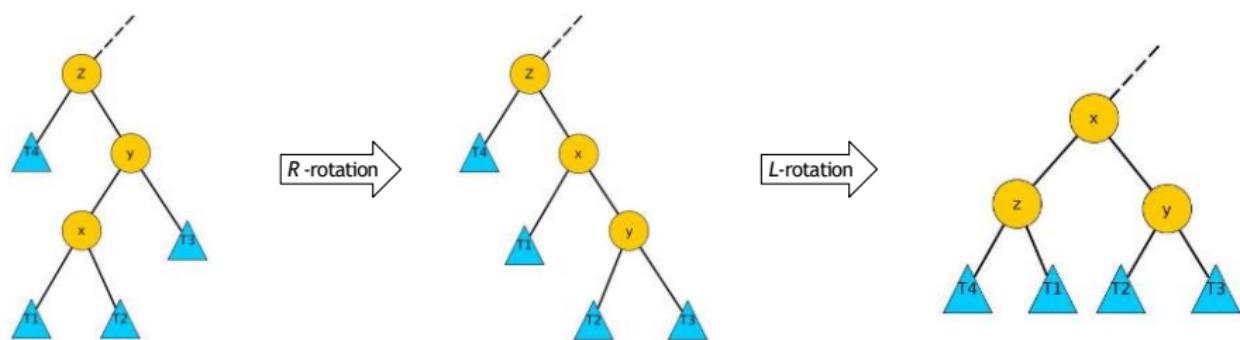
AVL trees - Rotations

LR (left-right) rotation:



AVL trees - Rotations

RL (right-left) rotation:



AVL trees - Insertion

Step 3: If tree is AVL-tree **unbalanced**, perform **rotation transformations** to make avl tree balanced again.

- We repair the tree at the node where we first detect imbalance (imbalance node that is closest to the inserted leaf node).

AVL trees - Insertion

Step 3: If tree is AVL-tree **unbalanced**, perform **rotation transformations** to make avl tree balanced again.

- We repair the tree at the node where we first detect imbalance (imbalance node that is closest to the inserted leaf node).

Four different cases to consider, each of which uses one of the rotations to rebalance tree.

AVL trees - Insertion

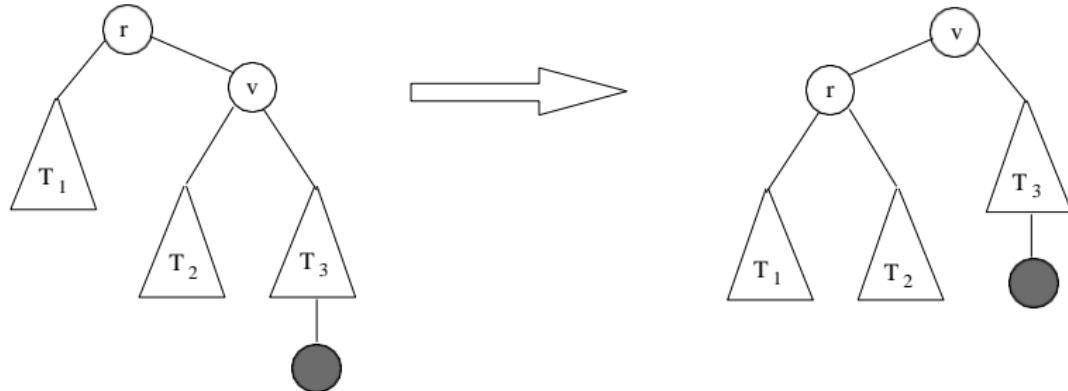
Step 3: If tree is AVL-tree **unbalanced**, perform **rotation transformations** to make avl tree balanced again.

- We repair the tree at the node where we first detect imbalance (imbalance node that is closest to the inserted leaf node).

Four different cases to consider, each of which uses one of the rotations to rebalance tree.

In the following, node r is the first **imbalanced** node (in terms of balance factor) detected when traversing from inserted node.

AVL trees - Insertion Scenarios

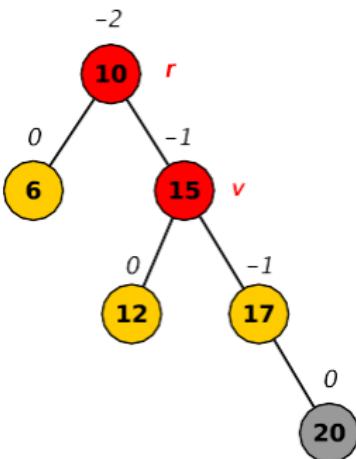


Case 1: Inserted node is in **right** subtree of node v , where:

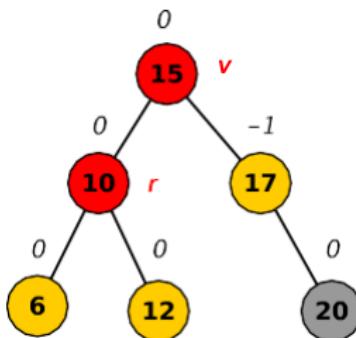
- node v is the **right** child of node r .

Operation: Single *L*-rotation, rotating nodes r and v .

AVL trees - Insertion Scenarios



(a) Before rotation.

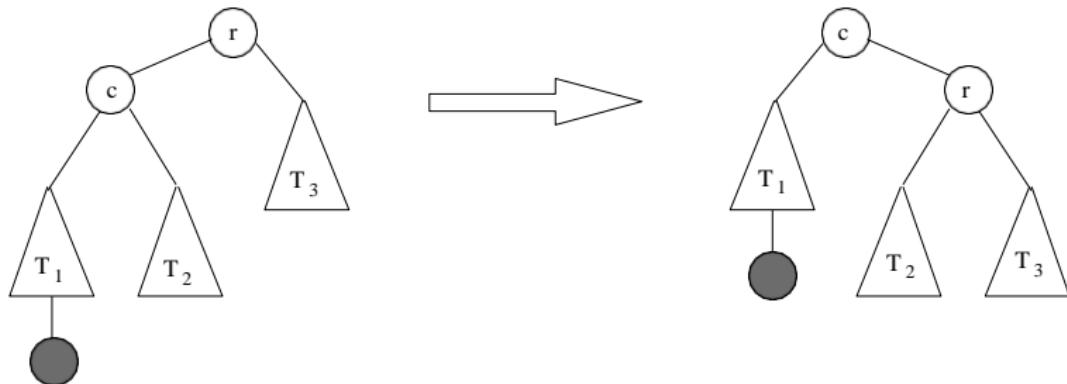


(b) After rotation.

Case 1: Inserted node is in **right** subtree of node v .

Operation: Single *L*-rotation, rotating nodes r and v .

AVL trees - Insertion Scenarios

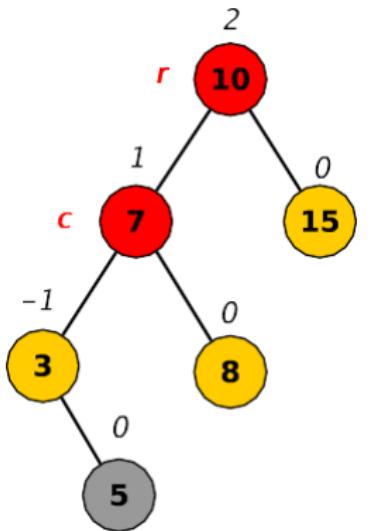


Case 2: Inserted node is in **left** subtree of node *c*, where:

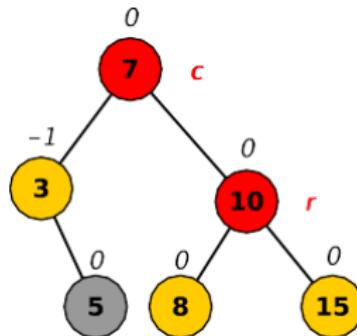
- node *c* is the **left** child of node *r*.

Operation: Single *R*-rotation, rotating nodes *r* and *c*.

AVL trees - Insertion Scenarios



(c) Before rotation.

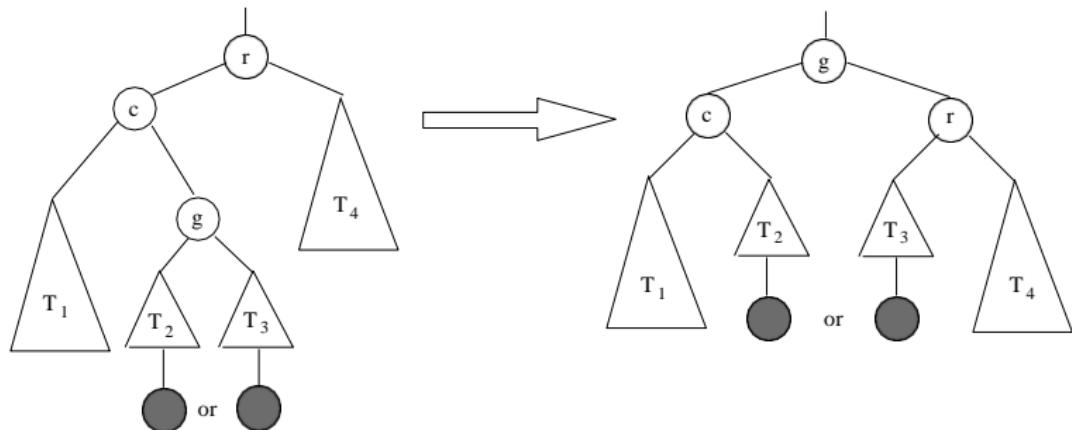


(d) After rotation.

Case 2: Inserted node is in **left** subtree of node **c**.

Operation: Single *R*-rotation, rotating nodes *r* and *c*.

AVL trees - Insertion Scenarios

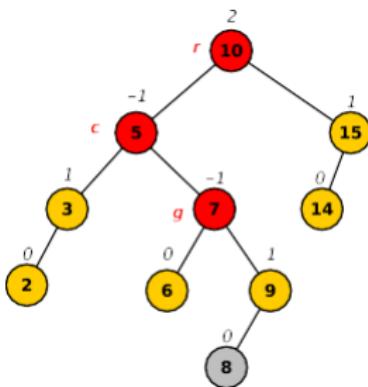


Case 3: Inserted node is in one of the subtrees of node **g** where:

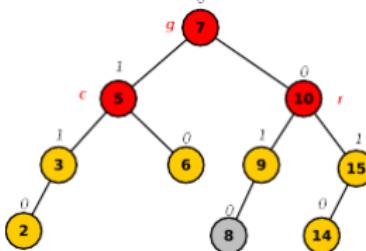
- nodes **r**, **c** and **g** form a **r-left-right** children subtree.

Operation: Double **LR**-rotation, rotating nodes **r**, **c** and **g**.

AVL trees - Insertion Scenarios



(e) Before rotation.

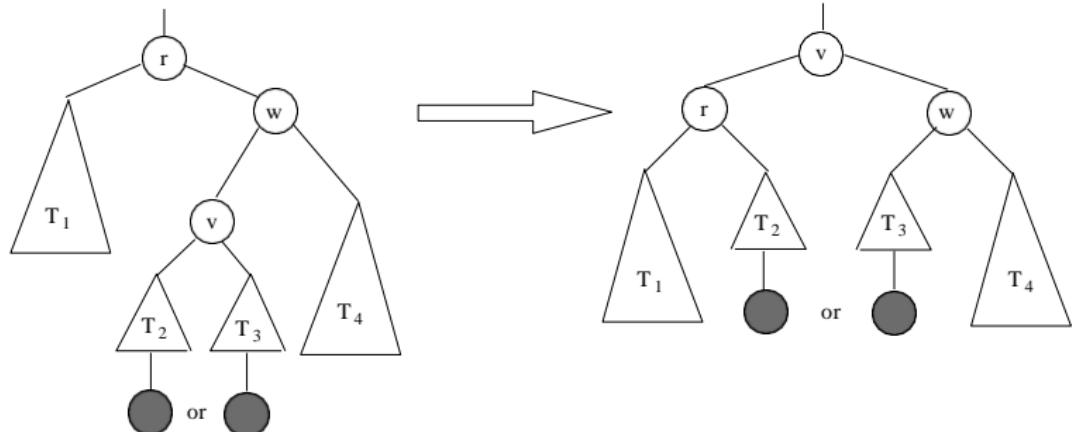


(f) After rotation.

Case 3: Inserted node is in one of the subtrees of node g .

Operation: Double LR -rotation, rotating nodes r , c and g .

AVL trees - Insertion Scenarios

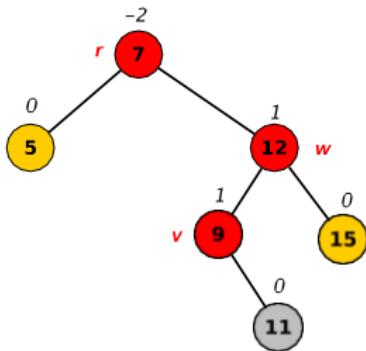


Case 4: Inserted node is in one of the subtrees of node v where:

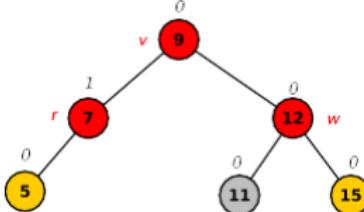
- nodes r , w and v form a **r -right-left** children subtree.

Operation: Double *RL*-rotation, rotating nodes r , w and v .

AVL trees - Insertion Scenarios



(g) Before rotation.



(h) After rotation.

Case 4: Inserted node is in one of the subtrees of node v.

Operation: Double *RL*-rotation, rotating nodes r, w and v.

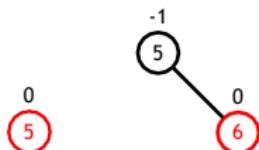
AVL trees Insertion Example

Construct an AVL tree for the sequence 5, 6, 8, 3, 2, 4, 7

0
5

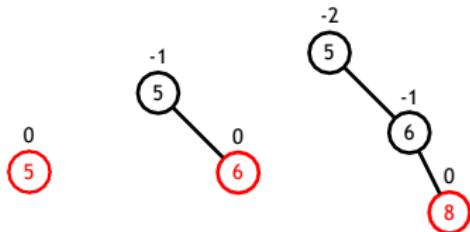
AVL trees Insertion Example

Construct an AVL tree for the sequence 5, 6, 8, 3, 2, 4, 7



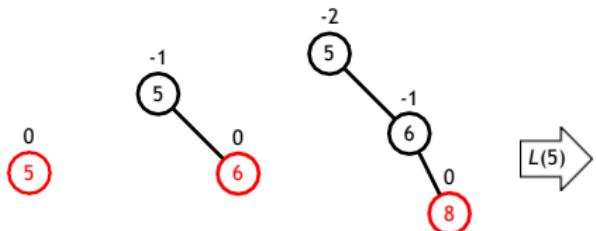
AVL trees Insertion Example

Construct an AVL tree for the sequence 5, 6, 8, 3, 2, 4, 7



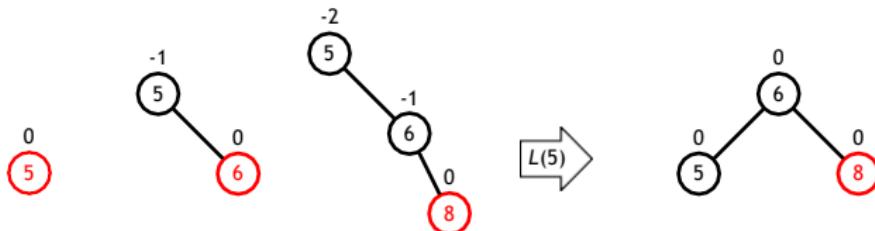
AVL trees Insertion Example

Construct an AVL tree for the sequence 5, 6, 8, 3, 2, 4, 7



AVL trees Insertion Example

Construct an AVL tree for the sequence 5, 6, 8, 3, 2, 4, 7



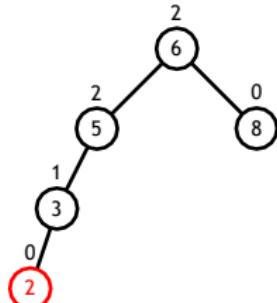
AVL trees Insertion Example

Construct an AVL tree for the sequence 5, 6, 8, 3, 2, 4, 7



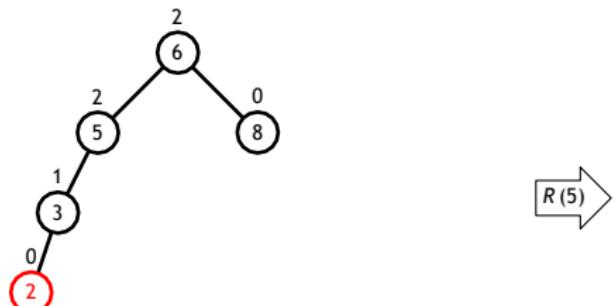
AVL trees Insertion Example

Construct an AVL tree for the sequence 5, 6, 8, 3, 2, 4, 7



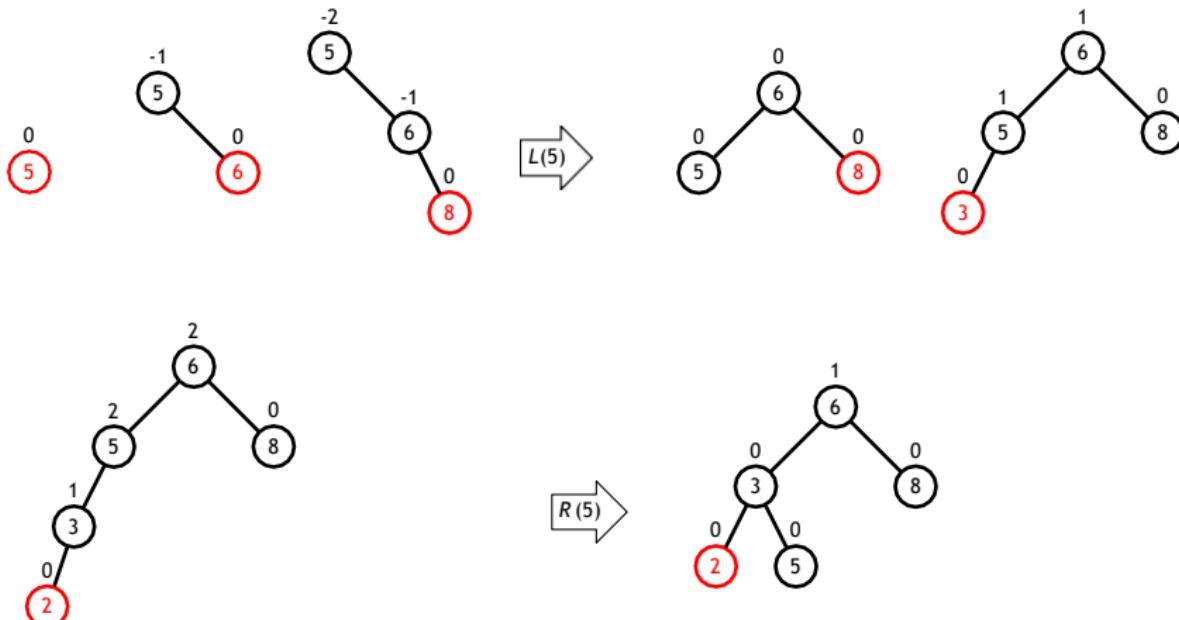
AVL trees Insertion Example

Construct an AVL tree for the sequence 5, 6, 8, 3, 2, 4, 7



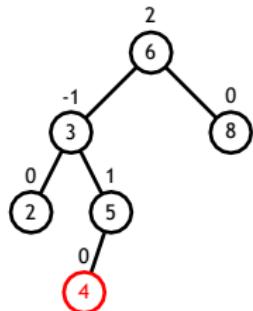
AVL trees Insertion Example

Construct an AVL tree for the sequence 5, 6, 8, 3, 2, 4, 7



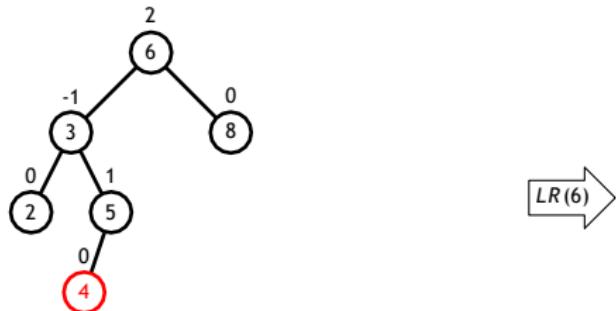
AVL trees

Construct an AVL tree for the sequence 5, 6, 8, 3, 2, 4, 7



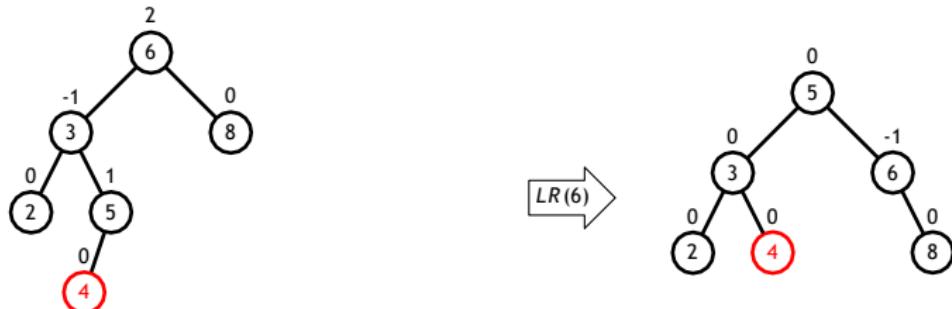
AVL trees

Construct an AVL tree for the sequence 5, 6, 8, 3, 2, 4, 7



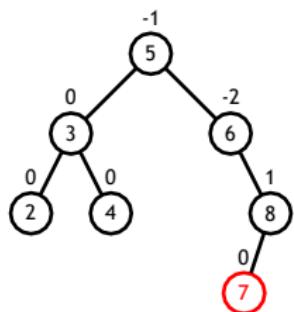
AVL trees

Construct an AVL tree for the sequence 5, 6, 8, 3, 2, 4, 7



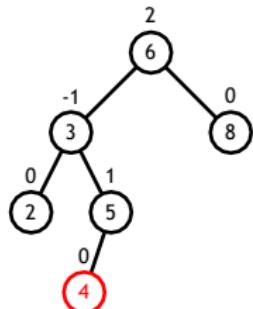
AVL trees

Construct an AVL tree for the sequence 5, 6, 8, 3, 2, 4, 7

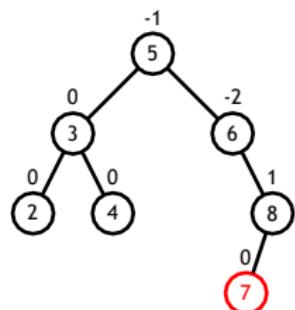
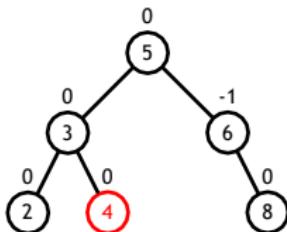


AVL trees

Construct an AVL tree for the sequence 5, 6, 8, 3, 2, 4, 7



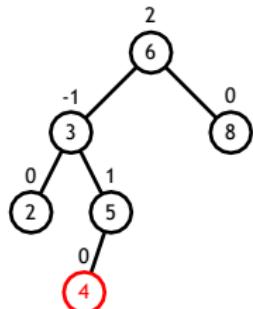
$LR(6)$



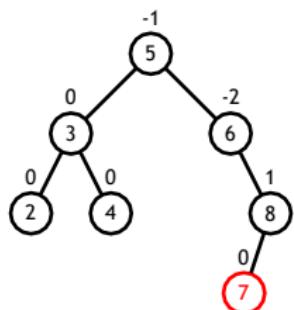
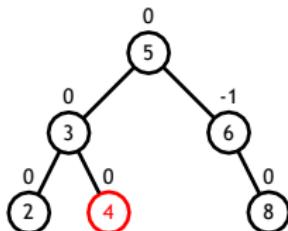
$RL(6)$

AVL trees

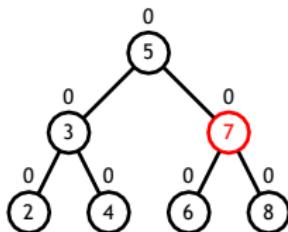
Construct an AVL tree for the sequence 5, 6, 8, 3, 2, 4, 7



LR(6)



RL(6)



AVL trees - Analysis

- $h \leq 1.4404 \log_2(n + 2) - 1.3277$ (h = height)
- Average height (found empirically): $1.01 \log_2 n + 0.1$ for large n .
- **SEARCH** and **INSERT** are $O(\log n)$
- **DELETE** is more difficult, but still in $O(\log n)$
- **rebalance (rotations)** in $O(\log n)$

AVL trees - Analysis

- $h \leq 1.4404 \log_2(n + 2) - 1.3277$ (h = height)
- Average height (found empirically): $1.01 \log_2 n + 0.1$ for large n .
- **SEARCH** and **INSERT** are $O(\log n)$
- **DELETE** is more difficult, but still in $O(\log n)$
- **rebalance (rotations)** in $O(\log n)$
- **Disadvantages:** Rotations are frequent, and implementation is complex.

AVL trees - Example

<https://www.cs.usfca.edu/~galles/visualization/AVLtree.html>

Overview

1 Overview

2 Presorting

3 Balanced Search Trees: AVL Trees

4 Balanced Search Trees: 2-3 Trees

5 Heaps and Heapsort

6 Problem Reduction

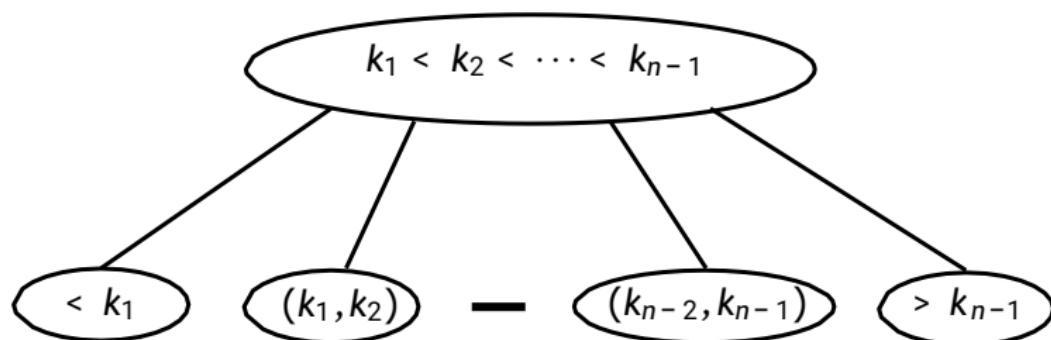
7 Summary

2-3 Trees - Multiway Search Trees

Definition

A **multiway search tree** is a search tree which allows more than one element per node.

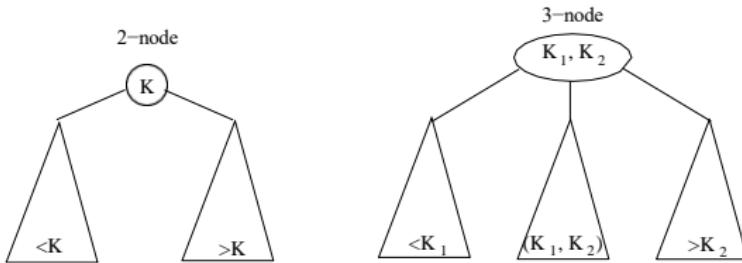
A node of a search tree is called **n-node** if it contains $n - 1$ ordered elements, dividing the entire element range into n intervals.



2-3 Trees

Definition

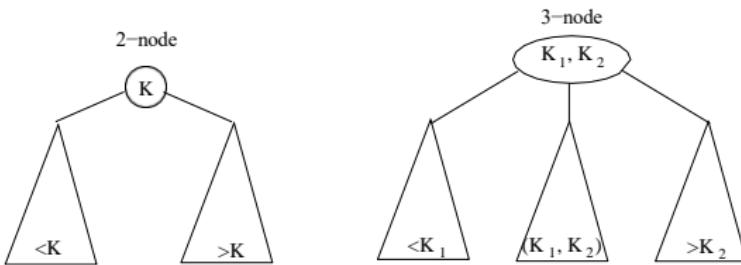
A **2-3 tree** is a search tree which mixes 2-nodes and 3-nodes to keep the tree height-balanced (all leaves are on the same level).



2-3 Trees

Definition

A **2-3 tree** is a search tree which mixes 2-nodes and 3-nodes to keep the tree height-balanced (all leaves are on the same level).



Construction:

- A 2-3 tree is constructed by successive insertions of given elements, with a new element always inserted into a **leaf** of the tree, following 2-3 parent-child rules.
- If the leaf becomes a **4-node** (has 3 elements), it is **split into three** nodes, with the middle element **promoted** to the parent node.

2-3 Trees

Construct a 2-3 tree for the sequence 9, 5, 8, 3, 2, 4, 7.

9

2-3 Trees

Construct a 2-3 tree for the sequence 9, 5, 8, 3, 2, 4, 7.

9 5, 9

2-3 Trees

Construct a 2-3 tree for the sequence 9, 5, 8, 3, 2, 4, 7.



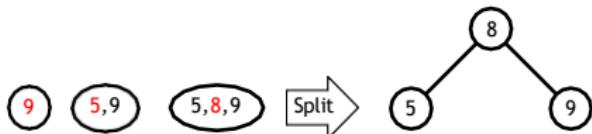
2-3 Trees

Construct a 2-3 tree for the sequence 9, 5, 8, 3, 2, 4, 7.



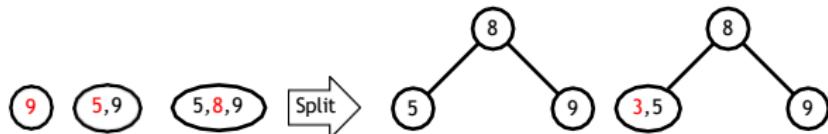
2-3 Trees

Construct a 2-3 tree for the sequence 9, 5, 8, 3, 2, 4, 7.



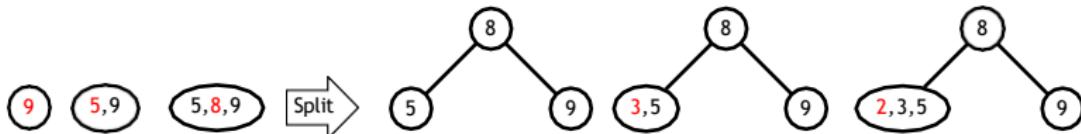
2-3 Trees

Construct a 2-3 tree for the sequence 9, 5, 8, 3, 2, 4, 7.



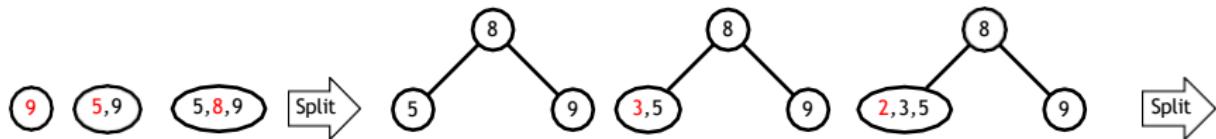
2-3 Trees

Construct a 2-3 tree for the sequence 9, 5, 8, 3, 2, 4, 7.



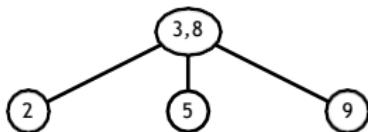
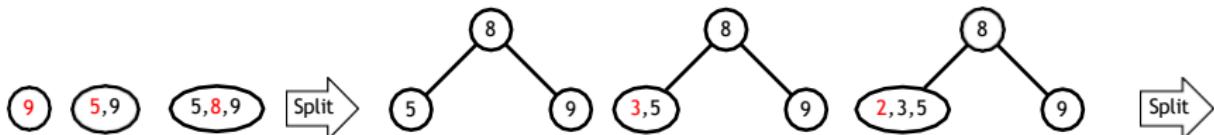
2-3 Trees

Construct a 2-3 tree for the sequence 9, 5, 8, 3, 2, 4, 7.



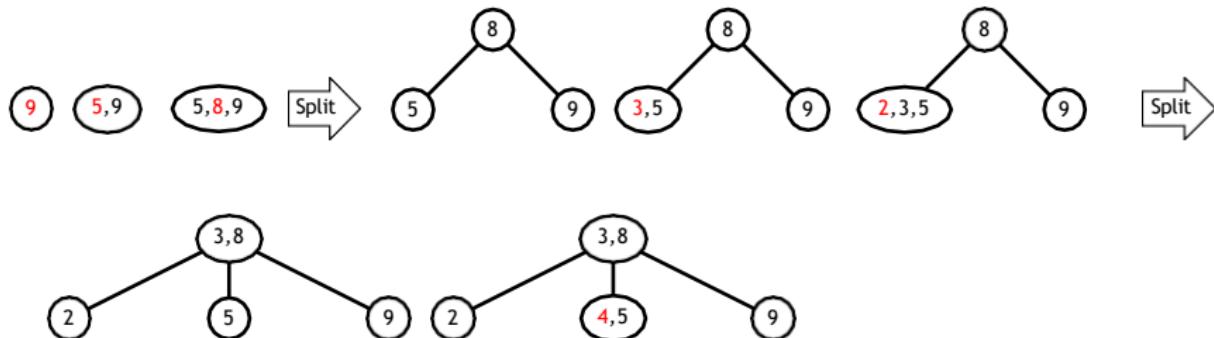
2-3 Trees

Construct a 2-3 tree for the sequence 9, 5, 8, 3, 2, 4, 7.



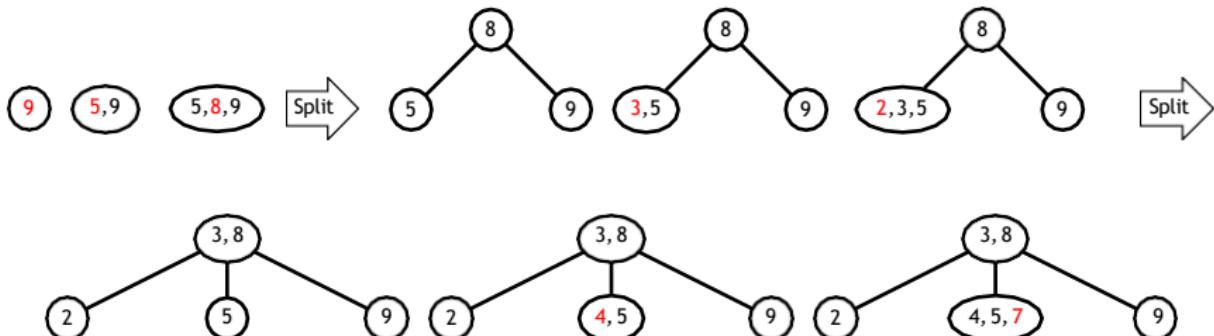
2-3 Trees

Construct a 2-3 tree for the sequence 9, 5, 8, 3, 2, 4, 7.



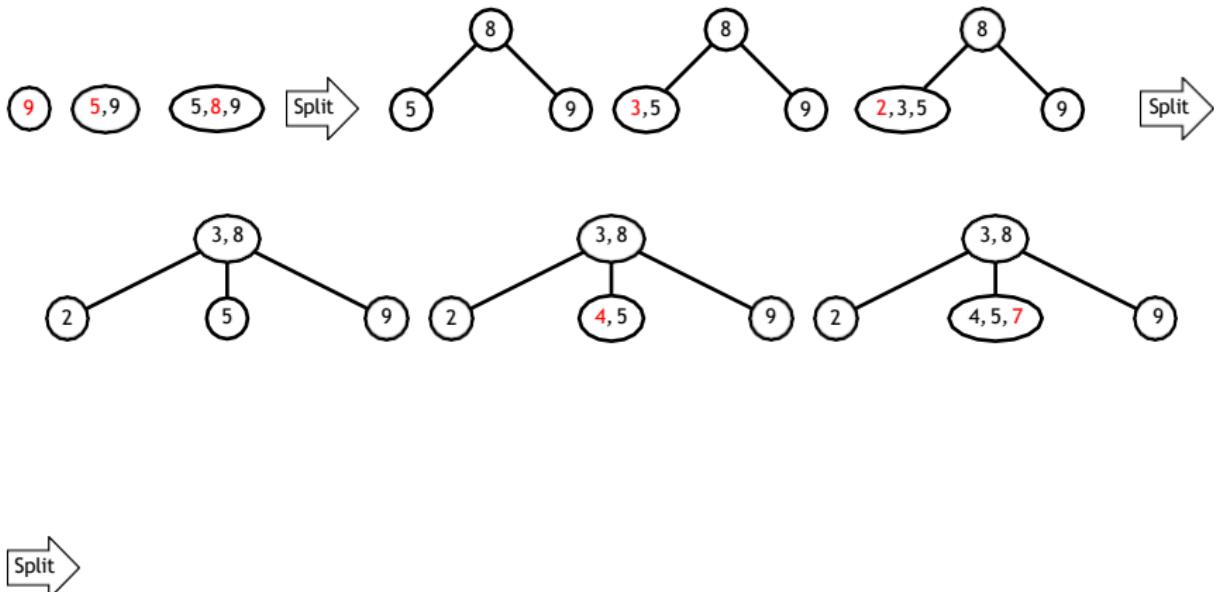
2-3 Trees

Construct a 2-3 tree for the sequence 9, 5, 8, 3, 2, 4, 7.



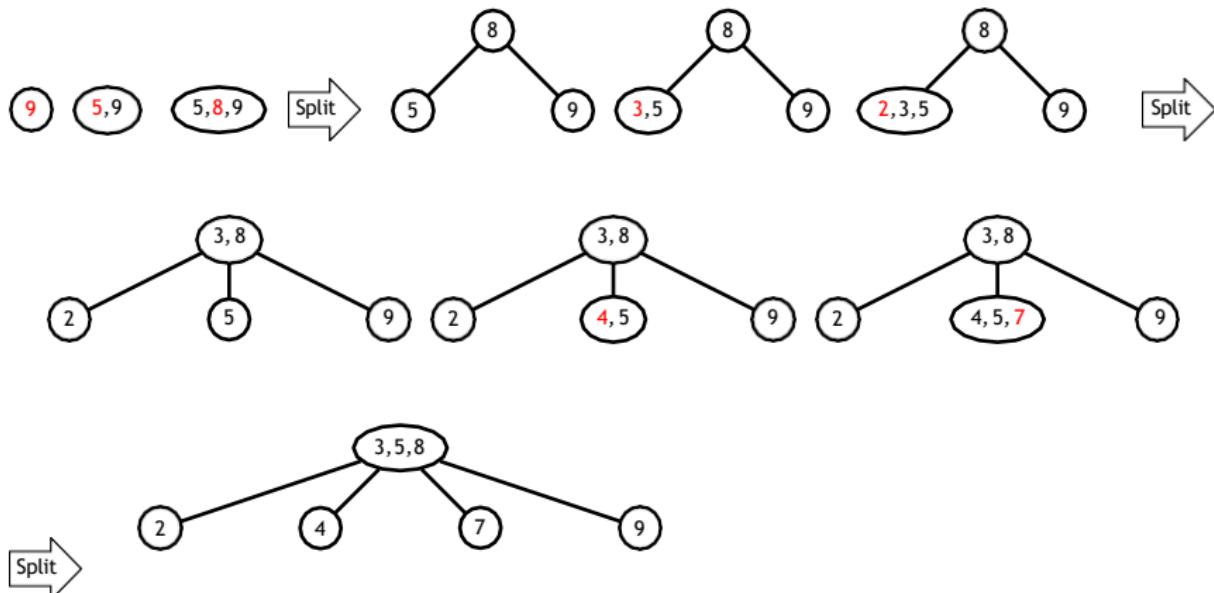
2-3 Trees

Construct a 2-3 tree for the sequence 9, 5, 8, 3, 2, 4, 7.



