CSI 2110 Summary

Fall 2012

Analysis of Algorithms

Algorithm - a step by step procedure for solving a problem in a finite amount of time. Analyzing an algorithm means determining its efficiency.
Primitive operations - low-level computations independent from the programming language that can be identified in the psuedocode.

Big-Oh - given two functions $f(n)$ and $g(n)$, we say $f(n)$ is $O(g(n))$ if and only if there are positive constant c and n_0 such that $f(n) \leq cg(n)$ for $n \geq n_0$. It means that an algorithm has the complexity of **AT MOST** $g(n)$**.**

- Just multiply all terms by the highest degree of n, add, and you have your c (the coefficient) and $g(n)$ (the function).
- $O(1) < O(\log n) < O(n) < O(n \log n) < O(n^2) < O(2^n)$ …
- Logarithms always in base 2 in this class unless otherwise stated
- Always want the lowest possible bound, approximation should be as tight as possible
- Drop lower order terms and constant factors

Big-Omega - $f(n)$ is $Omega(g(n))$ such that $f(n) \geq cg(n)$ for all $n \geq n_0$. It means an algorithm has the complexity of **AT LEAST** $g(n)$.
Big-Theta - $f(n)$ is $\theta(g(n))$ if it is $O\big(g(n)\big)$& $Omega\big(g(n)\big)$. It means the complexity **IS EXACTLY** $g(n)$.

Properties of logarithms and exponentials:

- $\log(xy) = \log(x) + \log(y)$
- $\log\left(\frac{x}{y}\right) = \log(x) - \log(y)$
- $\log(ax) = a\log(x)$
- $a^{b+c} = a^b a^c$
- $a^{bc} = (a^b)^c$
- $\frac{a^b}{a^c} = a^{b-c}$
- $b = a^{\log_a b}$
- $b^c = a^{(c \log_a b)}$

Remember the following equations:

- $\sum_1^{n-1} i = \frac{n(n+1)}{2}$
- $\sum_{i=0}^{n} id = \frac{d}{2} n(n+1)$
- $\sum_{i=1}^{n} r^i = \frac{(r^{n+1}-1)}{r-1}$
- $2^{\log(n)} = n$
- $\sum_{i=1}^{n} \log i = n \log n$
- $\sum_{v \in V} d(v) = 2m$ Only if the graph is implemented as an adjacency list

Stacks and Queues

Stacks - First in, last out.
Queues - First in, first out.

Extendable Arrays

- Growth function is $f(n)$

Regular push: 1
Special push: Cost of creating the new array ($f(n)$) + copying the old elements into the new array (length of old array) + 1
Phase: starts at creation of the new array and ends with the last element being pushed. The one pushed after that is a special push, and starts a new phase.

Complexities

Method	ArrayList	LinkedList	Unsorted Seq	SortedSeq
size	$O(1)$	$O(1)$		
isEmpty	$O(1)$	$O(1)$		
get	$O(1)$	$O(n)$		
replace	$O(1)$	$O(n)$		
insert	$O(n)$	$O(n)$	$O(1)$	$O(n)$
remove	$O(n)$	$O(n)$		
minKey/minElement			$O(n)$	$O(1)$
removeMin			$O(n)$	$O(1)$

Selection Sort

Using an external data structure - insert your elements into the first data structure, find the smallest one, add it into your second data structure. Takes $O(n)$ time.
In place (Not using an extra data structure) - search through the array, find the smallest, add it to the front of the array. Repeat until it's sorted.

- Complexity: Average and worst case: performs in $O(n^2)$ operations regardless of input, since $\mathrm{removeMin}()$ is always executed in $O(n)$.

Insertion Sort

Using an external data structure - insert the first element of the first data structure into the other one, but adding each element before or after the existing elements according to size. Then, $\mathrm{removeMin}()$ and add it into the old ADT.
In place - Take the second element and switch it with the first if needed. Take the third element and switch it with the second, and then the first if need be, and so on.

- Complexity: Average and worst case is $O(n^2)$.

Trees

Graph - consists of vertices and edges.
Tree - a graph that contains no cycles.
Root - a node without a parent.
Internal node - a node with at least one child.
External node - a node without any children.
Ancestor - parent, grandparent, great-grandparent, etc.
Descendent - child, grandchild, great-grandchild.
Subtree - a tree consisting of a node and its descendants.
Distance - number of edges between two nodes.

Depth - number of ancestors.
Height - the maximum depth.
Preorder - visit, left, right.
Postorder - left, right, visit.
Inorder - left, visit, right.

Binary Trees

- Each node has at most two children.
- Examples: decision trees (yes/no), arithmetic expressions,

Full binary tree - each node is either a leaf, or has two children.

- In the book, children are completed with dummy nodes, and all trees are considered full.

Perfect binary tree - a full binary tree with all the leaves at the same level.
Complete binary tree - perfect until the level $\mathrm{h}-1$, then one or more leaves at level h.

