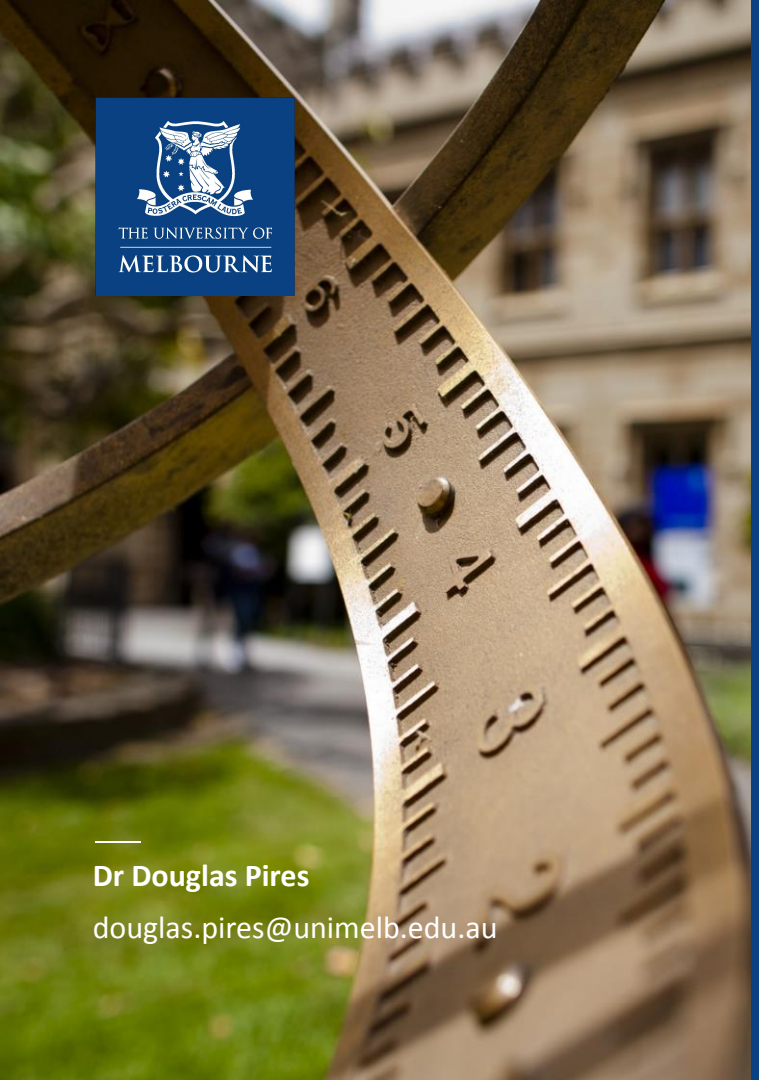
THE UNIVERSITY OF MELBOURNE

Algorithms and Complexity

COMP90038 - Lecture 23

Dr Douglas Pires
douglas.pires@unimelb.edu.au



Revision

Dr Douglas Pires

douglas.pires@unimelb.edu.au

Looking back

Algorithms

- **Data structures:** Arrays, Stacks, Queues, Graphs (directed, undirected, weighted), Binary trees, Binary search trees, AVL trees, 2-3 trees, Heaps (priority queues), Tries/Huffman tree.
- **Sorting algorithms:** Selection sort, insertion sort, (shellsort), mergesort, quicksort, heapsort, sorting by counting.
- **Search algorithms:** Sequential search, Binary search, Binary search on AVL trees and 2–3 trees, Hashing. k-th smallest elements.
- **Graph algorithms:** DFS, BFS, tree traversals, topological sort, Warshall, Floyd, Prim, Dijkstra.

Looking back

Complexity and Algorithm Design

- **Algorithm analysis:** Asymptotics, the Master Theorem.
- **Algorithmic techniques:** Brute force, decrease-and-conquer, divide-and-conquer, transform-and-conquer (presorting, representation change), dynamic programming, greedy methods, time/space tradeoffs.
- **String searching algorithms:** Brute-force string search, Horspool, Rabin-Karp.

Preparing for the exam

- **On Canvas - Exam Info**
  - Sample example question using the 'traditional 3-hour written exam format'.
  - You can expect to see similar questions on this years exam.

- **List of Examinable Materials**
  - As a general rule, everything covered in lectures is examinable

- Go over the **tutorial/quiz** questions again
- Go over the **assignment** questions again

Asymptotics

Functions Often Met in Algorithm Classification

1: Running time independent of input.

$\log n$: Typical for "divide and conquer" solutions, for example, lookup in a balanced search tree.