2-3 Trees

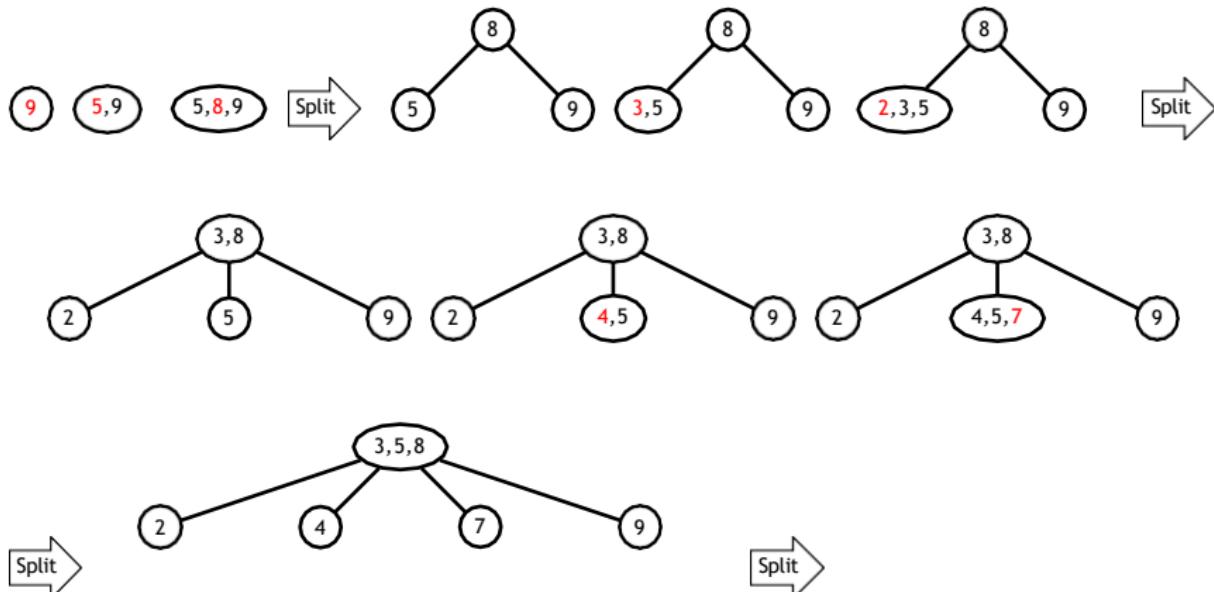
Construct a 2-3 tree for the sequence 9, 5, 8, 3, 2, 4, 7.



Intermediate state.

2-3 Trees

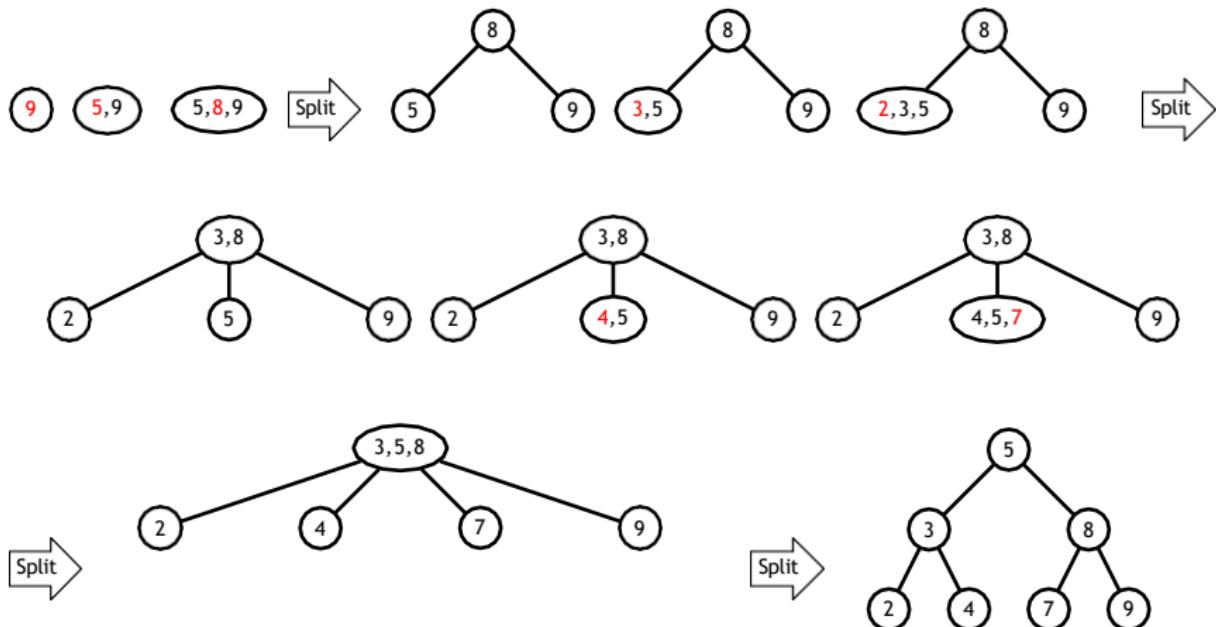
Construct a 2-3 tree for the sequence 9, 5, 8, 3, 2, 4, 7.



Intermediate state.

2-3 Trees

Construct a 2-3 tree for the sequence 9, 5, 8, 3, 2, 4, 7.



Intermediate state.

2-3 Trees - Analysis

- $\log_3(n + 1) - 1 \leq h \leq \log_2(n + 1) - 1$.
- **SEARCH**, **INSERT**, and **DELETE** are all $O(\log n)$
- **rebalancing** on average is cheaper and may occur less frequently than AVL tree
- 2-3 trees may waste some space because not every node is full

2-3 Trees - Example

Build a 2-3 tree containing: 20, 10, 50, 5, 15, 30, 40.

Visualization:

<https://www.cs.usfca.edu/~galles/visualization/BTree.html>

Overview

1 Overview

2 Presorting

3 Balanced Search Trees: AVL Trees

4 Balanced Search Trees: 2-3 Trees

5 Heaps and Heapsort

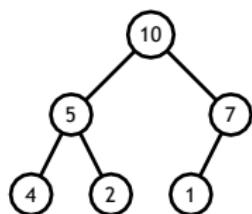
6 Problem Reduction

7 Summary

Definition

A **max heap** is a binary tree with a single key at each node with the following properties:

- ① **Shape Property:** the tree is complete, i.e. all levels are full except the last level, where some of the rightmost nodes are missing
- ② **Heap Property/Parental Dominance:** the key at each parent node is \geq to all child keys



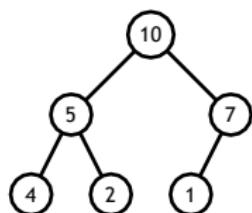
(a) a heap

Heaps

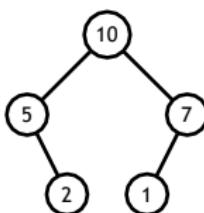
Definition

A **max heap** is a binary tree with a single key at each node with the following properties:

- ① **Shape Property:** the tree is complete, i.e. all levels are full except the last level, where some of the rightmost nodes are missing
- ② **Heap Property/Parental Dominance:** the key at each parent node is \geq to all child keys



(a) a heap



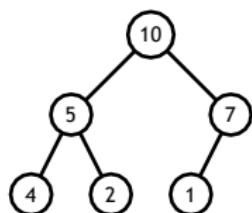
(b) not a heap

Heaps

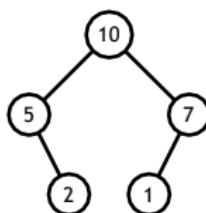
Definition

A **max heap** is a binary tree with a single key at each node with the following properties:

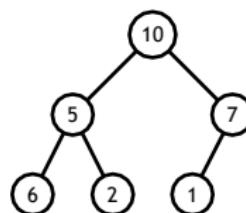
- ① **Shape Property:** the tree is complete, i.e. all levels are full except the last level, where some of the rightmost nodes are missing
- ② **Heap Property/Parental Dominance:** the key at each parent node is \geq to all child keys



(a) a heap



(b) not a heap



(c) not a heap

Heaps - Properties

Efficient data structure for several important applications, including:

- implement priority queues
- finding max/min in an array of elements
- fast implementations of graph algorithms like Prim's algorithm
- implement heapsort

Heaps - Properties

Efficient data structure for several important applications, including:

- implement priority queues
- finding max/min in an array of elements
- fast implementations of graph algorithms like Prim's algorithm
- implement heapsort

Properties:

- 1 The root contains the largest key.
- 2 The subtree rooted at any node of a heap is also a heap.

Heaps - Properties

Efficient data structure for several important applications, including:

- implement priority queues
- finding max/min in an array of elements
- fast implementations of graph algorithms like Prim's algorithm
- implement heapsort

Properties:

- ① The root contains the largest key.
- ② The subtree rooted at any node of a heap is also a heap.
- ③ A heap can be represented **implicitly** as an array. (conceptually easier to understand as trees, but more simple and efficient as arrays)

Heaps - Properties

Efficient data structure for several important applications, including:

- implement priority queues
- finding max/min in an array of elements
- fast implementations of graph algorithms like Prim's algorithm
- implement heapsort

Properties:

- ① The root contains the largest key.
- ② The subtree rooted at any node of a heap is also a heap.
- ③ A heap can be represented **implicitly** as an array. (conceptually easier to understand as trees, but more simple and efficient as arrays)

Heap Construction (bottom up)

Scenario: Have a set of keys, want to build a heap from them in one go, rather than adding them one by one.

Heap Construction (bottom up)

Scenario: Have a set of keys, want to build a heap from them in one go, rather than adding them one by one.

Step 1 : Initialize the structure with keys in the order given. (top-down, left-right)

Heap Construction (bottom up)

Scenario: Have a set of keys, want to build a heap from them in one go, rather than adding them one by one.

Step 1 : Initialize the structure with keys in the order given. (top-down, left-right)

Step 2 : Starting with the **lowest, rightmost** parent node, repair the heap rooted at it, if it does not satisfy the heap condition: keep exchanging it with its largest child until the heap condition holds.

Heap Construction (bottom up)

Scenario: Have a set of keys, want to build a heap from them in one go, rather than adding them one by one.

Step 1 : Initialize the structure with keys in the order given. (top-down, left-right)

Step 2 : Starting with the **lowest, rightmost** parent node, repair the heap rooted at it, if it does not satisfy the heap condition: keep exchanging it with its largest child until the heap condition holds.

Step 3 : Repeat **Step 2** for the parent nodes at the same level, right to left.

Heap Construction (bottom up)

Scenario: Have a set of keys, want to build a heap from them in one go, rather than adding them one by one.

Step 1 : Initialize the structure with keys in the order given. (top-down, left-right)

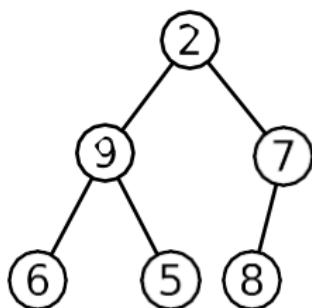
Step 2 : Starting with the **lowest, rightmost** parent node, repair the heap rooted at it, if it does not satisfy the heap condition: keep exchanging it with its largest child until the heap condition holds.

Step 3 : Repeat **Step 2** for the parent nodes at the same level, right to left.

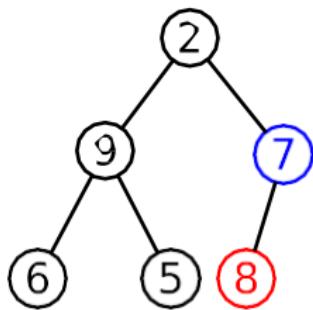
Step 4 : Repeat **Steps 2 and 3** for next level up.

Heap Construction

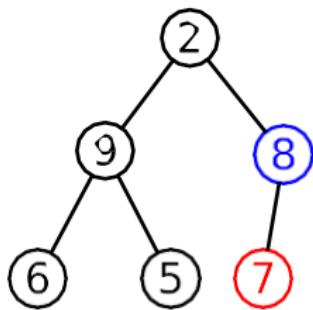
Given 2,9,7,6,5,8:



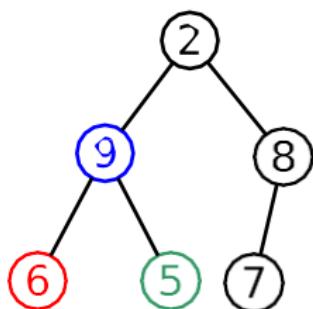
Heap Construction



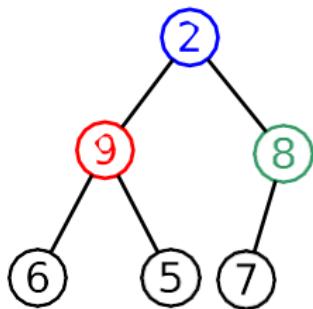
Heap Construction



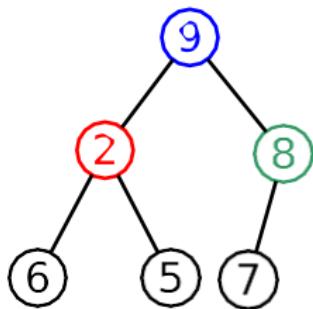
Heap Construction



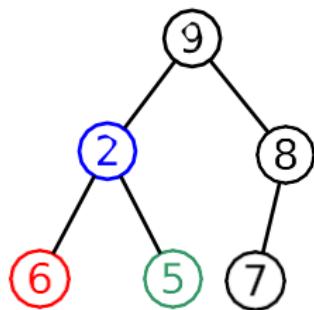
Heap Construction



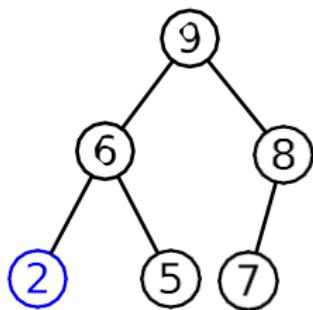
Heap Construction



Heap Construction



Heap Construction



Heap Construction - Analysis

Simple analysis:

- ① The height of the heap is $h = \log_2 n$
- ② Each call to **repair** takes $O(\log n)$ time (**Why?**)

Heap Construction - Analysis

Simple analysis:

- ① The height of the heap is $h = \log_2 n$
- ② Each call to **repair** takes $O(\log n)$ time (**Why?**) (Think of repairing the root...)

Heap Construction - Analysis

Simple analysis:

- ① The height of the heap is $h = \log_2 n$
- ② Each call to **repair** takes $O(\log n)$ time (**Why?**) (**Think of repairing the root...**)
- ③ There are $n/2 \in O(n)$ such calls (**Why?**)

Heap Construction - Analysis

Simple analysis:

- ① The height of the heap is $h = \log_2 n$
- ② Each call to **repair** takes $O(\log n)$ time (Why?) (Think of repairing the root...)
- ③ There are $n/2 \in O(n)$ such calls (Why?) (There are at most $n/2$ parents...)

Heap Construction - Analysis

Simple analysis:

- ① The height of the heap is $h = \log_2 n$
- ② Each call to **repair** takes $O(\log n)$ time (**Why?**) (**Think of repairing the root...**)
- ③ There are $n/2 \in O(n)$ such calls (**Why?**) (**There are at most $n/2$ parents...**)
- ④ Therefore, $O(n \log n)$ is an upper bound on the running time of **building the heap**

Heap Construction - Analysis

Simple analysis:

- ① The height of the heap is $h = \log_2 n$
- ② Each call to **repair** takes $O(\log n)$ time (**Why?**) (**Think of repairing the root...**)
- ③ There are $n/2 \in O(n)$ such calls (**Why?**) (**There are at most $n/2$ parents...**)
- ④ Therefore, $O(n \log n)$ is an upper bound on the running time of **building the heap**

Tighter (improved) upper bound:

- ① At level i of tree, we have 2^i nodes.
- ② The time required to perform **repair** on a node of level i is $2(h - i)$.
- ③ Therefore, the running time for **building the heap** is:

$$\sum_{i=0}^{h-1} 2(h-i)2^i = 2(n - \log_2(n + 1)) \in O(n)$$

Priority Queues

Definition

A **priority queue** is an abstract data type for an ordered set which supports the following operations:

- ① **FIND** an element with the highest priority;
- ② **DEQUEUE** an element with the highest priority;
- ③ **ENQUEUE** an element with an assigned priority.

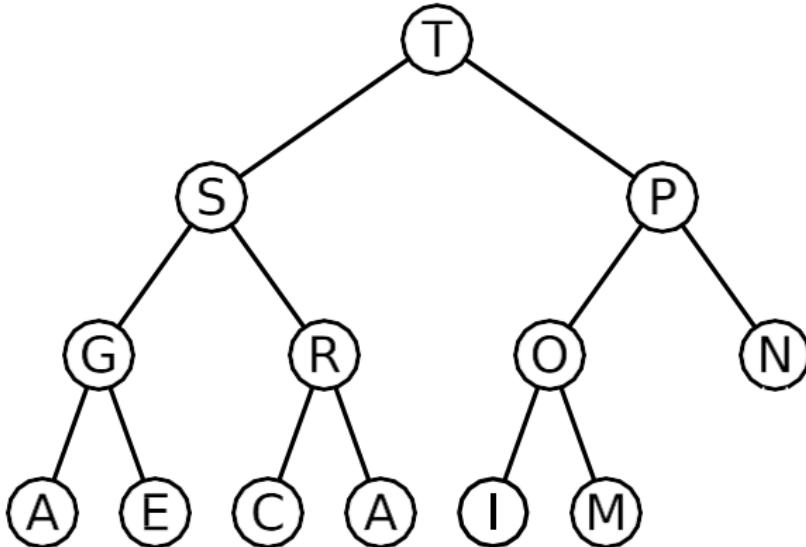
Heaps are an obvious data structure for a priority queue.

Priority Queues - ENQUEUE

Insert w into priority queue:

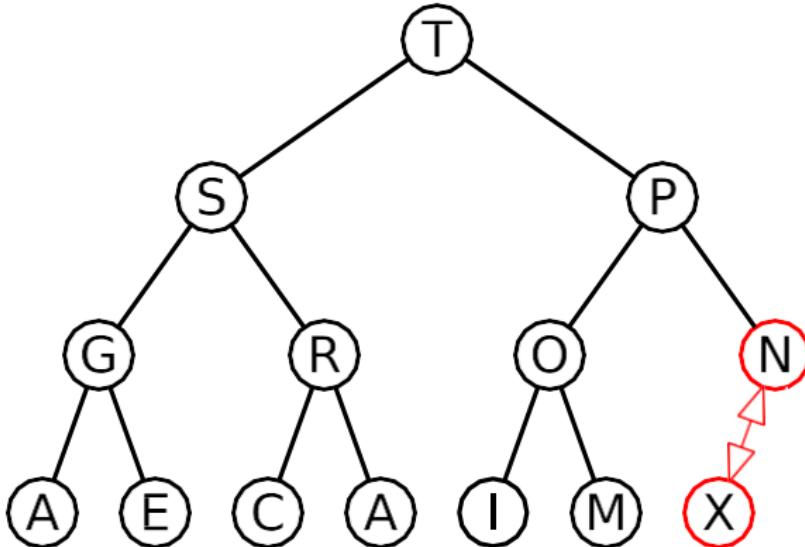
- ① Insert node holding w in rightmost position in bottom, leaf level in existing heap.
- ② Repair heap, starting from parent at inserted node w.

Priority Queues - ENQUEUE



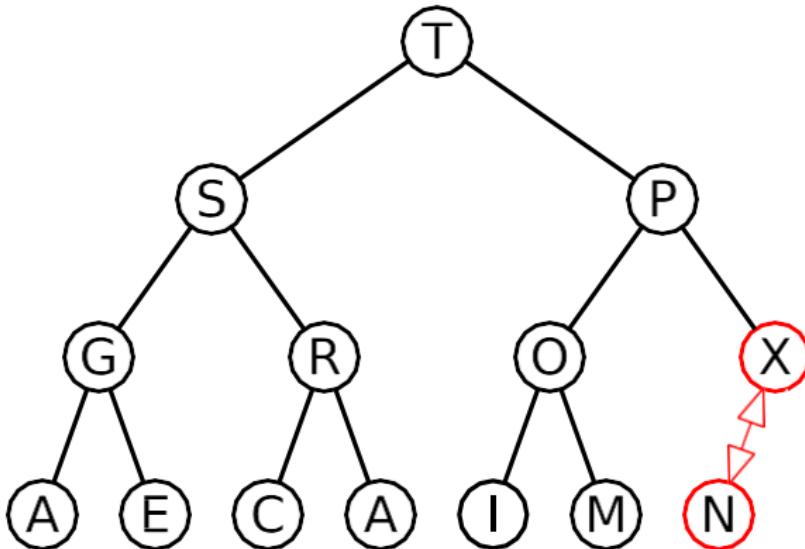
Insert “X” into a Priority Queue.

Priority Queues - ENQUEUE



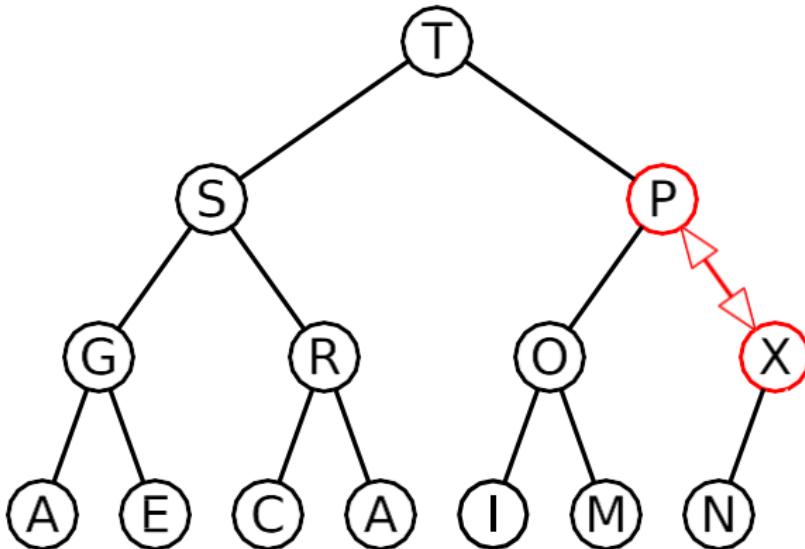
Insert “X” into a Priority Queue.

Priority Queues - ENQUEUE



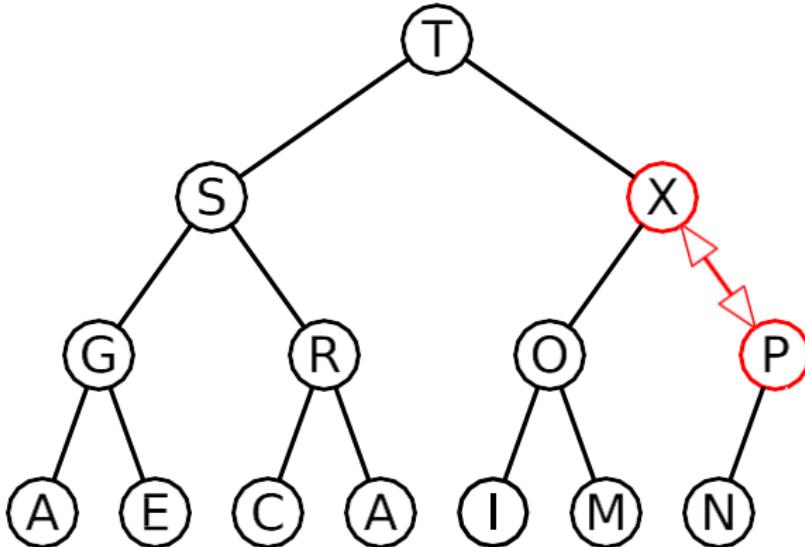
Insert “X” into a Priority Queue.

Priority Queues - ENQUEUE



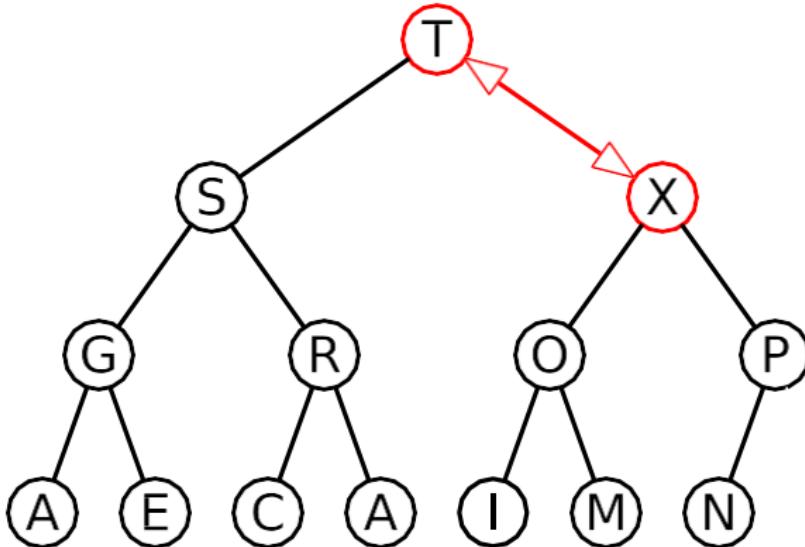
Insert “X” into a Priority Queue.

Priority Queues - ENQUEUE



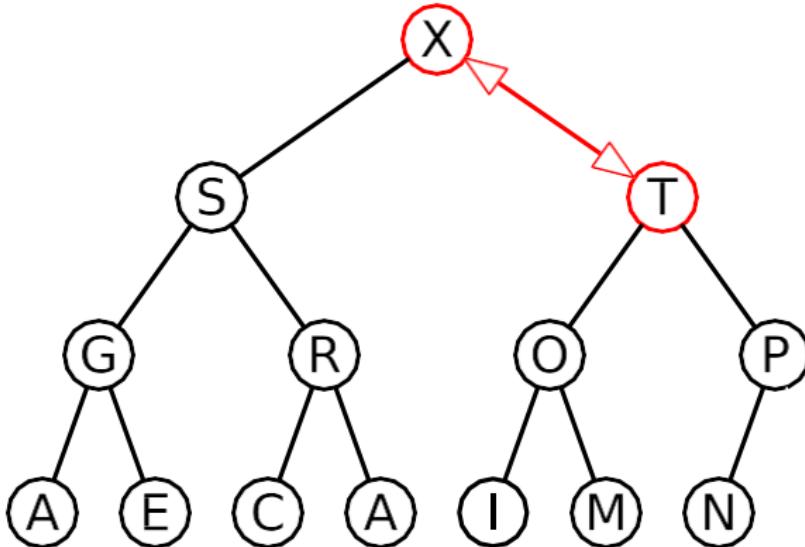
Insert “X” into a Priority Queue.

Priority Queues - ENQUEUE



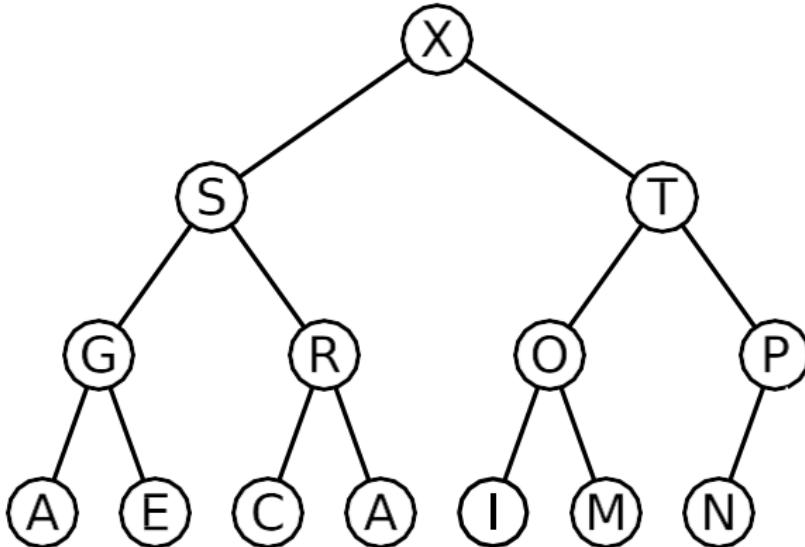
Insert “X” into a Priority Queue.

Priority Queues - ENQUEUE



Insert “X” into a Priority Queue.

Priority Queues - ENQUEUE



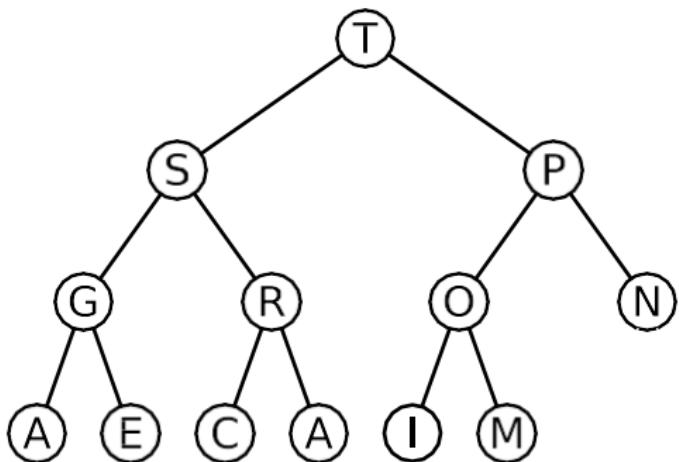
Insert “X” into a Priority Queue.

Priority Queues - DEQUE

Pop largest element off priority queue:

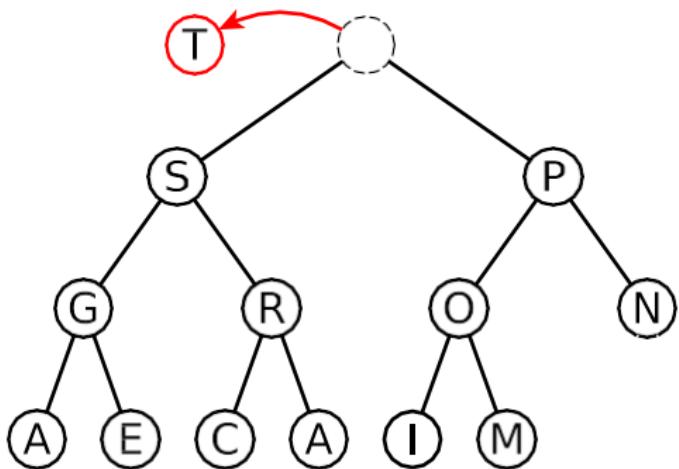
- ① Remove root node of heap (this is largest in heap).
- ② Bring rightmost, leaf node to become root, then repair heap.

Priority Queues - DEQUE



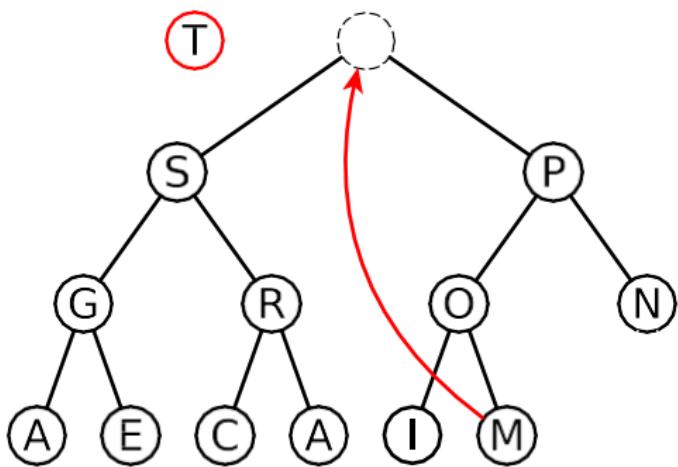
Call DEQUE on a priority queue.

Priority Queues - DEQUE



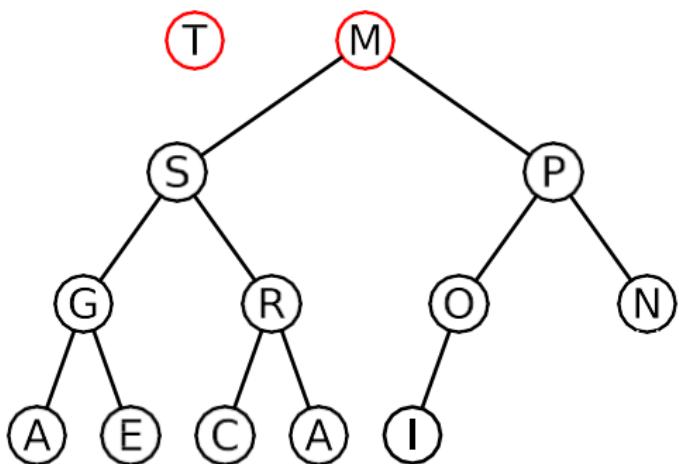
Call DEQUE on a priority queue.

Priority Queues - DEQUE



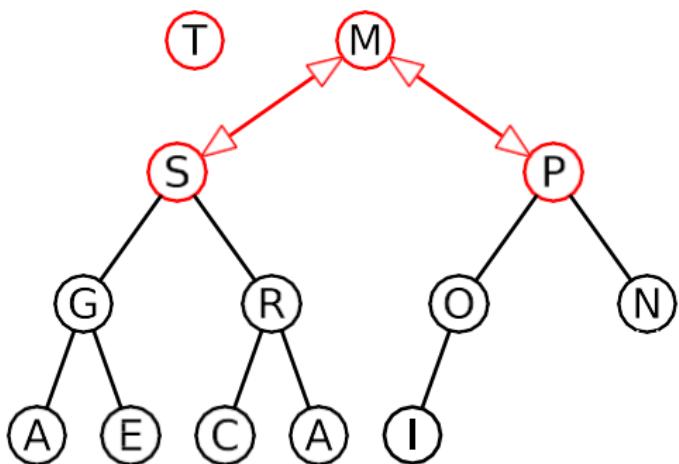
Call DEQUE on a priority queue.

Priority Queues - DEQUE



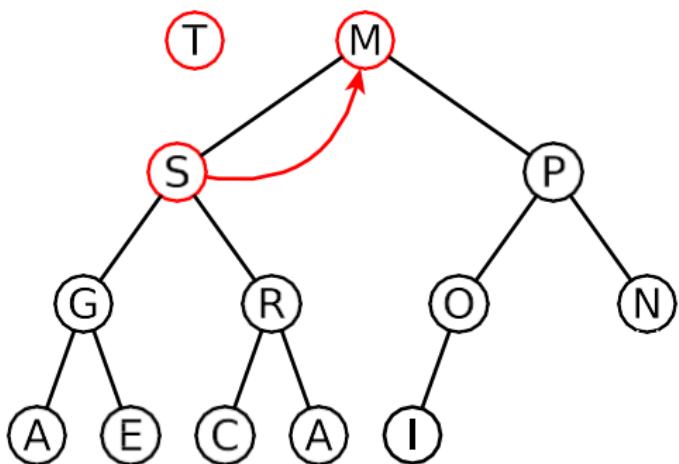
Call DEQUE on a priority queue.

Priority Queues - DEQUE



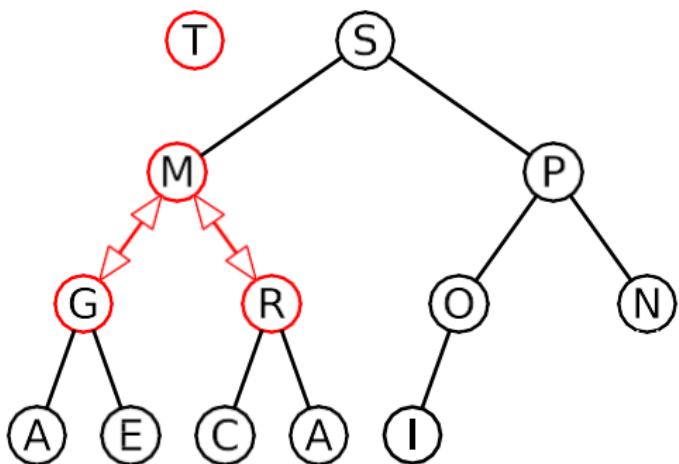
Call DEQUE on a priority queue.

Priority Queues - DEQUE



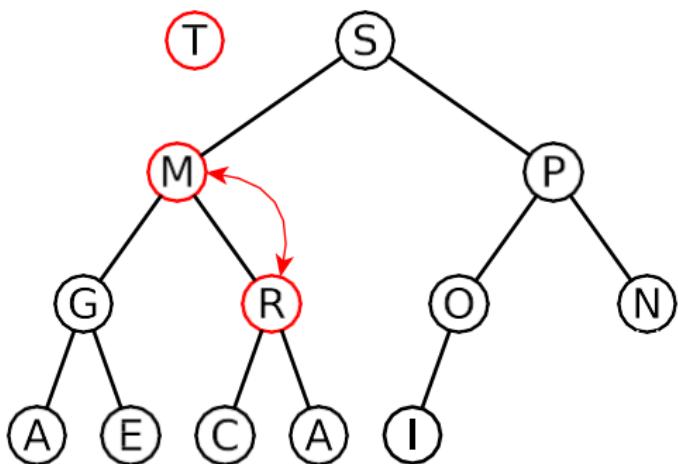
Call DEQUE on a priority queue.

Priority Queues - DEQUE



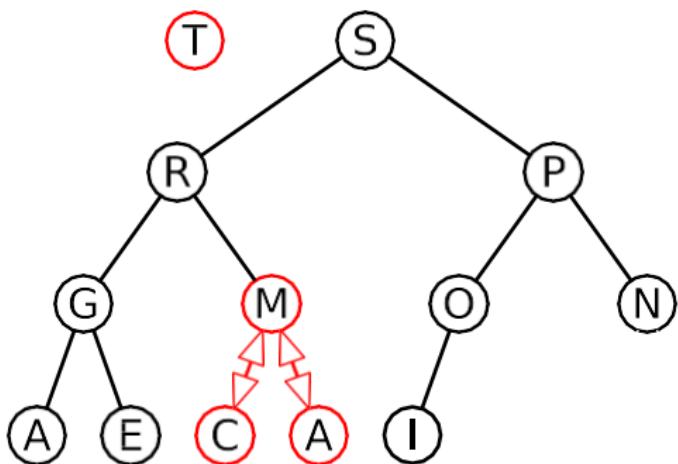
Call **DEQUE** on a priority queue.

