

The background of the slide is a painting of a field of blue irises. The irises are in various stages of bloom, with some fully open and others as buds. The leaves are long and green. In the background, there are other flowers, some yellow and orange, suggesting a diverse field. The painting style is expressive, with visible brushstrokes and a rich color palette.

C01 Recursion

Recursive data structures
Recursive algorithms



PAUL
NOTH



Dröste®

HOLLAND



COCOA

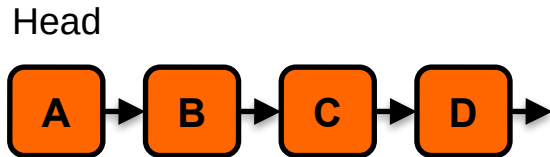
NET WT 250g e 8.8 oz.

Kosher

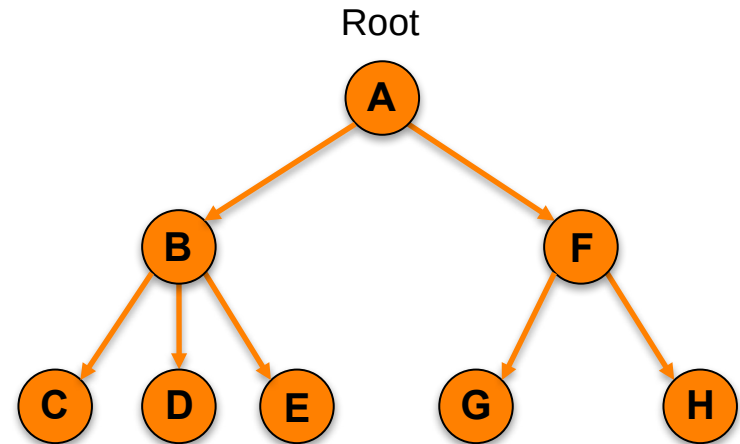


Recursive Data Structures

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



Linked list



Tree

Recursive Algorithms

A recursive algorithm references itself.

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

$\text{fib}(0) = 0$ (base case)

$\text{fib}(1) = 1$ (base case)

$\text{fib}(n) = \text{fib}(n-1) + \text{fib}(n-2)$ (for $n \geq 2$)



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)

38	27	43	3	9	82	10
----	----	----	---	---	----	----



Recursion

- 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 straight-forward.

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* is more limited in Java – but important cases still exist!

The background of the slide is a painting of a field of red poppies. The flowers are scattered across a green field, with some areas appearing more densely packed. The brushstrokes are visible, giving the painting a textured, impressionistic feel. The colors are vibrant, with various shades of green and red.

C02 Computational Complexity

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

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

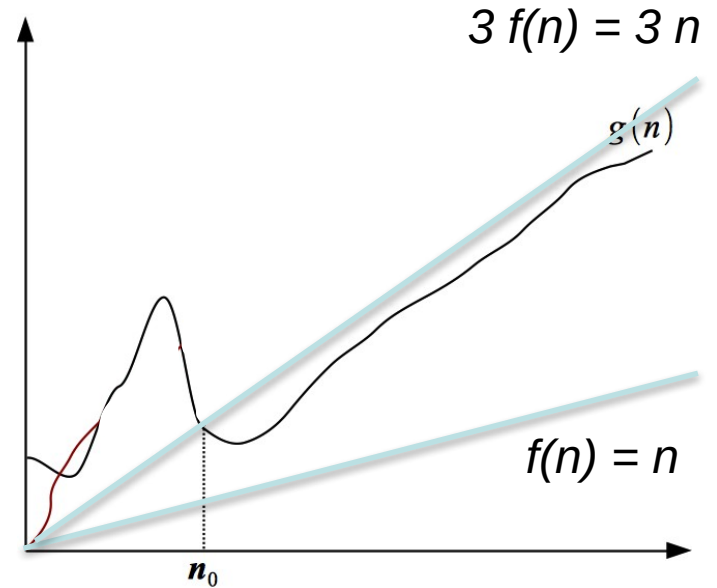
Suppose we have a problem of size n that takes $g(n)$ time to execute in the average case.

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.

Summing a List

Consider summing a list of size n ...

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

Linear time, $O(n)$

Minimum Difference

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

```
public int minDiff(ArrayList<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  
        }  
    }  
}
```

$$S(n) = 1 + n + 4 \frac{n(n - 1)}{2}$$
$$= 1 + n + 2n^2 - 2n$$
$$= 2n^2 - n + 1 \in O(n^2)$$

More Examples

- Constant $O(1)$
 - Time to perform an addition
- Logarithmic $O(\log(n))$
 - Time to find an element in a B-Tree (self-balancing tree)
- Linear $O(n)$
 - Time to find an element within a list
- $O(n \log(n))$
 - Average time to sort using mergesort
- Quadratic $O(n^2)$
 - Time to compare n elements with each other pair-wise

Example: Greatest Up To

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

Two approaches:

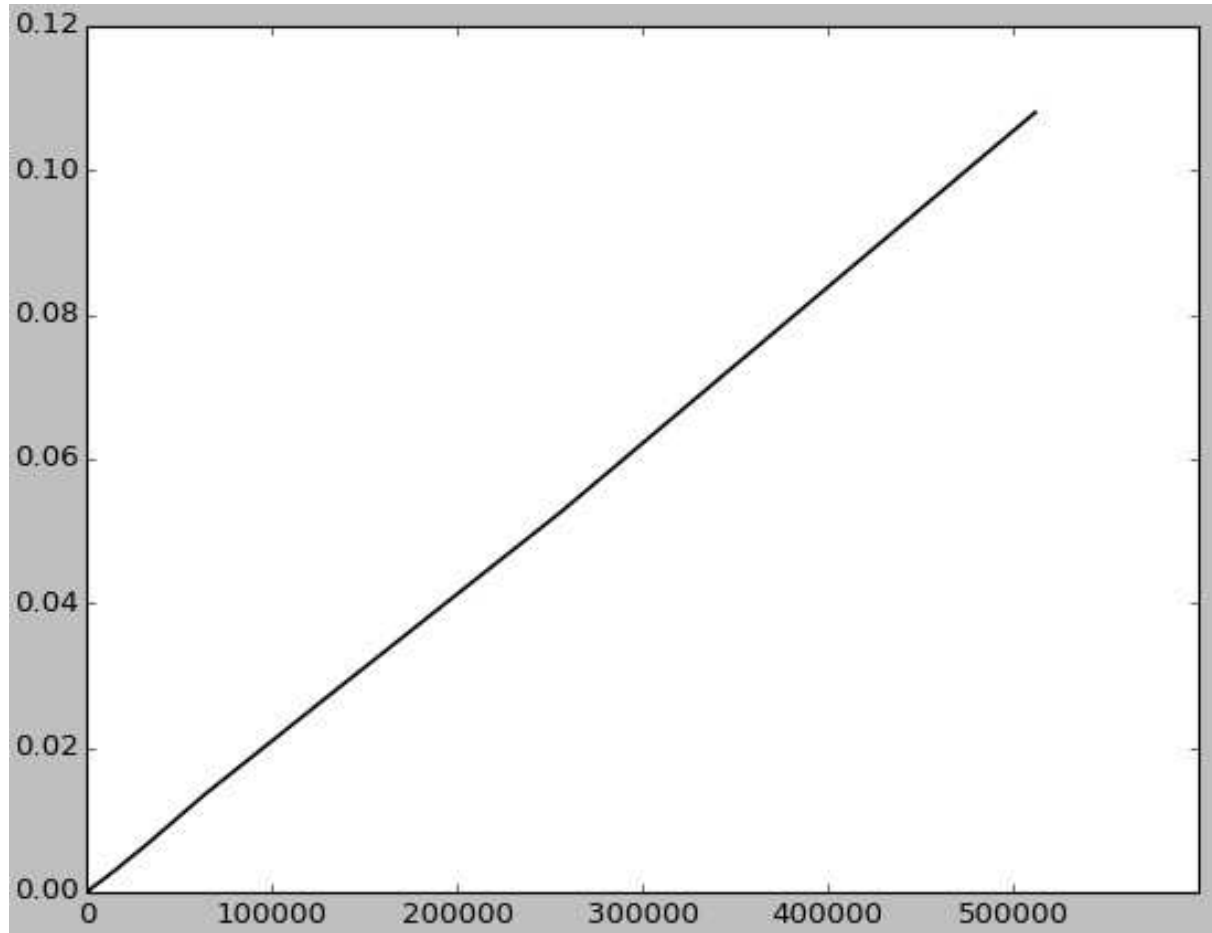
- a) search the unsorted sequence; or
- b) first sort the sequence, then search the sorted sequence.