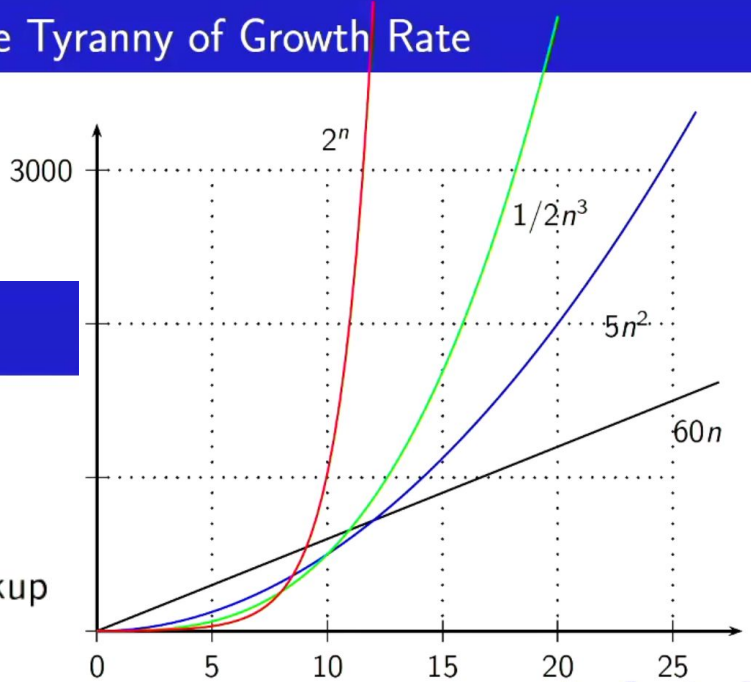
Linear: When each input element must be processed once.

$n \log n$: Each input element processed once and processing involves other elements too, for example, sorting.

n^2, n^3: Quadratic, cubic. Processing all pairs (triples) of elements.

2^n: Exponential. Processing all subsets of elements.

The Tyranny of Growth Rate



Complexity & asymptotic behaviour

Big-Oh Notation

$O(g(n))$ denotes the set of functions that grow no faster than g, asymptotically.

