The Ultimate Rubix Cube – A COMP2521 Notation

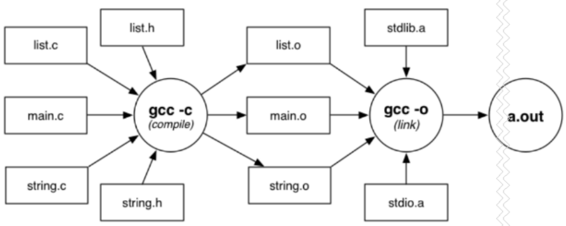
Analysis of Algorithms – *Enter the Matrix*

- **Algorithm**
    - A step-by-step procedure
    - Completes in a finite amount of time
- The main concern is time complexity
    - Ie the mathematical factor by which our time-taken is proportional
        - Number of steps of the algorithm
    - In order: 1, $\log(n)$, n (linear), $n\log(n)$, n^2, e^n
- Our analysis
    - Focus on worst case running time
    - We could take empirical measurements of time, but:
        - This may be difficult
        - Results may vary greatly based on input
        - Different computers get different results
    - Theoretical analysis
        - Characterises running times as a function of the input size, n
- Time-complexity
    - How to write pseudocode
        - Control flow
            - If…then…[else]…end if
            - While...do…end while
            - Repeat…until
            - For[all][each]…do…end for
        - Functions
            - f(arguments)
            - Input…
            - Output…
        - Verbal descriptions of simple operations is fine
    - We can write pseudocode and analyse the maximum number of primitive operations per step
        - We could be exact: "for all i = 1…n-1 do…" is $n + (n-1)$ operations
        - But to be honest we just care that it's linear, ie $O(n)$ (of order n)
    - Analysis strategy
        - Find the order (big-Oh) of all the lines and take the maximum to be the time complexity of the code
    - $\log(n)$
        - Problems in which our while loop continually halves x, etc
        - Eg binary search (finding a number by halving the interval)
    - **Relatives of $\boldsymbol{O(f(x))}$**
        - $\Omega(f(x))$ a lower order bound
        - $\theta(f(x))$ there are two constants that exist, and we are bound in between

- Complexity classes
    - P vs NP
        - P - Problems for which an algorithm can complete in polynomial time
        - NP - Problems where no such algorithm is known (nondeterministic, polynomial time)
    - Difficulty
        - Tractable – have polynomial-time algorithm
        - Intractable – not tractable
        - Non-computable – no algorithm can exist

- Generate and test algorithms
    - If the following criteria are met:
        - It is easy to generate new states
        - It is easy to test if a new state is a solution
    - We use generate and test when we are guaranteed to either find a solution or know that none exist

Compilation and Makefiles – *Build It*



- Compilers
    - Convert program source code to executable form
        - "Executable" – machine code or bytecode
    - **gcc compiles source code to produce object files**
    - **gcc links object files** and libraries to **produce executables**
        - Eg: gcc -c slaughter.c
            - Produces slaughter.o
            - -c means compile
            - -o means make executable
        - Eg: gcc -o murder kill.o slaughter.o
            - Links kill.o and slaighter.o to produce the executable program murder
- Makefiles

```
game : main.o  graphics.o  world.o
        gcc -o game main.o graphics.o world.o

main.o : main.c graphics.h world.h
        gcc -Wall -Werror -c main.c

graphics.o : graphics.c world.h
        gcc -Wall -Werror -c graphics.c

world.o : world.c
        gcc -Wall -Werror -c world.c
```

```
target : source1 source2
```

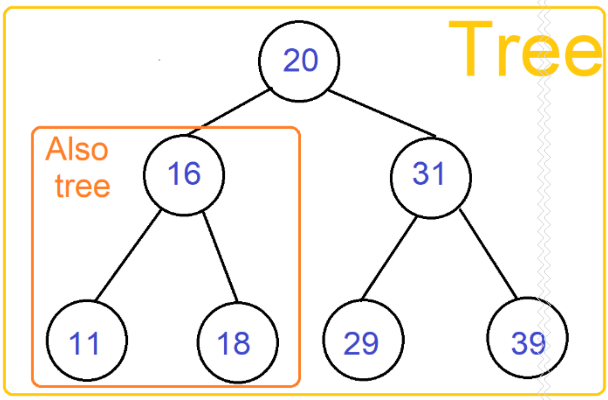
- Specify dependencies
    - Target should be built from sources
    - Target is dependent on sources
- They specify rules
    - Eg rebuild target if it is older than any source
- make world.o
    - Only builds that target
- make
    - Builds the first target in the Makefile