Properties of Binary Trees

- $n = \#of\ nodes = 2e - 1$
  - In Perfect Binary Trees: $n = 2^{\mathrm{h}+1} - 1$
- $e = \#of\ leaves = i + 1 \leq 2^{\mathrm{h}}$
- $i = \#of\ internal\ nodes$
- $\mathrm{h} = \mathrm{h}eight \leq i,\ \leq \frac{n-1}{2}, \geq \log e,\ \geq \log(n+1) - 1$
- Maximum number of nodes at each level is 2^i

Properties of Height

- Binary: $\mathrm{h} \geq \log(n+1) - 1$
- Binary (Full): $\log(n-1) - 1 \leq \mathrm{h} \leq \frac{n-1}{2}$
- Binary(Complete): $n \geq 2^{\mathrm{h}} \quad \mathrm{h} = \log n$ (integer part of $\log n$)
- Binary(Perfect): $n = 2^{\mathrm{h}+1} - 1 \quad \mathrm{h} = \log(n+1) - 1$

Implementing Complete Binary Trees with ArrayList

- $\mathrm{swapElements}()$, $\mathrm{replaceElement}()$, $\mathrm{isRoot}()$, $\mathrm{isInternal}()$, $\mathrm{isExternal}()$, $\mathrm{leftChild}()$, $\mathrm{rightChild}()$, $\mathrm{sibling}()$, all have complexity $O(1)$.
- Left child of $T[i]$ is $T[2i]$ if $2i \leq n$.
- Right child of $T[i]$ is $T[2i+1]$ if $2i+1 \leq n$.
- Parent of $T[i]$ is $T[i/2]$ if $i > n$.
- Root is $T[1]$ if $T \neq 0$
- Is $T[i]$ a leaf? True if $2i > n$.

Representing General Trees

A tree T is represented by a binary tree T' with the following algorithm, seen in class:

- $u \in T,\ u' \in T'$.
- First child in u in T is the left child of u' in T'
- First sibling of u in T is right child of u' in T'
- If u is a leaf and has no siblings, then the children of u' are leaves.

- If u is internal and v is its first child then v' is the left child of u' in T'
- If v has a sibling w immediately following it, w' is the right child of v' in T'.

Heaps

MinHeap - All children are larger than its parent. $\text{key}(parent) \leq \text{key}(child)$
MaxHeap - All children are smaller than its parent. $\text{key}(parent) \geq \text{key}(child)$
Height of a heap: a heap storing n keys has a height of $\log n$

Removal - $\text{removeMin}()$

- Remove the top element
- Replace with last key in the heap.
- Begin the downheap.

Downheap

- Compares the parent with the smallest child.
- If the child is smaller, switch the two.
- Keep going.
- Stops when the key is greater than the keys of both its children or the bottom of the heap is reached.
- $Total\ swaps \leq (\text{h} - 1) \Rightarrow O(\log n)$

Insertion

- Add key into the next available position.
- Begin upheap.

Upheap

- Similar to downheap.
- Swap parent-child keys out of order.

Regular Heap Construction -We could insert items one at a time with a sequence of heap insertions, $\sum_{k=1}^{n} \log k = O(n \log n)$. But we can do better with bottom-up heap construction.

Bottom-up Heap Construction (WILL be a question on midterm and/or final exam!)

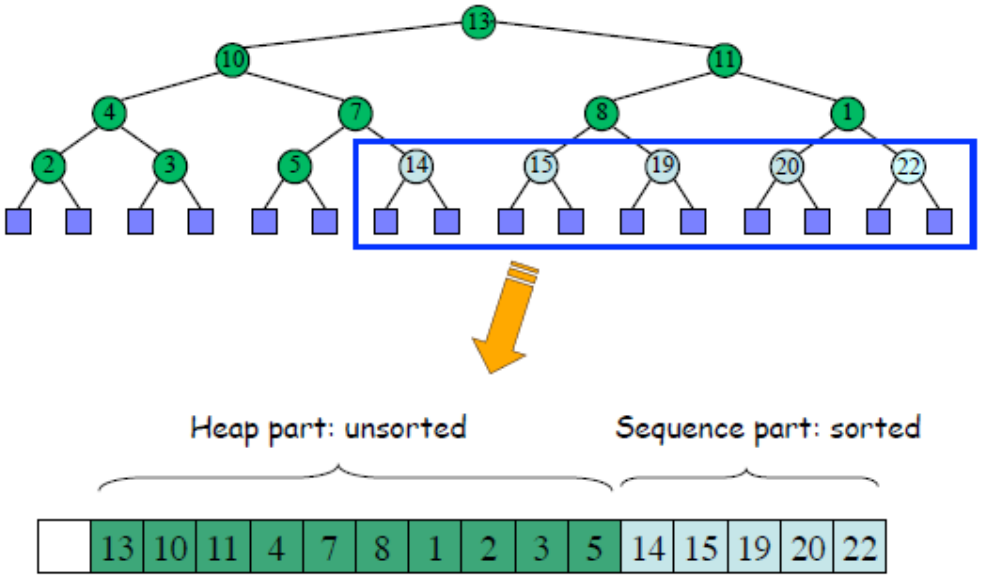
- Idea: recursively rearrange each subtree in the heap starting with the leaves.
- Switch starting from the bottom, work your way up to bigger subtrees and rearrange them.
- To construct a heap bottom-up, construct a tree with the number of nodes you need to insert. Start from the back of the array, and start adding from the right of the subtree, filling all the leaves.
- These are all okay. Move on to the right of the level $\text{h} - 1$ and fill in the entire level (again, working backwards from the array). Rearrange those subtrees.
- Continue to the next level. Fill, rearrange, etc.
- Number of Swaps: How many nodes at level i? There are 2^i. Will do $\text{h} - i$ swaps.
- Complexity is $O(n)$ which is the best we can possibly get.

