Algorithms

based on *Introduction to Algorithms* by Cormen et al

Kanyes Thaker

Last updated: May 19, 2024

Note to the Reader

Algorithmic thinking is a critical tool to create systems which are efficient and reliable. The purpose of these notes is to capture high-level patterns of algorithmic problem solving that are widely applicable to many problems both in and out of computer science. These notes are taken from Cormen et. al.'s *Algorithms* (CLRS), one of the most popular introductory algorithms textbooks.

These notes are not a comprehensive retelling of the text. Instead, they attempt to draw out the most relevant and widely applicable algorithmic patterns that I have observed in my time in the engineering world. Special topics, such as parallel algorithms, machine learning, combinatorics, and linear programming, are left to other texts which specialize in those fields. Topics included in these notes include Kolmogorov complexity, sorting, divide-and-conquer algorithms, common efficient data structures, dynamic programming, greedy algorithms, graph algorithms, and a brief overview of computational complexity theory.

These notes assume an undergraduate level familiarity with probability, discrete mathematics, and data structures.

Contents

List of Algorithms

Part I

Preliminaries

❊ Introduction

An **algorithm** is any well-defined procedure that takes in some **input** to produce an **output**. They are tools that assist in solving particular kinds of computational problem. Algorithms work hand-in-hand with **data structures**, which are formats for organizing information to make solving these problems easier. This text provides a general overview of common basic algorithms, but the world of algorithms is unbounded – the main purpose of studying existing algorithms is to learn patterns of thought from which you can create algorithms of your own. Additionally, it is critical to make algorithms *efficient*, so that they may be executed quickly and with minimal expenditure of resources.

As a gentle introduction, take **insertion sort**. It is an algorithm designed to solve the **sorting problem** of organizing a collection of objects where each pair of objects has an order relationship. The input is a sequence of n numbers, and the output is a reordering of those numbers in increasing order.

Algorithm 1 Insertion Sort

```
procedure INSERTION SORT($A[1..n]$)
    for $j \leftarrow 2..A.length$ do
        $key \leftarrow A[j]$
        $i \leftarrow j - 1$
        while $i > 0$ and $A[i] > key$ do
            $A[i+1] \leftarrow A[i]$
            $i \leftarrow i - 1$
```

Insertion sort goes through the collection from left to right. At each step j, we store the element j temporarily. We then look at the elements we've sorted so far (to the left of j), and make room for j by taking all elements greater than j and moving them up one space. Then we insert j into the empty slot. At the first time step, we have 1 element, which is obviously in sorted order. Assuming that at time $j - 1$ the elements on the left are sorted, the above reasoning shows that the algorithm puts j into a sorted order. Then, at the end of the procedure, the whole array must be in sorted order. This **proof by induction** serves as the **proof of correctness** for this algorithm.

Also essential to the analysis of algorithms is **runtime analysis**, a measure of how many "steps" were taken to complete the procedure. Normally we concentrate on the **worst-case running time**, i.e. the longest running time for any input, to give us an upper bound on performance. These values are usually algebraic functions represented in terms of their asymptotically greatest term, i.e. if a function takes $an^2 + bn + c$ operations for n inputs, we only report the n^2. Insertion sort does n work to iterate through the whole array, and n work to swap all previous elements on each iteration, meaning its **asymptotic runtime** is $\Theta(n^2)$.

Formally, $\Theta(g(n))$ means there exist positive c_1, c_2 for which $c_1 g(n) \leq g(n) \leq c_2 g(n)$ for all n. This is an asymptotically **tight** bound. If we cannot guarantee a lower asymptotic bound, we would write $O(g(n))$. If we cannot guarantee an upper asymptotic bound, we would write $\Omega(g(n))$.

Insertion sort uses an incremental approach, where we sort a subarray and incrementally sort the next item, growing that subarray. Alternatively, we might choose a **recursive** approach, where we split a problem into smaller subproblems over and over until we reach a very easy problem, and then combine the results to solve the original problem. This approach is usually dubbed **divide-and-conquer**. As an example, take the following sorting algorithm:

Algorithm 2 Merge Sort

```
procedure MERGE($A, p, q, r$)
    $L, R \leftarrow A[p..q], A[q..r]$
    $i, j \leftarrow 1, 1$
    for $k \leftarrow p..r$ do
        if $L[i] \leq R[j]$ then
            $A[k], i \leftarrow L[i], i + 1$
        else
            $A[k], j \leftarrow R[j], j + 1$
procedure MERGE SORT($A, p, r$)
    if $p < r$ then
        $q \leftarrow \lfloor (p + r)/2 \rfloor$
        MERGE SORT($A, p, q$)
        MERGE SORT($A, q + 1, r$)
        MERGE($A, p, q, r$)
```

The **merge sort** algorithm recursively subdivides the array by repeated calls to itself, at each time splitting into two left and half subarrays. The splitting is tracked by the indices p, q, r. Once the arrays reach length 1, no more splitting can be done, so the length-one arrays are merged. The merge step assumes that the array A is composed of two sub-arrays that are each sorted, and stores them as two separate arrays L and R temporarily. It then iterates through each element of L and R, and places the smallest from each back into A, zipping the two arrays together. A simple proof by induction proves the correctness of the algorithm. Since an array of length n can be divided at most $\log n$ times, and there is n work done on each iteration of the merge step, this algorithm is at worst $O(n \log n)$.

※ Divide-and-Conquer

Divide-and-conquer algorithms solve complex problems by dividing the problem into smaller subproblems that resemble the bigger problem, recursively solving the subproblems, and then combining the solutions to solve the largest problem. In general, these problems can be described by a **master equation** of form $T(n) = aT(n/b) + O(n^d)$ – we split the problem into a subproblems of size n/b each, and do $O(n^d)$ additional work to combine the results.

As an example, suppose you are tracking a stock, and have the ability to buy shares on one day and sell on any later date over n days. To maximize profit, you would want to find the two days where the difference in price is the greatest. A brute force approach would compare all $\binom{n}{2}$ options, which is $\Omega(n^2)$. Notice that the difference in price between days i and j is the sum of the *daily changes in price* for all of those days. (i.e. the profit between Monday and Wednesday is the price change from Monday to Tuesday plus the price change from Tuesday to Wednesday). So an alternate way to frame this problem is to find the **maximum sub-array** (by sum) within the array of daily price changes.

Suppose a divide-and-conquer approach where we split the array neatly in two. Then the maximum subarray must live entirely in the bottom array, live entirely in the top array, or straddle the splitting point. We can handle the midpoint case explicitly, and then recursively handle the other cases.

Algorithm 3 Maximum Crossing Subarray

```
procedure MAX CROSSING SUBARRAY($A, low, mid, high$)
    $leftSum, sum = -\infty, 0$
    for $i \leftarrow mid$ downto $low$ do
        $sum = sum + A[i]$
        if $sum > leftSum$ then
            $leftSum, maxLeft = sum, i$
    $rightSum, sum = -\infty, 0$
    for $j \leftarrow mid$ to $high$ do
        $sum = sum + A[j]$
        if $sum > rightSum$ then
            $rightSum, maxRight = sum, j$
    return $(maxLeft, maxRight, leftSum + rightSum)$
```

The crossing algorithm takes $O(n)$ time, since it involves at worst checking each element in the array. All other comparison operations are $O(1)$. And since we split the size

Algorithm 4 Maximum Subarray

```
procedure MAXIMUM SUBARRAY($A, low, high$)
    if $high == low$ then return $(low, high, A[low])$
    else
        $mid = \lfloor (low + high)/2 \rfloor$
        $leftLow, leftHigh, leftSum$ = MAXIMUM SUBARRAY($A, low, mid$)
        $rightLow, rightHigh, rightMid$ = MAXIMUM SUBARRAY($A, mid + 1, high$)
        $crossLow, crossHigh, crossMid$ = MAX CROSSING SUBARRAY($A, low, mid, high$)
    if $leftSum \geq rightSum$ & $leftSum \geq crossSum$ then
        return $(leftLow, leftHigh, leftSum)$
    else if $rightSum \geq leftSum$ & $rightSum \geq crossSum$ then
        return $(rightLow, rightHigh, rightSum)$
    else
        return $(crossLow, crossHigh, crossSum)$
```

n array into 2 size-$n/2$ arrays, we set up our recurrence as $T(n) = 2T(n/2) + O(n)$. This is the same recurrence as merge sort, so this algorithm is $O(n \log n)$. Divide and conquer methods can greatly improve on the naive approach. Note that this is not the most efficient solution – there is a dynamic programming solution that can solve this problem in $O(n)$. Practically, divide-and-conquer algorithms are especially useful for very large or cumbersome problems, such as in large dense matrix multiplication (**Strassen's algorithm**), multiplying large numbers, and computing the fast Fourier transform (the **Cooley-Tukey algorithm**).

In general it is possible to find the asymptotic bound from the recurrence relation directly. If $d > \log_b a$, then $T(n) = O(n^d)$. If $d = \log_b a$ then $T(n) = O(n^d \log n)$. And if $d < \log_b a$ then $T(n) = O(n^{\log_b a})$. This simplification is known as the **master theorem**.