We write

$$t(n) \in O(g(n))$$

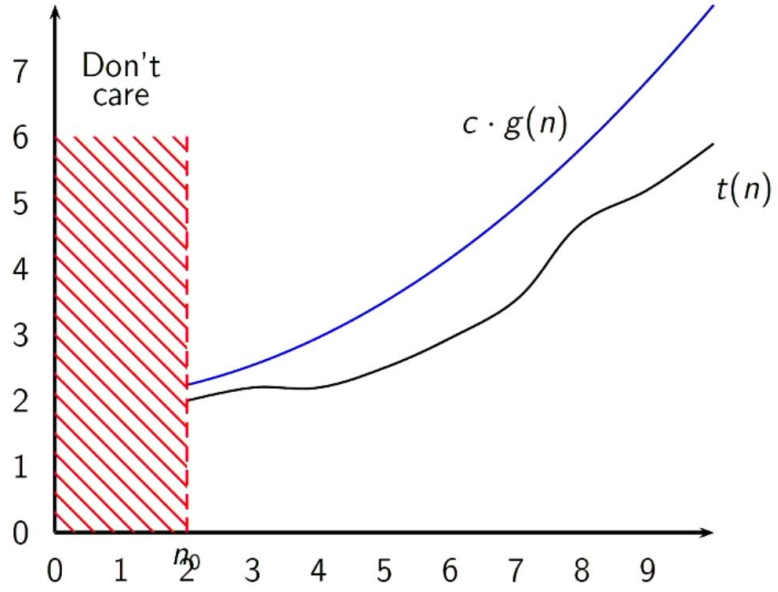
when, for some c and n_0,

$$n > n_0 \Rightarrow t(n) < c \cdot g(n)$$

For example,

$$1 + 2 + \cdots + n \in O(n^2)$$

Big-Oh: What $t(n) \in O(g(n))$ Means

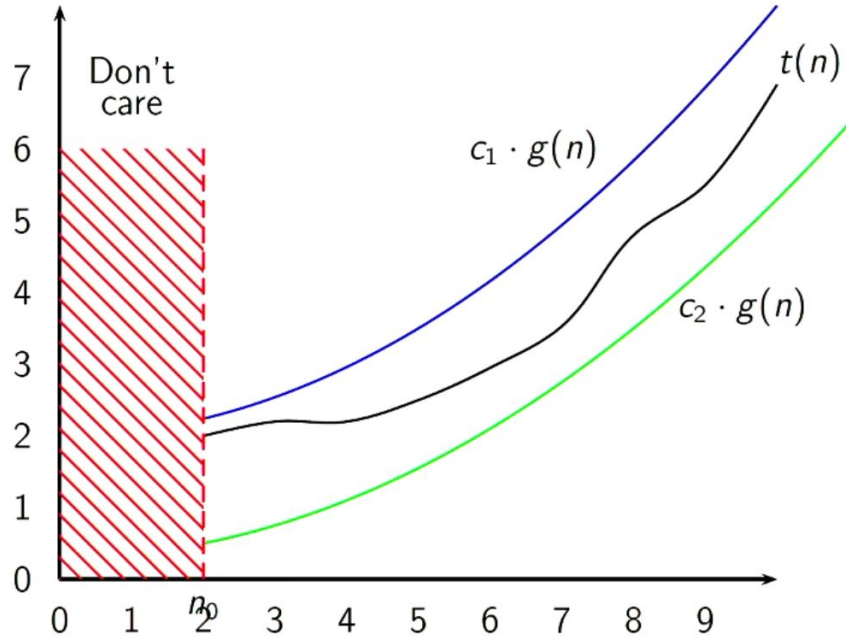


Upper bound

Complexity & asymptotic behaviour

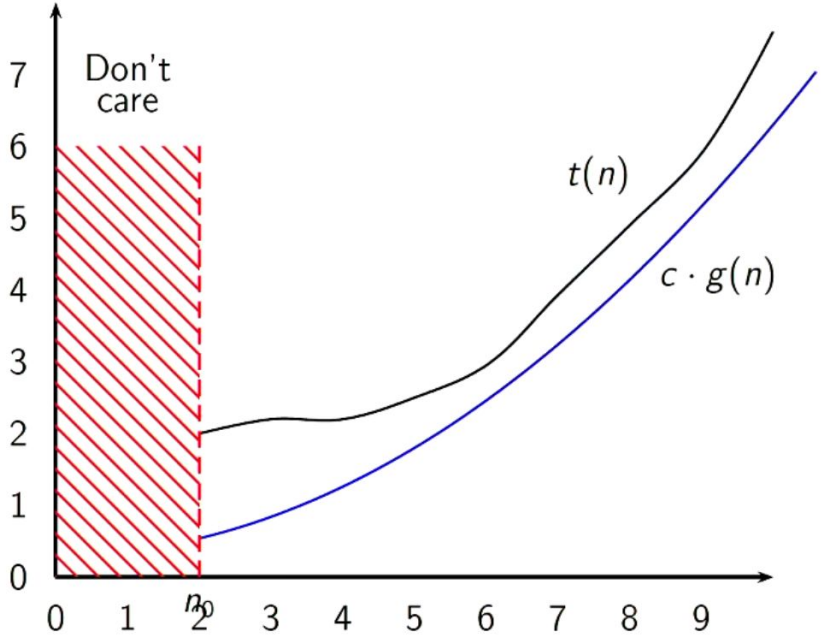
Exact order of growth

Big-Theta: What $t(n) \in \Theta(g(n))$ Means



Lower bound

Big-Omega: What $t(n) \in \Omega(g(n))$ Means



Given an algorithm…

```
MyAlgoritm (A, B)
1  n = A.rows
2  let C be a new n × n matrix
3  for i = 1 to n
4      for j = 1 to n
5          c_ij = 0
6          for k = 1 to n
7              c_ij = c_ij + a_ik · b_kj
8  return C
```

- Tell me the time complexity
- What is the algorithm doing?

For recursive algorithms

- Probably no question asking you to directly do this.
- But you might need to it to answer the question.

A Second Example: Binary Search in Sorted Array

function BinSearch($A[], lo, hi, key$)
  if $lo > hi$ **then return** -1
  $mid \leftarrow lo + (hi - lo)/2$
  if $A[mid] = key$ **then return** mid
  else
    if $A[mid] > key$ **then**
      return BinSearch($A, lo, mid - 1, key$)
    else return BinSearch($A, mid + 1, hi, key$)

The basic operation is the key comparison. The cost, recursively, in the worst case:

$$\begin{aligned} C(0) &= 0 \\ C(n) &= C(n/2) + 1 \quad \text{for } n > 0 \end{aligned}$$

Telescoping

A smoothness rule allows us to assume that n is a power of 2. The recursive equation was:

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

Use the fact $C(n/2) = C(n/4) + 1$ to expand, and keep going:

$$\begin{aligned} C(n) &= C(n/2) + 1 \\ &= [C(n/4) + 1] + 1 \\ &= [[C(n/8) + 1] + 1] + 1 \\ &\vdots \\ &= \underbrace{[[\ldots[[C(0) + 1] + 1] + \cdots + 1] + 1]}_{1+\log_2 n \text{ times}} \end{aligned}$$

Hence $C(n) = \Theta(\log n)$.

Master Theorem

The Master Theorem

(A proof is in Levitin's Appendix B.)

For integer constants $a \geq 1$ and $b > 1$, and function f with $f(n) \in \Theta(n^d), d \geq 0$, the recurrence

$$T(n) = aT(n/b) + f(n)$$

(with $T(1) = c$) has solutions, and

$$T(n) = \begin{cases} \Theta(n^d) & \text{if } a < b^d \\ \Theta(n^d \log n) & \text{if } a = b^d \\ \Theta(n^{\log_b a}) & \text{if } a > b^d \end{cases}$$

Note that we also allow a to be greater than b.

- What is the **complexity** of a given algorithm?
- Use the master theorem
- Do telescoping

Sorting algorithms

Properties of Selection Sort

While running time is quadratic, selection sort makes only about n exchanges.

So: A good algorithm for sorting small collections of large records.

In-place?

Stable?

Input-insensitive?

- Best case for Insertion?
- Worst case for quick?
- What if the array is almost sorted?

String matching

- How many comparisons for a given input?
- What about for Horspool's?

