**Lecture 1 -> Computer Architecture**

Many Data Structures & Algorithms are **specified** against an abstract machine and **optimised** for a specific architecture/implementation.

Programming is about manipulating information.

* Data structure are organised collections of data
* Algorithms are recipes for manipulating data
  + A series of operation that transforms our initial data into the desired form.

**Data Structures**

Specific data structures are strongly associated with. The architecture & organisation of computational systems. There is a wide range to implement a given architecture.

We have to know that many languages have similar, but not identical primitive datatypes.

**John Von Neumann**

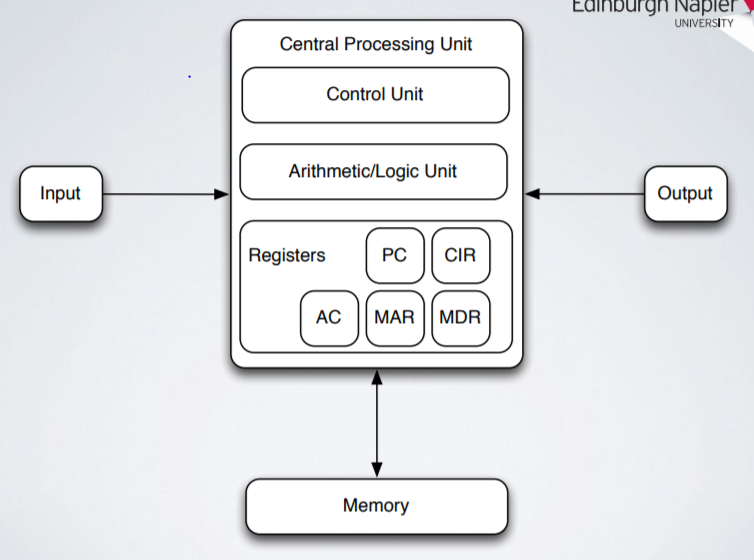
John Von Neumann introduces what is now referred to as the Von Neumann Architecture in 1945.

**Harvard Architecture**

**Princeton Architecture**

**EDVAC machine**

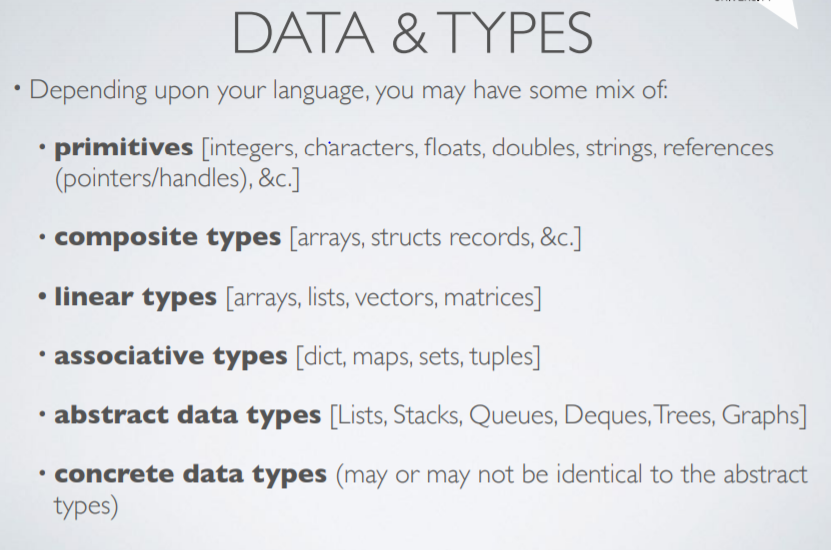
**Von Neumann Architecture**

It the design of a digital architecture made up of: central Processing Unit (**CPU**), Arithmetic Logic Unit(**ALU**), processor Registers, **Control Unit** (containing instruction register & program counter), **Memory** (storing data & instructions), mass storage, **I/O** Mechanism, all of them connected by a **bus** (communication pathways). 

Data us nived around on a bus. A computer basically works by moving information (data stored in memory cells, processor registers, various levels of cache, RAM) around and performing operations on it.

There is a limited bandwidth to move data from one location to anoter - if there is too much data then the limiting factor on speed is the bus - this is a known limitation of many computer architectures & is known as the Von Neumann Bottleneck.

Data is stored in memory. There are many types of hardware implementations of memory. Cells are arrange d in a regular pattern and of fixed size (think of an array). Eacjj cell can store a word (usuaully assume an integer per cell).



Pragmatism & Language Design

For many reasons, whilst a data structure has a theoretical shape the implementation must take account of practical & design issues:

* Avoid duplication
* Reuse existing data structures (stacks, queues, deques can are achieved using the Python list methods).
* Fitting with design of the language
* Optimisation

**C primitives**

Char, int, float, double. Actual size varies depending upon implementation (perhaps we'll investigate this in the lab). Each primitive data type has an associated **pointer** data type.

**Struct**

A **composite** type (in contrast to primitives) is an aggregation of multiple (potentially differing) primitive data types. It can contain pointers to other structured (used to build linked data structures).

**Sequential / Linear Structures**

A way to organise primitive data types in relation to each other - as various kinds of sequences.

Another composite datatype (in contrast to primitives). A linear data structure is made by elements form a sequence. They are always static data structures and it is a collection of values of the same type. Stored contiguously in memory.

**Array limitations**

Programming languages, particularly higher level ones, generally implemented using lower level languages. Often used within the implementation of more advanced data structures.

Algorithms can be used to do computation or process data. It is an unambiguous specification for how to solve a problem. An algorithm start with an initial state (and possibly empty input, proceed through a finite number of intermediate states by executing instructions and produce an output and termination state.

**Cost of computation**

There are cost involved in computation:

* Storing data uses memory.
* Finding data uses CPU
* Moving data around uses CPU & memory.

All these operations require **time (abstraction from CPU usage)** and **space** (abstraction from memory usage).

Data structures and Algorithms is concerned with evaluating & trading off between time & space usage.

real-world models of computation

[Topic 02](https://www.dropbox.com/s/nq7z703fc6ll24i/L02_algorithms%2Bcomplexity.pdf?dl=1) - Algorithms & Complexity

An algorithm is a list of instructions that can be followed to solve a problem. A;gprothms are used to calculate results or process data. Also Artificial Intelligence uses algorithms, as Path-finding, Machine Learning and Neural nets.

