C01 Recursion

Recursive data structures
Recursive algorithms

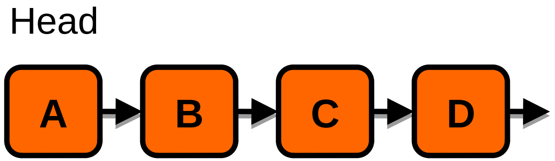
ANU – School of Computing – Structured Programming 1110 / 1140 / 6710

PAUL NOTH

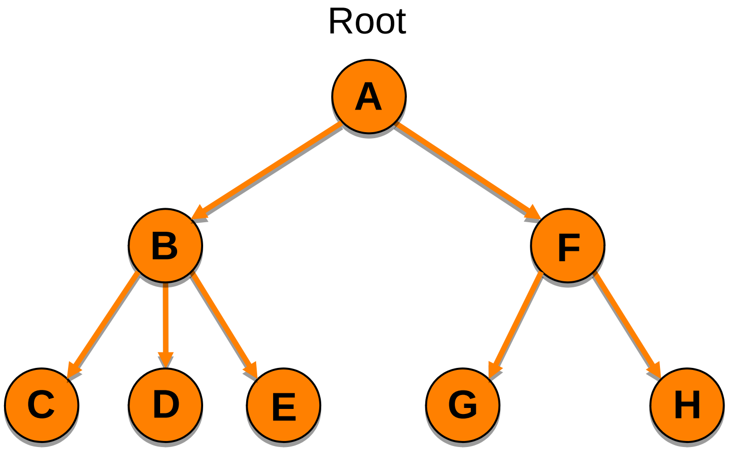
HAARLEM HOLLAND
Droste®
HOLLAND
Droste
Droste
cacao
cacao
cocoa
NET WT 250g ℮ / 8.8 oz.
Kosher Ⓤ

Recursive Data Structures

A recursive data structure is comprised of components that reference other components of the same type.



Linked list



Tree

Recursive Algorithms

A recursive algorithm reduces a computational problem to one or more smaller instances of the same problem, and composes the solution from their solutions.

A recursive algorithm is comprised of:

- Base case(s) that terminate the recursion
- Recursive call(s) that reduces towards the base case(s)

Example: Fibonacci Sequence

fib(0) = 0 (base case)

fib(1) = 1 (base case)

fib(n) = fib(n-1) + fib(n-2) (for n ≥ 2)

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377…

Example: Binary Search

Ordered list and a target value to find.

```
[1, 4, 5, 7, 9, 11, 15, 20, 25]  find 11
[1, 4, 5, 7, 9, 11, 15, 20, 25]  9 > 11?   right half
            [9, 11, 15, 20, 25]  15 > 11?  left half
            [9, 11]              9 > 11?   right half
               [11]
```

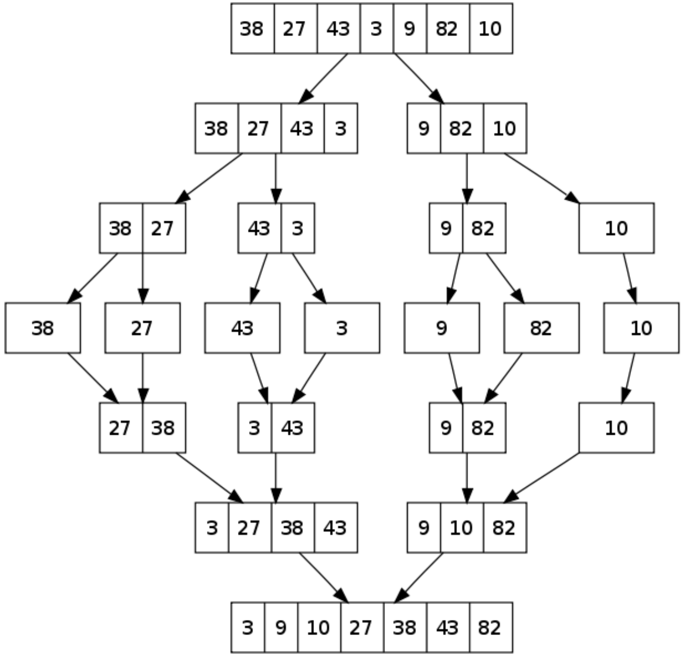
How does this compare to linear search?

What might the base case(s) be?

Example: Mergesort (von Neumann, 1945)

Sort a list

- List of size 1 (base case)
  - Already sorted
- List of size > 1
  - Split into two sub lists
  - Sort each sub list (recursion)
  - Merge the two sorted sub lists into one sorted list (by iteratively picking the lower of the two least elements)



Animation: Visualizing Algorithms, Mike Bostock, bost.ocks.org/mike/algorithms

Recursion

- A recursive method (function) calls itself: this works because of the *call stack*.
- A recursive method can always be rewritten into an iterative one and vice-versa (consequence of *Church-Turing thesis*).
- When to use **recursion** vs when to use **iteration** (`for` and `while` loops)?
  - The problem at hand might be more naturally written and read in one form (once you understand recursion!).
  - Converting between approaches not always straightforward.

Recursion and Java

- Overhead of calling calling methods often higher than iterating
- *Stack overflow* on larger problems
- Compilers in many other languages perform *tail-call elimination* for certain forms of recursion – Java doesn’t
- More functional languages (scheme, lisp, ocaml, haskell, f#, scala) make recursion more convenient
- Situations where recursion is *best* are more limited in Java – but important cases still exist!

C02 Computational Complexity

Time and Space Complexity
Algorithm vs Problem Complexity
Big O Notation
Examples

ANU – School of Computing – Structured Programming 1110 / 1140 / 6710

Computational Complexity

Key computational resources:

- **Time**
- **Space**
- Energy, communications, I/O, samples...

Computational complexity is the study of how problem size affects resource consumption (how it *scales*). Distinguish:

- **Algorithm Complexity**: for a given algorithm / implementation
- **Problem Complexity**: for *any* algorithm that solves the problem
  - Inherit difficulty of the problem (Computational Complexity Theory)

Algorithm Complexity

- Identify ***n***, the number that characterizes the problem size.
  - Number of pixels on screen
  - Number of elements to be sorted
  - etc.
- Study the algorithm to determine how resource consumption changes as a function of ***n***.
- The *content* of the input, not just its size, can be important. Can study:
  - **Worst** case (the worst input of size ***n***)
  - **Best** case (the best input of size ***n***)
  - **Average** case (average of distribution of inputs of size ***n***)
  - **Amortized** analysis (amortized cost over a sequence of ***n*** typical operations)
    - Useful for an operation with state that occasionally has an expensive step

Big O Notation

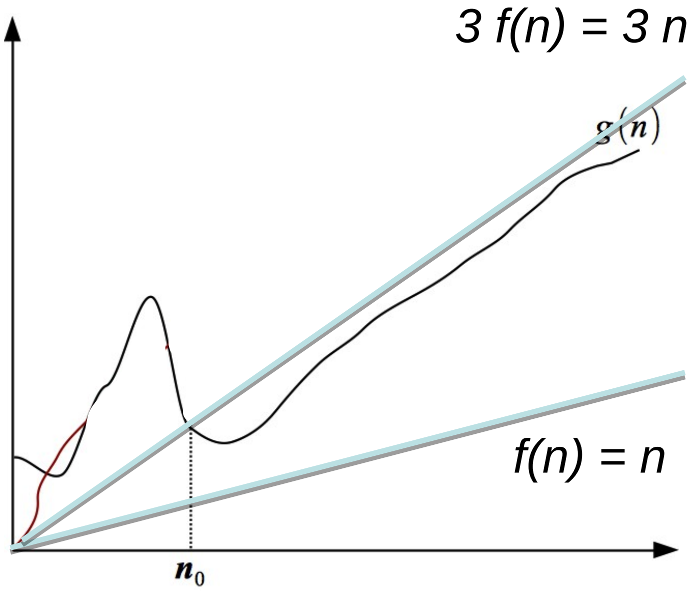
Suppose we have a problem of size *n* that takes *g(n)* time to execute in the average case.

We say:

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

iff there exists constants $c > 0$ and $n_0 > 0$ such that for all $n > n_0$:

$g(n) \leq c \times f(n)$



Time complexity

In analysis of algorithm time complexity, we are interested in the number of “**elementary operations/statements**” (not μs).

- Simple statements are constant time.
- Remember the factor *c* in *O(f(n))*.
- Beware: Library/subroutine calls can have arbitrary complexity.

Example: Greatest Up To

Find the greatest element ≤ x in an unsorted sequence of *n* elements (or else return `null`).

Two approaches:

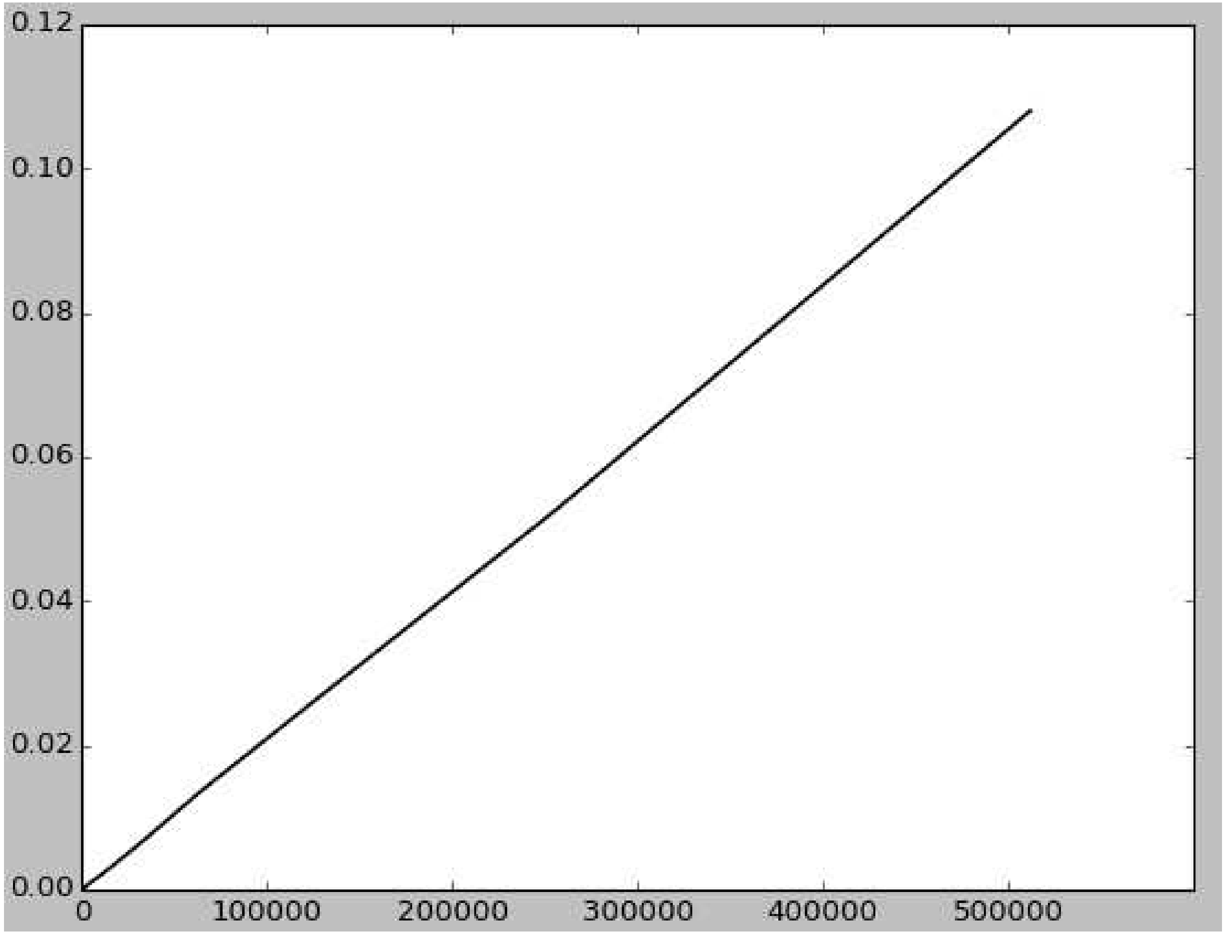
- a) search the unsorted sequence; or
- b) first sort the sequence, then search the sorted sequence.

Unsorted Greatest Up To

```
static Integer unsortedFind(int x, List<Integer> uList) {
    Integer best = null;
    for (var e : uList) {
        if (e == x)
            return e;
        if (e <= x && (best == null || e > best))
            best = e;
    }
    return best;
}
```

Analysis

- If we’re lucky, `uList[0] == x`.
- Worst case?
  - `uList = {x – n, ..., x – 2, x – 1}`
  - *f(n)* = *6n*, so *O(n)*

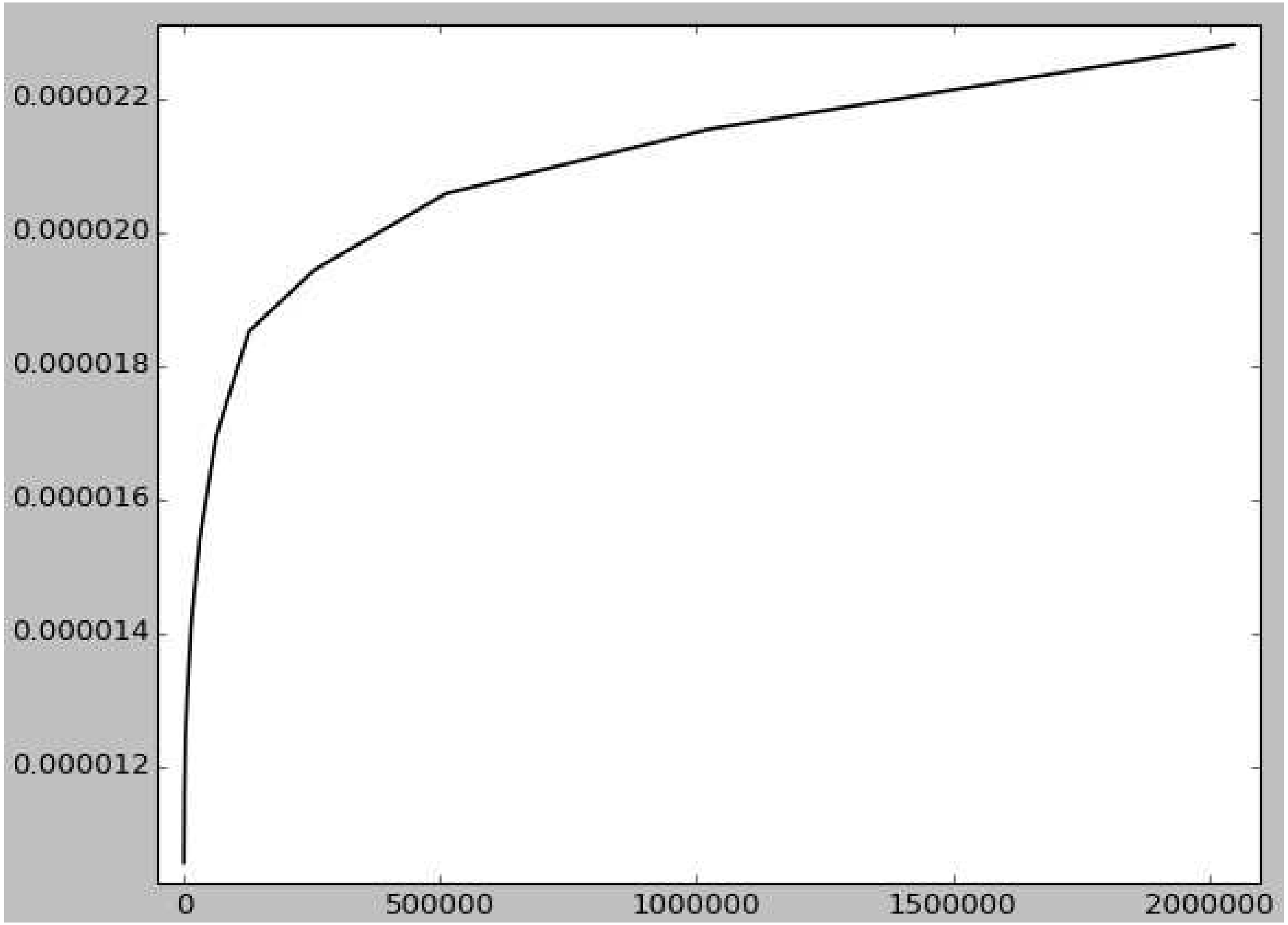

0.12
0.10
0.08
0.06
0.04
0.02
0.00
0
100000
200000
300000
400000
500000

Sorted Greatest Up To

```
static Integer sortedFind(int x, ArrayList<Integer> sList) {
    if (sList.isEmpty() || sList.get(0) > x)
        return null;
    int lower = 0;
    int upper = sList.size();  // one past the end
    while (upper - lower > 1) {
        int mid = (lower + upper) / 2;
        if (sList.get(mid) <= x)
            lower = mid;
        else
            upper = mid;
    }
    return sList.get(lower);
}
```

Analysis

- How many iterations of the loop?
- Initially, `upper – lower = n`.
- The difference is halved in every iteration.
- Can halve it at most $log_2(n)$ times before it becomes 1.
- $f(n) = a\ log_2(n) + b$, so $O(log(n))$.

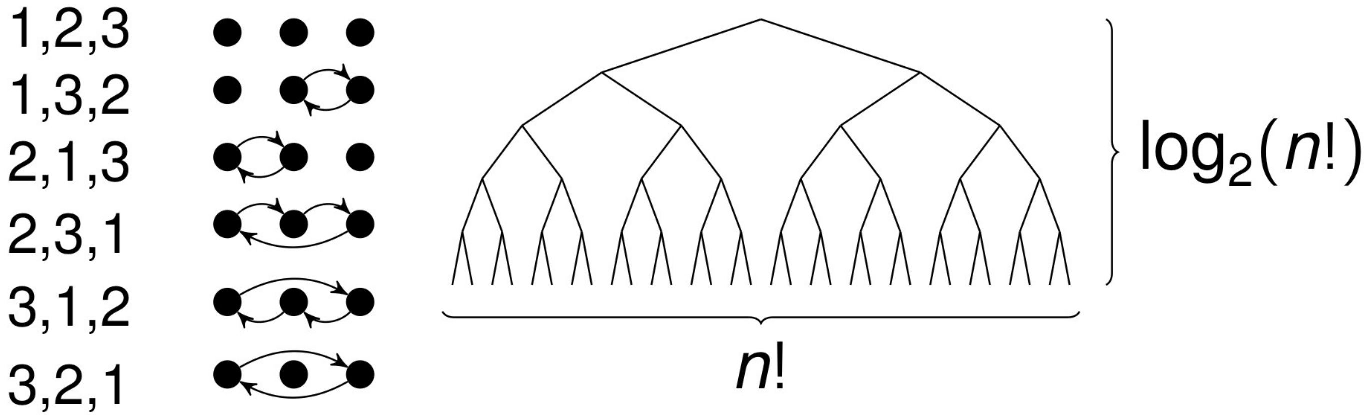

0.000022
0.000020
0.000018
0.000016
0.000014
0.000012
0
500000
1000000
1500000
2000000