Abstract Data Types – *Kandinsky's Computer*

- Data type
    - **A set of values**
    - **A collection of operations on those value**
    - Eg C strings
- Abstract data type
    - An approach to implementing data types
    - Can have multiple instances (set A, set B, set C, …)
    - **Separates interface from implementation**
    - **Users can only see the interface**
    - **Builders provide an implementation**
    - Eg C files
- Generic abstract data type (GADT)
    - Can have multiple instances and types (set<int>, set<char>, …)
- **Interface**
    - Provides a user-view of the data structure (eg FILE*)
    - Provides function prototypes for all operations
    - Describes all operations
    - Provides a contract between ADT and clients
- **Implementation**
    - Gives concrete definition of data structures
    - Defines all functions for all operations
- Collections – many ADTs consist of a collection of items
    - May be categorised by structure
        - Linear (list), branching (tree), cyclic (graph)
    - May be categorised by usage
        - Set, matrix, stack, queue, search-tree, dictionary

- The Set ADT as bit-strings
    - Each word is (say) 32 bits
    - Each value is represented *by the position of a word* in a large array of bits
    - Union and intersection become easy – simply a bunch of bitwise operators

Trees – *They Are Us*



- Trees are a data structure
    - Some data (eg an int value)
    - Two pointers
        - "Left" and "right"
        - Each point to a tree (or NULL)
- Definitions
    - **Height**: maximum steps from first node to lowest
    - **Level**: the number of steps from the first node to this node
        - We start at 0 and move down

        - Eg opposite: 20 is level 0, 16 31 are level 1, 11 18 29 39 are level 2
    - **Balanced**: describes a tree with relatively even levels
- Binary search trees
    - For an array
    - "Left" leaf value is smaller than the root value
    - "Right" leaf value is greater than the root value
    - We can search for a node with order $\log_2(n)$
- Inserting into a tree
    - Option 1: just move through the list and insert where expected
    - Option ?: a more optimal method, ensuring compact trees possible
- **Counting nodes – RECURSION**

```
if (t == NULL)
   return 0;
else
   return 1 + TreeNumNodes(left(t)) + TreeNumNodes(right(t));
```

- **Printing a tree – RECURSION**
    - [Notice that reading all nodes *STRICTLY* from left to right is ordered]

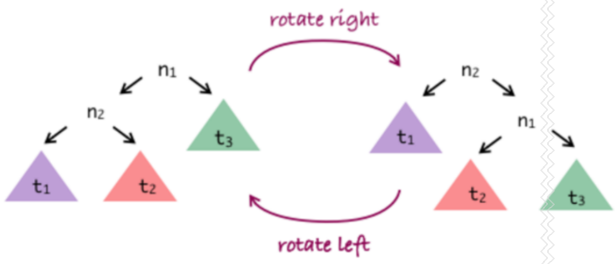
```
if (t == NULL) return;

BSTreeInfix (t->left);
showBSTreeNode (t);
BSTreeInfix (t->right);
```

- **Joining two BSTs** (given one is strictly greater than the other)
    - Find the min node in larger tree
        - Remove it
    - Make that the new tree
    - Lower tree is t->left
    - Upper tree is t->right
- **Deleting from BSTs**
    - If our node has no subtrees, just remove
    - If our node has one subtree, remove and set the node to be its subtree
    - If our node has two subtrees, remove and join the two subtrees, set the node to be the result

- **Rotation**
    - Left rotation algorithm

```
Tree rotateLeft(Tree n2) {
   if (n2 == NULL || right(n2) == NULL)
      return n2;
   Tree n1 = right(n2);
   right(n2) = left(n1);
   left(n1) = n2;
   return n1;
}
```



Note: since we return n1, this means we can rotate any subtree within a larger tree and still maintain proper tree order

- Insertion at root
  - With left and right rotation, we can move nodes up and down levels
  - With the right rearrangements, we can insert any node at the root
  - Useful if recent items more likely to be searched
  - This has the *tendency* (no guarantee) to be reasonably balanced
- Balanced trees
  - Goal: Min height = min worst case search cost
    - Balanced: $|\#height(LeftSubtree) - \#height(RightSubtree)| \leq 1$
      - For every node
    - Height of $\log_2 N$
  - Strategies to improve worst case search
    - Randomise – reduce chance of worst-case scenario
    - Amortise – do more work at insertion to make search faster
    - Optimise – implement all operations with performance bounds
- Balancing

```
Tree rebalance(Tree t) {
   int n = TreeNumNodes(t);
   if (n >= 3) {
      t = partition(t, n/2);           // put node with median key at root
      left(t) = rebalance(left(t));    // then rebalance each subtree
      right(t) = rebalance(right(t));
   }
   return t;
}
```

- **Partition** moves a node with index i to the top
  - Using rotations
  - Note: there is a 0^{th} node