Heaps are implemented the same way as binary trees.

Heap Sort - Cost is $O(n \log n)$

Heap Sort In Place

- Phase 1 - We build a max-heap to occupy the whole structure
- Phase 2 - We start the part "sequence" empty and we grow it by removing at each step $(i = i, \ \ldots, \ n)$ the max value from the heap and by adding it to the part "sequence", always maintaining the heap properties for the "heap" part.
- Take the root and switch it with the rightmost child. Reheap. Keep going along the last row the same way. Switch with root, reheap, etc.



Maps and Dictionaries

Map - multiple items with the same key are NOT allowed
Dictionary - multiple items with the same key ARE allowed

Unordered Sequence

- Searching and removing takes $O(n)$ time
- Inserting takes $O(1)$ time.
- Applications to log files (frequent insertions, rare searches and removals)

Ordered Sequence - Array-based

- Searching takes $O(\log n)$ with **binary search**.
- Inserting and removing takes $O(n)$ time.
- Applications to look-up tables (frequent searches, rare insertions and removals).
- Start in the middle, if the key we're searching for if larger, go to the right. If it's smaller, go to the left.

Binary Search Trees

- A binary tree such that keys stored to the left are less than k, and keys stored to the right are greater than or equal to k.
- External nodes do not hold elements but serve as place holders.
- Inorder traversal gives you the keys in increasing order

- Complexity:
  - Worst case: $O(n)$ where all keys are to the right or to the left.
  - Best case: $O(\log n)$ where leaves are on the same level or on an adjacent level.

Insertion

- Always insert at the end of the search tree based on the correct order
- If you're adding a double, always insert to the right, not to the left of the tree.

Deletion

- If it's at the end, you can just remove it.
- If it's not at the end, replace it with the next in the inorder traversal.

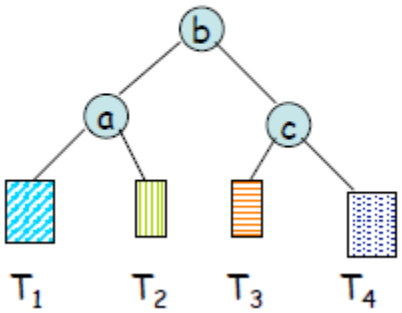
AVL Trees

- AVL trees are balanced.
- They are binary search trees that for every internal node v of T, the heights of the children v can differ by at most 1.
- Height of an AVL tree is always $O(\log n)$.
- The height h is $O(\log n)$.

Insertion

- **Balanced** - if for ever node v, the height of the c's children differ by at most 1.
- It tree becomes unbalanced, we need to rebalance.
- Rebalance
  - Identify three nodes (grandparent, parent, child) and the 4 subtrees attached to them.
  - Find the node whose grandparent is unbalanced. This node (child) is x, the parent is y, and grandparent is z.
    - Choice of x is not unique
  - Identify the four subtrees, left to right, as T_1, T_2, T_3, and T_4.
  - The first who comes in the inorder traversal is a, second is b, and third is c.

Replace the tree rooted at z with the following tree:



Removal

- Remove the same way as in a binary search tree. However, this may cause an imbalance.

Complexity

- Searching, inserting, and removing are all $O(\log n)$, that's what makes AVL trees so nice.

$(2,4)$ Trees

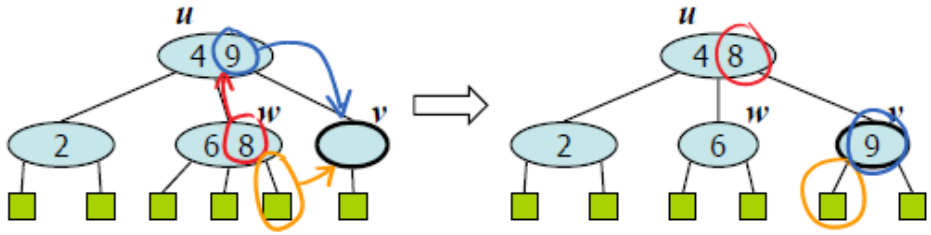
- A $(2,4)$ tree is a multi-way search tree with the following properties:
  - Node size property: every internal node has at most four children
  - Depth property: all external nodes have the same depth
- Can't have more than four children or less than two children.
- Depending on the number of children, an internal node of a $(2,4)$ tree is called a 2-node, 3-node, or 4-node.
- $\mathrm{h} \leq \log(n+1)$
- Searching in a $(2,4)$ tree with n items takes $O(\log n)$ time
- Min number of items: when all internal nodes have 1 key and 2 children: $n = 2^{\mathrm{h}+1} - 1$, $\mathrm{h} = O(\log n)$
- Maximum number of items n:when all internal nodes have three keys and four children. $n = \sum_{i=0}^{\mathrm{h}} 4^i = (4^{\mathrm{h}+1} - 1)$, $\mathrm{h} = O(\log_4 n) \Rightarrow Search\ O(\log n)$