Part II

Sorting

A sorting algorithm is one which solves a sorting problem, i.e. it permutes an input sequence such that each successive element is greater than or equal to the previous one. So far we have seen two sorting algorithms – insertion sort, which is $O(n^2)$, but has the benefit of being very fast for small arrays since the inner loop is tightly constrained; and merge sort, which is $O(n \log n)$, but does not sort in-place.

2.1 Heapsort

Heapsort, like merge sort, is an $O(n \log n)$ algorithm. Unlike merge sort, heap sort sorts in-place, meaning we only need a constant amount of memory to operate. Heapsort also uses a more sophisticated data structure – the **heap** – to manage data. A binary heap is an array object that we can think of as a binary tree stored as an array. For a given element at position i, its parent in the heap is element $\lfloor i/2 \rfloor$, its left child is $2i$, and its right child is $2i + 1$. Most computers can make these lookups in one or two instructions using bit shifts, making this format extremely efficient. Heaps are different from binary trees in that the values in the nodes must satisfy the **heap property**, wherein each element of the heap can be no greater than its parent (for a max-heap) or no less than its parent (for a min-heap). Basic heap operations take on the order of $\log n$ time, since they are proportional to how many "parent" or "child" lookups/writes we need to execute. So to sort n items, we must perform $\log n$ work for each item, and we expect an $O(n \log n)$ runtime.

Algorithm 5 Heapsort

```
procedure MAX-HEAPIFY($A, i$)
    $l, r \leftarrow$ LEFT($i$), RIGHT($i$)
    if $l \leq A.size$ & $A[l] > A[i]$ then $largest \leftarrow l$
    else $largest \leftarrow i$
    if $r \leq A.size$ & $A[r] > A[largest]$ then $largest \leftarrow r$
    if $largest \neq i$ then
        SWAP($A, i, largest$)
        MAX-HEAPIFY($A, largest$)
procedure BUILD-HEAP($A$)
    $A.size \leftarrow A.length$
    for $i \leftarrow \lfloor A.length/2 \rfloor$ downto 1 do MAX-HEAPIFY($A, i$)
procedure HEAPSORT($A$)
    BUILD-HEAP(A)
    for $i \leftarrow A.length$ downto 2 do
        SWAP($A, 1, i$)
        $A.size \leftarrow A.size - 1$
        MAX-HEAPIFY($A, 1$)
```

This algorithm has three main procedures, along with the left and right lookups, and a simple swap operation that swaps two elements in the array. The first procedure correctly bubbles element i down into the heap by ensuring it is larger than both of its children, and swapping it with its children recursively if it is not.

The second procedure builds a heap by finding all non-leaf nodes (note that all nodes after $\lfloor A.length/2 \rfloor$ must be leaf nodes, and therefore do not need to be bubbled down, due to the structure of the heap). This is $O(n \log n)$, but in reality, we do not perform $\log n$ work for every single node; the actual tight bound is $O(n)$, when taking into account that the heap only has half of its nodes at the maximum depth. A simple inductive argument shows that the heap building procedure is consistent. At the beginning, all leaf nodes are in the "second half" of the array, and so are already heapified. At step i, assume all elements to the right of i satisfy the heap property. But bubbling down i into the rest of the heap, we maintain that at iteration $i - 1$ all elements to the left of $i - 1$ are heapified. The algorithm terminates when all indices are correctly in the heap.

The final step is heapsort, where we take the top element of the heap and swap it with the last. Then by decrementing the size of the heap, we never touch the last element again. The last step is to appropriately bubble down the swapped element. The result is a fully sorted array, with the largest element last and the smallest first. The total runtime is $O(n)$ to build the heap, and $O(\log n)$ times for each call to the max-heapify procedure, totaling $O(n \log n)$.

The heap itself is a useful data structure to create a **priority queue**, a system for being able to retrieve the largest or smallest of a collection of objects. A priority queue usually has an insertion procedure, which is similar to the max-heapify procedure above, and a "pop" procedure, where the first element is removed and the heap is shrunk, like in the core heapsort loop. Priority queues additionally support the Decrease-Key$(Q, key, value)$ procedure, whereby a key in the queue has its value decreased and correspondingly bubbled around the heap.

2.2 Quicksort

Quicksort is a remarkably efficient algorithm, though more complex then the algorithms we have seen so far. Its worst case runtime is $O(n^2)$, but in the average case it is $O(n \log n)$, and the constant scaling factors absorbed by the $O(n \log n)$ are very small (like insertion sort). It is a divide-and-conquer algorithm, with two main steps; it selects a **pivot** element and makes swaps until all the elements to the left of the pivot are less than the pivot, and all elements to the right are greater. It then recursively calls quicksort on the left and right halves. Because the swaps in the first half fully segregate the two halves, no additional work is required to combine them.

The first call is made to Quicksort$(A, 1, A.length)$. The key of this procedure is the partition algorithm, which, for a subarray ranging from index p to r, starts by selecting the last element of the subarray as a partition. It then travels from left to right along the subarray, and maintains a pointer i marking the point where the next element is greater than $A[r]$. If it finds an element that is smaller than $A[r]$, it will swap the element with the element at i, and increment i. When it is done, it places $A[r]$ at position i. The result

Algorithm 6 Quicksort

```
procedure Partition($A, p, r$)
    $x, i \leftarrow A[r], p - 1$
    for $j \leftarrow p$ to $r - 1$ do
        if $A[j] \leq x$ then
            $i \leftarrow i + 1$
        Swap($A, i, j$)
    Swap($A, i + 1, r$)
    return $i + 1$
procedure Quicksort($A, p, r$)
    if $p < r$ then
        $q \leftarrow$ Partition($A, p, r$)
        Quicksort($A, p, q - 1$)
        Quicksort($A, q + 1, r$)
```

is that neither the left or right halves are sorted, but the left half is smaller than the pivot, and the left half is larger. This algorithm also admits a simple inductive proof, where we show that on each iteration of the partition function, the elements before i are smaller than the pivot and the elements after i are greater.

Quicksort suffers its worst case runtime if each partition yields a subproblem with $n - 1$ elements, in which case the runtime reaches $O(n^2)$. But in the average case, assuming a random starting state, we typically reduce the problem by $n/2$ each time, meaning we approach $O(n \log n)$ for similar reasons as with merge sort. As a result, quicksort implementations usually randomly select the pivot point, and we then expect the input array to split close to evenly on average. To make the system even more robust to a hacker that might try to anticipate the pseudorandom number generator used to generate the pivot, you can try to implement a median-based quicksort, where the first, middle, and last elements are sorted and their median becomes the pivot point, pushing us closer to the average case performance.

2.3 Linear Time Sorting

So far we have seen **comparison sorts** which directly compare elements to order a sequence. In this way we can think of comparison trees as forming a **decision tree** of binary greater-than comparisons. The maximum depth of a binary tree is $O(\log n)$ (or $O(n)$ in the worst case, with a spindly tree), and we must examine each element at least once, so the lower bound for the worst case comparison sort is $\Omega(n \log n)$ – heap sort and merge sort are asymptotically optimal sorts, while quick sort isn't because of it's $O(n^2)$ worst case with bad pivot selection.

Counting sort assumes that each input element is an integer between 0 and k, and runs in $O(n + k)$. Counting sort works by determining the number of elements m less than or equal to element i, and then puts element i at index m in the array.

Algorithm 7 Counting Sort

```
$B, C \leftarrow Array[1..n], Array[0..k]$
**for** $i \leftarrow 0$ **to** $k$ **do** $C[i] \leftarrow 0$
**for** $j \leftarrow 1$ **to** $n$ **do** $C[A[j]] \leftarrow C[A[j]] + 1$
**for** $i \leftarrow 1$ **to** $k$ **do** $C[i] \leftarrow C[i] + C[i-1]$
**for** $j \leftarrow n$ **downto** $1$ **do** $B[C[A[j]]], C[A[j]] \leftarrow A[j], C[A[j]] - 1$
**return** $B$
```

The first for loop initializes C to be an array of zeros. The second for loop passes over the array and counts the frequency of each element, and stores it in C. The third for loop counts the number of elements less than or equal to i by maintaining a running total of the elements in C. The final for loop inserts each element into its correct position inside of the output array B, and handles duplicates by decrementing $C[A[j]]$ along the way. We iterate through this array backwards, effectively emptying our count that we accumulated in the third loop. Counting sort is a type of **distribution sort** where we make no comparisons of the actual data, but rather rely on some knowledge of the distribution of data to make the data assume its sorted form. It is a **stable sort**, meaning identical elements in the input array will maintain their position relative to each other in the sorted array.

Counting sort is a subroutine of a more common distribution sort – **radix sort**. In radix sort, we perform a stable sort d times, where each time we sort based on the ith digit of each element, assuming d maximum digits within the array. We begin from the rightmost or least significant digit (LSD). Because we use a stable sort, we ensure that as we rearrange the elements according to increasingly significant digits, we preserve the order from the previous steps. We can also sort from the most significant digit (MSD), but the stability of that sort is not guaranteed. As a result, radix sort can sort n d-digit numbers in $O(nd)$ time if the stable sort it uses is $O(d)$.

