Contents

1 Week 1, A **3**

2 Week 1, B **4**
- 2.1 Data Structures 4

3 Week 2, A **5**

4 Week 2, B **8**

5 Week 3, A **9**

6 Graphs **11**
- 6.1 Types of graphs 11
- 6.2 Graph representation 12

7 Graph traversal **12**
- 7.1 Depth first search 12
- 7.2 Breadth first search 13
- 7.3 Topological Sorting 13

8 Greedy Algorithms, Prim and Dijkstra **14**
- 8.1 Prim's Algorithm 14
- 8.2 Dijkstra's 15

9 Divide and Conquer **15**
- 9.1 Binary tree 15
- 9.2 Binary tree traversal 16
- 9.3 Closest pair problem 17
- 9.4 Master theorem 17

10 String search **19**

11 Dynamic Programming Part 1 **20**

12 Dynamic Programming 2 **21**
- 12.1 Warshall's Algorithm 21
- 12.2 Floyd's algorithm 22

13 Sorting **23**
- 13.1 Mergesort 23
- 13.2 Quicksort 25

14 Heap sort **28**

15 Distribution Sorts **30**
15.1 Counting sort 30
15.2 Radix sort 31

16 Binary Search Trees **31**
16.1 AVL trees 32

17 2-3 Trees and BTrees **34**

18 Hash Table **35**
18.1 Collision Handling 35
18.2 Hashing functions 36

19 Data Compression **37**
19.1 Fixed length encoding 37
19.2 Variable length encoding 38
19.3 Huffman encoding 38

20 Complexity Theory (Optional) **39**
20.1 Problems 39
20.2 Complexity classes 40
20.3 Reduction 40

1 Week 1, A

What is an algorithmic problem? It is a question that we wish to find generic answers to, and it implies a family of instances of a generic problem. An algorithm to solve such a problem must handle all instances of it.

What is an algorithm? It is a finite sequence of instructions with no ambiguity, works for all well-formed inputs, and terminates in a reasonable period of time.

Historically, algorithms are numeric: such as Euclid's algorithm to find the greatest common divisor. They work on numeric problems.

```
Require: $n \geq m \geq 0$
1: function GCD(n,m)
2:     if $m = 0$ then
3:         return $n$
4:     else
5:         return $gcd(m, n \mod m)$
6:     end if
7: end function
```

This has logarithm time complexity in $O(\log(\min(n, m)))$

We can also find the prime decomposition for both values and find the product of their intersections. To find a factor of a number, we can use the Sieve of Eratosthenes: constructs an array of length n and for each factor not visited, visit all multiples of the factor, until one factor multiples to n.

Today, algorithms are often executed logically, that does A imply B.

Complexity theory focuses on the "hardness" of certain problems.

The steps to problem solving are

- Understand the problem
- Decide on the type of computation (parallel or sequential, exact or approximate)
- Decide on the algorithm method (type of algorithms and their dependencies)
- Design the data structures and algorithm (in theory)
- Check for correctness and trace inputs
- Evaluate it analytically (time complexity)
- Code it
- Evaluate it empirically with sample data

Things in the subject

- Algorithm analysis
- Important algorithms in: sorting, searching, string processing, graph algorithms
- Approaches to algorithm design: brute force, decrease and conquer, divide and conquer, transform and conquer

2 Week 1, B

Problem 2.1 (The mutilated checkerboard problem)**.** *Can we cover an arbitrary checkerboard with finitely many domino pieces.*

There are a finite many (exponentially) number of coverings possible, so an inefficient brute-force method is doable.

It is easy to find that it could not be covered, if there are more odd numbered cells than even numbered cells (by coloring adjacent cells different colors). This is because a domino will necessarily decrease the odd numbered and even numbered cells by one each.

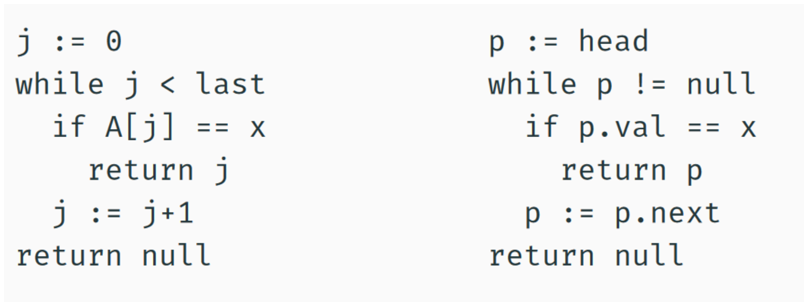
2.1 Data Structures

Algorithms transfers data; Data structures stores data in an optimal way for an algorithm.

An array is a sequence of consecutive cells in memory. It has constant lookup time and storage time, but can be difficult to maintain information in sequence.

A linked list contains a collection of objects stored in different memory locations, which links to each other. We can use a dummy first node to link to the first element. Inserting and deleting items is constant time, but indexing may take linear time.

To iterate through both an array and a linked list to find a specific element:

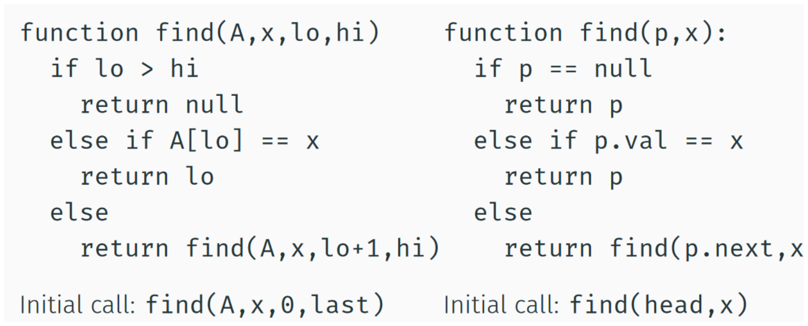


```
j := 0
while j < last
  if A[j] == x
    return j
  j := j+1
return null
```

```
p := head
while p != null
  if p.val == x
    return p
  p := p.next
return null
```

Figure 1: Iteration

We can also do it recursively (or with functional syntax)



```
function find(A,x,lo,hi)    function find(p,x):
  if lo > hi                  if p == null
    return null                 return p
  else if A[lo] == x          else if p.val == x
    return lo                   return p
  else                        else
    return find(A,x,lo+1,hi)    return find(p.next,x

Initial call: find(A,x,0,last)   Initial call: find(head,x)
```

Figure 2: Recursive iteration

Abstract data types are collections of data that has a family of operations on that data. It is crucial to separate the implementations of the operations and the concepts underlying each operation — things outside the specified operations in ADTs should not worry about the implementations.

A stack is a last-in-first-out structure, it is an ADT that is usually implemented by an array (insert at end) or a linked list (insert at head).

A queue is a first-in-first-out structure, it is similar in API and implementation as a stack.

Other DS includes: priority queue, trees, graphs.

3 Week 2, A

The algorithm efficiency is concerned with the resources of time and space.

To assess efficiency as a function of input size

- Theoretical or empirical assessment
- Average case or worst case

The program run time depends on

- Complexity of Algorithm
- Input size
- Underlying machine
- Language, compiler, operating system

We ignore 3 and 4, and only consider units of time. We wish to express 1 as a function of 2.

Assumption of the RAM Word Model

- Data is represented in words of fixed length of bits
- Fundamental operations on words take one unit of time: arithmetic, memory access, comparisons, logical and bitwise operators

Analysis of the time consumption is the counting of the operations with a given input size.

We can estimate the time consumption by letting c be the cost of a basic operation (the most expensive, most used) and $g(n)$ is the number of that operation for input sizes n. Then, the estimated running time is

$$t(n) \approx c \cdot g(n)$$

The running time can also depend on the type of input

- The worst case analysis makes the most adverse assumptions about the input
- The best case analysis makes the most optimistic assumptions about the input
- The average case analysis finds the expected running time across all possible input, which is equivalent to assume a randomly drawn input
- Amortized analysis spreads the cost of the algorithm over lots of function calls in a typical usage context

For the average case analysis, we have to consider the average time complexity across all inputs.

