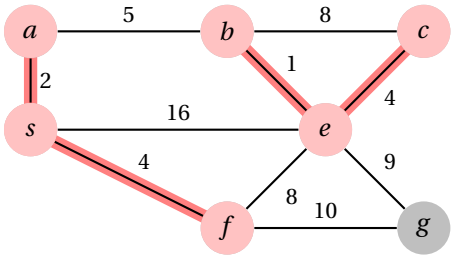
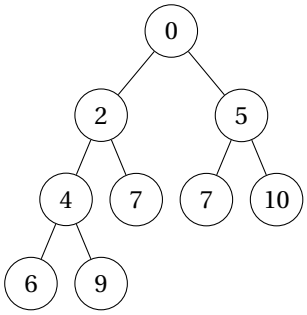
MONASH University

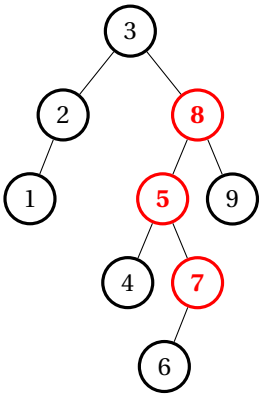
FIT2004

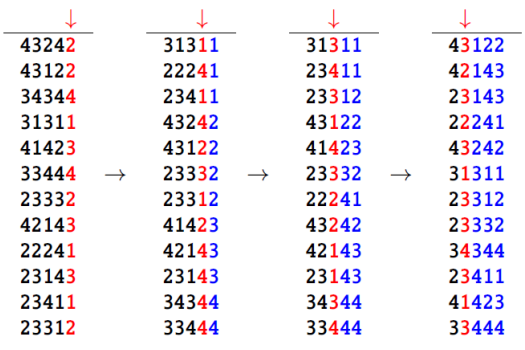
Algorithms and Data Structures

The original version of the course notes were developed by Daniel Anderson. Over the course of the years, additions, removals and modifications were done by members of FIT2004's teaching teams including: Nathan Companez, Rafael Dowsley.









Last updated October 9, 2024

Contents

List of Algorithms

Chapter 1

Analysis of Algorithms

When we write programs, we usually like to test them to give us confidence that they are correct. But testing a program can only ever demonstrate that it is wrong (if we find a case that breaks it). To prove that a program always works, we need to be more mathematical. The two major focuses of algorithm analysis are proving that an algorithm is **correct**, and determining the amount of **resources** (time and space) used by the algorithm.

Summary: Analysis of Algorithms : Part I

In this chapter, we cover:

- What it means to analyse an algorithm.
- Proving correctness of an algorithm.
- An analysis of binary search as an example.
- Analysing the time complexity of an algorithm.
- Recurrence relations, asymptotic notation, and complexity.
- Analysing basic sorting algorithms using these invariants.
- Strong and weak loop invariants.
- Best, average and worst-case performance of simple sorting algorithms.

1.1 Program Verification

Program verification or correctness hinges on two main principles. We must on one hand prove that our algorithm, if it terminates always produces the correct result. On the other, we must prove that it does indeed always terminate. We will explore these issues by analysing one of the most well-known fundamental search algorithms, binary search. An implementation of binary search is shown in Algorithm 1.

Binary search is an easy algorithm to get wrong. The most common mistakes are "off-by-one" errors in which the final index reached by the search is one away from where it should have been. For example, what if line 5 had read key > array[mid] instead? This would be a bug, but given how similar it is to the correct code, it would be easily missed. So how then do we reason that this algorithm is in fact the correct one? We will reason the correctness by establishing an **invariant** in the algorithm. Invariants are a powerful tool for establishing the correctness of algorithms. Arising frequently and being of particular importance are *loop invariants*, which are invariants that are maintained by an iterative procedure (or loop) of a program. Loop invariants help us to

Algorithm 1 Binary Search

```
1: function BINARY_SEARCH(array[1..n], key)
2:     lo = 1 and hi = n + 1
3:     while lo < hi − 1 do
4:         mid = ⌊(lo + hi)/2)⌋
5:         if key ≥ array[mid] then lo = mid
6:         else hi = mid
7:     if array[lo] = key then return lo          // key is located at array[lo]
8:     else return null                           // key not found
```

analyse and establish the correctness of many simple algorithms, which we will explore as an example in this chapter. The useful invariant for binary search is shown below. [1]

Invariant: Binary Search

If $key \in array$, then at each iteration:

1. $array[lo] \leq key$,
2. if $hi \neq n+1$, then $array[hi] > key$.