Algorithm 8 Radix Sort LSD

```
**procedure** RADIX SORT LSD($A, n, d$)
    **for** $i \leftarrow 1$ **to** $d$ **do**
        $StableSort(A[1...n])$ on digit $i$
```

2.4 Medians

An analogous class of problems to sorting is the **selection** problem, wherein we need to extract an element from a collection. A typical example is that of an **order statistic**, i.e. the ith smallest element of the collection. The median of a set of size n is the $n/2$th order statistic of that set. One solution is to sort the collection in $O(n \log n)$ time, but we can determine asymptotically faster methods.

In fact, we can use a divide-and-conquer method similar to quicksort that finds the ith smallest element, only in $O(n)$ time. This is because of instead of recursing over both sides of the partition, we only recurse over one side.

Algorithm 9 Randomized Select

```
procedure RANDOMIZED-SELECT($A, p, r, i$)
    if $p == r$ then return $A[p]$
    $q \leftarrow$ PARTITION($A, p, r$)
    $k \leftarrow q - p + 1$
    if $i == k$ then return $A[q]$ // Return the pivot value
    else if $i < k$ then return RANDOMIZED-SELECT($A, p, q - 1, i$)
    elsereturn  RANDOMIZED-SELECT($A, q + 1, r, i - k$)
```

The algorithm is itself straight forward. We randomly pivot around elements q selected on each iteration. If q has more than i elements smaller than it, we need to recurse on the left-hand partition, keeping i the same. If q has fewer than i elements smaller than it (let's call that number k), then we need to recurse on the right-hand partition, now trying to find the $i-k$th smallest element of that partition. In doing so we narrow down the ith smallest element, which occurs when $i == k$ or when there are no elements left. The worst-case running time for this algorithm is $O(n^2)$ if we consistently partition around the largest element in the array, but is $O(n)$ in the average case. This is because the partition function itself is $O(n)$. So on the first iteration we are doing $O(n)$ work, on the second $O(n/2)$, on the third $O(n/4)$, and so on, which is a convergent series that is still $O(n)$. We can optimize this procedure by more intelligently selecting the pivot so that it is always provably optimal.

Part III

Data Structures

Algorithms often involve manipulating objects known as **sets**, which are collections of data that can grow, shrink, or be manipulated. Set operations can typically be separated into **queries** which access information, and **modifiers**, which manipulate the set. Queries consist of things like finding the minimum, the maximum, searching for a key in a key-value set, etc., while modifiers include insertion and deletion.

The **array** is one of the most elementary data structures, represented as a contiguous sequence of bytes in memory. Most programming languages the elements of an array to be the same size; in the event that an array has objects of varying sizes, the array itself will usually only store a **pointer** to that object.

A **matrix** is a two-dimensional array of dimension $m \times n$ where m is the number of rows and n is the number of columns. Matrices can be stored in a variety of ways. They can be stored in **row-major order**, in which case the rows are stored contiguously, or **column-major order**, in which case the columns are stored contiguously. The contiguous blocks can either be stored one after the other in a single $m \times n$-element array, or they can be referenced with a pointer (so in row-major order, we would have an array of size m where each element is a pointer to an array of size n, and vice versa). Most modern matrix implementations use the single-array representation.

A **stack** is a set where the element deleted is always the one most recently inserted (last-in, first-out or **LIFO**). A **queue** is a set where the element deleted is always the one which was inserted earliest (first-in, first-out or **FIFO**). The insertion operation for a stack is often called **push** and the deletion operation is called **pop**. For a queue we typically say **enqueue** and **dequeue**. A stack can be implemented trivially with an array, where a pointer keeps track of the top of the stack. A queue can be implemented similarly, except with two pointers that track the insertion point and the deletion point, wrapping around to the front of the array when necessary.

A **linked list** is a data structure whose elements are arranged in a linear order. Instead of determining this order through indices, each element in the list contains a pointer to the next elements in the list. This idea can be expanded to a **doubly-linked list**, wherein each element points not only to the next element but to the previous one as well. The first element of the list is the **head**, and the final element is the **tail**; $head.prev$ points to a null value, as does $tail.next$.

Linked lists are extremely powerful data structures, and support all major set operations. They can be manipulated simply by moving pointers around; for example, to insert an element at the kth position, iterate through the list by following $next$ pointers until the kth position; create a new node in the linked list, and then adjust the $k-1$st pointer and the new node's pointer to keep the list contiguous. Operations like deletion follow a similar idea.

Not all elements can be represented through linear relationships, however. **Trees** have a rooted element that might point to multiple different elements. A **binary tree** is a tree where each node maintains three pointers – one up to the parent node, and two to a left and right child node. The heap implementation we mentioned when implementing heapsort is a type of binary tree.

❈ Hash Tables

Hash tables are incredibly efficient structures when the only necessary operations are insertion, lookup, and deletion. Search in a hash table is $O(1)$ in the average case. Naive array access is $O(1)$ since we can compute the memory address from the index directly; a hash table generalizes that notion – instead of making the key the array index, we derive the index from the key.

The $O(1)$ array lookup time is due to a technique called **direct addressing** which works when the universe of keys is small. In direct addressing, we initialize a table with enough slots for m addresses, given a universe of size u. Then any $k \leq u$ data can be written by occupying or updating a slot's value. The index of the object is then just the key in the direct address table.

However, this approach fails for very large universes, where storing the table is impossible. Commonly, the actual space of keys we need to store is much smaller than the total universe of keys. In a direct address table, key k is written to slot k, but in a hash table, key k is written to slot $h(k)$ for a **hash function** h. This makes it so that the billionth key in the "universe" could still be made the 10th element in a small array with a low chance of collision. A hash table might write multiple values to the same slot, known as a **collision** — but typically the hash function is as good as random, and we have ways of dealing with collisions.

The ideal hash function is one that deterministically outputs a value $h(k)$ between 0 and m, where $m \ll u$, where any two distinct keys have a $1/m$ chance of collision. This is an **independent uniform hash function** or a **random oracle**. This is a theoretical concept, and is not realistically implementable. One way to resolve collisions is by **chaining** – instead of each element in the table of size m containing a pointer to a single value, it instead has a pointer to the head of a doubly linked list, which has all the data corresponding to that hash "bucket." Then a search simply corresponds to finding the correct bucket and doing a linked-list traversal. The size of these chains is called a **load factor** α. With a good hash function, these traversals are asymptotically insignificant with respect to the total amount of data – it is $O(1 + \alpha)$, which we **amortize** to $O(1)$.

Good practical hash functions approximate the independent uniform hashing property. They should derive the hash value in a way that is independent of patterns in the data. Hash functions are typically designed to handle keys that are either integers or short integer vectors, which can then generalizeto more complex object types. Static hash functions do not have any kind of randomness – examples include the **division** hash, $h(k) = k \mod m$ (for some secret prime m) and **multiplication** hash $h(k) = \lfloor m(kAmod1) \rfloor$. The division hash takes the remainder of the key when divided by a prime (which is quite restrictive, since it means our hash table must be prime in size) and the multiplication hash takes the remainder when k is multiplied by a value $0 < A < 1$

and uses that remainder to bin the key (by multiplying by m). Neither of these hashes guarantees good average-case performance.

Adversaries can choose keys such that they are always hashed to the same value, which would lead to a search time of $O(n)$. This vulnerability exists with all static hash functions. Random hash functions do not rely on a single hash function – they instead select a hash function from a **family** at random during program initialization. Such a family $\mathcal{H}$ is **uniform** is, for any key k, the probability that it is hashed to a slot q is $1/m$. The family is ε**-universal** if $P(h(k_1) = h(k_2)) \leq \varepsilon$. And $\mathcal{H}$ is d**-independent** if for any set of distinct keys k_d and any set of slots q_d, the probability that $h(k_i) = q_i$ is $1/m^d$.

One simple number-theoretic example of a universal family is the family described by $h_{a,b}(k) = ((ak + b) \mod p) \mod m$ where p is a prime that is larger than any possible k, and m is the size of the hash table. a and b are integers selected at runtime where $a \in [1, p)$ and $b \in [0, p)$. The proof is omitted, but it can be shown that for distinct k_1 and k_2, $(ak + b) \mod p$ is distinct as well, meaning $h_{a,b}(k_1)$ and $h_{a,b}(k_2)$ has collision probability $1/m$. For sequences of values, we can use cryptographic functions. Most chip manufactureres provide instructions within their architectures for fast cryptographic function implementation. Such functions provide a fixed-length output for an arbitrarily sized input.For example we can have a hash function such as $h(k) = \text{SHA-256}(k) \mod m$ (or salt the input by prepending a string a to k before hashing).

Open address hash tables deal with collisions differently than our previous linked-list chaining idea. In an open address hash table, the values are stored directly in the table instead of as pointers. Each key has a "preference list" (or **probe sequence**) of slots it would like to be assigned to, independently of other keys. Insertion amounts to going through the key's probe sequence until an open slot is found; searching amounts to going through the key's probe sequence until the proper slot is discovered.