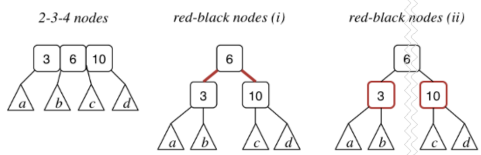
```
Tree partition(Tree t, int i) {
   if (t != NULL) {
      int m = TreeNumNodes(left(t));

      if (i < m) {
         left(t) = partition(left(t), i);
         t = rotateRight(t);
      } else if (i > m) {
         right(t) = partition(right(t), i-m-1);
         t = rotateLeft(t);
      }
   }
   return t;
}
```

- **Splay trees**
  - A kind of “self-balancing tree”
  - **Insertion-at-root** method (modified):

        - Consider **parent-child-grandchild** (p-c-g orientation)
            - Ie the algorithm "looks" two levels above for its rotation choice
        - Perform double rotations based on p-c-g orientation
        - Double rotations improve balance
            - By two rotations we can bring any grandchild to the top
    - **Rotation-in-search**
        - When an element x is accessed, it is moved to the root
        - Balance improved BUT search more expensive
        - Recently accessed elements faster to access again

- Better balanced binary search trees
    - We've seen
        - Randomised trees – poor performance unlikely
        - Occasionally rebalanced trees – fixed periodically
        - Splay trees – reasonable amortized performance
        - All have O(n) worst case
- **AVL trees**
    - Fix imbalances as soon as they occur
    - When we see an imbalance (height(left) – height(right) > 1)
    - This can be solved *by a single rotation*
      Note: we start from the bottom and work our way up until a subtree is deemed imbalanced. We apply the rotation on this subtree
        - If left subtree is too deep, rotate right
        - If right subtree is too deep, rotate left
    - What is an imbalance?
        - *Height of the left and right differ by more than 1*
    - For every given imbalance, there are four orientations of imbalance
        - LL imbalance: rotate right
        - RR imbalance: rotate left
        - LR imbalance: rotate left then right
        - RL imbalance: rotate right then left
        - https://en.wikipedia.org/wiki/File:AVL_Tree_Example.gif
    - https://www.geeksforgeeks.org/avl-tree-set-1-insertion/
- **2-3-4 trees** (these are B-trees with order 4)
    - Varying sized nodes assist balance
    - Nodes
        - 2-nodes – node has one value and two children
        - 3-nodes – node has two values and three children
        - 4-nodes – node has three values and four children
    - A node can only have



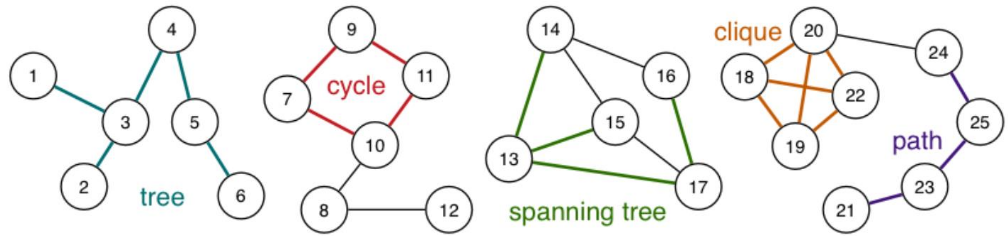
- **Red-black** trees
    - 2-3-4 trees but with binary nodes
        - **Red nodes** represent 234 siblings of their parents
        - **Black nodes** represent 234 children of their parent
- **B-trees**
    - You chose a value
        - That value is the max number of entries in one node
    - B-tree property
        - Every node must be "half-full"

- Insertion algorithm: we are given a new value
    - The algorithm tries to fill up the appropriate *(LOWEST)* node to have the max values
        - If we can, great
        - If not, pick the middle value *(BEFORE insertion)* and move that to the parent node
            - If it fits, great
            - If not, split up the parent node
- We want to delete a node
    - We just remove it if we can
    - If this violates the B-tree property, we fill the node using a sibling
        - The sibling gives to the parent and the parent gives to the child (this keeps order)
    - If we can't, we merge with our sibling
        - Take from the parent and merge with sibling

Graphs – *The Web of Data*

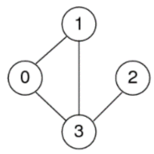
- A data type based on a collection of items and relationship between them
    - Eg
        - Maps: items are cities, connections are roads
        - Web: items are pages, connections are hyperlinks
        - Timetables: items are courses, connections are clashes
    - V is a set of vertices (nodes)
    - E is a set of edges (a subset of V*V)
    - Questions
        - Is there a way to get from item A to item B?
        - What is the best way to get from A to B?
        - Which items are connected?
    - Representation
        - No implicit order
        - Graphs may contain **cycles**
        - Algorithm complexity depends on connection complexity
- Properties
    - $\max(E) = \frac{V(V-1)}{2}$
    - The ratio $E:V$ can vary loads
        - **Dense** if E is closer to V^2
        - **Sparse** if E is closer to V
- Terminology
    - **Adjacent**: describes two nodes connected by an edge
    - **Degree**: the number of edges incident on a vertex
    - **Path**: a sequence of vertices where each is connected by an edge
    - **Cycle**: a path where the last vertex in the path is the first
    - **Length**: number of edges in a path or cycle
    - **Connected graph**: there is a path from every vertex to every vertex
    - **Connected components**: describes nodes which are connected
    - **Complete graph**: there is an edge from each vertex to every other