Problem complexity

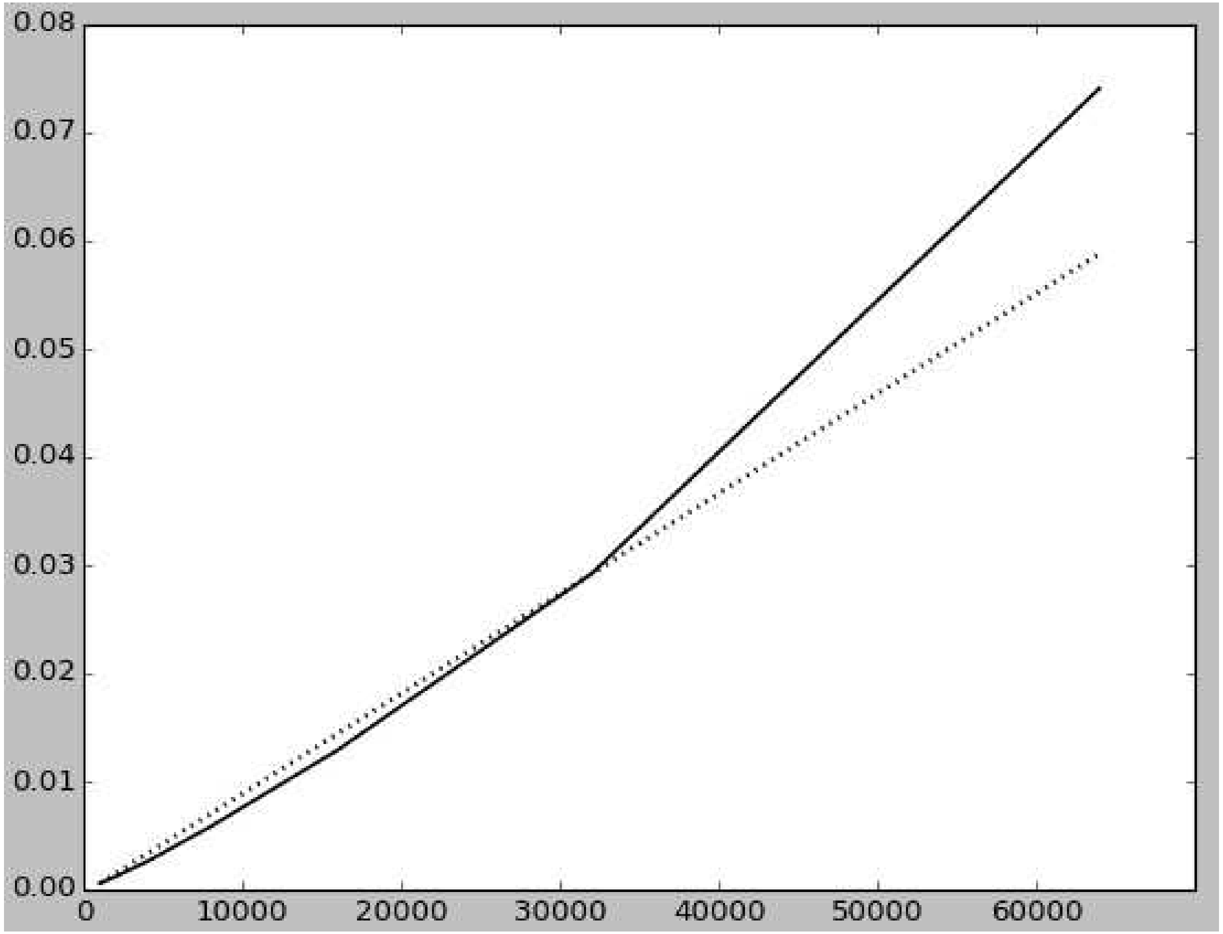
The complexity of a **problem** is the resources (time, memory, etc) that any algorithm *must* use, in the worst case, to solve the problem, as a function of instance size.

How fast can you sort?

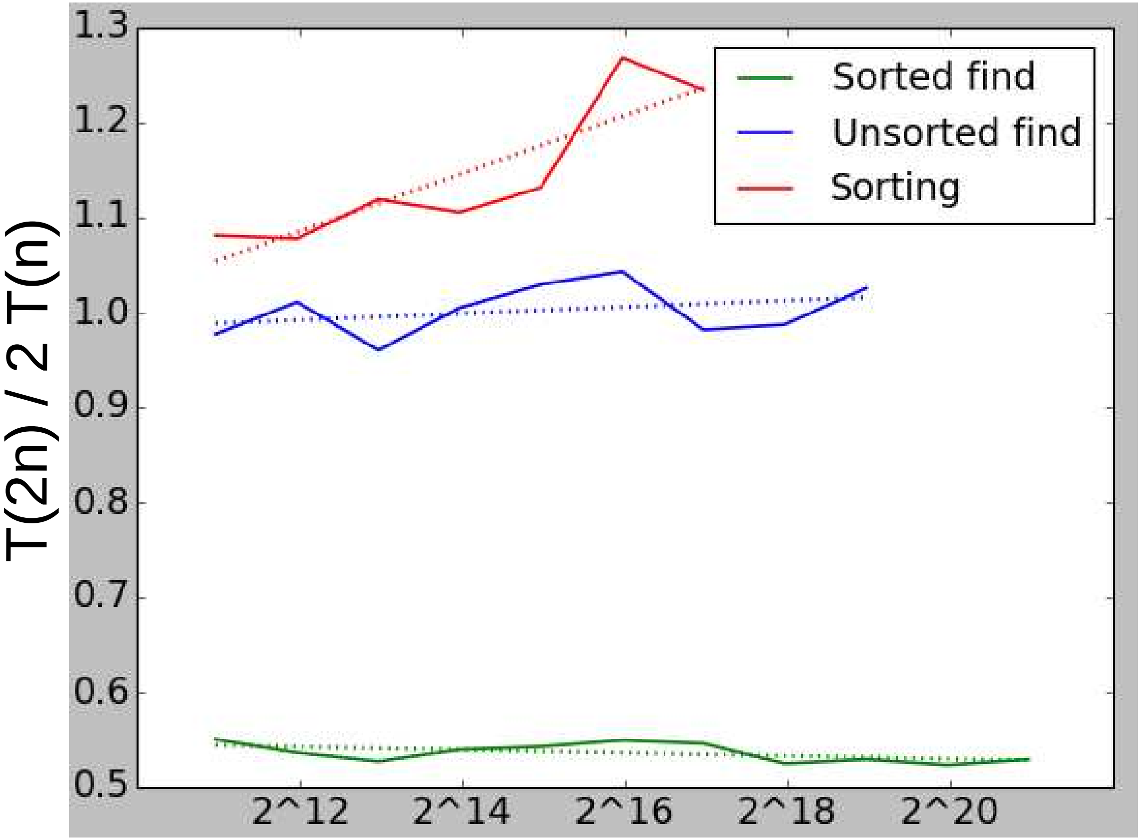
Any sorting algorithm that uses only pair-wise comparisons needs *O(n log(n))* comparisons in the worst case.



$log(n!) = log(1) + log(2) + \ldots + log(n) \leq n\ log(n)$ for large enough n.


0.08
0.07
0.06
0.05
0.04
0.03
0.02
0.01
0.00
0
10000
20000
30000
40000
50000
60000

Rate of Growth



Example: Summing a List

Consider summing a list of size n…

```
public int sum(List<Integer> list) {
    int result = 0;
    for (var i : list) {
        result += i;
    }
    return result;
}
```

Linear time, $O(n)$

Example: Minimum Difference

Note: $n - 1 + n - 2 + \ldots\ 2 + 1 = n(n - 1)/2$

```
public int minDiff(List<Integer> values) {
  int min = Integer.MAX_VALUE;  1
  for (int i = 0; i < values.size(); i++) {   n
    for (int j = i + 1; j < values.size(); j++) {  n(n – 1)/2
      int diff = values.get(i) - values.get(j);  n(n – 1)/2
      if (Math.abs(diff) < min)   n(n – 1)/2
        min = Math.abs(diff);   <= n(n – 1)/2
   }
  }
 }
```

$$
\begin{aligned}
S(n) &= 1 + n + 4\,(n(n - 1)/2) \\
&= 1 + n + 2\,n^2 - 2n \\
&= 2n^2 - n + 1 \ \in\ O(n^2)
\end{aligned}
$$

More Examples

- Constant *O(1)*
  - Time to perform an addition; swap two elements in an array; compare two numbers
  - Time to do any of the above 1000 times.
- Logarithmic *O(log(n))*
  - Time to find an element in a B-Tree (self-balancing tree)
- Linear *O(n)*
  - Time to find an element in a list; sum a list of numbers
  - Find the min/max in a list?
- *O(n log(n))*
  - Time to sort using mergesort
- Quadratic $O(n^2)$
  - Time to compare *n* elements with each other pair-wise.

Caution

“Premature optimization is the root of all evil in programming.”

(C.A.R. Hoare)

Scaling behaviour becomes important when problems become large, or when they need to be solved very frequently.

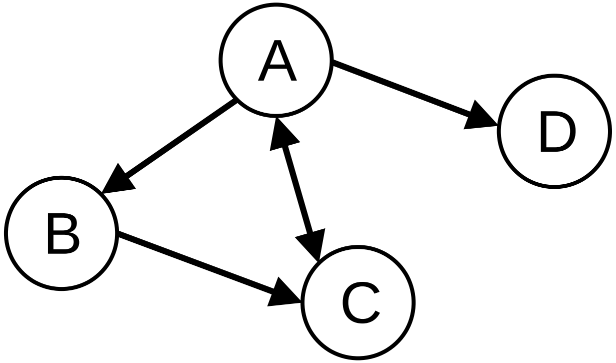
C03 Graph Traversal

Graphs and Trees
Traversal

ANU – School of Computing – Structured Programming 1110 / 1140 / 6710

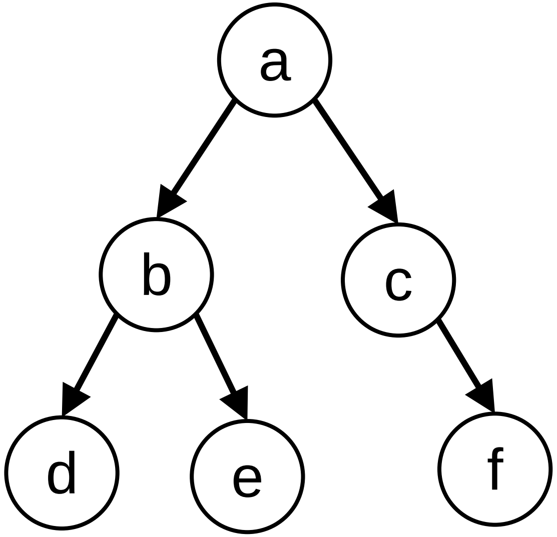
Graphs and Trees

- A powerful abstraction in computing.