※ Binary Search Trees

Binary search trees support all the previously mentioned set operations, meaning they function as both a dictionary and a priority queue. These operations take time proportional to the height of the tree – in the worst case (a **spindly**, one-sided tree) $O(n)$ and in the best case (a **bushy** tree) $O(\log n)$.

Entries in a BST always satisfy a key property, which is that all the nodes in the left child subtree of a parent node must be less than or equal to the parent, and all nodes in the right child subtree must be greater than or equal to the parent. This has the nice property of being able to retrieve a sorted set of entries by traversing the left subtree, retrieving the parent, and then traversing the right subtree (an **inorder traversal**). In the case where we emit x first before traversing the children is a **preorder traversal**, and when we emit x last, a **postorder** traversal.

Algorithm 10 Inorder-Traversal

```
procedure INORDER-TRAVERSAL($x$)
    if $x \neq nil$ then
        INORDER-TRAVERSAL($x.left$)
        EMIT($x.value$)
        INORDER-TRAVERSAL($x.right$)
```

Because BSTs inherently capture the ordering of their constituents, search can be done in $\log n$ time through recursively selecting and traversing through left-or-right branches.

Algorithm 11 BST-Search

```
procedure BST-SEARCH($x, k$)
    if $x == nil$ or $k == x.key$ then return $x$
    if $k < x.key$ then return BST-SEARCH($x.left, k$)
    else return BST-SEARCH($x.right, k$)
```

More precisely, this algorithm runs in $O(h)$ for a tree of height h, but we typically try to guarantee that the tree is as bush as possible, i.e. $h = O(\log n)$. Likewise, finding the minimum (and maximum) of a BST is easy, simply by traversing through all the left-children (right-children) of the tree until a leaf node is reached.

Inserting into a BST amounts to a search traversal, except this time the search is guaranteed to be nil – so we insert the element at the nil location where we'd expect to find the value. Deletion is more complex, since it forces a reorganization of all subtrees of the deleted node. In the event that the deleted node only has one child or no children, the procedure is trivial – we just delete the node and optionally promote the singular child subtree. In the event the node has both children, we (1) replace the value of the deleted node with the value of that node's *successor* (the minimum value of the right subtree), and then (2) assign the now-hanging right-child of the successor (it cannot have a left child) to the successor's parent. Take a moment to reason about why this procedure must work. We call this procedure **transplant**ing.

4.1 Red-Black Trees

Red-black trees (RB) are BSTs that have one extra bit of storage – a *color*, either red or black. Red-black are always approximately **balanced** by maintaining a strict set of conditions:

1. Every node is either red or black
2. The root node and all nil nodes are black
3. If a node is red, its children must be black

4. All paths from a node to any of its nil nodes must contain the same number of black nodes. This is the **black-height** of that node (does not include the first node itself).

A proof by induction shows that this must be bushy. A tree is bushy if each node x has at least $2^{bh(x)} - 1$ internal nodes (the total number of nodes of its children). This is trivially true if node x has no children. Now suppose that a child of node x has nonzero height, and is not the root node. If x has a red child, that child has the same black-height as x, so $bh(x)$. If it has a black child, then that child has a black-height of $bh(x) - 1$. Since by the inductive hypothesis, the child of x has at least $2^{bh(x)-1} - 1$ internal nodes, then x must have at least $2 * 2^{bh(x)-1} - 1) + 1 = 2^{bh(x)} - 1$ internal nodes. A tree with height h must have a black-height of at least $h/2$ (since any red node must have black children), meaning that the number of nodes $n \geq 2^{h/2} - 1$. Rearranging, this indicates that $h \leq 2\log(n + 1)$.

Insertion and deletion are nontrivial, since the BST implementation will not preserver the red-black property. Insertion and deletion require **rotation**, the idea that if x is a left-child of y, then the tree can be re-arranged so that y is a right-child of x, by making it so the right-child of x becomes the left-child of y. This is reminiscent of the BST deletion algorithm, and is called a **right rotation**. A left rotation is defined similarly. Since only three pointers are modified – the pointer from the parent of y now points to x, $x.right$ now points to y, and $y.left$ now points to $x.right$ – this rotation happens in $O(1)$ time.

Algorithm 12 RBTree-Rotate-Right

```
procedure RBTree-Rotate-Right(T, y)
    $x \leftarrow y.left$
    $y.left, x.right.parent \leftarrow x.right, y$
    $x.parent \leftarrow y.parent$
    $y.parent.[left/right] \leftarrow x$ // Whichever child $y$ was
    $x.right, y.parent \leftarrow y, x$
```

RB-tree insertion starts similarly to normal BST insertion, where the inserted node is colored red. Then a series of rotations takes place until the RB-property is not violated. At a high level, properties (1) and (4) cannot be violated by this insertion, so the only constraints we worry about are (2) and (3).

The procedure for fixing these issues has some symmetry. If the parent $z.p$ of the new node z is a left-child, we are symmetric to the case where $z.p$ is a right-child. We only consider one of these cases for brevity.

1. Check if $z.p$ is red. If it is not, then set the tree's root node to black and end.

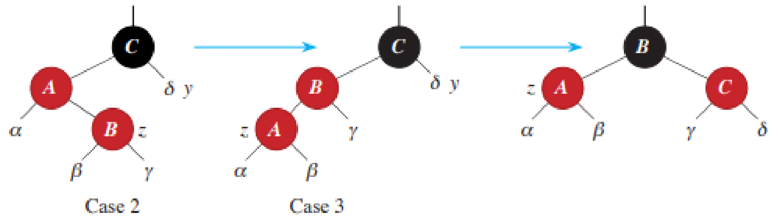
2. Suppose WLOG that $z.p$ is a left-child. Then there are three cases

   (a) The sibling to $z.p$ is red. In this case, we can turn $z.p$ and its sibling both black and turn the grandparent, $z.p.p$, red. This preserves the black-height of the tree.

   (b) $z.p$'s sibling is black and z is a right-child. Performing a left-rotation on $z.p$ then turns this into the next case (c):

   (c) $z.p$'s sibling is black and z is a left-child. In this case, a right-rotation on $z.p.p$ would make z and $z.p.p$ siblings, and swapping the colors of $z.p$ and $z.p.p$ now means that all colors are organized.

3. Set $z = z.p.p$ and continue at (1) until termination.



Note that in this process it is impossible for there to be more than 1 "error" in the tree. Furthermore, the only errors are (a) the root node is the wrong color or (b) there exist two consecutive red nodes. The process of either flipping the colors of the nodes or (b) performing two rotations so that we can recolor the nodes correctly never increases or decreases the black-height of the tree, and therefore can never increase the number of double-red violations. This is an $O(\log n)$ operation since insertion is $O(\log n)$ and the fixing process is $O(\log_4 n)$ (each iteration moves up 2 levels in the tree).

Deletion is more complicated. It likewise begins just like deletion in a BST. We use the same "transplant" idea as before, except we recolor the successor node to be the same color as the deleted node. This will violate the red-black property if the successor node is black. If that node is red, then there can be no problems because the black-height will not change, and the recoloring prevents two adjacent reds from appearing. The red-black deletion algorithm performs a series of conditional rotations and recolorings to correct these errors. For brevity, the details of all the if-statement logic is omitted here, but the core mechanism is what's important. Each of the conditions where we can have an "extra black" can also be permuted into each other, making the core algorithm simpler.

Part IV

Advanced Techniques

❈ Dynamic Programming

Dynamic programming is similar to divide-and-conquer, in that it aims to solve a large problem by dividing it into subproblems. However, while divide-and-conquer algorithms split up problems into a neat partition, dynamic programming techniques deal with the case where the subproblems overlap. Dynamic programming algorithms define the character of an optimal solution, define that solution recursively, and then compute that value bottom-up.

The rod-cutting problem is as follows: Given a rod of n inches and a price table p_i for all $i \leq n$, determine the maximum revenue obtained r_n by cutting up the rod and selling the pieces (up to $n-1$ cuts). There are 2^{n-1} ways to do this, but a recurrence immediately stands out. If we cut a length of rod k inches long, then the maximum revenue we can get is p_k plus the maximum revenue of cutting a rod of length $n-k$. So then we can design an algorithm that, for $i \leq n$, calculates $\max(currentMax, p[i] + \textsc{Cut-Rod}(p, n-i)$. This is an abysmally inefficient implementation (it is $O(2^n)$, since we are manually trying each of the 2^n configurations of cuts).