Brute Force String Matching

Pattern p: A string of m characters to search for.

Text t: A long string of n characters to search in.

```
for i ← 0 to n − m do
    j ← 0
    while j < m and p[j] = t[i + j] do
        j ← j + 1
    if j = m then
        return i
return −1
```

Horspool's String Search Algorithm

```
S T R I N G S E A R C H E X A M P
E X A M
        E X A M
```

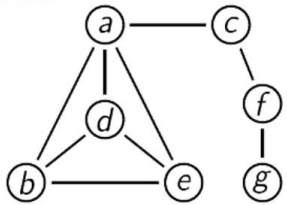
Here we can slide the pattern 3 positions, because the last occurrence of E in the pattern is its first position.

```
S T R I N G S E A R C H E X A M P
E X A M
        E X A M
            E X A M
                E X A M
                  E X A M
```

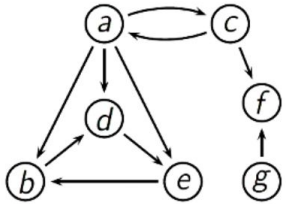
Graph concepts

Graph Concepts

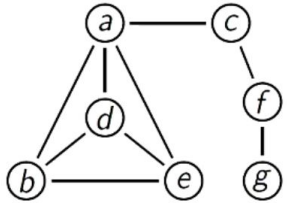
Undirected:



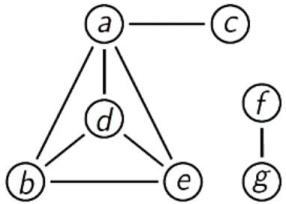
Directed:



Connected:

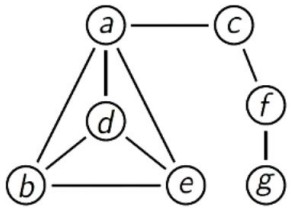


Not connected, two components:

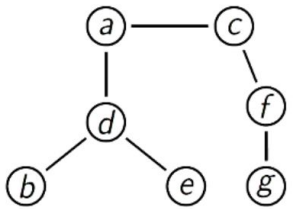


More Graph Concepts

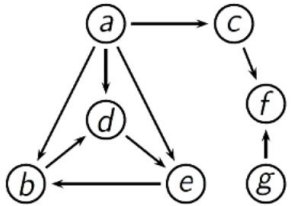
Cyclic:



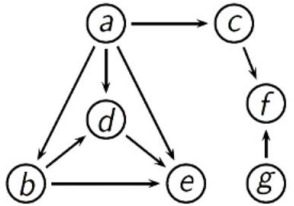
Acyclic (actually, a tree):



Directed cyclic:

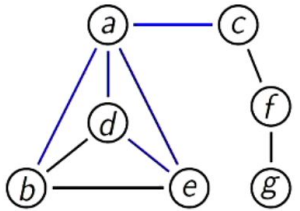


Directed acyclic (a dag):



Graph concepts

More Graph Concepts: Paths and Cycles



Path b, a, d, e, a, c shown in blue

A path in $\langle V, E \rangle$ is a sequence of nodes $v_0, v_1, \ldots, v_k$ from V, so that $(v_i, v_{i+1}) \in E$.

The path $v_0, v_1, \ldots, v_k$ has length k.

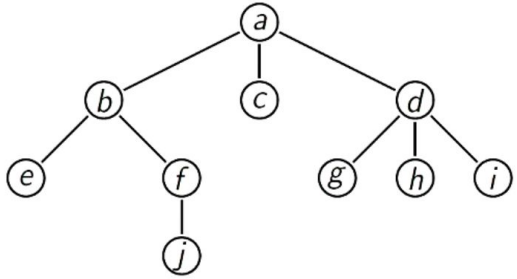
A simple path is one that has no repeated nodes.

A cycle is a simple path, except that $v_0 = v_k$, that is, the last node is the same as the first node.

Rooted Trees

A (free) tree is a connected acyclic graph.

A rooted tree is a tree with one node identified as special. Every other node is reachable from the node.

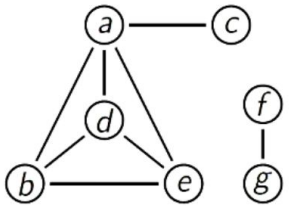


When the root is removed, a set of rooted (sub-)trees remain.

We should draw the rooted tree as a directed graph, but usually we instead rely on the layout: "parents" sit higher than "children".

Graph representations

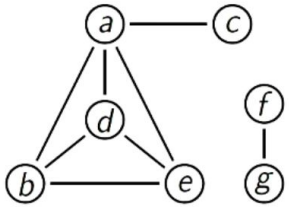
Graph Representations, Undirected Graphs



	a	b	c	d	e	f	g
a	0	1	1	1	1	0	0
b	1	0	0	1	1	0	0
c	1	0	0	0	0	0	0
d	1	1	0	0	1	0	0
e	1	1	0	1	0	0	0
f	0	0	0	0	0	0	1
g	0	0	0	0	0	1	0

The adjacency matrix for the graph.