Priority Queues - DEQUE



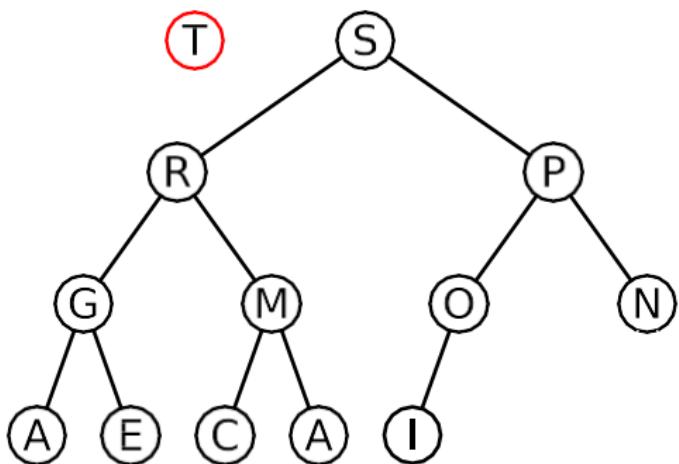
Call DEQUE on a priority queue.

Priority Queues - DEQUE



Call DEQUE on a priority queue.

Priority Queues - DEQUE



Call **DEQUE** on a priority queue.

Heapsort

Selection sort that uses a heap to find the next largest item among the remaining items.

Stage 1: Construct a heap for a given list of n keys.

Stage 2: Repeat operation of deque (root removal) n times.

Heapsort - Analysis

Worst-Case Analysis:

Stage 1 : Build heap for a given list of n keys (where the number of nodes at level $i = 2^i$).

$$C_w(n) \in O(n).$$

Stage 2 : Repeat `DEQUE` n times (see textbook)

$$\begin{aligned} C_w(n) &\leq 2\lceil \log_2(n-1) \rceil + 2\lceil \log_2(n-2) \rceil + \dots + 2\lceil \log_2 1 \rceil \\ &\leq \sum_{i=1}^n 2 \log_2 i \in O(n \log n). \end{aligned}$$

Heap sort is **not stable**. It can be done in-place. The total worst-case efficiency is $O(n \log n) + O(n) \in O(n \log n)$.

Heapsort - Question

Why are we discussing Heapsort as a “Transform and Conquer” algorithm?

Heapsort - Question

Why are we discussing Heapsort as a “Transform and Conquer” algorithm?

Heapsort vs mergesort vs quicksort?

- Average case: Heapsort comparable to Mergesort and slower than Quicksort.
- Worst case: Heapsort comparable with Mergesort and faster than Quicksort.
- Stability: Mergesort is only stable sorting algorithm out of the three.
- In-place: Heapsort and Quicksort are in-place.

Overview

1 Overview

2 Presorting

3 Balanced Search Trees: AVL Trees

4 Balanced Search Trees: 2-3 Trees

5 Heaps and Heapsort

6 Problem Reduction

7 Summary

Problem Reduction

- Solve a problem by transforming it into a different problem for which an algorithm already exists.
- Examples:
 - Least Common Multiples
 - Reduction to Graph Problems

Least Common Multiple

Problem

The **Least Common Multiple** of two positive integers m and n , denoted $\text{LCM}(m, n)$ is defined as the smallest integer that is divisible by both m and n .

- Example $\text{LCM}(24, 60) = 120$, $\text{LCM}(5, 11) = 55$.

Least Common Multiple

Problem

The **Least Common Multiple** of two positive integers m and n , denoted $\text{LCM}(m, n)$ is defined as the smallest integer that is divisible by both m and n .

- Example $\text{LCM}(24, 60) = 120$, $\text{LCM}(5, 11) = 55$.
- **Simple approach:** Compute the common prime factors of m and n . The **LCM** is the product of all the **common prime factors** times each **non-common factor** of n and m .

$$24 = 2 \cdot 2 \cdot 2 \cdot 3$$

$$60 = 2 \cdot 2 \cdot 3 \cdot 5$$

$$\begin{aligned}\text{LCM}(24, 60) &= (2 \cdot 2 \cdot 3) \cdot 2 \cdot 5 \\ &= 120.\end{aligned}$$

Least Common Multiple

- Finding primes by brute force is inefficient.
- The problem can be solved by reduction using Euclid's algorithm.

Least Common Multiple

- Finding primes by brute force is inefficient.
- The problem can be solved by reduction using Euclid's algorithm.
- Recall that the Greatest Common Divisor ($\text{GCD}(m, n)$) is the product of all the common prime factors of m and n .

Least Common Multiple

- Finding primes by brute force is inefficient.
- The problem can be solved by reduction using Euclid's algorithm.
- Recall that the Greatest Common Divisor ($\text{GCD}(m, n)$) is the product of all the common prime factors of m and n .

$$\text{LCM}(m, n) = \frac{m \cdot n}{\text{GCD}(m, n)}$$

- GCD can be computed efficiently using Euclid's method.

Least Common Multiple

How it works? Lets use the example to illustrate:

$$\text{LCM}(24, 60) = \frac{24 \cdot 60}{\text{GCD}(24, 60)} = \frac{1440}{12} = 120.$$

Reduction to Graph Problems

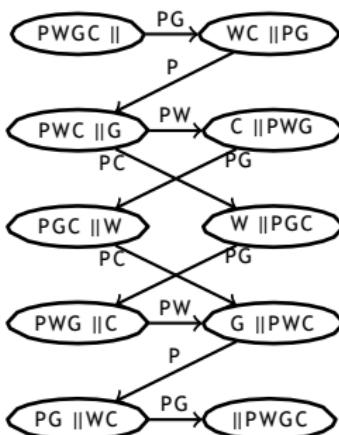
Problem

A peasant finds himself on a river bank with a wolf, a goat, and a head of cabbage. He needs to transport all three to the other side of the river. His boat will only hold himself and one occupant per trip. In his absence, the wolf will eat the goat and the goat will eat the cabbage.

Reduction to Graph Problems

Problem

A peasant finds himself on a river bank with a wolf, a goat, and a head of cabbage. He needs to transport all three to the other side of the river. His boat will only hold himself and one occupant per trip. In his absence, the wolf will eat the goat and the goat will eat the cabbage.



Overview

1 Overview

2 Presorting

3 Balanced Search Trees: AVL Trees

4 Balanced Search Trees: 2-3 Trees

5 Heaps and Heapsort

6 Problem Reduction

7 Summary

Summary

The three types of transformation covered:

- **instance simplification**
 - presorting, uniqueness checking, search
 - balanced search trees (AVL trees)
- **representation change**
 - balanced search trees (2-3 trees)
 - heaps and heapsort
- **problem reduction**
 - LCM
 - reductions to graph problems

Levitin - The design and analysis of algorithms

This week we will be covering the material from Chapter 8.

Learning outcomes:

- Understand and be able to apply dynamic programming to solving problems.
- Examples:
 - Coin-row problem
 - Computing the edit distance
 - Knapsack
 - Transitive closure - Warshall's algorithm

Outline

- ① Overview
- ② Edit Distance
- ③ Knapsack Problem
- ④ Warshall's Algorithm
- ⑤ Summary

Overview

1 Overview

2 Edit Distance

3 Knapsack Problem

4 Warshall's Algorithm

5 Summary

Dynamic Programming

Dynamic Programming is a general algorithm approach for solving problems using the solutions of (overlapping) subproblems.

Dynamic Programming

Dynamic Programming is a general algorithm approach for solving problems using the solutions of (overlapping) subproblems.

Main idea:

- ① Setup a recurrence relating a solution of larger instances to the solutions of smaller instances.
- ② Solve smaller instances once.
- ③ Record solutions in a table.
- ④ Extract solutions to the initial instance from the table, i.e., use solutions of smaller instances to construct solutions of larger initial problem instance.

Dynamic Programming

Sounds similar? Divide-and-conquer?

Difference?

- Dynamic programming can be thought of as divide-and-conquer and storing sub-solutions.
- Why have both then?

Dynamic Programming

Sounds similar? Divide-and-conquer?

Difference?

- Dynamic programming can be thought of as divide-and-conquer and storing sub-solutions.
- Why have both then?
 - Divide-and-conquer algorithms are preferred when the sub-problems/instances are **independent**, e.g., Mergesort.

Dynamic Programming

Sounds similar? Divide-and-conquer?

Difference?

- Dynamic programming can be thought of as divide-and-conquer and storing sub-solutions.
- Why have both then?
 - Divide-and-conquer algorithms are preferred when the sub-problems/instances are **independent**, e.g., Mergesort.
 - Dynamic programming approach better when the sub-problems/instances are **dependent**, i.e., the solution to a sub-problem may be needed multiple times. Hence saving solutions allow them to be reused rather than recomputed. Tradeoff space (more) for time (faster).

Dynamic Programming Approaches

Two basic approaches to dynamic programming.

For both first: Study a recursive divide-and-conquer algorithm and figure out the dependencies between the subproblems.

Dynamic Programming Approaches

Two basic approaches to dynamic programming.

For both first: Study a recursive divide-and-conquer algorithm and figure out the dependencies between the subproblems.

① Top-Down

- Start with a divide-and-conquer algorithm, and begin dividing recursively.
- Only solve/recuse on a subproblem if the solution is not available in the table.
- Save solutions to subproblems in a table.

Dynamic Programming Approaches

Two basic approaches to dynamic programming.

For both first: Study a recursive divide-and-conquer algorithm and figure out the dependencies between the subproblems.

① Top-Down

- Start with a divide-and-conquer algorithm, and begin dividing recursively.
- Only solve/recuse on a subproblem if the solution is not available in the table.
- Save solutions to subproblems in a table.

② Bottom-Up

- Solve all subproblems, and use solutions to subproblems to construct solutions to larger problems.

Dynamic Programming example: Coin-row Problem

Coin-row Problem

Given a row of n coins with positive integer values c_1, c_2, \dots, c_n (not necessarily distinct), pick up the **maximum amount** of money with the constraint that **no two adjacent coins** can be selected.

EXAMPLE: 5, 1, 2, 10, 6, 2. What is the best selection?

Dynamic Programming example: Coin-row Problem

EXAMPLE: 5, 1, 2, 10, 6, 2. What is the best selection?

Dynamic Programming example: Coin-row Problem

EXAMPLE: 5, 1, 2, 10, 6, 2. What is the best selection?

Rough sketch of dynamic programming approach: Say we are considering whether to select the last coin (value 2) in our selection. We have two choices: select or not select the last coin (to maximise total value of coins selected):

Dynamic Programming example: Coin-row Problem

EXAMPLE: 5, 1, 2, 10, 6, 2. What is the best selection?

Rough sketch of dynamic programming approach: Say we are considering whether to select the last coin (value 2) in our selection. We have two choices: select or not select the last coin (to maximise total value of coins selected):

- If we don't select the last coin, then the best solution must be the same as the sequence of coins selected for sub-problem 5, 1, 2, 10, 6.

Dynamic Programming example: Coin-row Problem

EXAMPLE: 5, 1, 2, 10, 6, 2. What is the best selection?

Rough sketch of dynamic programming approach: Say we are considering whether to select the last coin (value 2) in our selection. We have two choices: select or not select the last coin (to maximise total value of coins selected):

- If we **don't select the last coin**, then the best solution must be the same as the sequence of coins selected for sub-problem 5, 1, 2, 10, 6.
- If we **do select the last coin**, then we can't select the 2nd last coin (hence cannot use the optimal selection for sub-problem 5, 1, 2, 10, 6). However, the best solution will include this last coin + the best solution for the sub-problem 5, 1, 2, 10.

Dynamic Programming example: Coin-row Problem

- Let $F(n)$ denote the maximum total amount of money picked up after considering all n coins in row.
- c_i denote the monetary value of coin number i .

Dynamic Programming example: Coin-row Problem

- Let $F(n)$ denote the maximum total amount of money picked up after considering all n coins in row.
- c_i denote the monetary value of coin number i .

Then we can write recurrence relationship:

$F(n) = \max\{\text{total value of solution that selects } n\text{th coin},$
 $\text{total value of solution not does not select } n\text{th coin}\}$ for $n > 1$

$F(0) = 0, F(1) = \text{value from picking up first coin.}$

Dynamic Programming example: Coin-row Problem

- Let $F(n)$ denote the maximum total amount of money picked up after considering all n coins in row.
- c_i denote the monetary value of coin number i .

Then we can write recurrence relationship:

$$F(n) = \max\{\text{total value of solution that selects } n\text{th coin}, \\ \text{total value of solution not select } n\text{th coin}\} \text{ for } n > 1$$

$$F(0) = 0, F(1) = \text{value from picking up first coin.}$$

Formally:

$$F(n) = \max\{c_n + \text{total value of solution to the } n - 2 \text{ coin sub-problem}, \\ \text{total value of solution to the } n - 1 \text{ coin sub-problem}\} \\ \text{for } n > 1,$$

$$F(0) = 0, F(1) = \text{value from picking up first coin.}$$

Dynamic Programming example: Coin-row Problem

- Let $F(n)$ denote the maximum total amount of money picked up after considering all n coins in row.
- c_i denote the monetary value of coin number i .

Then we can write recurrence relationship:

$$F(n) = \max\{c_n + \text{total value of solution to the } n - 2 \text{ coin sub-problem}, \\ \text{total value of solution to the } n - 1 \text{ coin sub-problem}\}$$

for $n > 1$,

$$F(0) = 0, F(1) = \text{value from picking up first coin.}$$

Dynamic Programming example: Coin-row Problem

- Let $F(n)$ denote the maximum total amount of money picked up after considering all n coins in row.
- c_i denote the monetary value of coin number i .

Then we can write recurrence relationship:

$$F(n) = \max\{c_n + \text{total value of solution to the } n - 2 \text{ coin sub-problem}, \\ \text{total value of solution to the } n - 1 \text{ coin sub-problem}\} \\ \text{for } n > 1,$$

$$F(0) = 0, F(1) = \text{value from picking up first coin.}$$

Formally:

$$F(n) = \max\{c_n + F(n - 2), F(n - 1)\} \text{ for } n > 1,$$

$$F(0) = 0, F(1) = c_1.$$

Coin-row Problem

(continued)

$$F(n) = \max\{c_n + F(n-2), F(n-1)\} \text{ for } n > 1,$$

$$F(0) = 0, F(1) = c_1.$$

index (i)	0	1	2	3	4	5	6
coins	-	5	1	2	10	6	2
$F(i)$							

Coin-row Problem (continued)

$$F(n) = \max\{c_n + F(n-2), F(n-1)\} \text{ for } n > 1,$$

$$F(0) = 0, F(1) = c_1.$$

index (i)	0	1	2	3	4	5	6
coins	-	5	1	2	10	6	2
$F(i)$	0						

Coin-row Problem (continued)

$$F(n) = \max\{c_n + F(n-2), F(n-1)\} \text{ for } n > 1,$$

$$F(0) = 0, F(1) = c_1.$$

index (i)	0	1	2	3	4	5	6
coins	-	5	1	2	10	6	2
$F(i)$	0	5					

$$F[0] = 0, F[1] = c_1 = 5$$

Coin-row Problem (continued)

$$F(n) = \max\{c_n + F(n-2), F(n-1)\} \text{ for } n > 1,$$

$$F(0) = 0, F(1) = c_1.$$

index (i)	0	1	2	3	4	5	6
coins	-	5	1	2	10	6	2
$F(i)$	0	5	5				

$$F[2] = \max(1 + 0, 5) = 5$$

Coin-row Problem (continued)

$$F(n) = \max\{c_n + F(n - 2), F(n - 1)\} \text{ for } n > 1,$$

$$F(0) = 0, F(1) = c_1.$$

index (i)	0	1	2	3	4	5	6
coins	-	5	1	2	10	6	2
$F(i)$	0	5	5	7			

$$F[3] = \max(2 + 5, 5) = 7$$

Coin-row Problem (continued)

$$F(n) = \max\{c_n + F(n - 2), F(n - 1)\} \text{ for } n > 1,$$

$$F(0) = 0, F(1) = c_1.$$

index (i)	0	1	2	3	4	5	6
coins	-	5	1	2	10	6	2
$F(i)$	0	5	5	7	15		

$$F[4] = \max(10 + 5, 7) = 15$$

Coin-row Problem (continued)

$F(n) = \max\{c_n + F(n - 2), F(n - 1)\}$ for $n > 1$,

$F(0) = 0, F(1) = c_1.$

index (i)	0	1	2	3	4	5	6
coins	-	5	1	2	10	6	2
$F(i)$	0	5	5	7	15	15	

$$F[5] = \max(6 + 7, 15) = 15$$

Coin-row Problem (continued)

$$F(n) = \max\{c_n + F(n - 2), F(n - 1)\} \text{ for } n > 1,$$

$$F(0) = 0, F(1) = c_1.$$

index (i)	0	1	2	3	4	5	6
coins	-	5	1	2	10	6	2
$F(i)$	0	5	5	7	15	15	17

$$F[6] = \max(2 + 15, 15) = 17$$

Coin-row Problem (continued)

$$F(n) = \max\{c_n + F(n-2), F(n-1)\} \text{ for } n > 1,$$

$$F(0) = 0, F(1) = c_1.$$

index (i)	0	1	2	3	4	5	6
coins	-	5	1	2	10	6	2
$F(i)$	0	5	5	7	15	15	17

Coin-row Problem

(continued)

- In order to retrieve the coins in the maximum subset, we must **backtrack** from the last coin.

Coin-row Problem

(continued)

- In order to retrieve the coins in the maximum subset, we must **backtrack** from the last coin.
- This is called a **backtrace**.

Coin-row Problem

(continued)

- In order to retrieve the coins in the maximum subset, we must **backtrack** from the last coin.
- This is called a **backtrace**.
- First, we find the previous value.
- The previous value is the maximum of $c_n + F(n - 2)$ or $F(n - 1)$.

- In order to retrieve the coins in the maximum subset, we must **backtrack** from the last coin.
- This is called a **backtrace**.
- First, we find the previous value.
- The previous value is the maximum of $c_n + F(n - 2)$ or $F(n - 1)$.
- Say $F(n - 1)$ is the maximum, then this must have been the previous cell we got to $F(n)$.

- In order to retrieve the coins in the maximum subset, we must **backtrack** from the last coin.
- This is called a **backtrace**.
- First, we find the previous value.
- The previous value is the maximum of $c_n + F(n - 2)$ or $F(n - 1)$.
- Say $F(n - 1)$ is the maximum, then this must have been the previous cell we got to $F(n)$.
- Then backtracking from $F(n - 1)$, we see how we got to $F(n - 1)$ - either $c_{n-1} + F(n - 3)$ or $F(n - 2)$

Coin-row Problem

(continued)

- In order to retrieve the coins in the maximum subset, we must **backtrack** from the last coin.
- This is called a **backtrace**.
- First, we find the previous value.
- The previous value is the maximum of $c_n + F(n - 2)$ or $F(n - 1)$.
- Say $F(n - 1)$ is the maximum, then this must have been the previous cell we got to $F(n)$.
- Then backtracking from $F(n - 1)$, we see how we got to $F(n - 1)$ - either $c_{n-1} + F(n - 3)$ or $F(n - 2)$
- Continue until we reach $F(0)$

Coin-row Problem

(continued)

- The previous value is the maximum of $c_n + F(n - 2)$ or $F(n - 1)$.

index (i)	0	1	2	3	4	5	6
coins	-	5	1	2	10	6	2
$F(i)$	0	5	5	7	15	15	17

Coin-row Problem (continued)

- The previous value is the maximum of $c_n + F(n - 2)$ or $F(n - 1)$.

index (i)	0	1	2	3	4	5	6
coins	-	5	1	2	10	6	2
$F(i)$	0	5	5	7	15	15	17

- So, we used $F(5) = 15$ or $c_6 + F(4) = 17$. We used the latter, so $c_6 = 2$ was selected.

Coin-row Problem (continued)

- The previous value is the maximum of $c_n + F(n - 2)$ or $F(n - 1)$.

index (i)	0	1	2	3	4	5	6
coins	-	5	1	2	10	6	2
$F(i)$	0	5	5	7	15	15	17

- So, we used $F(5) = 15$ or $c_6 + F(4) = 17$. We used the latter, so $c_6 = 2$ was selected.
- We look at solution to $F(4)$. Using same process, the maximum at $F(4)$ came from $c_4 + F(2)$, so $c_4 = 10$ was selected.

Coin-row Problem (continued)

- The previous value is the maximum of $c_n + F(n - 2)$ or $F(n - 1)$.

index (i)	0	1	2	3	4	5	6
coins	-	5	1	2	10	6	2
$F(i)$	0	5	5	7	15	15	17

- So, we used $F(5) = 15$ or $c_6 + F(4) = 17$. We used the latter, so $c_6 = 2$ was selected.
- We look at solution to $F(4)$. Using same process, the maximum at $F(4)$ came from $c_4 + F(2)$, so $c_4 = 10$ was selected.
- Finally, we have $F(2) = F(1)$, so best solution for $F(2)$ must be not to select c_2 but select $c_1 = 5$.
- The optimal solution is $\{c_1, c_4, c_6\}$.

Coin-row Problem

(continued)

- The previous value is the maximum of $c_n + F(n - 2)$ or $F(n - 1)$.

index (i)	0	1	2	3	4	5	6
coins	-	5	1	2	10	6	2
$F(i)$	0	5	5	7	15	15	17

- So, we used $F(5) = 15$ or $c_6 + F(4) = 17$. We used the latter, so $c_6 = 2$ was selected.
- We look at solution to $F(4)$. Using same process, the maximum at $F(4)$ came from $c_4 + F(2)$, so $c_4 = 10$ was selected.
- Finally, we have $F(2) = F(1)$, so best solution for $F(2)$ must be not to select c_2 but select $c_1 = 5$.
- The optimal solution is $\{c_1, c_4, c_6\}$.

Question

What is the worst case time and space complexity of this solution?

Overview

1 Overview

2 Edit Distance

3 Knapsack Problem

4 Warshall's Algorithm

5 Summary

Edit Distance

Consider the problem of comparing two sequences/strings.

Applications

- Spell checkers, diff file revisions, plagiarism detection
- Bioinformatics: comparisons of DNA sequences
- Speech recognition

Edit Distance

Consider the problem of comparing two sequences/strings.

Applications

- Spell checkers, diff file revisions, plagiarism detection
- Bioinformatics: comparisons of DNA sequences
- Speech recognition

Solution? Direct position-wise comparison will not work.

Edit Distance

Consider the problem of comparing two sequences/strings.

Applications

- Spell checkers, diff file revisions, plagiarism detection
- Bioinformatics: comparisons of DNA sequences
- Speech recognition

Solution? Direct position-wise comparison will not work.

One possible solution is the edit distance.

Definition

The **Levenshtein distance** or **edit distance** of two strings/sequences S_1 and S_2 is the minimum number of point mutations required to change S_1 into S_2 , where a point mutation is one of the following

- ① substitution of a character, or
- ② insertion of a character, or
- ③ deletion of a character.

- A classic problem for dynamic programming solutions.

Edit Distance

- The Edit distance is a generalization of the **Hamming distance**.
The Hamming distance compares the number of differences in the two strings. It compares the i th position of string S_1 with the i th position of string S_2 .

Edit Distance

- The Edit distance is a generalization of the **Hamming distance**. The Hamming distance compares the number of differences in the two strings. It compares the i th position of string S_1 with the i th position of string S_2 . Example:

$S_1 =$	$A \quad T \quad A \quad T \quad A \quad T \quad A \quad T$
	$\downarrow \quad \downarrow \quad \downarrow \quad \downarrow \quad \downarrow \quad \downarrow \quad \downarrow \quad \downarrow$
$S_2 =$	$T \quad A \quad T \quad A \quad T \quad A \quad T \quad A$

Edit Distance

- The Edit distance is a generalization of the Hamming distance. The Hamming distance compares the number of differences in the two strings. It compares the i th position of string S_1 with the i th position of string S_2 . Example:

$$\begin{array}{rcl} S_1 & = & A \quad T \quad A \quad T \quad A \quad T \quad A \quad \textcolor{red}{T} \\ & & \downarrow \quad \downarrow \\ S_2 & = & \textcolor{red}{T} \quad A \quad T \quad A \quad T \quad A \quad T \quad A \end{array}$$

- The Hamming distance $d(S_1, S_2) = 8$.

Edit Distance

- The Edit distance is a generalization of the Hamming distance. The Hamming distance compares the number of differences in the two strings. It compares the i th position of string S_1 with the i th position of string S_2 . Example:

$$\begin{array}{l} S_1 = A \ T \ A \ T \ A \ T \ A \ T \\ \quad \downarrow \ \downarrow \\ S_2 = \textcolor{red}{T} \ A \ T \ A \ T \ A \ T \ A \end{array}$$

- The Hamming distance $d(S_1, S_2) = 8$.
- The edit distance, on the other hand, allows insertions/deletions/substitutions of characters. Example:

$$\begin{array}{l} S_1 = - \ A \ T \ A \ T \ A \ T \ A \ T \ A \ T \\ \quad \downarrow \ \downarrow \\ S_2 = \textcolor{red}{T} \ A \ T \ A \ T \ A \ T \ A \ - \end{array}$$

- The Edit distance $Ed(S_1, S_2) = 2$: insert T at the front and remove T at the end of S_1 to turn it into S_2

Edit Distance

Idea: Imagine we have two strings X , of length n , and Y , of length m .

Edit Distance

Idea: Imagine we have two strings X , of length n , and Y , of length m .

Let $M[n, m]$ represent the edit distance between the strings $X[1 \dots n]$ and $Y[1 \dots m]$.

Edit Distance

Idea: Imagine we have two strings X , of length n , and Y , of length m .

Let $M[n, m]$ represent the edit distance between the strings $X[1 \dots n]$ and $Y[1 \dots m]$.

We essentially do a character by character comparison, but one that allows for insertions/deletions/substitutions. Similar to coin-row problem, we want to reuse solutions to the edit distance between substrings of X and Y . But how?

Edit Distance

Idea: Imagine we have two strings X , of length n , and Y , of length m .

Let $M[n, m]$ represent the edit distance between the strings $X[1 \dots n]$ and $Y[1 \dots m]$.

We essentially do a character by character comparison, but one that allows for insertions/deletions/substitutions. Similar to coin-row problem, we want to reuse solutions to the edit distance between substrings of X and Y . But how?

Lets study how to match $X[1 \dots n]$ and $Y[1 \dots m]$ and develop a recurrence relationship that will tell us how.

Edit Distance

Idea: Consider the following scenarios when computing the edit distance between X and Y and we are comparing the final characters of the strings, $X[n]$ and $Y[m]$:

Edit Distance

Idea: Consider the following scenarios when computing the edit distance between X and Y and we are comparing the final characters of the strings, $X[n]$ and $Y[m]$:

Case 1 ($X[n] == Y[m]$): This means the last characters **match**, so no additional distance to add to total edit distance so far.

- Hence, edit distance $M[n, m] = \text{edit distance of matching the substrings } X[1 \dots n - 1] \text{ and } Y[1 \dots m - 1].$

Edit Distance

Idea: Consider the following scenarios when computing the edit distance between X and Y and we are comparing the final characters of the strings, $X[n]$ and $Y[m]$:

Case 1 ($X[n] == Y[m]$): This means the last characters **match**, so no additional distance to add to total edit distance so far.

- Hence, edit distance $M[n, m] = \text{edit distance of matching the substrings } X[1 \dots n - 1] \text{ and } Y[1 \dots m - 1]$.
- Or, $M[n, m] = M[n - 1, m - 1]$.

Edit Distance

Idea: Consider the following scenarios when computing the edit distance between X and Y and we are comparing the final characters of the strings, $X[n]$ and $Y[m]$:

Case 1 ($X[n] == Y[m]$): This means the last characters **match**, so no additional distance to add to total edit distance so far.

- Hence, edit distance $M[n, m] = \text{edit distance of matching the substrings } X[1 \dots n - 1] \text{ and } Y[1 \dots m - 1]$.
- Or, $M[n, m] = M[n - 1, m - 1]$.
- E.g., matching strings

$$\begin{array}{rcl} X & = & a \ b \ c \\ Y & = & u \ v \ c \end{array}$$

Edit Distance

Case 2 ($X[n] \neq Y[m]$): This means the last characters **mismatch**, and we need to apply one of the edit operations. Which one to choose, and which calculated edit distance to reuse?

- Delete $X[n]$ from X , at the cost of 1 unit to total edit distance.
Hence $M[n, m] = 1 + M[n - 1, m]$.

$$\begin{array}{rcl} X & = & a \ b \ d \\ Y & = & a \ b \end{array}$$

Edit Distance

Case 2 ($X[n] \neq Y[m]$): This means the last characters **mismatch**, and we need to apply one of the edit operations. Which one to choose, and which calculated edit distance to reuse?

- Delete $X[n]$ from X , at the cost of 1 unit to total edit distance.
Hence $M[n, m] = 1 + M[n - 1, m]$.

$$\begin{array}{rcl} X & = & a \ b \ d \\ Y & = & a \ b \end{array}$$

- Insert $Y[n]$ into X . Hence $M[n, m] = 1 + M[n, m - 1]$.

$$\begin{array}{rcl} X & = & a \ b \\ Y & = & a \ b \ d \end{array}$$

Edit Distance

Case 2 ($X[n] \neq Y[m]$): This means the last characters mismatch, and we need to apply one of the edit operations. Which one to choose, and which calculated edit distance to reuse?

- Delete $X[n]$ from X , at the cost of 1 unit to total edit distance.
Hence $M[n, m] = 1 + M[n - 1, m]$.

$$\begin{array}{rcl} X & = & a \ b \ d \\ Y & = & a \ b \end{array}$$

- Insert $Y[n]$ into X . Hence $M[n, m] = 1 + M[n, m - 1]$.

$$\begin{array}{rcl} X & = & a \ b \\ Y & = & a \ b \ d \end{array}$$

- Substitute $Y[m]$ for $X[n]$, so the characters now match. Hence $M[n, m] = 1 + M[n - 1, m - 1]$.

$$\begin{array}{rcl} X & = & a \ b \ d \\ Y & = & a \ b \ e \end{array}$$

Edit Distance

Recall $M[n, m]$ represents the edit distance (minimum number of edit operations) needed to match strings $X[1 \dots n]$ and $Y[1 \dots m]$.

Edit Distance

Recall $M[n, m]$ represents the edit distance (minimum number of edit operations) needed to match strings $X[1 \dots n]$ and $Y[1 \dots m]$.

Computing the Edit Distance

$$M[n - 1, m - 1] \quad \text{if } X[n] = Y[m]$$

$$M[n, m] = \begin{cases} 1 + \min(M[n - 1, m - 1], M[n - 1, m], M[n, m - 1]) \\ \quad \quad \quad \text{otherwise} \end{cases}$$

$$M[n, 0] = n, M[0, m] = m$$

Edit Distance

Recall $M[n, m]$ represents the edit distance (minimum number of edit operations) needed to match strings $X[1 \dots n]$ and $Y[1 \dots m]$.

Computing the Edit Distance

$$M[n - 1, m - 1] \quad \text{if } X[n] = Y[m]$$

$$M[n, m] = \begin{cases} 1 + \min(M[n - 1, m - 1], M[n - 1, m], M[n, m - 1]) \\ \quad \quad \quad \text{otherwise} \end{cases}$$

$$M[n, 0] = n, M[0, m] = m$$

Recursive definition! Similar to row-coin problem, we build a **dynamic programming table/matrix** for $M[n, m]$.

Edit Distance

Recall $M[n, m]$ represents the edit distance (minimum number of edit operations) needed to match strings $X[1 \dots n]$ and $Y[1 \dots m]$.

Computing the Edit Distance

$$M[n - 1, m - 1] \quad \text{if } X[n] = Y[m]$$

$$M[n, m] = \begin{cases} 1 + \min(M[n - 1, m - 1], M[n - 1, m], M[n, m - 1]) \\ \quad \quad \quad \text{otherwise} \end{cases}$$

$$M[n, 0] = n, M[0, m] = m$$

Recursive definition! Similar to row-coin problem, we build a **dynamic programming table/matrix** for $M[n, m]$.

I.e., To compute $M[n, m]$, a table $M[1 \dots n, 1 \dots m]$ is computed and filled.

Edit Distance - Example

	a	n	n	e	a	l	i	n	g
0	1	2	3	4	5	6	7	8	9
a	1								
n		2							
n			3						
u				4					
a					5				
l						6			

Compute $Ed(\text{annual}, \text{annealing})$.

Edit Distance - Example

		a	n	n	e	a	l	i	n	g
	0	1	2	3	4	5	6	7	8	9
a	1	0								
n	2									
n	3									
u	4									
a	5									
l	6									

Compute $Ed(\text{annual}, \text{annealing})$.

Edit Distance - Example

	a	n	n	e	a	l	i	n	g
0	1	2	3	4	5	6	7	8	9
a	1	0	1						
n	2								
n	3								
u	4								
a	5								
l	6								

Compute $Ed(\text{annual}, \text{annealing})$.

Edit Distance - Example

	a	n	n	e	a	l	i	n	g
0	1	2	3	4	5	6	7	8	9
a	1	0	1	2					
n		2							
n		3							
u		4							
a		5							
l		6							

Compute $Ed(\text{annual}, \text{annealing})$.

Edit Distance - Example

	a	n	n	e	a	l	i	n	g
0	1	2	3	4	5	6	7	8	9
a	1	0	1	2	3				
n	2								
n	3								
u	4								
a	5								
l	6								

Compute $Ed(\text{annual}, \text{annealing})$.

Edit Distance - Example

	a	n	n	e	a	l	i	n	g
0	1	2	3	4	5	6	7	8	9
a	1	0	1	2	3	4			
n	2								
n	3								
u	4								
a	5								
l	6								

Compute $Ed(\text{annual}, \text{annealing})$.

Edit Distance - Example

	a	n	n	e	a	l	i	n	g
0	1	2	3	4	5	6	7	8	9
a	1	0	1	2	3	4	5		
n	2								
n	3								
u	4								
a	5								
l	6								

Compute $Ed(\text{annual}, \text{annealing})$.

Edit Distance - Example

	a	n	n	e	a	l	i	n	g
0	1	2	3	4	5	6	7	8	9
a	1	0	1	2	3	4	5	6	
n	2								
n	3								
u	4								
a	5								
l	6								

Compute $Ed(\text{annual}, \text{annealing})$.

Edit Distance - Example

	a	n	n	e	a	l	i	n	g
0	1	2	3	4	5	6	7	8	9
a	1	0	1	2	3	4	5	6	7
n	2								
n	3								
u	4								
a	5								
l	6								

Compute $Ed(\text{annual}, \text{annealing})$.

Edit Distance - Example

	a	n	n	e	a	l	i	n	g
0	1	2	3	4	5	6	7	8	9
a	1	0	1	2	3	4	5	6	7
n	2								8
n	3								
u	4								
a	5								
l	6								

Compute $Ed(\text{annual}, \text{annealing})$.

Edit Distance - Example

	a	n	n	e	a	l	i	n	g
0	1	2	3	4	5	6	7	8	9
a	1	0	1	2	3	4	5	6	7
n	2	1							
n	3								
u	4								
a	5								
l	6								

Compute $Ed(\text{annual}, \text{annealing})$.

Edit Distance - Example

	a	n	n	e	a	l	i	n	g
0	1	2	3	4	5	6	7	8	9
a	1	0	1	2	3	4	5	6	7
n	2	1	0						
n	3								
u	4								
a	5								
l	6								

Compute $Ed(\text{annual}, \text{annealing})$.

Edit Distance - Example

	a	n	n	e	a	l	i	n	g
0	1	2	3	4	5	6	7	8	9
a	1	0	1	2	3	4	5	6	7
n	2	1	0	1					
n	3								
u	4								
a	5								
l	6								

Compute $Ed(\text{annual}, \text{annealing})$.

Edit Distance - Example

	a	n	n	e	a	l	i	n	g
0	1	2	3	4	5	6	7	8	9
a	1	0	1	2	3	4	5	6	7
n	2	1	0	1	2				
n	3								
u	4								
a	5								
l	6								

Compute $Ed(\text{annual}, \text{annealing})$.

Edit Distance - Example

	a	n	n	e	a	l	i	n	g
0	1	2	3	4	5	6	7	8	9
a	1	0	1	2	3	4	5	6	7
n	2	1	0	1	2	3			
n									
u									
a									
l									

Compute $Ed(\text{annual}, \text{annealing})$.

Edit Distance - Example

And so on ...

Edit Distance - Example

	a	n	n	e	a	l	i	n	g
0	1	2	3	4	5	6	7	8	9
a	1	0	1	2	3	4	5	6	7
n	2	1	0	1	2	3	4	5	6
n	3	2	1	0	1	2	3	4	5
u	4	3	2	1	1	2	3	4	5
a	5	4	3	2	2	1	2	3	4
l	6	5	4	3	3	2	1	2	3

Compute $Ed(\text{annual}, \text{annealing})$.

Edit Distance - Backtrace

- In addition to minimum edit distance, we would like to know what is the sequence of operations (insertion, deletion, substitution, match) to obtain this edit distance and how the strings align.
- Use **backtrace**

Edit Distance - Backtrace

- In addition to minimum edit distance, we would like to know what is the sequence of operations (insertion, deletion, substitution, match) to obtain this edit distance and how the strings align.
- Use **backtrace**
- From (m,n) cell (bottom right corner of table), evaluate which operation and adjacent cell we used to arrive to (m,n) . Repeat this process until traverse to $(0,0)$ cell.

Edit Distance - Backtrace

- In addition to minimum edit distance, we would like to know what is the sequence of operations (insertion, deletion, substitution, match) to obtain this edit distance and how the strings align.
- Use **backtrace**
- From (m,n) cell (bottom right corner of table), evaluate which operation and adjacent cell we used to arrive to (m,n) . Repeat this process until traverse to $(0,0)$ cell.
- This produces a path from $(0,0)$ to (m,n) that is non-decreasing in edit distance.

Edit Distance - Backtrace

- In addition to minimum edit distance, we would like to know what is the sequence of operations (insertion, deletion, substitution, match) to obtain this edit distance and how the strings align.
- Use **backtrace**
- From (m,n) cell (bottom right corner of table), evaluate which operation and adjacent cell we used to arrive to (m,n) . Repeat this process until traverse to $(0,0)$ cell.
- This produces a path from $(0,0)$ to (m,n) that is non-decreasing in edit distance.
- May not be unique (i.e., several alignments/sequence of operations) lead to same edit distance. This means multiple backtraces.

Edit Distance - Backtrace

	a	n	n	e	a	l	i	n	g
0	1	2	3	4	5	6	7	8	9
a	1	0	1	2	3	4	5	6	7
n	2	1	0	1	2	3	4	5	6
n	3	2	1	0	1	2	3	4	5
u	4	3	2	1	1	2	3	4	5
a	5	4	3	2	2	1	2	3	4
l	6	5	4	3	3	2	1	2	3

MMMSMMIII

'M' - Match, 'S' - Substitution, 'I' - Insertion, 'D' - Deletion

Compute Ed (annual, annealing).

Edit Distance

- The time complexity for table construction is $\Theta(nm)$ where $|S_1| = n$ and $|S_2| = m$. Backtrace's worst-case complexity is $\Theta(m + n)$ (zigzag).
The solution also uses $\Theta(nm)$ space.