It is a finite number of exact finite instructions. It always produces a correct answer and it always terminater after a finite number of steps.

Bloom filters:

Instructions describe a computation. When executed there are a finite number of well-defined successive states that eventually produce an output and the computation terminates at a final ending state. Transitions between states need not be deterministic

Effective methods`:

QUIPUS

Big OH notation

Big Oh is a words for insight & practices that many progressional developers know & use (often without realising it).

Big Oh refers to the “order” associated with the performance the degree of complexity, os O(n) is read “The read of N”.

I really refers to the Order function. A function’s Big Oh notation is generally determined by how it responds to different inputs. A function is Big.

There are of course other notationS:

* Big Oh give the upper bound.
* Big Ω (omega) gives the lower bound.
* There is Big Ɵ (Theta) notations to asymptotically bound the growth to within constant factors above and below. Important because a single notation doesn’t always give the full story. Each notation can also be used to reason about best, worst & average cases.

A single notation doesn’t always give the full story. Each notation can also be used to reason about best, worst & average cases.

**Constant time**

An algorithm runs in constant time if it requires the same amount of time regardless of input size. Big Oh notation/ complexity is O(I). No matter how big the input will always take the same amount of time.

**Linear time**

An algorithm runs in linear time if the time it takes to execute is directly proportional to the input size.

Complexity is O(n), for example in:

* Array: linear search, Traversal, Find the minimum
* ArrayList: contains
* Queue: contains

Linear time approximately corresponds to the number of items in the array.

**Logarithmic Time**

If the execution time is proportional to the logarithm of the input size, we have a logarithmic complexity algorithm. A common attribute of the algorithm with logarithmic running times is that there is often a choice of a new element on which to perform an action & only one needs to be chosen.

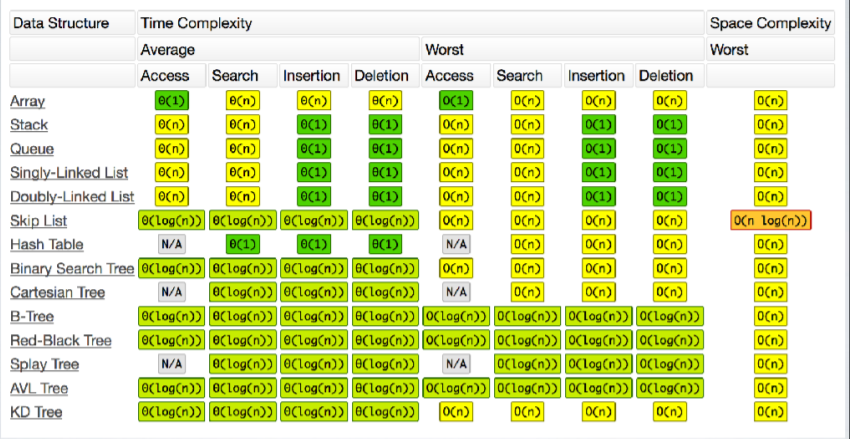
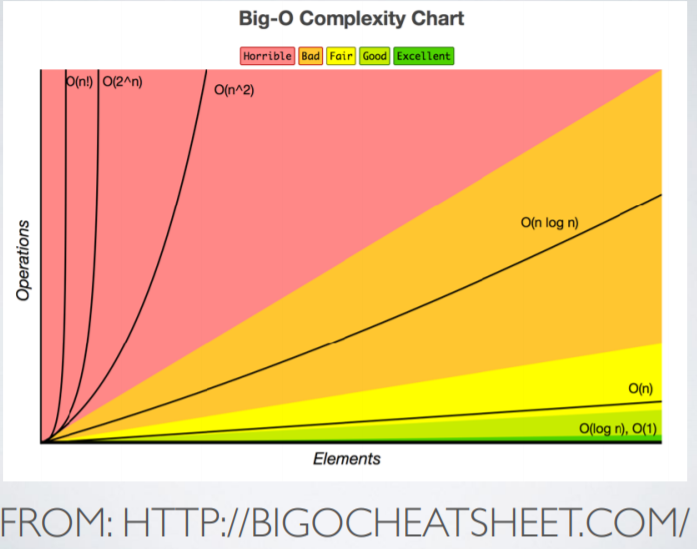
Ex: binary search.

Quadratic Time

An algorithm runs in quadratic time if its execution time is proportional to the square of the input size. For example, given a list [1,2,3], get back all combinations.

Quadratic time algorithm can be Bubble, selection & Insertion sorts.

For every item, n, in the list we have t do n operations. n\*n == n^2 O(n^2)

**segment tree:**

How programmers evaluate & optimise their code they will talk about things like. Input size & looping as an indicator of where a program will spend time computing. Profiling (using tools to determine where your program spends time).

Halting problem:

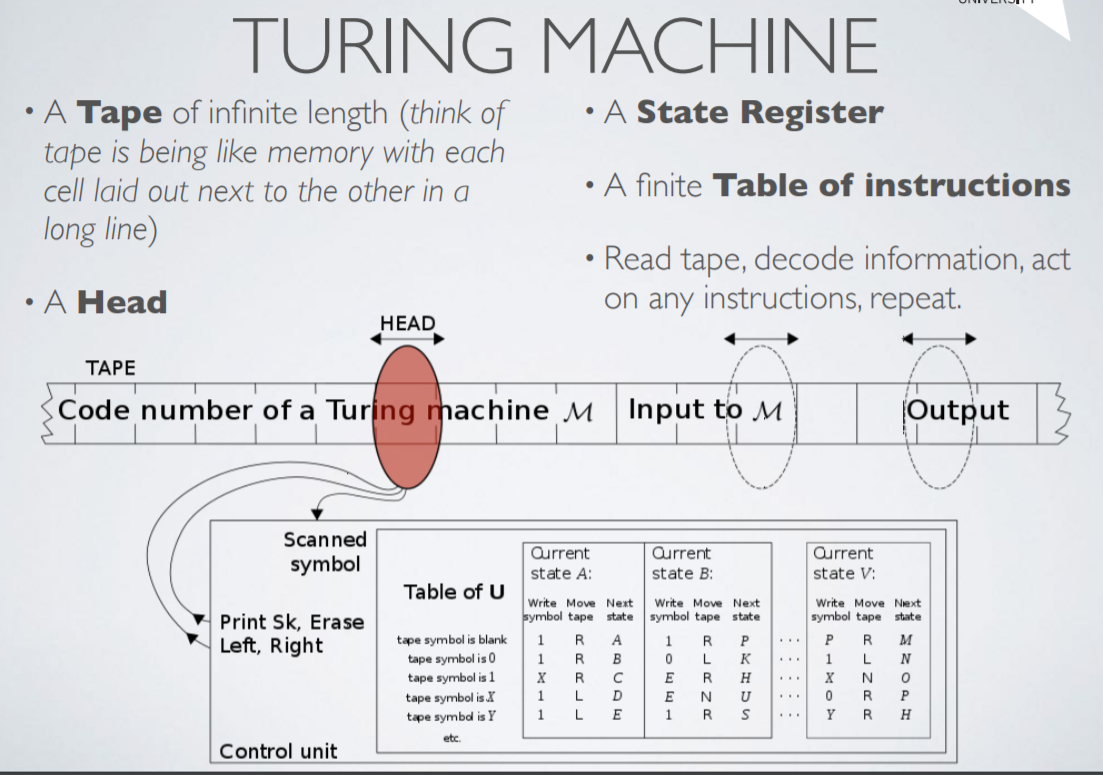
Turing proved that it is not possible to create a program solves (given an answer) tp the Halting Problem for all possible inputs.

Elements of the proof of this specified a mathematical definition of computer and program - This became the know as the Turing Machine.

Significance because of one of the first problems proven to be unsolvable.

**Turing ‘s model of Computations**

Theoretical mathematical model computation. An abstract machine that manipulates symbol on a strip of tape according to a set of rules. Recall we saw another model earlier in the V von Neumann Architecture.



Halting problem

Determining from a description of an arbitrary program & an input whether the program will finish running or continue forever. Been shown that not possible to construct a Turing machine that can answer this question.

Undecidable problem

Non-computable problem

An instance of a class of problems is called **decision problems.**

Limits of computation

**Features and function of the Turing Machine**

TL/DR ->

Not every problem is solvable using a computer. CS is all about working out what the characteristics and performance of problems that are (not) solvable by computers.

**Sequential Data structures -> Lectures 03**

We can organise data in memory so that the form of the organisation helps us to solve problems. These forms are **data structures** and their possible shapes are partly dictated by the computer architecture.

Method for organising, managing, and storing data for efficient access & modification:

* A collection of data (values, variables).
* Relationships between them
* Operations (functions that manipulate them).

Data and memory

Data stored in memory are often needed to model non-trivial problems. Want to group things together in useful ways.

Memory organised logically as a huge, sequence of buckets. Data of various type occupies various number of these buckets.

**Contiguous allocation:**

**Non-contiguous memory:** this only alternative with current hardware architectures is to allocate whatever memory we have non-contiguously wherever it can be located, then to keep track of that.

Many data structures are merely patterns for organising and tracking memory allocations.

**Types of data structure**

* Aggregates: structs, Unions.
* Linear(Sequential): Arrays, Linked Lists, Stacks, Queues.
* Linear (Associative): Dictionaries.
* Non-Linear(Binary):

**Aggregates**

A way to cluster arbitrary collections of data together so that they can be treated as a related group. In C, we use structs a lot to do this. Contents of a struct highly dependent upon specific programming problem. In other languages, these are data that we might group together in a class.

**Sequential Structures**

A way to organise primitive datatypes in relation to each other - as various kinds of linear sequence - branches.

Arrays are our core sequential data structure. Can use arrays to implement other kinds of data structure \9subject to the same limitation of arrays), using contiguous memory.

Sequential structures lead to two forms:

* **Contiguous**: Each element of the sequence is next to neighbour until you get to the end.
* Non-contiguous: each element of the sequence stores both its own data &metadata about the location of the next & possible also the previous element in the sequence.

Sequential APIS

If we have a sequence of elements then it shapes suggest natural ways to interact with it.

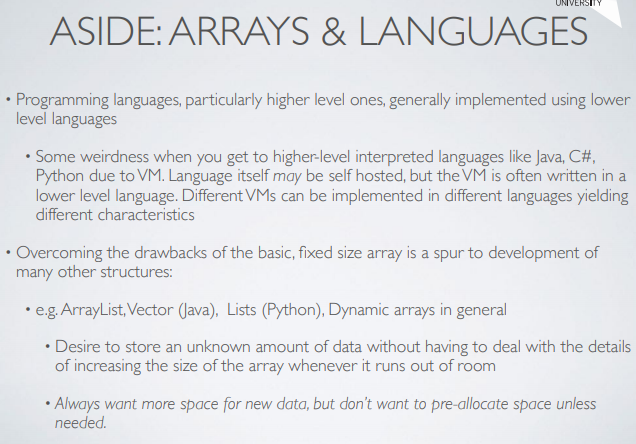
If we start at one end, that can move from one element to the next until we reach the other end. By restricting how we interact with the sequence we get the behaviour of various data structures, stack, queu, deque (logical variations on the theme of the sequential API).

**Non - linear structures**

Instead of pointing to next and previous, lets:

* Point to children - leads to a form of a tree data structure.
* Arbitrarily point to other elements (possibly including itself) leads to a graph (network) data structure.
* A tree is a sub-type of the graph in which there are no cycles.

Arrays a data structure based upon the contiguous allocation of memory. An array is a series of buckets that we can store our data in-however each bucket in the array is touching the next. We must keep track of how many buckets have (or risk disaster).



The list is a data structure that we can find in any language like Python, Java (called ArrayList/Vector), List (list)or Prolog(list). It is used like an array (but without some of the drawbacks). It is a dynamic data structure and we don’t have to manage allocation and deallocation.

Do they grow as required?

**The stack** is a collection that is based on the sequential structure so member elements are organised as a sequence. They have a well-defined interface. The stack has a top & button, items can be pushed on the top and popped off of the top also.