Graph Representations, Undirected Graphs



a	$\rightarrow b \rightarrow c \rightarrow d \rightarrow e$
b	$\rightarrow a \rightarrow d \rightarrow e$
c	$\rightarrow a$
d	$\rightarrow a \rightarrow b \rightarrow e$
e	$\rightarrow a \rightarrow b \rightarrow d$
f	$\rightarrow g$
g	$\rightarrow f$

The adjacency list representation.

(Assuming lists are kept in sorted order.)

- When to choose one or the other?
- What about directed graphs?

Graph traversals

- Show the node visitation order
- Use or modify them to other purposes
- Tree-traversals as well
  - In-order, pre-order, post-order, level-order

Depth-First Search: The Algorithm

```
function DFS(⟨V, E⟩)
    mark each node in V with 0
    count ← 0
    for each v in V do
        if v is marked 0 then
            DfsExplore(v)

function DfsExplore(v)
    count ← count + 1
    mark v with count
    for each edge (v, w) do                ▷ w is v's neighbour
        if w is marked with 0 then
            DfsExplore(w)
```

This works both for directed and undirected graphs.

Breadth-First Search: The Algorithm

```
function BFS(⟨V, E⟩)
    mark each node in V with 0
    count ← 0, init(queue)                 ▷ create an empty queue
    for each v in V do
        if v is marked 0 then
            count ← count + 1
            mark v with count
            inject(queue, v)               ▷ queue containing just v
            while queue is non-empty do
                u ← eject(queue)           ▷ dequeues u
                for each edge (u, w) adjacent to u do
                    if w is marked with 0 then
                        count ← count + 1
                        mark w with count
                        inject(queue, w)   ▷ enqueues w
```

Partitioning

- Lead to quicker ways to do sorting
- **Quicksort** (hoare partitioning)
  - How the choice of pivot influences performance?
- **Divide-and-conquer**

Finding the kth Smallest Element

Here is how we can use partitioning to find the kth smallest element.

```
function QuickSelect(A[.], lo, hi, k)
    s ← LomutoPartition(A, lo, hi)
    if s − lo = k − 1 then
        return A[s]
    else
        if s − lo > k − 1 then
            QuickSelect(A, lo, s − 1, k)
        else
            QuickSelect(A, s + 1, hi, (k − 1) − (s − lo))
```

Hoare Partitioning

This is the standard way of doing partitioning for quicksort:

```
function Partition(A[lo..hi])
    p ← A[lo]; i ← lo; j ← hi
    repeat
        while i < hi and A[i] ≤ p do i ← i + 1
        while j ≥ lo and A[j] > p do j ← j − 1
        swap(A[i], A[j])
    until i ≥ j
    swap(A[i], A[j]) — undo the last swap
    swap(A[lo], A[j]) — bring pivot to its correct position
    return j
```

Divide-and-conquer: sorting

Mergesort

Perhaps the most obvious application of divide-and-conquer:

To sort an array (or a list), cut it into two halves, sort each half, and merge the two results.

function MERGESORT($A[0..n-1]$)
  if $n > 1$ **then**
    copy $A[0..\lfloor n/2 \rfloor - 1]$ to $B[0..\lfloor n/2 \rfloor - 1]$
    copy $A[\lfloor n/2 \rfloor..n-1]$ to $C[0..\lceil n/2 \rceil - 1]$
    MERGESORT($B[0..\lfloor n/2 \rfloor - 1]$)
    MERGESORT($C[0..\lceil n/2 \rceil - 1]$)
    MERGE(B, C, A)

Quicksort

Very short and elegant:

function QUICKSORT($A[lo..hi]$)
  if $lo < hi$ **then**
    $s \leftarrow$ PARTITION($A[lo..hi]$)
    QUICKSORT($A[lo..s-1]$)
    QUICKSORT($A[s+1..hi]$)

Binary trees

- Using `height()` for other purposes

Binary Tree Concepts

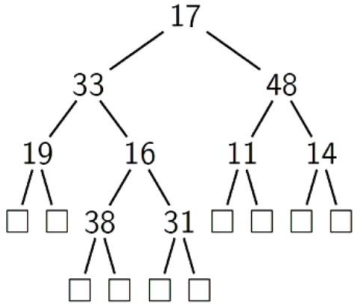
A non-empty tree T has a root T_{root}, a left subtree T_{left}, and a right subtree T_{right}.

Recursion is the natural way of calculating the height:

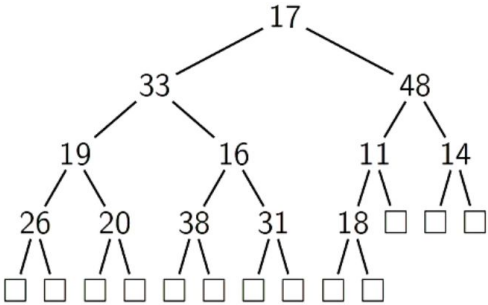
```
function HEIGHT(T)
    if T is empty then
        return −1
    else
        return max(HEIGHT(T_left), HEIGHT(T_right)) + 1
```

Binary Tree Concepts

Special trees have their external nodes □ only at level h and $h+1$ for some h:



A full binary tree: Each node has 0 or 2 children.