Edit Distance

- The time complexity for table construction is $\Theta(nm)$ where $|S_1| = n$ and $|S_2| = m$. Backtrace's worst-case complexity is $\Theta(m + n)$ (zigzag).
The solution also uses $\Theta(nm)$ space.
- DEMO: <http://www.let.rug.nl/kleiweg/lev/>
- The canonical reference on Edit Distance is: D. Gusfield.
Algorithms on Strings, Trees, and Sequences. Cambridge University Press, New York, New York, USA, 1997.

Overview

1 Overview

2 Edit Distance

3 Knapsack Problem

4 Warshall's Algorithm

5 Summary

Knapsack Problem

Knapsack Problem

Given n items of known weights w_1, \dots, w_n and the values v_1, \dots, v_n and a knapsack of capacity W , find the most valuable subset of the items that fit into the knapsack.

- Recall that the exact solution for all instances of this problem has been proven to be $\Omega(2^n)$.
- We can solve the problem using dynamic programming in “pseudo-polynomial” time.

DP Knapsack Problem - Sketch

- Consider an instance of the knapsack problem defined by the first i items, $1 \leq i \leq n$, with weights $w_1 \dots w_i$, values $v_1 \dots v_i$, and capacity j , $1 \leq j \leq W$.

DP Knapsack Problem - Sketch

- Consider an instance of the knapsack problem defined by the first i items, $1 \leq i \leq n$, with weights $w_1 \dots w_i$, values $v_1 \dots v_i$, and capacity j , $1 \leq j \leq W$.
- Let $V[i, j]$ be an optimal value to the subproblem instance of having the first i items and a knapsack capacity of j .

DP Knapsack Problem - Sketch

- Consider we are solving a (sub)instance of the knapsack problem $V[i, j]$
- We can **divide** all the **subsets** of the first i items that fit into the knapsack of capacity j into two categories:

DP Knapsack Problem - Sketch

- Consider we are solving a (sub)instance of the knapsack problem $V[i, j]$
- We can **divide** all the **subsets** of the first i items that fit into the knapsack of capacity j into two categories:
 - Among the subsets that **do not** include the i -th item, the value of the optimal subset is, by definition, $V[i - 1, j]$

DP Knapsack Problem - Sketch

- Consider we are solving a (sub)instance of the knapsack problem $V[i, j]$
- We can **divide** all the **subsets** of the first i items that fit into the knapsack of capacity j into two categories:
 - Among the subsets that **do not** include the i -th item, the value of the optimal subset is, by definition, $V[i - 1, j]$
 - Among the subsets that **do** include the i -th item ($j - w_i \geq 0$), an optimal subset is made up of this item and an optimal subset of the first $i - 1$ items that fit into the knapsack of capacity $j - w_i$. The value of such an optimal subset is $v_i + V[i - 1, j - w_i]$.

DP Knapsack Problem - Sketch

- Consider we are solving a (sub)instance of the knapsack problem $V[i, j]$
- We can **divide** all the **subsets** of the first i items that fit into the knapsack of capacity j into two categories:
 - Among the subsets that **do not** include the i -th item, the value of the optimal subset is, by definition, $V[i - 1, j]$
 - Among the subsets that **do** include the i -th item ($j - w_i \geq 0$), an optimal subset is made up of this item and an optimal subset of the first $i - 1$ items that fit into the knapsack of capacity $j - w_i$. The value of such an optimal subset is $v_i + V[i - 1, j - w_i]$.
- Whether we choose to include i -th item dependent on whether the i -th item can fit into knapsack and if so, which option leads to larger value ($V[i, j]$).

DP Knapsack Problem - Sketch

This leads to the following recursion:

$$V[i, j] = \begin{cases} \max(V[i - 1, j], v_i + V[i - 1, j - w_i]) & \text{if } j - w_i \geq 0, \\ V[i - 1, j] & \text{if } j - w_i < 0. \end{cases}$$

$$V[0, j] = 0 \text{ for } j \geq 0 \text{ and } V[i, 0] = 0 \text{ for } i \geq 0.$$

Bottom-Up DP algorithm

Bottom-up Dynamic Programming: What we have been doing up to this point, computing solutions to all entries in the dynamic programming table.

Bottom-Up DP algorithm

Bottom-up Dynamic Programming: What we have been doing up to this point, computing solutions to all entries in the dynamic programming table.

For example the dynamic programming solution to the *coin row problem* and calculating the *edit distance*.

Given the following problem, how do we solve it using a Bottom-Up Dynamic Programming algorithm?

Bottom-Up DP algorithm

Bottom-up Dynamic Programming: What we have been doing up to this point, computing solutions to all entries in the dynamic programming table.

For example the dynamic programming solution to the *coin row problem* and calculating the *edit distance*.

Given the following problem, how do we solve it using a Bottom-Up Dynamic Programming algorithm?

Knapsack capacity
 $W = 6$.

i	1	2	3	4	5
weight(w_i)	3	2	1	4	5
value(v_i)	\$25	\$20	\$15	\$40	\$50

Bottom-Up DP algorithm

We record the solutions to each smaller problems in table.

$\downarrow i$	$W \rightarrow$	0	1	2	3	4	5	6
0								
1								
2								
3								
4								
5								GOAL

Bottom-Up DP algorithm

We record the solutions to each smaller problems in table.

$\downarrow i$	$W \rightarrow$	0	1	2	3	4	5	6
0								
1								
2								
3						?		
4								
5								GOAL

$V[3, 4] = ?$ stores the optimal value for a knapsack with capacity 4 of the first 3 items

Bottom-Up DP algorithm

$$V[i, j] = \begin{cases} \max(V[i - 1, j], v_i + V[i - 1, j - w_i]) & \text{if } j - w_i \geq 0, \\ V[i - 1, j] & \text{if } j - w_i < 0. \end{cases}$$

$V[0, j] = 0$ for $j \geq 0$ and $V[i, 0] = 0$ for $i \geq 0$.

$\downarrow i$	$W \rightarrow$	0	1	2	3	4	5	6
		0						
$w_1 = 3$	$v_1 = 25$	1						
$w_2 = 2$	$v_2 = 20$	2						
$w_3 = 1$	$v_3 = 15$	3						
$w_4 = 4$	$v_4 = 40$	4						
$w_5 = 5$	$v_5 = 50$	5						

Bottom-Up DP algorithm

$$V[i, j] = \begin{cases} \max(V[i - 1, j], v_i + V[i - 1, j - w_i]) & \text{if } j - w_i \geq 0, \\ V[i - 1, j] & \text{if } j - w_i < 0. \end{cases}$$

$V[0, j] = 0$ for $j \geq 0$ and $V[i, 0] = 0$ for $i \geq 0$.

$\downarrow i$	$W \rightarrow$	0	1	2	3	4	5	6
	0	0	0	0	0	0	0	0
$w_1 = 3$	$v_1 = 25$	1	0					
$w_2 = 2$	$v_2 = 20$	2	0					
$w_3 = 1$	$v_3 = 15$	3	0					
$w_4 = 4$	$v_4 = 40$	4	0					
$w_5 = 5$	$v_5 = 50$	5	0					

Bottom-Up DP algorithm

$$V[i, j] = \begin{cases} \max(V[i - 1, j], v_i + V[i - 1, j - w_i]) & \text{if } j - w_i \geq 0, \\ V[i - 1, j] & \text{if } j - w_i < 0. \end{cases}$$

$V[0, j] = 0$ for $j \geq 0$ and $V[i, 0] = 0$ for $i \geq 0$.

$\downarrow i$	$W \rightarrow$	0	1	2	3	4	5	6
	0	0	0	0	0	0	0	0
$w_1 = 3$	$v_1 = 25$	1	0	0	0	0	0	0
$w_2 = 2$	$v_2 = 20$	2	0	0	0	0	0	0
$w_3 = 1$	$v_3 = 15$	3	0	0	0	0	0	0
$w_4 = 4$	$v_4 = 40$	4	0	0	0	0	0	0
$w_5 = 5$	$v_5 = 50$	5	0	0	0	0	0	0

Bottom-Up DP algorithm

$$V[i, j] = \begin{cases} \max(V[i - 1, j], v_i + V[i - 1, j - w_i]) & \text{if } j - w_i \geq 0, \\ V[i - 1, j] & \text{if } j - w_i < 0. \end{cases}$$

$V[0, j] = 0$ for $j \geq 0$ and $V[i, 0] = 0$ for $i \geq 0$.

$\downarrow i$	$W \rightarrow$	0	1	2	3	4	5	6
	0	0	0	0	0	0	0	0
$w_1 = 3$	$v_1 = 25$	1	0	0				
$w_2 = 2$	$v_2 = 20$	2	0	0	0			
$w_3 = 1$	$v_3 = 15$	3	0					
$w_4 = 4$	$v_4 = 40$	4	0					
$w_5 = 5$	$v_5 = 50$	5	0					

Bottom-Up DP algorithm

$$V[i, j] = \begin{cases} \max(V[i - 1, j], v_i + V[i - 1, j - w_i]) & \text{if } j - w_i \geq 0, \\ V[i - 1, j] & \text{if } j - w_i < 0. \end{cases}$$

$V[0, j] = 0$ for $j \geq 0$ and $V[i, 0] = 0$ for $i \geq 0$.

$\downarrow i$	$W \rightarrow$	0	1	2	3	4	5	6
	0	0	0	0	0	0	0	0
$w_1 = 3$	$v_1 = 25$	1	0	0				
$w_2 = 2$	$v_2 = 20$	2	0	0				
$w_3 = 1$	$v_3 = 15$	3	0	15				
$w_4 = 4$	$v_4 = 40$	4	0					
$w_5 = 5$	$v_5 = 50$	5	0					

Bottom-Up DP algorithm

$$V[i, j] = \begin{cases} \max(V[i - 1, j], v_i + V[i - 1, j - w_i]) & \text{if } j - w_i \geq 0, \\ V[i - 1, j] & \text{if } j - w_i < 0. \end{cases}$$

$V[0, j] = 0$ for $j \geq 0$ and $V[i, 0] = 0$ for $i \geq 0$.

$\downarrow i$	$W \rightarrow$	0	1	2	3	4	5	6
	0	0	0	0	0	0	0	0
$w_1 = 3$	$v_1 = 25$	1	0	0				
$w_2 = 2$	$v_2 = 20$	2	0	0				
$w_3 = 1$	$v_3 = 15$	3	0	15				
$w_4 = 4$	$v_4 = 40$	4	0	15				
$w_5 = 5$	$v_5 = 50$	5	0	15				

Bottom-Up DP algorithm

$$V[i, j] = \begin{cases} \max(V[i - 1, j], v_i + V[i - 1, j - w_i]) & \text{if } j - w_i \geq 0, \\ V[i - 1, j] & \text{if } j - w_i < 0. \end{cases}$$

$V[0, j] = 0$ for $j \geq 0$ and $V[i, 0] = 0$ for $i \geq 0$.

$\downarrow i$	$W \rightarrow$	0	1	2	3	4	5	6
	0	0	0	0	0	0	0	0
$w_1 = 3$	$v_1 = 25$	1	0	0	0	0	0	0
$w_2 = 2$	$v_2 = 20$	2	0	0	0	0	0	0
$w_3 = 1$	$v_3 = 15$	3	0	15	0	0	0	0
$w_4 = 4$	$v_4 = 40$	4	0	15	0	0	0	0
$w_5 = 5$	$v_5 = 50$	5	0	15	0	0	0	0

Bottom-Up DP algorithm

$$V[i, j] = \begin{cases} \max(V[i - 1, j], v_i + V[i - 1, j - w_i]) & \text{if } j - w_i \geq 0, \\ V[i - 1, j] & \text{if } j - w_i < 0. \end{cases}$$

$V[0, j] = 0$ for $j \geq 0$ and $V[i, 0] = 0$ for $i \geq 0$.

$\downarrow i$	$W \rightarrow$	0	1	2	3	4	5	6
	0	0	0	0	0	0	0	0
$w_1 = 3$	$v_1 = 25$	1	0	0	0			
$w_2 = 2$	$v_2 = 20$	2	0	0	20			
$w_3 = 1$	$v_3 = 15$	3	0	15				
$w_4 = 4$	$v_4 = 40$	4	0	15				
$w_5 = 5$	$v_5 = 50$	5	0	15				

Bottom-Up DP algorithm

$$V[i, j] = \begin{cases} \max(V[i - 1, j], v_i + V[i - 1, j - w_i]) & \text{if } j - w_i \geq 0, \\ V[i - 1, j] & \text{if } j - w_i < 0. \end{cases}$$

$V[0, j] = 0$ for $j \geq 0$ and $V[i, 0] = 0$ for $i \geq 0$.

$\downarrow i$	$W \rightarrow$	0	1	2	3	4	5	6
	0	0	0	0	0	0	0	0
$w_1 = 3$	$v_1 = 25$	1	0	0	0			
$w_2 = 2$	$v_2 = 20$	2	0	0	20			
$w_3 = 1$	$v_3 = 15$	3	0	15	20			
$w_4 = 4$	$v_4 = 40$	4	0	15	20			
$w_5 = 5$	$v_5 = 50$	5	0	15	20			

Bottom-Up DP algorithm

$$V[i, j] = \begin{cases} \max(V[i - 1, j], v_i + V[i - 1, j - w_i]) & \text{if } j - w_i \geq 0, \\ V[i - 1, j] & \text{if } j - w_i < 0. \end{cases}$$

$V[0, j] = 0$ for $j \geq 0$ and $V[i, 0] = 0$ for $i \geq 0$.

$\downarrow i$	$W \rightarrow$	0	1	2	3	4	5	6
	0	0	0	0	0	0	0	0
$w_1 = 3$	$v_1 = 25$	1	0	0	0	25		
$w_2 = 2$	$v_2 = 20$	2	0	0	20			
$w_3 = 1$	$v_3 = 15$	3	0	15	20			
$w_4 = 4$	$v_4 = 40$	4	0	15	20			
$w_5 = 5$	$v_5 = 50$	5	0	15	20			

Bottom-Up DP algorithm

$$V[i, j] = \begin{cases} \max(V[i - 1, j], v_i + V[i - 1, j - w_i]) & \text{if } j - w_i \geq 0, \\ V[i - 1, j] & \text{if } j - w_i < 0. \end{cases}$$

$V[0, j] = 0$ for $j \geq 0$ and $V[i, 0] = 0$ for $i \geq 0$.

$\downarrow i$	$W \rightarrow$	0	1	2	3	4	5	6
	0	0	0	0	0	0	0	0
$w_1 = 3$	$v_1 = 25$	1	0	0	0	25		
$w_2 = 2$	$v_2 = 20$	2	0	0	20	25		
$w_3 = 1$	$v_3 = 15$	3	0	15	20			
$w_4 = 4$	$v_4 = 40$	4	0	15	20			
$w_5 = 5$	$v_5 = 50$	5	0	15	20			

Bottom-Up DP algorithm

$$V[i, j] = \begin{cases} \max(V[i - 1, j], v_i + V[i - 1, j - w_i]) & \text{if } j - w_i \geq 0, \\ V[i - 1, j] & \text{if } j - w_i < 0. \end{cases}$$

$V[0, j] = 0$ for $j \geq 0$ and $V[i, 0] = 0$ for $i \geq 0$.

$\downarrow i$	$W \rightarrow$	0	1	2	3	4	5	6
	0	0	0	0	0	0	0	0
$w_1 = 3$	$v_1 = 25$	1	0	0	0	25		
$w_2 = 2$	$v_2 = 20$	2	0	0	20	25		
$w_3 = 1$	$v_3 = 15$	3	0	15	20	?		
$w_4 = 4$	$v_4 = 40$	4	0	15	20			
$w_5 = 5$	$v_5 = 50$	5	0	15	20			

Bottom-Up DP algorithm

$$V[i, j] = \begin{cases} \max(V[i - 1, j], v_i + V[i - 1, j - w_i]) & \text{if } j - w_i \geq 0, \\ V[i - 1, j] & \text{if } j - w_i < 0. \end{cases}$$

$V[0, j] = 0$ for $j \geq 0$ and $V[i, 0] = 0$ for $i \geq 0$.

$\downarrow i$	$W \rightarrow$	0	1	2	3	4	5	6
	0	0	0	0	0	0	0	0
$w_1 = 3$	$v_1 = 25$	1	0	0	0	25		
$w_2 = 2$	$v_2 = 20$	2	0	0	20	25		
$w_3 = 1$	$v_3 = 15$	3	0	15	20	?		
$w_4 = 4$	$v_4 = 40$	4	0	15	20			
$w_5 = 5$	$v_5 = 50$	5	0	15	20			

Bottom-Up DP algorithm

$$V[i, j] = \begin{cases} \max(V[i - 1, j], v_i + V[i - 1, j - w_i]) & \text{if } j - w_i \geq 0, \\ V[i - 1, j] & \text{if } j - w_i < 0. \end{cases}$$

$V[0, j] = 0$ for $j \geq 0$ and $V[i, 0] = 0$ for $i \geq 0$.

$\downarrow i$	$W \rightarrow$	0	1	2	3	4	5	6
	0	0	0	0	0	0	0	0
$w_1 = 3$	$v_1 = 25$	1	0	0	0	25		
$w_2 = 2$	$v_2 = 20$	2	0	0	20	25		
$w_3 = 1$	$v_3 = 15$	3	0	15	20			
$w_4 = 4$	$v_4 = 40$	4	0	15	20			
$w_5 = 5$	$v_5 = 50$	5	0	15	20			

Bottom-Up DP algorithm

$$V[i, j] = \begin{cases} \max(V[i - 1, j], v_i + V[i - 1, j - w_i]) & \text{if } j - w_i \geq 0, \\ V[i - 1, j] & \text{if } j - w_i < 0. \end{cases}$$

$V[0, j] = 0$ for $j \geq 0$ and $V[i, 0] = 0$ for $i \geq 0$.

$\downarrow i$	$W \rightarrow$	0	1	2	3	4	5	6
	0	0	0	0	0	0	0	0
$w_1 = 3$	$v_1 = 25$	1	0	0	0	25		
$w_2 = 2$	$v_2 = 20$	2	0	0	20	25		
$w_3 = 1$	$v_3 = 15$	3	0	15	20	35		
$w_4 = 4$	$v_4 = 40$	4	0	15	20			
$w_5 = 5$	$v_5 = 50$	5	0	15	20			

Bottom-Up DP algorithm

$$V[i, j] = \begin{cases} \max(V[i - 1, j], v_i + V[i - 1, j - w_i]) & \text{if } j - w_i \geq 0, \\ V[i - 1, j] & \text{if } j - w_i < 0. \end{cases}$$

$V[0, j] = 0$ for $j \geq 0$ and $V[i, 0] = 0$ for $i \geq 0$.

$\downarrow i$	$W \rightarrow$	0	1	2	3	4	5	6
	0	0	0	0	0	0	0	0
$w_1 = 3$	$v_1 = 25$	1	0	0	0	25		
$w_2 = 2$	$v_2 = 20$	2	0	0	20	25		
$w_3 = 1$	$v_3 = 15$	3	0	15	20	35		
$w_4 = 4$	$v_4 = 40$	4	0	15	20	35		
$w_5 = 5$	$v_5 = 50$	5	0	15	20	35		

Bottom-Up DP algorithm

$$V[i, j] = \begin{cases} \max(V[i - 1, j], v_i + V[i - 1, j - w_i]) & \text{if } j - w_i \geq 0, \\ V[i - 1, j] & \text{if } j - w_i < 0. \end{cases}$$

$V[0, j] = 0$ for $j \geq 0$ and $V[i, 0] = 0$ for $i \geq 0$.

$\downarrow i$	$W \rightarrow$	0	1	2	3	4	5	6
	0	0	0	0	0	0	0	0
$w_1 = 3$	$v_1 = 25$	1	0	0	0	25	25	
$w_2 = 2$	$v_2 = 20$	2	0	0	20	25	25	
$w_3 = 1$	$v_3 = 15$	3	0	15	20	35		
$w_4 = 4$	$v_4 = 40$	4	0	15	20	35		
$w_5 = 5$	$v_5 = 50$	5	0	15	20	35		

Bottom-Up DP algorithm

$$V[i, j] = \begin{cases} \max(V[i - 1, j], v_i + V[i - 1, j - w_i]) & \text{if } j - w_i \geq 0, \\ V[i - 1, j] & \text{if } j - w_i < 0. \end{cases}$$

$V[0, j] = 0$ for $j \geq 0$ and $V[i, 0] = 0$ for $i \geq 0$.

$\downarrow i$	$W \rightarrow$	0	1	2	3	4	5	6
	0	0	0	0	0	0	0	0
$w_1 = 3$	$v_1 = 25$	1	0	0	0	25	25	
$w_2 = 2$	$v_2 = 20$	2	0	0	20	25	25	
$w_3 = 1$	$v_3 = 15$	3	0	15	20	35	?	
$w_4 = 4$	$v_4 = 40$	4	0	15	20	35		
$w_5 = 5$	$v_5 = 50$	5	0	15	20	35		

Bottom-Up DP algorithm

$$V[i, j] = \begin{cases} \max(V[i - 1, j], v_i + V[i - 1, j - w_i]) & \text{if } j - w_i \geq 0, \\ V[i - 1, j] & \text{if } j - w_i < 0. \end{cases}$$

$V[0, j] = 0$ for $j \geq 0$ and $V[i, 0] = 0$ for $i \geq 0$.

$\downarrow i$	$W \rightarrow$	0	1	2	3	4	5	6
	0	0	0	0	0	0	0	0
$w_1 = 3$	$v_1 = 25$	1	0	0	0	25	25	
$w_2 = 2$	$v_2 = 20$	2	0	0	20	25	25	
$w_3 = 1$	$v_3 = 15$	3	0	15	20	35	?	
$w_4 = 4$	$v_4 = 40$	4	0	15	20	35		
$w_5 = 5$	$v_5 = 50$	5	0	15	20	35		

Bottom-Up DP algorithm

$$V[i, j] = \begin{cases} \max(V[i - 1, j], v_i + V[i - 1, j - w_i]) & \text{if } j - w_i \geq 0, \\ V[i - 1, j] & \text{if } j - w_i < 0. \end{cases}$$

$V[0, j] = 0$ for $j \geq 0$ and $V[i, 0] = 0$ for $i \geq 0$.

$\downarrow i$	$W \rightarrow$	0	1	2	3	4	5	6
	0	0	0	0	0	0	0	0
$w_1 = 3$	$v_1 = 25$	1	0	0	0	25	25	
$w_2 = 2$	$v_2 = 20$	2	0	0	20	25	25	
$w_3 = 1$	$v_3 = 15$	3	0	15	20	35		
$w_4 = 4$	$v_4 = 40$	4	0	15	20	35		
$w_5 = 5$	$v_5 = 50$	5	0	15	20	35		

Bottom-Up DP algorithm

$$V[i, j] = \begin{cases} \max(V[i - 1, j], v_i + V[i - 1, j - w_i]) & \text{if } j - w_i \geq 0, \\ V[i - 1, j] & \text{if } j - w_i < 0. \end{cases}$$

$V[0, j] = 0$ for $j \geq 0$ and $V[i, 0] = 0$ for $i \geq 0$.

$\downarrow i$	$W \rightarrow$	0	1	2	3	4	5	6
	0	0	0	0	0	0	0	0
$w_1 = 3$	$v_1 = 25$	1	0	0	0	25	25	
$w_2 = 2$	$v_2 = 20$	2	0	0	20	25	25	
$w_3 = 1$	$v_3 = 15$	3	0	15	20	35	40	
$w_4 = 4$	$v_4 = 40$	4	0	15	20	35		
$w_5 = 5$	$v_5 = 50$	5	0	15	20	35		

Bottom-Up DP algorithm

$$V[i, j] = \begin{cases} \max(V[i - 1, j], v_i + V[i - 1, j - w_i]) & \text{if } j - w_i \geq 0, \\ V[i - 1, j] & \text{if } j - w_i < 0. \end{cases}$$

$V[0, j] = 0$ for $j \geq 0$ and $V[i, 0] = 0$ for $i \geq 0$.

$\downarrow i$	$W \rightarrow$	0	1	2	3	4	5	6
	0	0	0	0	0	0	0	0
$w_1 = 3$	$v_1 = 25$	1	0	0	0	25	25	
$w_2 = 2$	$v_2 = 20$	2	0	0	20	25	25	
$w_3 = 1$	$v_3 = 15$	3	0	15	20	35	40	
$w_4 = 4$	$v_4 = 40$	4	0	15	20	35	40	
$w_5 = 5$	$v_5 = 50$	5	0	15	20	35	40	

Bottom-Up DP algorithm

$$V[i, j] = \begin{cases} \max(V[i - 1, j], v_i + V[i - 1, j - w_i]) & \text{if } j - w_i \geq 0, \\ V[i - 1, j] & \text{if } j - w_i < 0. \end{cases}$$

$V[0, j] = 0$ for $j \geq 0$ and $V[i, 0] = 0$ for $i \geq 0$.

$\downarrow i$	$W \rightarrow$	0	1	2	3	4	5	6
	0	0	0	0	0	0	0	0
$w_1 = 3$	$v_1 = 25$	1	0	0	0	25	25	25
$w_2 = 2$	$v_2 = 20$	2	0	0	20	25	25	45
$w_3 = 1$	$v_3 = 15$	3	0	15	20	35	40	45
$w_4 = 4$	$v_4 = 40$	4	0	15	20	35	40	
$w_5 = 5$	$v_5 = 50$	5	0	15	20	35	40	

Bottom-Up DP algorithm

$$V[i, j] = \begin{cases} \max(V[i - 1, j], v_i + V[i - 1, j - w_i]) & \text{if } j - w_i \geq 0, \\ V[i - 1, j] & \text{if } j - w_i < 0. \end{cases}$$

$V[0, j] = 0$ for $j \geq 0$ and $V[i, 0] = 0$ for $i \geq 0$.

$\downarrow i$	$W \rightarrow$	0	1	2	3	4	5	6
	0	0	0	0	0	0	0	0
$w_1 = 3$	$v_1 = 25$	1	0	0	0	25	25	25
$w_2 = 2$	$v_2 = 20$	2	0	0	20	25	25	45
$w_3 = 1$	$v_3 = 15$	3	0	15	20	35	40	45
$w_4 = 4$	$v_4 = 40$	4	0	15	20	35	40	55
$w_5 = 5$	$v_5 = 50$	5	0	15	20	35	40	

Bottom-Up DP algorithm

$$V[i, j] = \begin{cases} \max(V[i - 1, j], v_i + V[i - 1, j - w_i]) & \text{if } j - w_i \geq 0, \\ V[i - 1, j] & \text{if } j - w_i < 0. \end{cases}$$

$V[0, j] = 0$ for $j \geq 0$ and $V[i, 0] = 0$ for $i \geq 0$.

$\downarrow i$	$W \rightarrow$	0	1	2	3	4	5	6
	0	0	0	0	0	0	0	0
$w_1 = 3$	$v_1 = 25$	1	0	0	0	25	25	25
$w_2 = 2$	$v_2 = 20$	2	0	0	20	25	25	45
$w_3 = 1$	$v_3 = 15$	3	0	15	20	35	40	45
$w_4 = 4$	$v_4 = 40$	4	0	15	20	35	40	55
$w_5 = 5$	$v_5 = 50$	5	0	15	20	35	40	

Bottom-Up DP algorithm

$$V[i, j] = \begin{cases} \max(V[i - 1, j], v_i + V[i - 1, j - w_i]) & \text{if } j - w_i \geq 0, \\ V[i - 1, j] & \text{if } j - w_i < 0. \end{cases}$$

$V[0, j] = 0$ for $j \geq 0$ and $V[i, 0] = 0$ for $i \geq 0$.

$\downarrow i$	$W \rightarrow$	0	1	2	3	4	5	6
	0	0	0	0	0	0	0	0
$w_1 = 3$	$v_1 = 25$	1	0	0	0	25	25	25
$w_2 = 2$	$v_2 = 20$	2	0	0	20	25	25	45
$w_3 = 1$	$v_3 = 15$	3	0	15	20	35	40	45
$w_4 = 4$	$v_4 = 40$	4	0	15	20	35	40	55
$w_5 = 5$	$v_5 = 50$	5	0	15	20	35	40	55

Bottom-Up DP algorithm

$$V[i, j] = \begin{cases} \max(V[i - 1, j], v_i + V[i - 1, j - w_i]) & \text{if } j - w_i \geq 0, \\ V[i - 1, j] & \text{if } j - w_i < 0. \end{cases}$$

$V[0, j] = 0$ for $j \geq 0$ and $V[i, 0] = 0$ for $i \geq 0$.

$\downarrow i$	$W \rightarrow$	0	1	2	3	4	5	6
	0	0	0	0	0	0	0	0
$w_1 = 3$	$v_1 = 25$	1	0	0	0	25	25	25
$w_2 = 2$	$v_2 = 20$	2	0	0	20	25	25	45
$w_3 = 1$	$v_3 = 15$	3	0	15	20	35	40	45
$w_4 = 4$	$v_4 = 40$	4	0	15	20	35	40	55
$w_5 = 5$	$v_5 = 50$	5	0	15	20	35	40	55

Bottom-Up DP algorithm

$$V[i, j] = \begin{cases} \max(V[i - 1, j], v_i + V[i - 1, j - w_i]) & \text{if } j - w_i \geq 0, \\ V[i - 1, j] & \text{if } j - w_i < 0. \end{cases}$$

$V[0, j] = 0$ for $j \geq 0$ and $V[i, 0] = 0$ for $i \geq 0$.

$\downarrow i$	$W \rightarrow$	0	1	2	3	4	5	6
	0	0	0	0	0	0	0	0
$w_1 = 3$	$v_1 = 25$	1	0	0	0	25	25	25
$w_2 = 2$	$v_2 = 20$	2	0	0	20	25	25	45
$w_3 = 1$	$v_3 = 15$	3	0	15	20	35	40	45
$w_4 = 4$	$v_4 = 40$	4	0	15	20	35	40	55
$w_5 = 5$	$v_5 = 50$	5	0	15	20	35	40	55
								65

Bottom-Up DP algorithm - Backtrace

How to find the set of items to include? Use [backtrace](#), similar to edit distance and coin-row problem.

Bottom-Up DP algorithm - Backtrace

How to find the set of items to include? Use **backtrace**, similar to edit distance and coin-row problem.

- ① From $V[n, W]$, trace back how we arrived at this table cell - either from $V[n - 1, W]$ or $V[n - 1, W - w_n]$.
- ② Repeat this step until reach $V[0, 0]$.
- ③ Items that were included in the backtrace form the final solution for knapsack problem.

Bottom-Up DP algorithm - Backtrace

$\downarrow i$	$W \rightarrow$	0	1	2	3	4	5	6
	0	0	0	0	0	0	0	0
$w_1 = 3 v_1 = 25$	1	0	0	0	25	25	25	25
$w_2 = 2 v_2 = 20$	2	0	0	20	25	25	45	45
$w_3 = 1 v_3 = 15$	3	0	15	20	35	40	45	60
$w_4 = 4 v_4 = 40$	4	0	15	20	35	40	55	60
$w_5 = 5 v_5 = 50$	5	0	15	20	35	40	55	65

Lets do the backtrace!

Bottom-Up DP algorithm - Backtrace

$\downarrow i$	$W \rightarrow$	0	1	2	3	4	5	6
	0	0	0	0	0	0	0	0
$w_1 = 3 v_1 = 25$	1	0	0	0	25	25	25	25
$w_2 = 2 v_2 = 20$	2	0	0	20	25	25	45	45
$w_3 = 1 v_3 = 15$	3	0	15	20	35	40	45	60
$w_4 = 4 v_4 = 40$	4	0	15	20	35	40	55	60
$w_5 = 5 v_5 = 50$	5	0	15	20	35	40	55	65

Bottom-Up DP algorithm - Backtrace

$\downarrow i$	$W \rightarrow$	0	1	2	3	4	5	6
	0	0	0	0	0	0	0	0
$w_1 = 3 v_1 = 25$	1	0	0	0	25	25	25	25
$w_2 = 2 v_2 = 20$	2	0	0	20	25	25	45	45
$w_3 = 1 v_3 = 15$	3	0	15	20	35	40	45	60
$w_4 = 4 v_4 = 40$	4	0	15	20	35	40	55	60
$w_5 = 5 v_5 = 50$	5	0	15	20	35	40	55	65

Bottom-Up DP algorithm - Backtrace

$\downarrow i$	$W \rightarrow$	0	1	2	3	4	5	6
	0	0	0	0	0	0	0	0
$w_1 = 3 v_1 = 25$	1	0	0	0	25	25	25	25
$w_2 = 2 v_2 = 20$	2	0	0	20	25	25	45	45
$w_3 = 1 v_3 = 15$	3	0	15	20	35	40	45	60
$w_4 = 4 v_4 = 40$	4	0	15	20	35	40	55	60
$w_5 = 5 v_5 = 50$	5	0	15	20	35	40	55	65

Bottom-Up DP algorithm - Backtrace

$\downarrow i$	$W \rightarrow$	0	1	2	3	4	5	6
	0	0	0	0	0	0	0	0
$w_1 = 3 v_1 = 25$	1	0	0	0	25	25	25	25
$w_2 = 2 v_2 = 20$	2	0	0	20	25	25	45	45
$w_3 = 1 v_3 = 15$	3	0	15	20	35	40	45	60
$w_4 = 4 v_4 = 40$	4	0	15	20	35	40	55	60
$w_5 = 5 v_5 = 50$	5	0	15	20	35	40	55	65

Bottom-Up DP algorithm - Backtrace

$\downarrow i$	$W \rightarrow$	0	1	2	3	4	5	6
	0	0	0	0	0	0	0	0
$w_1 = 3 v_1 = 25$	1	0	0	0	25	25	25	25
$w_2 = 2 v_2 = 20$	2	0	0	20	25	25	45	45
$w_3 = 1 v_3 = 15$	3	0	15	20	35	40	45	60
$w_4 = 4 v_4 = 40$	4	0	15	20	35	40	55	60
$w_5 = 5 v_5 = 50$	5	0	15	20	35	40	55	65

Question: In general, using the dynamic programming table, how can we tell if there is multiple optimal solutions to a Knapsack problem?

DP Knapsack Problem

- The complexity of constructing the dynamic table is $\Theta(nW)$ in time and space.
- The complexity of performing the backtrace to find the optimal subset is $\Theta(n + W)$.

NOTE : The running time of this algorithm is not a polynomial function of n ; rather it is a polynomial function of n and W , the largest integer involved in defining the problem. Such algorithms are known as *pseudo-polynomial*. They are efficient when the values $\{w_i\}$ are small, but less practical as these values grow large.

DP Knapsack Problem - Top-Down

- Divide and conquer type of (top down) approach of solving knapsack generally recompute many previously computed sub-problems, hence inefficient.

DP Knapsack Problem - Top-Down

- Divide and conquer type of (top down) approach of solving knapsack generally recompute many previously computed sub-problems, hence inefficient.
- Bottom up dynamic programming approach avoids recomputation, but can compute many unnecessary solutions to sub-problems.

DP Knapsack Problem - Top-Down

- Divide and conquer type of (top down) approach of solving knapsack generally recompute many previously computed sub-problems, hence inefficient.
- Bottom up dynamic programming approach avoids recomputation, but can compute many unnecessary solutions to sub-problems.
- Combine space saving of divide and conquer and speed up of bottom up approaches?