Directed **Graph**

Nodes: A B C D
Edges: (A, B) (B, C) (A, C) (C, A) (A, D)



Directed Rooted **Tree**

(*connected acyclic directed* graph)

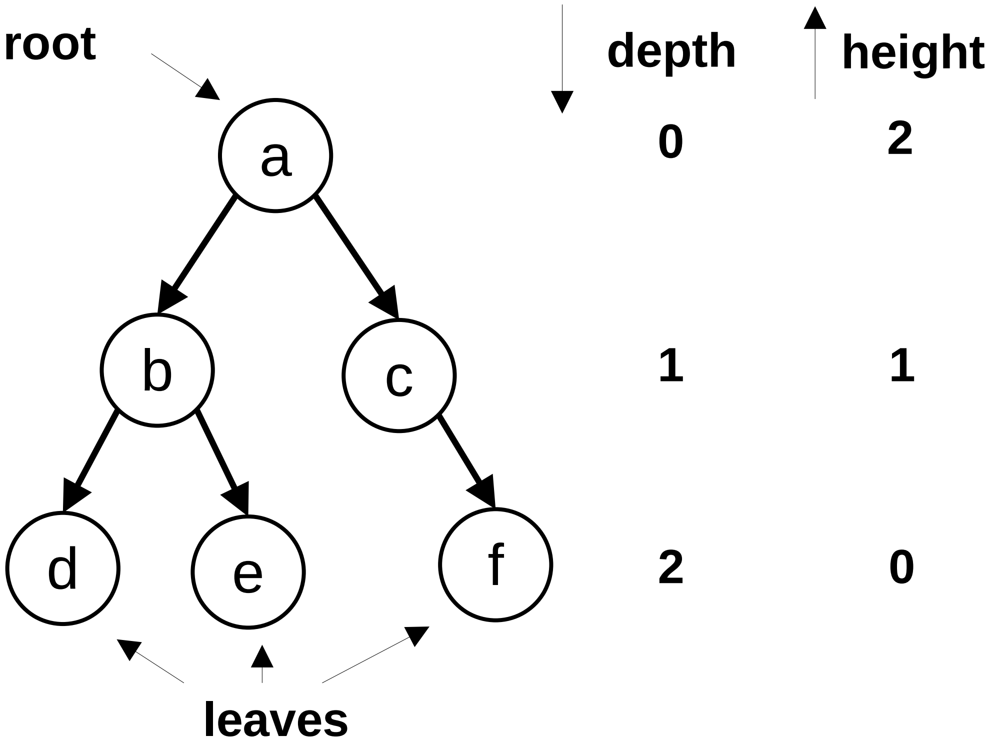
With ordering of children: *Ordered* Tree

Tree Features

b is the **parent** of d and e

d is a **child** of b

b has a **branching factor** (outdegree) of 2 (the number of children)



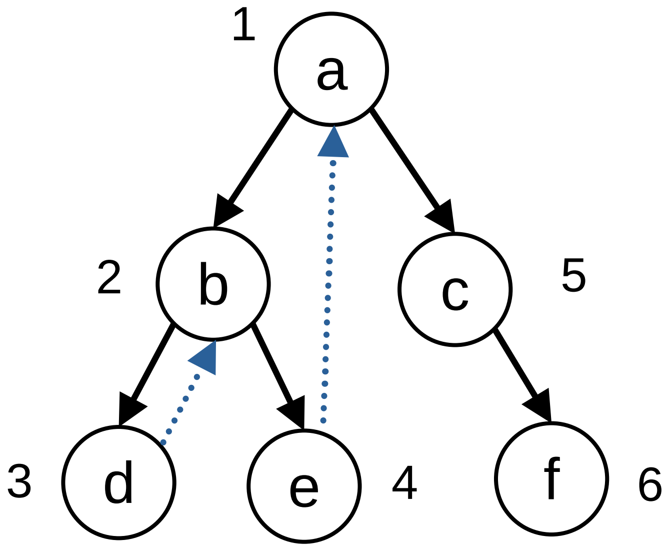
Traversal

- Visiting the elements in a data structure:
  - searching
  - modifying
  - reachability
  - path finding
- Lists / arrays are a form of “linear data structure” that has a natural sequence for traversal.
- Trees and Graphs can be traversed in many ways.

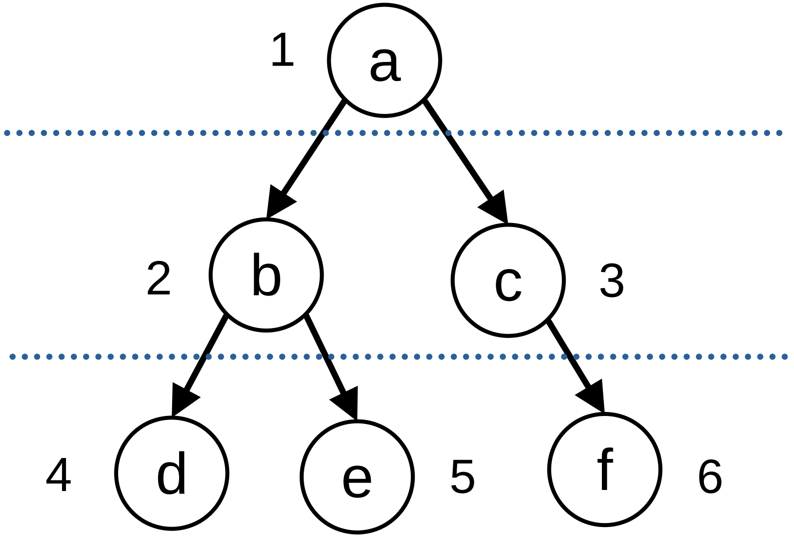
Tree Traversal

- Special case of graph traversal.
- Two common forms:
  - **Depth-First Search (DFS)**
    - Explore as deep as possible along a branch until a leaf is reached.
    - *Backtrack* to another branch (e.g., *sibling* of leaf, or sibling of parent, or …).
  - **Breadth-First Search (BFS)**
    - Starting at root, visit all nodes at given depth before going deeper.

DFS and BFS



Pre-order DFS traversal
a b d e c f



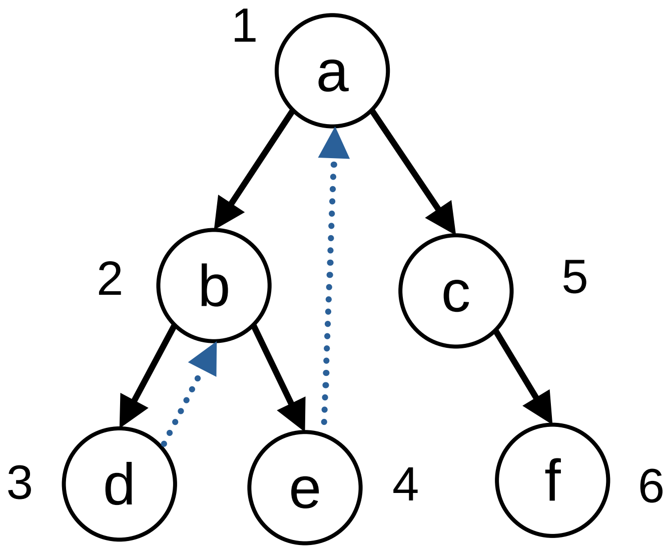
BFS traversal
a b c d e f

Implementing Tree Traversal

- **Depth-First Search (DFS)**
  - Iteratively using a **Stack**: Last-In First-Out (LIFO) data structure
  - Recursively by implicitly using the *call stack*
  - Variations on ordering: post-order, pre-order, in-order
- **Breadth-First Search (BFS)**
  - Iteratively using a **Queue**: First-In First-Out (FIFO) data structure
  - *Corecursively** by passing all sub-trees of same level
  - Only one ordering

* Building (generating) data from a simple “base case”, rather than breaking down (reducing) data until base case reached.

Implementation DFS: Stack



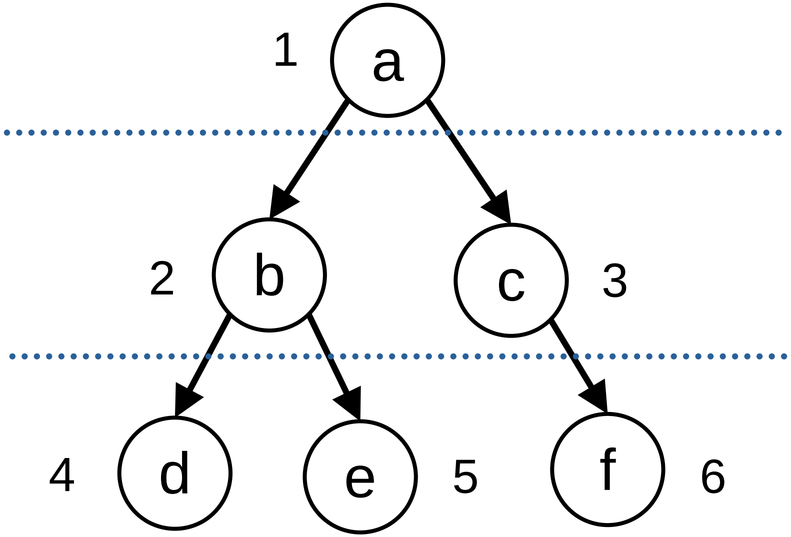
Pre-order DFS traversal
a b d e c f

Stack [ ]: ***push*** onto end, ***pop*** off end

DFS: pop node, push it's children, repeat.

```
0 push a:   [a]
1 pop:      []       a
  push c:   [c]
  push b:   [c b]
2 pop:      [c]      b
  push e:   [c e]
  push d:   [c e d]
3 pop:      [c e]    d
4 pop:      [c]      e
5 pop:      []       c
  push f:   [f]
6 pop:      []       f
```