Insertion

- Insert similar to a binary search tree. Insert at the end after (binary) searching where it goes.
- May cause overflow, since you're only allowed three elements in one node.
  - Take the third element, send it up, and make new nodes out of the first two (one) and the fourth one (two).
- Insertion takes $O(\log n)$ time

Deletion

- Replace the deleted key with the inorder successor
- Can cause underflow, might need to fuse nodes together to fix this. To handle an underflow at node v with parent u, we consider two cases:
  - Case 1: the adjacent siblings of v are 2-nodes.
    - Fusion operation: we merge v with an adjacent sibling w and move an item from u to the merged node v'.
    - After a fusion, the underflow may propagate to the parent u.
  - Case 2: an adjacent sibling w of v is a 3-node or a 4-node.
    - Transfer operation:
      1. We move a child from w to v
      2. We move an item from u to v
      3. We move an item from w to u
    - After a transfer, no underflow occurs.



- Deleting takes $O(\log n)$ time. Note that searching, inserting, and deleting all take $O(\log n)$ time.

Hash Tables

Problem A / Address Generation: Construction of the function $\mathrm{h}(K_i)$. It needs to be simple to calculate, and must uniformly distribute the elements in the table. For all keys K_i, $\mathrm{h}(K_i)$ is the position of K_i in the table. This position is an integer. Also, $\mathrm{h}(K_i) \neq \mathrm{h}(K_g)$ if $i \neq g$. Searching for a key and inserting a key (all dictionary ADT operations) takes $O(1)$ time. We have the function $\mathrm{h}(x) = \mathrm{h}_2(\mathrm{h}_1(x))$ where we have the two following sub-functions:

1. Hash code map $\mathrm{h}_1: \mathrm{keys} \rightarrow \mathrm{integers}$
   - They reinterpret a key as an integer. They need to:
     i. Give the same result for the same key
     ii. Provide a good "spread"
   - Polynomial accumulation
     - We partition the bits of the key into a sequence of components of fixed length, $a_0, a_1, \ldots a_{n-1}$.
     - Then we evaluate the polynomial $p(z) = a_0 + a_1 z + \cdots + a_{n-1} z^{n-1}$ at a fixed value z, ignoring overflows.
     - Especially suitable for strings.
   - Examples
     - Memory address (we reinterpret the memory address of the key object as an integer, this is the default hash code for all Java objects)
     - Integer cast (Reinterpret the bits of the key as an integer)
     - Component sum (We partition the bits of the key into components of fixed length and we sum the components).
2. Compression map $\mathrm{h}_2: \mathrm{integers} \rightarrow [0, \mathrm{TableSize} - 1]$
   - They take the output of the hash code and compress into the desired range.
   - If the result of the hash code was the same, the result of the compression map should be the same.
   - Compression maps should maximize "spread" so as to minimize collisions
   - Examples
     - Division: $\mathrm{h}_2(y) = y \bmod N$ where N is usually chosen to be a prime number (number theory).
     - Multiply Add Divide (MAD): $\mathrm{h}_2(y) = ay + b \bmod N$ where a and b are nonnegative integers such that $a \bmod N \neq 0$.

Problem B / Collision Resolution: What strategy do we use if two keys map to the same location $\mathrm{h}(K_i)$**?**
Load factor of a Hash table: $\alpha = \frac{n}{N}$ where n is the number of elements and N is the number of cells. The smaller the load factor, the better.

Linear Probing: We have $\mathrm{h}_j(K_i) = [\mathrm{h}(K_i) + j] \bmod N$. Consider a hash table A that uses linear probing. In order to search, we do the following:

- $\mathrm{findElement}(k)$:We start at cell $\mathrm{h}(K)$, and we probe consecutive locations until one of the following occurs:
  - An item with key K is found
  - An empty cell is found
  - N cells have been unsuccessfully probed

To handle insertions and deletions, we introduce a special object, called AVAILABLE, denoted AV, which replaces deleted elements.

- $\text{removeElement}(k)$: We search for an item with key K. If such an item (k, o) is found, re replace it with the special object AV and we return element o. Otherwise, we return NO_SUCH_KEY
- $\text{insertItem}(k, o)$: We throw an exception if the table is full. We start at cell $\text{h}(k)$. We probe consecutive cells until one of the following occurs:
  - A cell i is found that is either empty or labelled AV
  - N cells have been unsuccessfully probed

  We store item (k, o) in cell i.

Performance of Linear Probing: Average number of probes is $C(\alpha)$ where α is the loading factor. Even if it'd 90% full, on average you'll find it in 5 or 6 tries, so it's $O(1)$, which is very good. In the worst case, hash is not better than AVL, when clustering occurs, it's $O(n)$.