For example, the average case analysis for linear search can be derived by the following. Suppose p is the probability that the element is in the array, if the element is in the array, the average number of comparisons is $(n+1)/2$ and if the element is not, the average number if n. Taking the weighted average, we have the average case complexity is

$$\frac{p(n+1)}{2} + (1-p)n$$

We worry about the performance of the algorithm when the input sizes are big.

Some complexity functions are

- 1, running time is independent of input size
- $\log n$, divided and conquer solutions like binary search

- n, linear, where each input element is processed once
- $n \log n$, each element is processed once and involves other elements like sorting (divide and conquer but each recursion takes linear time)
- n^2, each pair of inputs are processed
- 2^n, each subset is processed
- $n!$, each permutation is processed

When analyzing the growth rate of algorithms, we ignore constant factors and ignore small input sizes. This asymptotic analysis.

Define $f(n)$ to be bounded in growth by $g(n)$ when

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = 0$$

note that

$$1 < \log n < n^c < n^{\log n} < c^n < n^n$$

and also

$$\log n < n^{0.00001}$$

because we are thinking about big numbers.

The set $O(g(n))$ denotes all functions that grow slower or equal to $g(n)$, asymptotically. Namely that

$$t(n) \in O(g(n)) \iff \exists c, n_0 \in \mathbb{R}, n > n_0 \implies t(n) < cg(n)$$

This implies that there exists a point where afterwards, $cg(n)$ will always be greater than $t(n)$.

The set $\Omega(g(n))$ contains all functions that grow faster or equal to $g(n)$. That is,

$$t(n) \in \Omega(g(n)) \iff \exists n_0, c \in \mathbb{R}, n > n_0 \implies t(n) > cg(n)$$

The set $\Theta(g(n))$ contains all functions that grow exactly at the growth rate of $g(n)$

$$t(n) \in \Theta(g(n)) \iff t(n) \in \Omega(g(n)) \wedge t(n) \in O(g(n))$$

that is, its upperbound can be $g(n)$ and its lower bound is $g(n)$.

So big-O is upperbound, big-omega is lowerbound, and big-theta is exact case. Note that all of these classes applies to all best case, worst case, and average case analysis.

Another simpler way to compute the asymptotic association by

$$\lim_{n\to\infty} \frac{t(n)}{g(n)} = \begin{cases} 0 & t(n) \in O(g(n)) \\ c & t(n) \in \Theta(g(n)) \\ \infty & t(n) \in \Omega(g(n)) \end{cases}$$

The proof is directly from the epsilon delta and definitions of the asymptotic classes.

4 Week 2, B

We can use L'Hopital's rule to compute this limit

$$\lim_{n\to\infty} \frac{t(n)}{g(n)} = \lim_{n\to\infty} \frac{t'(n)}{g'(n)}$$

Logarithm has some change of base identities

$$\log_a(x) = (\log_a b)(\log_b x)$$

There are two ways to compute time complexities: iteratively or recursively.

We can compute the exact time complexities of

- the maximum element in an array, $\Theta(n)$
- selection sort, $\Theta(n^2)$
- matrix multiplication, $\Theta(n^3)$

We can also find the time complexities of recursive function by recurrence analysis. Suppose the cost function (of factorial) is recursively defined by

$$\begin{aligned} M(0) &= 0 \\ M(n) &= M(n-1) + 1 \end{aligned}$$

The close form can be found by backward substitution (or telescoping down to zero) on the recursive part.

$$\begin{aligned} M(n) &= M(n-1) + 1 \\ &= M(n-2) + 2 \\ &= M(n-k) + k \\ &= 0 + n \\ &= n \end{aligned}$$

Hence

$$M(n) \in \Theta(n)$$

For binary search, the cost (of element comparisons) in the worst case where the element is not found

$$\begin{aligned} C(0) &= 0 \\ C(n) &= C(n/2) + 1 \end{aligned}$$

We can derive an explicit close form equation by telescoping.

The smoothness rule states that if we have a closed form equation when n is a power of 2, the same equation applies when n is not an exact power of 2.

Hence we have log cost for we are dividing every step

$$\begin{aligned} C(n) &= 1 + \log_2(n) \\ C(n) &\in \Theta(\log n) \end{aligned}$$

Big O summarizing

$$\begin{aligned} O(f(n)) + O(g(n)) &= O(\max\{f(n), g(n)\}) \\ c \cdot O(f(n)) &= O(f(n)) \\ O(f(n)) \cdot O(g(n)) &= O(f(n)g(n)) \end{aligned}$$

Stirling's formula

$$n! \in O(n^{n+1/2}) \in O(n^n)$$

Summation closed formed formulas.

5 Week 3, A

Brute force and exhaustive search algorithms are straightforward problem solving approaches, but they are slow.

Selection sort is brute force. It only makes n memory exchange operations. It is not stable in that sense.

Sorting algorithms in general has properties of

- In place if it does not require more than constant memory
- Stable if it preserves the relative order of elements with same keys

- Input-insensitive if the running time is independent to input properties other than size (independent to maximum input element, to order of input element)

Brute force string matching in general has $O(nm)$ where the pattern is size m and text is size n. If n is much greater than m, it will be $O(n)$.

For random text over a large alphabet, the average running time is linear in n. Because on average, we won't have as many matches as the worst case.

There are better algorithms for string matching on smaller alphabets like binary strings or DNA details.

The closest pair problem, find the pair which the minimum distance between them. The bruteforce algorithm checks every pair of points and maintain the min distance found so far. This takes $\Theta(n^2)$ where n is the number of points.

There is a divide and conquer algorithm in $\Theta(n \log n)$ using sorting and binary partitioning to solve this.

Exhaustive search focuses on problems like:

- Combinatorial decisions or optimization
- Search for an element with a particular property
- All permutaitons

This search approach generate and tests every possible solution.

The travelling salesperson problem finds the shortest Hamiltonian tour in a weighted undirected graph. To find all permutations of nodes will be factorial.

The knapsack problem has n items with weights, values, and a knapsack capacity. Find the most valuable selection of items that will fit under the capacity. To test every subset of items will take exponential time.

Exhaustive search algorithms only have acceptable running time on small instances. There are many better alternatives, but some problems are NP hard so have no better time complexities (then exponential).

An output sensitive algorithm has time complexity depending on the output side (solution size).

The Hamiltonian tour problem aims to find a tour in an undirected graph that visits each node exactly once. An Eulerian tour aims to find a path (not necessarily a tour) that visits each edge exactly once.

The Hamiltonian tour problem is NP-complete, because there are no better algorithms over exhaustive search. The Eulerian tour has a faster algorithm. It is common to find similar looking problems to have completely different complexity classes.

6 Graphs

Graph traversal is a method to exhaustive search, it aims to visit every node of a graph exactly once. This forms the backbone for a lot of algorithms.

Define a graph as an ordered tuple of vertices and edges. $G = (V, E)$. Where V is a set of nodes and E is a binary relation on V.

Symmetric closure on a binary relation means that flipping the input/output is also a valid edge. Transitive closure is functional composition, where $(a, b)(b, c) \implies (a, c)$

6.1 Types of graphs

Undirected graphs have all undirected edges. Directed graphs have all directed edges.

Connected graphs mean that there is a path from every node to every other node. Not connected graph is not connected.

If $(u, v) \in E$, then u and v are neighbours and are adjacent. the edge (u, v) connects u and v, and u and v are incident to the edge (u, v).

The degree of a node is the number of edges incident to it (number of edges connecting to it) in an undirected graph. For directed graphs, the in-degree counts the number of edges going to the node, the out-degree counts the number of edges going from the node.

A path in a graph is a sequence of nodes in V, where successive nodes are connected by edges. The path's length is the number of edges it passes (one minus nodes). A simple path has no repeated vertices. A cycle is a simple path where the first node is the last node.

Weighted graphs have weights between nodes, unweighted does not. Dense graph has the edges grow quadratic to the number of nodes, a sparse graph has edges grow slower or equal to linearly on the number of nodes.