Implementation BFS: Queue



BFS traversal
a b c d e f

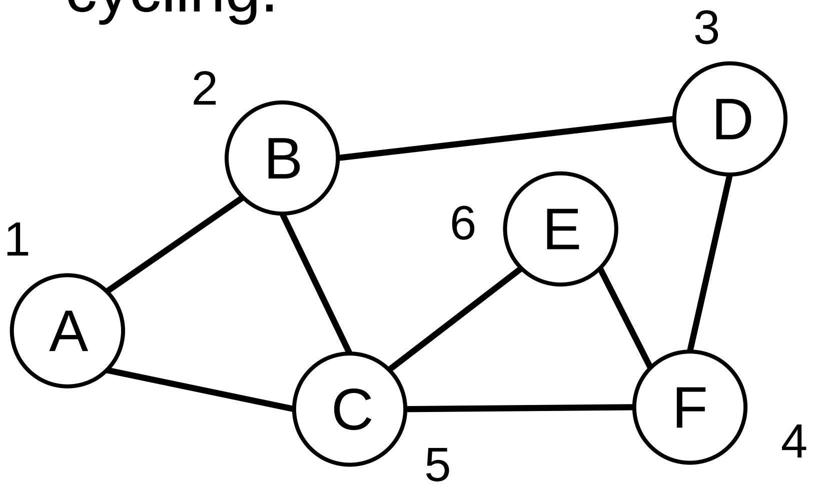
Queue { }: *enqueue* onto back, *dequeue* off front

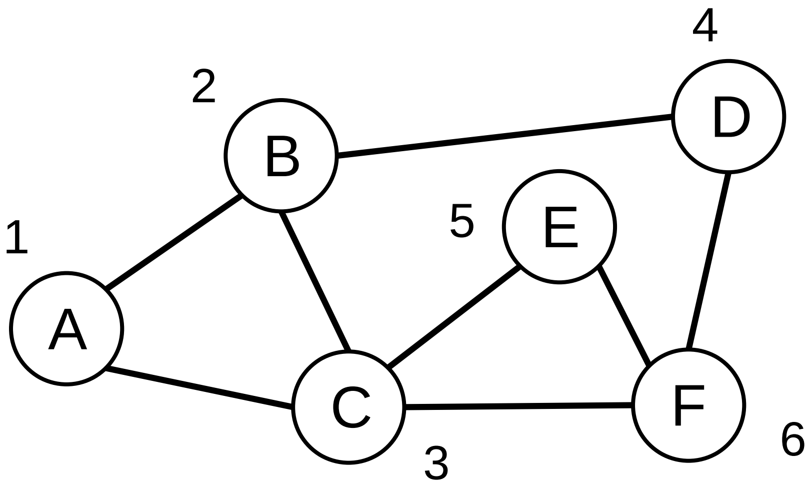
BFS: dequeue node, enqueue it’s children, repeat.

```
0 enq a:    {a}
1 deq:      {}       a
  enq b:    {b}
  enq c:    {b c}
2 deq:      {c}      b
  enq d:    {c d}
  enq e:    {c d e}
3 deq:      {d e}    c
  enq f:    {d e f}
4 deq:      {e f}    d
5 deq:      {f}      e
6 deq:      {}       f
```

Graph Traversal

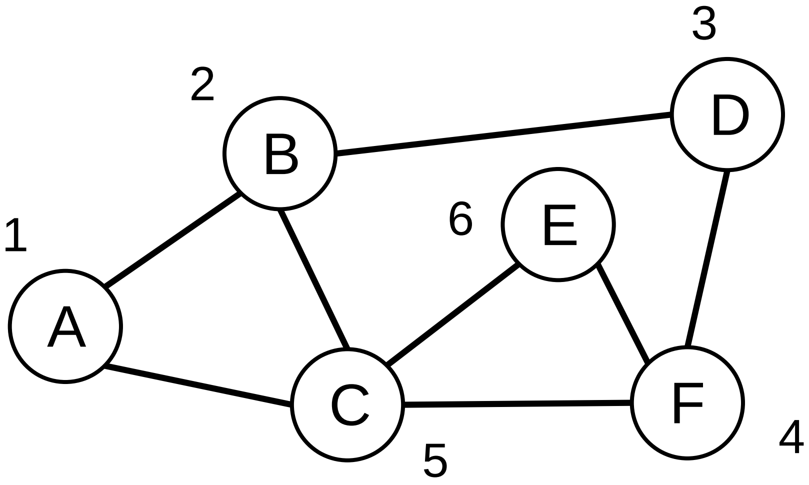
- DFS and BFS generalise from tree traversal.
- Starting node selected based on problem.
- Additionally need to **keep track of “visited”** nodes to avoid cycling.

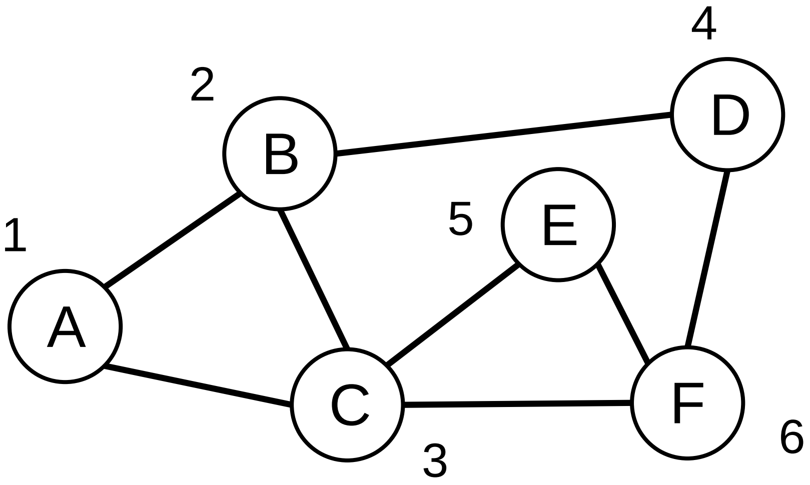




Example: Distance Between Nodes

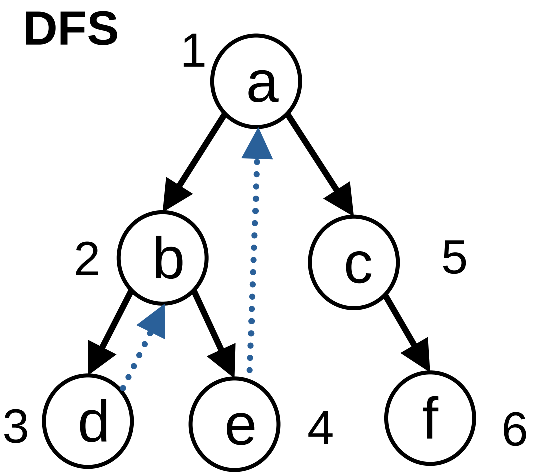
- The *distance* between A and E is the number of edges on a *shortest path* between the two nodes.
- **BFS** can naturally track the distance.
- **DFS** might visit E via a non-shortest *path* – need to revisit nodes

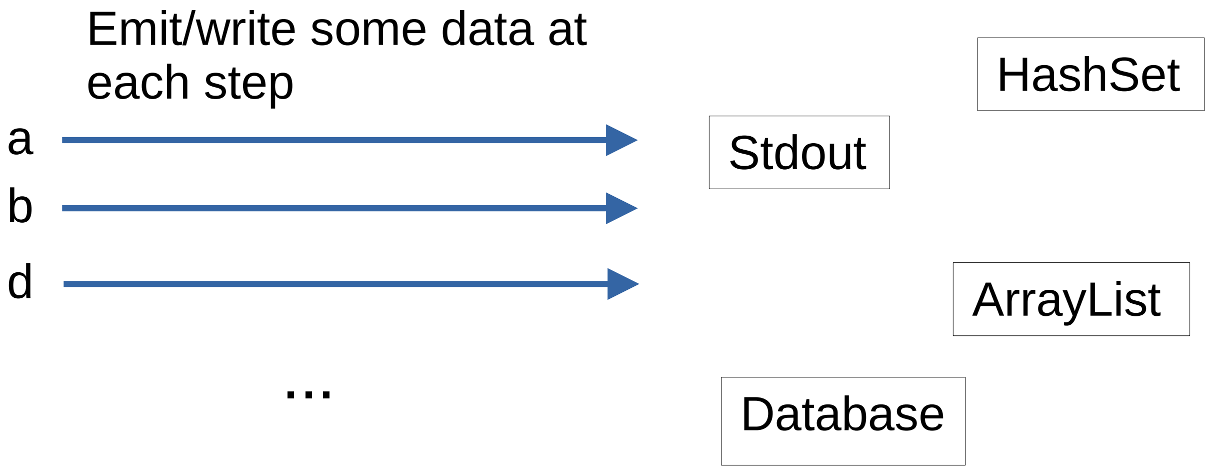




Styles of Using DFS

- Using DFS as skeleton for our code, i.e. we only really care about the traversal pattern.





Emit/write some data at each step

a

b

d

...

HashSet

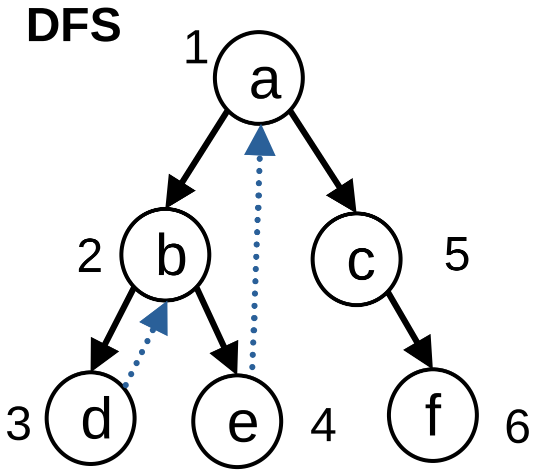
Stdout

ArrayList

Database

Styles of Using DFS

- Height/longest path calculation using a single counter

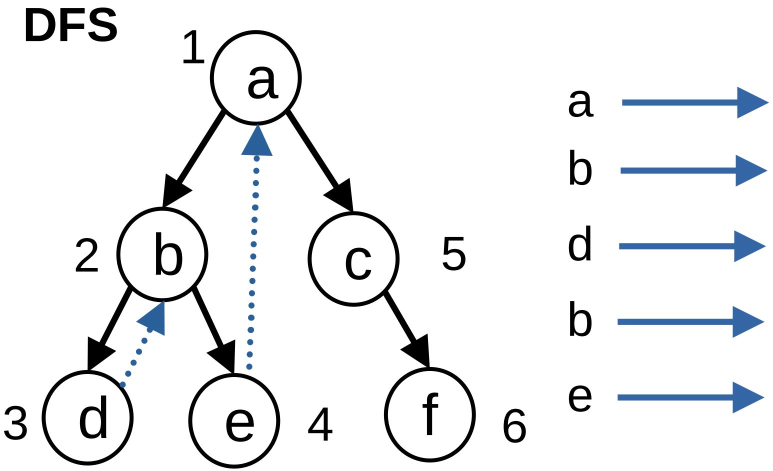


	Action	Current	Best
		0	0
a →	+1	1	0
b →	+1	2	0
d →	+1, update best	3	3
d →	-1 (undo)	2	3
b →	jump to next child	2	3
e →	+1, update best	3	3
e →	-1 (undo)	2	3