Combined with the sortedness of array, this implies that *key* (if it exists) lies within the range *[lo..hi)*.

We can now re-interpret the binary search algorithm such that every action it takes is simply aiming to maintain these invariants. This observation can lead us to prove that the algorithm is indeed correct and that it terminates.

1.1.1 Arguing Correctness

When we wish to prove an algorithm's correctness via invariants, we are required to prove three things:

1. Initialisation: We must prove that the invariants hold at the beginning.
2. Maintenance: We must prove that the invariants remain true throughout the algorithm.
3. Termination: We must prove that the invariants at termination imply correctness of the algorithm.

We will argue the proof in two parts. First we will show binary search is correct if the key that we are searching for is in the array. Then we will prove that it is correct when the key is not in the array.

Case 1: key ∈ array

We must first argue that the invariants established above are correct when the algorithm begins. Once established, we subsequently argue that the invariants are maintained throughout the

[1] Here we are considering that the keys are unique, but it is also possible to analise the correctness of binary search with duplicate keys using loop invariants.

algorithm. When we first begin, we initialise $lo = 1$, $hi = n + 1$, so:

1. If *key* ∈ *array*, since the array is sorted, *lo = 1* implies that *array[lo]* is the minimum element, and hence *array[lo]* ≤ *key*.
2. Initially, $hi = n + 1$, so part 2 is satisfied vacuously.

Therefore the invariant is true at the beginning of the binary search algorithm. As we perform iterations of the main loop of the binary search, what happens to this invariant? At each step, we compute $mid = \lfloor (lo + hi) \rfloor / 2$ and compare *array[mid]* to *key*. The conditional statement in the loop then enforces the invariant:

1. If *key* ≥ *array[mid]*, then after setting $lo = mid$, we will still have *array[lo]* ≤ *key*.
2. If *key* < *array[mid]*, then after setting $hi = mid$, we will still have *array[hi]* > *key*.

Hence the invariant holds throughout the iterations of the loop. Therefore, if the loop terminates, it is true that *array[lo]* ≤ *key* < *array[hi]* (or $hi = n + 1$). Since the condition for the loop to terminate is that $lo \geq hi - 1$, combined with the previous inequality, it must be true that *lo = hi - 1*. Therefore if *key* exists in *array*, it must exist at position *lo*, and hence the binary search algorithm correctly identifies whether or not *key* is an element of *array*.

Case 2: key ∉ array

Since key is not in the array, provided that the algorithm terminates, regardless of the value of *lo*, *array[lo]* ≠ *key* and hence the algorithm correctly identifies that *key* is not an element of *array*.

1.1.2 Arguing Termination

We have argued that binary search is correct **if it terminates**, but we still need to prove that the algorithm does not simply loop forever, otherwise it is not correct. Let us examine the loop condition closely. The loop condition for binary search reads *lo* < *hi* - 1, hence while the algorithm is still looping, this inequality holds. We now recall that

$$mid = \left\lfloor \frac{lo + hi}{2} \right\rfloor.$$

Theorem: Finiteness of Binary Search

If *lo* < *hi* - 1, then

$$lo < mid < hi$$

Proof

Since *mid* is the floor of $(lo + hi)/2$, it is true that

$$\frac{lo + hi}{2} - 1 < mid \leq \frac{lo + hi}{2}.$$

Multiplying by two, we find

$$lo + hi - 2 < 2 \times mid \leq lo + hi.$$

Since $lo < hi$-1, we have $lo \leq hi$-2. We replace hi-2 with lo on the left most term in the inequality.

$$2 \times lo < 2 \times mid \leq lo + hi.$$

Next, we add 1 to the right most term.

$$2 \times lo < 2 \times mid < lo + hi + 1.$$

Since $lo < hi$-1, we have lo+1 $< hi$. We replace lo+1 with hi on the right most term.

$$2 \times lo < 2 \times mid < 2 \times hi.$$

and hence

$$lo < mid < hi$$

as desired.

Since this is true, and we always set either lo or hi to be equal to mid, it is always the case that at every iteration, the interval *[lo..hi)* decreases in size by at least one. The interval *[lo..hi)* is finite in size, therefore after some number of iterations it is true that $lo \geq hi$ - 1. Therefore the loop must exit in a finite number of iterations and hence the binary search algorithm terminates in finite time.