The last item in is the first item out (LIFO)- means that if we have a finite sequence of elements then it is trivial to reserve that sequence.

Stack doesn’t specify how the underlying collection should be implemented:

* An array is an easy approach - limited size, need reorganising.
* A linked list is an alternative - increased flexibility & complexity.

With the stack we can reverse a string, evaluating expressions (reverse polish notation for arithmetic and for some programming languages). Keep track of user actions, backtracking, looking for a path through a maze.

**Queues**

The queue has a front and a back. Elements are **enqueued**  (added to the back) or **dequeued (removed or service from the front).**

This lead to a natural order in how an element is dealt with. In a queue, the oldest element is dealt with first and the newest must wait its turn.

Leader to a FIRST IN, FIRST OUT (FIFO) data structure. Elements are dealt with in the order in which they arrive.

The queue is used for scheduling: CPU/HDD, buffers, communication buffers, particularly when data is transmitted asynchronously at a different rate to which it is sent. Job queu, store & replay order of moves in a game.

Circular Queue

One the complexities of an array implementation of a queue are that elements need to be moved up each time one is dequeued.

The queue is fixed in size so:

* Each item enqueue gets us closer to the end of the array.
* Each item dequeued leaves a gap between the start of the array and the first element in it.
* So must shift our elements along each time we dequeue so that the gap at the front is closed and there is more space at the end.
* Means dequeuing involves shifting a potentially large number of array contents.



The circular queue is also known as a ring buffer. Useful if your problem requires LIFO storage of a fixed size (some communication protocols).

Also useful in “producer-consumer problem” scenarios - if the consumer is unable to kep up then new data will overwrite older data which is discarded.

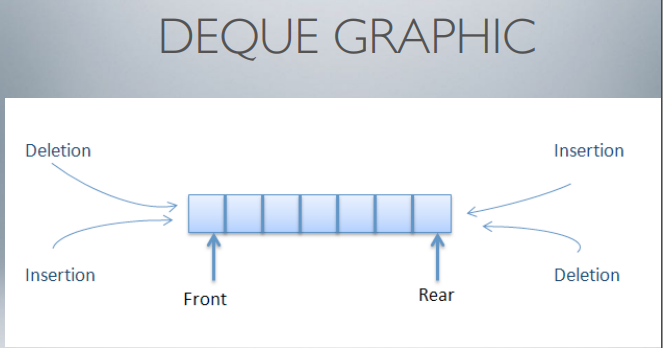
**Deques**

It is a version of a queue in which elements can be added to or removed from either end.

Can be further specialised by restricting this behaviour:

* **Input restricted:** Remove from either end but insert at one end only.
* **Output restricted:** insert at either end but delete from one end only.

A dequeue is a kind of archetype structure for the queue and stack. A list can usually support all the behaviors of a Deque (but generally has many more features thana dequeue should officially support).



Linked Lists

An alternative to the dynamic array that overcomes some of the limitations of the basic arrays but introduces a further complication, is the linked list.

A list created by linking discrete items of data together to that they are spread across the memory.

It starts with on item, the **head** of the list and each item **points to** the next item. Move through the sequence by accessing the current item & following the link to the next item.

Many structures can be implemented quite efficiently using a linked list.

* Singly linked lists: each element stores it’s own data and a pointer to their next elementSingly Linked List Graphic.
* Doubly Linked List: each element stores its own data and a pointer to both the next element the previous element. We can traverse the list in either direction. Head and tail immediately accessible (no traversal necessary) all internal elements require traversal.

Linked List characteristics

Constant time insertion and deletion - no reallocation or reorganisation needed because we’ve removed the contiguous constraint on memory layout.

Use more memory than arrays. No random access to or efficient indexing of data in the list.

Many common operations: getting the last node, finding a particular node containing specific data, locating insertion point for new data - require iterating through the list.

List, stacks, queue, and deques can all b implemented using a linked list instead of a dynamic array.

Tree:

* Head of the the list is the root node of the tree
* Each node point to further nodes.
* Depending upon the type of tree, each node point to 2 or more child nodes.

Graphs:

* Trees are a sub-type of graph.
* A graph is made of nodes but has no root node
* Each node can point to any other node.
* In some graphs nodesmight point to each other.
* In some graphs, nodes might even point to themselves.

Starting with a basic data structure (array), we added dynamism (list), restricted behaviours for modelling clarity(stacks, queues), and discovered an archetypal structure (Deque) which is essentially a sub type of List.

An important point here is that pragmatically a list and a Dequeue are the same but theoretically a Deque only has the operations of a deck, remove, peek(front/back) rather than the full complement of List functions.

**Lecture 04 -> Algorithms and searching**

Search paradigms are Linear search and Binary Search.

Data structures are, at least in one sense, static: we put data and we wait for someone to interact with it.

Without an algorithm to operate on them, data structures don’t really do much.

They are also the primary method through which we interact without data structures. We’ve already seen how algorithms define the construction procedures & APIs for our data structures.

Whilst algorithms are finite lists of instructions that solve problems, produce an output terminate. They atre also the primary method through which we interact with our data structures.

Classifying Algorithms

Algorithms aren’t just an undifferentiated collection of lists of instructions however, they can be classified on the basis of various characteristics.

These can be classified on the basis of various characteristics.

All the algoritmhs are classified in according with: the purpose, implementation, design paradigms, probabilistic / heuristic paradigm.

Heuristic paradigm:

Probabilist:

Design paradigm:

Each algorithm has a goal, sort data in ascending or descending order.

Algorithms can be classified on the basis of how they are internally organised. On the main implementational process that is used to get from the start state to the termination state.

Can anyone things of a way that we could sort our algorithms from labs so far on the basis of implementation?

**Implementation**

* **Recursive or Iterative:** algorithm repeatedly calls itself until a certain condition is matched. Common in functional programming (Haskell, Erlang).
* **Logical or procedural:** a logical algorithm has problem expressed in terms of axioms, to which rules are applied to deduce a solution (Prolog).
* **Serial or parallel:** serial algorithms execute one step at a time where's a parallel algorithm allows multiple steps to occur together (which can take advantage of multiple CPUs, Cores, Threads).
* **Deterministic or non-deterministic:** Deterministic algorithms solve problem sing a predefined process at each step whereas non-deterministic perform the best guess.