- **Tree**: graph with no cycles
- **Spanning tree**: tree containing all vertices
- **Clique**: complete subgraph



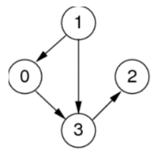
- Undirected graph: if there are no “arrowheads” on edges
- Directed graph: $edge(u, v) \neq edge(v, u)$
- Weighted graph: each edge has a value
- Multi-graph: multiple edges can exist between vertices

- Representation of edges
  - A literal array of all edges
  - Adjacency matrices
    - Represent edges
    - Symmetric for direct graphs
  - Adjacency lists
    - An array of linked lists
    - Each array element is the from node
    - Each element in the linked list is a to node



Undirected graph

A	0	1	2	3
0	0	1	0	1
1	1	0	0	1
2	0	0	0	1
3	1	1	1	0



Directed graph

A	0	1	2	3
0	0	0	0	1
1	1	0	0	1
2	0	0	0	0
3	0	0	1	0

- Graph ADT
  - Data
    - Set of edges
    - Set of vertices
  - Operations
    - Create graph
    - Add edge
    - Delete edge
    - Remove graph
    - Check if graph contains a given edge
  - Note
    - Set of vertices is fixed when graph initialised

- Graph traversals
  - **Loop-Guard™**
    - We keep an array of size V
    - Every time we visit a node, we mark it off on the array
    - We make sure we never go to a marked node twice
  - **Breadth-first**
    - Start with a given node (X)
    - Enqueue all the neighbours of X
    - Enqueue all the neighbours of the neighbours of X
    - Etc
    - *Will find the fastest path from X to whatever*

- Depth-first
    - Start with a given node (X)
    - Pick a neighbour, add it to the stack
    - Pick a neighbour of the neighbour, add it to the stack
    - Etc
    - After we finish a given path, pop, and try to take an alternate route
    - Etc, until we cover all nodes
    - Alternate to stack: the role of the "visited" array
        - Have array
        - When we move from A to B
        - array[B] = A
        - We can later reverse-engineer our path

```
int reachableCount (Graph g, int nV, Vertex v) {
    Vertext w;
    int total = 0;
    visited[v] = 1;

    for (int w = 0; w < nV; w++) {
        if (adjacent(g, v, w) && visited[w]  == -1) {
            total += reachableCount(g, nV, w);
        }
    }
    total++;
    return total;
}
```

- **Hamiltonian Paths and Circuits**
    - Simple paths connecting v, w in a graph such that each vertex is included exactly once
    - If $v = w$ we have a Hamiltonian path and circuit
    - Approach:
        - Generate all possible simple paths
        - Keep a counter of vertices in current path
        - Stop when we find a path containing V vertices
- **Euler Paths and Circuits**
    - Same as above, but we want every edge, not node
    - A graph has a Euler circuit IFF it is connected and all vertices have even degree
    - A graph has a non-circuitous Euler path IFF it is connected and exactly two vertices have an odd degree
- **Directed Graphs (Digraphs)**
    - Properties:
        - Matrix representation is non-symmetric
        - Max edges is V^2
        - Degree of vertex is number of edges going out of v
        - The indegree is the number of edges going into v

- Reachability if we can get from v to w
- Strong connectivity if every vertex is reachable from every vertex
- Directed acyclic graph contains no directed cycles

- **Transitive Closure**
  - Is t reachable from s?
  - We use **Warshall's algorithm**
    - for a (for b (for c (if we can go a→b and b→c we can go a→c)))
    - Ie we allow any step of V to count as a viable step