A dynamic programming approach might start simply – instead of recomputing $\textsc{Cut-Rod}(p, n-k)$ over and over, **save** the value the first time and re-use it every time we see it again (this is called **memoization**). That way, we only need to save n values in memory and perform lookups, meaning our total runtime goes from $O(2^n)$ to $O(n^2)$! This approach is called **memoization**. This can be done in a top-down way, where we keep the natural recursion and just save our intermediate results once we hit the bottom, and a bottom-up method, where we actually start from the bottom and begin building up the table until we hit the case that we wanted to solve for. So a top-down approach would lay out the same recursion as before, and start caching when it hits the bottom, i.e. when we have rods of length 1. A bottom-up approach starts by finding the best return for a rod of length 1, then uses that to compute the best return for r_2, then r_3, and so on until we get to rod r_n. Bottom-up approaches usually have better constant scaling factors as compared to top-down. The rod cutting algorithm might then look something like this:

A more practical example is the similar **matrix-chain multiplication problem**, where the problem is to determine the parenthesization for multiplying n matrices together, where the multiplication is guaranteed to be valid but the dimensions of each matrix are different, such that the total number of operations is minimized. The ordering in which the matrices are multiplied can have a dramatic impact on the overall performance of

Algorithm 13 Cut-Rod

```
procedure Cut-Rod($p, n$)
    $r \leftarrow Array[0..n]$
    $r[0] \leftarrow 0$
    for $j \leftarrow 1$ to $n$ do
        $q \leftarrow -\infty$
        for $i \leftarrow 1$ to $j$ do
            $q \leftarrow \max q, p[i] + r[j-i]$
        $r[j] = q$
    return $r[n]$
```

the multiplication.

Dynamic programming problems are a good approach when certain properties of the problem are true. The problem should exhibit an **optimal substructure**, i.e. if the optimal solution to the problem contains optimal solutions for subproblems. Take for example finding the shortest simple path in a graph $G(V, E)$ between u and v. If a vertex w is part of the shortest path p between u and v, then the path from u to w in p must also be the shortest path between u and w (likewise for w and v). This is not true for the *longest* path problem, since the longest path between u and w is not independent of the path from w to v.

DP problems should also be overlapping, i.e. the naive recursive algorithm would have to solve some subproblems over and over again as part of the recursion. A divide-and-conquer problem would solve a new problem every time, so DP techniques are not useful, since information is not shared across the recursion. This lets us reconstruct the optimal solution from stored (memoized) optimal solutions to subproblems, and efficiently compute the true solution.

Another common dynamic programming problem is the **longest common subsequence** (LCS) problem. In this problem we aim to find the longest common subsequence (meaning ordered, but not necessarily consecutive) between two collections of ordered items. While an enumeration of all subsequences is $O(2^n)$, infeasible for long sequences, this exhibits the optimal substructure property nicely. This is because any LCS of two strings X_n and Y_m must contain within it the LCS of every prefix of X_n and Y_m. Let Z_k be the LCS. If $x_n = y_m$ then $z_k = x_n = y_m$, meaning Z_{k-1} is the LCS of X_{n-1} and Y_{m-1}. If $x_n \neq y_m$ then Z_k is either the LCS of X_{n-1}, Y_m or X_n, Y_{m-1}. This exhausts all possibilities. The recursion then looks like this:

$$LCS_{i,j} = \begin{cases} 0 & i = 0 | j = 0 \\ 1 + LCS_{i-1,j-1} & x_i = y_j \\ \max LCS_{i,j-1}, LCS_{i-1,j} & \text{otherwise} \end{cases}$$

The resulting algorithm will be $O(mn)$.

※ Greedy Algorithms

Optimization algorithms usually go through a series of steps with a choice at each step. Greedy algorithms always make a locally optimal choice in the hopes that it will coincide with the globally optimal solution. Greedy algorithms are usually simple to implement but highly effective in a wide range of settings.

Suppose the **activity selection problem**. In this problem we are given a collection of n activities $S_{0,n}$ characterized by their start times s_i and end times f_i. The objective is to return the maximum number of activities k that can be performed such that $s_{i+1} > f_i$, i.e. the maximum number of non-overlapping activities. This problem exhibits the optimum substructure; for a given activity a_i, the maximum number of activities must include the maximum number of activities that finished before a_i and also the maximum number of activities that start after a_i, i.e. $Act_{ij} = \max Act_{i,k} + Act_{k,j} + 1 : a_k \in S_{i,j}$ However, we do not need to bookkeep all such activities. In this problem, we can instead pursue a greedy approach. Take the activity which ends *earliest*, ending with f_1. There is no way to exclude this activity from the collection and end up with any *more* activities, since no activity could be scheduled before the one that finishes first. So without loss of generality the activity that finishes earliest is a member of the set, since it also leaves the most time for subsequent activities. This yields in an $O(n)$ algorithm where each activity is only considered once.

In the dynamic programming sense, by selecting the first activity as the one that finishes first, we eliminate one side of the recursion and only recurse over future events. Greedy algorithms typically have this DP underpinning – they exhibit some optimal substructure, with the additional restriction that by making a choice only a single subproblem remains (instead of multiple). The choice made must be locally optimal, i.e. an optimal solution would have made the same choice. And the final step is to show that a globally optimal solution can be assembled from locally optimal choices. Greedy algorithms differ from dynamic programming algorithms in that while DP problems try to construct the full set of subproblem solutions before proposing a choice, greedy algorithms make the first choice with no knowledge of what future problems may yield.

Consider two problems as an example. In the **0-1 knapsack** problem, a thief needs to steal as many items as possible from a collection of size n where each item has a value v_i and a weight w_i, such that the thief's knapsack never exceeds total weight W. In the **fractional knapsack** problem the thief is permitted to take any fraction of an item instead of needing to take the whole thing. The fractional problem admits a greedy approach, where the thief can rank the items by "value per unit weight", and start filling the knapsack from most to least value-per-weight, taking a fraction of the last item to fill up the knapsack. But this solution will not work for the 0-1 problem; suppose $W = 50$ and item 1 has value 60 and weight 10, item 2 with value 100 and weight 20, and item 3 with value 110 and weight 30. The greedy approach would give us 160 while a dynamic programming approach would give us 220.

6.1 Huffman Codes

Encoding a file of n characters with a vocabulary of size k will take $O(n \log k)$ characters if each character is given an equal $\log k$-bit representation. We can do considerably better with variable-length codewords, wherein more frequent characters get shorter encodings. In particular we seek an encoding where no character's code is a prefix for another character's code (so that the decoding is unambiguous). As such the encoding can be represented by a tree where the leaf nodes are characters and the from the root to leaf, when demarcated by 1s and 0s, is the encoding. The algorithm to build this tree is as follows, assuming a frequency table C

Algorithm 14 Huffman

```
procedure HUFFMAN(C)
    $n, Q \leftarrow |C|, C$
    for $i = 1$ to $n - 1$ do
        $z \leftarrow Node.new$
        $x, y$ = EXTRACT-MIN($Q$), EXTRACT-MIN($Q$)
        $z.left, z.right, z.freq \leftarrow x, y, x.freq + y.freq$
        INSERT($Q, z$)
    return EXTRACT-MIN($Q$)
```

This is a tree-building algorithm, starting from C which is a flat list of leaf nodes. At each iteration, the algorithm removes the two least frequent leaves and gives them a shared parent z which it then inserts the parents of those leaves. Then an assignment of 0 and 1 for left and right will yield the appropriate prefix-tree codes (they must be prefix-free because all the characters are in leaf nodes and there is a unique path to each leaf node). The greedy algorithm works well here because this problem has the greedy-choice property – the least frequent characters should always receive the highest number of bits, so they should be placed lowest in the tree.

Part V

Advanced Data Structures

Sometimes it is necessary to augment data structures to solve more intricate problems. A red-black tree can be augmented with more information to solve the ith order statistic problem (the ith smallest element) by having each node store the size of the trees below it. That information is sufficient to calculate the ith order statistic in $\log n$ time. The tricky part is that this information must be updated alongside the actual node geneology when a rotation, insertion, or deletion occurs.

Augmenting a data structure usually begins with deciding which data structure to select; as discussed above, each data structure is specially suited to solve a class of problems. Then, after deciding which data structure to use and what data to store at each location of that structure, all that remains is to determine how to augment that data structure's operations to maintain the validity of its carried data.

❈ B-Trees

B-trees are balanced search trees designed for databases and disk drives. Unlike BSTs or RB-trees, each node in a B-tree can have thousands of children (its **branching factor**). A node in a B-tree consists of a series of k keys in sorted order. Its children are nodes which each handle the $k+1$ *intervals* of keys specified by the parent. So if, for 100 items, a node had range 70 to 90 and its keys are 75, 82, 86 then it will have four children – one from 70-74, one 76-81, one 83-85, and one 87-90.

Data structures on disk drives are different from data structures used in RAM. Disk drives are mechanical objects, and accessing data amounts to waiting for the disk drive's platter to spin the correct amount so that the read/write head finds the appropriate data – over 100,000 times slower than accessing data in RAM. To amortize this cost disk drives usually do not access one item, but rather **blocks** of bits – usually between 512 and 4096 bytes. B-tree algorithms will typically read entire blocks into main memory for processing, and the range of values managed by a single node is usually limited to the size of a block. If a node gets too full, it will "split" into two nodes, where the median value of that node is inserted into its parent recursively. Deletion will work in the opposite way, where if a node becomes underful it will get absorbed into the nodes below it if possible. For a comprehensive understanding of B-trees see Martin Kleppmann's work in *Designing Data Intensive Applications*.