1.2 Complexity Analysis

We have proven that the binary search algorithm terminates, and that it terminates with the correct answer. The only thing left to do is to prove that it terminates **fast**. We can express the number of operations performed by the binary search algorithm $T(n)$ as a function of the size n of the input data. The number of operations taken by the binary search algorithm can be expressed as

$$T(n) = \begin{cases} T\left(\frac{n}{2}\right) + a & \text{if } n > 1, \\ b & \text{if } n = 1, \end{cases} \tag{1.1}$$

where a is some constant number of operations that we perform each step, and b is some constant number of operations required at the end of the algorithm.

Theorem: Time Complexity of Binary Search

The recurrence relation (1.1) has the explicit solution

$$T(n) = a \log_2(n) + b$$

Proof

We will prove that $T(n) = a \log_2(n) + b$ by induction on n.

Base case:

Suppose $n = 1$, then $a\log_2(n) + b = b$ which agrees with our definition of $T(n)$.

Inductive step:

Suppose that for some n, it is the case that $T\left(\frac{n}{2}\right) = a\log_2\left(\frac{n}{2}\right) + b$. It is then true that

$$\begin{aligned} T(n) &= T\left(\frac{n}{2}\right) + a \\ &= a\log_2\left(\frac{n}{2}\right) + b + a \\ &= a(\log_2(n) - \log_2(2)) + b + a \\ &= a\log_2(n) - a + b + a \\ &= a\log_2(n) + b \end{aligned}$$

as desired. Hence by induction on n, it is true that

$$T(n) = a\log_2(n) + b.$$

Given this, we can conclude that the time complexity of binary search is $O(\log(n))$.

1.2.1 Common Recurrence Relations

When analysing the time and space complexity of our algorithms, we will frequently encounter some very similar recurrence relationships whose solutions we should be able to recall.

Logarithmic Complexity

The solution to the recurrence relation

$$T(n) = \begin{cases} T\left(\frac{n}{2}\right) + a & \text{if } n > 1, \\ b & \text{if } n = 1, \end{cases}$$

is given by

$$T(n) = a\log_2(n) + b.$$

This is the complexity of binary search that we proved above.

Linear Complexity

The solution to the recurrence relation

$$T(n) = \begin{cases} T(n-1) + a & \text{if} n > 0, \\ b & \text{if} n = 0, \end{cases}$$

is given by

$$T(n) = a\,n + b,$$

Superlinear Complexity

The solution to the recurrence relation

$$T(n) = \begin{cases} 2T\left(\frac{n}{2}\right) + an & \text{if } n > 1, \\ b & \text{if} n = 1, \end{cases}$$

is given by

$$T(n) = an\log(n) + bn.$$

This is the complexity of many divide and conquer sorting algorithms, such as Mergesort.

Quadratic Complexity

The solution to the recurrence relation

$$T(n) = \begin{cases} T(n-1) + cn & \text{if } n > 0, \\ b & \text{if} n = 0, \end{cases}$$

is given by

$$T(n) = c\left(\frac{n(n+1)}{2}\right) + b.$$

This recurrence shows up for example, as the worst-case complexity of Quicksort.

Exponential Complexity

The solution to the recurrence relation

$$T(n) = \begin{cases} 2 \times T(n-1) + a & \text{if } n > 0, \\ b, & \text{if } n = 0, \end{cases}$$

is given by

$$T(n) = (a+b) \times 2^n - a.$$

1.3 Asymptotic Notation

When analysing the complexity of algorithms, we typically concern ourselves only with the order of magnitude of the running time for large values of n. We express time and space complexities in terms of *asymptotic notation*, with which you should be familiar. We use the following notation throughout the course.

Big-O notation

Big-O is the notation that we will use most frequently. Informally, big-O notation denotes an **upper bound** on the size of a function. That is, $f(n) = O(g(n))$ means $f(n)$ is no bigger than than order of magnitude of $g(n)$. Formally, we say that $f(n) = O(g(n))$ as $n \to \infty$ if there are constants c and n_0 such that

$$f(n) \leq c \cdot g(n) \qquad \text{for all } n \geq n_0.$$

This says that for values of n above some threshold, f is always no bigger than some constant multiple of g. Note that this means the behaviour of f for small values of n is irrelevant. Also, remember that big-O bounds can be overestimates. For example, it is correct to say that $2n+1 = O(n^3)$, and also that $2n+1 = O(n)$.