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 $O(n)$

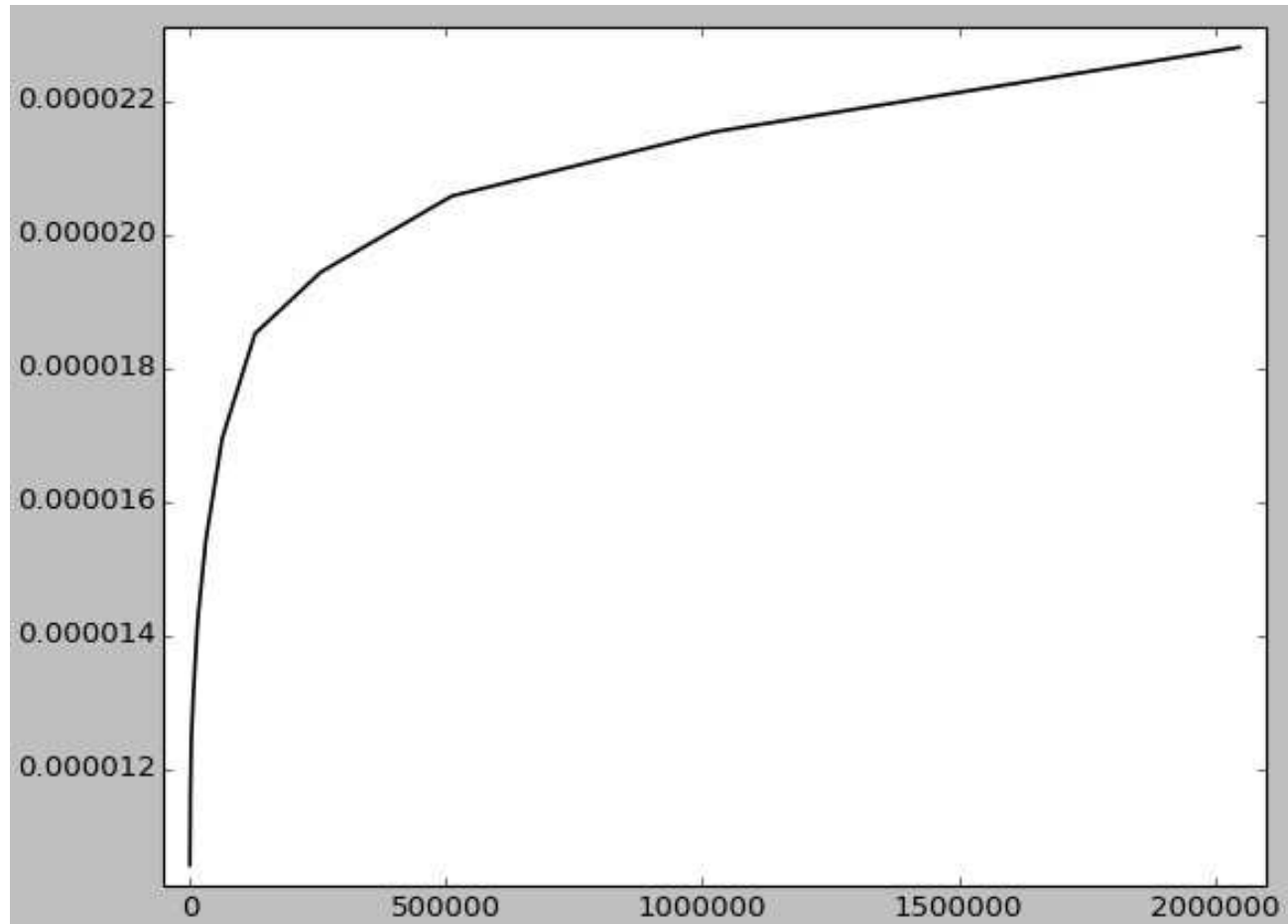


Sorted Greatest Up To

```
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 $O(\log(n))$.

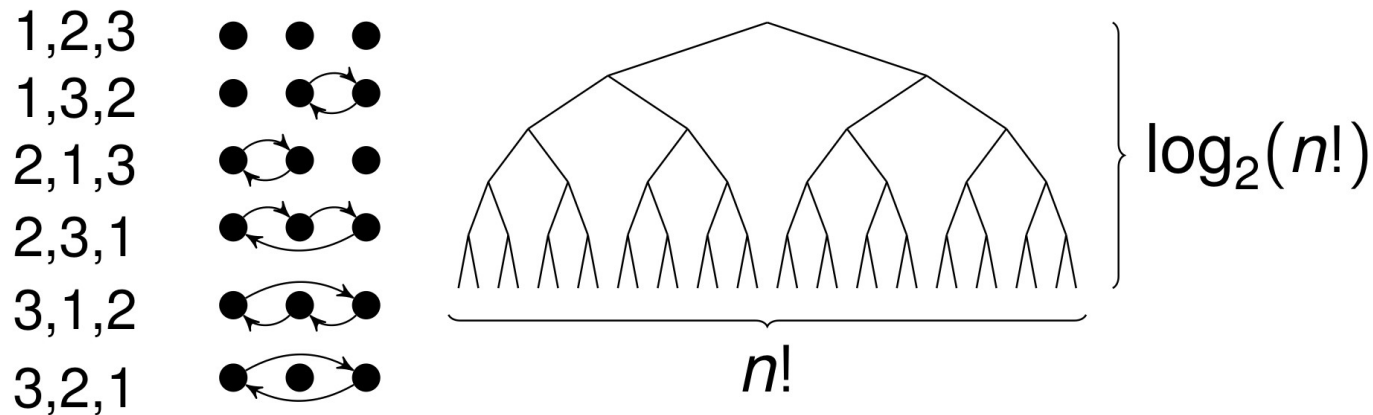


Problem complexity

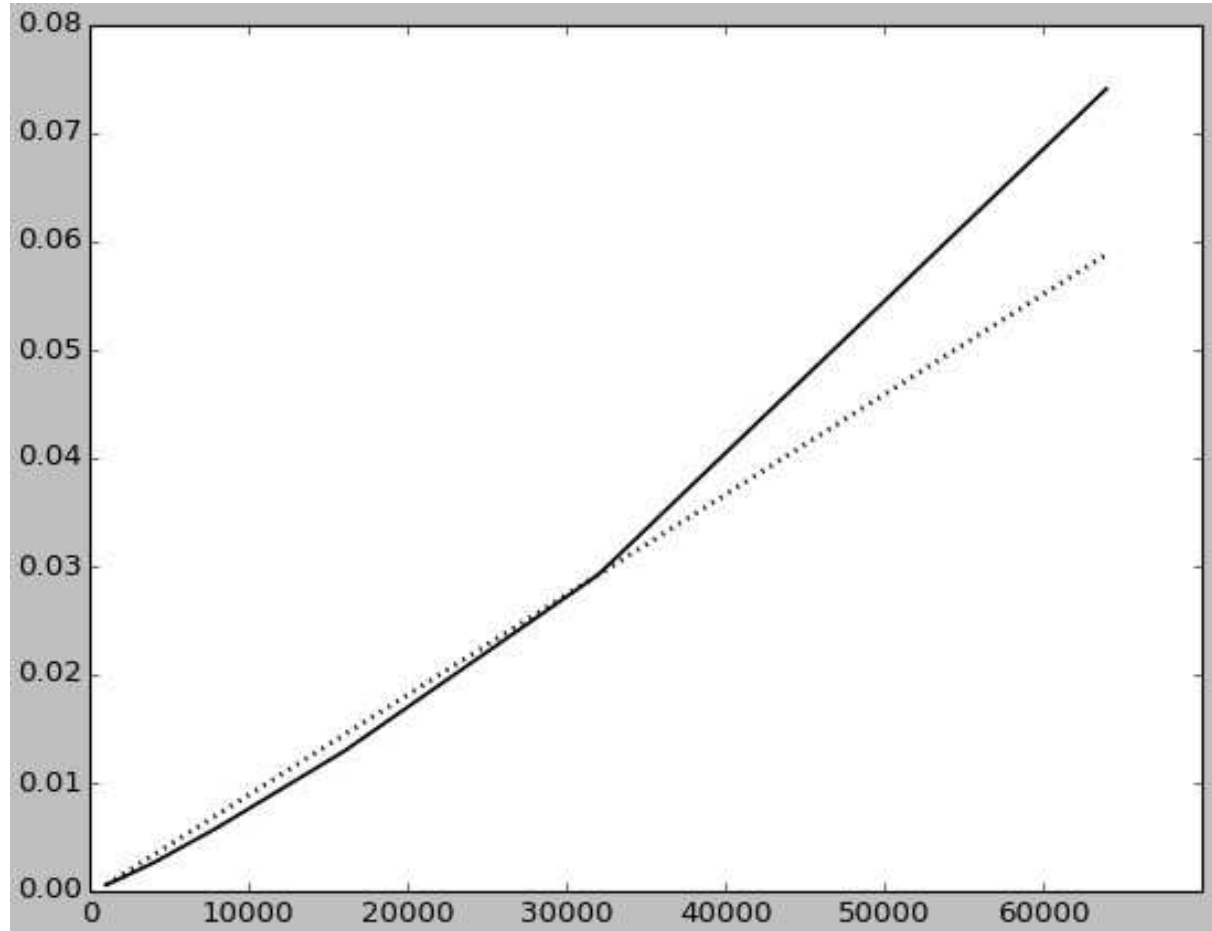
The complexity of a **problem** is the resources (time, memory, etc) that any algorithm *must* use, in the worst case, to solve the problem, as a function of instance size.

How fast can you sort?

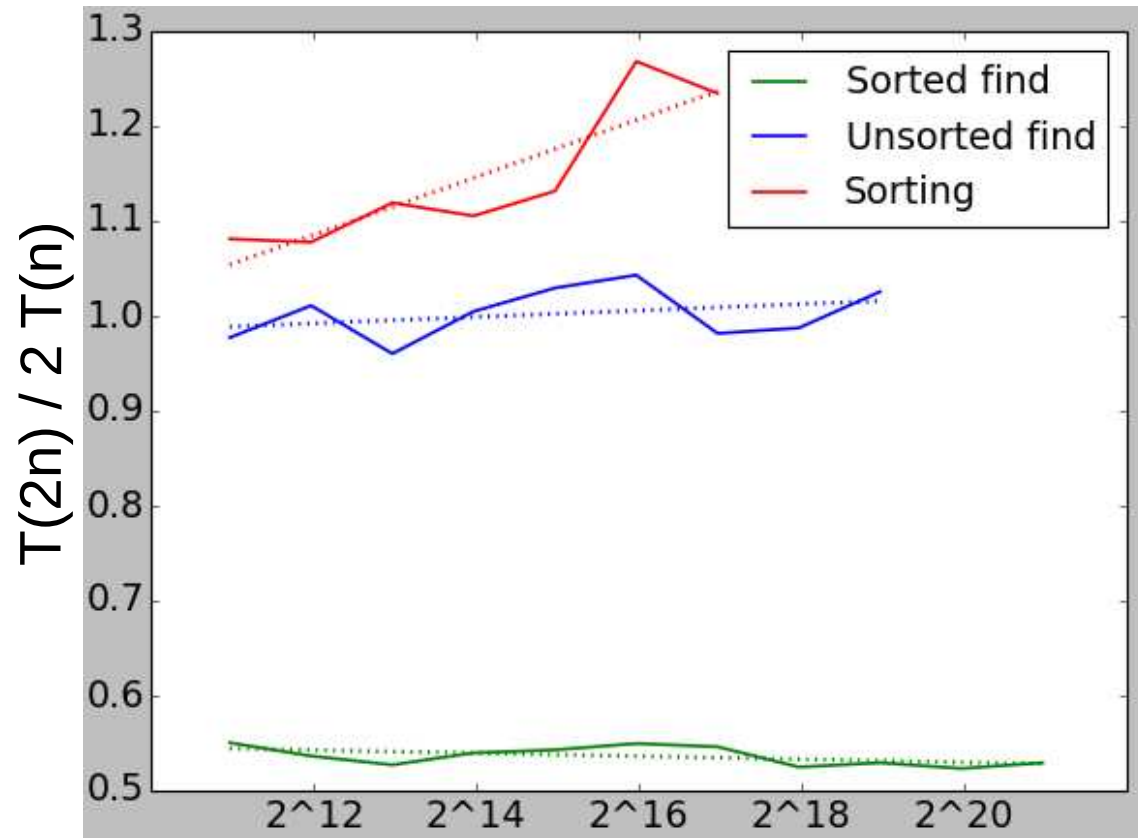