Quadratic Probing: We have $\text{h}_j(K_i) = [\text{h}(K_i) + j^2] \bmod N$. The problem with this is that modulo is hard to calculate and it only visits half of the table but it's not a big deal. Similar problem: you avoid linear clustering, but every key that's mapped to the same cell will follow the same path. There's more distributed clustering, which should be avoided. Called **secondary clustering**.

Double Hashing: We have $\text{h}_j(K_i) = [\text{h}(K_i) + j \cdot d(K_i)] \bmod N$ where h is the primary hashing function and d is the secondary hashing function.

Bubble Sort

- You literally bubble up the largest element. Move from the front to the end, Bubble the largest value to the end using pairwise comparisons and swapping. Once you go through the whole array, you start from the first element again and go through the same way.
- In order to detect that an array is already sorted so we don't have to go through it again unnecessarily, we can use a boolean flag. If no swaps occurred, we know that the collection is already sorted. The flag needs to be reset after each "bubble up".
- Complexity is $O(n^2)$.

Recursive Sorts

Divide and Conquer paradigm:

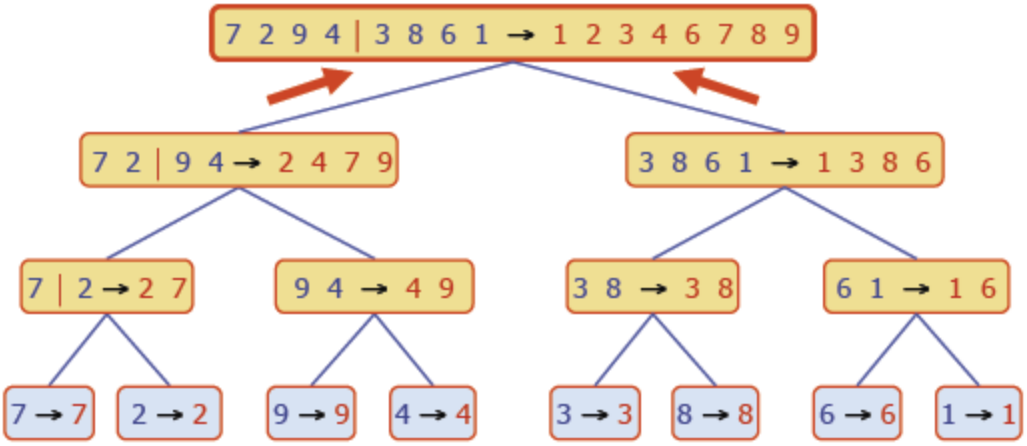
- *Divide*: divide one large problem into two smaller problems of the same type.
- *Recur*: solve the subproblems.
- *Conquer*: combine the two solutions into a solution to the larger problem.

Merge Sort

Merge sort on an input sequence S with n elements consists of three steps. It is based on the divide-and-conquer paradigm:

- *Divide*: partition into two groups of about $n/2$ each.
- *Recur*: recursively sort s_1 and s_2.
- *Conquer*: merge s_1 and s_2 into a unique sorted sequence.

The conquer step merged the two sorted sequences A and B into one sorted sequence S by comparing the lowest element of each A and B and insert whichever is smaller. Merging two sorted sequences, each with $n/2$ elements and implemented by means of a doubly linked list, takes $O(n)$ time.
It's depicted in a binary search tree kind of style.



The height h of the merge sort tree is $O(\log n)$, since at each recursive call we divide the sequence in half. The overall amount or work done at the nodes of depth i is $O(n)$, since we partition and merge 2^i of size $n/2^i$, and we make 2^{i+1} recursive calls. From this we get:

- Complexity: $O(n \log n)$

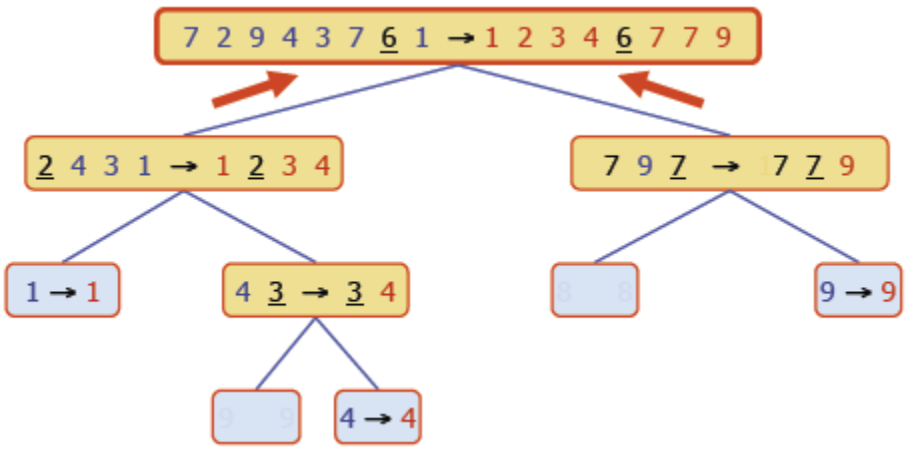
Quick Sort

Quick sort is also based on the divide-and-conquer paradigm.