Big-Ω notation

Big-Ω (Omega) notation is a counterpart to big-O notation that denotes a **lower bound** instead of an upper bound. That is, $f(n) = \Omega(g(n))$ means that $f(n)$ is at least as big as the order of magnitude of $g(n)$. Formally, we say that $f(n) = \Omega(g(n))$ as $n \to \infty$ if there are constants c and n_0 such that

$$f(n) \geq c \cdot g(n) \qquad \text{for all } n \geq n_0.$$

Note that this is equivalent to saying that $g(n) = O(f(n))$. Just like big-O bounds could be overestimates, big-Ω bounds can be underestimates. For example, $n^5 = \Omega(n^2)$, and also $n^5 = \Omega(n^5)$.

Big-Θ notation

Big-Θ (Theta) notation means that the bound given is both big-O and big-Ω at the same time, i.e. both an upper bound and a lower bound. Informally, $f(n) = \Theta(g(n))$ means that $f(n)$ is the same order of magnitude as $g(n)$. Formally, we say that $f(n) = \Theta(g(n))$ as $n \to \infty$ if

$$f(n) = O(g(n)) \text{ and } f(n) = \Omega(g(n)) \text{ as } n \to \infty,$$

or equivalently if

$$f(n) = O(g(n)) \text{ and } g(n) = O(f(n)) \text{ as } n \to \infty.$$

Big-Θ notation implies that the bound given is precise, i.e. it is neither an overestimate nor an underestimate. So $n^2 + 3n + 5 = \Theta(n^2)$, but $n^2 + 3n + 5 \neq \Theta(n^3)$, and $n^2 + 3n + 5 \neq \Theta(n)$.

1.4 Measures of Complexity

Best-case, worst-case, and average-case complexity

When measuring the complexity of a deterministic algorithm, we commonly distinguish between three kinds: best case, average case, and worst case. We can define these formally as follows. Let $T_A(I)$ denote the running time of some algorithm A on some input I, and let $\mathbb{I}(n)$ denote the set of all possible valid inputs to the algorithm of size n.

Best-case complexity. The *best-case* complexity of an algorithm is fewest possible instructions that the algorithm will execute over any possible input, as a function of the input size. Formally, the best case complexity is

$$t_{\text{best}}(n) = \min_{I \in \mathbb{I}(n)} T_A(I).$$

Do not fall into the common trap of thinking that the best-case complexity of an algorithm is determined by its behaviour for small inputs (the best case is **not** "when the input is empty").

Worst-case complexity. The *worst-case* complexity is likewise, the most instructions that the algorithm could possibly execute on any possible input, as a function of the input size. Formally,

$$t_{\text{worst}}(n) = \max_{I \in \mathbb{I}(n)} T_A(I).$$

Average-case complexity. *Average-case* complexity is slightly more tricky to define, and in fact there is not a universally agreed upon definition. The simplest definition, and the one that we will use is that the average-case complexity is the average number of operations that the algorithm will execute, averaged over all possible inputs of a given size (i.e., a uniformly random probability distribution over all possible inputs of a given size), as a function of the input size. Formally,

$$t_{\text{average}}(n) = \mathop{\mathbb{E}}_{I \in \mathbb{I}(n)} [T_A(I)].$$

This is typically much more difficult to compute than the other two.

Remark: In this unit we are mostly concerned about worst-case complexity, and when complexity is mentioned without further specification you should assume we are talking about worst-case complexity.

Space complexity vs. auxiliary space complexity. In this unit we differentiate between space complexity and auxiliary space complexity. Space complexity is the total amount of space taken by an algorithm as a function of input size. Auxiliary space complexity is the amount of space taken by an algorithm excluding the space taken by the input.

An in-place algorithm is an algorithm that has $O(1)$ auxiliary space complexity. Some authors use the following definition instead: an algorithm is called *in place* if it does not store or copy the input into any additional data structures, but modifies the input directly to produce the output. More concretely, an algorithm is in place if it never stores more than $O(1)$ elements of the input outside of the input.

Complexity of randomised algorithms

The use of randomisation is a powerful tool for algorithm designers. When the time taken by an algorithm is influenced by random decisions, we typically analyse its time complexity in terms of its *expected complexity*, or we give analyses that hold *with high probability*.

Expected complexity. Expected complexity tells us the average time that a randomised algorithm will take over all possible random decisions that it could make. Note that this is distinct from average-case complexity, which averages over all possible inputs. Unless stated otherwise, we will always analyse expected worst-case performance, i.e. we will analyse the expected performance over all possible random choices for the worst possible input. We could also define best-case, and even average-case expected time complexity, but these are seldom used.