A cyclic graph has at least one cycle. An acyclic graph (tree) does not have any cycle. Directed cyclic graphs has one or more directed cycles. The directed acyclic graph (DAG) has no directed cycles.

A tree is a connected acyclic graph. A rooted tree is a tree where one node is special. Every other node can be reached (simple path) from the root node. Removing the root node generates more rooted trees.

A rooted tree can be converted to a acyclic directed graph, but we usually rely on the layout and parental structure.

6.2 Graph representation

An adjacency matrix is a n by n matrix where A_{ij} is the weight of the directed edge from i to j, or zero if it doesn't exist. The matrix is symmetric for an undirected graph.

An adjacency list is a map from vertices to other vertices connected to it by an edge (or directed vertices in a directed graph).

An adjacency list is usually better for sparse graphs; an adjacency matrix is usually better for dense graphs.

To represent weighted graphs in these representations, we set the matrix element to be the directed weight, and use a struct with a weight in the linked list for the adjacency list.

7 Graph traversal

To explore all nodes in a graph from a starting node.

The breadth first search approach explores all the node's immediate neighbours before exploring their neighbours. We use a queue for this.

The depth first search approach explores all the node's neighbours' neighbours before exploring its own neighbours. We use a stack for this.

Both methods rely on marking nodes as we visit them.

7.1 Depth first search

DFS is based on backtracking and is easy to implement recursively. Its backtracking stack contains a list of nodes that lead to the current node.

The other notation from the book has assigns two indices for each vertex in the node, the index where it is pushed onto the stack, and the index where it is pop offed the stack.

To depict a depth first traversal, we can use a DFS tree (for a connected graph) and a DFS forest (for an unconnected forest). The tree nodes are the vertices, and tree edges indicates that the child is a neighbour node of the parent and is visited directly after the parent, with back edges as virtual edges connecting the nodes that are connected in the graph but not in the forest.

To do DFS on the graph have the same node ordering as performing DFS on the DFS forest/tree.

The depth first search algorithm uses recursion and a visited boolean array. It works for both directed and undirected graphs.

Under an adjacency matrix, the time complexity of dfs is $\Theta(V + V^2) = \Theta(V^2)$ for we need to consider all nodes and all edge entries in the matrix. With an adjacency list, we only need to consider all nodes and all edges, taking $\Theta(V + E)$.

To check if the graph is connected, run dfs on all node once, take the maximum vertices visited for all visits, and it against the total number of vertices. If it is not equal to the total number of vertices, the graph is not connected.

To check if the graph has a cycle, we place a check when visiting all neighbors and check if we've visited that node in the specific dfs search (we ignore visited nodes from previous dfs).

We can identify a directed acyclic graph if the graph is acyclic. This uses the previous algorithm.

7.2 Breadth first search

BFS uses a queue data structure to keep the next nodes to visit.

We can identify each vertex with a number, which is the order which the nodes are placed into the queue during bfs. The subscripts indicate when the nodes are placed into the queue.

The BFS search forest is a tree/forest where nodes parents are visited before their child in the search and are neighbors. The BFS forest does not have any back edges, but has cross edges between nodes on the same level (depth) which are virtual edges that are in the graph but not in the tree/forest.

BFS has the same time complexity as DFS, for adjacency matrices it is $\Theta(V^2)$; for adjacency lists, it is $\Theta(V + E)$.

To find the shortest path between two nodes, we do BFS and also keep track of the node depth in the search queue.

7.3 Topological Sorting

Suppose for a directed graph, we want to make a schedule where a directed edge from a to b implies that a is before b in the schedule array. This is a topological sort on the graph.

A topological sort exists if any only if it is a DAG. Finding the topological sort is to linearize the graph. It is to find an order of the nodes $v_1, v_2, \ldots, v_n$ such that for each edge $(v_i, v_j) \in E \implies i < j$.

To find a topological sort assuming that it exists, there are two algorithms: depth first search or the bruteforce algorithm.

The depth first search algorithm starts a depth first search on every non-visited node, and put order that the vertices are popped off the queue (when all its children are visited) into an array. We get a topological sort when reversing this array. This has the same time complexity as DFS.

The bruteforce algorithm notes that every DAG must have a source, a node with zero in-degree. So we iteratively remove the source in the graph and place it into an array until the graph is empty. This is a decrease and conquer algorithm with the worst case time complexity of $O(V^2)$.

8 Greedy Algorithms, Prim and Dijkstra

Greedy algorithm selects the local best choice at each step. We generally can't expect the local best solution construct a global best solution.

A priority queue is a set of elements with priorities. It supports the following operation on a set of elements

- Find an element with the largest priority
- Insert a new item with an associated priority
- Test whether the queue is empty
- Remove the highest priority element

Using a binary heap, we can do these operations in $O(\log n)$ and $O(1)$ time.

Stack is a priority queue where the "lateness" is the priority, a queue is a priority queue where the "earliness" is the priority.

A spanning tree on a connected graph (V, E) is a tree (V, E') where $E' \subset E$. This implies a connected and acyclic tree on the vertices of the original graph using only the edges from the graph.

Usually, edges has a weight/cost associated. Some spanning trees on weighted graphs will be better than other spanning trees. If the cost of a spanning tree is the sum of the edge costs in the tree, the minimum spanning tree is the smallest spanning tree in the weighted graph.

The MST is unique if all the edge weights are unique.

8.1 Prim's Algorithm

Prim's algorithm is a greedy algorithm that finds the minimum spanning tree (must be undirected). It constructs a sequence of subtrees T, each adding the closest node and its edge to a node in the last subtree. (For each step it picks the closest node to the subtree from outside the subtree and add that).

In each iteration, the tree grows by one edge. To find the minimum weight, we use a priority queue on the nodes not yet in the spanning tree with the edge cost to the tree as the priority.

Because we visit each edge twice and each node once, and for each edge/node, we add its target/node to the priority queue which has a maximum of E elements. This will take $\Theta((V+E)\log E) = \Theta((V+E)\log V)$ time (we have the second bound because $E \in O(V^2)$ so $\log E = 2\log V \in O(\log V)$).

8.2 Dijkstra's

Dijkstra's algorithm aims to find the minimum path to every point starting at a source. At every step, it selects the unvisited node with the minimum distance to the source node using a binary heap, and set the distance to that node to be that distance from the binary heap. We can store the previous node in a prev array if we wish to backtrack the path.

The algorithm has the same complexity as Prim's algorithm, with $\Theta((E+V)\log V)$ for the same reason. In particular, every node is inserted into the queue at least once, taking $\Theta(V\log E) = \Theta(V\log V)$; every edge is also visited twice (once from each node) with the worst case of updating/adding the node each time, taking $\Theta(E\log E) = \Theta(E\log V)$. Hence the algorithm takes $\Theta((E+V)\log V)$ overall.

Dijkstra's algorithm does not work with negative weights. It only works on zero or positive weighted graphs.

9 Divide and Conquer

Divide and conquer focuses on splitting the problem, then merging their solutions.

9.1 Binary tree

Define an empty binary tree (external node) with a height of -1, and a tree having the height 1 plus the max height of its subtree. The operation for this recursion is the empty tree check.

We let null pointers for a tree be external nodes, and the rest be internal nodes. The number of external nodes is always one greater than the number of internal nodes.

Proof. Let n be the number of internal nodes, and e be the number of external nodes. For every internal node, there is two other internal/external nodes. Every node will be counted once because their parent is an internal node (and thus visited), so they will be counted.

The only node not counted is the root node. Hence the number of nodes is $2n+1$. This is equal to the number of internal and external nodes, $n+e$, so

$$2n+1 = n+e \implies e = n+1$$

□

A full binary tree is where each node has exactly two or zero subtrees; a complete binary tree is where the tree fills the lower heights before the larger heights, from left to right, where the external nodes are all on heights h and $h-1$.

9.2 Binary tree traversal

Types of traversals over binary trees are

- Preorder traversal visits the root, then the left, then the right
- Inorder traversal visits the left, then the root, then the right
- Postorder traversal visits the left, then the right, then the root
- Level order traversal visits the nodes at the smaller heights before the higher heights, from left to right.