DP Knapsack Problem - Top-Down

ALGORITHM **MFKnapsack** (i, j)

/* Implement the memory function method (top-down) for the knapsack problem. */
/* INPUT : A non-negative integer i indicating the number of the first items being considered and
a non-negative integer j indicating the knapsack capacity. */
/* OUTPUT : The value of an optimal, feasible subset of the first i items. */
/* NOTE: Requires global arrays $w[1 \dots n]$ and $v[1 \dots n]$ of weights and values of n items, and
table $F[0 \dots n, 0 \dots W]$ initialized with -1 s, except for row 0 and column 0 being all 0s. */

- 1: **if** $F[i, j] < 0$ **then**
- 2: **if** $j < w[i]$ **then**
- 3: $x = \text{MFKnapsack}(i - 1, j)$
- 4: **else**
- 5: $x = \max(\text{MFKnapsack}(i - 1, j), v[i] + \text{MFKnapsack}(i - 1, j - w[i]))$
- 6: **end if**
- 7: $F[i, j] = x$
- 8: **end if**
- 9: **return** $F[i, j]$

DP Knapsack Problem - Top-Down

```
if  $F[i, j] < 0$  then
    if  $j < w[i]$  then
         $x = \text{MKnapsack}(i - 1, j)$ 
    else
         $x = \max(\text{MKnapsack}(i - 1, j), v[i] + \text{MKnapsack}(i - 1, j - w[i]))$ 
    end if
     $F[i, j] = x$ 
end if
```

$\downarrow i$	$W \rightarrow$	0	1	2	3	4	5	6
	0	0	0	0	0	0	0	0
$w_1 = 3, v_1 = 25$	1	0	-1	-1	-1	-1	-1	-1
$w_2 = 2, v_2 = 20$	2	0	-1	-1	-1	-1	-1	-1
$w_3 = 1, v_3 = 15$	3	0	-1	-1	-1	-1	-1	-1
$w_4 = 4, v_4 = 40$	4	0	-1	-1	-1	-1	-1	-1
$w_5 = 5, v_5 = 50$	5	0	-1	-1	-1	-1	-1	-1

DP Knapsack Problem - Top-Down

```
if  $F[i, j] < 0$  then
    if  $j < w[i]$  then
         $x = \text{MKnapsack}(i - 1, j)$ 
    else
         $x = \max(\text{MKnapsack}(i - 1, j), v[i] + \text{MKnapsack}(i - 1, j - w[i]))$ 
    end if
     $F[i, j] = x$ 
end if
```

$\downarrow i$	$W \rightarrow$	0	1	2	3	4	5	6
	0	0	0	0	0	0	0	0
$w_1 = 3, v_1 = 25$	1	0	-1	-1	-1	-1	-1	-1
$w_2 = 2, v_2 = 20$	2	0	-1	-1	-1	-1	-1	-1
$w_3 = 1, v_3 = 15$	3	0	-1	-1	-1	-1	-1	-1
$w_4 = 4, v_4 = 40$	4	0	-1	-1	-1	-1	-1	-1
$w_5 = 5, v_5 = 50$	5	0	-1	-1	-1	-1	-1	Q

DP Knapsack Problem - Top-Down

```
if  $F[i, j] < 0$  then
    if  $j < w[i]$  then
         $x = \text{MKnapsack}(i - 1, j)$ 
    else
         $x = \max(\text{MKnapsack}(i - 1, j), v[i] + \text{MKnapsack}(i - 1, j - w[i]))$ 
    end if
     $F[i, j] = x$ 
end if
```

$\downarrow i$	$W \rightarrow$	0	1	2	3	4	5	6
	0	0	0	0	0	0	0	0
$w_1 = 3, v_1 = 25$	1	0	-1	-1	-1	-1	-1	-1
$w_2 = 2, v_2 = 20$	2	0	-1	-1	-1	-1	-1	-1
$w_3 = 1, v_3 = 15$	3	0	-1	-1	-1	-1	-1	-1
$w_4 = 4, v_4 = 40$	4	0	Q	-1	-1	-1	-1	Q
$w_5 = 5, v_5 = 50$	5	0	-1	-1	-1	-1	-1	Q

DP Knapsack Problem - Top-Down

```
if  $F[i, j] < 0$  then
    if  $j < w[i]$  then
         $x = \text{MKnapsack}(i - 1, j)$ 
    else
         $x = \max(\text{MKnapsack}(i - 1, j), v[i] + \text{MKnapsack}(i - 1, j - w[i]))$ 
    end if
     $F[i, j] = x$ 
end if
```

$\downarrow i$	$W \rightarrow$	0	1	2	3	4	5	6
	0	0	0	0	0	0	0	0
$w_1 = 3, v_1 = 25$	1	0	-1	-1	-1	-1	-1	-1
$w_2 = 2, v_2 = 20$	2	0	-1	-1	-1	-1	-1	-1
$w_3 = 1, v_3 = 15$	3	0	Q	-1	-1	-1	-1	-1
$w_4 = 4, v_4 = 40$	4	0	Q	-1	-1	-1	-1	Q
$w_5 = 5, v_5 = 50$	5	0	-1	-1	-1	-1	-1	Q

DP Knapsack Problem - Top-Down

```
if  $F[i, j] < 0$  then
    if  $j < w[i]$  then
         $x = \text{MKnapsack}(i - 1, j)$ 
    else
         $x = \max(\text{MKnapsack}(i - 1, j), v[i] + \text{MKnapsack}(i - 1, j - w[i]))$ 
    end if
     $F[i, j] = x$ 
end if
```

$\downarrow i$	$W \rightarrow$	0	1	2	3	4	5	6
	0	0	0	0	0	0	0	0
$w_1 = 3, v_1 = 25$	1	0	Q	-1	-1	-1	-1	-1
$w_2 = 2, v_2 = 20$	2	0	Q	-1	-1	-1	-1	-1
$w_3 = 1, v_3 = 15$	3	0	Q	-1	-1	-1	-1	-1
$w_4 = 4, v_4 = 40$	4	0	Q	-1	-1	-1	-1	Q
$w_5 = 5, v_5 = 50$	5	0	-1	-1	-1	-1	-1	Q

DP Knapsack Problem - Top-Down

```
if  $F[i, j] < 0$  then
    if  $j < w[i]$  then
         $x = \text{MKnapsack}(i - 1, j)$ 
    else
         $x = \max(\text{MKnapsack}(i - 1, j), v[i] + \text{MKnapsack}(i - 1, j - w[i]))$ 
    end if
     $F[i, j] = x$ 
end if
```

$\downarrow i$	$W \rightarrow$	0	1	2	3	4	5	6
	0	0	0	0	0	0	0	0
$w_1 = 3, v_1 = 25$	1	0	Q	-1	-1	-1	-1	-1
$w_2 = 2, v_2 = 20$	2	0	Q	-1	-1	-1	-1	-1
$w_3 = 1, v_3 = 15$	3	0	Q	Q	-1	-1	-1	Q
$w_4 = 4, v_4 = 40$	4	0	Q	-1	-1	-1	-1	Q
$w_5 = 5, v_5 = 50$	5	0	-1	-1	-1	-1	-1	Q

DP Knapsack Problem - Top-Down

```
if  $F[i, j] < 0$  then
    if  $j < w[i]$  then
         $x = \text{MKnapsack}(i - 1, j)$ 
    else
         $x = \max(\text{MKnapsack}(i - 1, j), v[i] + \text{MKnapsack}(i - 1, j - w[i]))$ 
    end if
     $F[i, j] = x$ 
end if
```

$\downarrow i$	$W \rightarrow$	0	1	2	3	4	5	6
	0	0	0	0	0	0	0	0
$w_1 = 3, v_1 = 25$	1	0	Q	Q	Q	Q	Q	Q
$w_2 = 2, v_2 = 20$	2	0	Q	Q	-1	-1	Q	Q
$w_3 = 1, v_3 = 15$	3	0	Q	Q	-1	-1	-1	Q
$w_4 = 4, v_4 = 40$	4	0	Q	-1	-1	-1	-1	Q
$w_5 = 5, v_5 = 50$	5	0	-1	-1	-1	-1	-1	Q

DP Knapsack Problem - Top-Down

```
if  $F[i, j] < 0$  then
    if  $j < w[i]$  then
         $x = \text{MKnapsack}(i - 1, j)$ 
    else
         $x = \max(\text{MKnapsack}(i - 1, j), v[i] + \text{MKnapsack}(i - 1, j - w[i]))$ 
    end if
     $F[i, j] = x$ 
end if
```

$\downarrow i$	$W \rightarrow$	0	1	2	3	4	5	6
	0	0	0	0	0	0	0	0
$w_1 = 3, v_1 = 25$	1	0	Q	Q	Q	Q	Q	Q
$w_2 = 2, v_2 = 20$	2	0	Q	Q	-	-	Q	Q
$w_3 = 1, v_3 = 15$	3	0	Q	Q	-	-	-	Q
$w_4 = 4, v_4 = 40$	4	0	Q	-	-	-	-	Q
$w_5 = 5, v_5 = 50$	5	0	-	-	-	-	-	Q

DP Knapsack Problem - Top-Down

```
if  $F[i, j] < 0$  then
    if  $j < w[i]$  then
         $x = \text{MKnapsack}(i - 1, j)$ 
    else
         $x = \max(\text{MKnapsack}(i - 1, j), v[i] + \text{MKnapsack}(i - 1, j - w[i]))$ 
    end if
     $F[i, j] = x$ 
end if
```

$\downarrow i$	$W \rightarrow$	0	1	2	3	4	5	6
$w_1 = 3, v_1 = 25$	1	0	0	0	0	0	0	0
$w_2 = 2, v_2 = 20$	2	0	0	20	-	-	45	45
$w_3 = 1, v_3 = 15$	3	0	15	20	-	-	-	60
$w_4 = 4, v_4 = 40$	4	0	15	-	-	-	-	60
$w_5 = 5, v_5 = 50$	5	0	-	-	-	-	-	65

Top-Down vs. Bottom-Up

In general, when to use top-down or bottom-up dynamic programming?

Top-Down vs. Bottom-Up

In general, when to use top-down or bottom-up dynamic programming?

Top-down incurs additional space and time cost of maintaining recursion stack. Hence:

Top-Down vs. Bottom-Up

In general, when to use top-down or bottom-up dynamic programming?

Top-down incurs additional space and time cost of maintaining recursion stack. Hence:

- **Bottom-up:** When the final problem instance requires most or all of the sub-problem instances to be solved, e.g., edit distance.
- **Top-down:** When the final problem instance only requires a subset of the sub-problem instances to be solved, e.g., possibly knapsack.

Overview

- 1 [Overview](#)
- 2 [Edit Distance](#)
- 3 Knapsack Problem
- 4 Warshall's Algorithm
- 5 Summary

Transitive closure

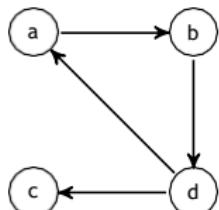
Definition

The **transitive closure** of a **directed** graph with n vertices is an $n \times n$ boolean matrix, used to determine if there is a path between any pair of vertices in the graph. Yes/True/1 if there is a path, No/False/0 if not. We want to determine for all pairs of vertices.

Transitive closure

Definition

The **transitive closure** of a **directed** graph with n vertices is an $n \times n$ boolean matrix, used to determine if there is a path between any pair of vertices in the graph. Yes/True/1 if there is a path, No/False/0 if not. We want to determine for all pairs of vertices.



$$A = \begin{matrix} & \begin{matrix} a & b & c & d \end{matrix} \\ \begin{matrix} a \\ b \\ c \\ d \end{matrix} & \begin{matrix} 0 & 1 & 0 & 0 \\ \square & 0 & 0 & 1 \\ \square & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 \end{matrix} \end{matrix}$$

(a) digraph

(b) adjacency matrix

$$T = \begin{matrix} & \begin{matrix} a & b & c & d \end{matrix} \\ \begin{matrix} a \\ b \\ c \\ d \end{matrix} & \begin{matrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \end{matrix} \end{matrix}$$

(c) transitive closure

Warshall's Algorithm

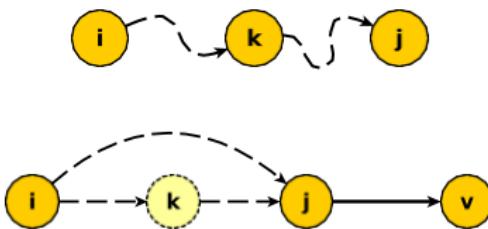
Why compute transitive closure?

- In spreadsheet software, when a cell is changed, what are the other cells whose computation directly or indirectly depend on it (i.e., might need to update also)?
- Critical communication systems: Is it possible for any nodes in this system to communicate with any other node?

Warshall's Algorithm - Sketch

Idea: Progressively use each vertex as an intermediate to join two paths.

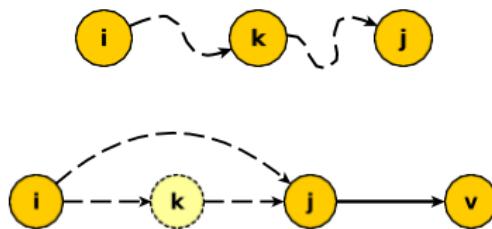
If we know there is a path between vertex i to intermediate node k , and a path from intermediate node k to j , then we know there is a path from i to j .



Warshall's Algorithm - Sketch

Idea: Progressively use each vertex as an intermediate to join two paths.

If we know there is a path between vertex i to intermediate node k , and a path from intermediate node k to j , then we know there is a path from i to j .



Warshall's algorithm progressively considers using each vertex as the intermediate, and when all the vertices have been considered as intermediates, we have the transitive closure of the graph.

Warshall's Algorithm - Sketch

Idea: Construct the transitive closure of a given digraph with n vertices through a series of n -by- n binary matrices:

$$R^{(0)}, \dots, R^{(k-1)}, R^{(k)}, \dots, R^{(n)}.$$

Warshall's Algorithm - Sketch

Idea: Construct the transitive closure of a given digraph with n vertices through a series of n -by- n binary matrices:

$$R^{(0)}, \dots, R^{(k-1)}, R^{(k)}, \dots, R^{(n)}.$$

The element $R[i, j]$ in the i -th row and the j -th column is equal to 1 if there is a directed path from the i -th vertex to the j -th vertex.

Warshall's Algorithm - Sketch

Idea: Construct the transitive closure of a given digraph with n vertices through a series of n -by- n binary matrices:

$$R^{(0)}, \dots, R^{(k-1)}, R^{(k)}, \dots, R^{(n)}.$$

The element $R[i, j]$ in the i -th row and the j -th column is equal to 1 if there is a directed path from the i -th vertex to the j -th vertex.

$R^{(0)}$ is the initial adjacency matrix and $R^{(n)}$ specifies the complete transitivity between vertices.

Warshall's Algorithm - Sketch

Idea: Construct the transitive closure of a given digraph with n vertices through a series of n -by- n binary matrices:

$$R^{(0)}, \dots, R^{(k-1)}, R^{(k)}, \dots, R^{(n)}.$$

The element $R[i, j]$ in the i -th row and the j -th column is equal to 1 if there is a directed path from the i -th vertex to the j -th vertex.

$R^{(0)}$ is the initial adjacency matrix and $R^{(n)}$ specifies the complete transitivity between vertices.

Updating from R^{k-1} to R^k :

- If a path exists between i and j vertices, then it remains there, i.e., if $R[i, j]^{(k-1)} = 1$, it remains 1 ($R[i, j]^{(k)} = 1$).

Warshall's Algorithm - Sketch

Idea: Construct the transitive closure of a given digraph with n vertices through a series of n -by- n binary matrices:

$$R^{(0)}, \dots, R^{(k-1)}, R^{(k)}, \dots, R^{(n)}.$$

The element $R[i, j]$ in the i -th row and the j -th column is equal to 1 if there is a directed path from the i -th vertex to the j -th vertex.

$R^{(0)}$ is the initial adjacency matrix and $R^{(n)}$ specifies the complete transitivity between vertices.

Updating from R^{k-1} to R^k :

- If a path exists between i and j vertices, then it remains there, i.e., if $R[i, j]^{(k-1)} = 1$, it remains 1 ($R[i, j]^{(k)} = 1$).
- If we can use vertex k as intermediate vertex to traverse from i to j vertices and we previously didn't know there was such a path, then update our transitivity information, i.e.,

Warshall's Algorithm - Sketch

Idea: Construct the transitive closure of a given digraph with n vertices through a series of n -by- n binary matrices:

$$R^{(0)}, \dots, R^{(k-1)}, R^{(k)}, \dots, R^{(n)}.$$

The element $R[i, j]$ in the i -th row and the j -th column is equal to 1 if there is a directed path from the i -th vertex to the j -th vertex.

$R^{(0)}$ is the initial adjacency matrix and $R^{(n)}$ specifies the complete transitivity between vertices.

Updating from R^{k-1} to R^k :

- If a path exists between i and j vertices, then it remains there, i.e., if $R[i, j]^{(k-1)} = 1$, it remains 1 ($R[i, j]^{(k)} = 1$).
- If we can use vertex k as intermediate vertex to traverse from i to j vertices and we previously didn't know there was such a path, then update our transitivity information, i.e.,
 - if $R[i, j]^{(k-1)} = 0$, and $R[i, k]^{(k-1)} = 1$ and $R[k, j]^{(k-1)} = 1$, then change $R[i, j]^{(k)}$ to 1.

Warshall's Algorithm - Example

$$R^{(0)} = \begin{matrix} & \square & 1 & 2 & 3 & 4 & \square \\ 1 & \square & 0 & 1 & 0 & 0 & \square \\ & \square & 0 & 0 & 0 & 1 & \square \\ 2 & \square & 0 & 0 & 0 & 0 & \square \\ 3 & \square & 0 & 0 & 0 & 0 & \square \\ 4 & 1 & 0 & 1 & 0 & & \end{matrix}$$

Applying Warshall's algorithm to calculate the transitive closure of a digraph.

Warshall's Algorithm - Example

$$R^{(1)} = \begin{matrix} & \square & 1 & 2 & 3 & 4 & \square \\ 1 & \square & 0 & 1 & 0 & 0 & \square \\ & 0 & \square & 0 & 0 & 1 & \square \\ 2 & 0 & \square & 0 & 0 & 0 & \square \\ 3 & 0 & 0 & \square & 0 & 0 & \square \\ 4 & 1 & 0 & 1 & 0 & \square & \end{matrix}$$

Applying Warshall's algorithm to calculate the transitive closure of a digraph.

Warshall's Algorithm - Example

$$R^{(1)} = \begin{matrix} & \square & 1 & 2 & 3 & 4 & \square \\ 1 & \square & 0 & 1 & 0 & 0 & \square \\ & 0 & \square & 0 & 0 & 1 & \square \\ 2 & \square & 0 & 0 & 0 & 0 & \square \\ 3 & \square & 0 & 0 & 0 & 0 & \square \\ 4 & 1 & 0 & 1 & 0 & & \end{matrix}$$

Applying Warshall's algorithm to calculate the transitive closure of a digraph.

Warshall's Algorithm - Example

$$R^{(1)} = \begin{matrix} & \square & 1 & 2 & 3 & 4 & \square \\ 1 & \square & 0 & 1 & 0 & 0 & \square \\ & 0 & \square & 0 & 0 & 1 & \square \\ 2 & \square & 0 & 0 & 0 & 0 & \square \\ 3 & \square & 0 & 0 & 0 & 0 & \square \\ 4 & 1 & 1 & 1 & 0 & & \end{matrix}$$

Applying Warshall's algorithm to calculate the transitive closure of a digraph.

Warshall's Algorithm - Example

$$R^{(2)} = \begin{matrix} & \square & 1 & 2 & 3 & 4 & \square \\ 1 & 0 & \boxed{1} & 0 & 0 & 0 & \\ & \boxed{0} & 0 & 0 & 1 & \boxed{0} \\ 3 & 0 & \boxed{0} & 0 & 0 & 0 & \\ 4 & 1 & 1 & 1 & 0 & & \end{matrix}$$

Applying Warshall's algorithm to calculate the transitive closure of a digraph.

Warshall's Algorithm - Example

$$R^{(2)} = \begin{matrix} & \square & 1 & 2 & 3 & 4 & \square \\ 1 & 0 & \textcolor{red}{1} & 0 & 0 & 0 & \\ & \square & 0 & 0 & 0 & 1 & \square \\ 3 & 0 & \textcolor{blue}{0} & 0 & 0 & 0 & \\ & 1 & 1 & 1 & 1 & 0 & \end{matrix}$$

Applying Warshall's algorithm to calculate the transitive closure of a digraph.

Warshall's Algorithm - Example

$$R^{(2)} = \begin{matrix} & \square & 1 & 2 & 3 & 4 & \square \\ 1 & 0 & \textcolor{red}{1} & 0 & 1 & \square \\ & \square & 0 & 0 & 0 & \textcolor{red}{1} & \square \\ 3 & 0 & \textcolor{blue}{0} & 0 & 0 & 0 & \square \\ & 1 & 1 & 1 & 1 & 0 & \end{matrix}$$

Applying Warshall's algorithm to calculate the transitive closure of a digraph.

Warshall's Algorithm - Example

$$R^{(2)} = \begin{matrix} & \square & 1 & 2 & 3 & 4 & \square \\ 1 & 0 & \boxed{1} & 0 & 1 & & \\ R^{(2)} = & 2 & \boxed{0} & 0 & 0 & \boxed{1} & \square \\ 3 & 0 & \boxed{0} & 0 & 0 & 0 & \square \\ 4 & 1 & \boxed{1} & 1 & 1 & 0 & \end{matrix}$$

Applying Warshall's algorithm to calculate the transitive closure of a digraph.

Warshall's Algorithm - Example

$$R^{(2)} = \begin{matrix} & \square & 1 & 2 & 3 & 4 & \square \\ 1 & 0 & \boxed{1} & 0 & 1 & & \\ R^{(2)} = & 2 & \boxed{0} & 0 & 0 & \boxed{1} & \square \\ 3 & 0 & \boxed{0} & 0 & 0 & 0 & \square \\ 4 & 1 & 1 & 1 & 1 & 1 & \end{matrix}$$

Applying Warshall's algorithm to calculate the transitive closure of a digraph.

Warshall's Algorithm - Example

$$R^{(3)} = \begin{matrix} & \square & 1 & 2 & 3 & 4 & \square \\ 1 & \square & 0 & 1 & 0 & 1 & \square \\ & 2 & \square & 0 & 0 & 0 & 1 & \square \\ & 3 & \square & 0 & 0 & 0 & 0 & \square \\ 4 & 1 & 1 & 1 & 1 & 1 & \end{matrix}$$

Applying Warshall's algorithm to calculate the transitive closure of a digraph.

Warshall's Algorithm - Example

$$R^{(4)} = \begin{matrix} & \square & 1 & 2 & 3 & \textcolor{blue}{4} & \square \\ 1 & \square & 0 & 1 & 0 & \textcolor{blue}{1} & \square \\ & \square & 0 & 0 & 0 & \textcolor{blue}{1} & \square \\ 2 & \square & 0 & 0 & 0 & \textcolor{blue}{1} & \square \\ 3 & \square & 0 & 0 & 0 & \textcolor{blue}{0} & \square \\ 4 & \textcolor{blue}{1} & \textcolor{blue}{1} & \textcolor{blue}{1} & \textcolor{blue}{1} & \textcolor{blue}{1} & \end{matrix}$$

Applying Warshall's algorithm to calculate the transitive closure of a digraph.

Warshall's Algorithm - Example

$$R^{(4)} = \begin{matrix} & \square & 1 & 2 & 3 & 4 & \square \\ 1 & \square & 0 & 1 & 0 & 1 & \square \\ & 2 & \square & 0 & 0 & 0 & 1 & \square \\ & 3 & \square & 0 & 0 & 0 & 0 & \square \\ 4 & 1 & 1 & 1 & 1 & 1 & \end{matrix}$$

Applying Warshall's algorithm to calculate the transitive closure of a digraph.

Warshall's Algorithm - Example

$$R^{(4)} = \begin{matrix} & \square & 1 & 2 & 3 & 4 & \square \\ 1 & \square & \textcolor{red}{1} & 1 & 0 & \textcolor{red}{1} & \square \\ & 2 & \square & 0 & 0 & 0 & \textcolor{blue}{1} & \square \\ & 3 & \square & 0 & 0 & 0 & \textcolor{blue}{0} & \square \\ 4 & \textcolor{blue}{1} & \textcolor{red}{1} & \textcolor{blue}{1} & 1 & 1 & \end{matrix}$$

Applying Warshall's algorithm to calculate the transitive closure of a digraph.

Warshall's Algorithm - Example

$$R^{(4)} = \begin{matrix} & \square & 1 & 2 & 3 & 4 & \square \\ 1 & \square & 1 & 1 & 0 & 1 & \square \\ 2 & \square & 0 & 0 & 0 & 1 & \square \\ 3 & \square & 0 & 0 & 0 & 0 & \square \\ 4 & 1 & 1 & 1 & 1 & 1 & \end{matrix}$$

Applying Warshall's algorithm to calculate the transitive closure of a digraph.

Warshall's Algorithm - Example

$$R^{(4)} = \begin{matrix} & \square & 1 & 2 & 3 & \textcolor{blue}{4} & \square \\ 1 & \square & 1 & 1 & \textcolor{red}{1} & \textcolor{red}{1} & \square \\ & 2 & \square & 0 & 0 & 0 & \textcolor{blue}{1} & \square \\ & 3 & \square & 0 & 0 & 0 & \textcolor{blue}{0} & \square \\ 4 & \textcolor{blue}{1} & \textcolor{blue}{1} & \textcolor{blue}{1} & \textcolor{red}{1} & \textcolor{blue}{1} & \end{matrix}$$

Applying Warshall's algorithm to calculate the transitive closure of a digraph.

Warshall's Algorithm - Example

$$R^{(4)} = \begin{matrix} & \square & 1 & 2 & 3 & 4 & \square \\ 1 & \square & 1 & 1 & 1 & 1 & \square \\ 2 & \square & 0 & 0 & 0 & 1 & \square \\ 3 & \square & 0 & 0 & 0 & 0 & \square \\ 4 & 1 & 1 & 1 & 1 & 1 & \end{matrix}$$

Applying Warshall's algorithm to calculate the transitive closure of a digraph.

Warshall's Algorithm - Example

$$R^{(4)} = \begin{matrix} & \square & 1 & 2 & 3 & 4 & \square \\ 1 & \square & 1 & 1 & 1 & 1 & \square \\ 2 & \square & \textcolor{red}{1} & 0 & 0 & \textcolor{red}{1} & \square \\ 3 & \square & 0 & 0 & 0 & 0 & \square \\ 4 & \textcolor{blue}{1} & \textcolor{red}{1} & \textcolor{blue}{1} & \textcolor{blue}{1} & \textcolor{blue}{1} & \end{matrix}$$

Applying Warshall's algorithm to calculate the transitive closure of a digraph.

Warshall's Algorithm - Example

$$R^{(4)} = \begin{matrix} & \square & 1 & 2 & 3 & 4 & \square \\ 1 & \square & 1 & 1 & 1 & 1 & \square \\ 2 & \square & 1 & 0 & 0 & 1 & \square \\ 3 & \square & 0 & 0 & 0 & 0 & \square \\ 4 & 1 & 1 & 1 & 1 & 1 & \end{matrix}$$

Applying Warshall's algorithm to calculate the transitive closure of a digraph.

Warshall's Algorithm - Example

$$R^{(4)} = \begin{matrix} & \square & 1 & 2 & 3 & 4 & \square \\ 1 & \square & 1 & 1 & 1 & 1 & \square \\ 2 & \square & 1 & \textcolor{red}{1} & 0 & \textcolor{red}{1} & \square \\ 3 & \square & 0 & 0 & 0 & 0 & \square \\ 4 & \textcolor{blue}{1} & \textcolor{blue}{1} & \textcolor{red}{1} & \textcolor{blue}{1} & \textcolor{blue}{1} & \end{matrix}$$

Applying Warshall's algorithm to calculate the transitive closure of a digraph.

Warshall's Algorithm - Example

$$R^{(4)} = \begin{matrix} & \square & 1 & 2 & 3 & 4 & \square \\ 1 & \square & 1 & 1 & 1 & 1 & \square \\ 2 & \square & 1 & 1 & 0 & 1 & \square \\ 3 & \square & 0 & 0 & 0 & 0 & \square \\ 4 & 1 & 1 & 1 & 1 & 1 & \end{matrix}$$

Applying Warshall's algorithm to calculate the transitive closure of a digraph.

Warshall's Algorithm - Example

$$R^{(4)} = \begin{matrix} & \square & 1 & 2 & 3 & \textcolor{blue}{4} & \square \\ 1 & \square & 1 & 1 & 1 & \textcolor{blue}{1} & \square \\ & 2 & \square & 1 & 1 & \textcolor{red}{1} & \textcolor{red}{1} & \square \\ & 3 & \square & 0 & 0 & 0 & \textcolor{blue}{0} & \square \\ & 4 & \textcolor{blue}{1} & \textcolor{blue}{1} & \textcolor{blue}{1} & \textcolor{red}{1} & \textcolor{blue}{1} & \end{matrix}$$

Applying Warshall's algorithm to calculate the transitive closure of a digraph.

Warshall's Algorithm - Example

$$T = R^{(4)} = \begin{matrix} & & 1 & 2 & 3 & 4 & \\ & 1 & \square & 1 & 1 & 1 & 1 & \square \\ 2 & R & 1 & 1 & 1 & 1 & \square & \\ 3 & 0 & \square & 0 & 0 & 0 & 0 & \square \\ 4 & 1 & 1 & 1 & 1 & 1 & & \end{matrix}$$

Applying Warshall's algorithm to calculate the transitive closure of a digraph.

Warshall's Algorithm - Pseudocode

```
ALGORITHM Warshall (A[1...n, 1...n])
/* Compute the transitive closure of a graph using Warshall's algorithm.
*/
/* INPUT : The adjacency matrix A of a digraph with n vertices. */
/* OUTPUT : The transitive closure of the digraph. */

1:  $R^{(0)} = A$ 
2: for  $k = 1$  to  $n$  do
3:   for  $i = 1$  to  $n$  do
4:     for  $j = 1$  to  $n$  do
5:        $R^{(k)}[i, j] = R^{(k-1)}[i, j]$  or ( $R^{(k-1)}[i, k]$  and ( $R^{(k-1)}[k, j]$ ))
6:     end for
7:   end for
8: end for
9: return  $R^{(n)}$ 
```

Warshall's Algorithm

- The time efficiency of Warshall's algorithm is $\Theta(n^3)$ and uses $\Theta(n^2)$ space.
- For sparse graphs, the transitive closure can be calculated more efficiently using an adjacency list representation with a depth-first or breadth-first traversal.
- Using a DFS or BFS traversal with n vertices and m edges takes $\Theta(n + m)$ time. Doing it n times takes $\Theta(n^2 + nm)$ time.
- If the graph is sparse, i.e. $m \approx n$, the time efficiency is $\Theta(n^2)$.

Overview

1 Overview

2 Edit Distance

3 Knapsack Problem

4 Warshall's Algorithm

5 Summary

Summary

- Described dynamic programming and how it is useful to solve problems that require solving sub-problems multiple times.
- Examples:
 - Coin-row problem
 - Computing the edit distance
 - Knapsack - Bottom up and top down
 - Transitive closure - Warshall's algorithm

Levitin - The design and analysis of algorithms

This week we will be covering the material from Chapter 9.

Learning outcomes:

- Understand and be able to apply the greedy approach to solve interesting problems.
- Examples:
 - spanning tree - Prim's algorithm
 - spanning tree - Kruskal's algorithm
 - single source shortest-path - Dijkstra's algorithm
 - data compression

Outline

- ① Overview
- ② Prim's Algorithm
- ③ Kruskal's Algorithm
- ④ Dijkstra's Algorithm
- ⑤ Data Compression
- ⑥ Summary

Overview

- ① [Overview](#)
- ② [Prim's Algorithm](#)
- ③ [Kruskal's Algorithm](#)
- ④ [Dijkstra's Algorithm](#)
- ⑤ [Data Compression](#)
- ⑥ Summary

Greedy Algorithms

Greedy Algorithms build up a solution piece by piece, always choosing the next piece that offers the most immediate and obvious benefit.

Sometimes such an approach can lead to an inferior solution, but in other cases it can lead to a simple and optimal solution.

Overview

1 [Overview](#)

2 [Prim's Algorithm](#)

3 [Kruskal's Algorithm](#)

4 [Dijkstra's Algorithm](#)

5 [Data Compression](#)

6 Summary

Minimum Spanning Tree

Spanning Tree Problem

A **spanning tree** of a connected graph is a connected acyclic subgraph (i.e. tree) which contains all the vertices of the graph and a subset of edges from the original graph.

Minimum Spanning Tree

Spanning Tree Problem

A **spanning tree** of a connected graph is a connected acyclic subgraph (i.e. tree) which contains all the vertices of the graph and a subset of edges from the original graph.

Minimum Spanning Tree Problem

A **minimum spanning tree** of a **weighted** connected graph is the spanning tree of the smallest total weight (sum of the weights on all of the tree's edges).

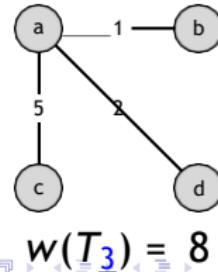
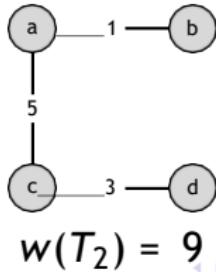
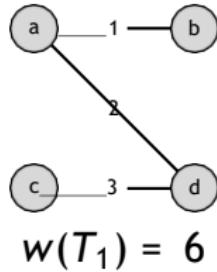
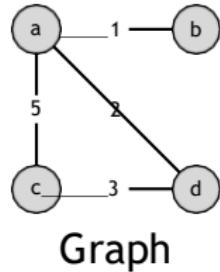
Minimum Spanning Tree

Spanning Tree Problem

A **spanning tree** of a connected graph is a connected acyclic subgraph (i.e. tree) which contains all the vertices of the graph and a subset of edges from the original graph.

Minimum Spanning Tree Problem

A **minimum spanning tree** of a **weighted** connected graph is the spanning tree of the smallest total weight (sum of the weights on all of the tree's edges).



Applications of Minimum Spanning Tree

- Designing networks (phones, computers etc): Want to connect up a series of offices with telephone or wired lines, but want to minimise cost.
- Approximate solutions to hard problems: travelling salesman
- Generation of perfect mazes

Prim's Algorithm - Sketch

Prim's Algorithm is one approach to find minimum spanning tree.

Prim's Algorithm - Sketch

Prim's Algorithm is one approach to find minimum spanning tree.

Idea: Select one vertex at a time and add to tree.

- ① Start with one arbitrary selected vertex and add this to tree.

Prim's Algorithm - Sketch

Prim's Algorithm is one approach to find minimum spanning tree.

Idea: Select one vertex at a time and add to tree.

- ① Start with one arbitrary selected vertex and add this to tree.
- ② Then at each iteration add a **neighbouring** vertex to the tree that has **minimum edge weight** to one of the vertices in the current tree. It must not be in the tree.

Prim's Algorithm - Sketch

Prim's Algorithm is one approach to find minimum spanning tree.

Idea: Select one vertex at a time and add to tree.

- ① Start with one arbitrary selected vertex and add this to tree.
- ② Then at each iteration add a **neighbouring** vertex to the tree that has **minimum edge weight** to one of the vertices in the current tree. It must not be in the tree.
- ③ When all vertices added to tree, we are done.

Prim's Algorithm - Sketch

Prim's Algorithm is one approach to find minimum spanning tree.

Idea: Select one vertex at a time and add to tree.

- ① Start with one arbitrary selected vertex and add this to tree.
- ② Then at each iteration add a **neighbouring** vertex to the tree that has **minimum edge weight** to one of the vertices in the current tree. It must not be in the tree.

When all vertices added to tree, we are done.

Note:

- Use a **min priority queue** to quickly find this neighbouring vertex with minimum edge weight

Prim's Algorithm - Sketch

Prim's Algorithm is one approach to find minimum spanning tree.

Idea: Select one vertex at a time and add to tree.

- ① Start with one arbitrary selected vertex and add this to tree.
- ② Then at each iteration add a **neighbouring** vertex to the tree that has **minimum edge weight** to one of the vertices in the current tree. It must not be in the tree.

When all vertices added to tree, we are done.

Note:

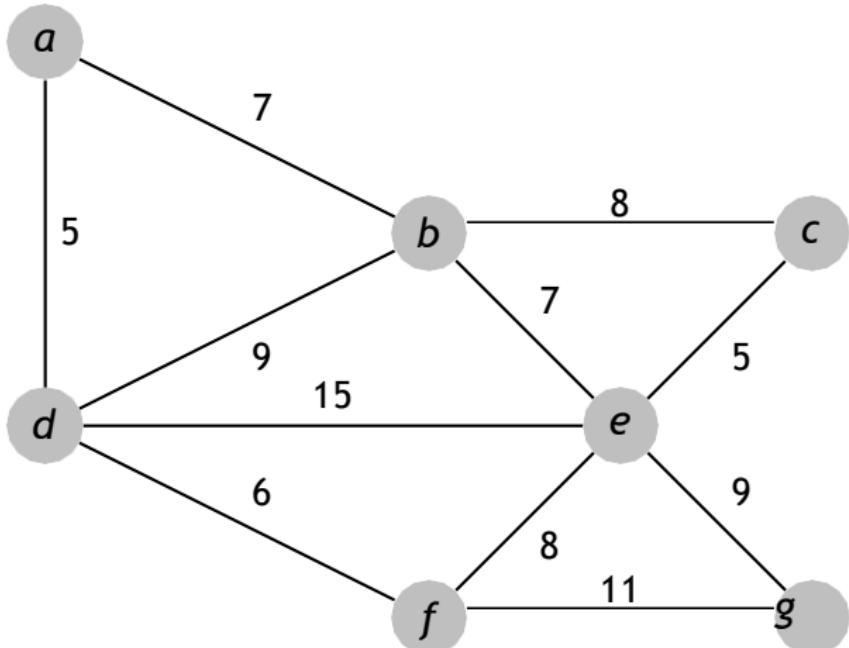
- Use a **min priority queue** to quickly find this neighbouring vertex with minimum edge weight
- When adding, we may need to update the smallest edge weight to a vertex in neighbour set as there may be a smallest edge weight from updated tree to new neighbour set.