As before, let $T_A(I)$ denote the running time of some algorithm A on some input I. Note that $T_A(I)$ is no longer a single value, but a probability distribution since the algorithm may take different times based on the outcomes of random choices. The expected (worst-case) time complexity of A can then be defined as

$$\max_{I \in \mathbb{I}(n)} \mathbb{E}[T_A(I)].$$

Notice how we still take the maximum value since we care about the worst-case behaviour. It is important to not conflate worst-case behaviour (which depends on the input to the algorithm) and unlucky random decisions.

High probability analyses. We will not cover high probability analysis in this course, but we'll define it here for completeness, since it is frequently used in the analysis of randomised algorithms. We say that an algorithm takes $O(f(n))$ time *with high probability* if it takes $O(c \cdot f(n))$ time with probability at least $1 - \frac{1}{n^c}$ for all $c \geq 1$. Note that this is much stronger than having expected time complexity $O(f(n))$. That is, if an algorithm runs in $O(f(n))$ time with high probability, then its expected time complexity is at most $O(f(n))$ as well (and may even be lower).

1.5 Analysis of Basic Sorting Algorithms

Sorting algorithms are some of the most important and fundamental routines that we will study. They are also very good case studies for the analysis of algorithms. You may be familiar with some of the quadratic ($O(n^2)$) sorting algorithms such as:

- Bubble sort
- Selection sort
- Insertion sort

You probably also know some much faster $O(n \log(n))$ sorting algorithms that we will deal with later. In this chapter, we will analyse selection sort and insertion sort, compare the two, and explore their fundamental underlying properties.

1.5.1 Selection Sort

The selection sort algorithm is based on the following ideas:

Key Ideas: Selection sort

- Consider the list to be sorted as being divided into two parts:
  1. The first part consists of the part of the list that is currently sorted.
  2. The remaining part consists of the elements that are yet to be sorted.
- Initially, the first (sorted) part is empty, while the remaining part is the entire (to be sorted) list.
- We then sort the list like so:
  1. Search the unsorted part for the smallest element.
  2. Swap this smallest element with the first element of the unsorted part.
  3. The sorted part is now one element longer.

Expressed more concretely, an implementation of selection sort is depicted in Algorithm 2.

Algorithm 2 Selection sort

```
1: function SELECTION_SORT(array[1..n])
2:     for i = 1 to n do
3:         min = i
4:         for j = i+1 to n do
5:             if array[j] < array[min] then
6:                 min = j
7:         swap(array[i], array[min])
```

Let's try to understand what is going on here more logically. At some given value of i in the main loop of the algorithm, we have the two sublists

- $array[1..i-1]$, the sorted part
- $array[i..n]$, the yet to be sorted part

The key strategy behind selection sort simply says to progressively make i larger while maintaining the following invariants.

Invariant: Selection sort

For any given value of i in the main loop of selection sort, at the beginning of the iteration, the following invariants hold:

1. $array[1..i-1]$ is sorted
2. For any x in $array[1..i-1]$ and y in $array[i..n]$, x ≤ y

Using these invariants, arguing the correctness of selection sort is easy.

Initial state of the invariants

Initially, when $i = 1$, the sorted part of the array $array[1..0]$ is empty, so the invariants trivially hold. Note that we will often use the notation $array[1..0]$ (or similarly $array[n+1..n]$) to mean an empty array for convenience.

Maintenance of the invariants

At each iteration of the main loop, we seek the minimum *array[j]* for $i \leq j \leq n$. Since at the beginning of iteration i, it is true that $array[1..i-1] \leq array[i..n]$, the value of *array[min]* is no smaller than any $x \in array[1..i-1]$, and hence the extended sorted part $array[1..i]$ remains sorted after swapping $array[i]$ and array[min], so invariant 1 is maintained.

Since we are selecting *array[min]* as a minimum of $array[i..n]$, it must be true that after swapping *array[i]* and *array[min]*, that *array[min]* ≤ $array[i+1..n]$, so invariant 2 is also maintained.

Termination

When the i loop terminates, the invariant must still hold, and since it was maintained in iteration $i = n$, this implies that $array[1..n]$ is sorted, i.e. the entire list is sorted, and we are done.

Time and space complexity of selection sort

Selection sort always does a complete scan of the remainder of the array on each iteration. It is therefore not hard to see that the number of operations performed by selection sort is independent of the contents of the array. Therefore the best, average and worst-case time complexities are all exactly the same.