Part VI

Graph Algorithms

❊ Elementary Graph Algorithms

A **graph** is a mathematical construction consisting of objects (called **nodes** or **vertices**) that exhibit a pairwise relationship (an **edge**). A graph is usually denoted as $G(E, V)$. Graphs are typically represented as either adjacency lists (a linked list for each vertex corresponding to the vertices directly connected to that vertex) or an adjacency matrix (a binary matrix where a 1 denotes an edge existing between vertices i and j). Graphs can alternatively be **weighted**, where each edge is assigned some value w_i. Graphs can also be **directed**, meaning an edge can exist from u to v but may not exist from v to u.

The most common graph problem is one of search, i.e. looking up a vertex in the graph (to retrieve its associated data). One of the simplest algorithms for finding a vertex in a graph is through **breadth-first search** (BFS). BFS tries to find a path from a **source** vertex to a target vertex by building a breadth-first tree. The path described by this tree between the source vertex and any child node in the tree is the path with the *smallest number of edges*, which can be seen through a straightforward proof by contradiction. BFS starts at a target node and adds all that node's children into a **queue**. It then pops then pops a child from the front of the queue, enqueues that child's unvisited children, and marks the child as visited. Each child's depth and its parent is recorded on that node at the moment it is enqueued.

Algorithm 15 BFS

```
procedure BFS($G, s$)
    $Q \leftarrow \emptyset$
    for $u \in G.V - s$ do
        $u.status, u.depth, u.parent \leftarrow nil, \infty, nil$
    Enqueue($Q, s$)
    $s.status, s.depth, s.parent \leftarrow queued, \infty, nil$
    while $Q \neq \emptyset$ do
        $u \leftarrow$ Dequeue($Q$)
        for $v \in G.V[u].E$ do
            if $v.status == nil$ then
                $v.status, v.depth, v.parent \leftarrow queued, u.depth + 1, u$
                Enqueue($Q, v$)
        $u.status = visited$
```

Depth first search is an alternate graph search algorithm where an entire adjacency list is explored before the parent of that list is marked as visited. The algorithm goes from child to child until it can go no further, and then traces backwards up the stack of children. In doing so DFS also forms a tree (or several trees). This is somewhat arbitrary – BFS can also be run from several sources. However, the manner in which BFS is most commonly used – finding the shortest path between two vertices – reflects the single-tree nature.

Algorithm 16 DFS

```
procedure DFS($G$)
    for $u \in G.V$ do
        $u.status, u.parent \leftarrow nil, nil$
    for $u \in G.V$ do
        if $u.status == nil$ then VISIT($G, u$)
procedure VISIT($G, u$)
    $u.status \leftarrow visited$
    for $v \in G.Adj[u]$ do
        if $v.status == nil$ then
            $v.parent \leftarrow u$
            VISIT($G, v$)
```

DFS uniquely captures some of the structure of the graph. If we were to record the "discovery time" and the "finish time" of each vertex, i.e. the point at which the vertex is first visited to the time that the algorithm recurses to that vertex's parent, we would see a **parenthesis** structure – each vertex is visited before all of its children, and it is departed after all of its children. So if v and u are vertices, and $v.a$ and $v.d$ are the arrival and departure times at v, then the intervals $[v.a, v.d]$ and $[u.a, u.d]$ are either *entirely disjoint* (in which case v and u are not in the same lineage in the DFS tree) or one interval is a strict subset of the other.

8.1 Toplogical Sort

DFS can be used to perform a **topological sort** of a **directed acyclic graph** (DAG). A topological sort is an ordering such that for any edge (u, v) in the graph, u appears before v in the sort. The algorithm is straightforward – perform DFS on the DAG, and sort the vertices in the reverse order of their departure times. There is one caveat though – the graph may have no **back edges**. This means that during VISIT(G, v) we can never fail the query of $v.status == nil$. If we do, that means we are considering an edge that connects to an already-visited node, which implies that the graph has a cycle.

8.2 Strongly Connected Components

A classic application of DFS is to decompose a directed graph into strongly connected components. Many graph algorithms rely on such a decomposition, and then proceed to iterate on the SCCs and combine the results. A subgraph G^{SCC} of a graph G is a strongly connected component is there is a path between any two vertices of V^{SCC} utilizing edges only in E^{SCC}. This definition means that there can be no cycle between two SCCs (as doing so would make them a single connected component). Hence the **contraction** of all the vertices of an SCC into a single meta-node yields a DAG in any directed graph.

Algorithm 17 Kosaraju's Algorithm

```
procedure KOSARAJU($G$)
    Get finish times for all vertices from DFS($G$)
    Call DFS($G$) in the reverse order of the finish times from the previous step
    Each DFS tree from the previous step is an SCC
```

The algorithm for finding SCCs relies on the fact that, by definition, the SCCs of G are exactly the SCCs of $G^{\top}$, i.e. G with the direction of its edges flipped. The algorithm calls $DFS(G)$ to find departure times $u.d$ for all $u \in V$. Then it runs $DFS(G^{\top})$, where the vertices are considered in the order of decreasing $u.d$. The resulting DFS trees are exactly the SCCs of G.

The first DFS pass reveals information about graph connectivity. The vertex with the highest departure time is guaranteed to be in a "source" SCC (i.e. no other SCC will feed into it). In the reverse graph, this means all vertices reachable from that first vertex will be nodes either in the same SCC or in a parent SCC. Since the first vertex can have no parent (due to having the highest departure time), all nodes reachable from the first node will be in an SCC. A proof by induction using this reasoning proves correctness.

※ Minimum Spanning Trees

A common graph problem is to find the subgraph which connects all vertices V with minimum total edge weight. This subgraph must be a tree, and is called the **minimum spanning tree** of the graph. There are two popular algorithms for finding the MST(s) of a graph, both of which are greedy. Critical to the construction of MSTs is the idea of a **cut**, a partition of V. A cut respects a set of edges if all those edges bridge vertices on one side of the cut. A **light edge** is an edge of minimal weight that crosses the cut. The core observation here is that, given a partially built MST A, and a cut of G that respects A, any light edge for that cut will be in the MST. This is trivally true by minimality – if such a light edge were not part of the cut, a smaller weight edge would be needed to reach that section of the graph, yet that is impossible – so that light edge must be part of the cut.

Kruskal's algorithm grows a minimum spanning tree by finding, out of all edges that connect any two disconnected subtrees, the edge of minimum weight and adding it to the tree. This works because for any minimal subtree C the edge of least weight between C and $G \setminus C$ must be part of the MST of G.

Algorithm 18 MST-Kruskal

```
procedure MST-KRUSTKAL($G, w$)
    $A \leftarrow \emptyset$
    for $v \in G.V$ do MAKE-SET($v$)
    Sort $G.E$ by weight
    for $(u, v) \in G.E$ do
        if FIND-SET($u$) $\neq$ FIND-SET($v$) then
            $A \leftarrow A \cup (u, v)$
            UNION($u, v$)
```

Prim's algorithm is similar, and also uses a greedy algorithm. It differs from Kruskal's algorithm in that there is always only 1 minimum spanning subtree within the graph. Instead of considering all edges in the graph, it only considers those at each time step which are adjacent to the existing subtree. This can be done using a heap, and appending vertices to the heap at the time they are first "reachable" by the growing tree.

In general, both algorithms perform similarly in $O(E \log V)$. Prim's algorithm is substantially better for dense graphs, since using an efficient heap structure like a Fibonacci heap gives us $O(E + V \log V)$. Kruskal's is preferred for sparse graphs since it lets us use simpler data structures.

※ Shortest Paths

Graph algorithms are often used to solve **shortest-paths problems**, in which we aim to find the path of least total cost between two nodes in a weighted directed graph. These algorithms rely on the property that the shortest path between to vertices contains within it other shortest paths as well between pairs of intermediate vertices, making this problem a good candidate for greedy and DP approaches.

10.1 Bellman-Ford

The **Bellman-Ford** algorithm solves the shortest path problem in cases where the graph may have negative weights. This can present a problem for many algorithms since falling into a cycle with a negative edge can yield a path with $-\infty$ weight. The Bellman-Ford algorithm detects and aborts if a negative weight cycle exists; else it yields the shortest paths and the associated weights.

To motivate this algorithm we introduce two more procedures. We begin by initializing each vertex in the graph to have a value $v.d = \infty$ which captures the shortest depth from a source s to v. We also bookkeep vertex parents along this shortest path. Additionally we introduce the concept of **relaxing** an edge, where for two vertices u and v we check if we can decrease $v.d$ by going through u. Namely, if $v.d > u.d + w(u, v)$ then $v.d = u.d + w(u, v)$.

Algorithm 19 Bellman-Ford

```
procedure BELLMAN-FORD($G, w, s$)
    INITIALIZE-SOURCE($G, s$)
    for $i \leftarrow 1 \rightarrow |G.V| - 1$ do
        for $(u, v) \in G.E$ do RELAX($u, v, w$)
    for $(u, v) \in G.E$ do
        if $v.d > u.d + w(u, v)$ then
            return False
    return True
```

The algorithm is simple. The algorithm makes $|V|-1$ passes over the graph and attempts to relax the edges of graph. If after $|V| - 1$ passes any vertices can still be relaxed, there must be a negative weight cycle, so the algorithm terminates. This algoritm takes $O(V^2 + VE)$ time. If the algorithm returns True, then the value $v.d$ is the length of the shortest path from s to v.