A complete tree: Each level filled left to right.

- Important for balanced trees
- Tree-traversals and recursive definitions of trees:
  - Left- and right- subtrees
  - For binary trees

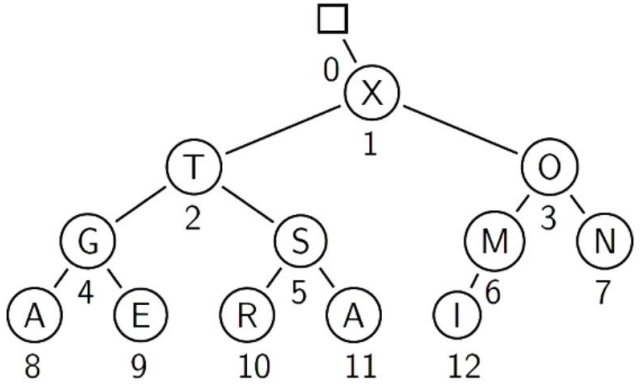
Heaps & Heapsort

- Is this array a heap?
- Operations:
  - Insert, Delete, Build

Heaps as Arrays

This way, the heap condition is very simple:

For all $i \in \{0, 1, \ldots, n\}$, we must have $H[i] \leq H[i/2]$.



H:

	X	T	O	G	S	M	N	A	E	R	A	I
0	1	2	3	4	5	6	7	8	9	10	11	12

Heapsort

Heapsort is a $\Theta(n \log n)$ sorting algorithm, based on the idea from this exercise.

Given unsorted array $H[1..n]$:

Step 1 Turn H into a heap.

Step 2 Apply the eject operation $n - 1$ times.

- **Priority queues**
  - Max-heaps
  - Min-heaps

Transform-and-conquer

- **Instance simplification**
  - *E.g.*: pre-sorting
- **Representation change**
  - *E.g.*: heaps
- **Problem reduction**

Uniqueness Checking, with Presorting

Sorting makes the problem easier:

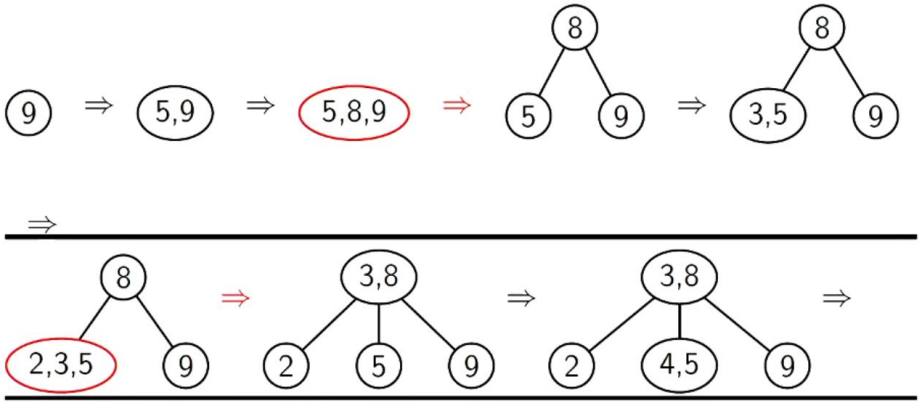
```
SORT(A[0..n − 1])
for i ← 0 to n − 2 do
    if A[i] = A[i + 1] then
        return False
return True
```

What is the complexity of this?

AVL and 2-3 Trees

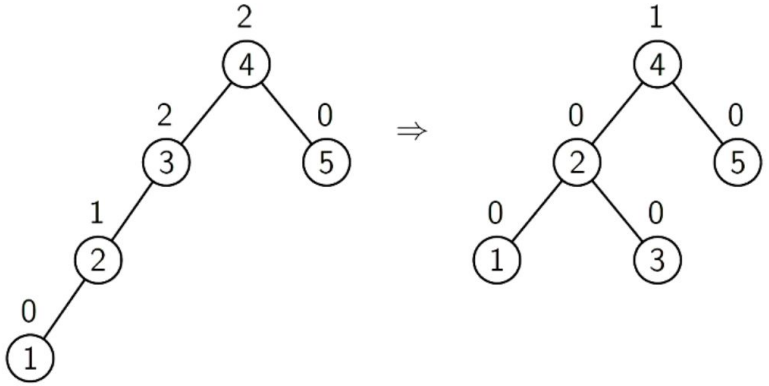
- **Rotations**
- Where/Which **node** to start balancing?

Example: Build a 2–3 Tree from 9, 5, 8, 3, 2, 4, 7



AVL Trees: Where to Perform the Rotation

Along an unbalanced path, we may have several nodes with balance factor 2 (or -2):



It is always the lowest unbalanced subtree that is re-balanced.

- **Inserting** items in a AVL or 2-3 tree
- Why use them?
- What can we say about their heights?