Prim's Algorithm - Pseudocode

ALGORITHM **Prim** (G)

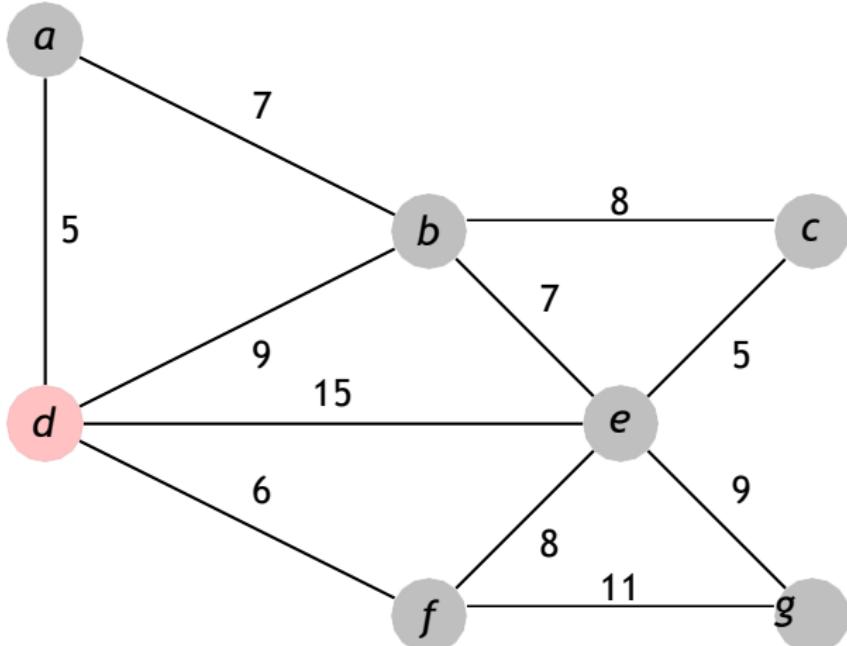
```
/* Prim's algorithm for constructing a minimum spanning tree. */
/* INPUT : A weighted connected graph  $G = \langle V, E \rangle$  */
/* OUTPUT : The set of edges  $E_T$  composing a minimum spanning tree.
*/
1:  $V_T = \{v_0\}$                                 d Initialized with arbitrary vertex  $v_0$ .
2:  $E_T = \emptyset$ 
3: for  $i = 1$  to  $|V| - 1$  do
4:   find a minimum-weight edge  $e^* = (v^*, u^*)$  among all
5:   the edges  $(v, u)$  such that  $v$  is in  $V_T$  and  $u$  is in  $V - V_T$ .
6:    $V_T = V_T \cup \{u^*\}$ 
7:    $E_T = E_T \cup \{e^*\}$ 
8: end for
9: return  $E_T$ 
```

Prim's Algorithm - Example



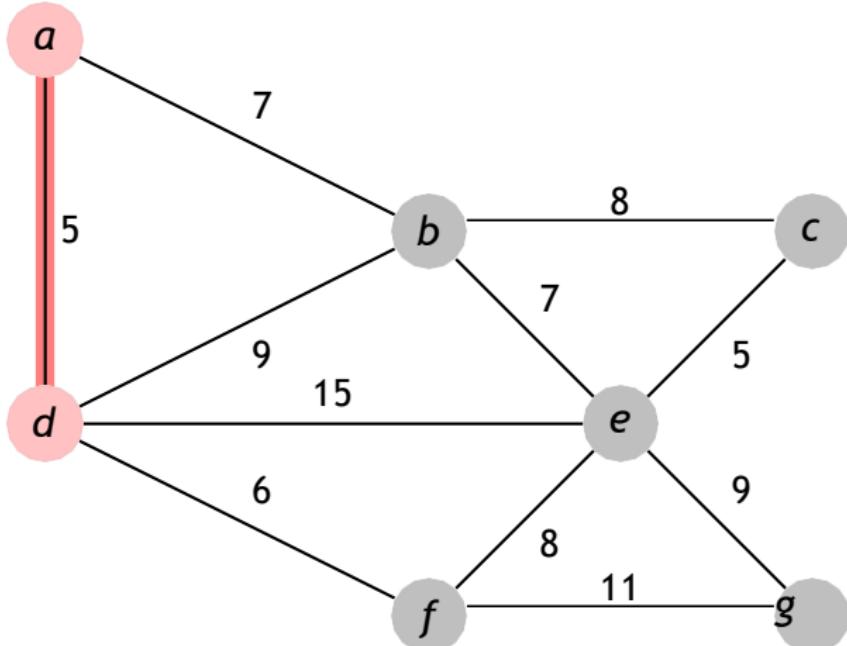
$$V_T = \{\}, PQ = \{\}$$

Prim's Algorithm - Example



$$V_T = \{d\}, PQ = \{(a, 5), (f, 6), (b, 9), (e, 15)\}$$

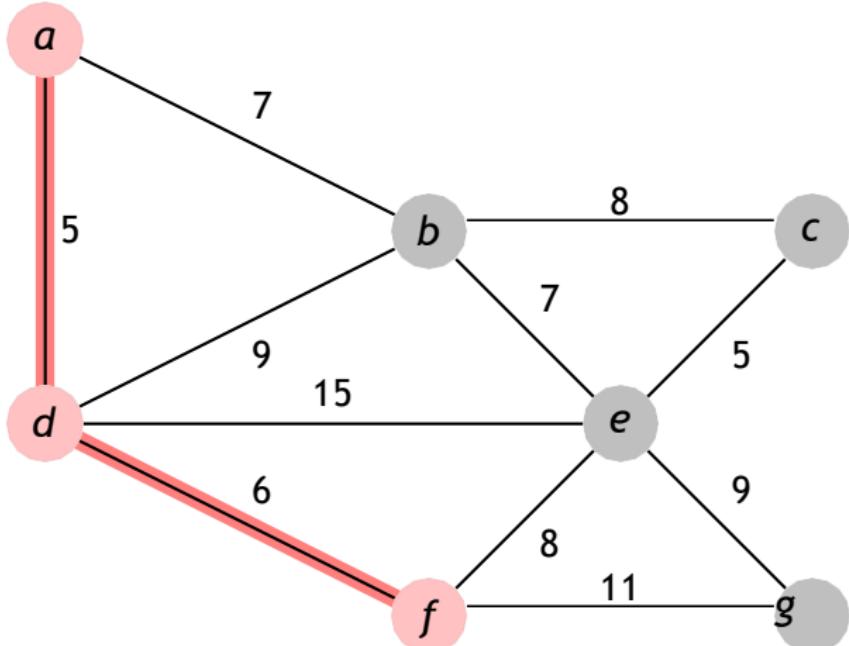
Prim's Algorithm - Example



$$V_T = \{d, \textcolor{red}{a}\}, PQ = \{(f, 6), (\textcolor{red}{b}, 7), (b, 9), (e, 15)\}$$

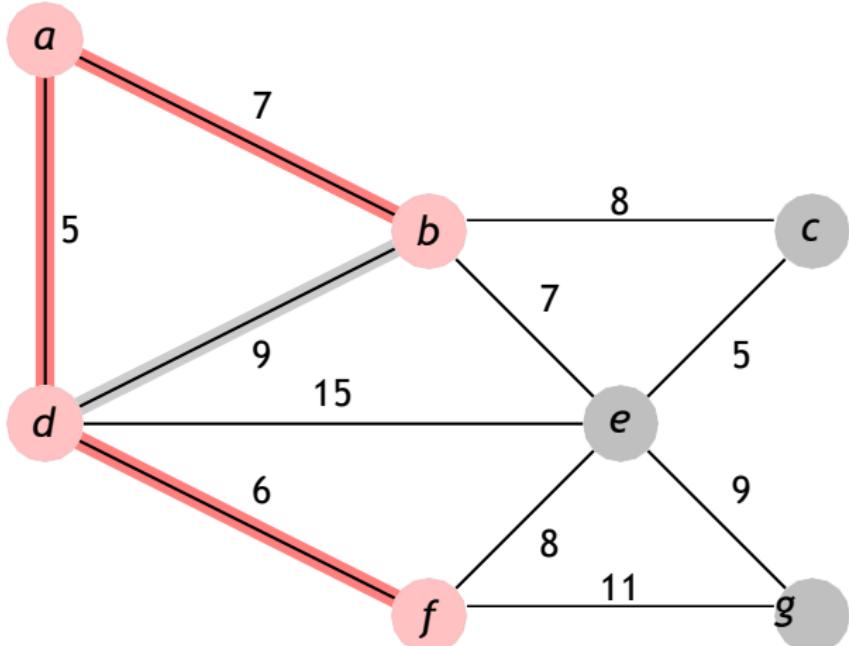
=

Prim's Algorithm - Example



$V_T = \{d, a, f\}$, $PQ = \{(b, 7), (e, 8), (g, 11)\}$

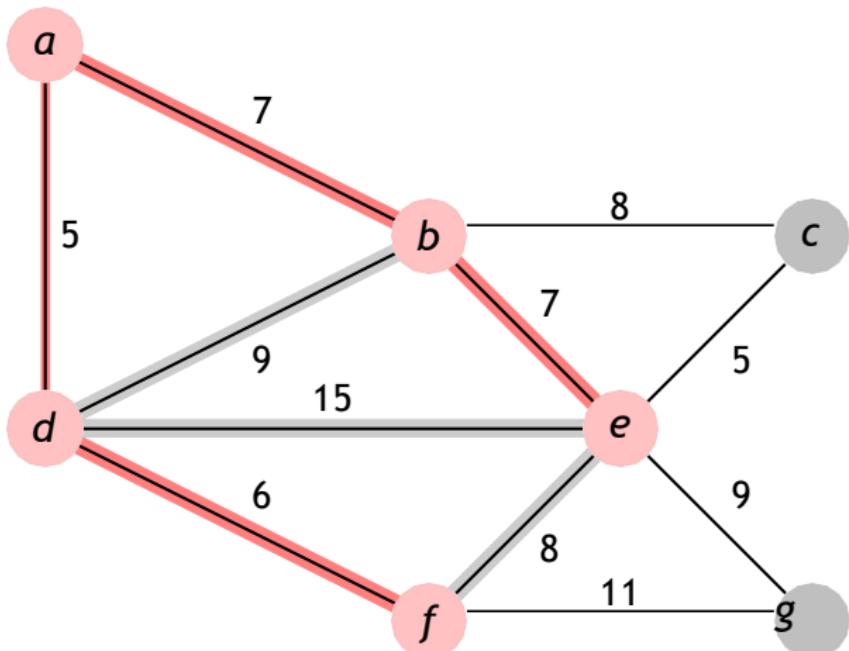
Prim's Algorithm - Example



$$V_T = \{d, a, f, \textcolor{red}{b}\}, PQ = \textcolor{red}{(e, 7)}, (e, 8), (c, 8), (g, 11)\}$$

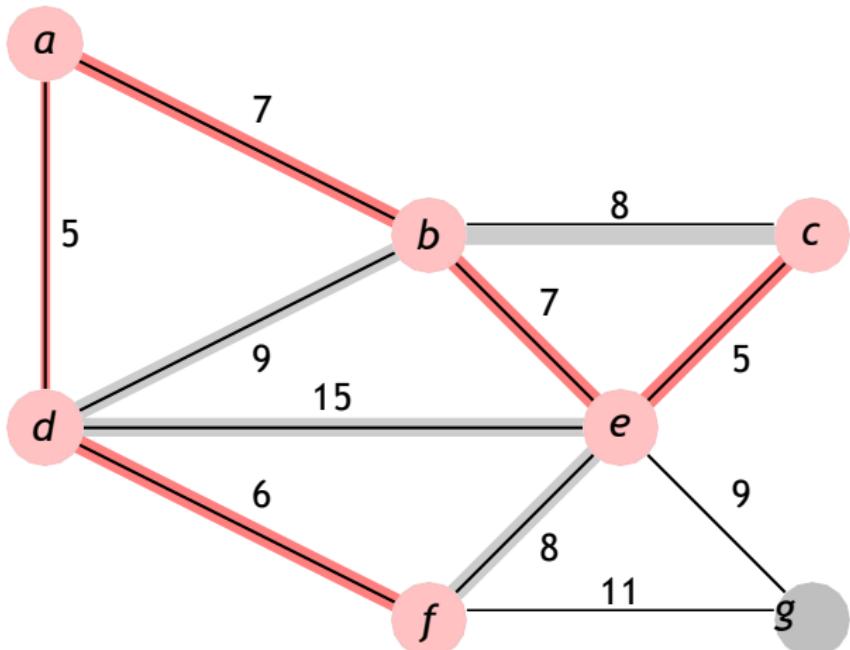
{

Prim's Algorithm - Example



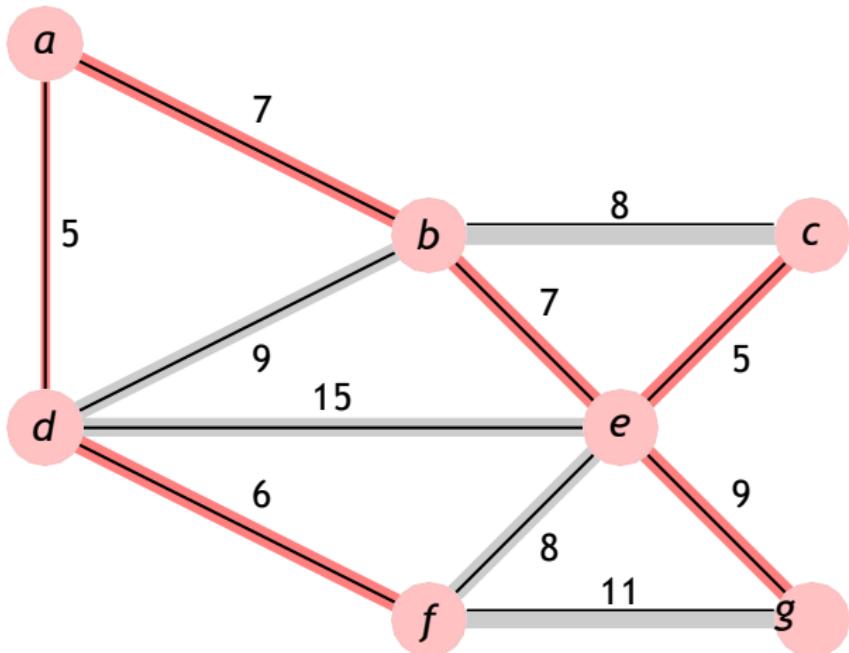
$$V_T = \{d, a, f, b, \textcolor{red}{e}\}, PQ = \{(\textcolor{red}{c}, 5), (c, 8), (\textcolor{red}{g}, 9), (g, 11)\}$$

Prim's Algorithm - Example



$$V_T = \{d, a, f, b, e, \textcolor{red}{c}\}, PQ = \{(g, 9)\}$$

Prim's Algorithm - Example



$$V_T = \{d, a, f, b, e, c, \textcolor{red}{g}\}, PQ = \{\}$$

Minecraft Maze Generation

https://www.youtube.com/watch?v=5mn0C1C0_9o

Prim's Algorithm - Summary

- The algorithm always gives the optimal solution, regardless of where you start. See Levitin Chp 9.1 for the proof.

Prim's Algorithm - Summary

- The algorithm always gives the optimal solution, regardless of where you start. See Levitin Chp 9.1 for the proof.
- The efficiency of the algorithm using a min-heap and an adjacency list is $O(|E| \log |V|)$.

Overview

- 1 [Overview](#)
- 2 [Prim's Algorithm](#)
- 3 [Kruskal's Algorithm](#)
- 4 [Dijkstra's Algorithm](#)
- 5 [Data Compression](#)
- 6 Summary

Kruskal's Algorithm - Sketch

Kruskal's algorithm is an alternative method to find minimum spanning trees. (We will explore why have two algorithms at end of this section)

Kruskal's Algorithm - Sketch

Kruskal's algorithm is an alternative method to find minimum spanning trees. (We will explore why have two algorithms at end of this section)

Idea: Select one edge at a time and add to tree.

- 1 Start with empty tree.

Kruskal's Algorithm - Sketch

Kruskal's algorithm is an alternative method to find minimum spanning trees. (We will explore why have two algorithms at end of this section)

Idea: Select one edge at a time and add to tree.

- ① Start with empty tree.
- ② Add one edge at a time to the current tree. Edge selected is the one with **minimum weight** and **doesn't create a cycle**.

Kruskal's Algorithm - Sketch

Kruskal's algorithm is an alternative method to find minimum spanning trees. (We will explore why have two algorithms at end of this section)

Idea: Select one edge at a time and add to tree.

- ① Start with empty tree.
- ② Add one edge at a time to the current tree. Edge selected is the one with **minimum weight** and **doesn't create a cycle**.
- ③ Terminate when $|V| - 1$ such edges have been added. (A tree of $|V|$ nodes has $|V| - 1$ edges).

Kruskal's Algorithm - Sketch

Kruskal's algorithm is an alternative method to find minimum spanning trees. (We will explore why have two algorithms at end of this section)

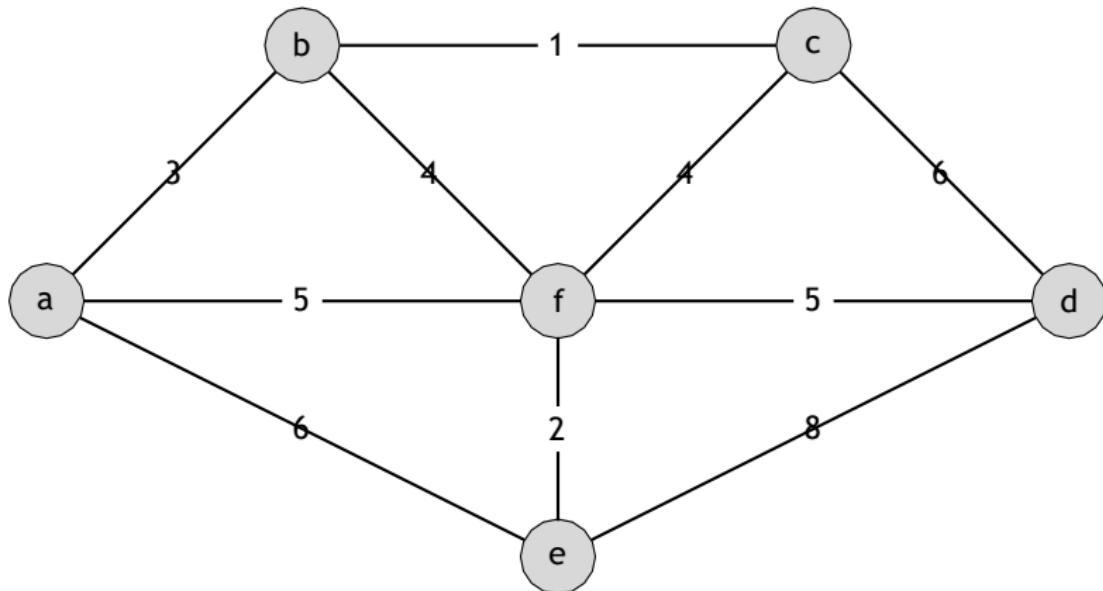
Idea: Select one edge at a time and add to tree.

- ① Start with empty tree.
- ② Add one edge at a time to the current tree. Edge selected is the one with **minimum weight** and **doesn't create a cycle**.
- ③ Terminate when $|V| - 1$ such edges have been added. (A tree of $|V|$ nodes has $|V| - 1$ edges).

Note:

- To be fast in edge selection, we initially sort all edges from smallest to largest by weight.
- Note that the nodes are not always connected in the intermediate stages of the algorithm.

Kruskal's Algorithm - Example



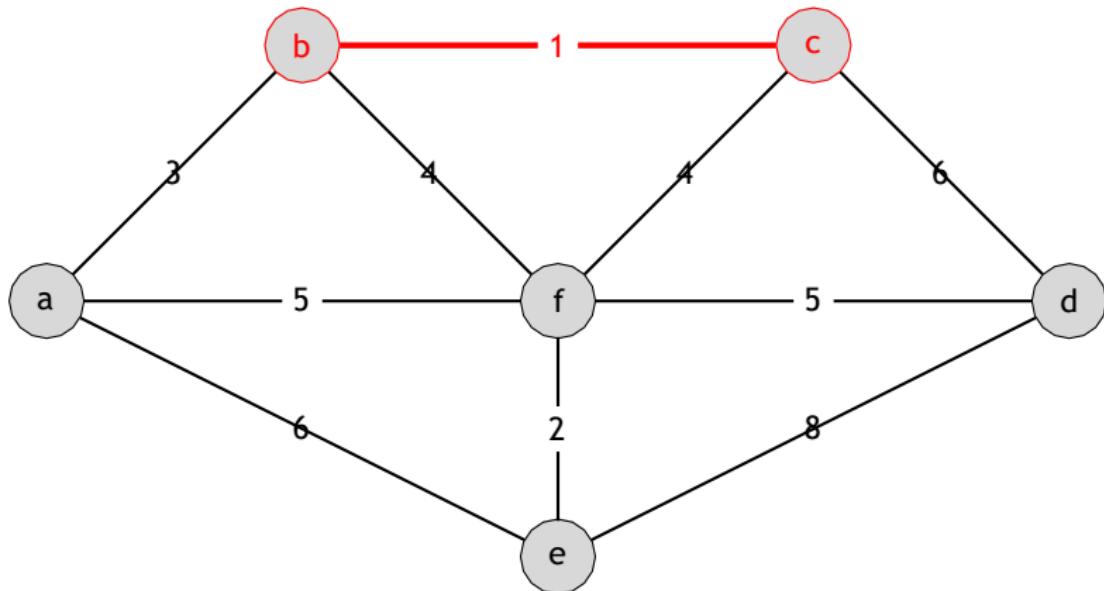
All Edges : bc ef ab bf cf af df ae cd de

 1 2 3 4 4 5 5 6 6 8

Tree Edges : \emptyset

\emptyset

Kruskal's Algorithm - Example



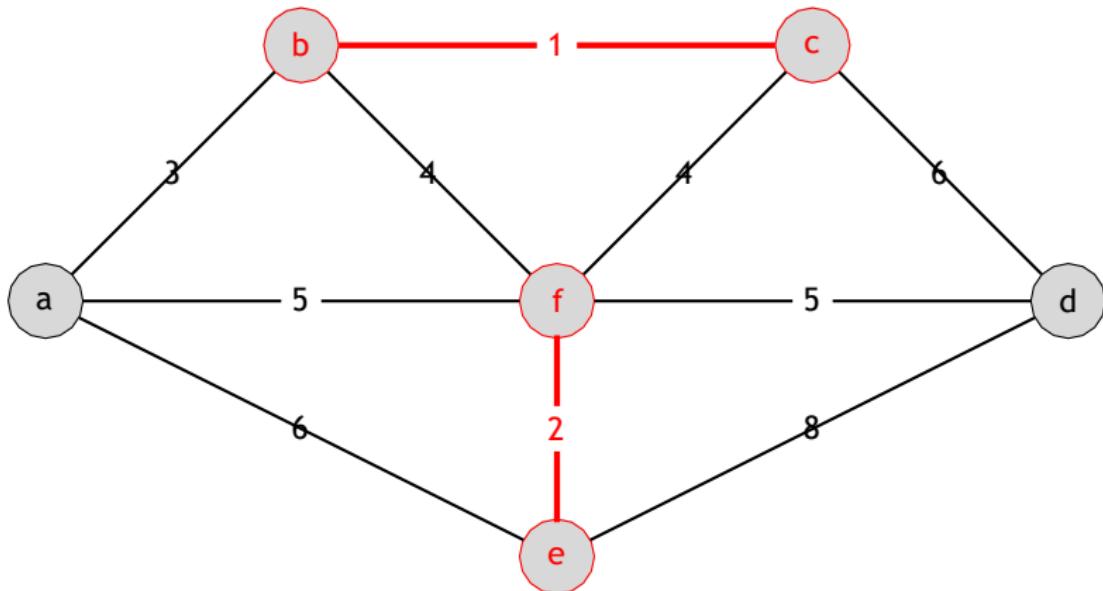
All Edges : bc ef ab bf cf af df ae cd de

 1 2 3 4 4 5 5 6 6 8

Tree Edges : \emptyset

\emptyset

Kruskal's Algorithm - Example



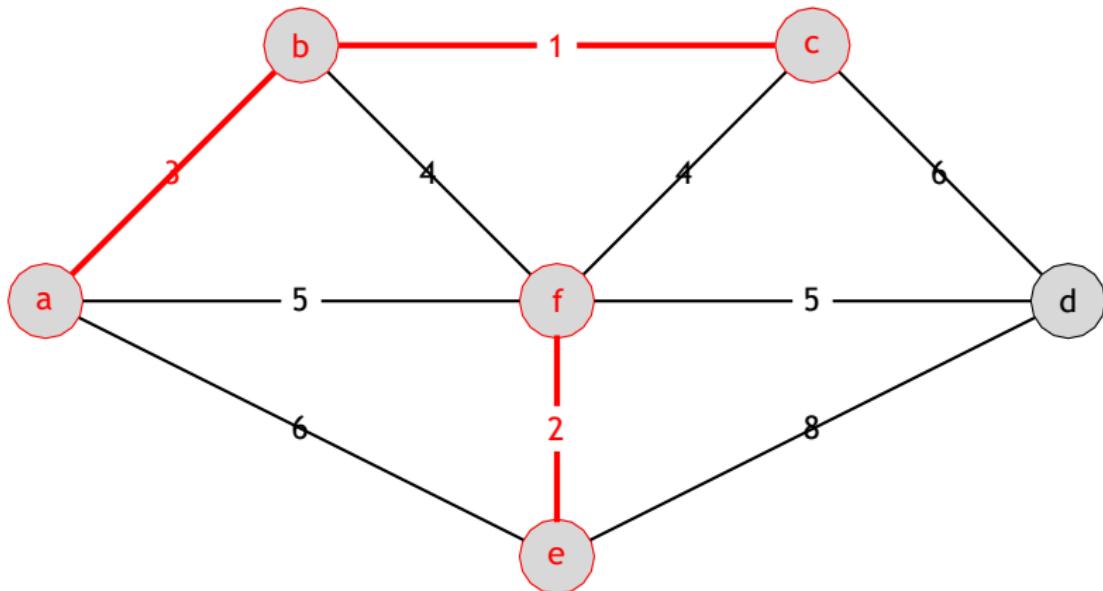
All Edges : bc ef ab bf cf af df ae cd de

 1 2 3 4 4 5 5 6 6 8

Tree Edges : bc

 1

Kruskal's Algorithm - Example



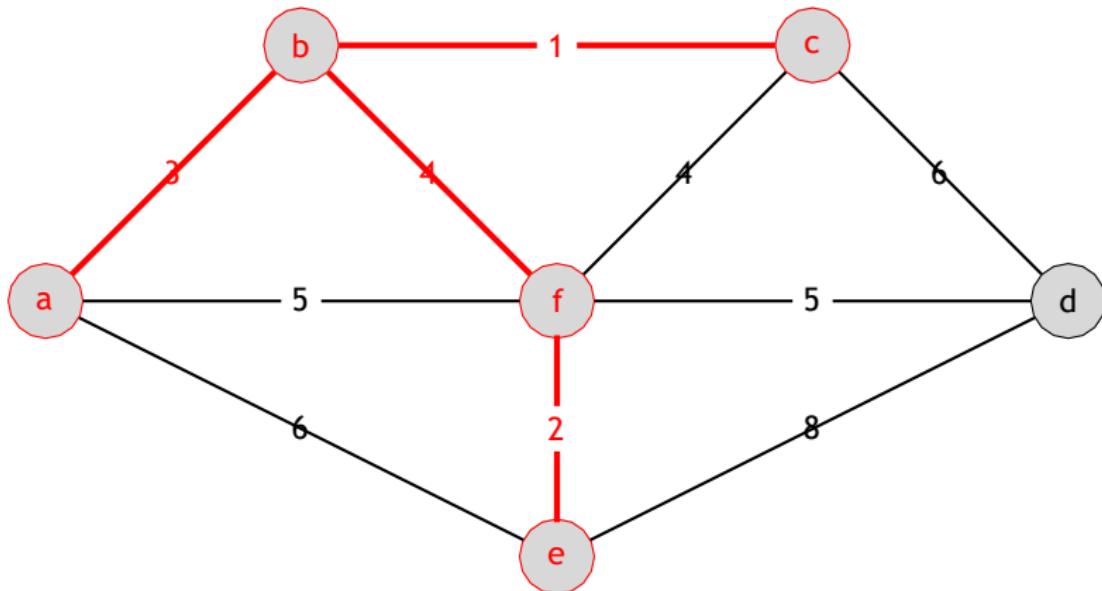
All Edges : bc ef ab bf cf af df ae cd de

 1 2 3 4 4 5 5 6 6 8

Tree Edges : bc ef

 1 2

Kruskal's Algorithm - Example



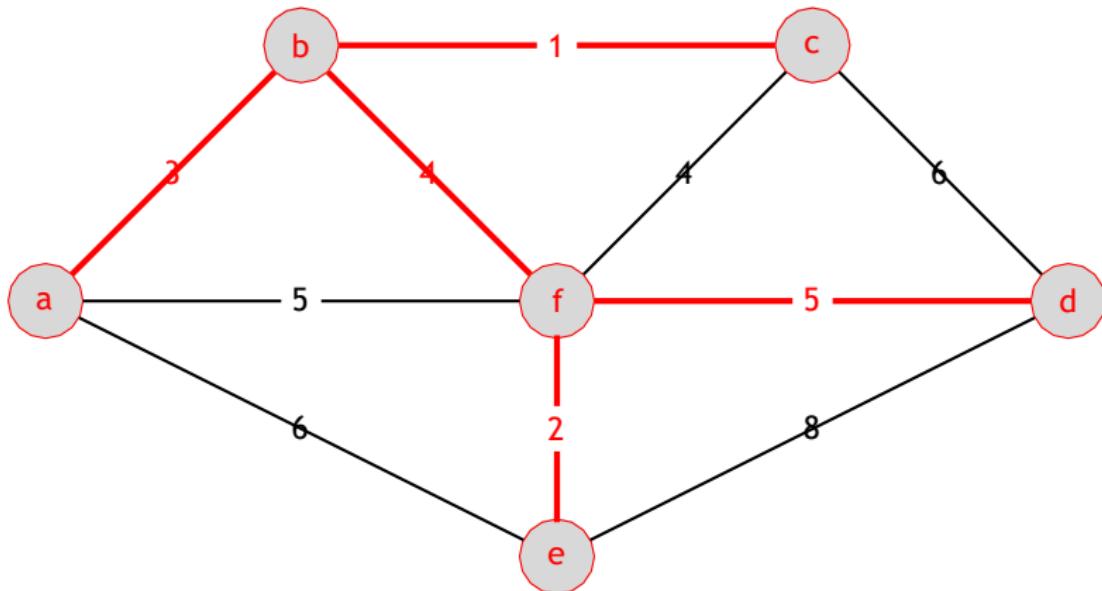
All Edges : bc ef ab **bf** cf af df ae cd de

 1 2 3 4 4 5 5 6 6 8

Tree Edges : bc ef ab

 1 2

Kruskal's Algorithm - Example



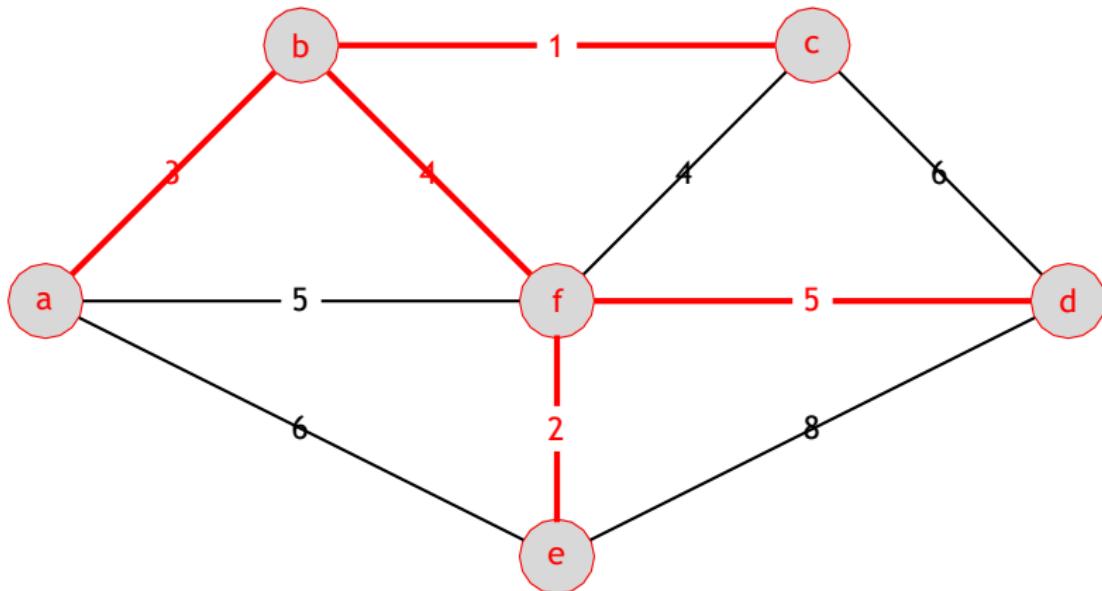
All Edges : bc ef ab bf cf af df ae cd de

 1 2 3 4 4 5 5 6 6 8

Tree Edges : bc ef ab bf

 1 2 3 4

Kruskal's Algorithm - Example



All Edges : bc ef ab bf cf af df ae cd de

 1 2 3 4 4 5 5 6 6 8

Tree Edges : bc ef ab bf df

 1 2 3 4 5

Kruskal's Algorithm - Summary

- Conceptually, Kruskal's algorithm appears easier than Prim's algorithm, but it is not.

Kruskal's Algorithm - Summary

- Conceptually, Kruskal's algorithm appears easier than Prim's algorithm, but it is not.
- It is harder to implement because checking for cycles requires us to maintain a forest of trees on the intermediate steps.

Kruskal's Algorithm - Summary

- Conceptually, Kruskal's algorithm appears easier than Prim's algorithm, but it is not.
- It is harder to implement because checking for cycles requires us to maintain a forest of trees on the intermediate steps.
- This is essentially a case of **Union-Find** algorithm.

Kruskal's Algorithm - Summary

- Conceptually, Kruskal's algorithm appears easier than Prim's algorithm, but it is not.
- It is harder to implement because checking for cycles requires us to maintain a forest of trees on the intermediate steps.
- This is essentially a case of **Union-Find** algorithm.
- With an efficient Union-find algorithm, Kruskal's algorithm is slightly less efficient than Prim's algorithm, requiring $O(|E| \log |E|)$ time (dominated by the time to sort the edges).

Kruskal's Algorithm - Summary

- Conceptually, Kruskal's algorithm appears easier than Prim's algorithm, but it is not.
- It is harder to implement because checking for cycles requires us to maintain a forest of trees on the intermediate steps.
- This is essentially a case of **Union-Find** algorithm.
- With an efficient Union-find algorithm, Kruskal's algorithm is slightly less efficient than Prim's algorithm, requiring $O(|E| \log |E|)$ time (dominated by the time to sort the edges).
- When the graph is *sparse*, $|E|$ same order of magnitude as $|V|$, then Kruskal's algorithm can usually be faster than Prim's (although same asymptotic complexity)

Shortest Spanning Tree or Minimum Spanning Tree?

Finally, I regret that the phrase “minimum spanning tree” has taken hold. For one thing “minimum” often degenerates to the badly incorrect “minimal”, and for another, “minimum” is so vague; minimum in what way? I always think of the concept as “shortest spanning subtree”, and hope someday to see SST replace MST.

JOSEPH B. KRUSKAL
BELL LABS
LUCENT TECHNOLOGIES
ROOM 2C-281
MURRAY HILL, NJ 07974

Overview

1 [Overview](#)

2 [Prim's Algorithm](#)

3 [Kruskal's Algorithm](#)

4 [Dijkstra's Algorithm](#)

5 [Data Compression](#)

6 Summary

Shortest Paths in Graphs

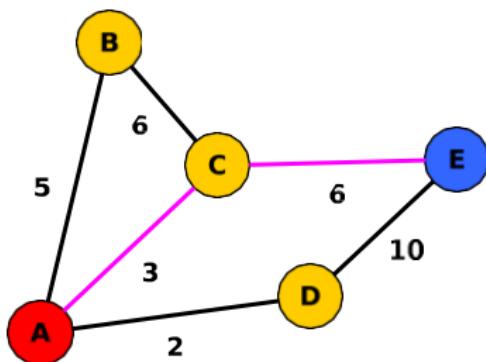
Problem

Given a weighted connected graph, the **shortest-path problem** asks to find the shortest path from a starting **source** vertex to a destination **target** vertex.

Shortest Paths in Graphs

Problem

Given a weighted connected graph, the **shortest-path problem** asks to find the shortest path from a starting **source** vertex to a destination **target** vertex.



Dijkstra's Algorithm

Problem

Given a weighted connected graph, the **single-source shortest-paths problem** asks to find the shortest path to all vertices given a single starting **source** vertex.

Idea:

- At all times, we maintain our **best estimate** of the shortest-path distances from source vertex to all other vertices.

Dijkstra's Algorithm

Problem

Given a weighted connected graph, the **single-source shortest-paths problem** asks to find the shortest path to all vertices given a single starting **source** vertex.

Idea:

- At all times, we maintain our **best estimate** of the shortest-path distances from source vertex to all other vertices.
- Initially we do not know, so all these distance estimates are ∞ .

Dijkstra's Algorithm

Problem

Given a weighted connected graph, the **single-source shortest-paths problem** asks to find the shortest path to all vertices given a single starting **source** vertex.

Idea:

- At all times, we maintain our **best estimate** of the shortest-path distances from source vertex to all other vertices.
- Initially we do not know, so all these distance estimates are ∞ .
- But as the algorithm explores the graph, we update our estimates, which converges to the true shortest path distance.

Dijkstra's Algorithm - Sketch

Maintain a set S of vertices whose final shortest-path weights from the source s have already been determined.

Dijkstra's Algorithm - Sketch

Maintain a set S of vertices whose final shortest-path weights from the source s have already been determined.

- ① Initially S is empty. Initialise distance estimates to ∞ for all non-source vertices. Distance of source vertex is 0.

Dijkstra's Algorithm - Sketch

Maintain a set S of vertices whose final shortest-path weights from the source s have already been determined.

- ① Initially S is empty. Initialise distance estimates to ∞ for all non-source vertices. Distance of source vertex is 0.
- ② Select the vertex v not in S with the **minimum** shortest-path estimate.

Dijkstra's Algorithm - Sketch

Maintain a set S of vertices whose final shortest-path weights from the source s have already been determined.

- ① Initially S is empty. Initialise distance estimates to ∞ for all non-source vertices. Distance of source vertex is 0.
- ② Select the vertex v not in S with the **minimum** shortest-path estimate.
- ③ Add v to S .

Dijkstra's Algorithm - Sketch

Maintain a set S of vertices whose final shortest-path weights from the source s have already been determined.

- ① Initially S is empty. Initialise distance estimates to ∞ for all non-source vertices. Distance of source vertex is 0.
- ② Select the vertex v not in S with the **minimum** shortest-path estimate.
- ③ Add v to S .
- ④ **Update** our distance estimates to **neighbouring** vertices that are not in S .

Dijkstra's Algorithm - Sketch

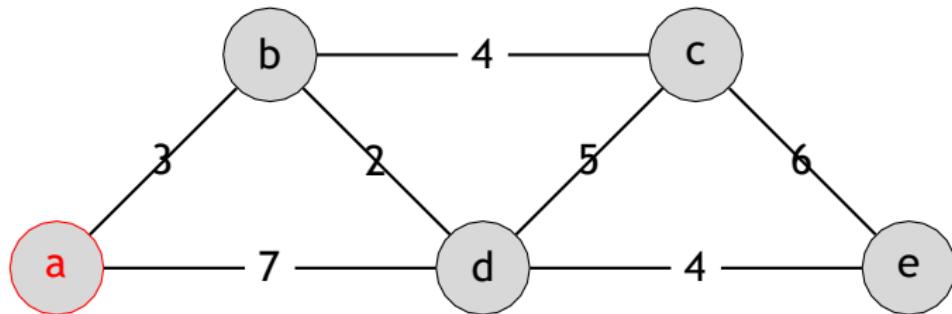
Maintain a set S of vertices whose final shortest-path weights from the source s have already been determined.

- ① Initially S is empty. Initialise distance estimates to ∞ for all non-source vertices. Distance of source vertex is 0.
- ② Select the vertex v not in S with the **minimum** shortest-path estimate.
- ③ Add v to S .
- ④ **Update** our distance estimates to **neighbouring** vertices that are not in S .
- ⑤ Repeat from step 2, until all vertices have been added to S .