Theorem: Time complexity of selection sort

The time complexity of selection sort is $O(n^2)$.

Proof

The number of iterations performed is n - i for each i from 1 to n, therefore we perform

$$\begin{aligned}\sum_{i=1}^{n}(n-i) &= n^2 - \sum_{i=1}^{n} i \\ &= n^2 - \frac{n(n+1)}{2} \\ &= \frac{n^2-n}{2} \\ &= O(n^2)\end{aligned}$$

total operations.

We also only require one extra variable and some loop counters, so the amount of extra space required for selection sort is constant.

	Best case	**Average Case**	**Worst Case**
Time	$O(n^2)$	$O(n^2)$	$O(n^2)$
Auxiliary Space	$O(1)$	$O(1)$	$O(1)$

1.5.2 Insertion Sort

Insertion sort is a very similar algorithm to selection sort, based on a similar but *weaker* invariant. The idea behind insertion sort, much like selection sort is to maintain two sublists, one that is sorted and one that is yet to be sorted. At each iteration of the algorithm, we move one new element from the yet-to-be sorted sublist into its correct position in the sorted sublist. An implementation of insertion sort might look something the code shown in Algorithm 3.

Selection sort was based on the idea that we maintain two invariants with respect to the sorted and yet-to-be-sorted sublists. We required that the first sublist be sorted, and that the second sublist contain elements greater or equal to those in the sorted sublist. Insertion sort maintains a weaker invariant shown below. We call the invariant *weaker* since it maintains a subset of the invariants that selection sort did.

Algorithm 3 Insertion sort

```
1: function INSERTION_SORT(array[1..n])
2:     for i = 2 to n do
3:         key = array[i]
4:         j = i - 1
5:         while j > 0 and array[j] > key do
6:             array[j+1] = array[j]
7:             j = j - 1
8:         array[j+1] = key
```

Invariant: Insertion sort

For any given value of i in the main loop of insertion sort, at the beginning of the iteration, the following invariant holds:

1. $array[1..i-1]$ is sorted

Let's prove that this invariant holds and is sufficient to prove that insertion sort is correct.

Initial state of the invariant

Again, the sorted sublist is empty at the beginning of the main loop, so the invariant holds.

Maintenance of the invariant

At the beginning of iteration i, the sublist $array[1..i-1]$ is sorted. The inner while loop of insertion sort swaps the item array[j] one place to the left until $array[j-1] \leq array[j]$. Since $array[1..i-1]$ was originally sorted, it is therefore true that after the termination of the while loop, $array[1..j]$ is sorted. Since the sublist $array[j+1..i]$ was originally sorted, and for each $x \in array[j+1..i]$, we had $array[j] < x$, it is also true that $array[j..i]$ is sorted. Combining these observations, we can conclude that $array[1..i]$ is now sorted, and hence the invariant has been maintained.

Termination

At each iteration of the inner while loop of insertion sort, j is decremented, therefore either the loop condition $j \geq 2$ must eventually be false, or it will be the case that $array[j-1] \leq array[j]$. Since the loop invariant was maintained when $i = n$, it is true that $array[1..n]$ is sorted, and we are done.

Time and space complexity of insertion sort

Unlike selection sort, insertion sort behaves differently depending on the contents of the array. Suppose we provide insertion sort with an already-sorted list. The inner loop will never iterate since it will always be true that $array[j-1] \leq array[j]$. Therefore the only operations required by insertion sort are to perform the outer loop from from 1 to n, so in the best case, insertion sort only takes $O(n)$ time. In the worst case, suppose we provide an array that is sorted in reverse order. We can see that in this case, the inner while loop of insertion sort will have to loop the entire way from i to 1 since it will always be true that $array[j] < array[j-1]$. We observe

that in this case, the amount of work done is the same as selection sort, since we perform i - 1 operations for each i from 2 to n. Therefore in the worst case, insertion sort takes $O(n^2)$ time.

For a random list, on average we will have to swap $array[j]$ with half of the proceeding elements, meaning that we need to perform roughly half of the amount of operations as the worst case. This means that in the average case, insertion sort still takes $O(n^2)$ time[2]. Finally, just like selection sort, the amount of extra space required is constant, since we keep only one extra variable and a loop counter.

	Best case	**Average Case**	**Worst Case**
Time	$O(n)$	$O(n^2)$	$O(n^2)$
Auxiliary Space	$O(1)$	$O(1)$	$O(1)$

1.5.3 Algorithms' Properties

Insertion sort and selection sort are both examples of in-place algorithms for both definitions.