- Regular BST (insert items in descending order)

Space/Time tradeoffs

- Sorting **without key comparisons**
- **Conditions**
- **Complexity**

Sorting by Counting

We can now create a sorted array $S[1..n]$ of the items by simply slotting items into pre-determined slots in S (a third linear scan).

6 3 3 8 1 0 8 7 9 2 5 3 5 3 1 8 7 6 5 1 2 1 5 3

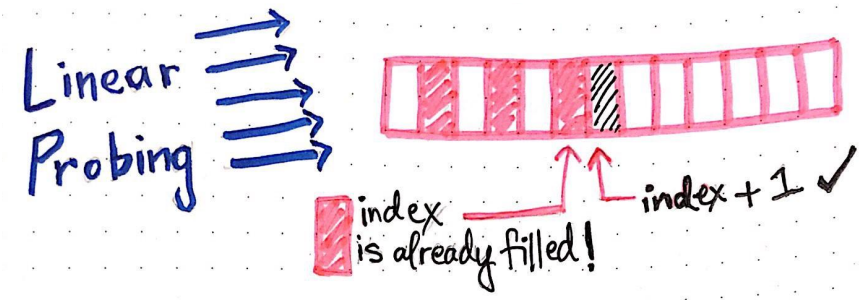
key	0	1	2	3	4	5	6	7	8	9
Occ	1	5	7	12	12	16	18	20	23	24

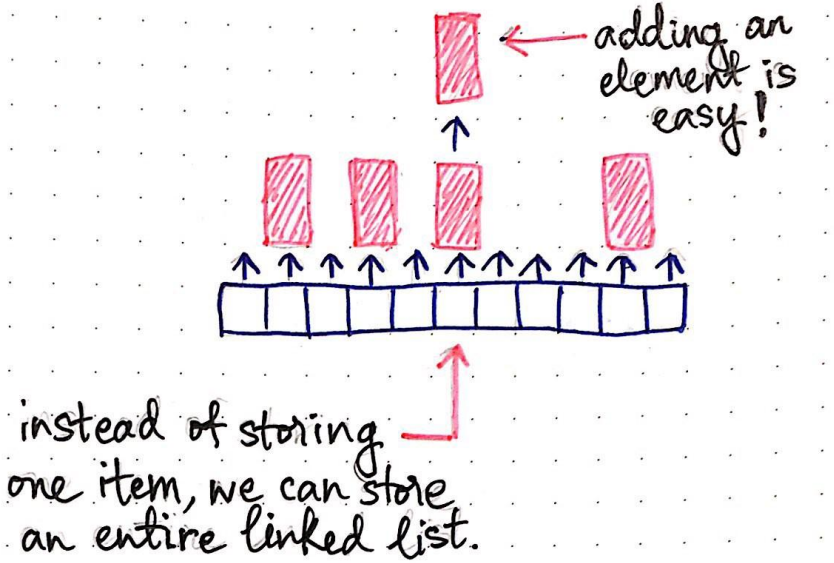
Place the first record (with key 6) in $S[18]$ and decrement $Occ[6]$ (so that the next '6' will go into slot 17), and so on.

for $i \leftarrow 1$ **to** n **do**
    $S[Occ[A[i]]] \leftarrow A[i]$
    $Occ[A[i]] \leftarrow Occ[A[i]] - 1$

Hash Tables

- **Collision handling**
  - Linear Probing
  - Separate chaining
  - Double-hashing
- **Load factor** (α)
- **Pros** and **cons**
  - Clustering?
  - Deletions?
- **Applications**
  - Rabin-Karp String Search





Dynamic Programming

- **Overlapping subproblems**
  - Recurrence relation (and base cases)
  - Build a table storing partial solutions for smaller sub-problems (bottom-up)

Algorithm

for $i \leftarrow 0$ to n **do**
  $K[i, 0] \leftarrow 0$
for $j \leftarrow 1$ to W **do**
  $K[0, j] \leftarrow 0$

Base cases

for $i \leftarrow 1$ to n **do** Items
  for $j \leftarrow 1$ to W **do** Weights
    if $j < w_i$ **then**
      $K[i, j] \leftarrow K[i-1, j]$ Line above
    else
      $K[i, j] \leftarrow max(K[i-1, j], K[i-1, j-w_i] + v_i)$
return $K[n, W]$

Example 2: The Knapsack Problem

Now it is easy to express the solution recursively:

$$K(i, w) = 0 \text{ if } i = 0 \text{ or } w = 0$$

Base case

Initialise a table

Otherwise:

Does not include i

Includes i

$$K(i, w) = \begin{cases} max(K(i-1, w), K(i-1, w-w_i) + v_i) & \text{if } w \geq w_i \\ K(i-1, w) & \text{if } w < w_i \end{cases}$$

Doesn't fit

Dynamic Programming on Graphs

- **Warshall's algorithm**
  - Transitive closure

Warshall's Algorithm

This leads to a better version of Warshall's algorithm:

```
for k ← 1 to n do
    for i ← 1 to n do
        if A[i,k] then
            for j ← 1 to n do
                if A[k,j] then
                    A[i,j] ← 1
```

Node k as a stepping stone

If each row in the matrix is represented as a bit-string, the innermost for loop (and j) can be gotten rid of—instead of iterating, just apply the "bitwise or" of row k to row i.