Any sorting algorithm that uses only pair-wise comparisons needs $O(n \log(n))$ comparisons in the worst case.



$$\log(n!) = \log(1) + \log(2) + \dots + \log(n) \leq n \log(n) \text{ for large enough } n.$$



Rate of Growth



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.

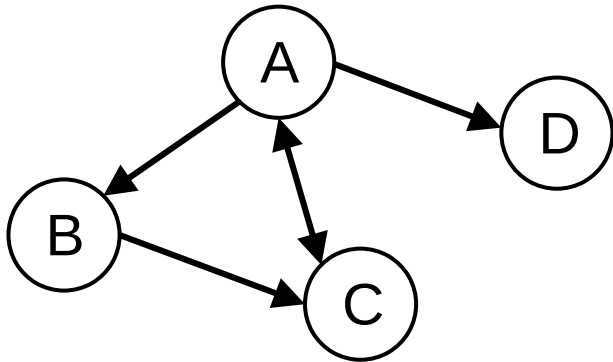
An impressionist painting of a garden path. The scene is dominated by vibrant green foliage and numerous roses in shades of pink, red, and white. In the upper right, there are clusters of bright blue flowers. The brushwork is visible and textured, creating a sense of depth and light. The overall composition is a lush, sunlit garden scene.

C03 Graph Traversal

Graphs and Trees
Traversal

Graphs and Trees

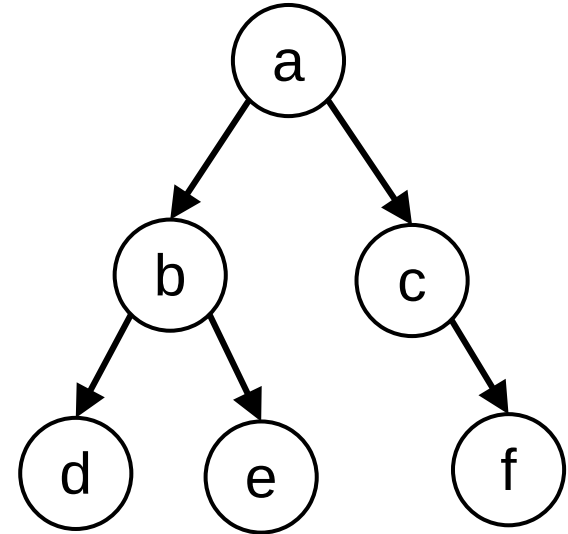
- A powerful abstraction in computing.



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)

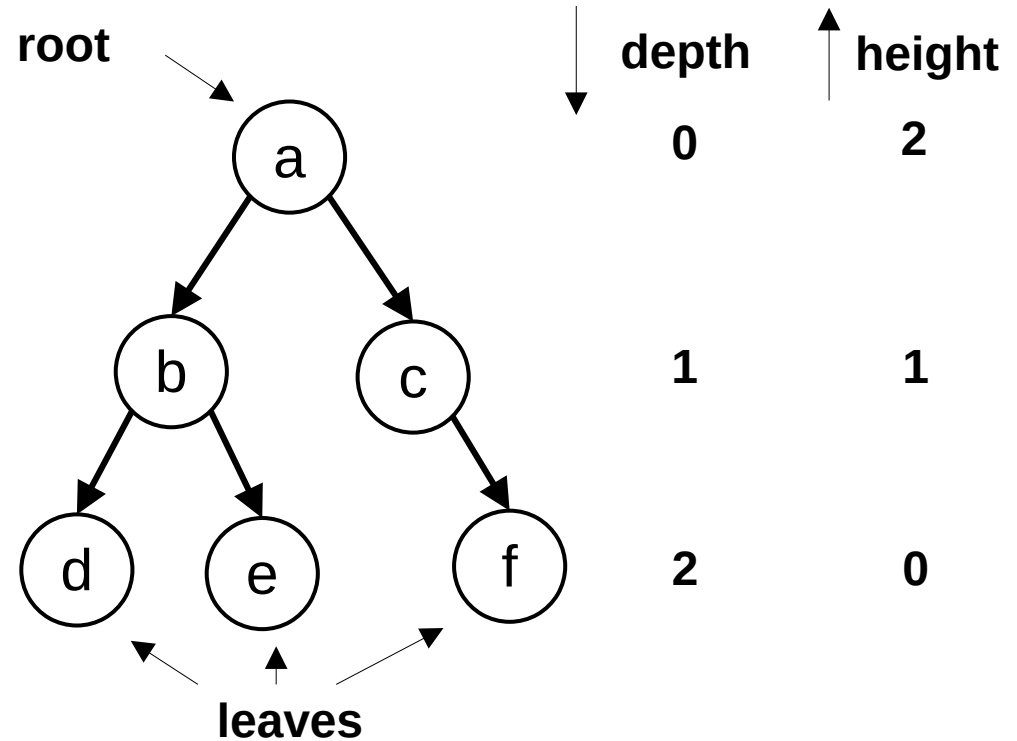
With ordering of children: *Ordered Tree*

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)



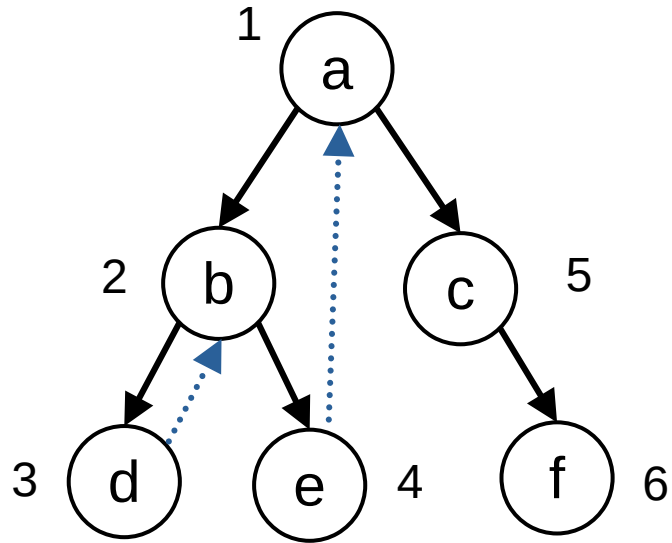
Traversal

- 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.