Stable Sorting

A sorting algorithm is said to be *stable* if the relative ordering of elements that compare equal is maintained by the algorithm. What does this mean? Say for example that we are going to sort a list of people's names **by their length**. If multiple names have the same length, a stable sorting algorithm will guarantee that for each class of names with the same length, that their ordering is preserved. For example, if we want to sort the names `Bill`, `Bob`, `Jane`, `Peter`, `Kate` by their length, the names `Bill`, `Jane`, `Kate` all share the same length. A stable sorting algorithm will produce the following sorted list

```
Bob, Bill, Jane, Kate, Peter
```

noticing that `Bill`, `Jane`, `Kate` are relatively in the same order as the initial list. An *unstable* sorting algorithm may have shuffled the order of `Bill`, `Jane`, `Kate` while still producing a list of names that is sorted by length. Insertion sort is an example of a stable sorting algorithm. Selection sort on the other hand is not a stable sorting algorithm.

Online Algorithms

An algorithm (sorting or otherwise) is called *online* if it can process its input sequentially without knowing it all in advance, as opposed to an *offline* algorithm which must know the entire input in advance. Insertion sort is an example of an online sorting algorithm. If you only know some of the input, you can begin to run insertion sort, and continue to read in the rest of the input while continuing to sort, and the algorithm will still work. On the other hand, selection sort is not an online algorithm, since if any new elements are added to the input, the algorithm would have to start all over.

[2] This is not a rigorous argument and hence not a formal proof. Formally proving average-case complexities is actually quite tricky, but we don't usually care about average-case complexity, so we won't delve into the details for now.

Chapter 2

Divide and Conquer

In this chapter we will present the divide-and-conquer algorithm design paradigm and give a few examples of algorithms that use this paradigm.

Summary: Divide and Conquer

In this chapter, we cover:

- The Divide and Conquer paradigm for algorithm design.
- Karatsuba's multiplication algorithm.
- Merge Sort algorithm (revision).
- Counting inversions.

The divide-and-conquer algorithm design paradigm works in 3 steps:

1. *Divide* the problem into smaller subproblems.
2. *Conquer* (i.e., solve) the smaller subproblems.
3. *Combine* the solutions of the smaller subproblems to obtain the solution of the original problem.

Analysing the time complexity of a divide-and-conquer algorithm normally involves solving a recurrence relation.

In this chapter we present a few examples of divide-and-conquer algorithms: we explain the Karatsuba's multiplication algorithm, revise Merge Sort, and introduce an efficient algorithm for counting inversions. In Chapter 3 we deal with Quicksort, a further example of divide-and-conquer algorithm (in which the subproblems are not necessarily of the same size though). Further important examples of applications of the divide-and-conquer strategy (which will not be discussed in this course notes, check algorithms textbooks for details) include: finding the closest pair of points in a plane in $O(n \log n)$, Strassen's sub-cubic algorithm for matrix multiplication, and the Fast Fourier Transform.

Normally a polynomial time solution to the problem is already know, and the divide-and-conquer strategy is used to reduce the time complexity to a lower polynomial.

2.1 Karatsuba's Multiplication Algorithm

Back in our first years in school we learned an algorithm for multiplying two numbers by using only addition and multiplication of single digit numbers (getting partial products as an inter-

mediate step), see Figure 2.1 for a calculation example.

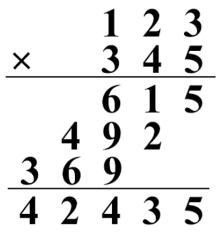


Figure 2.1: Grade school multiplication algorithm using partial products.

By the time we learned that algorithm, our teachers almost certainly did not talk about the efficiency and correctness of this algorithm as by then we were simple users of the algorithm. But, as IT professionals, understanding the efficiency and correctness of algorithms is a central skill in our careers.

Considering addition, subtraction and multiplication of single digit numbers as the basic operations, for the multiplication of two n-digit numbers x and y, this algorithm clearly has complexity $O(n^2)$. And this brings up one of the most important questions in algorithm design: **Can we solve the problem more efficiently?**

In the quest for a sub-quadratic multiplication algorithm, a first improvement idea is the following dividing-and-conquer strategy:

- Split the numbers between the $n/2$ most significant digits and the $n/2$ least significant digits.
- Let x_M and y_M denote the $n/2$ most significant digits of x and y, respectively.
- Let x_L and y_L denote the $n/2$ least significant digits of x and y, respectively.
- Do recursive calls with x_M, x_L, y_M, y_L and use the results to obtain $x \cdot y$.