- *Divide*: pick an element x, called the pivot, and partition S into
  - L elements less than x
  - E elements equal to x
  - G elements greater than x
- *Recur*: sort L and G
- *Conquer*: join L, E, and G.

Pivot can always be chosen randomly, or we can decide always to choose the first element of the array, or the last.

- Complexity:
  - Worst case: $O(n^2)$
  - Average case: $O(n \log n)$



Summary of Sorts

Algorithm	Time	Notes
selection-sort	$O(n^2)$ w.c. and av.	• in-place • slow (good for small inputs)
insertion-sort	$O(n^2)$ w.c. and av.	• in-place • slow (good for small inputs)
quick-sort	$O(n^2)$ w.c. $O(n \log n)$ average	• in-place, randomized • fastest (good for large inputs)
heap-sort	$O(n \log n)$ w.c. and av.	• in-place • fast (good for large inputs)
merge-sort	$O(n \log n)$ w.c. and av.	• sequential data access • fast (good for huge inputs)

Radix-Sort

- Crucial point of this whole idea is the stable sorting algorithm, which is a sorting algorithm which preserves the order of items with identical key.
- Question: the best sorts that we have seen so far have been $n \log n$, and there is no way to beat that unless we're under certain circumstances, in which case we can reach $O(n)$.

Bucket Sort

Let S be a sequence of n (key, element) items with keys in the range $[0, n-1]$. Bucket sort uses the keys as indices to an auxiliary array B of sequences (buckets).

- Phase 1: Empty sequence S by moving each item (k, o) into its bucket $B[k]$.
- Phase 2: For $i = 0, \dots, N-1$, move the items of bucket $B[i]$ to the end of sequence S.
- Takes $O(n+N)$ time.

Lexicographic Order

- A d-tuple is a sequence of d keys $(k_1, \dots k_d)$ where k_i is said to be the ith dimension of the tuple.
- The lexicographic order of two d-tuples is recursively defined as $(x_1, \dots, x_d) < (y_1, \dots, y_d)$, ie the tuples are compared by the first dimension, then the second, etc.
- **Lexicographic sort**: Let C_i be the comparator that compares two tuples by their ith dimension, ie for C_2, $(x_1, x_2, x_3) \leq (y_1, y_2, y_3)$ if $x_2 \leq y_2$. Let $\text{stableSort}(S, C)$ be any stable sorting algorithm that uses comparator C. Lexicographic sort sorts a sequence of d-tuples in lexicographic order by executing d times algorithm stableSort, one per dimension.
- You do it by starting from dimension d, putting those in order, then moving to $d-1$ and putting them in order, then continue this way until you reach dimension 1, put that in order, and you're done.
- Lexicographic sort runs in $O(dT(n))$ where $T(n)$ is the running time of the stable-sort algorithm.

Radix Sort Variation 1: This one uses the bucket-sort as the stable sorting algorithm. Applicable to tuples where keys in each dimension i are integers in the range $[0, N-1]$.

- This one runs in $O(d(n+N))$ time.

Radix Sort Variation 2: Consider a sequence of n b-bit integers $x = x_{b-1} \dots x_1 x_0$ We represent each element as a b-tuple of integers in the range [0,1] and apply radix sort with N=2. It sorts Java integers (32-bits) in linear time.

- This one runs in $O(bn)$ time.

Radix Sort Variation 3: The keys are integers in the range $[0, N^2-1]$. We represent a key as a 2-tuple of digits in the range $[0, N-1]$ and apply variation 1, ie write it in base N notation.

- This means write a number in this notation: $(a_0, a_1, \dots, a_m)$ where a_i are the coefficients of the following equation:
- $n = a_0 \cdot N^0 + a_1 \cdot N^1 + a_2 \cdot N^2 + \cdots + a_m N^m$
  $= a_0 + a_1 \cdot N + a_2 \cdot N^2 + \cdots + a_m N^m$ where n is the number you're putting in base N.
- Examples
  - If $N = 10$, write 75 as $(7,5)$, since $75 = 5 + 7 \cdot 10$
  - If $N = 8$, write 35 as $(4,3)$, since $35 = 3 + 4 \cdot 8$
- This one runs in $O(n+N)$ time.

Graph Traversals

Subgraphs

Subgraph: A subgraph S of a graph G is a graph such that the vertices of S are a subset of the vertices of G and the edges of S are a subset of the edges of G.
Spanning subgraph: A subgraph that contains all the vertices of G.
Connected: When there is a path between every pair of vertices.
Connected component: A maximal connected subgraph of G.
(Free) Tree: An undirected graph T such that T is connected and has no cycles.
Forest: A collection of trees. The connected components of a forest are trees.
Spanning tree: A spanning subgraph that is a tree.
Spanning forest: A spanning subgraph that is a forest.
Traversal: A traversal of a graph G visits all vertices and edges of G, determines whether G is connected, computes the connected components of G, computes a spanning forest of G, builds a spanning tree in a connected graph.