- **Weighted Graphs**
  - Minimum spanning tree (MST)
    - (Cheapest way to connect all vertices)
    - Assumes edges weighted and undirected
    - Spanning → all vertices
    - Tree → no cycles
    - Minimum → sum of edge weights Is no larger than any other
    - **Kruskal's algorithm (Kruskal is krazy for first principles)**
      - Begin with empty MST
      - Consider all edges in increasing weight order
        - Ie consider A-C 4, D-C 6, B-E 7 etc etc
      - Add the given edge if it does not form a cycle
      - Repeat until $V - 1$ edges are added
    - **Prim's algorithm (Prim is primed to expand)**
      - Start from any vertex v and empty MST
      - Choose edge not already in MST to add if it satisfies:
        - Incident on a vertex s already connected to v in MST
        - Incident on a vertex t not already connected to v in MST
        - Must have minimal weight
        - Ie: we choose the minimal edge that connects [one node inside existing tree to one node outside existing tree] and add it to the tree
  - Shortest path
    - (Cheapest way to get from A to B)
    - Assumes edge weights are positive
    - Directed or undirected
    - **Dijkstra's algorithm** – we require:
      - dist[] V-indexed array of cost of shortest path from s
      - pred[] V-indexed array of the predecessor in shortest path from s
      - These each contain data for the shortest paths *discovered so far*
        - dist[v] is length of shortest known path from s to v
        - dist[w] is length of shortest known path from s to w
      - Relaxation
        - If we find a shorter path from s to w we update data for w
        - Ie if $dist[v] + weight < dist[w]$ then update
    - The algorithm:
      - We just iterate V times and find the shortest paths (similar to Warshall)

Heaps – *Yeet*

- Heaps are trees with top-to-bottom ordering
    - Rules
        - Every parent node is greater than both children
        - In the last level, must fill from left to right
        - Removal always takes from the top
            - When we remove, bottom-right value goes head, and then we perform swaps until we satisfy heap requirements
        - **Summary of adding**: keep adding values in lowest, leftest spot, and swap with parent while > parent
        - **Summary of removing**: remove the top value, new top is the lowest, rightest spot, while < children swap with greatest child
    - Representation
        - The entire heap can be represented as an array
        - Usually array[0] is inf
        - Given a parent node at i, left child is at $2i$, right at $2i+1$
        - Ie given a node at j, parent is at $j/2$
            - Thus given array representation we can easily do our swaps

Hashing – *Mmm Tasty*

- Purpose
    - Key-index arrays had perfect $O(1)$ search performance
    - Required a dense range of index values
    - Used a fixed-size array (larger more useful but spacious)
- Hashing
    - Arbitrary types of keys
    - Map (hash) keys not compact ranges of index values
    - Store items in array, accessed by index value
- How it works
    - Uses arbitrary values as keys
    - We need a set of **key** values, each identifying one **item**
    - An array of size N to store **item**s
    - *Key values are spread uniformly over address range*
    - A hash function $h()$ to map key→ $[0, N-1]$
        - If $(x == y)$ then $h(x) == h(y)$
    - **Collisions**
        - Occur when $x \mathrel{!}= y \;\; \&\& \;\; h(x) == h(y)$
        - Inevitable when there are more than N keys
    - Resolving collisions
        - **Separate chaining**
            - Use like a linked list or something – the length “won’t be that long”
        - Or maybe just have an algorithm to chuck it in another slot
            - **Linear probing** – just try move to the next slot
            - **Quadratic probing** – similar

- **Double hashing** – use a second hash method (for the offset)
    - **Note**: linear probing is double hashing with $h_2(x) = 1$
        - Or change size of array, but this brings its own issues
- Characteristics
    - The cost of computation must be fast in order for maximum usefulness
    - Algorithms to hash are extremely arbitrary
        - Usually random and very disconnected to the key
        - This is to ward against key-related bias
        - Random formulas are ***empirically tested*** to ensure effectiveness
- Problems
    - Rely on size of array
        - Hash functions often depend on array size
        - Resizes necessitate rehashing
    - Collisions often inevitable
    - Sorting is effectively random
- Cost analysis
    - If good hash and keys < N, cost is 1
    - If good hash and keys > N, cost is $\left(\frac{M}{N}\right)/2$
    - **Load**
        - Ratio of items/slots
        - $\alpha = \frac{M}{N}$

Function Pointers and Generic Types – *Outta Nowhere!*

- Function names are just pointers to functions

```
int square(int x) {return x*x;}
int (*fp)(int);

fp = &square;
// OR
fp = square;

n = (*fp)(10);
```

- We can now pass functions as arguments in functions

```
void printNode (list n);
void printGrade (list n);

void traverse(list ls, void(*fp) (list));

traverse(myList, printNode);
traverse(myList, printGrade);
```

- Polymorphism: the ability for the same code to perform the same action on different types of data
    - Parametric polymorphism
        - The code takes the type as a parameter (explicitly or implicitly)
    - Subtype polymorphism
        - Associated with inheritance hierarchies
- void *value
    - A generic data type
        - The programmer can pass in type-specific functions that can cast the void * to the appropriate type before manipulation
    - Advantages
        - One code works with multiple objects
        - The approach supports generic structures and algorithms
    - Downcasting can be dangerous, since runtime checks are not performed in C
    - The code can appear cluttered