Approach to design

How an algorithm interacts with algorithms domain gives us another basis for classification.

How do we solve a problem? W have to think of the coursework specification.

**Design approaches**

* **Divide & Conquer:**  repeatedly / reduce the problem into smaller independent instances of same problem (usually recursive) until small enough to solve easily **binary search algoritmh.**
* **Dynamic programming:** if an optimal solution can be constructed from overlapping optimal solutions to sub-problems then can avoid recomputing solutions.
* **Greedy methods:** similar to dynamic programming but solutions ot sub-problems don’t need to be known at each stage. Instead, make a greedy choice of what looks the best solution at the moment algorithm.
* **Linear programming:** express problem as a set of linear inequalities then attempt to maximise or minimise the inputs.
* **Reduction (transform & conquer): s**olve the problem by transforming into another problem.
* **Graphs:** model problem as a graph then apply a graph exploration algorithm.
* **Machine learning:** usually data intensive + lots of different sub-approaches.

Probabilistic & Heuristic approaches

**Probabilist Search** - build a probabilist model of candidate solutions for a given search space.

**Genetic** - find solutions by taking inspiration from biological or evolutionary processes, define generation, randomly mutate, test against benchmarks.

**Heuristic -** find an acceptable approximate solution rather than an optimal solution.

**Searching**

Any algorithm to retrieve information stored in a data structure.Searching for & retrieving data becomes an acute problem at scale.

An algorithm used to find items that have specific properties within a collection of items.

Items might be:

* Basic data-types or objects.
* Database records or structs.
* Elements of a search space

The datum that defines the search target is known as the **search key or**, less frequently, **search term.**

**Search Strategies**

Search algorithms implement **search strategies.**

There are Many strategies, some are simple, other more complex, all with different characteristics.

Basic differentiation based upon the data being searched:

* Is the data organised in any way - if it is then this influences how we can approach finding a particular element with the collection.
* What else for we know about the data? Sometimes the nature of the data and it’s sorting can give us a little boost in our search strategies.

**Sorted vs unsorted**

Strategies that work on unsorted data are simple and robust but can have very **poor performance** characteristics. Most efficient search strategies rely on **sorted data.** This is why we often talk about searching & sorting though they are inextricably linked.

**Linear search**

The **simplest** search algorithm is the **linear search.** list need not be ordered. It is a brute force or exhaustive search. It check each element of list in sequence until either you find the desired element or the list is exhausted.

Trade-off between set-up/sort time & search time for other algorithms versus cost of searching with linear search.

**Binary Search**

It finds the position of a target value within a sorted array. Itcompare target value to value of middle element of array. If target = element then search is finished.

If target < element then repeat in lower half of array.

If target > element then repeat in upper half of array.

Target will either be found or not found. Each iteration will half the search space.

**Search algorithm selection**

Linear search or binary search are usually the basic choices for searching. If small amount of unsorted data or time taken to sort data will exceed time taken to search through it then a **Linear search** may be appropriate, otherwise, **Binary search.**

**Jump search**

Jump search algorithms are algorithm for sorted arrays. It check fewer elements by skyping a fixed number of elements instead of searching all elements.

Because array is sorted or comparison tells us if we have jumped too far.

The time complexity is O(rad(n)) which is between linear search O(n) and Binary Search O(Log n).

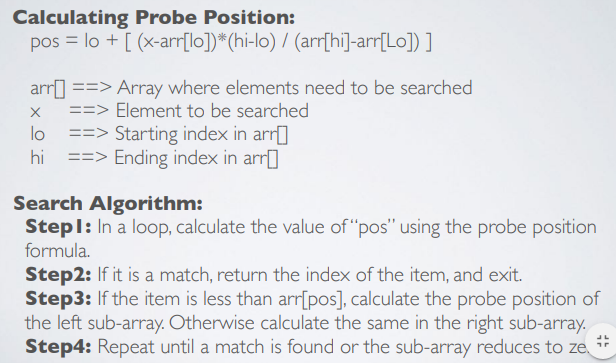
**Interpolation search**

Sorted arrays (notice a pattern yet). If values are uniformly distributed then this can be an improvement over binary search.

Does binary search always goes to middle element?

What if the element are spari or dispari.

Probe position:



**Exponential search**

Misleading name (actually runs in O(Log n) time). Useful when you need to search unbounded data, size of array is infinite.

Can work better than Binary Search in a bounded array when the target is closer to the first element.

**Ternary search**

Ternary search algorithm is composed of three parts.

An uncommon form of divide & conquer. It is slightly less efficient than binary search.

2Log3(N) versurs Log2(n)

Ternary search does more comparisons than binary search in worst case.

**Summing UP searching**

We only really have to ways of searching:

* Entire collection if it unsorted (linear search).
* Divide & conquer (binary search) if collection is sorted.

Tree and graphs can give us additional ways to access our data.

Minimal spanning tree is essentially a search for a set of nodes such that the sum of the edges connecting them is minimised.

Minimal spanning tree:

Route finding:

Linear search & Binary Search (+ some variations).

The biggest efficiency when searching is found when dealing with sorted data. If wwe can sort our data then we can retrieve items more swiftly.

Assuming that the sort procedure takes less time than a linear search.

**Lectures 05 -> Searching & sorting**

Computers have probably spent more time sorting data than performing any other task.

Common sorting algorithms:

Sorting algorithm: puts elements of an **unordered collection** into a **particular order.**

**Bubblesort**

A bubble sort is a type of comparison sort. It is simple but so inefficient that it is not considered practical particularly as data size grows.

On each pass, sorts largest value to end of collections so, sorted list grows back towards beginning on each iteration.