Depth First Search

DFS is a graph traversal technique that, on a graph with n vertices and m edges, has a complexity of $O(n+m) = O(m)$ and can be further extended to solve other graph problems.
Idea: The idea is to start at an arbitrary vertex, follow along a simple path until you get to a vertex which has no unvisited adjacent vertices, then start tracing back up the path, one vertex at a time, to find a vertex with unvisited adjacent vertices.

With a stack: Start at a vertex, add it to your visited set $\{\text{visited}\}$. Push all the edges into the stack, then pop the first one. Add the vertex it brings you to to $\{\text{visited}\}$. Push that vertex's vertices, then pop, visit that vertex if it hasn't been visited yet, push its vertices, etc. until there are no edges left in the stack.
With recursion: $\text{DFS}(v)$: mark v visited, for all vertices w that are adjacent to v, visit w if it hasn't been visited yet, $\text{DFS}(w)$.

Unexplored vertex: A vertex that hasn't been visited yet.
Visited vertex: A vertex that we've already included in our $\{\text{visited}\}$ set.
Unexplored edge: An edge that we have not visited yet.
Discovery edge: An edge that leads to an unexplored vertex.
Back edge: An edge leading to an already visited vertex.

Properties of DFS:

- $\text{DFS}(G, v)$ visits all the evrtices and edges in the connected component of v
- The discovery edges labeled by $\text{DFS}(G, v)$ form a spanning tree of the connected component of v.
- Setting/getting a vertex/edge label takes $O(1)$ time.
- Each vertex is labeled twice, once as unexplored and once as visited.
- Each edge is labeled twice, once as unexplored and once as discovery (or back).
- Complexity:
    - Adjacency List
        - Average case is $O(m)$
        - Worst case is when $m = n^2$, so $O(n^2)$.
    - Adjacency Matrix
        - $O(n^2)$

Applications:

- **Path Finding**: We can specialize the DFS algorithm to find a path between two given vertices u and z using the template method pattern. We call $\text{DFS}(G, u)$ with u as the starting vertex, using a stack S to keep track of the path between the start vertex and the current vertex. As soon as we reach z, we return the path, which is the contents of stack S.
- **Cycle Finding**: We can specialize the DFS algorithm to find a simple cycle using the template method pattern. We use a stack S to keep track of the path between the start vertex and the current vertex. As soon as a back edge (v, w) is encountered, we return the cycle as the portion of the stack from the top to vertex w.

Breadth-First Search

BFS is a graph traversal that can be further extended to solve other problems and, on a graph with n vertices and m edges, takes $O(n + m)$ time (with adjacency list implementation).
Idea: Visit a vertex, then visit all unvisited vertices that are adjacent to it before visiting a vertex which is two sports away from it.

With a queue: Add your starting vertex v to $\{\text{visited}\}$. Enqueue its adjacent vertices, then dequeue and visit that vertex if it hasn't been visited already. Enqueue its adjacent vertices. Dequeue the next one, visit, enqueue adjacent… and so on until everything has been visited.

Unexplored vertex: A vertex that hasn't been visited yet.
Visited vertex: A vertex that we've already included in our $\{\text{visited}\}$ set.
Unexplored edge: An edge that we have not visited yet.
Discovery edge: An edge that leads to an unexplored vertex.
Cross edge: An edge leading to an already visited vertex. They're always at the same or at an adjacent level.

Properties of BFS:

- Notation: G_s is the connected component of S.
- $\mathrm{BFS}(G, s)$ visits all the vertices and edges of G_s
- The discovery edges labeled by $\mathrm{BFS}(G, s)$ form a spanning tree T_s of G_s
- For each vertex v in L_i, the path of T_s from s to v has i edges, and every path from s to v in G_s has at least i edges.
- Setting/getting a vertex/edge label takes $O(1)$ time.
- Each vertex is labeled twice (once as unexplored, once as visited).
- Each edge is labeled twice (once as unexplored, once as discovery or cross)
- Runs in $O(n + m)$ time given the graph is represented in an adjacency list.

Applications: Using the template method pattern, we can specialize the BFS traversal of a graph G to solve the following problems in O(n+m) time:

- Compute connected components of G
- Compute spanning forest of G
- Find a simple cycle in G or report that G is a forest
- Given two vertices of G, find a path between them with the minimum number of edges, or report that no such path exists.

DFS vs BFS

Application	DFS	BFS
Spanning forest	X	X
Connected components	X	X
Paths	X	X
Cycles	X	X
Shortest paths		X
Biconnected components (if the removal of any single vertex, and all edges incident on that vertex, cannot disconnect the graph)	X	
Edges that lead to an already visited vertex	Back edge	Cross edge

Shortest Path

Shortest path describes the shortest path to the starting vertex.

Properties:

- A subpath of a shortest path is itself a shortest path.
- There is a tree of shortest paths from a start vertex to all other vertices.

Dijkstra's Algorithm

- The distance of a vertex v from a vertex s is the length of a shortest path between s and v. Dijkstra's algorithm computes the distances of all the vertices from a given start vertex s.
- **Assumptions**: Graph is connected, Edges are undirected, The edge weights are nonnegative.
- We can grow a cloud of vertices, beginning with s and eventually covering all the vertices. At each vertex v, we store $d(v)$, which is the best distance of v from s in the subgraph consisting of the cloud and its adjacent vertices.