Preorder traversal, inorder traversal, and postorder traversal can all be implemented by dfs and recursion.

To do preorder traversal, we can also use a stack: while the stack is not empty, pop the element and visit it, then push the right and then left subtree on the stack, repeat.

To level order traversal, we replace the stack of the preorder traversal with a queue

```
function LEVELORDER(root)
    queue ← new queue
    queue emplace root
    while queue is not empty do
        value ← queue pop
        visit(value)
        if value left not empty then
            queue emplace value left
        end if
        if value right not empty then
            queue emplace value right
        end if
    end while
end function
```

9.3 Closest pair problem

We solve the closest pair problem using divide and conquer.

The bruteforce method takes $\Theta(n^2)$, we can use divide and conquer to get $\Theta(n \log n)$.

First sorted the points by x values, then sort the points by y value.

For each recursion call, find the median x of the elements considered. Process the elements to the left of the median to get d_l and the elements to the right of the median to get d_r. Now that the minimum distance of the points is either $d = \min(d_l, d_r)$ or two points spanning the left and right side (between $x - d$ and $x + d$). Now we scan over all pairs of points within that span in the order of their y value, taking their minimum paired distance. We can prove that for each point in the span, we only have to check its pairing with a constant number of next points, taking cn comparisons [1]. This creates a time complexity of

$$T(n) = 2T\left(\frac{n}{2}\right) + 8n$$

Specifically, we will maintain two sorted arrays of points in x and y. For every call, we find the number of points to the left/right of the x median sorted by x and y by scanning over the sorted lists and pass them to the recursive calls (n operations and $2T(n/2)$). Then we can process the center column ($8n$ operations), hence

$$T(n) = 2T(n/2) + cn \quad T(1) = 1$$

and by the master theorem

$$T \in \Theta(n \log n)$$

9.4 Master theorem

We can generalize the close form exact bound for a divide and conquer algorithm where we split the problem into b instances of size n/b and process only a instances, then taking $f(n)$ operations to merge the a instances.

This is the problem of solving for the bound of $T(n)$. Let $a \geq 1, b > 1$ and are integers, $f(n) \in \Theta(n^d)$ for $d \geq 0$, the recurrence

$$T(n) = aT(n/b) + f(n)$$

[1]Divide the spanning region into areas of 4 columns of squares of size $d/2$, there can be at most one element in each square due to the definition of d, hence for every element we only have to check the next 8 elements which are two rows below under the pairs exceed distance d

where $T(1)$ is a constant, the master theorem tells us that

$$T(n) \in \begin{cases} \Theta(n^d) & a < b^d \\ \Theta(n^d \log n) & a = b^d \\ \Theta(n^{\log_b a}) & a > b^d \end{cases}$$

this corresponds to the cases where $f(n)$ contributes more than the recursion, $f(n)$ contributes the same as the recursion, and the recursion contributes more than $f(n)$, respectively

Proof. Consider the recursion tree of $T(n)$. The tree has $1 + \log_b n$ levels with each level (0-indexed level) with level k having a^k nodes of complexity $f(n/b^k)$.

The last ($k = \log_b n$th) level has a^k number of $T(1) = c$, the operations at the last level sums to

$$a^{\log_b n} T(1) = cn^{\log_b a}$$

and the previous levels sum to

$$\begin{aligned} S &= \sum_{k=0}^{\log_b n - 1} a^k f(n/b^k) \\ &= \sum_{k=0}^{\log_b n - 1} a^k c_1 (n/b^k)^d \end{aligned}$$

Suppose that $a = b^d$

$$\begin{aligned} S &= \sum_{k=0}^{\log_b n - 1} (b^d)^k c_1 (n/b^k)^d \\ &= \sum_{k=0}^{\log_b n - 1} n^d \\ &= n^d \log_b n \end{aligned}$$

combined with the last level, this has the complexity of

$$\begin{aligned} T(n) &\in \Theta(cn^{\log_b a} + n^d \log_b n) \\ &\in \Theta(cn^d + n^d \log_b n) \\ &\in \Theta(n^d \log_b n) \end{aligned}$$

When $a < b^d$, the sum S will be around $c_2 n^d$ and the last level being less than cn^d, so netting a $\Theta(n^d)$. This is when $f(n)$ dominates.

When $a > b^d$, the sum S is still bounded by a polynomial multiple of n^d and the $n^{\log_b a}$ will dominate. Netting a $\Theta(n^{\log_b a})$ complexity. $\square$

10 String search

Some algorithms improve the average case time complexity of a problem, some algorithms improve the worst case time complexity of a problem.

The bruteforce string search algorithm is $O(nm)$, where n is the length of text and m is the length of pattern.

We can use input enhancement to make longer shifts on mismatch to improve the average case. The horspool's algorithm uses these shifts to make the search faster, and it also utilizes scanning the pattern from the right to left.

The cases for Horspool's algorithm is, starting by aligning the pattern with the text

- If the pattern's $m-1$ first characters does not match the text, shift the whole pattern by m positions
- If the pattern does not match and the last character is in $m-1$, shift the pattern until the last occurrence of the last text character.

So essentially shift the pattern fully if the last text character is not in the pattern, or until the last occurrence.

The preprocessing step of Horspool realizes that the shifts depend only on the character type in the pattern. So we

- Initialize a shift table of size 256 with all m
- Iterate the pattern from left to right, set the shift of that character to be its position to the end

The main algorithm of Horspool is

- Align the pattern on the left of the text
- Scan from the end of the pattern to the start
- On a mismatch, shift the pattern depending on the shift table on the last character in text
- Continuous until all characters in pattern matched

Worse case for Horspool is still $O(mn)$, and it happens when the text repeats characters and the pattern never matched. On random strings, it is linear.

Other string algorithms are

- Boyer Moore, extends Horspool to allow shifts on suffixes
- KMP, preprocess the pattern on a finite state automaton
- Rabin Karp, uses hash function to filter negative cases

11 Dynamic Programming Part 1

Dynamic programming can be used to solve the fibonacci numbers. It is because fibonacci numbers have overlapping subproblems.

With a recursive solution, we need $O(2^n)$ states. With a dp array to store the lower solutions, we only need $O(n)$ time. The time complexity goes from exponential to linear complexity, and the space complexity remains constant because we only need a constant number of previous solutions.

Dynamic programming solutions exist when

- The solution to a problem can be broken down into subproblems
- The subproblem solutions requires overlapping previous solutions

Dynamic programming is a design technique that trades memory for speed: breaking problem into subproblems, and store the overlapping solutions in memory.

The knapsack problem has n items with weights w_i, values v_i, and capacity W. We aim to find the most valuable selection of items that fits under the weight limit of the knapsack. Assume all values are positive integers. Essentially, we are aiming to find an indices set k_i where

$$\sum_i w_{k_i} \leq W \quad V = \sum_i v_{k_i}$$

with V is maximized.

The bruteforce solution tries every subset of items, taking $O(n2^n)$ time complexity for checking a solution takes $O(n)$ time.

The greedy algorithm aims to add the items in an arbitrary order if they fit. This does not grant an optimal solution.

But the idea of building up the knapsack item by item works when combined with the bruteforce method. That is to say that we have a dp table $dp[i][j]$ representing the maximum value of the first i items with a weight limit j, and we are finding $dp[n][W]$. The recursion is

$$dp[i+1][j] = \begin{cases} dp[i][j] & w[i+1] > j \\ \max(dp[i][j], dp[i][j-w[i+1]] + v[i+1]) & j \geq w[i+1] \end{cases}$$

The base cases are $dp[0][j] = 0$. We can iterate through i and j.

12 Dynamic Programming 2

The transitive closure is a binary function that checks if two nodes in a graph are reachable by edges through a path (transitive on the edge connection relation). We want to find the transitive closure matrix without brute force.

12.1 Warshall's Algorithm

We can find the transitive closure using dp (Warshall's algorithm). The idea is that the solution from $dp[i][j]$ is either if i and j are directly connected, or that there exists another node k where $dp[i][k]$ and $dp[k][j]$ are connected. This has overlapping subproblems if two paths both pass through a common vertex k. So we can dp on the vertices, where for each step, we incorporate the next vertex into the current transitive closures.