The second statement must be true, since if the algorithm returns True then there is no relaxation that could produce a shorter path for any vertex in the graph. The intuition behind the $|V| - 1$ iterations is as follows. After one relaxation, we may guarantee that all shortest paths of length 1 from s are fully discovered. After the second relaxation, all shortest paths of length 2 are correctly calculated. And for a graph of $|V|$ nodes, the maximum length of a path free of cycles is $|V| - 1$; therefore if after $|V| - 1$ iterations a path can still be shortened, the only reason is that there is a cycle of negative total weight somewhere in the graph.

10.2 Dijkstra's Algorithm

Dijkstra's algorithm solves the shortest path problem on weighted directed grpahs with strictly nonnegative edge weights. Dijkstra's generalizes BFS to weighted graphs. Dijkstra's performs vertex selection similar to BFS, except it uses the weighting to determine how to select items from the queue. Think of the weights as time and the vertex selection as a wave. Each time a vertex is visited a wave propagates along its children. When the wave hits a vertex a new wave is immediately sent from that vertex onwards. The time when a wave hits a vertex depends on the weight of its incident edge.

Dijkstra's algorithm is a greedy strategy – it says that at each iteration, choose the closest

Algorithm 20 Dijkstra

```
procedure DIJKSTRA($G, w, s$)
    INITIALIZE-SOURCE($G, s$)
    $S, Q \leftarrow \emptyset, G.V$
    while $Q \neq \emptyset$ do
        $u, S \leftarrow$ EXTRACT-MIN($Q$), $S \cup \{u\}$
        for $v \in G.Adj[u]$ do
            RELAX($u, v, w$)
            if $v.d$ was relaxed then DECREASE-KEY($Q, v, v.d$)
```

vertex to S and add it to S. In this way it is very similar to Prim's MST algorithm from earlier, with the additional relaxation step. Suppose a four-vertex graph. $s \rightarrow u$ has weight 10, $s \rightarrow v$ has weight 3, and $v \rightarrow w$ has weight 8. v is taken first and given a distance of 3. Then $w.d$ is relaxed to $3 + 8 = 11$. Then u gets taken with total path cost 10, and finally w with 11.

Dijkstra's algorithm, like Prim's, depends on the heap implementation used. With a Fibonacci heap the runtime goes down to $O(V \log V + E)$ compared to $O(V^2)$ with a naive heap implementation, mostly coming from the amortized $O(1)$ DECREASE-KEY($Q, v, v.d$) runtime.

10.3 A*

The **A* search algorithm** is a refinement of Dijkstra's algorithm that is specially designed to find the shortest path between two points on a graph instead of the entire shortest paths tree. It does this by considering a **heuristic function** that determines how close the algorithm is to the goal. So while the normal Dijkstra's algorithm can be thought of as using the function $g(n)$ which yields the shortest cost from the source to the current node n. A* adds onto that a **heuristic** $h(n)$ that estimates (optimistically) the remaining cost to the goal. The heuristic must be **admissible**, meaning that it can never be greater than the true cost to the goal. Otherwise the algorithm runs the risk of rejecting the true shortest path. Then we run Dijkstra's as we previously had, with $f(n) = g(n) + h(n)$ as our priority queue weighting, and get an extremely fast path-finding algorithm.

※ All-Pairs Shortest Paths

Instead of finding the shortest path from a single source, a related problem is finding the shortest path between every pair of vertices in a graph. This is useful for, say, query optimization in a large database or traversing through a social graph.

11.1 Floyd-Warshall

The **Floyd-Warshall algorithm** is a DP algorithm to solve the all-pairs problem in $O(V^3)$ time. The algorithm operates on an idea formed by only considering a subset V_k of k of the n total vertices V. For any two vertices i and j, consider the family of paths that travel through exclusively V_k. If the kth vertex is not on the shortest of such paths, then the shortest path from $i \to j$ through V_k is the same as the shortest path from $i \to j$ through V_{k-1}. If k is on the shortest path, then the shortest path from $i \to j$ through V_k is the shortest path from $i \to k$ through V_k. This structure lends itself well to a recursive algorithm. Here W is the matrix of edge weights where entry i, j is the weight of the path from i to j.

Algorithm 21 Floyd-Warshall

```
procedure Floyd-Warshall($W, n$)
    $D^{(0)} \leftarrow W$
    for $k \leftarrow 1 \rightarrow n$ do
        $D^{(k)}$ = Matrix-Zeros($n, n$)
        for $i \leftarrow 1 \rightarrow n$ do
            for $j \leftarrow 1 \rightarrow n$ do
                $d_{ij}^{(k)} \leftarrow \min\{d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\}$
    return $D^{(n)}$
```

Effectively, this algorithm relies on a similar princple to the relaxation argument in the previous section. This algorithm can be done in $O(n^2)$ space, as we do not really need to have different matrices for each iteration of k – rather we can update the matrix in-place, since at each state the weight values are consistently monotonically lower. Note that, for similar reasons to Dijkstra's, the Floyd-Warshall algorithm will fail in graphs with negative cycles.

11.2 Johnson's Algorithm

Johnson's algorithm is an $O(V^2 \log V + VE)$ algorithm which is asymptotically faster than $O(V^3)$ for sparse graphs where $E \ll V^2$. It uses as subroutines both Dijkstra's and Bellman-Ford via a technique called **reweighting**. If all edge weights are nonnegative, running Dijkstra's from each vertex will find all shortest paths, which is $O(V^2 \log V + VE)$ with a fib heap. If G has negative-weight edges but no negative-weight cycles, we reweight the graph so that we can run Dijkstra's, where the new weights strictly preserve shortest paths and are non-negative.

To satisfy these properties, we formulate the reweighting as follows. For any edge $w(u, v)$ we reweight it as $\hat{w}(u, v) = w(u, v) + h(u) - h(v)$ for some function h. This is easy to prove – instead of u, v take some arbitrary path p from v_0 to v_k. p is a shortest path if and only if it is the shortest path by both $w(p)$ and $\hat{w}(p)$. If you sum all the path segments of p under $\hat{w}$, the sum telescopes – meaning the result is $\hat{w}(p) = w(p) + h(v_0) - h(v_k)$. So if p is shortest by w it must also be shortest by $\hat{w}$, since h is independent of p.

To get the reweighted graph, extend G by one vertex $s \notin V$, with out-edges to each vertex $v \in V$ with weight 0. Then let $h(v) = \delta(s, v)$ where δ is the shortest path from s to v. The resulting edges, after updating them with our weight function above, will be nonnegative by the triangle inequality, since $h(v) \leq h(u) + w(u, v)$.

Algorithm 22 Johnson

```
procedure JOHNSON($G, w$)
    Compute $G'$ according to the above description – add $s \notin V$, draw a zero-weight out edge from $s$ to all $v \in V$
    if BELLMAN-FORD($G', w, s$) == FALSE then
        return NIL
    else
        for $v \in G'.V$ do
            $h(v) \leftarrow \delta(s, v)$ from BELLMAN-FORD
        for $(u, v) \in G'.E$ do
            $\hat{w}(u, v) = w(u, v) + h(u) - h(v)$
        for $v \in G.V$ do
            DIJKSTRA($G, \hat{w}, u$) and record the shortest path $d(u, v) = \hat{\delta}(u, v) + h(v) - h(u)$
```

Like Prim's and Kruskal's, the Floyd-Warshall algorithm is better for dense graphs while Johnson's algorithm is asymptotically faster for sparse graphs, and Johnson's algorithm is the only one of the two that can effectively deal with negative edge weights.

❈ Max-Flow/Min-Cut

The maximum-flow problem attempts to compute the greatest rate for moving material from a source node to a sink node in the graph without violating any capacity constraints. A flow network is a directed graph with non-negative edge weights, where all edges are strictly one-directional. Flow networks have a determined source and sink vertex. A **flow** $f : V \times V \rightarrow \mathbb{R}$ satisfies a **capacity** constraint (meaning $f(u, v) \leq c(u, v)$) and is also fully conserved, i.e. the total flow into every vertex equals the total flow out of every vertex (except the source and sink vertices). The maximum flow problem is to find the greatest total flow through the system.

12.1 Ford-Fulkerson

The Ford-Fulkerson method is a meta-algorithm that describes a solution to the max-flow problem. It is not an algorithm *per se* because it can be implemented in several different ways.

For a flow network G the **residual network** G_f is a network of edges whose capacities represent the changes in flow on the edges of G. The edge's residual is $c(u,v) - f(u,v)$ (only edges with nonzero residual capacity are part of the residual network). Unlike the original G, G_f can also contain edges that go from v to u (backward), with capacity $c_f(v,u) = f(u,v)$. This represents the fact that we might want to actually decrease the flow on (u,v) in order to increase flow somewhere else. An **augmentation** to a flow f is then $f(u,v) + f'(u,v) - f'(v,u)$ for a residual flow f' in the residual network.