Problem: Not all data structures have a clear notion of “undo”, e.g. set

Styles of Using DFS

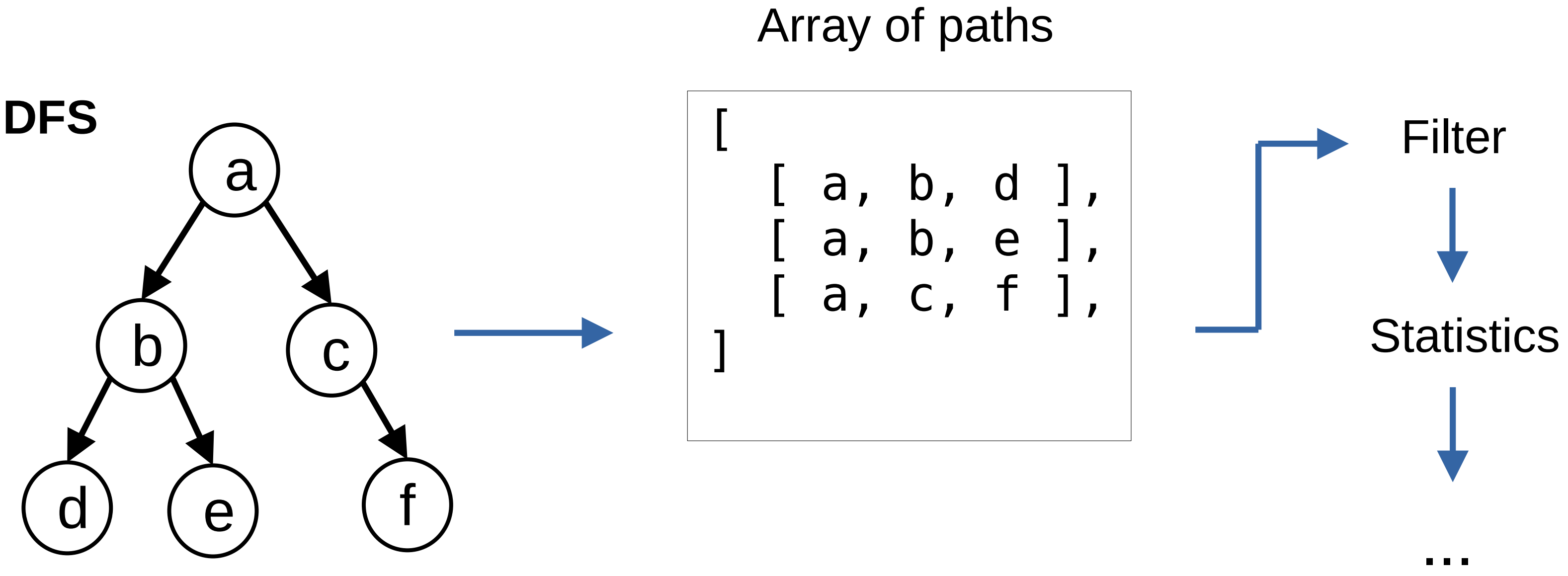
- Height/longest path calculation using record of history



Current	Action	Lengths/history
1		[ ]
2		[ ]
3	Record cur	[3]
2		[3]
3	Record cur	[3, 3]

Styles of Using DFS

- Using DFS to produce well structured data to pass to next stage in a self contained way.



Array of paths

```
[
   [ a, b, d ],
   [ a, b, e ],
   [ a, c, f ],
]
```

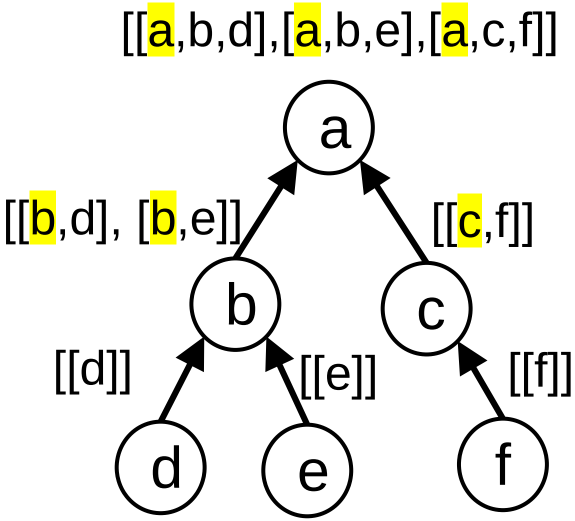
Filter

Statistics

...

Building the data bottom up

- Using DFS to produce well structured data to pass to next stage in a self contained way.

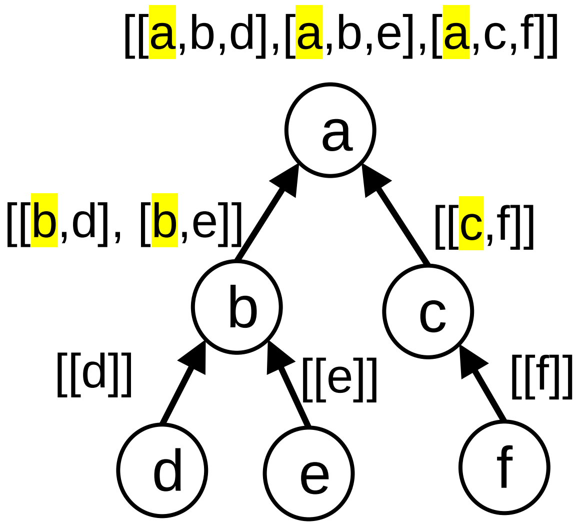


“Concatenation” here is in some sense “combine and flatten”

```
  [[x0,x1,…]] + [[y0,y1…]]

- combine →
  (combining directly adds one layer of container,
  i.e. we have container of containers of containers)

  [[[x0,x1,…], [y0,y1,…]]]

- flatten →
  (flattening removes that extraneous layer,
  so we get “container of containers” back)

  [[x0,x1,…],[y0,y1,...]]
```

Building the data bottom up with flat map

- Using DFS to produce well structured data to pass to next stage in a self contained way.



[[a,b,d],[a,b,e],[a,c,f]]

[[b,d], [b,e]]

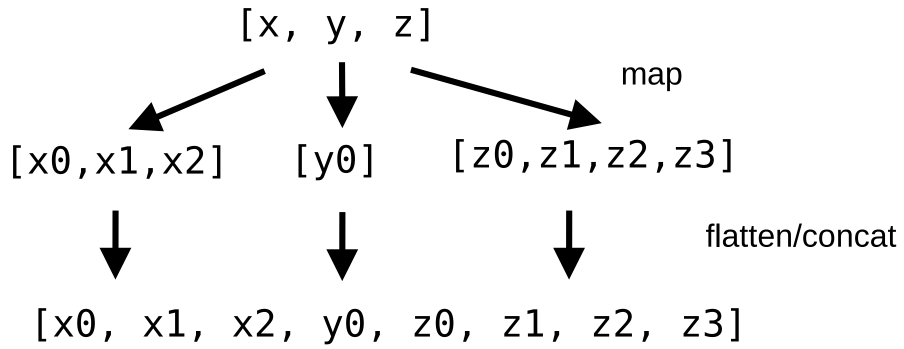
[[c,f]]

[[d]]

[[e]]

[[f]]

“Flat map” (or “concat map”) is then an extension of that idea



[x, y, z]

map

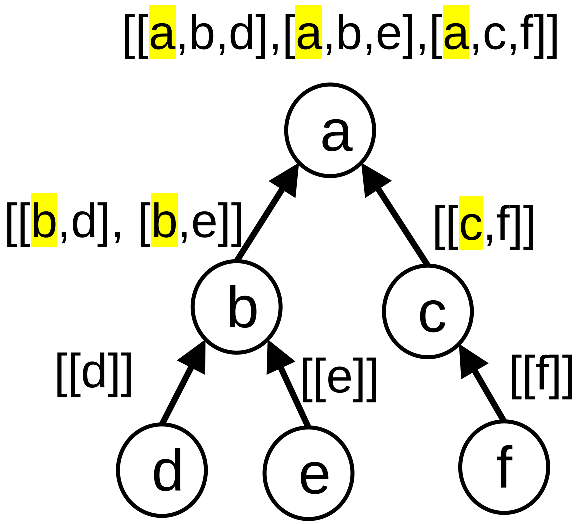
[x0,x1,x2] [y0] [z0,z1,z2,z3]

flatten/concat

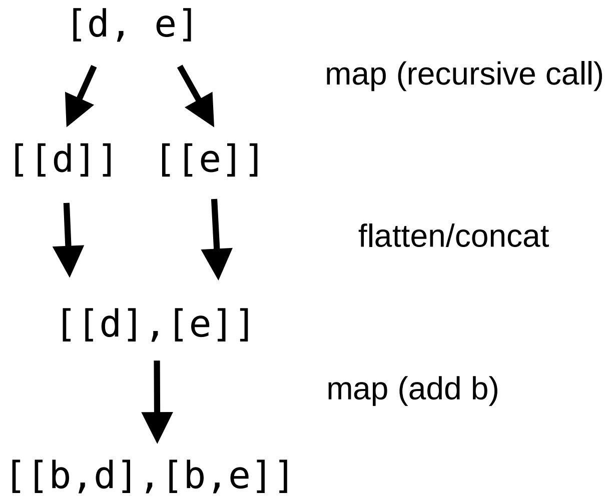
[x0, x1, x2, y0, z0, z1, z2, z3]

Building the data bottom up with flat map

- Using DFS to produce well structured data to pass to next stage in a self contained way.