- At each step, we add to the cloud the vertex u outside the could with the smallest distance label, and we update the labels of the vertices adjacent to u, ie we change the labels if there's a better way to get to the s vertex with the newly augmented cloud.
- We use a priority queue Q to store the vertices not in the cloud, where $D[v]$ is the key of a vertex v in Q.
- **Using a heap**: Add the vertex to the cloud (distance in the priority queue is 0). Then you $\mathrm{removeMin}()$ and update. Whatever $\mathrm{removeMin}()$ returned is the new vertex in your cloud. Updating means removing old keys and putting in new ones. Again, $\mathrm{removeMin}()$, add to cloud, update, etc. until the heap is empty.
- **Complexity**:
  - In a heap, it is $O(m \log n)$.
  - In an unsorted sequence, it is $O(n^2)$

Minimum Spanning Tree

Refers to the shortest way into the cloud, not to the root or starting vertex. Doesn't matter where you start, it'll be the same all the time. It's a spanning tree where you minimize the sum of the weights. Applications to communication networks and transportation networks.

Cycle property: Let T be a minimum spanning tree of a weighted graph G and e be an edge of G that is not in T and let C be the cycle formed by adding e to T, then for every edge f of C, $\mathrm{weight}(f) \leq \mathrm{weight}(e)$. In human language that means for every cycle, the largest weight is excluded from the minimum spanning tree.

Partition property: Consider a partition of the vertices of G into subsets U and V. Let e be an edge of minimum weight across the partition. There is a minimum spanning tree of G containing edge e.

Prim-Jarnik Algorithm

- We assume the graph is connected.
- We pick an arbitrary vertex s, and we grow the minimum spanning three as a cloud of vertices, starting from s. We store with each vertex v a label $d(v)$, which is the smallest weight of an edge connecting v to *any vertex in the cloud* (not the root as in Dijkstra).
- At each step, we add to the cloud the vertex u outside the cloud with the smallest distance label. We update the labels of the vertices adjacent to u.
- We use a priority queue Q whose keys are D labels, and whose elements are vertex-edge pairs. Key: distance, Element: vertex. Any vertex v can be the starting vertex. We still initialize all the $D[u]$ values to infinite, and also initialize $E[u]$ (edge associated with u) to null. It returns the minimum spanning tree T.
- It is an application of the cycle property.
- Complexity is $O(m \log n)$.

Kruskal's Algorithm

- Each vertex is initially stored as its own cluster.
- At each iteration, the minimum weight edge is added to the spanning tree if it joins two distinct clusters.
- The algorithm ends when all the vertices are in the same cluster.
- Application of the partition property.

- A priority queue stores the edges outside the cloud. Key: weight, Element: edge. At the end of the algorithm, we are left with one cloud that encompasses the minimum spanning tree, and with a tree T which is our minimum spanning tree.
- **Essentially**: start with the edge with the lowest weight, add it to the tree. Continue with the next lowest weight, and add it to the tree (if it does not form a cycle with existing edges of the tree). Continue until you've gone through all the edges. The resulting tree is your minimum spanning tree.
- Complexity is $O\big((n+m)\log n\big)$

Pattern Matching

Brute-Force

- Compares the pattern P with the text T for each possible shift of P relative to T, until either a match is found or all placements of the pattern have been tried.
- Compare, shift over one, compare, shift over one…
- Worst case: $T = aaa \dots aaa\text{h}$, $P = aaa\text{h}$
- Complexity $O(nm)$, where n is the size of the text and m is the size of the pattern that we're trying to find.

Boyer-Moore

- Based on two heuristics
  - Looking-glass heuristic: Compare P with a subsequence of T moving backwards.
  - Character-jump heuristic: When a mismatch occurs at $T[i] = c$, where c is the character in T at which the mismatch occurs (with $P[j]$):
    - If P contains c, shift P to align the last occurrence of c in P with $T[i]$.
    - If P does not contain c, shift P completely past $P[j]$ to align $P[0]$ with $T[i+1]$.

    If a match occurs, compare the previous two characters. Of they match, keep comparing right to left. If a mismatch occurs, do what it says up there.
- Worst case: $T = aaa \dots a$, $P = baaa$
- Complexity is $O(nm + |\Sigma|)$ where $|\Sigma|$ is the size of the alphabet you're using.

KMP Algorithm

- Compares the pattern to the text in left-to-right, but shifts the pattern more intelligently than brute force.
- Compare each letter of P, left to right, to T. When you find a mismatch, look at the word to the left of the mismatch. Find the largest prefix such that there is an identical suffix, and move P by matching up the suffix with the prefix.
- Failure function $F(j)$ is defined as the size of the largest prefix of $P[0 \dots j]$ that is also a suffix of $P[1 \dots j]$. Usually organized into a table. It is computed in $O(m)$ time.
- The complexity of this algorithm is $O(m+n)$.