Dijkstra's Algorithm - Pseudocode

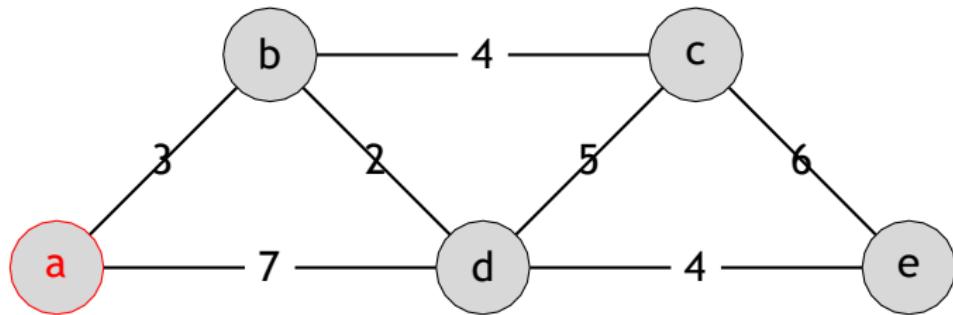
```
ALGORITHM Dijkstra ( $G, s$ )
/* Dijkstra's algorithm for single-source shortest paths. */
/* INPUT : A weighted connected graph  $G = \langle V, E \rangle$  and a starting vertex  $s$ . */
/* OUTPUT : The length  $d_v$  of a shortest path from  $s$  to  $v$  and its penultimate vertex  $p_v$  for every
vertex  $v$  in  $V$ . */
1: INITIALIZE( $Q$ )                                     d Initialize vertex priority queue.
2: for each  $v$  in  $V$  do
3:    $d_v = \infty$ ;  $p_v = \emptyset$ 
4:   INSERT( $Q, v, d_v$ )                                d Initialize vertex priority in the queue.
5: end for
6:  $d_s = 0$ ;  $V_T = \emptyset$ 
7: UPDATE( $Q, s, d_s$ )                                d Update the priority of  $s$  with  $d_s$ .
8: for  $i = 0$  to  $|V| - 1$  do
9:    $u^* = \text{DELETEMIN}(Q)$                          d Delete the minimum priority element.
10:   $V_T = V_T \cup \{u^*\}$ 
11:  for each  $u$  in  $V - V_T$  adjacent to  $u^*$  do
12:    if  $d_{u^*} + w(u^*, u) < d_u$  then
13:       $d_u = d_{u^*} + w(u^*, u)$ ;  $p_u = u^*$ 
14:      UPDATE( $Q, u, d_u$ )
15:    end if
16:  end for
17: end for
```

Dijkstra's Algorithm - Example



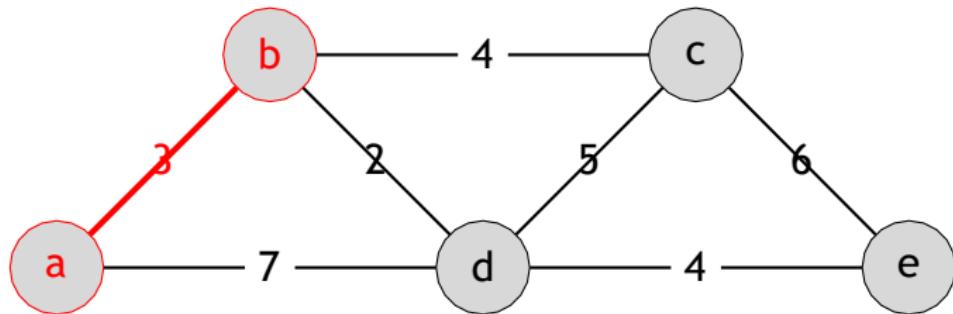
$a(a, 0)$ $b(-, \infty)$ $c(-, \infty)$ $d(-, \infty)$ $e(-, \infty)$
 $S = \{ \}$

Dijkstra's Algorithm - Example



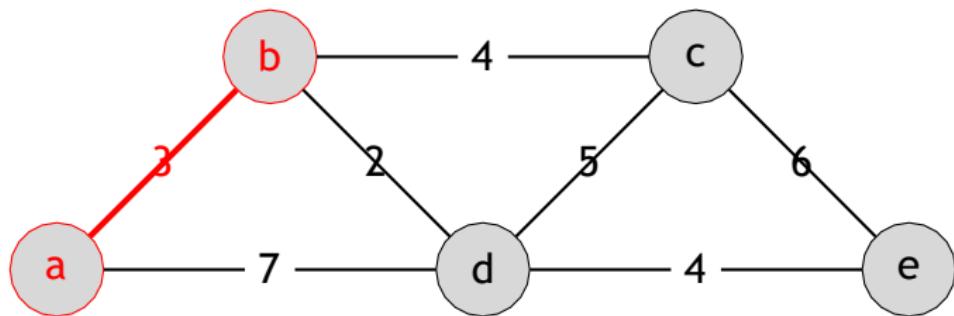
b(a,3) c(-,∞) d(a,7) e(-,∞)
 $S = \{a(a,0)\}$

Dijkstra's Algorithm - Example



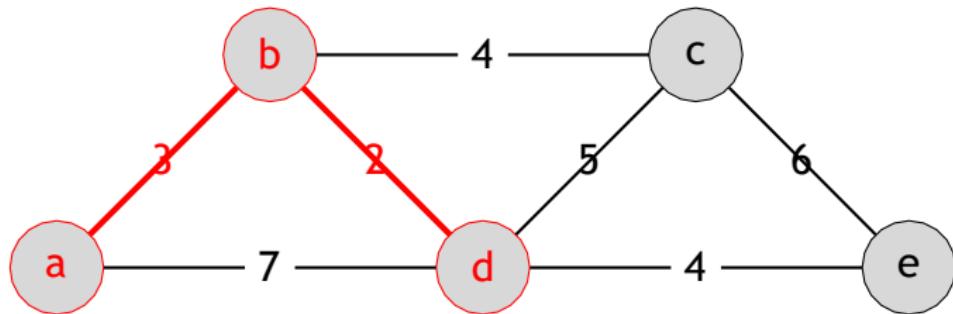
$b(a, 3)$ $c(-, \infty)$ $d(a, 7)$ $e(-, \infty)$
 $S = \{a(a, 0)\}$

Dijkstra's Algorithm - Example



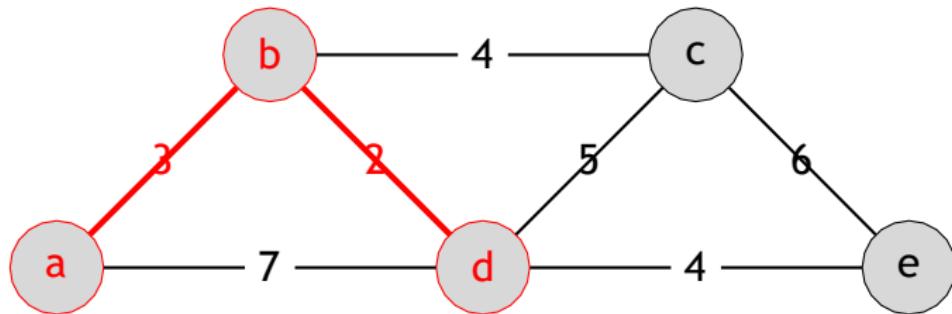
$c(b, 3 + 4) \quad d(b, 3 + 2) \quad e(-, \infty)$
 $S = \{a(a, 0), b(a, 3)\}$

Dijkstra's Algorithm - Example



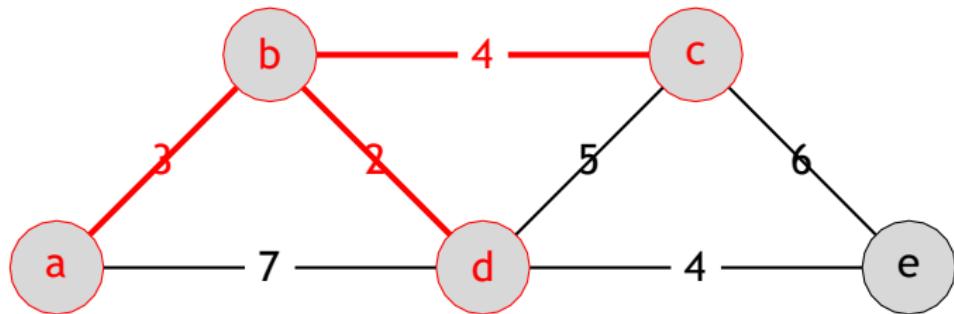
$c(b, 7)$ $d(b, 5)$ $e(-, \infty)$
 $S = \{a(a, 0), b(a, 3)\}$

Dijkstra's Algorithm - Example



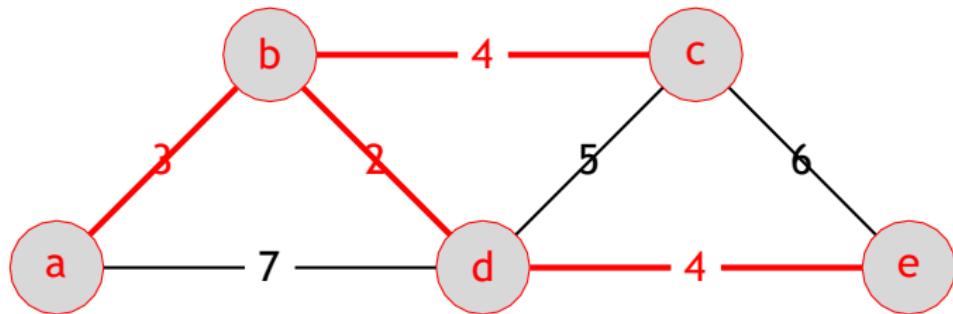
$c(b, 7)$ $e(d, 5+4)$
 $S = \{a(a, 0), b(a, 3), d(b, 5)\}$

Dijkstra's Algorithm - Example



$e(d, 9)$
 $S = \{a(a, 0), b(a, 3), d(b, 5), c(b, 7)\}$

Dijkstra's Algorithm - Example



$$S = \{a(a,0), b(a,3), d(b,5), c(b,7), e(d,9)\}$$

Dijkstra's Algorithm - Example

So, we have the following distances from vertex a :

$a(a,0)$ $b(a,3)$ $d(b,5)$ $c(b,7)$ $e(d,9)$

Which gives the following shortest paths:

Length	Path
3	$a - b$
5	$a - b - d$
7	$a - b - c$
9	$a - b - d - e$

Dijkstra's Algorithm - Summary

- Dijkstra's algorithm is guaranteed to always return the optimal solution. This is not necessarily true for all greedy algorithms.
- If we use an adjacency list and a min-heap, the algorithm runs in $\Theta(|E| \log |V|)$ time.

What is data compression?

- Data compression is the process of representing a data source in a reduced form. If there is no loss of information, it is called lossless compression.

Fixed Length Codes

The simplest encoding/decoding approach is to create a mapping from source alphabet to strings (**codewords**), where these codewords are **fixed** in length.

Example (Fixed Length Coding)

Given $S = \{a, c, g, t\}$, $\Sigma = \{0, 1\}$, and the encoding scheme,

$$a \rightarrow 00,$$

$$c \rightarrow 01,$$

$$g \rightarrow 10,$$

$$t \rightarrow 11,$$

then $\varphi(gattaca) = 10001111000100$.

Fixed Length Codes - ASCII

ASCII

American Standard Code for Information Interchange is a fixed length character encoding scheme over an alphabet of 128 characters.

ASCII Code Chart															
0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
	!	"	#	\$	%	&	.	()	:	*	+	,	-	.	/
0	1	2	3	4	5	6	7	8	9	I	J	K	L	M	N
@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
P	Q	R	S	T	U	V	W	X	Y	Z	[\]	^	-		
~	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
p	q	r	s	t	u	v	w	x	y	z	{ }	~	DEL		

char* string = AARDVARK

A	A	R	D	V	A	R	K
0x41	0x41	0x52	0x44	0x56	0x41	0x52	0x4B
1000001	1000001	1010010	1000100	1010110	1000001	1010010	1001011

Variable Length Codes

- Fixed length codewords are not the optimal in average bits per source symbol.
- Why? The frequency of appearance of each member of the source alphabet may not be uniformly distributed.
- Consider the letters ‘e’ and ‘z’ in natural language text, and using the same length codewords to represent both.
- e.g., “zee”
- Using ascii where each letter represented by 7 bits, this is $3 * 7 = 21$ bits

Character Frequency

Character	Frequency	Probability
e	24,600,752	0.0880
t	18,443,242	0.0660
a	17,379,446	0.0621
:	:	.
j	671,765	0.0024
q	264,712	0.0009
z	186,802	0.0007

The frequency of appearance of characters from the English alphabet extracted from a 267 MB segment of SGML-tagged newspaper text drawn from the WSJ component of the TREC data set.

Variable Length Codes

Solution?

- A **variable length code** maps each member of a source alphabet to a codeword string, but the **length** of codewords is no longer fixed.
- E.g., use a shorter codeword for 'e' and a larger one for 'z'.
- “zee” (hypothetically use 2 bit code for 'e' and '10' bit code for z, this is $10 + 2 \cdot 2 = 14$ bits)
- However, not all possible variable length coding schemes are decodeable.

Variable Length Codes - Decoding

Symbol	a	b	c	d	e	f	g
Frequency	25	12	9	4	3	2	1

Symbol	Codeword	l_i
a	0	1
b	1	1
c	00	2
d	01	2
e	10	2
f	11	3
g	110	3

Decode: 0010100010000111001011

Variable length codes must be chosen so text is uniquely decodeable.

Variable Length Codes - Prefix codes

Prefix Codes: Variable length codewords where no codeword is a prefix of any other codeword. Prefix codes are uniquely decodeable.

Symbol	Codeword	l_i
a	0	1
b	100	3
c	110	3
d	111	3
e	1010	4

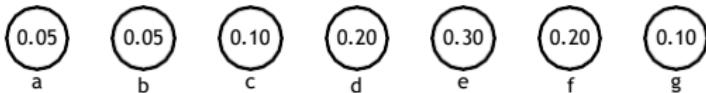
Huffman's Code - Sketch

Huffman Algorithm generates prefix codes that are optimal in the average number of bits per symbol.

Idea: Build prefix tree bottom up. Read the codes from this prefix tree.

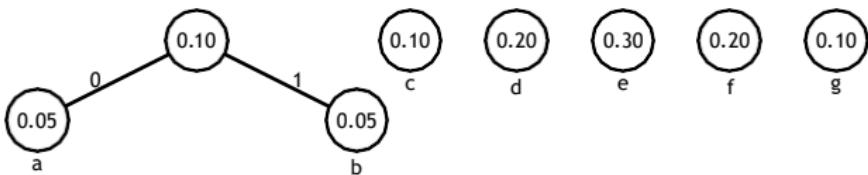
- ① For each symbol, calculate the probability of appearance.
Construct a leaf node for it.
- ② Put these leaf nodes to the set of candidate nodes (to merge).
- ③ Select the two nodes with the lowest probability (from candidate nodes) and combine them in a “bottom-up” tree construction.
- ④ The new parent has a probability equal to the sum of the two child probabilities, and replaces the two children in the set of candidate nodes. Add links to the children, one link with labelled ‘0’, the other ‘1’.
- ⑤ When only one candidate node remains, a tree has been formed, and codewords can be read from the edge labels of a tree.

Huffman Trees - Example



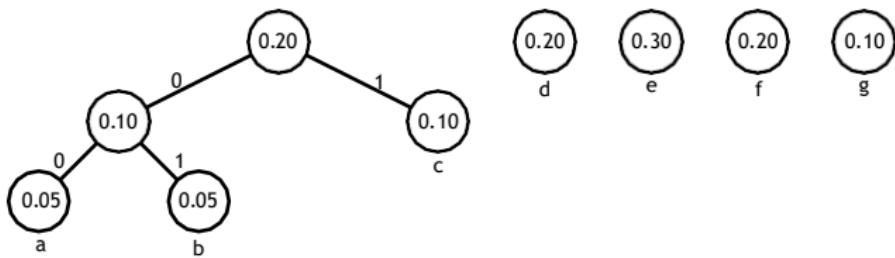
Construct a Huffman Tree from the Alphabet
 $\{(a, 0.05), (b, 0.05), (c, 0.1), (d, 0.2), (e, 0.3), (f, 0.2), (g, 0.1)\}$.

Huffman Trees - Example



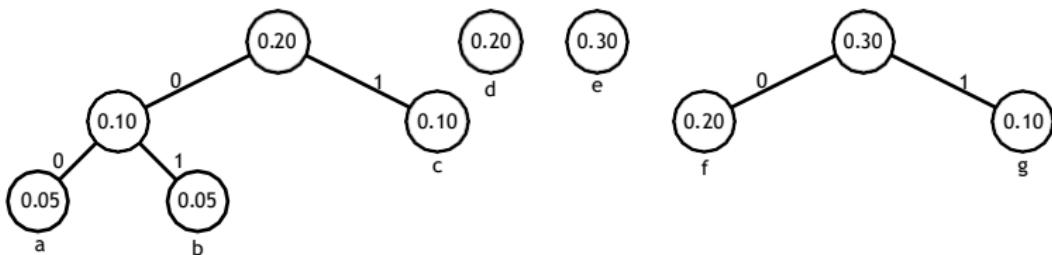
Construct a Huffman Tree from the Alphabet
 $\{(a, 0.05), (b, 0.05), (c, 0.1), (d, 0.2), (e, 0.3), (f, 0.2), (g, 0.1)\}$.

Huffman Trees - Example



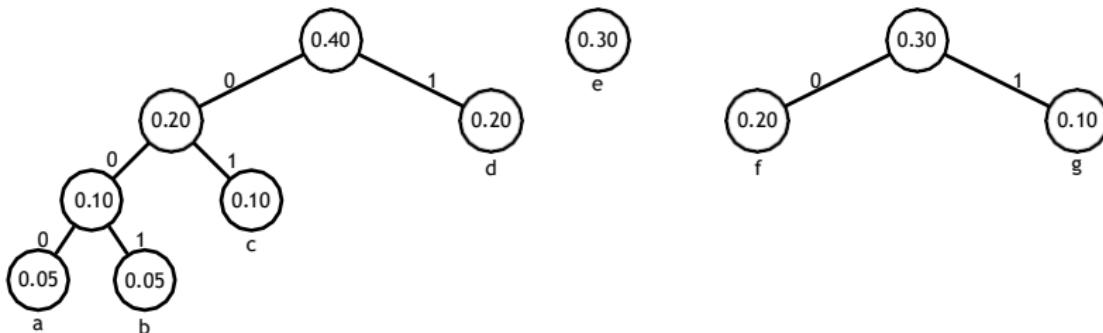
Construct a Huffman Tree from the Alphabet
 $\{(a, 0.05), (b, 0.05), (c, 0.1), (d, 0.2), (e, 0.3), (f, 0.2), (g, 0.1)\}$.

Huffman Trees - Example



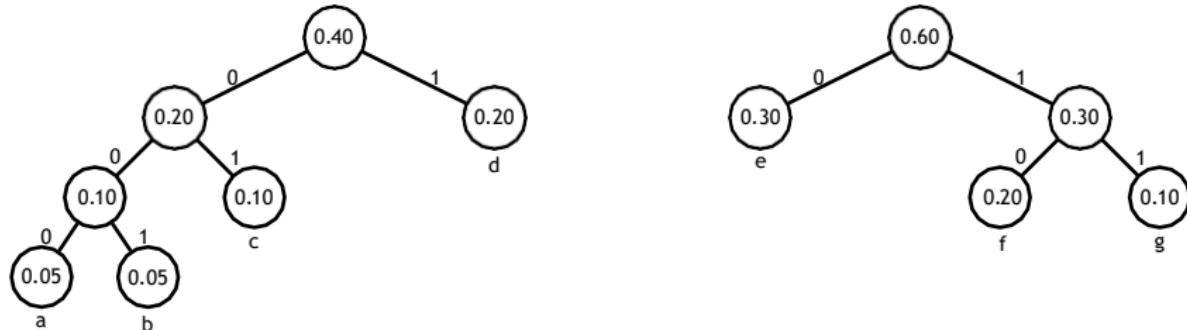
Construct a Huffman Tree from the Alphabet
 $\{(a, 0.05), (b, 0.05), (c, 0.1), (d, 0.2), (e, 0.3), (f, 0.2), (g, 0.1)\}$.

Huffman Trees - Example



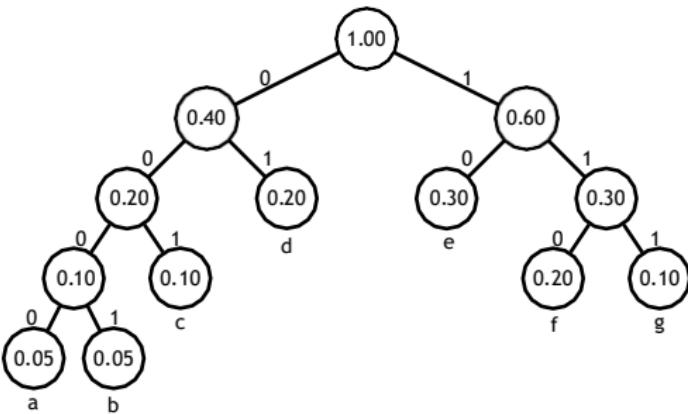
Construct a Huffman Tree from the Alphabet
 $\{(a, 0.05), (b, 0.05), (c, 0.1), (d, 0.2), (e, 0.3), (f, 0.2), (g, 0.1)\}$.

Huffman Trees - Example



Construct a Huffman Tree from the Alphabet
 $\{(a, 0.05), (b, 0.05), (c, 0.1), (d, 0.2), (e, 0.3), (f, 0.2), (g, 0.1)\}$.

Huffman Trees - Example



Construct a Huffman Tree from the Alphabet
 $\{(a, 0.05), (b, 0.05), (c, 0.1), (d, 0.2), (e, 0.3), (f, 0.2), (g, 0.1)\}$.

Huffman Codes

Symbol	Codeword	l_i
a	0000	4
b	0001	4
c	001	3
d	01	2
e	10	2
f	110	3
g	111	3

The Huffman Codes and the corresponding codeword lengths.

Huffman Codes

- This approach requires $\Theta(n \log n)$ time if a min heap (priority queue) is used to manage the set of candidates and their weights.
- If the input list is already sorted by their probabilities, then the codes can be constructed in $\Theta(n)$ time.

Summary

- Understand and be able to apply the greedy approach to solving problems.
- Examples:
 - spanning tree - Prim's algorithm
 - spanning tree - Kruskal's algorithm
 - single source shortest-path - Dijkstra's algorithm
 - data compression

Levitin - The design and analysis of algorithms

This week we will be covering the material from Chapters 10.

Learning outcomes:

- Understand the paradigm of iterative improvement
- Understand and apply examples of iterative improvement:
 - Maximum-flow problem
 - Stable marriage problem (Gale-Shapeley algorithm)

Iterative Improvement

Previously, we looked at greedy approaches to solving optimisation problems. It constructs a solution piece by piece, in a greedy fashion.

In contrast, **iterative improvement** starts with a feasible solution (one that satisfies all constraints), then proceed to improve it by repeated application of simple steps.

Maximum-Flow Problem

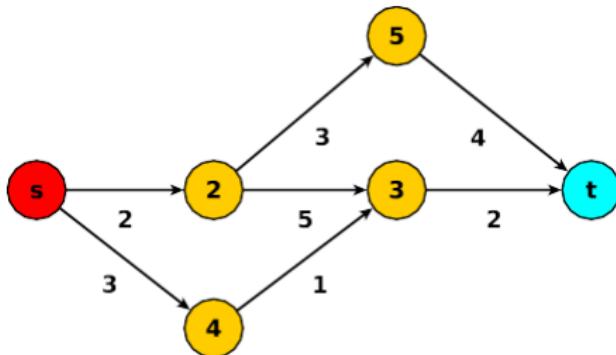
Imagine you are given this problem:

Problem

Yarra Valley Waters needs to move water from a dam to a local water reservoir. There is a network of pipes and junctions that they can transport water over. Assume there is no loss within the network. How do we maximise the amount of water transported each day?

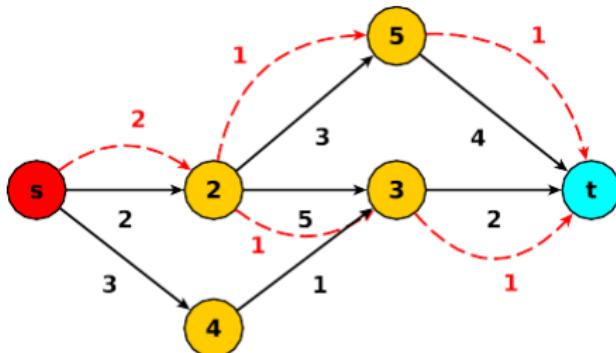
This is an instance of a **maximum-flow** problem.

Flow Network



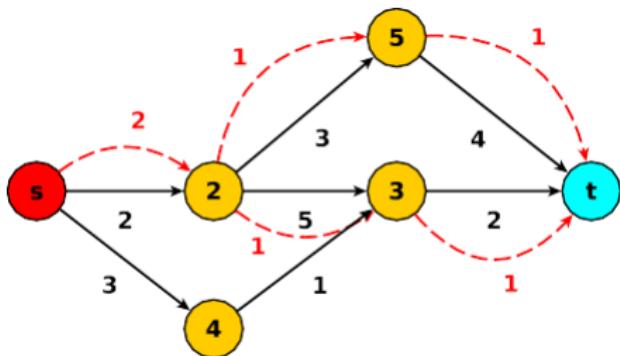
- **Source**: vertex which has no incoming edges.
- **Sink**: vertex with no outgoing edges.
- Edge has a weight representing its **capacity**.
- Graphs satisfying above properties called **flow network**.

Maximum-Flow Problem



- **Source** is the origin of all “material” into the flow network.
- **Sink** is the final destination of all “material” in the network.
- Maximum amount of flow on an edge cannot exceed its capacity; **capacity** constraint.
- All other vertices are transit points - flow in = flow out; called **flow-conservation**.
- Total material leaving source = total material flowing into sink; called **value** of the flow.

Maximum-Flow Problem



Problem

Given a flow network, the **maximum-flow problem** is find a flow of maximum **value**, subject to (edge) capacity constraints and flow-conservation.

Ford Fulkerson Method - Sketch

Idea:

- Given an initial flow network, find an initial **feasible** flow.
- Find a path from source to sink that can increase the total flow.
- Increase the flow along that path.

Ford Fulkerson Method - Sketch

Idea:

- Given an initial flow network, find an initial **feasible** flow.
- Find a path from source to sink that can increase the total flow.
- Increase the flow along that path.
- Repeat until no more such paths.

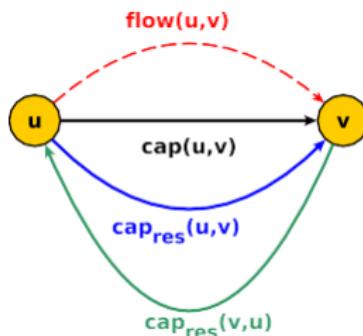
Preliminaries: Residual Network

Purpose: Given a flow, the **residual network** shows which edges in the flow network can admit more flow.

For each edge (u,v) in flow network, we have two edges in the residual network with following residual capacity:

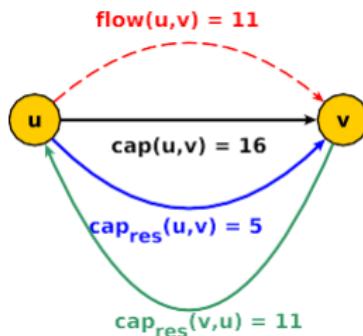
$$\text{cap}_{\text{res}}(u,v) = \text{cap}(u,v) - \text{flow}(u,v), \text{cap}_{\text{res}}(u,v) > 0 \text{ (forward edge)}$$

$$\text{cap}_{\text{res}}(v,u) = \text{flow}(u,v), \text{cap}_{\text{res}}(v,u) > 0 \text{ (backward edge)}$$



Preliminaries: Residual Network

Example: $\text{cap}(u,v) = 16$, $\text{flow}(u,v) = 11$, then can still increase the flow by $\text{cap}_{\text{res}}(u,v) = 5$ in the (u,v) direction, but can also send up to $\text{cap}_{\text{res}}(v,u) = 11$ units in the other (v,u) direction to cancel out $\text{flow}(u,v)$.

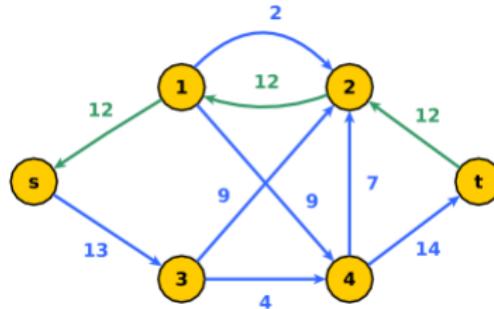


Preliminaries: Augmenting Path

Given a residual network, an **augmenting path** is a path from s to t in the residual graph.

From the definition of a residual network, we know each edge in the augmenting path can admit **additional positive flow** without violating the capacity of the edge.

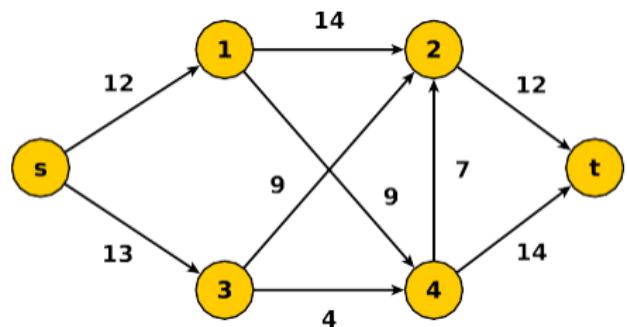
This path basically tells us which individual flows to **increase** (when traversing forward edge), and which to **decrease** (when traversing backward edge) to increase the total flow/value, while satisfying capacity and flow conservation constraints.



Ford-Fulkerson Method

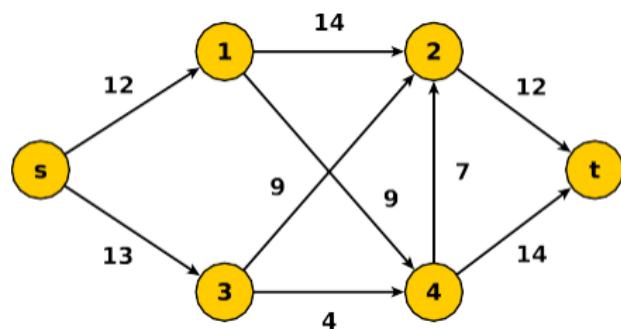
- ① For each edge (u,v) in flow network, set $\text{flow}(u,v) = 0$ and $\text{flow}(v,u) = 0$.
 - ② Construct residual network from flow network + current flow.
 - ③ If there is a (augmenting) path* p from s to t in the residual network:
 - a. For all the edges in path p in residual network, find the one with minimum residual capacity (cap_{\minres})
 - b. For each edge (u,v) in path p , update the flows on flow network:
if $((u,v)$ is forward) then $\text{flow}(u,v) += \text{cap}_{\minres}$;
if $((u,v)$ is backward) then $\text{flow}(u,v) -= \text{cap}_{\minres}$
 - c. Update residual network
 - ④ repeat step 3 until no more paths from s to t in residual network
- * There are generally many possible augmenting path. We select the shortest one (in terms of number of edges) for step 3.

Ford-Fulkerson Method - Example

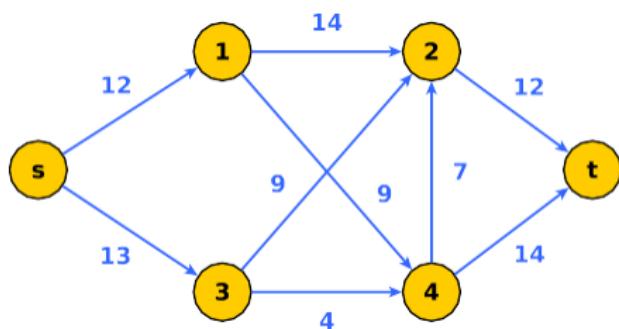


(a) Flow network

Ford-Fulkerson Method - Example

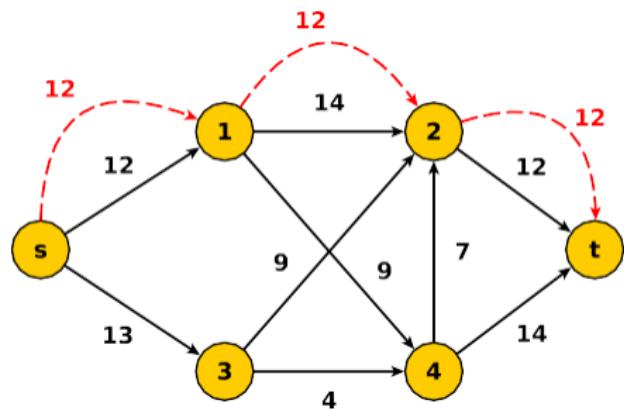


(c) Flow network



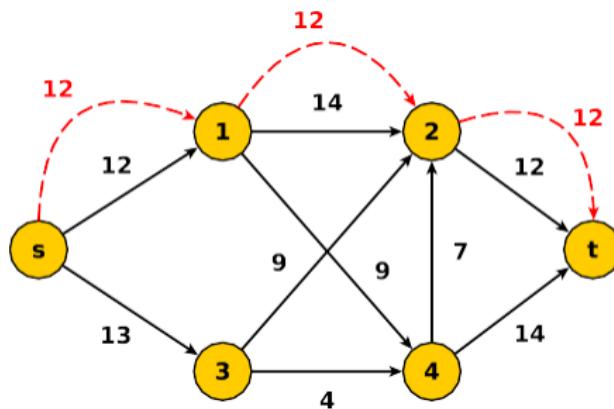
(d) Residual network

Ford-Fulkerson Method - Example

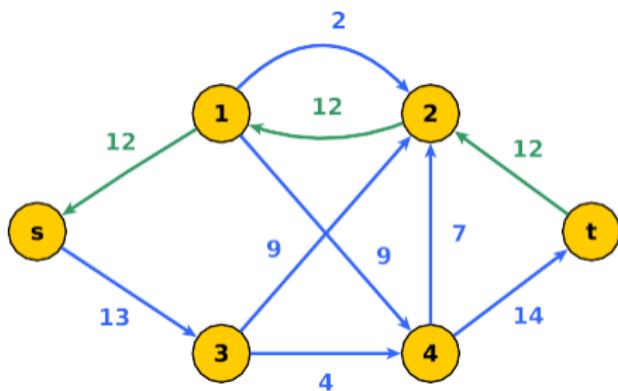


(e) Flow network

Ford-Fulkerson Method - Example

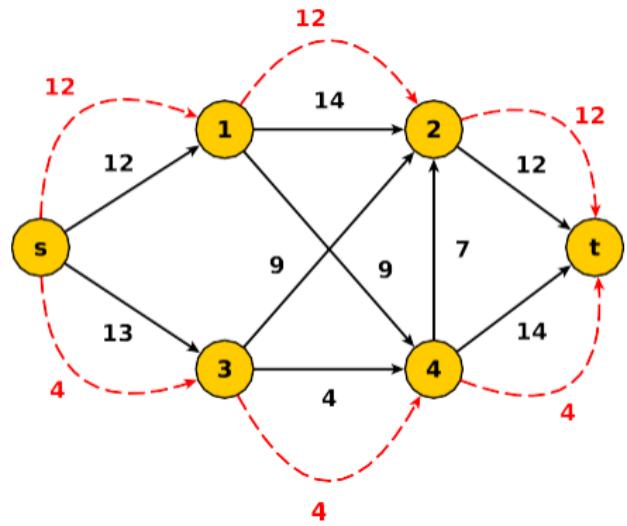


(g) Flow network



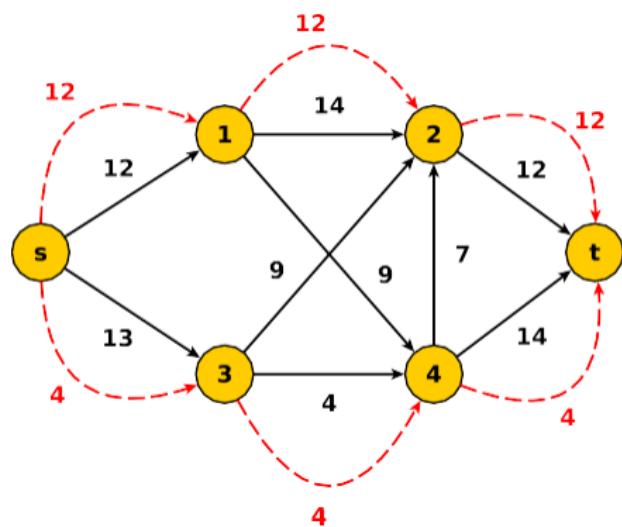
(h) Residual network

Ford-Fulkerson Method - Example

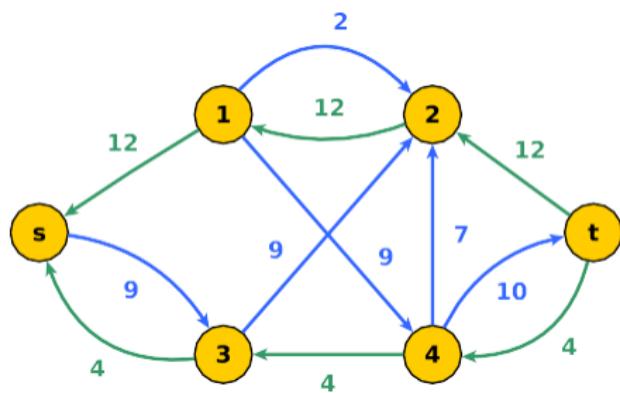


(i) Flow network

Ford-Fulkerson Method - Example

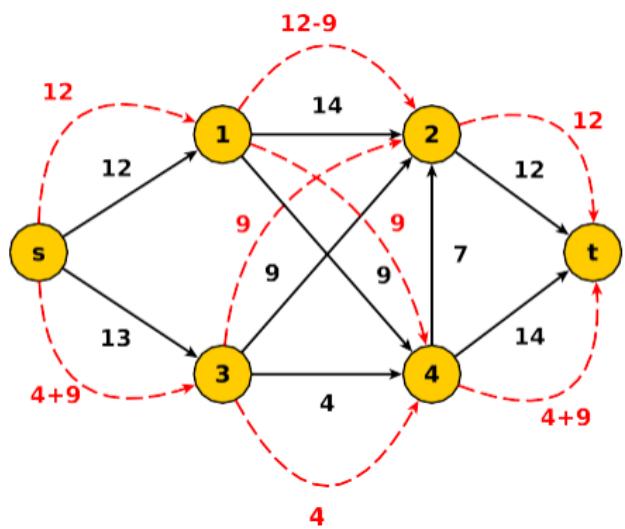


(k) Flow network



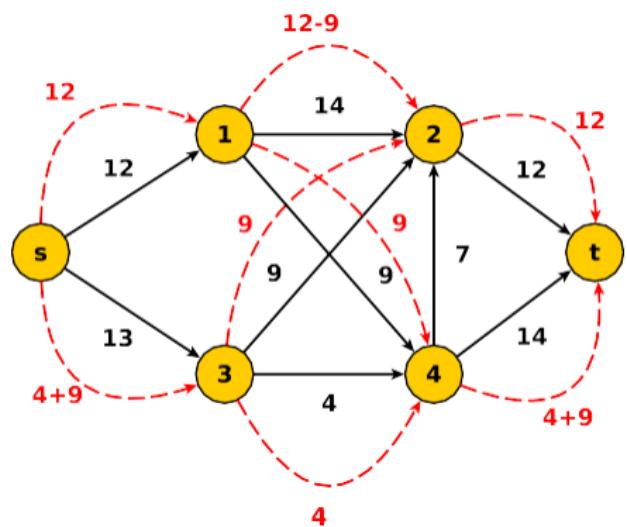
(l) Residual network

Ford-Fulkerson Method - Example

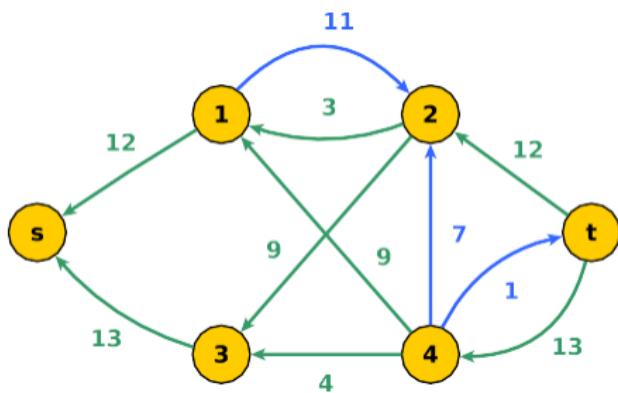


(m) Flow network

Ford-Fulkerson Method - Example

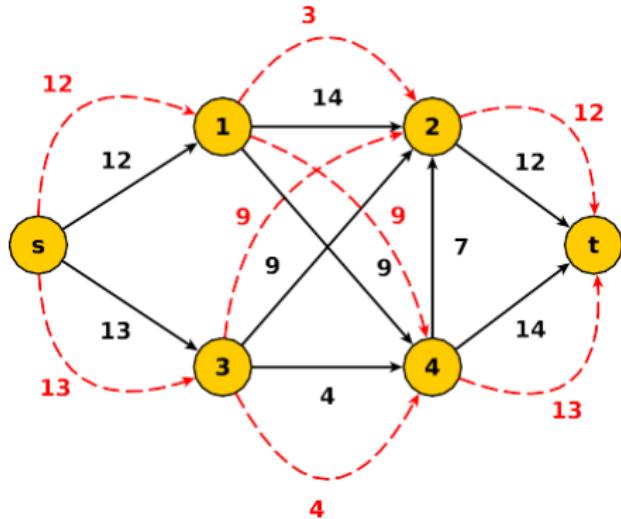


(o) Flow network



(p) Residual network

Ford-Fulkerson Method - Example



Value of flow?

Maximum-Flow Problem

Applications of maximum-flow problem:

<https://www.youtube.com/watch?v=D36MJCXT4Qk>

Stable Marriage Problem



Danny



Anita



Chris



Sarah



John



Barbie



Ken



Michelle

(q) Males and Females.