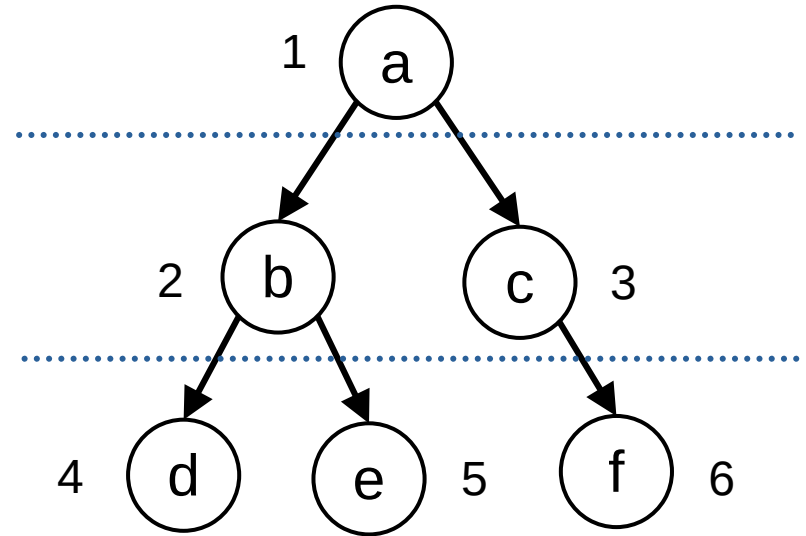
Tree Traversal

- 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



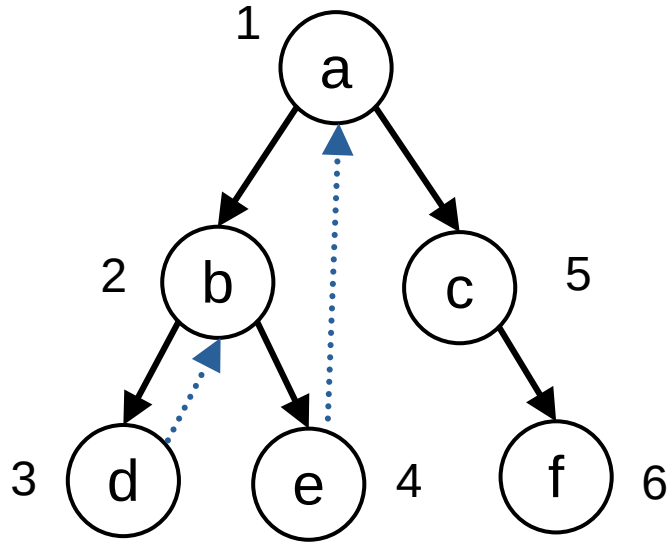
BFS traversal
a b c d e f

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



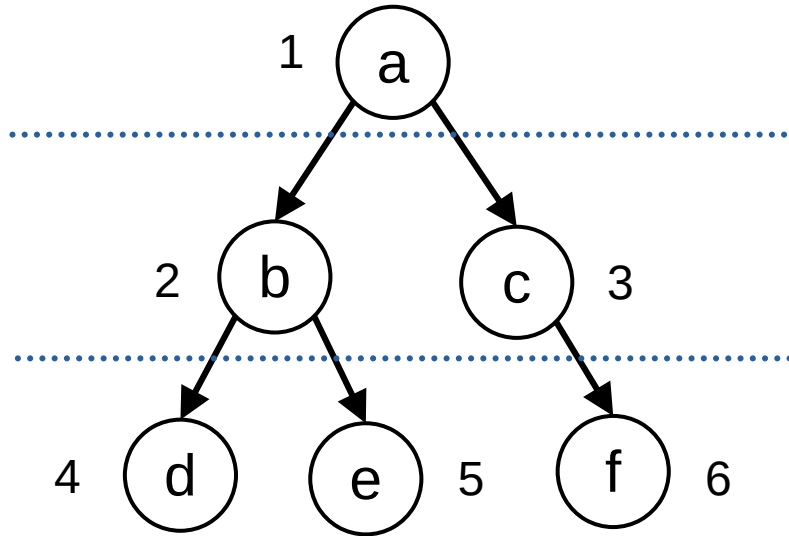
Pre-order DFS traversal
a b d e c f

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

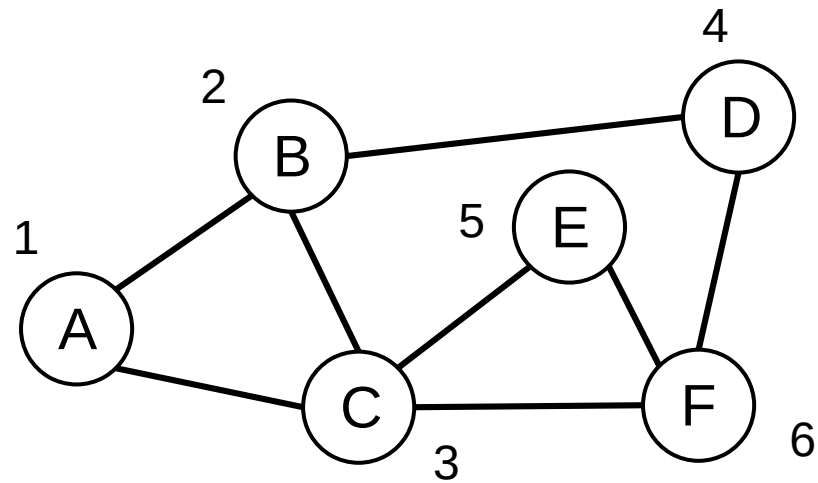
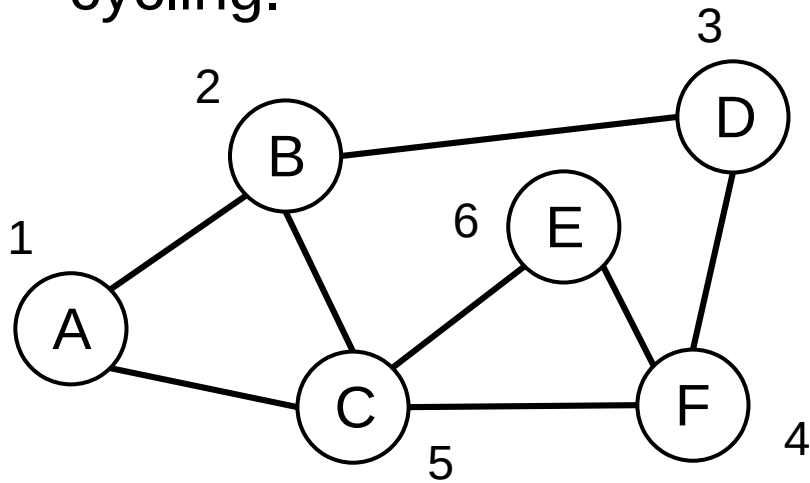
Queue { }: *enqueue* onto back, *dequeue* off front

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
```