Sorting Algorithms – *Bubble Sort is Bad*

- Framework

```
#define key(A)       A
#define less(A,B)    (key(A) < key(B))
#define swap(A,B)    {Item t; t = A; A = B; B = t;}
#define swil(A,B)    {if (less(A,B)) swap (A,B);}
```

- **Selection Sort**
    - Algorithm:
        - Put smallest element into first array slot
        - Put second element into second array slot
        - Repeat
    - Method:
        - When we "put xth number in xth slot", we swap the relevant entries
- **Insertion Sort**
    - Algorithm:
        - Assume the first n terms are sorted
        - Insert the $(n+1)^{th}$ term into the list in its sorted position (using swaps)
        - Repeat
- **Bubble Sort**
    - Algorithm:
        - Move through the array and keep sorting pairs of two
        - n^2
- **Shell Sort** – an improved Insertion Sort
    - Algorithm:
        - h-sort arrays (**strange empirical values of h**)
        - Continue for all specified h until $h = 1$

        - Some sequences are $O\left(n^{\frac{4}{3}}\right)$

- Quick Sort
    - Algorithm:
        - Pick a number
        - Put all numbers less on the left
        - Put all numbers more on the left
        - Repeat on the smaller intervals until we are at size 1

```
void quicksort(Item a[], int lo, int hi)
{
   int i; // index of pivot
   if (hi <= lo) return;
   i = partition(a, lo, hi);
   quicksort(a, lo, i-1);
   quicksort(a, i+1, hi);
}
```

    - Partitioning phase:
        - Find the middle number
        - Swap until we have perfect balance
    - The swap method:
        - Begin i pointing to start of array and j pointing to end
        - i moves along until $a[i] > pivot$ (ie we find a no. on the wrong side)
        - Then j moves along until $a[j] < pivot$ (ie we find a no. on the wrong side)
        - Swap 'em
        - Keep this up until $i == j$, then put the pivot in the right spot (i) and return i for use in quicksort()

```
int partition(Item a[], int lo, int hi)
{
   Item pivot = a[lo];                              // pivot
   int  i = lo+1, j = hi;
   for (; ;) {
      while (less(a[i], pivot) && i < j) i++;
      while (less(pivot, a[j]) && j > i) j--;
      if (i == j) break;
      swap(a,i,j);
   }
   j = less(a[i], pivot) ? i : i-1;
   swap(a,lo,j);
   return j;
}
```

    - Complexity
        - Best case $O(n \log n)$
        - Worst case $O(n^2)$
    - Improvements
        - "Median of three" / randomisation can help choose a better starting pivot
        - Handle smaller partitions differently (insertion sort)
            - (There is little benefit to partitioning when $n < \sim 5$)

- **Merge Sort**
    - Algorithm:
        - Given two sorted lists we can merge them easily
        - Partition out list in two recursively
        - Sort if list is of size 1 or 2
        - End recursion merging lists

```
void mergesort(Item a[], int lo, int hi)
{
   int mid = (lo+hi)/2; // mid point
   if (hi <= lo) return;
   mergesort(a, lo, mid);
   mergesort(a, mid+1, hi);
   merge(a, lo, mid, hi);
}
```

    - Complexity
        - Always $O(n \log n)$
    - Can be done non-recursively with arrays

```
void mergesort(Item a[], int lo, int hi)
{
   int i, m; // m = length of runs
   int end; // end of 2nd run
   for (m = 1; m <= hi-lo; m = 2*m) {
      for (i = lo; i <= hi-m; i += 2*m) {
         end = min(i+2*m-1, hi);
         merge(a, i, i+m-1, end);
      }
   }
}
```

- **External Merge Sort**
    - Merge Sort for external files
    - Have two files A/B
    - Need to sort A:
        - 2-sort it, write to B
        - 4-sort that, write to A
        - 8-sort that, write to B
        - Etc…

- **Heap Sort**
    - Chuck everything in a heap, pop off the elements in order, and it's sorted!

<u>Non-comparative sorts</u>

- **Radix sort**
  - Represent key as tuple $(k_1, k_2, \ldots, k_m)$
    - Sydney → (s,y,d,n,e,y)
    - 372 → (3,7,2)
  - Finite possible values of k_i
    - Eg numeric 0-9
    - Eg alpha-numeric 0-9, a-z
  - Algorithm:
    - Stable sort on k_m
    - Then stable sort on k_{m-1}
    - Etc until k_1
  - Complexity:
    - Given an order $O(n)$ stable sort, our algorithm is $O(mn)$
      - m-tuple
- **Bucket/Pigeonhole Sort**
  - Have an array of finite values
  - Create an array of this size (bucket array)
  - According to value, slot into correct bucket in bucket array
  - Move entries in bucket array in required order to output
  - $O(n)$ assuming number of buckets is not large

Sorting summary:

- **Stable** – the sort will maintain original order for same-key elements
- **Adaptive** – the sort will be faster if the list is partially ordered

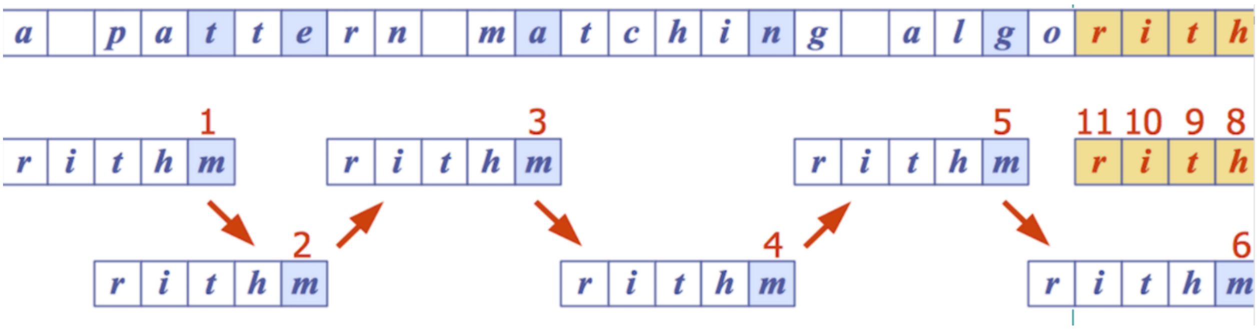
Sort	Overview	Order	Type	Stable	Adaptive
Selection Sort	Iterate over elements and find smallest n times	$O(n^2)$	Comparative	Can be	No
Insertion Sort	Given the first k elements are sorted, insert the $(k+1)^{th}$	$O(n^2)$	Comparative	Can be	Can be
Bubble Sort	Move through the array and keep sorting adjacent pairs	$O(n^2)$	Comparative	Can be	Yes
Shell Sort	h-sort arrays for decreasing h	$\leq O\left(n^{\frac{4}{3}}\right)$	Comparative	No	Yes
Quick Sort	Partition the array and recurse on the two halves	$O(n^2)$ - $O(n \log n)$	Comparative	Yes (lists easier than arrays)	Can be
Merge Sort	Use recursion to merge increasingly less small lists	$O(n \log n)$	Comparative	Can be	Can be
Heap Sort	Throw everything in a heap and pop it all off	$O(n \log n)$	Heap-y	No	Can be
Radix sort	Represent key as tuple $\{k\}$, then perform stable sort last-to-first on each k_i.	$O(mn)$	Non-comparative	Yes	?

Bucket / Pigeonhole Sort	Works with array of finite values. Create new array and slot element in according to value.	$O(n)$	Non-comparative	Yes	?

Text Processing Algorithms – *wrord*

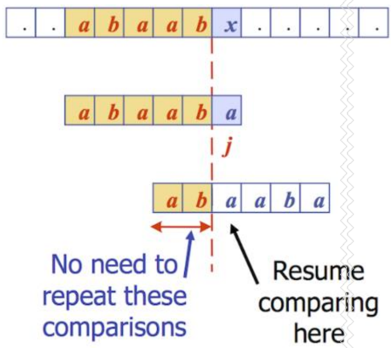
- Terms
  - **String** – a sequence of characters
    - Egs
      - C program
      - HTML document
      - DNA sequence
      - Digitized image
  - **Alphabet** – the set of possible characters in string
    - Egs
      - ASCII
      - Unicode
      - {0,1}
      - {A,C,G,T}
  - **length(P)** – number of characters in P
  - $\boldsymbol{\lambda}$ – empty string
  - $\boldsymbol{\Sigma^m}$ – set of all strings of length m over alphabet Σ
  - $\boldsymbol{\Sigma^*}$ – set of all strings over alphabet Σ
  - **Substring** – a string which appears within another string
  - **Prefix** – a substring which begins a string
    - λ is always one
  - **Suffix** – a substring which ends a string
    - λ is always one
- Pattern matching
  - Given two strings T (text) and P (pattern), find a substring of T equal to P
- **Boyer-Moore Algorithm**
  - Looking-glass heuristic:
    - Compare the last digit of P first
  - Character-jump heuristic:
    - When we mismatch $T[i\} = c$
    - If P contains c, shift P to align the last occurrence of c in P with $T[i]$

- Otherwise, shift P to align $P[0]$ with $T[i + 1]$ (aka “big jump”)



  - Overall heuristic:
    - Do jumps which are the size of P
    - We have a safety mechanism for double checking we don’t skip anything
    - This mechanism is the alignment of characters which aren’t exact matches but are in P
  - Uses **last occurrence function**:
    - For each unique letter of the substring, we store *the number of the index of the last occurrence of it*
    - Shift: $(spot\ in\ the\ array\ we're\ at) - (last\ occurrence\ function\ value)$
      - OR if we don’t find the letter, do a full big shift
  - With English text, generally great
  - *Works best on large alphabets*