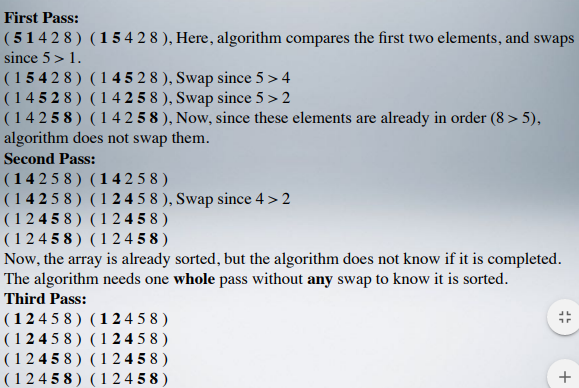
It compare first pair of items. If first element is great than second then swap them. Get next pair of items (shift windows along so that second item of first pair is now first item of second pair).

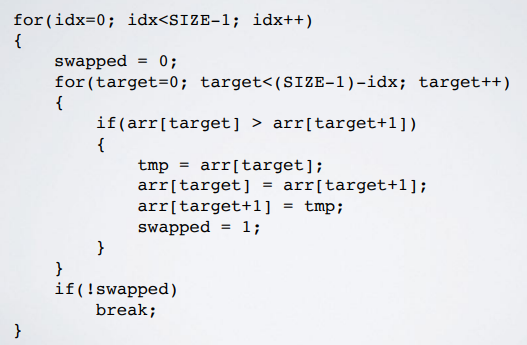
If first great than second then swap them, repeat until end of data is reached. Repeat from beginning again until one pass through without any swaps.

It is very sensitive to size of dataset. Positions of elements play a large part in performance. Best case scenario: data are already sorted: O(n).

Do a single iteration through the data. In the worst case scenario, data in reverse order: O(n^2). Small elements at end take many passes to bubble through to the other end.

<https://www.youtube.com/watch?v=lyZQPjUT5B4>





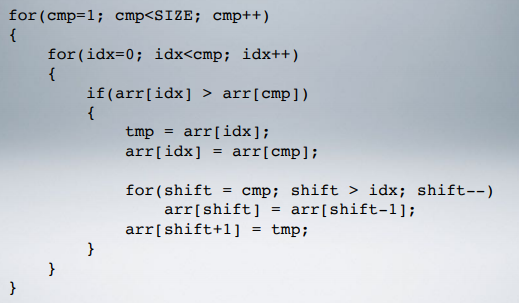
An other optimisation could be track of position of last swap in an iteration and put the largest item to final location at end.

If smallest element is at the end then this can take n-1 passes to get to it’s position.

**Insertion sort**

Relatively efficient on small lists and lists that are mostly sorted. Constructs sorted list one element at a time but not final sorted list.

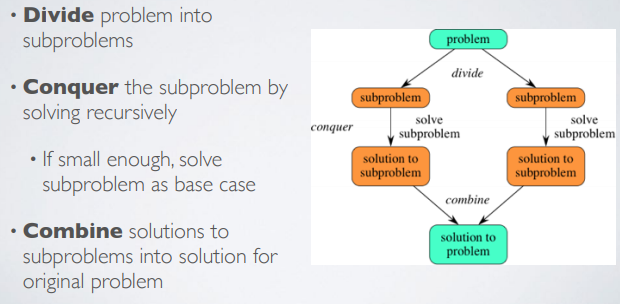
Remove element from input, shift everything along until the right position is found.

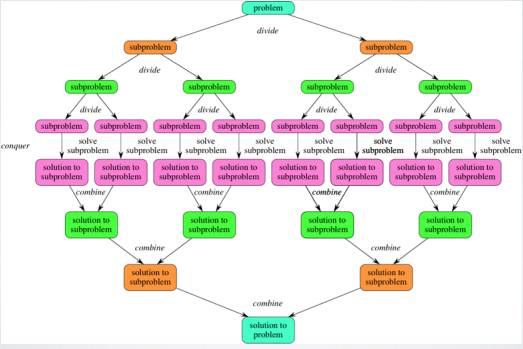


<https://www.youtube.com/watch?v=ROalU379l3U>

Sorting in-place:

Sorting out-of-place:





Merge sort & quick sort have better running times.

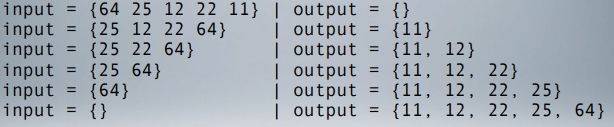
**Selection sort**

It is inefficient on larger list and generally performs **worse** than insertion sort. It needs no additional storage beyond input array and loop counters. It constructs a **final sorted list.** Agter n iteration, at least n elements are in their correct final place.

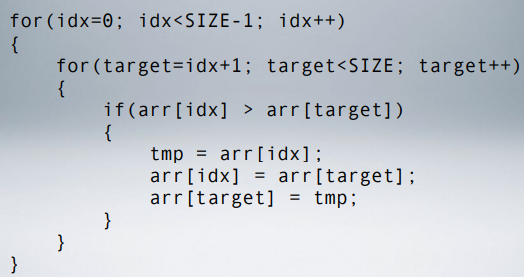
It iterate over the input until you find the smallest value:

* Append the selected value to the output.
* Repeat.

The selection sort append the smallest element to the output and continue until input is empty and output contains sorted collection.



<https://www.youtube.com/watch?v=0-W8OEwLebQ&list=PLOmdoKois7_FK-ySGwHBkltzB11snW7KQ&index=6&t=0s>



**Merge sort**

**Old** - invented in 1945 by von Neumann.

We divide up the elements of the collection & at most compare each element to haf the collection.

O(n log n) -> performance in best, worst & average cases.