To dp, we assume vertices have an arbitrary order from $1..n$. Let $dp_k[i][j] = R_{ij}^k$ to be true if there is a path between i and j that goes through the subset of nodes from $1..k$ with us hoping to get the transitive closure R_{ij}^n, and we will dp on k. Suppose A_{ij} is the adjacency matrix denoting the connection from nodes i to j, we see that $R_{ij}^0 = A_{ij}$ for the only path passing no other nodes are direct paths. The recurrence step is

$$R_{ij}^k = R_{ij}^{k-1} \vee (R_{ik}^{k-1} \wedge R_{kj}^{k-1})$$

which implies that the pair (i, j) is reachable using the first k vertices if and only if it is connected using the first $k-1$ vertices (k not in the path) or k is in the path. If k is in the path, then both i to k and k to j are reachable using the first $k-1$ vertices (the subpaths must not use other vertices because the entire path must use only the first k vertices).

We can reduce the space complexity to $O(1)$ additional space by reusing the adjacency matrix. For the kth iteration, we directly update the matrix after every (i, j) for the only time it differs to saving R^{k-1} is when (i, k) or (k, j) are updated and other paths uses them. Both part doesn't matter for they don't change from the kth iteration (because neither path will become more reachable by including the additional element k for the endpoints are already k).

The pseudocode is

```
function WARSHALL(A[1..n][1..n])
    for $k \in 1..n$ do
        for $i \in 1..n$ do
            for $j \in 1..n$ do
                $A_{ij} \leftarrow A_{ij}$ or ($A_{ik}$ and $A_{kj}$)
            end for
        end for
    end for
end function
```

Notice that in the inner most loop, we dont have to check if A_{ik} or A_{kj} is false. And the A_{ik} condition does not depend on the inner j loop, so we can hoist it outside and ignore the j loop if A_{ik} is false. Additionally, assuming that A_{ik} is true, the j loop is an OR operation on row i against row k. So if we can represent each row as a bitmask, we can use a single OR operation instead.

The algorithm takes $\Theta(n^3)$ time where $n = |V|$. By hoisting the A_{ik} check outside, we can parallelize the outer i loop (because the inner part does not depend on other rows excluding k and i) and improve empirical performance.

The BFS/DFS method on each node is also $O(n^3)$ using adjacency lists (it is $\Theta(n^3)$ when the graph is dense, it takes $\Theta(n^2)$ for a sparse graph), but is empirically slower than Warshall's algorithm.

12.2 Floyd's algorithm

Floyd's algorithm generalizes the transitive closure, it solves the all-pair shortest path problem for weighted graphs of positive weights.

Similar to Warshall's algorithm, it uses a weight matrix W with ∞ as missing edges and 0 on diagonals, and continuously update/mins itself.

The recurrence relation is

$$D^0_{ij} = W_{ij}$$
$$D^k_{ij} = \min(D^{k-1}_{ij}, D^{k-1}_{ik} + D^{k-1}_{kj})$$

where D^k_{ij} is the minimum distance from i to j using the passing through subsets of the first k vertices.

This relation works for the shortest path including the first k vertices would be the minimum between the shortest path using first $k-1$ vertices and having k in the path.

The pseudocode is

```
function Floyd(W[1..n][1..n])
    for $k \in 1..n$ do
        for $i \in 1..n$ do
            for $j \in 1..n$ do
                $W_{ij} \leftarrow \min(W_{ij}, W_{ik} + W_{kj})$
            end for
        end for
    end for
end function
```

The reason we can reuse the matrix is similar to Warshall's: that the entries on the $k$ths row and column don't change including the node k.

To obtain the shortest, make a new matrix that stores the last element (predecessor nodes) in the shortest path and backtrack later by repeating searching for the last element.

The algorithm works for negative weights if there are no negative cycles. When there is a negative cycle, the shortest distances using that cycle will be negative infinity. We can detect negative cycles in Floyd's algorithm by checking if any of the matrix diagonals are negative (implying a negative cycle using that vertex exists), and terminate with error.

Both Warshall's algorithm and Floyd's algorithm on a graph G produces a new graph with the same vertices. The transitive closure graph contains an edge between two nodes (u, v) if there is a path from u to v in G, and the all-pair closest path graph has the weighted edge (u, v, w) when w is the shortest distance between (u, v) in G. The matrix output of the two algorithms forms the adjacency matrix of the new graphs.

13 Sorting

Sorting algorithms arrange elements with an order into order in an unordered array.

13.1 Mergesort

Mergesort is a divide and conquer approach to sorting: we sort the list on the left side and the right side, then merge the two sorted array.

The main mergesort call is

```
function MERGESORT($A[0..n-1]$)
    if $n > 1$ then
        $B \leftarrow A[0..n/2-1]$
        $C \leftarrow A[n/2..n-1]$
        MERGESORT(B)
        MERGESORT(C)
        MERGE($B, C, A$)
    end if
end function
```

The merge function uses an auxiliary array of size $\Theta(n)$. It compares the head of the two arrays and merge them one by one, then copying the rest unmerged elements to the end.

```
function MERGE($B[0..p-1], C[0..q-1], A[0..n-1]$)
    $i \leftarrow 0$; $j \leftarrow 0$
    while $i < p$, $j < q$ do
        if $B[i] \leq C[j]$ then
            $A[i+j] \leftarrow B[i]$; $i \leftarrow i+1$
        else
            $A[i+j] \leftarrow C[j]$; $j \leftarrow j+1$
        end if
    end while
    if $i == p$ then
        $A[i+j..n-1] \leftarrow C[j..q-1]$
    else
        $A[i+j..n-1] \leftarrow B[i..p-1]$
    end if
end function
```

There are two important properties for a sorting algorithm:

- An inplace algorithm: whether the operations happen within the input array or requires additional memory (non constant additional memory)
- A stable algorithm: for two equal elements, does the algorithm preserve their initial relative ordering after the sort

Merge sort is not in place for it requires $\Theta(n)$ additional space, it is stable and keeps relative ordering if we bias on the left array in the merge for equal elements (for two equal elements, copy the left element first).

To derive the mergesort time complexity, let $C(n)$ be the number of operations/comparisons for sorting an array of n elements. The recurrence relation is

$$C(1) = 0$$
$$C(n) = 2C(n/2) + n$$

by assuming n is a power of 2 using the smoothness rule, and noticing that merging always takes an order $\Theta(n)$ comparisons. Therefore, from the master theorem

$$C(n) \in \Theta(n \log n)$$

The time complexity does not depend on the elements.

Merge sort in practice has $\Theta(n \log n)$ complexity, can be highly parallelized, multiway (multiple splits) mergesort good for secondary memory to reduce copying/writing. In summary, mergesort is a good choice for a sorting algorithm if want stability and don't care about the extra memory.

13.2 Quicksort

An alternative divide and conquer sorting algorithm.

```
function QUICKSORT($A[l..r]$)
    if $l < r$ then
        $s \leftarrow$ PARTITION($A[l..r]$)
        QUICKSORT($A[l..s-1]$)
        QUICKSORT($A[s+1..r]$)
    end if
end function
```

The partition method splits the array with items less than the anchor on its left, and greater than the anchor on its right, then returns the anchor index.

Lomuto partition is a scanning partition algorithm that is not used in practice, only in theory.

```
function LomutoPartition($A[l..r]$)
    $p \leftarrow A[l]$
    $s \leftarrow l$
    for $i \in l + 1..r$ do
        if $A[i] < p$ then
            $s \leftarrow s + 1$
            swap($A[i], A[s]$)
        end if
    end for
    swap($A[l], A[s]$)
    return $s$
end function
```

Intuitive, the partition $l..s$ contains all the elements less than or equal to pivot p at all times at every iteration (and $s + 1..i$ contains the greater than pivot elements). If we find an element belong to the partition s, we swap the leftmost element not in the partition $s + 1$ with it, then set $s = s + 1$ to increase the partition size. At the end, we switch p with the last element in s for the remaining elements in s are all less than or equal to p.