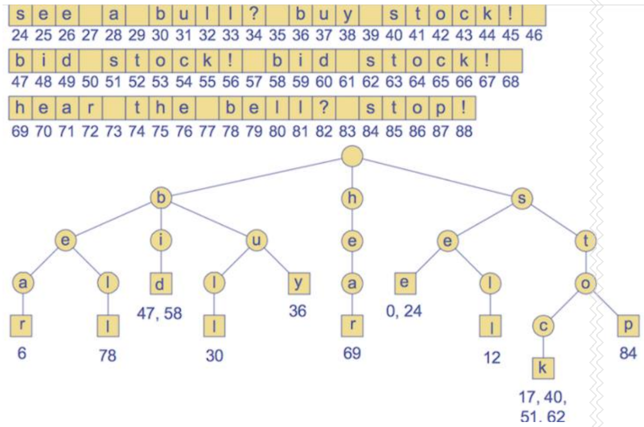
- **Knuth-Morris-Pratt Algorithm**



  - Compares left to right, but shifts the algorithm smartly
  - We shift so the largest prefix of $P[0-j]$ that is also the largest suffix of $P[1-j]$
  - Uses **failure function**:
    - $F[j] =$ (size of the largest prefix of $P[0-j]$ that is also a suffix of $P[1-j]$)
    - Shift: $(spot\ in\ the\ array\ we\ fail\ on) - F[spot - 1]$
  - Steps:
    - Form the table initially
    - Number for each index represents how suffix-able it is
  - *Works best on small alphabets*

- Pre-processing Strings
  - Pre-processing the pattern speeds up matching enquiries
  - If the text is large, immutable, and searched often, it can be pre-processed
- **Tries**
  - What it is
    - A trie is a compact data structure for representing a set of strings
    - Supports pattern matching queries in time proportional to the pattern size

- Implementation



    - Trees using parts of keys (rather than whole keys)
    - Nodes may have up to 26 children
    - May be tagged as a "finishing node"
    - "Finishing nodes" may still have children
    - Depth d is length of longest key value (word)
    - Cost of searching is $O(d)$
- Compressed tries
    - You can compress by taking trails of letters that don't branch and shoving them into one word
    - You can compress by not storing letter combos but ranges of indices
- Suffix matching
    - You literally put all the suffixes in a trie as well lmao
    - Can be constructed in $O(n)$ time and $O(n)$ space

- **Huffman's algorithm** (loss-less text compression)
    - Given a string X, encode it by a smaller string Y
    - Algorithm:
        - Computes frequency for each character
        - Encodes high-frequency characters with short binary code
        - *No code word is a prefix of another code word*
            - Eg a→0, b→10, c→110 etc
            - Thus computer can always read them without confusion
        - Uses optimal **encoding tree** to determine code words
    - Build tree:
        - Bottom two nodes are least common
        - Parent node is sum of their frequencies, becomes new node
        - Bottom two nodes are least common
        - Parent node is sum of their frequencies, becomes new node
        - Etc

Software Development Process – *Do We Really Need Notes on This?*

- Testing
    - Increases our confidence in our solution
        - We can never demonstrate our program is correct
    - Good practice:
        1. Determine classes / partitions of the input data set
        2. Choose representative input values from each class
        3. Determine expected output from each input
        4. Execute program using all representative inputs
        - Ensure all common bugs will be exercised

- Types
  - "Big Bang"
    - Write whole program then test
    - Bad idea bad idea bad idea
  - "As You Go"
    - Write a small piece of code, then test
    - Integrate with other pieces and test again
    - Repeat iteratively until the program is constructed
  - "Regression"
    - Re-run all testing after any changes to the system

- Debugging strategies
  - Make the bug reproducible
  - Search for patterns in the failure
  - Divide and conquer (isolate the buggy region)
  - Write self-checking code (assert, etc)
  - Use a log file
  - Draw a picture
- gdb
  - Programs must be compiled with the gcc -g flag (debugging option)
  - gdb executable core (core optional)
    - quit – quits
    - help [command]
    - run ARGS – runs program with arguments
    - where – find which function the program was executing when it crashed
    - list [LINE] – display five lines either side of current statement
    - print EXPR – display expression values
      - a@1 – shows the whole array a
    - break [PROC|LINE] – set break point
    - next – single step, does entire functions
    - step – single step
- Performance analysis
  - Empirical study shows the 90/10 rule generally holds
    - 90% of the execution time is spent in 10% of the code
    - Small regions of the code are "hot-spots"
  - These should be targets for efficiency

The End?