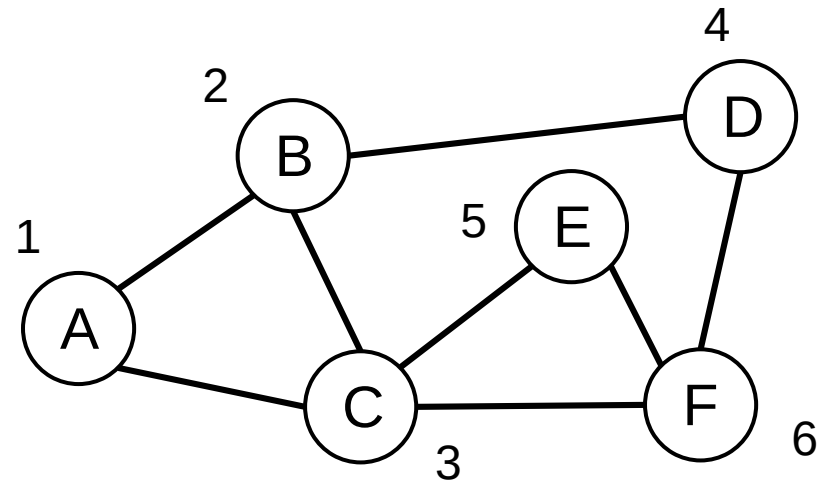
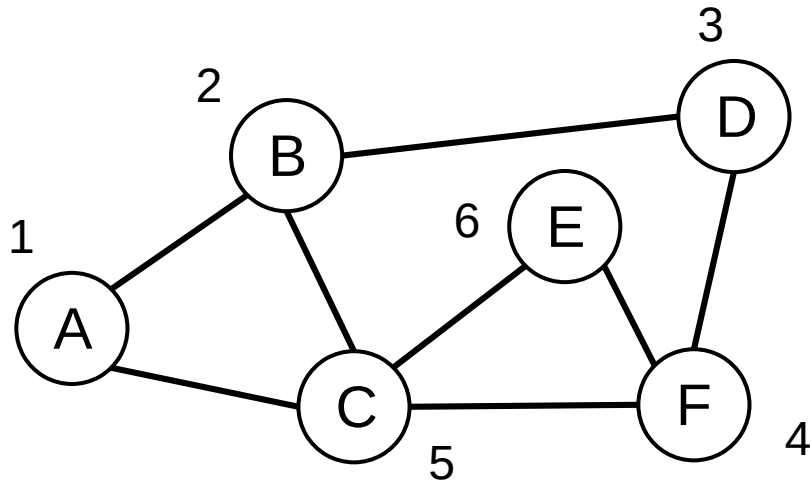
Graph Traversal