Lomuto partitioning has a worst case complexity of $n - 1$ if the one of the partitions is essentially empty and the pivot element is the greatest element (every iteration will be a swap).

Quicksort is technically in-place for we are not using any additional arrays, it only requires $O(\log n)$ memory for the recursion stack. It is not stable for the arbitrary partitioning methods.

Hoare's partitioning improves on Lomuto's partitioning by getting more balanced partitions and better average time complexities.

The algorithm is

```
function HOAREPARTITION($A[l..r]$)
    $p \leftarrow A[l]$
    $i \leftarrow l$; $j \leftarrow r + 1$
    repeat
        repeat
            $i \leftarrow i + 1$
        until $A[i] \geq p$
        repeat
            $j \leftarrow j + 1$
        until $A[j] \leq p$
        SWAP($A[i], A[j]$)
    until $i \geq j$
    SWAP($A[i], A[j]$)
    SWAP($A[l], A[j]$)
end function
```

The intuition is that we want $l..i$ to contain the elements less than/equal to the pivot, and the $j..r$ to contain elements greater than the pivot. So while there is a gap between i and j, for each iteration, we shift i to the right as much as possible and j to the left as much as possible until an inversion. We then inverse the inversion by a swap, and continue.

We will always do one extra inversion when $i \geq j$ before termination, so we undo that after the loop and switch the pivot into j (which is the right most element less than or equal to the pivot).

Let $C(n)$ be the operations of quicksort in the best case. We will assume that all partitions are equal sized $n/2$ and takes exactly n comparisons for the partitioning for $i = j$ and they don't cross. The relation is

$$\begin{aligned} C(1) &= 1 \\ C(n) &= C(n/2) + n \end{aligned}$$

by the master theorem, the complexity is $C \in \Theta(n \log n)$.

In the worst case, we assume that all partitioning has one empty partition and takes $n + 1$ comparisons when $i > j$.

$$\begin{aligned} C(n) &= C(n-1) + n + 1 \\ C(1) &= 1 \end{aligned}$$

and by guessing, we see that $C(n) \in \Theta(n^2)$.

The average case for quicksort is $O(n \log n)$.

In practice, quicksort is the fastest sorting algorithm. We will use quicksort when speed matters and stability is not required.

Summary of all sorting algorithms

- Selection sort is slow and takes $\Theta(n^2)$ time, but only makes $O(n)$ key exchanges
- Merge sort is better for mid sized array and when stability matters
- Quicksort is empirically faster for large arrays where we dont care about stability

14 Heap sort

Priority queues is a queue data structure with quick insertion and max deletion operations. Its operations are

- Inject a new element with a key
- Eject the element with the highest priority key

a binary heap can do both these operations in $O(\log n)$ time.

Used in dijkstras and OS job scheduling.

A sorting algorithm using a max queue consists of treating the unsorted values in the array as a max queue, and iteratively moving the maximum element to the back.

- If we use an unsorted array as a max queue, we get selection sort
- If we use a binary heap, we get heap sort

The heap is a tree with the properties

- Binary with at most two children
- Complete, all levels are full except the last, where only the rightmost leaves are missing
- Parental dominance, the key of parent nodes must be higher or equal to its child keys (or all child node keys must be less than or equal to their parent key)

To do heap sort, we have to make the array into a heap, and iteratively eject the max element from the heap to the end.

The stupid top down method to heapify the array is to iteratively insert the elements of the array from left to right into the heap, then propagating it upwards to maintain the heap property. This takes n insertions with each insertion requiring $\log n$, a time complexity of $O(n \log n)$.

The better top down method to heapify the array is for each node we call heapify on, heapify the left and right child, then propagate the node downwards. This has the recurrence relation

$$T(n) = 2T(n/2) + \log n$$

which has the time complexity of $T(n) \in O(n)$. The first call is to heapify the root node.

The bottom up method to heapify array does the propagation down call in reverse order. We iterate the array from the last node to the first node, and propagate (bubble) down that node. This uses the property that a node's child always has higher indices than itself, hence by building the heap bottom up, we ensure that the children are always valid heaps. This has the time complexity of $O(n)$.

We do not need to prove the $O(n)$ time complexity of heapify.

To eject from a binary heap, we swap the root element with the last element (bottom right node) in the heap, then propagate (bubble) down the root element and ignoring the last element. This takes $O(\log n)$ to bubble down the new root element.

The iterative ejection from the binary heap has the time complexity of

$$\log(n-1) + \log(n-2) + \log(n-3) + \cdots + \log(1) \in O(n \log n)$$

we can prove this bound by

$$\begin{aligned} C &= \sum_{i=1}^{n-1} \log(i) \\ &\leq \sum_{i=1}^{n-1} \log(n-1) \\ &\leq n \log n \in O(n \log n) \end{aligned}$$

and this is only an upper bound.

The heap sort algorithm first heapify the array in $O(n)$, then iteratively eject the max element in $O(n \log n)$ time. This has the total time complexity of $O(n \log n)$.

To represent a heap, we use an array with the nodes in level order from left to right. In the lectures we start at index 1. The children of node i are in indices $2i$ and $2i + 1$, and the parent is in $i/2$. The heap condition using an array representation is that for all $i \in 1, \ldots, n$, the node is less or equal to its parent $A[i] \leq A[i/2]$.

The heapsort property is

- It is in place
- Not stable for the random heapify element changes

Empirically, heapsort is slower than other $O(n \log n)$ algorithms but has low memory need and guaranteed $O(n \log n)$ time complexity.

15 Distribution Sorts

15.1 Counting sort

Counting sort builds a distribution function on the array elements through an auxiliary array, then maps each element in the original array by the quantile they belong to via the df.

The algorithm is

```
function COUNTINGSORT(A[0..n-1])
    $m \leftarrow \min A$
    $M \leftarrow \max A$
    $D[0..(M-m)] \leftarrow 0$
    for $a \in A$ do
        $D[a-m] \leftarrow D[a-m]+1$
    end for
    for $i \in 0..(M-m-1)$ do
        $D[i+1] \leftarrow D[i+1]+D[i]$
    end for
    $R[0..n-1] \leftarrow 0$
    for $i \in n-1..0$ do
        $R[D[A[i]-m]-1] \leftarrow A[i]$
        $D[A[i]-m] \leftarrow D[A[i]-m]-1$
    end for
    return $R$
end function
```

which requires allocating a distribution array and an output array.

Suppose n is the length of the array and k is the spread between the min and max elements, the time complexity of counting sort is $\Theta(n+k)$, the space complexity is $\Theta(n+k)$ due to the allocation of the output array and distribution function.

Assumptions of counting sort

- Elements have integer keys
- The range of keys is small/fixed

Properties of counting sort

- Not in place, requires allocating an output array
- Stable sort

15.2 Radix sort

Radix sort on items with dictionary keys (strings) sorts the items in lexicographic order (character wise comparisons). The idea is to run a stable sorting algorithm, commonly counting sort, on the key characters starting at the least significant character.

The reason is that stable sorts always leaves elements with the same character value in the same order, which is the order of their less significant characters, and also sorts the character values that are not equal without considering less sig characters. This is exactly how a lexicographic order is computed, hence sorting the entire array.

We can use radix sort on integers using their binary values, or on strings using their character list representation.

The time complexity of radix sort is $\Theta(d(n+k)) = \Theta(nd + kd)$ if n is the length of the array, d is the length of key, k is the spread between the min and max character. The space complexity is $\Theta(n+k)$.

Radix sort has properties

- Stable sort
- Not in place

It is useful when the key length d is fixed/small and size of alphabet k is small (english strings), then it will take approximately linear time on n.