Floyd's Algorithm

If G's weight matrix is W then we can express the recurrence relation for minimal distances as follows:

$$\begin{aligned} D_{ij}^{0} &= W[i,j] \\ D_{ij}^{k} &= min(D_{ij}^{k-1}, D_{ik}^{k-1} + D_{kj}^{k-1}) \end{aligned}$$

And then the algorithm follows easily:

```
function FLOYD(W[1..n, 1..n])
    D ← W
    for k ← 1 to n do
        for i ← 1 to n do
            for j ← 1 to n do
                D[i,j] ← min(D[i,j], D[i,k] + D[k,j])
    return D
```

- **Floyd's algorithm**
  - All-pairs shortest pairs

Greedy Algorithms on Graphs

- **Greedy choice**
  - Locally best
  - Feasible
  - Irrevocable

Prim's Algorithm

function PRIM($\langle V, E \rangle$)
  for each $v \in V$ **do**
    $cost[v] \leftarrow \infty$
    $prev[v] \leftarrow nil$
  Costs & parents
  pick initial node v_0
  $cost[v_0] \leftarrow 0$
  $Q \leftarrow$ INITPRIORITYQUEUE(V) ▷ priorities are cost values
  MinHeap
  while Q is non-empty **do**
    $u \leftarrow$ EJECTMIN(Q)
    for each $(u, w) \in E$ **do**
      if w in Q and $weight(u, w) < cost[w]$ **then**
        $cost[w] \leftarrow weight(u, w)$
        $prev[w] \leftarrow u$
        Update costs in the heap
        UPDATE($Q, w, cost[w]$) ▷ rearranges priority queue

Dijkstra's Algorithm

function DIJKSTRA($\langle V, E \rangle, v_0$)
  for each $v \in V$ **do**
    $dist[v] \leftarrow \infty$
    $prev[v] \leftarrow nil$
  Accumulated distances instead of cost
  $dist[v_0] \leftarrow 0$
  $Q \leftarrow$ INITPRIORITYQUEUE(V) ▷ priorities are distances
  while Q is non-empty **do**
    $u \leftarrow$ EJECTMIN(Q)
    Greedy choice
    for each $(u, w) \in E$ **do**
      if w in Q and $dist[u] + weight(u, w) < dist[w]$ **then**
        $dist[w] \leftarrow dist[u] + weight(u, w)$
        $prev[w] \leftarrow u$
        UPDATE($Q, w, dist[w]$) ▷ rearranges priority queue

- **Time complexity**
- **Priority queues**
  - What happens if we change it?
- Negative weights or negative cycles

Huffman Encoding for Data Compression

- **Data compression**
  - Run-length encoding
  - Variable-length encoding

Tries for Variable-Length Encoding

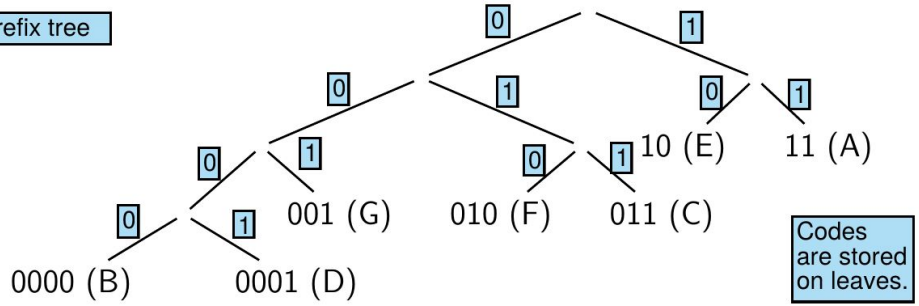
What is the optimal structure of this trie?
- Minimise number of bits to encode chars

A trie is a binary tree for search applications.

To search for a key we look at individual bits of a key and descend to the left whenever a bit is 0, to the right whenever it is 1.

Using a trie to determine codes means that no code will be the prefix of another!

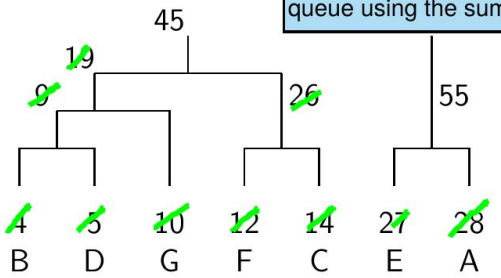
Prefix tree



Codes are stored on leaves.

Huffman Trees

Algorithmically
- Treat table as a priority queue
- Eject the two smallest elements
- Join them in a binary tree
- Inject an element back in the priority queue using the sum of their weights



- Use **less** bits to encode **more** frequent characters
- No shared **prefixes**

THE UNIVERSITY OF MELBOURNE

Algorithms and Complexity

COMP90038 - Lecture 23

Dr Douglas Pires

douglas.pires@unimelb.edu.au