- 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





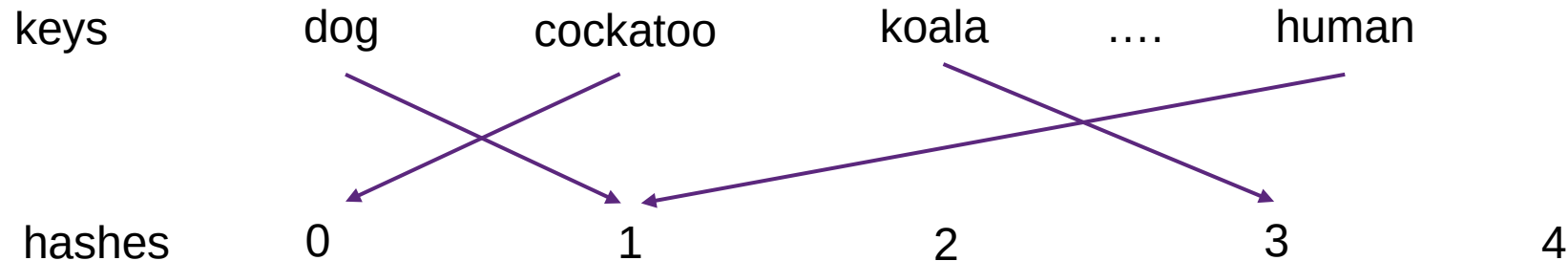
C04 Hash Functions

Hash functions
Choosing a good hash function

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 $k \in P$.



C05 Hashing Applications

Uses of hashing
Java hashCode()

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

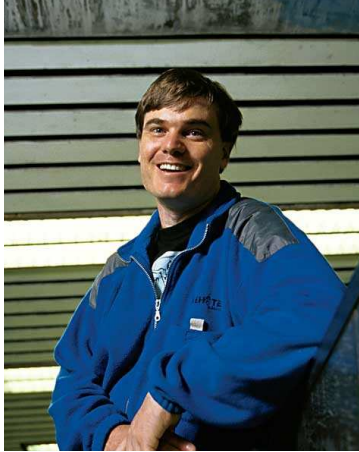
Used to check for transcription errors in credit cards (last digit checksum).



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

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.





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 `bytes` 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)	8364	
1000 0010 (130)		
1010 1100 (172)		

Buffering I/O

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

Disk: ~2-10ms **SSD:** ~10-100 μ s **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

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