Looking at the bottom left subtree with b as root



```
[d, e]
```

map (recursive call)

```
[[d]]  [[e]]
```

flatten/concat

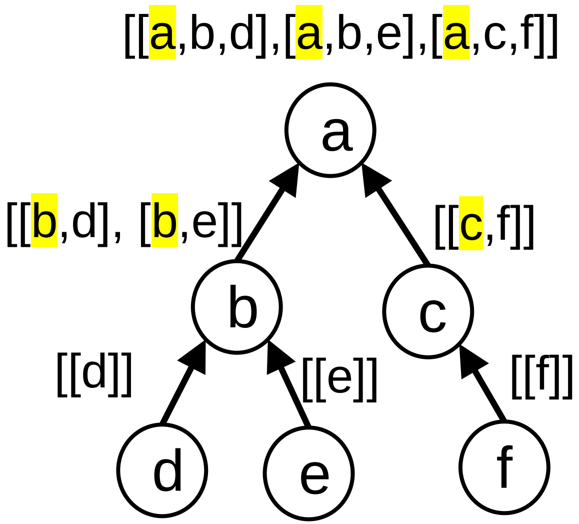
```
[[d],[e]]
```

map (add b)

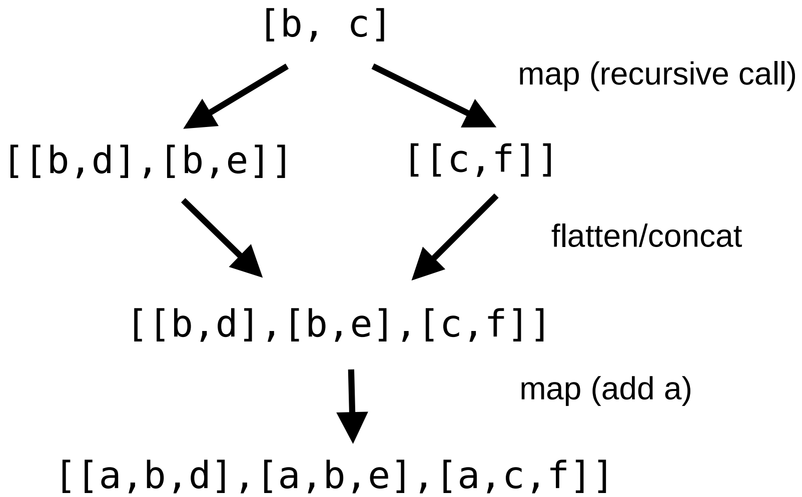
```
[[b,d],[b,e]]
```

Building the data bottom up with flat map

- Using DFS to produce well structured data to pass to next stage in a self contained way.

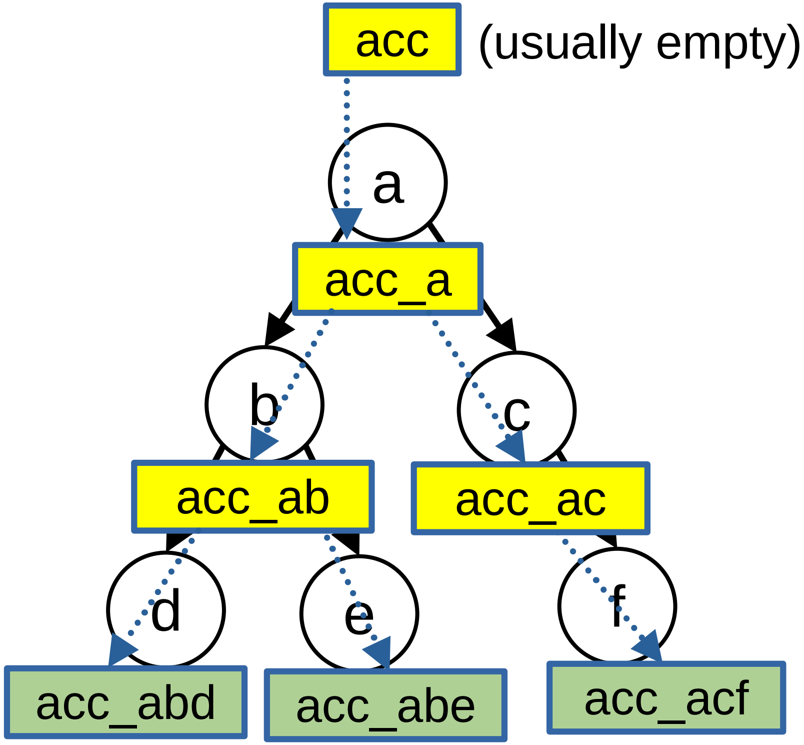


Looking at the entire tree with a as root



Styles of Using DFS

- A more general pattern is an accumulator pattern.



Accumulated value may be:
- Nodes visited
- Path from root so far
- All of above

You can mix accumulator and previous "building bottom up" style by just passing accumulator as argument during recursion

C04 Hash Functions

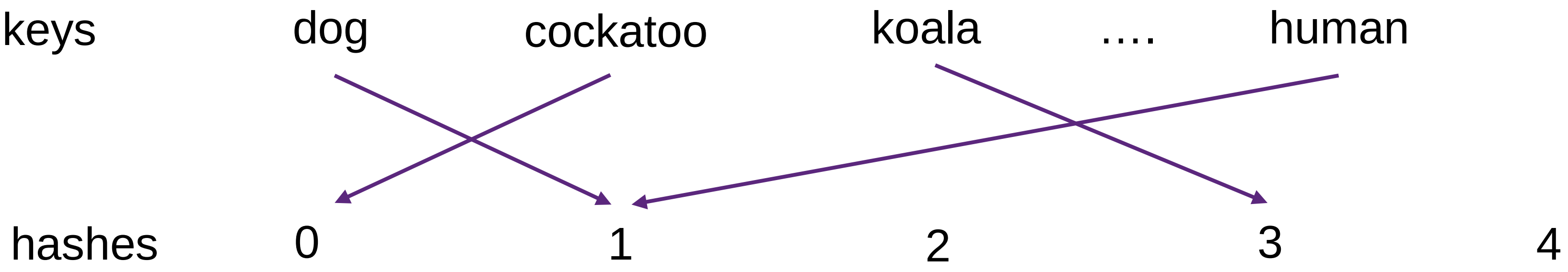
Hash functions
Choosing a good hash function

ANU – School of Computing – Structured Programming 1110 / 1140 / 6710

Hash Functions

A hash function is a function *f* that maps a key *k*, to a value *f(k)*, within a prescribed range. It maps arbitrary sized keys to fixed-sized *hashes*.

A hash is deterministic. (for a given key, *k*, *f(k)* will always be the same)



Choosing a Good Hash Function

A **good hash** for a given population, *P*, of keys, $k \in P$, will distribute *f(k)* evenly within the prescribed range for the hash.

A ***perfect* hash** will give a unique *f(k)* for each $k \in P$.

(Perfect hash is rarely possible:

Pigeon hole principle.)

https://upload.wikimedia.org/wikipedia/commons/thumb/5/5c/TooManyPigeons.jpg/220px-TooManyPigeons.jpg

Why value determinism and even distribution?

- Lets reword how we stated determinism a bit:
  - Given x, y, if x == y, then h(x) == h(y).
  - It follows that (by contraposition):
    - If h(x) != h(y), then x != y
- Even though we cannot give positive result (x is y) confidently,
  - We can for the negative result (x is **not** y)

Why value determinism and even distribution?

- Now lets suppose h(x) gives an integer in range [0, 9]
- And suppose input is uniformly random
- With 10 values (or buckets), given inputs x and y, we have 90% chance of deciding x != y in O(1)
- There is still a 10% chance of collision, but we have cut down our average workload of later stage by 90%
  - HashSet vs ArrayList
  - More applications in C05

Why so many different hashes?

- We outlined the basic properties we look for in a hash
  - Deterministic
    - This is fundamental, and by definition of a mathematical function
    - No exception to this requirement
  - Even/uniform distribution of output
    - This is not as indisputable – we don’t know what the distribution of input is like
    - But we try to obtain this by guessing what the “usual” input looks like, e.g. statistical analysis of past usage
- The second point is roughly where the divergence begins

Why so many different hashes?

- For each input distribution, we would need a different hash function to get an even distribution

Evenly distributed output if input is normally distributed

Deterministic

Evenly distributed output if input is evenly distributed

Deterministic

Evenly distributed output if input is bimodal

Deterministic

Why so many different hashes?