<https://www.youtube.com/watch?v=XaqR3G_NVoo>

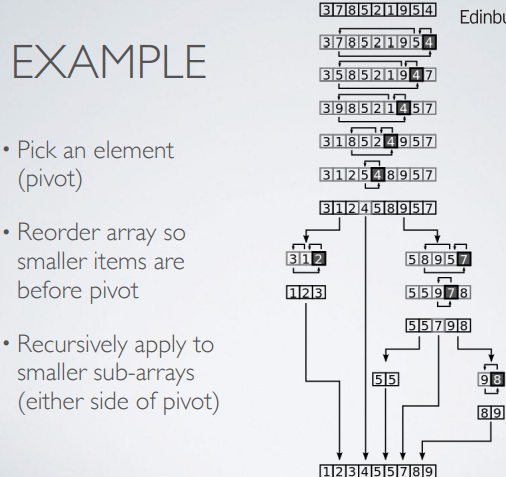
**Quicksort**

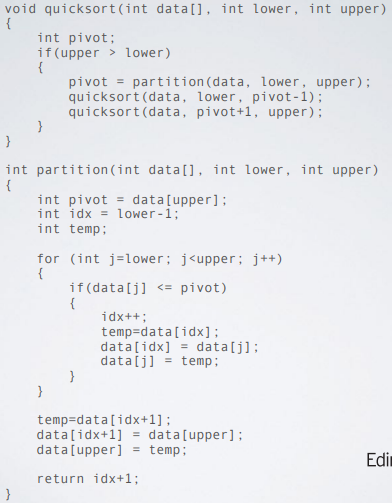
It sort items of any type where there is a less than relation defined. It can operate in place on an array so sorts can be very space efficient.

Divide & conquer algorithm that relies on a partition operation that is oriented around a **pivot value**.

Elements smaller than pilot are moved before it and elements greater are moved after it. Lesser & greater sublists then sorted recursively.

1. We pick an element from the dataset. This is element is the **pivot.**
2. Reorder the dataset so that elements greater than the pivot are to one side of it, and items less than the pivot are to the other side of it. Once complete, the pivot value will be in it’s final sorted position.
3. Repeat, applying the pivot selection & partition operation to each separate collection formed respectively from the values smaller than and greater than the pivot in the previous iteration.





<https://www.youtube.com/watch?v=ywWBy6J5gz8>

15 sorting algorithms in 6 minutes -> <https://www.youtube.com/watch?v=kPRA0W1kECg>

**Sorting & searching**

Efficient sorting can be important for optimising the use of other algorithms.

**Simple** (but inefficient): Bubble sort ( & variants).

**Simple** (efficient on **small data**): Insertion & Selection sorts.

**Efficient:** heap, Merge, QuickSort.

Sorting is a solve problem. Tim sort (2002) and library sort (2006), are new algorithms still being developed.

Why do when to sort data? We need to sort data because it will be easier to find items. It is useful canonicalising data - putting into a standard from for human output.

Preprocessing Can lead to faster search. Algorithms can be very simple if they are dealing with sorted data.

Conditions

Itis expected that a sorting algorithm will also satisfy the following conditions:

* Output in non-decreasing order.
* An element is no smaller than the previous element according to the desired ordering. This allows equivalent size items to exist in the same collection.

It is expected that a sorting algorithm will also satisfy the following conditions: output in non-decreasing order.

An element is no smaller than the previous element according to the desired ordering. This allows equivalent size items to exist in the same collection.

Out is a **permutation.** This is a reordering of the collection that retains all of the original elements.

**Comparison relations**

Varios comparison relations for data can make sense. But Numeric and lexicob graphical order are obvious choices for many sequences, strings and tuples.

There can be various lexicographical ordering for strings.

**Stable sorting**

Sorting algorithms can be distinguished on the basis of whether the sorts that they generate are **stable**ds

Whether repeated elements in the input appear in the same order in the output. This doesn’t have to be a problem but can be important when considering real-world problems:

-Two members of the input can sort equivalently but actually refer to different things - sorting doesn’t occur in isolation from the rest of the world.

**Bogo sort**

Bogo sort, also known as slow, **stupid or monkey sort.**

Generate all permutations of your data until you get the right (sorted) one. It has really bad performance: **O((n+1)!)**in average & worst cases.

Relies on the idea that there is some probability of getting the right permutation at each try.

While the collection is not in order: shuffle it or iterate it.

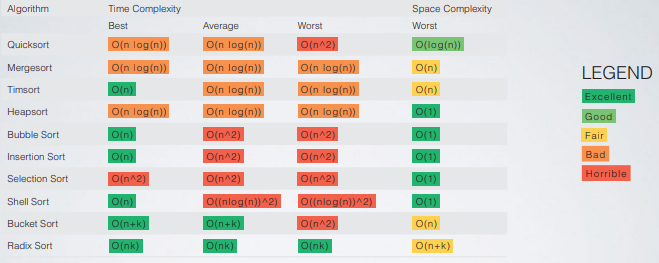
**Bad sorts**

Bad sort is not useful. But an interesting starting place for considering other sort algorithms.

**Bozo sort:** if list is not sorted, we pick two random items & we swap them.

**Gorosort:** if list is not sorted: randomly permute a subset of the collection.

It can be morefficient to find data within a pre-sorted data-structure (but sorting that data in the first place also has a cost).



It can be more efficient to find data within a pre-sorted data-structure (but sorting data in the first place also has a cost). Smaller problems are easier to solve than bigger problem (but knowing how to brek a problem up makes all the difference).

Finished all the 5 presentations about the lectures online (in the website of the demostrator).

Lecture 4 -> Simone POwers -> 07/02/2020 -> Linked LIst

Why implement specific data structure using existing libraries?

* Code already tested
* Libraries provide also APIs implementation for specific data structures
* LIst: countable number of ordered values, where the same value can occur more than once.
  + Apis available:
    - Constructor (create empty List)
    - Is empty
    - Insert to front and remove
    - Append to end and remove
    - Insert at index
    - Access element at index.
* Issue with the array-based implementation:
  + Remove a specific element probably in the middle of the array, it would take lots of resources to shift the part of the array after the element to remove.
  + Need to have sufficient memory free in contiguous cells.
* What if we don’t have to store data data items in contiguous cells memory?
  + We use a Linked List. D.S. that uses pointers(just a memory address).
  + The pointer to the end of the Linked lIst is NULL (address 0 of the memory). If we try to access to the memory location 0 (NULL), program will crash immediately on linux, while On windows we will have an error dialog.

Size of a pointer:

* 4 bytes i we are in a 32bits architecture.
* 8 bytes if we are in a 64 bits architecture.