In practice

- Distribution sorts faster than comparison algorithms when keys and alphabets are small
- distribution sorts are less general, and good sorting algorithms can come very close to linear performance
- they can be useful as part of a more complex algorithm (suffix arrays), and in controlled environments with guaranteed key lengths

16 Binary Search Trees

Dictionaries are

- Abstract data structures

- Collection of key value pairs, values are records, keys are unique identifiers
- Operations are: search with key, insert key value pair, delete with key

Implementation of dictionaries

- Unsorted array, searching will take $\Theta(n)$ comparisons
- Sorted array, insert will take $\Theta(n)$ swaps
- Binary search tree can be modified to have $\Theta(\log n)$ search, insertion and deletion time

The simple bst is a tree with the order property: for every node, all its left children are smaller or equal to it, and all its right children are greater to it. The operations

- Searching is a recursive call on each node to search the left or right node
- Insertion does searching, then insert the new element as a leaf
- Deletion has three conditions: if the deleted element is a leaf than just remove it, if the deleted element has one child just attach that child to its parent, if the deleted element has two children, swap the element with the successor, then perform the same deletion algorithm on the element

But simple bst can be unbalanced and have $O(n)$ search complexity.

To avoid the degeneracy case we can use self balancing trees (AVL and red-black tree) or a change of representation (B trees).

16.1 AVL trees

An AVL tree is a BST where each node has a balance factor: the difference in height between the left and right subtree with positive values indicating that the left subtree is taller. The balance factor can only be -1, 0, or 1, if it exceeds this range, we need to rotate the tree to rebalance it.

Searching is done as a BST. Insertion and deletion also uses the BST, with additional steps at the end to update the balance factors, and rotate the tree if needed.

The rotations are categorized into: R, L, LR, RL rotations.

The R rotation works when the node has 2 balance, and the left child does not have a -1 balance. It does a single right tree rotation and lifts the left child to the root and the root node as the right child of the new root. The L rotation is similar and is performed when the node has -2 balance, and the right child does not have a 1 balance.

The LR rotation is a left rotation on the left child, followed by a right rotation on the root node. Use a LR when the root is 2 but the left child has a -1 balance. A RL rotation is similar, using a right rotation on the right child and a left rotation on the root. It is used when the root has -2 balance and the right node has a 1 balance.

When performing multiple rotations, we always start at the lowest (largest depth) unbalanced subtree to rebalance. This implies that we can balance and recompute the balance factors starting at the insertion/deletion location, and bubble upwards through its parents rebalancing/updating along the way.

The entire algorithm can be summarized as

- Insert and delete using BST
- For the final deletion or insertion node, update the balance factor, and rebalance if needed using one of the four methods, update the balance factors in constant time
- Repeat for the parent, and the parent of its parent, all the way to the root

To implement the balance factor updating after insertion/deletion and rebalancing, we keep a height number for each node. To compute the balance factor, we compare the heights of the left and right nodes. To update the heights, we can update the lowest node after the insertion/balance through its children and then the higher nodes.

Rotations ensure that an AVL tree is always balanced. An AVL tree with n node has depth $\Theta(\log n)$. This ensures that it has $\Theta(\log n)$ swaps for each insertion and deletion, and $\Theta(\log n)$ comparisons for each search.

Insertion has $\Theta(\log n)$ worst case complexity because we need to search down to insert, taking $\log n$, then update the balances for all its parents in constant time, taking another $\log n$. Deletion has $\Theta(\log n)$ worst case complexity because we need to search for the node swapping along the way taking $\Theta(\log n)$

Alternative self balancing tree is a red black tree. The main idea is that the longest height is at most twice as long as the shorted height, oppose to $+1$ and -1 for AVL. Most ordered dictionaries in programming languages use red-black trees.

The comparison between red-black trees and AVL are

- AVL trees are more balanced but require more frequent rotations. They are better if searches are more frequent than insertions/deletions.
- Red-black trees are less balanced but require less rotations. They are better if insertions/deletions are more frequent than searches

17 2-3 Trees and BTrees

A 2-3 is a BST with a change of representation, keeping the tree balanced. Instead of using rotations, we allow multiple elements per node and multiple children per node.

A 2-3 tree contains a maximum of two elements and three children. A 2-node is a node with two children, a 3-node is a node with three children. This can be extended to a b tree allowing n-nodes.

Searching in a 2-3 tree does a recursive search down the 2-3 tree. If it is a 2-node, we do the BST recursion. If it is a 3-node, we check if the key is lower than the first element, between the first element and the second, or larger than the second element, and recurse on their child nodes.

Insertion in a 2-3 tree first search for the node to insert the new element. If the node is a 2-node, we make it a 3-node with the new element. If the node is already a 3-node, we temporarily change the node to contain 3 elements, then promote the middle element to its parent. If the promotion is done on the root node, we create a new 2-node and make 2-nodes for the two remaining elements and attach them. If the promotion is to a 2-node, we change that to a 3-node, and create new 2-nodes for the remaining elements and attach to new 3-node. If the promotion is to a 3-node, we temporarily change it to contain 4 elements, promote it recursively, then make new 2-nodes for the remaining elements and attach it to the 3-node.

Notice that the 2-3 tree after insertion is always complete with no holes. Therefore the height is always $\Theta(\log n)$, allowing the three operations to take $\Theta(\log n)$ in the worst case.

Deletion is more complex than AVL trees, they are not covered.

Extensions for the 2-3 trees are

- 2-3-4 trees includes 4-nodes, used to reduce the number of promotions
- B-trees are generalizations of 2-3 trees. A btree with order n can have n-nodes
- B plus trees are b trees where internal nodes only contain keys in main memory, and values are all in the leaves in secondary memory

The key property for b-trees are that balances are maintained by allowing multiple elements per node.

Comparing against self-balancing trees

- Self balancing trees are better when the dictionary is fully in main memory, because both comparison and modifications are cheap

- Multiple elements per nodes are better when the dictionary is in secondary memory, because comparison is cheap but modifications are expensive, B-trees are used in SQL databases and file systems for they are secondary memories

18 Hash Table

Hash table implements the dictionary abstract data type. It is a continuous data structure with pre allocation of m buckets, with a hash function and collision handling. The time complexity for searching, insertion, and deletion is

- On average, $\Theta(1)$
- The worst case, $O(n)$

assuming that hashing takes constant time $\Theta(1)$ on the keys and is a good hash function $h(K) \to [0, m-1]$ that distributes keys evenly between the range.

Collisions are what happens when key hashes are equal for non-equal keys. The solutions create subclasses of hash table implementations.

In practice, the efficiency will depend on the table load factor $\alpha = n/m$, which is the number of keys divided by the number of buckets.

18.1 Collision Handling

Separate chaining handles collision by creating a linked list at every bucket location containing all the keys that hashes to it. The searching, insertion, and deletion will operate on the linked list in linear time. Assuming an equal distribution of keys, the time complexities are

- Successful search requires $1 + \alpha/2$ operations on average
- Unsuccessful search requires α operations on average
- Same complexity for insertion and deletion

The algorithm will hit worst case mostly when the hash function is bad (the load factor is generally more important). It requires additional memory to store the chains.

Linear probing is an open addressing collision handling method. Each cell has one of three states: empty, deleted, filled. The operations are implemented as

- Search starting from the hash location and linearly search for the value if not found, stopping at an empty cell. A successful search will take $(1/2)(1+1/(1-\alpha))$ operations on average. An unsuccessful search will take $(1/2)(1 + 1/(1 - \alpha)^2)$ operations on average.

- Insertion starts at the hash location and linearly search for an empty/deleted cell. This has similar operations as searching.
- Deletion starts at the hash location and linearly search for the value then setting that cell to deleted, also stopping at an empty cell. The complexity is complicated

The method does not require extra memory. It has worst case $\Theta(n)$ operations with a bad hash function and/or has clusters in the table. Linear probing is generally more sensitive to higher load factors due to the $1/\alpha$ behavior instead of a linear α relation of the separate chaining model.

Double hashing generalizes linear probing. The successive locations to probe are

- $h(K)$
- $h(K) + s(K) \mod m$
- $h(K) + 2s(K) \mod m$

for the hash functions $h(K), s(K)$, and the table size m. The implementation is identical to linear probing. This is called open addressing because we are open to changing the address of the key value pair, while separate chaining is a close addressing method because we are closed within the cell for the hash computed.