Stable Marriage Problem



Danny



Anita



Chris



Sarah



John



Barble



Ken



Michelle



(s) Males and Females.

(t) Matched couples!

Stable Marriage Problem

Stable Marriage Problem

Given a set of n men and n women, who has a list of preferences to the other sex (in terms of a ranking), the problem is how to find a matching between them such that the matching is *stable*?

A matching is stable if:

- No matched pair of man and woman can find other partners and **both** do better, i.e., both man and woman prefer other partners over their existing match.

Is a stable marriage (matching) always possible?

- Yes, if equal number of men and women.

Gale-Shapley Algorithm

Idea: Female proposing variant:

- ① Over a number of rounds, each unmatched female proposes to their remaining highest male preferences.
- ② Each round, males reply “yes” to proposal from their highest female proposer and “no” to all other proposers.
- ③ This continues until all females (and males) are matched.

Gale-Shapley Algorithm

Female Proposing variant:

- ① Round 1: Each female **proposes** to their **first** male preferences. Each male receives 0 or more proposals. They reply “maybe” to the female they **most prefer** and “no” to all other proposals. For each “maybe” reply, the corresponding female-male are **tentatively** matched.
- ② Round 2: Each **unmatched** female proposes to their **next** preferred male (2nd ranked), even if the male is tentatively matched already. Each male evaluates their proposals, and again reply “maybe” to the female they most prefer (this can be their existing matched partner) and “no” to all other proposals. For each “maybe” reply, the corresponding female-male are **tentatively** matched.
- ③ Round 3 onwards: We continue with this process until all females and males are matched.

Gale-Shapley Algorithm Example



Danny



Anita



Chris



Sarah



John



Barble



Ken



Michelle

Gale-Shapley Algorithm Example

Anita	Sarah	Barbie	Michelle
Chris	Chris	Ken	Chris
Ken	Ken	Danny	Ken
Danny	Danny	John	Danny
John	John	Chris	John

Table: Female preferences

Danny	Chris	John	Ken
Michelle	Anita	Barbie	Michelle
Sarah	Sarah	Anita	Barbie
Barbie	Michelle	Michelle	Sarah
Anita	Barbie	Sarah	Anita

Table: Male preferences

Gale-Shapley Algorithm Example

Round 1 (proposing):

Anita	Sarah	Barbie	Michelle
Chris	Chris	Ken	Chris

Table: Female preferences

Danny	Chris	John	Ken
Michelle	Anita	Barbie	Michelle
Sarah	Sarah	Anita	Barbie
Barbie	Michelle	Michelle	Sarah
Anita	Barbie	Sarah	Anita

Table: Male preferences

Gale-Shapley Algorithm Example

Round 1 (end):

Anita	Sarah	Barbie	Michelle
Chris	Chris	Ken	Chris

Table: Female preferences

Danny	Chris	John	Ken
Michelle	Anita	Barbie	Michelle
Sarah	Sarah	Anita	Barbie
Barbie	Michelle	Michelle	Sarah
Anita	Barbie	Sarah	Anita

Table: Male preferences

Gale-Shapley Algorithm Example

Round 2 (proposing):

Anita	Sarah	Barbie	Michelle
Chris	Chris Ken	Ken	Chris Ken

Table: Female preferences

Danny	Chris	John	Ken
Michelle	Anita	Barbie	Michelle
Sarah	Sarah	Anita	Barbie
Barbie	Michelle	Michelle	Sarah
Anita	Barbie	Sarah	Anita

Table: Male preferences

Gale-Shapley Algorithm Example

Round 2 (end):

Anita	Sarah	Barbie	Michelle
Chris	Chris Ken	Ken	Chris Ken

Table: Female preferences

Danny	Chris	John	Ken
Michelle	Anita	Barbie	Michelle
Sarah	Sarah	Anita	Barbie
Barbie	Michelle	Michelle	Sarah
Anita	Barbie	Sarah	Anita

Table: Male preferences

Gale-Shapley Algorithm Example

Round 3 (proposing):

Anita	Sarah	Barbie	Michelle
Chris	Chris Ken Danny	Ken Danny	Chris Ken

Table: Female preferences

Danny	Chris	John	Ken
Michelle	Anita	Barbie	Michelle
Sarah	Sarah	Anita	Barbie
Barbie	Michelle	Michelle	Sarah
Anita	Barbie	Sarah	Anita

Table: Male preferences

Gale-Shapley Algorithm Example

Round 3 (end):

Anita	Sarah	Barbie	Michelle
Chris	Chris Ken Danny	Ken Danny	Chris Ken

Table: Female preferences

Danny	Chris	John	Ken
Michelle	Anita	Barbie	Michelle
Sarah	Sarah	Anita	Barbie
Barbie	Michelle	Michelle	Sarah
Anita	Barbie	Sarah	Anita

Table: Male preferences

Gale-Shapley Algorithm Example

Round 4 (proposing):

Anita	Sarah	Barbie	Michelle
Chris	Chris Ken Danny	Ken Danny John	Chris Ken

Table: Female preferences

Danny	Chris	John	Ken
Michelle	Anita	Barbie	Michelle
Sarah	Sarah	Anita	Barbie
Barbie	Michelle	Michelle	Sarah
Anita	Barbie	Sarah	Anita

Table: Male preferences

Gale-Shapley Algorithm Example

Round 4 (end):

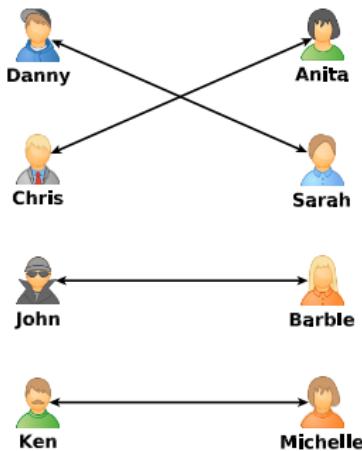
Anita	Sarah	Barbie	Michelle
Chris	Chris Ken Danny	Ken Danny John	Chris Ken

Table: Female preferences

Danny	Chris	John	Ken
Michelle	Anita	Barbie	Michelle
Sarah	Sarah	Anita	Barbie
Barbie	Michelle	Michelle	Sarah
Anita	Barbie	Sarah	Anita

Table: Male preferences

Gale-Shapley Algorithm Example



Gale-Shapley Algorithm

Properties of algorithm:

- It will always find a stable marriage configuration.
- All males and females will be matched.
- For the female proposing variant, the females are matched with the best male partners under any stable marriage, but the males can be matched with their worst female partners under any stable marriage.
- Stable matchings may not be unique, e.g., the male proposing variant could come up with a different stable matching.

Time complexity?

Stable Marriage Video

<https://www.youtube.com/watch?v=fudb8DuzQlM>

Summary

- Maximum flow problem (Ford-Fulkerson method)
- Stable marriage problem (Gale-Shapley algorithm)

Overview

Levitin - The design and analysis of algorithms

This week we will be covering the material from Chapter 7.

Learning outcomes:

- Understand space-time tradeoffs in algorithm design.
- Two of the paradigms:
 - input enhancement (sort by counting)
 - prestructuring (hashing)

Time & Space Tradeoffs

We can gain time by using more space - whole idea behind this lecture.

In this lecture, we discuss two varieties of Time & Space tradeoffs:

- ① **Input Enhancement** - preprocess the input to store extra information that will accelerate the solving of the problem.
 - counting sorts
- ② **Prestructuring** - use extra space to make accessing its elements easier or faster.
 - hashing

Count-based Sorting

When sorting a list with many repeated values, can we do better than the sorts we have seen?

- Rough idea: Imagine we have array with three unique values, 1, 2, 3
- Imagine if our array consist of 2,1,2,3,1
- If I arrange the array with the 1s, then the 2s, then the 3s, then I have sorted the array (1,1,2,2,3).
- **Distribution sort** does exactly this, in a smart way.

Distribution Sorting

12	4	12	9	12	15	9	4	9
----	---	----	---	----	----	---	---	---

4	9	12	15
0	0	0	0

1	2	3	4	5	6	7	8	9

Distribution Sorting

12	4	12	9	12	15	9	4	9
----	---	----	---	----	----	---	---	---

4	9	12	15
0	0	1	0

1	2	3	4	5	6	7	8	9

Distribution Sorting

12	4	12	9	12	15	9	4	9
----	---	----	---	----	----	---	---	---

4	9	12	15
1	0	1	0

1	2	3	4	5	6	7	8	9

Distribution Sorting

12	4	12	9	12	15	9	4	9
----	---	----	---	----	----	---	---	---

4	9	12	15
1	0	2	0

1	2	3	4	5	6	7	8	9

Distribution Sorting

12	4	12	9	12	15	9	4	9
----	---	----	---	----	----	---	---	---

4	9	12	15
1	1	2	0

1	2	3	4	5	6	7	8	9

Distribution Sorting

12	4	12	9	12	15	9	4	9
----	---	----	---	----	----	---	---	---

4	9	12	15
1	1	3	0

1	2	3	4	5	6	7	8	9

Distribution Sorting

12	4	12	9	12	15	9	4	9
----	---	----	---	----	----	---	---	---

4	9	12	15
1	1	3	1

1	2	3	4	5	6	7	8	9

Distribution Sorting

12	4	12	9	12	15	9	4	9
----	---	----	---	----	----	---	---	---

4	9	12	15
1	2	3	1

1	2	3	4	5	6	7	8	9

Distribution Sorting

12	4	12	9	12	15	9	4	9
----	---	----	---	----	----	---	---	---

4	9	12	15
2	2	3	1

1	2	3	4	5	6	7	8	9

Distribution Sorting

12	4	12	9	12	15	9	4	9
----	---	----	---	----	----	---	---	---

4	9	12	15
2	3	3	1

1	2	3	4	5	6	7	8	9

Distribution Sorting

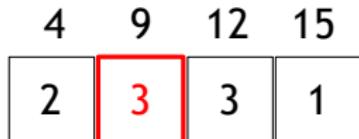
12	4	12	9	12	15	9	4	9
----	---	----	---	----	----	---	---	---

4	9	12	15
2	3	3	1

1	2	3	4	5	6	7	8	9

Distribution Sorting

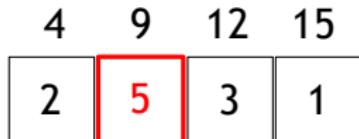
12	4	12	9	12	15	9	4	9
----	---	----	---	----	----	---	---	---



1	2	3	4	5	6	7	8	9

Distribution Sorting

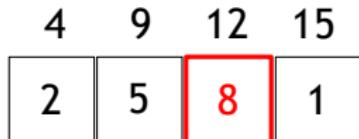
12	4	12	9	12	15	9	4	9
----	---	----	---	----	----	---	---	---



1	2	3	4	5	6	7	8	9

Distribution Sorting

12	4	12	9	12	15	9	4	9
----	---	----	---	----	----	---	---	---



1	2	3	4	5	6	7	8	9

Distribution Sorting

12	4	12	9	12	15	9	4	9
----	---	----	---	----	----	---	---	---

4	9	12	15
2	5	8	9

1	2	3	4	5	6	7	8	9

Distribution Sorting

12	4	12	9	12	15	9	4	9
----	---	----	---	----	----	---	---	---

4	9	12	15
2	5	8	9

1	2	3	4	5	6	7	8	9

Distribution Sorting

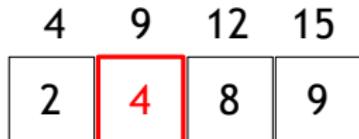
12	4	12	9	12	15	9	4	9
----	---	----	---	----	----	---	---	---

4	9	12	15
2	5	8	9

1	2	3	4	5	6	7	8	9
				9				

Distribution Sorting

12	4	12	9	12	15	9	4	9
----	---	----	---	----	----	---	---	---



1	2	3	4	5	6	7	8	9
				9				

Distribution Sorting

12	4	12	9	12	15	9	4	9
----	---	----	---	----	----	---	---	---

4	9	12	15
2	4	8	9

1	2	3	4	5	6	7	8	9
	4			9				

Distribution Sorting

12	4	12	9	12	15	9	4	9
----	---	----	---	----	----	---	---	---

4	9	12	15
1	4	8	9

1	2	3	4	5	6	7	8	9
	4			9				

Distribution Sorting

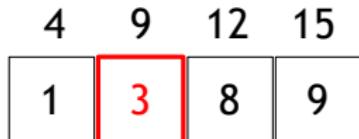
12	4	12	9	12	15	9	4	9
----	---	----	---	----	----	---	---	---

4	9	12	15
1	4	8	9

1	2	3	4	5	6	7	8	9
	4		9	9				

Distribution Sorting

12	4	12	9	12	15	9	4	9
----	---	----	---	----	----	---	---	---



1	2	3	4	5	6	7	8	9
	4		9	9				

Distribution Sorting

12	4	12	9	12	15	9	4	9
----	---	----	---	----	----	---	---	---

4	9	12	15
1	3	8	9

1	2	3	4	5	6	7	8	9
	4		9	9				15

Distribution Sorting

12	4	12	9	12	15	9	4	9
----	---	----	---	----	----	---	---	---

4	9	12	15
1	3	8	8

1	2	3	4	5	6	7	8	9
	4		9	9				15

Distribution Sorting

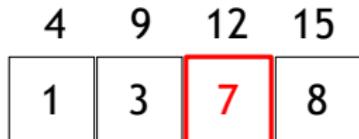
12	4	12	9	12	15	9	4	9
----	---	----	---	----	----	---	---	---

4	9	12	15
1	3	8	8

1	2	3	4	5	6	7	8	9
	4		9	9			12	15

Distribution Sorting

12	4	12	9	12	15	9	4	9
----	---	----	---	----	----	---	---	---



1	2	3	4	5	6	7	8	9
	4		9	9			12	15

Distribution Sorting

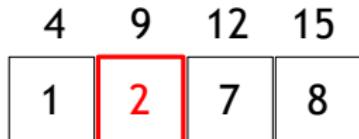
12	4	12	9	12	15	9	4	9
----	---	----	---	----	----	---	---	---

4	9	12	15
1	3	7	8

1	2	3	4	5	6	7	8	9
	4	9	9	9			12	15

Distribution Sorting

12	4	12	9	12	15	9	4	9
----	---	----	---	----	----	---	---	---



1	2	3	4	5	6	7	8	9
	4	9	9	9			12	15

Distribution Sorting

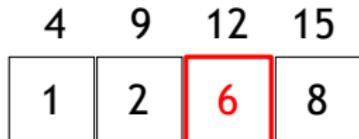
12	4	12	9	12	15	9	4	9
----	---	----	---	----	----	---	---	---

4	9	12	15
1	2	7	8

1	2	3	4	5	6	7	8	9
	4	9	9	9		12	12	15

Distribution Sorting

12	4	12	9	12	15	9	4	9
----	---	----	---	----	----	---	---	---



1	2	3	4	5	6	7	8	9
	4	9	9	9		12	12	15

Distribution Sorting

12	4	12	9	12	15	9	4	9
----	---	----	---	----	----	---	---	---

4	9	12	15
1	2	6	8

1	2	3	4	5	6	7	8	9
4	4	9	9	9		12	12	15

Distribution Sorting

12	4	12	9	12	15	9	4	9
----	---	----	---	----	----	---	---	---

4	9	12	15
0	2	6	8

1	2	3	4	5	6	7	8	9
4	4	9	9	9		12	12	15

Distribution Sorting

12	4	12	9	12	15	9	4	9
----	---	----	---	----	----	---	---	---

4	9	12	15
0	2	6	8

1	2	3	4	5	6	7	8	9
4	4	9	9	9	12	12	12	15

Distribution Sorting

12	4	12	9	12	15	9	4	9
----	---	----	---	----	----	---	---	---

4 9 12 15

0	2	5	8
---	---	---	---

1 2 3 4 5 6 7 8 9

4	4	9	9	9	12	12	12	15
---	---	---	---	---	----	----	----	----

Distribution Sorting

```
ALGORITHM DistCountSort( $A[0 \dots n-1], u, l$ )
```

//Sort an array by distribution counting

//INPUT : An array $A[0 \dots n-1]$ of orderable integers between l and u ($l \leq u$), and $n_{\max} = u - l$

//OUTPUT : An array $S[0 \dots n-1]$ of integers sorted in nondecreasing order.

```
1: for  $j = 0$  to  $n_{\max}$  do d Initialize frequencies
2:    $\sigma[j] = 0$ 
3: end for
4: for  $i = 0$  to  $n-1$  do d Compute frequencies
5:    $\sigma[A[i]-l] = \sigma[A[i]-l] + 1$ 
6: end for
7: for  $j = 1$  to  $n_{\max}$  do d Compute cumulative frequencies
8:    $\sigma[j] = \sigma[j-1] + \sigma[j]$ 
9: end for
10: for  $i = n-1$  down to 0 do
11:    $j = A[i]$ 
12:    $S[\sigma[j]-1] = A[i]$ 
13:    $\sigma[j] = \sigma[j]-1$ 
14: end for
15: return  $S$ 
```

Distribution Sorting

The worst-case analysis for distribution sorting is:

$$\begin{aligned} C(n) &= \sum_{j=0}^{n_{\max}} 1 + \sum_{i=0}^{n-1} 1 + \sum_{j=0}^{n_{\max}} 1 + \sum_{i=0}^{n-1} 1 \\ &\quad \text{initialise freqs} \quad \text{compute freqs} \quad \text{compute cumulative freq} \quad \text{map values} \\ &= 2 \sum_{i=0}^{n-1} 1 + 2 \sum_{j=0}^{n_{\max}} 1 \\ &= 2O(n) + 2O(n_{\max}) \\ &\in O(n), \text{ if } n > n_{\max} \end{aligned}$$

The algorithm also uses an additional $O(n) + O(n_{\max})$ space.

Set ADT

Recall the definition of a set, where all the keys are unique. (Note this applies to maps also, where we have key->value pairs).

What data structures can we use to implement a Set ADT? E.g.,

- List
- Tree (balanced)
- Array

What are the average case time complexities of `INSERT`, `DELETE`, `MEMBER`?

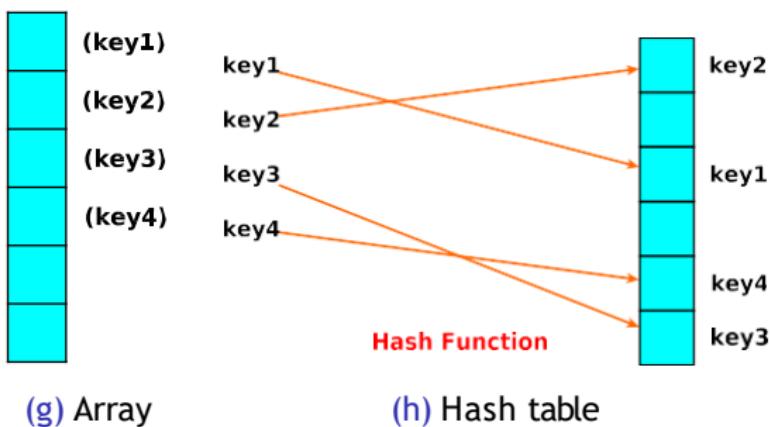
	List	Bal Tree	Array
<code>INSERT</code>	$O(1)$	$O(\log n)$	$O(n)$
<code>DELETE</code>	$O(n)$	$O(\log n)$	$O(n)$
<code>MEMBER</code>	$O(n)$	$O(\log n)$	$O(\log n)$

Is it possible to achieve better efficiency?

Hash Tables

If we have an array holding keys, and $O(1)$ method to map a key to a position in this array, we can improve on previous average case complexities.

This is the idea behind **hash tables**. Array is called hash table, mapping method called hash function.



Hash Tables

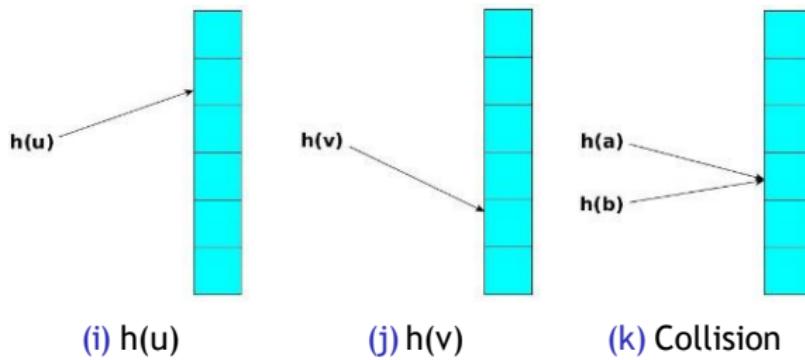
Formally:

- Let H be an array of size n storing the values. H is called the **hash table**.
- Let the set of possible keys be denoted by the **universe** U .
- Let the h denote a **hash function**, $h : U \rightarrow \{0, 1, \dots, n - 1\}$, which maps keys of U to array positions in H .
- E.g., $h(u)$ maps key u to a position in array H .

collisions:

- If two distinct keys u and v map to the same array position/index, i.e., $h(u) = h(v)$, we say that a **collision** has occurred.

Hash Tables



Hash Table Choices

- Hash function.
- Size of hash table.
- collision resolution.

Hash Functions

Ideal: Hash function that have no collisions.

- A **perfect** hash function is one that has no collisions.
- A “good” hash function has to satisfy two requirements which are often in tension:
 - ① A hash function needs to distribute keys among positions/cells of the hash table as uniformly as possible. (avoid collisions)
 - ② A hash function has to be easy and fast to compute.
- Example: $h(u) = u \bmod n$, produces a position index between 0 and $n - 1$.

Perfect Hash Functions (static set)

- If we have a **static set**, we can achieve perfect hashing and $O(1)$ average (and worst) case timing (given array is big enough).
- One approach to achieve this bound is to generate a **perfect hash function** for all of the elements *a priori*.
- Example 1 : Given $S_1 = \{10, 21, 32, 43, 54, 65, 76, 87\}$, then the function $h_1(x) = x \bmod 10$ is perfect.
- Example 2 : Given $S_2 = \{110, 210, 310, \dots, 810\}$, then the function $h_2(x) = (x - 10)/100$ is perfect.

Size of Hash Table

If table size (n) < number of keys (p), we are guaranteed to get collisions.

Solutions?

- Choose an initial $n \approx p$
- If dynamic set and p becomes bigger than n , increase size of table (n) and rehash all existing keys.

Collision Resolution

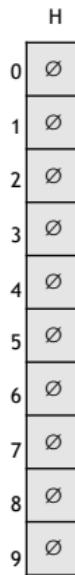
There are two major approaches to handle **collisions**:

- ① Separate Chaining Hashing.
- ② Open Address Hashing.

Separate Chaining Hashing - Overview

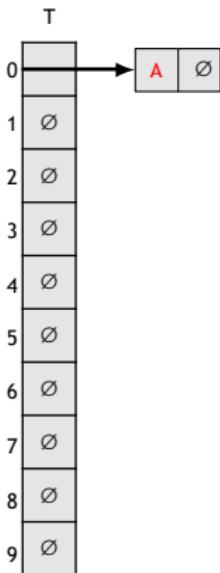
- Allow **more than one key** to be stored in a position of the hash table.
- Each position has a **linked list**, that stores all the keys hashed to that position.
- For completeness, if no key hashed to a position, set linked list pointer to null.

Separate Chaining Hashing - Example



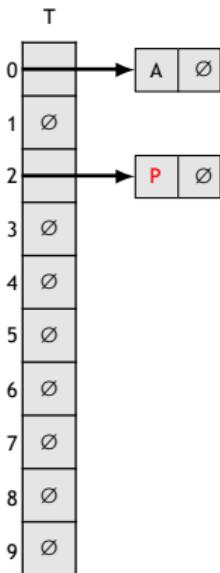
A chained hash table for the sequence $T_c = \text{“APJQDHBWM”}$.

Separate Chaining Hashing - Example



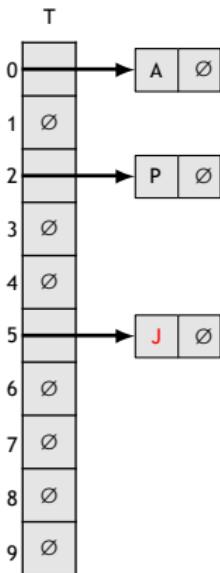
A chained hash table for the sequence $T_c = \text{“APJQDHBWM”}$.

Separate Chaining Hashing - Example



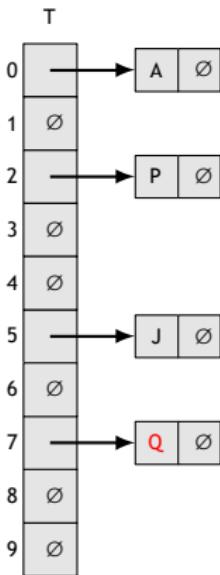
A chained hash table for the sequence $T_c = \text{"APJQDHBWM"}$.

Separate Chaining Hashing - Example



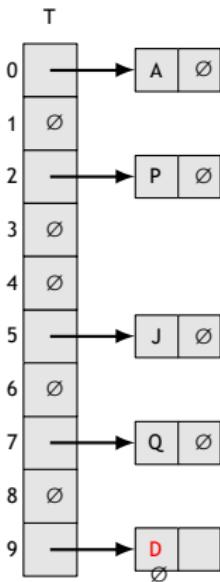
A chained hash table for the sequence $T_c = \text{"APJQDHBWM"}$.

Separate Chaining Hashing - Example



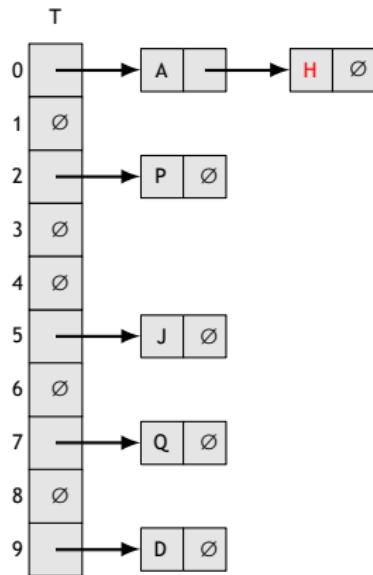
A chained hash table for the sequence $T_c = \text{“APJQDHBWM”}$.

Separate Chaining Hashing - Example



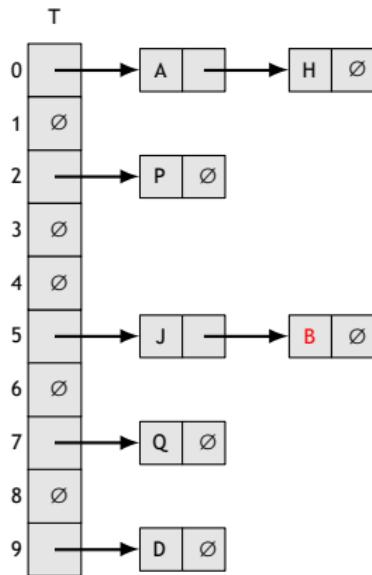
A chained hash table for the sequence $T_c = \text{“APJQDHBWM”}$.

Separate Chaining Hashing - Example



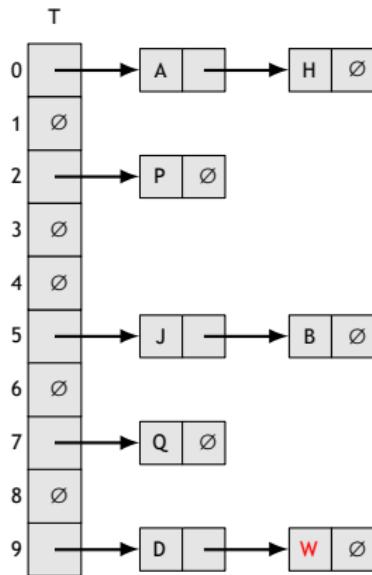
A chained hash table for the sequence $T_c = \text{"APJQDHBWM"}$.

Separate Chaining Hashing - Example



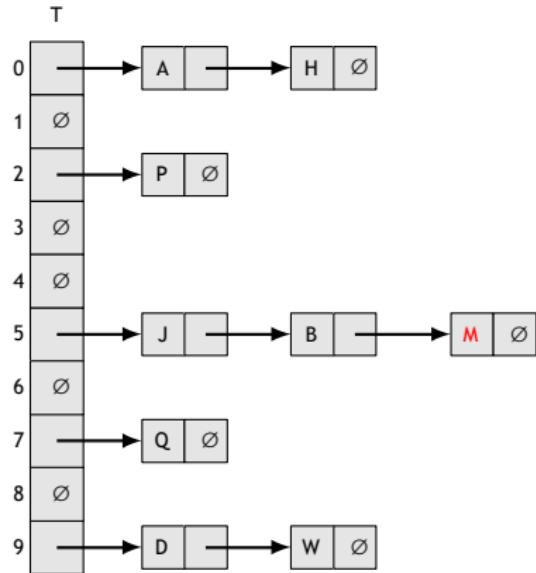
A chained hash table for the sequence $T_c = \text{"APJQDH}\textcolor{red}{B}WM"$.

Separate Chaining Hashing - Example



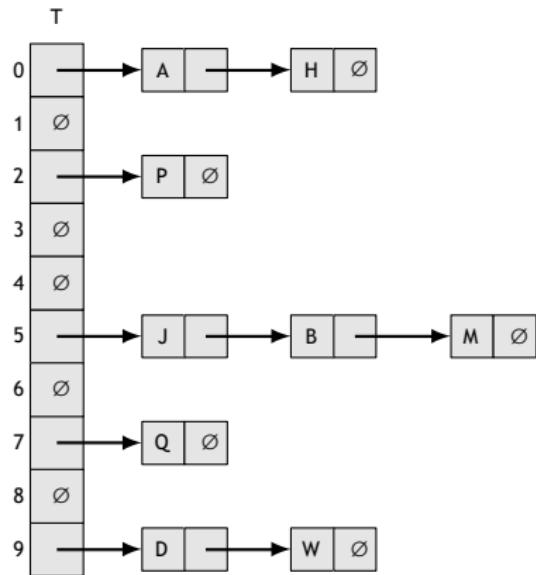
A chained hash table for the sequence $T_c = \text{"APJQDH}\textcolor{red}{B}WM"$.

Separate Chaining Hashing - Example



A chained hash table for the sequence $T_c = \text{"APJQDH}\textcolor{red}{B}WM\text{".}$

Separate Chaining Hashing - Example



A chained hash table for the sequence $T_c = \text{"APJQDHBWM"}$.

Separate Chaining Hashing - Cost

- **INSERT** in $O(1)$ worst-case by inserting a new element at the front or end of the list depending on the implementation.
- **DELETE** proportional to length of list.
- **MEMBER** proportional to length of list.
- Average case time is $O(1)$ for all operations, assuming simple uniform hashing (distribute keys uniformly) and constant load factor.

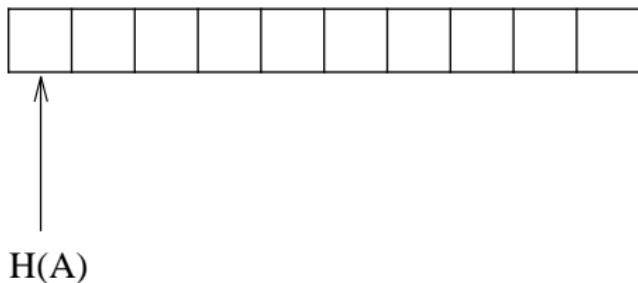
Separate Chaining Hashing - Analysis

- It is not unusual for $p > n$ in separate chaining hash tables in practice (p = number of keys, n = size of array).
- If the hash function distributes keys uniformly (simple uniform hashing), the average length of any linked list will be $a = p/n$. This ratio is called the **load factor**.
- The expected number of probes for a successful MEMBER is $\approx 1 + a/2$ (see Cormen et al. "Algorithms: Design and Analysis", p. 259).
- The expected number of probes for an unsuccessful MEMBER is a .
- The load a is typically kept small (and ideally it is 1).

Open Address Hashing - Overview

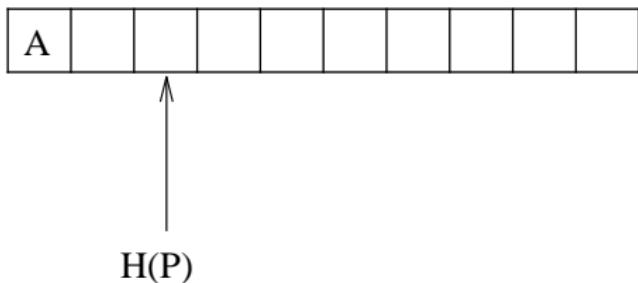
- Open address hashing is an alternative method to handle collisions.
- Each cell in the base array can store exactly one item.
- **Linear probing** - store the item in the next free cell.
- **Double hashing** - use a second hash function to compute the increment.

Open Address Hashing: Linear probing



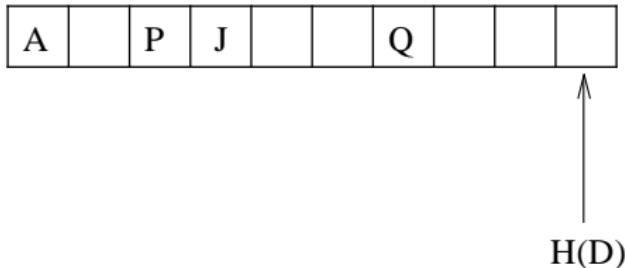
For the sequence $T_c = \text{“APJQDHBWM”}$, construct an open addressing hash table.

Open Address Hashing: Linear probing



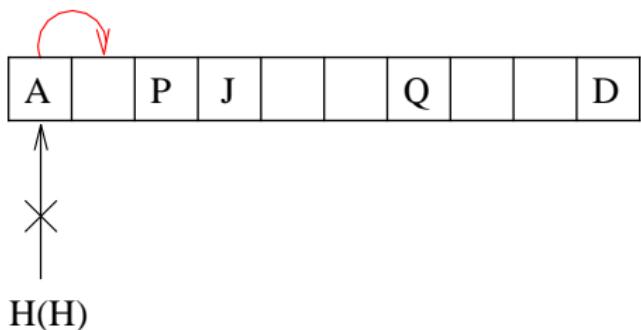
For the sequence $T_c = \text{“APJQDHBWM”}$, construct an open addressing hash table.

Open Address Hashing: Linear probing



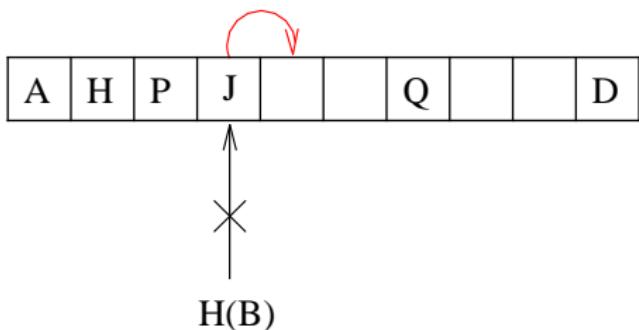
For the sequence $T_c = \text{“APJQDHBWM”}$, construct an open addressing hash table.

Open Address Hashing: Linear probing



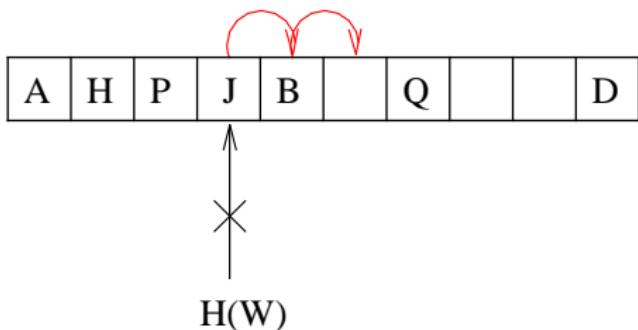
For the sequence $T_c = \text{“APJQDHBWM”}$, construct an open addressing hash table.

Open Address Hashing: Linear probing



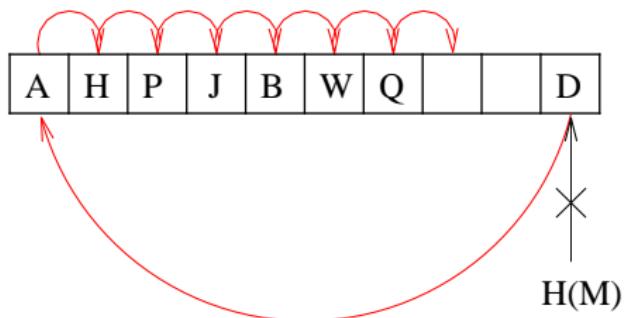
For the sequence $T_c = \text{“APJQDHBWM”}$, construct an open addressing hash table.

Open Address Hashing: Linear probing



For the sequence $T_c = \text{"APJQDHBWM"}$, construct an open addressing hash table.

Open Address Hashing: Linear probing



For the sequence $T_c = \text{"APJQDHBW M"}$, construct an open addressing hash table.

Open Address Hashing: Linear probing

A	H	P	J	B	W	Q	M		D
---	---	---	---	---	---	---	---	--	---

For the sequence $T_c = \text{"APJQDHBWM"}$, construct an open addressing hash table.

Linear probing - Analysis

- Does not work if $p > n$. (If p becomes bigger than n , we can resize and rehash.)
- Number of probes for the three key set operations depend on the load factor $a = p/n$.
- For linear probing, a successful search is $\approx \frac{1}{2}(1 + 1/(1 - a))$ probes and an unsuccessful search is $\approx \frac{1}{2}(1 + 1/(1 - a)^2)$ probes.
- It is much more difficult to derive these bounds than in the separate chaining hash table.

Linear probing - Analysis

As the table fills ($a \rightarrow 1$), the number of probes necessary increases dramatically.

a	$\frac{1}{2}(1 + \frac{1}{1-a})$	$\frac{1}{2}(1 + \frac{1}{(1-a)^2})$
50%	1.5	2.5
75%	2.5	8.5
90%	5.5	50.5

Open Address Hashing - Double hashing

Double hashing uses **two** hash functions:

- one is to determine the initial position (same as linear probing)
- the other to determine the size of interval to step (linear probing always has interval of 1).

Given two (usually independent universal) hashing functions h_1 and h_2 :

- We first do: $h_1(u) \bmod n$
- If clash then do: $h_1(u) + 1 \cdot h_2(u) \bmod n$
- If clash again then do: $h_1(u) + 2 \cdot h_2(u) \bmod n$
- etc

E.g. $h_1(a) = 4$, but $H[4]$ is occupied. Next position to check is not 5.

Let $h_2(a) = 3$, then next position to check is $h_1(u) + h_2(u) = 7$.

If h_2 is chosen well, we can avoid clustering effects, which can lead to faster collision resolution.

Double hashing - Comments

- Difficult to analyse the complexity of successful and unsuccessful searches.
- Empirically shown double hashing performs better than linear probing, especially when table is more full.

Other Applications of Hashing

Checksums and signatures:

<https://www.youtube.com/watch?v=b4b8ktEV4Bg>

Locality sensitive hashing:

https://www.youtube.com/watch?v=Ha7_Vf2eZvQ

Summary

The two types of space-time tradeoffs discussed:

- **input enhancement**
 - preprocess input and store relevant information that speeds up solving the problem.
 - e.g., distribution sorting
- **prestructuring**
 - construct data structures (space) that have faster or more flexible access to data.
 - e.g., hashing