- Even more variations if we want additional properties

Fast on CPU

Evenly distributed output if input is normally distributed

Deterministic

Secure

Low memory usage

Evenly distributed output

Deterministic

Low memory usage

Evenly distributed output if input is evenly distributed

Deterministic

Very high memory usage

Very slow

Evenly distributed output

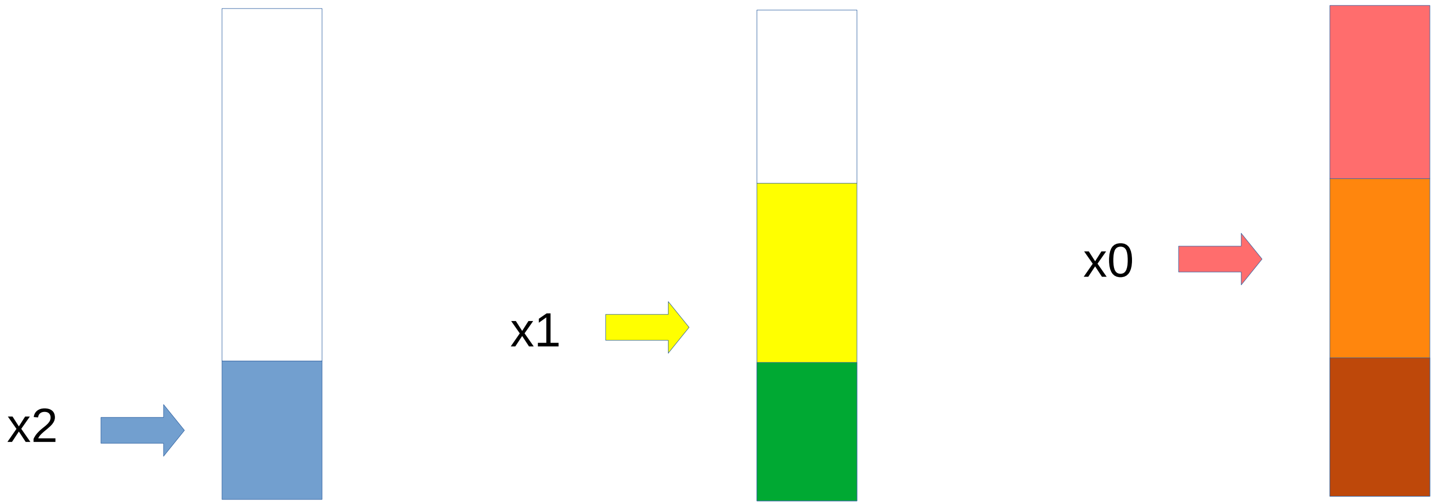
Deterministic

Assume whatever distribution, pick a recipe

- From “Effective Java”, Josh Bloch
- (An approximate translation below in pseudo code)
- Assume you have fields (or more generally values) field0, field1, field2, …
-
```
int result = 0; // accumulator
for (var field : fields) {
    var x = convertToInt(field); // recursively call this hash if needed
    result = 31 * result + x;
}
```
- How does this work? Suppose we have fields: x0, x1, x2
- After loop 0, result = x0
- After loop 1, result = 31 * x0 + x1
- After loop 2, result = 31 * (31 * x0 + x1) + x2 = 961 * x0 + 31 * x1 + x2

Intuition behind this pattern

- Why 961 * x0 + 31 * x1 + x2 (or similar)
- Each factor is used to disperse the field to a different band/partition of the output range
- So it is sensitive to change of any field



Why 31?

- From the book, multiplication with 31 is very efficient:
  - 31 * x = (x << 5) - 1
- A more impactful answer (my guess) is we don’t use odd prime very often. Suppose we use 100 instead of 31:
  - 10000 * x0 + 100 * x1 + x2
- Suppose we reduce the range of hash by doing % 10, above becomes
  - x2

Why 31?

- Of course if we modulo 31, then we run into the same problem
- But not a super common number to use
  - We see a lot of things using base 10, e.g. 10, 100, 1000
    - Natural to human
  - Or base 2, e.g. 1024, 2048, 4096
    - Natural to machine
  - Odd primes, less so. (We could have replaced 31 with 7 etc.)

Converting things into int

- Again mostly based on the recipe from Effective Java book
- Any numeric primitive type: multiply by prime, hashCode(), Float.floatToIntBits(x)
- Recursive: 31 * node.left.hashCode() + node.right.hashCode()
- Linear/array: treat each element as a field in previous recipe

More complex hash

- We can always mix and match, and use the recipe as the base skeleton
- Suppose we parameterise the recipe as
  - hash(int prime, List<int> fieldHashes)
- Examples:
  - hash(31, fields in some order) // original reciple
  - hash(31, fields in some order) + hash(7, fields in reverse order)
  - Use a mix of primes: 67 * 31 * x0 + 31 * x1 + x2

C05 Hashing Applications

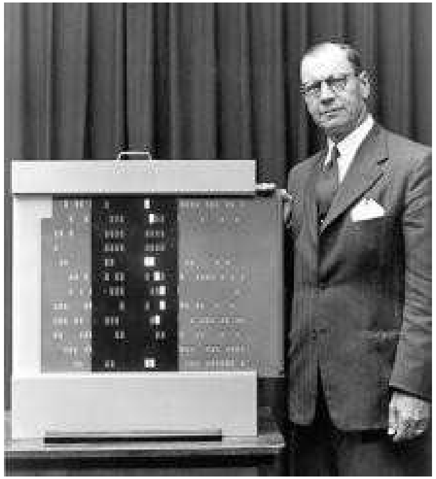
Uses of hashing
Java hashCode()

ANU – School of Computing – Structured Programming 1110 / 1140 / 6710

Uses of Hashing

- Hash table (implement a set or map)
- Checksums
  - Error detection and/or correction
- Compression
  - A hash is typically much more compact than the key
- Pruning a search
  - Looking for duplicates
- Cryptographic

Practical Examples…



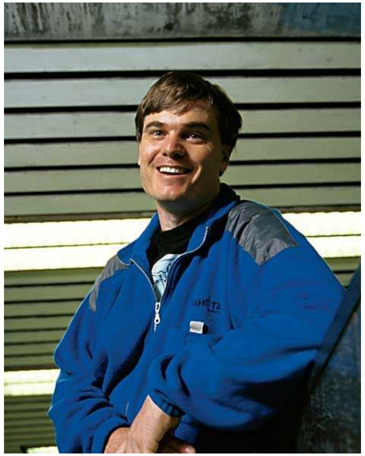
Luhn Algorithm

Used to check for transcription errors in credit cards (last digit checksum).

Hamming Codes

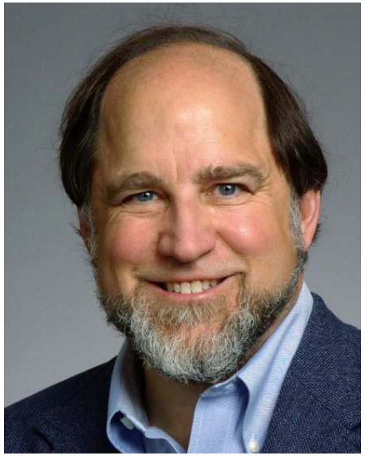
Error correcting codes (as used in EEC memory).

Practical Examples…



rsync (Tridgell)

Synchronize files by (almost) only moving the parts that are different.



MD5 (Rivest)

Previously used to encode passwords (but no longer).

Java `hashCode()`

Java provides a hash code for every object.

- 32-bit signed integer
- Inherited from `Object`, but may be overwritten
- Objects for which `equals()` is `true` must also have the same `hashCode()`.
- The hash need not be perfect (i.e. two different objects may share the same hash).

C06 Files

Java File IO
Streams
Standard IO
Buffering

ANU – School of Computing – Structured Programming 1110 / 1140 / 6710

What is a file?

A file is a collection of data on secondary storage (hard drive, USB key, network file server).

Data in a file is a sequence of bytes (integer $0 \leq b \leq 255$).

- The program reading a file must interpret the data (as text, image, sound, etc).
- Standard libraries provide support for interpreting data as text.

I/O streams

A stream is a standard abstraction used for files:

- A sequence of values are read.
- A sequence of values are written.

The stream reflects the sequential nature of file IO and the physical characteristics of the media on which files traditionally reside (e.g. tape or a spinning disk).

Other I/O (e.g., network, keyboard) is also typically accessed as streams.

A
maxell
maxell
UR
90
POSITION NORMAL

TDK
DATA CARTRIDGE
DC 6250
Cartouche Numérique
FUJIFILM
DG4-150
DataLife

I/O in Java: Byte streams

The classes `java.io.InputStream` and `java.io.OutputStream` allow reading and writing bytes to and from streams.

- Subclasses: `FileInputStream` and `FileOutputStream` for files.
  - Open the stream (create stream object)
  - Read or write `byte`s from the stream
  - Wrap operations in a `try` clause
  - Use `finally` to close the streams

I/O in Java: Character streams

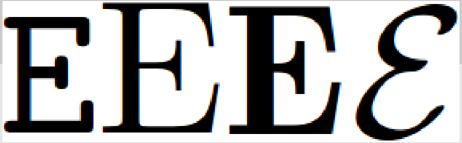
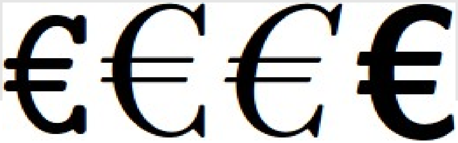
To read/write text files, use `java.io.Reader` and `java.io.Writer` which convert between **bytes** and **characters** according to a specified encoding.

- Subclasses: `InputStreamReader` and `OutputStreamWriter`
- Subclasses `FileReader` and `FileWriter` (shortcuts for wrapping a `FileInputStream` / `FileOutputStream` in a `InputStreamReader` / `OutputStreamWriter`).

Text encoding

Each character is assigned a number.

Unicode defines a unique number (“code point”) for > 120,000 characters (space for > 1 million).

Encoding (UTF-8)	Font	
Bytes →	Code point →	Glyph
0100 0101 (69)	69	
1110 0010 (226) 1000 0010 (130) 1010 1100 (172)	8364	

Buffering I/O

In traditional storage media, accessing a specific byte (point in a file) is time consuming:

Disk: ~2-10ms **SSD**: ~10-100µs **RAM**: ~100ns **Cache**: ~1-15ns

But reading a consecutive “block” at one time is not much more so. Hence, buffering is used to absorb some of the overhead.

- `BufferedReader` and `BufferedWriter` can be wrapped around other reader/writer (e.g., `FileReader` and `FileWriter`) to buffer I/O.
- To flush the buffer, call `flush()`, or close the file.

Terminal I/O

Three standard I/O streams:

- standard input: (usually typed) input to the program
- standard output: normal printed program output
- standard error: program error messages (not buffered)
- Available in Java as System.in, and System.out and System.err.

```
byte b = (byte) System.in.read();
System.out.write(b);
System.out.flush();
System.err.write(b);
```

C07 Threads

Concurrency
Threads

ANU – School of Computing – Structured Programming 1110 / 1140 / 6710

Concurrency, processes and threads

- Concurrency
  - Multiple activities (appear to) occur simultaneously.
  - ‘Time slicing’ allows a single execution unit to give the appearance of concurrent execution.
- Process
  - Distinct execution context that (by default) shares nothing.
- Thread
  - Intra-process execution context.
  - Multiple threads can (and do) execute the same methods on the same objects.

Why threads?

- ‘Concurrency’
  - Separate concerns (e.g. rendering vs. logic)
  - Good for: distinct tasks that naturally occur concurrently
- ‘Parallelism’ (a special case of concurrency)
  - Break task into pieces, exploit parallel hardware
  - Good for: computationally intensive problems that can be readily partitioned