An augmenting path is a simple path from s to t in G_f, which is typically found with a simple pathfinding algorithm like BFS or DFS. The flow on an edge (u,v) in an augmenting path can increase up to $c_f(u,v)$ (or decrease by up to $c_f(v,u)$) without violating a capacity constraint. The Ford-Fulkerson method iteratively finds augmenting paths and augments the flow of G until no more augmenting paths remain. The resulting flow is the maximum flow of the network.

This algorithm must terminate since the total capacity is strictly monotonically increasing and the network's capacity cannot be infinite. But when it does terminate, how do we know that the result is the maximum flow? A **cut** of a flow network is a partiition of V into S and T such that $s \in S$ and $t \in T$. The **net flow** across a cut is the sum of flows spanning the cut and the capacity of a cut is the sum of capacities spanning the cut. A **minimum cut** is then a cut whose capacity is minimized.

Max-Flow Min-Cut

If f is a flow in a flow network $G(V,E)$ with source s and sink t, then the following are equivalent:

1. f is a maximum flow in G
2. G_f contains no augmenting paths
3. $|f| = c(S,T)$ for a cut (S,T) of G

The max-flow min-cut theorem tells us that the maximum flow from s to t is equal to the total capacity of the minimum cut of the network. This is not a surprising result – all flow must pass through the minimum cut, so its total capacity should provide an upper bound on the total flow, and this bound is tight.

When the Ford-Fulkerson method is implemented using BFS to select paths in order of increasing total capacity, the resulting algorithm is known as the **Edmonds-Karp** algorithm.

❈ Maximum Bipartite Graph Matching

A common real-world problem is finding optimal matchings within a group of objects. A matching is a subset of edges in a graph such that each vertex has at most one incident edge. In this way this can be modeled as a **bipartite graph**, where the graph is partitioned into subsets L and R such that all the edges cross between L and R. One solution to find a matching of maximum cardinality is to model the graph as a flow network where all flows have unit value, and add a source vertices going into L with a sink vertex coming out of R. Then running the Ford-Fulkerson method should give us the maximum flow, which equals the cardinality of the maximal matching.

A more efficient method works by incrementally increasing the size of a matching. For a matching M, an M**-alternating** path is a path whose edges alternate between being in M and $E - M$. An M**-augmenting** path is an M-alternating path whose first and last edges are both in $E - M$. The general idea is as follows: if we find an M-augmenting path in a graph, there must be one more edge in $E - M$ than there is in the subset of M in the path. Therefore, if we make it so that we "recolor" the edges – so the M-edges are removed from M and the $E - M$-edges are added – we will get a matching with one more edge. The following algorithm uses this to find a maximum bipartite matching:

Algorithm 23 Hopcroft-Karp

```
procedure HOPCROFT-KARP($G$)
    $M \leftarrow \emptyset$
    do
        $\mathcal{P} \leftarrow \{P_1, P_2, ..., P_k\}$ maximal set of vertex-disjoint shortest $M$-augmenting paths
        $M \leftarrow M \oplus (P_1 \cup P_2 \cup ... \cup P_k)$
    while $\mathcal{P} == \emptyset$
    return $M$
```

The actual work lies in actually computing the maximal set of vertex-disjoint shortest M-augmenting paths – that is, the largest collection of augmenting paths that share no vertices. To do this, there are three main steps. The first converts G into a directed graph, which is then topologically sorted into a DAG using BFS. Edges in this DAG are alternatingly marked as matching and non-matching. Then the depth of each node with respect to the starting node forms a level graph. Running DFS on the reverse level graph will yield our desired paths. Hopcroft-Karp gives us our maximum matching in $O(E\sqrt{V})$.

Part VII

NP Completeness

Most of the above algorithms are **polynomial time**, i.e. their Kolmogorov complexity is $O(n^k)$. Not all problems can be solved in polynomial time. Problems which can are typically called **tractable**. This section focuses on problems whose tractability is not fully known; no polynomial-time algorithm has yet been discovered, but there is no proof that no polynomial-time algorithm *exists*. It questions the difference between problems which can determinstically be solved in polynomial time (P) and those which can be *verified* determinsitcally in polynomial time (NP). A problem in NP is also defined as being solvable in polynomial time by a **nondeterministic** Turing machine – that is, one which may nondeterminsitically branch into many computational paths. This question – whether $P = NP$ – is among the most famous research problems in theoretical computer science.

An NP problem is one whose solution can be verified in polynomial time (so $P \subseteq NP$). Furthermore, if $P = NP$ then it is accepted that if *any* NP-complete problem is in P then *every* NP-complete problem is in P.

To show that a problem is NP-complete, the goal is to show that the problem is *at least as hard* as a problem in NP. This is done through a mechanism known as a **reduction**. In a reduction, a problem of unknown complexity is manipulated, in polynomial time, into a different problem of known complexity. Both problems must then belong to the same class as the inner problem. A problem H is **NP-hard** if, for *any* NP problem L, there is a polynomial time reduction from L to H. This means that H is at *least* as hard as any problem in NP.

As an example, take the **general boolean satisfiability problem** or SAT. SAT asks whether a satisfying assignment of boolean values can be given to a boolean equation consisting of $\wedge$ (AND), $\vee$ (OR), $\neg$ (NOT), $\implies$ (IMPLICATION), $\iff$ (IF AND ONLY IF), and grouping parentheses. SAT is NP because, given a satisfying assignment, plugging the assignment into the equation can be done in polynomial time. It is NP-hard by the **Cook-Levin theorem**, which, in brief, explains that any problem in NP can be modeled as a program given to a non-deterministic Turing machine, and that the states of such a NDTM may be modeled as a SAT problem. Therefore, accepting this as true, a problem in NP is NP-complete if SAT reduces to it.

A special class of the SAT problem involves a special form, wherein the boolean equation is comprised entirely of conjunctions ($\wedge$) of clauses where each clause is a disjunction ($\vee$) of two (2-SAT) or three (3-SAT) variables or negations. This is known as **conjunctive normal form** (CNF). Any SAT problem can be converted, in polynomial time, into a

3-SAT problem by breaking up long clauses using auxiliary variables. For example, the clause $(x_1 \vee x_2 \vee x_3 \vee x_4, \vee x_5)$ can be rewritten with the two clauses $(x_1 \vee x_2 \vee y_1)$, $(\neg y_1 \vee x_3 \vee y_2)$, $(\neg y_2, \vee x_4 \vee x_5)$. Therefore 3-SAT is NP-complete (it is in NP for the same reason as SAT). However, this is not true for 2-SAT. Looking at the previous example, there is no obvious way break apart long clauses into two-element clauses. In fact, SAT does not reduce to 2-SAT; it is in P. The details of the 2-SAT algorithm are omitted here, but they model the formula as a graph and determine satisfiability by finding SCCs in that graph.

A few more problems are mentioned here, though the exact mechanism of each reduction is omitted. Instead, a high-level overview is provided.

1. **Clique**: A clique in a graph is a subset of vertices which are all mutually pair-connected. The size of a clique is the number of vertices it contains. The decision framing of the clique problem is determining whether a clique of size k exists in a graph G. It can be shown that 3-SAT reduces to CLIQUE. This is done by creating a 3-CNF formula (CNF with clauses of 3 variables each) with k clauses and building a graph out of the variables, where an edge exists between any two variables that are not in the same clause and also do not directly negate each other. If such a graph contains a clique of size k then a satisfying assignment can be found, meaning that the clique problem is as hard as 3-SAT.

2. **Vertex-cover**: A vertex cover is a subset of vertices such that every edge in the graph has at least one vertex in the cover. VERTEX-COVER aims to find whether a graph has a cover of size k. This problem can be shown to be NP-complete by finding a reduction from CLIQUE, by showing that the clique problem on the *complement* graph of G and the cover problem on G are in fact equivalent.

3. **Hamiltonian cycle**: The Hamiltonian cycle problem, i.e. deciding whether a graph contains a cycle that passes through each vertex exactly once, is NP-complete since VERTEX-COVER can be reduced to it.

4. **Traveling Salesman**: The traveling salesman problem (TSP) is a general case of the Hamiltonian cycle problem involving weighted edges on the graph, where instead of just looking for a Hamiltonian cycle, we aim to find a Hamiltonian cycle with total cost (sum of edge weights) at most k. HAM-CYCLE reduces to TSP, and TSP is NP-complete.

5. **Subset sum**: The subset sum problem involves finding if a subset of positive integers S has sum exactly equal to $t > 0$. It can be shown that 3-SAT reduces to SUBSET-SUM, which is itself NP-complete.

Determining whether a reduction exists can be tricky. Typically, reductions usually involve taking a known NP-hard problem, and showing that a polynomial-time transformation of that problem via adding slack variables, dummy nodes to graphs, etc. will result in the desired problem. This, in conjunction with showing the problem is in NP (not just NP-hard), is usually sufficient. Even though there do not exist known polynomial-time true solutions for these problems, in practical settings relaxed versions (where the result is within some cost ratio $\rho(n)$ of the true optimum cost) can be determined with polynomial-time approximation algorithms.