Rehashing reduces the performance deterioration of high load factors. It allocates a new table around double the original size, and reinsert every item from the previous table to the new one. It is a really expensive operation but is very infrequent.

18.2 Hashing functions

To hash integers that may be large or unbounded, we use

$$h(K) = K \mod m$$

for the table size m. If m is small, we will lots of collisions; if m is large, it will have excessive memory. We usually want m to be a prime number, this implies that when we rehash the table, we want the next prime bigger than $2m$ and use that for the size.

To hash strings, we assign each character to a bitstring and concat them modded by the table size. Suppose that $chr(C)$ gives the number for the character, the hash is

$$h(s) = \sum_{i=0}^{|s|} chr(s_i) p^{|s|-i-1}$$

where $p = 32$ must be greater than the alphabet size and represents the shift for each bitstring.

Horner's rule gives a faster method to compute the hash, it states that

$$a_n p^{n-1} + a_{n-1} p^{n-2} + a_{n-2} p^{n-3} + \cdots = ((a_n p + a_{n-1})p + a_{n-2})p + a_{n-3} + \ldots$$

and recurse. We then apply mod on each operation/sub-expression recursively because of the modulo properties.

Summary, hash tables:

- implements dictionary
- Has average $\Theta(1)$ search, insert, and delete operations
- Preallocates memory m
- Needs a good hash function
- Needs good collision handling

Compared to BST

- Hash tables takes more memory compared to bst
- Hash tables ignore the key ordering, queries of the form: return all values in the key range, is hard implement
- Usually if hashing is available and ordering is not needed, they perform empirically faster

In practice, python dictionary uses open addressing and rehash when $\alpha = 2/3$, c++ unordered map uses separate chaining and rehash when $\alpha = 1$.

19 Data Compression

We've only analyzed algorithms from a speed and space performance, but we've assumed that the records it uses can fit in memory. So what if we have large records?

19.1 Fixed length encoding

Fixed length encoding encodes a list of characters by assigning the same length code (bits) to each character then concatenating them. ASCII uses fixed-length encoding, but this coding has redundant information (information that is not needed).

Run-length encoding aims to reduce redundancies by representing runs of the same character as a number followed by a character (denoting number times character), then run a fixed-length (or other) encoding on the resulting output.

Run-length encoding works well on text with small alphabets and lots of repeats; it can work on binary data or DNA. It does not work well on human text.

19.2 Variable length encoding

Variable length encoding uses the idea that some characters appear more frequently than others, so we use a variable number of bits for each character by

- More frequent symbols use fewer bits
- Less frequent symbols use more bits

This encoding scheme requires a prefix-free code, that no symbol code is a prefix of another symbol's code.

To encode a string with a code

- Store the code in a dictionary with characters as keys and code as values
- For each character in text, pick the code from the hash table, then concatenate the entire coding

To decode a string from its encoding

- Construct a trie representing the coding dictionary
- Iterate over the encoding and traverse down the trie following the bit values, adding the character to the output and restarting the trie from the root if reached a leaf

with could also iteratively widening the window and check if the bits is a valid code, but it will have a lot of dictionary misses.

Tries are types of trees where each edge represents a specific character or bit, and nodes can contain values. A trie implements a dictionary when the keys can be decomposed into characters in some order within the key (strings).

By creating a trie where the values are only in leaves, we maintain the prefix property of the coding. The binary prefix-free trie that represents a coding has the left edge represent 0 and the right edge represent 1, with the character values only at the leaves. The path taken from the root to get to a character forms the code of the character.

19.3 Huffman encoding

Huffman encoding obtains the optimal encoding dictionary given the text using symbol frequencies. The optimal encoding will generate an encoding with the shortest bits.

The algorithm to build the prefix binary trie is

- Treat each symbol as a leaf with its frequency value, add them into a priority queue
- Iteratively fuse two of the smallest frequency valued nodes from the queue, fusing implies creating a parent node with the two smallest nodes as its children, and the

value being the sum of their frequencies, then readding the parent node to the priority queue

- When the size of the queue is one, we get a huffman tree

On paper, arrange the characters in increasing frequency order. For each fuse, draw a parent node on top connecting the two child nodes with lowest frequency with the value being the new frequency. The resulting binary tree structure is the huffman tree with the leafs being the initial set of characters and the branches being zero on left and one on right.

Data compression in practice

- Lempel-Ziv compression assigns codes to sequences of symbols. Used in GIF, PNG, and ZIP
- For sequential data like audio and video, the alternative is linear prediction which predicts the next frame/data given the previous one. FLAC uses linear prediction
- Lossy compression in audio and video also uses huffman as the last step. Used in JPEG, MP3, MPEG

20 Complexity Theory (Optional)

Complexity theory focuses on the inherit difficulty of the problem instead of the time complexity of algorithms. It focuses on how hard problems are to solve.

It uses asymptotic notation to denote the lower bound of the problem. We aim for tighter lower bounds to verify that our algorithm is theoretically optimal. A tight bound implies there is an algorithm with worst case complexity of that time.

Comparison sorting has a trivial lower bound of $\Omega(n)$, a less trivial lower bound is $\Omega(n!) = \Omega(n \log n)$ found by a decision tree (binary search) on all permutations of the array. We have a tight bound because there is a $\Theta(n \log n)$ algorithm implying that the algorithm is theoretically optimal.

Matrix multiplication has a trivial lower bound of $\Omega(n^2)$. But we don't know if the bound is tight for the current best algorithms are $O(n^{2.37})$.

20.1 Problems

A decision problem takes an input and returns a yes or no answer.

Optimization problems can be framed as a sequence of decision problems in its dp. Knapsack is a sequence of decision problems of: is there a set of items of values at least i with weight at most j.

A verification problem takes an input, a proposed solution, and verifies if the solution satisfies the input (if the solution is valid).

20.2 Complexity classes

A problem is in P if its decision version has a polynomial time solution on the input size. They are easy problems.

A problem is in NP if its verification version has a solution in polynomial time on the input size.

We can turn solving a decision problem to solving a sequence of verification problems.

- A non-deterministic machine (theoretical machine) generates a candidate
- The verification algorithm verifies the solution
- Repeat until verified

We can then say that a problem is in NP if its decision version has a solution in non-deterministic polynomial time on the input size. It is non-deterministic because we don't know if generating the solutions is in polynomial time.

All problems in P is in NP, because we can verify a P solution by solving the decision version and comparing the result.

The reverse is unknown (if NP is P), because the non-deterministic step does not provide any efficiency guarantees (we dont know if we can guess a solution in polynomial time).

A decision problem is in NP-Complete when

- It is in NP
- Every problem in NP has a polynomial reduction to it

We can also show NP-completeness if there is a polynomial reduction from an existing NP-complete problem.

The key idea is that if there is a polynomial time solution to any NP-Complete problem, then all NP problems can be solved in polynomial time.

NP-Hard problems implies that every problem in NP has a polynomial reduction to it, but it doesn't need to be in NP. NP-complete is the intersection between NP and NP-Hard.

20.3 Reduction

It is difficult to prove bounds for every new problem.

Reduction allows us to frame an unknown problem as equivalent to another problem we know the class of.

The hamiltonian circuit problem can be reduced to the decision version of TSP (we can frame solving hamiltonian circuits as solving TSP). The reduction step is polynomial time, so TSP is also in NP because HAM is in NP.

The reduction from HAM to TSP is

- Build a new graph where connected nodes in G has an edge of weight 0, and non-connected nodes has an edge with weight 1. This is polynomial time
- We can frame solving HAM as solving the decision-TSP "Is there a circuit that visits all nodes once with at most weight 0".

3SAT is the first NP-Complete problem. It is the Boolean 3-satisfiability Problem. The 3SAT decision version is: given a boolean formula with a maximum of three literal, is there an assignment of boolean variables that results in true.

The sequence of 3SAT reduction to other problems results in a sequence of NP-Complete problems

- SAT
- Clique
- Vertex cover
- Hamiltonian circuit
- Decision TSP

A polynomial time algorithm to any of these NP problems implies P is NP.

Some problems are undecidable (there are no solutions). Some tried to prove that the P is NP problem is undecidable.