Access by:

* Index: it’s structure supports direct indexing. Arrays are useful kust iff somethin has stored the index
* By value:
* LinkedLIst APIs complexity:
  + Append (always at the end : o(N)). We can potentially have O1 if we keep constantly track of the tail (end of the LnkedList) with another pointer.
  + Insert: O(1): we have to modify just two pointers instead to shift an Array.
  + Search: O(N)
  + Remove: Search TIme + 0(1).
  + Lindell list work poorly wit h the caches of the CPU. Array have optiomal locality of reference and so work well with CPU caches. Rememner that the Big-O notation says how performance scales with increasing list size N;

Lecture 5 -> binary Search tree

* A tree
  + Consist of nodes, where at the top level there is the root Node.
  + Each node has one parent, but a node can have many children.
  + Branching factor: the max number
* A list, stack or queue is a linear structure that consists of a sequence of elements. A binary tree is a hierarchical structure. It is either empty ot consists of an element, called the root, and two distinct binary trees, called the left subtree and right subtree.

Struct Node { class Node {

Int data; public int data;

Node \* left; public Node left;

Node \* right; public Node right;

}; };

* A tree is recursively composed of many smaller subtrees.
* Visit a tree recursively:
  + Preorder traversal
    - Visit the root
    - Visit the left subtree
    - Visit the right subtree
  + Inorder traversal : 20, 39, 42, 44, 60, 67, 75
    - Visit the left subtree
    - Visit the root
    - Visit the right subtree
  + Postorder traversal 20, 42, 39, 60, 75, 67, 44
    - Left subtree
    - Right subtree
    - Visit the root
* Height of a tree:
  + If the tree is null, height = -1;
  + The height of a tree with two levels is 1.
    - Write a recursive algorithm to calculate he height of a tree.

Int height (Node root)

{

If (root == NULL)

Return -1;

Int hl = height(root.left);

Int h2 = height(root.right);

Return 1 + Math.max (hl , hr);

}

Binary Search Tree

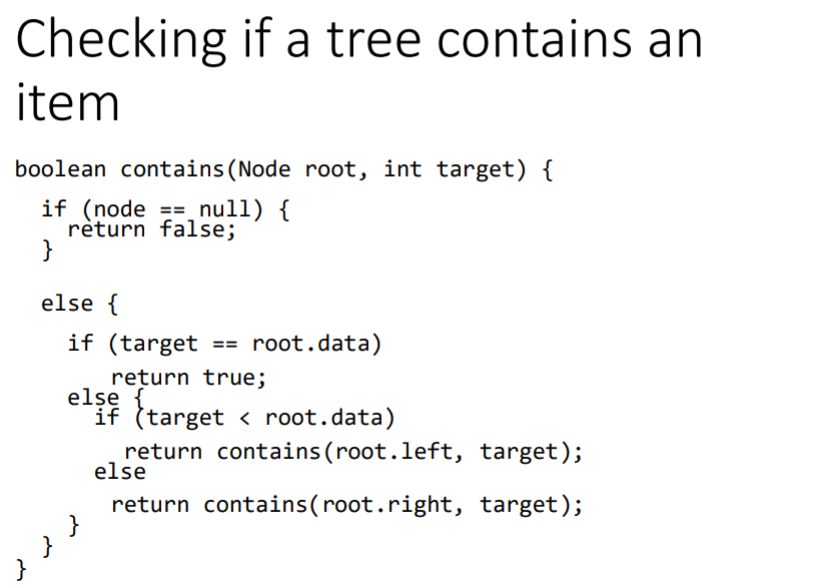
[https://liveexample.pearsoncmg.com/dsanimation/BSTeBoo k.html](https://liveexample.pearsoncmg.com/dsanimation/BSTeBoo%20k.html)

Create something like this

BST:

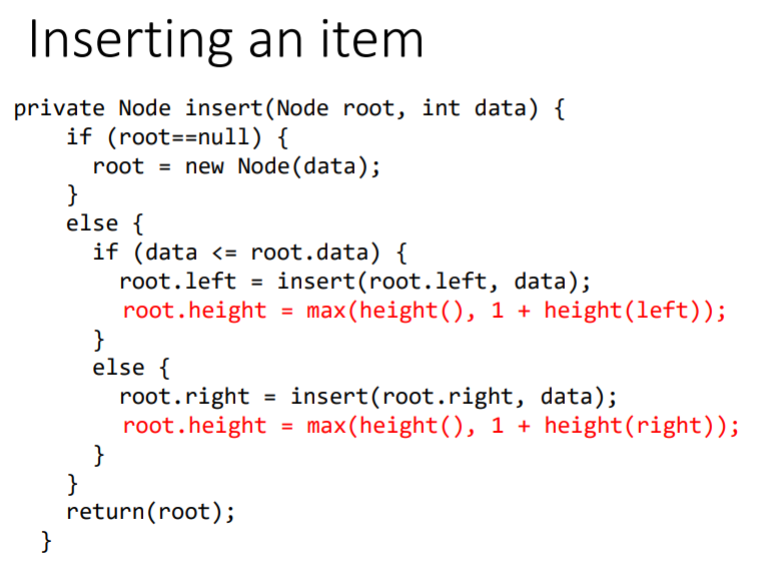
* Organize data for efficient searching.
* Everything to the left of the root is smaller than it.
* Everything to the right of the root is greater than it.

Checking if a tree contains a specific item.



Inserting an item in a BST:

* If a BST is empty, create a root node with the new element.
* Otherwise, locate the parent node for the new element node
* If the new element is less than the parent element, the node for the new element becomes the left child of the parent.
* If the new element is greater than the parent, the node for the new element becomes the right child of the parent.



Delete an elements in a BST

* To delete an element from a Binary Tree, we need to first locate the node that contains the element and also its parent node.
* The current node may a left child or a right child of the parent node. There are two cases to consider.
* What are the cases when deleting a Binary Search Tree?

Data compression Huffman Coding

In ASCII, every character is encoded in 8 bits. Huffman coding compresses data by using fewer bits to encode more frequently occurring characters. The codes for character are constructred based on the occurrence of characters in the text using a Binary Tree, called the Huffamn codign tree.

Construct Huffman Tree:

* Begin with a forest of tree. Each tree contains a node of character. The weifht of the node is the frequency of the character in the text.
* Repeat this step until there is only one tree
  + Choose two trees with the smallest weight and create a new node as their parent. The weight of the new tree is the sum of the weight of the subtrees.