From math we know that

$$\begin{aligned} x \cdot y &= (x_M \cdot 10^{n/2} + x_L) \cdot (y_M \cdot 10^{n/2} + y_L) \\ &= x_M \cdot y_M \cdot 10^n + x_M \cdot y_L \cdot 10^{n/2} + x_L \cdot y_M \cdot 10^{n/2} + x_L \cdot y_L. \end{aligned}$$

So, we can reduce one instance of the problem of size n to four instances of size $n/2$. Note that multiplications by powers of 10 are trivial left shifts. Are we making some progress?

Not really, if you solve the recurrence $T(n) \leq 4T(n/2) + c \cdot n$, you will verify that the complexity is still $O(n^2)$.

Therefore, if we want to follow this approach to improve the efficiency, we should use at most three recursive calls to subproblems of size $n/2$.

For a long time, no sub-quadratic multiplication algorithm was known, and eventually Russian mathematician Andrey Kolmogorov (one of the greatest mathematicians of the 20th century,

with great contributions to areas such as probability theory and computational complexity) conjectured that sub-quadratic multiplication was impossible.

But then, Anatoly Karatsuba (by the time a 23 years old student) heard the conjecture in one of Kolmogorov's seminar presentations and within one week came up with the first sub-quadratic multiplication algorithm! His solution used the approach described above, but doing only three recursive calls.

Coming back to the equations, the problem in question is to compute

$$\begin{aligned} x \cdot y &= x_M \cdot y_M \cdot 10^n + x_M \cdot y_L \cdot 10^{n/2} + x_L \cdot y_M \cdot 10^{n/2} + x_L \cdot y_L \\ &= x_M \cdot y_M \cdot 10^n + (x_M \cdot y_L + x_L \cdot y_M) \cdot 10^{n/2} + x_L \cdot y_L \end{aligned}$$

using only three recursions to subproblems of size $n/2$.

The first idea presented above computed $x \cdot y$ by using four recursive calls to compute $x_M \cdot y_M$, $x_M \cdot y_L$, $x_L \cdot y_M$ and $x_L \cdot y_L$. But Karatsuba's algorithm only uses the following three recursive calls:

1. $x_M \cdot y_M$
2. $x_L \cdot y_L$
3. $(x_M + x_L) \cdot (y_M + y_L)$

The term $x_M \cdot y_M$ that is multiplied by 10^n is still directly computed by the 1st recursive call. Similarly, the term $x_L \cdot y_L$ that is multiplied by no power of 10 (equivalently, multiplied by $10^0 = 1$) is also still directly computed by the 2nd recursive call. But we need a trivial way to compute $(x_M \cdot y_L + x_L \cdot y_M)$ given the results of the three recursive calls. Let $z = (x_M \cdot y_L + x_L \cdot y_M)$ denote the term we need to find.

The trick is to note that

$$\begin{aligned} (x_M + x_L) \cdot (y_M + y_L) &= x_M \cdot y_M + x_M \cdot y_L + x_L \cdot y_M + x_L \cdot y_L \\ &= z + x_M \cdot y_M + x_L \cdot y_L. \end{aligned}$$

Therefore

$$z = (x_M + x_L) \cdot (y_M + y_L) - x_M \cdot y_M - x_L \cdot y_L$$

and to obtain the desired term z we just need to subtract the results of the 1st and 2nd recursive calls from the result of the 3rd recursive call!

Where does this trick comes from? This trick traces back to the method developed in the 19th century by the German mathematician Carl Friedrich Gauss (one of the greatest mathematicians in history, if not the greatest) to multiply complex numbers using three multiplications of real numbers instead of four. *Adapting previous ideas to solve your new problem can be very useful!*

Note that all the techniques would work the same if binary representation is used instead of decimals (or any other base for that matter).

By solving the recurrence $T(n) \leq 3T(n/2) + c \cdot n$ it can be verified that the time complexity of Karatsuba's algorithm is $O(n^{\log_2 3})$, where $\log_2 3 < 1.59$.

Algorithm 4 Karatsuba's multiplication algorithm

function KARATSUBA(x, y) ▷ x and y are n-digits positive numbers
  if $n = 1$ **then**
    Compute $x \cdot y$ in one step and return the result.
  else
    Let x_M and x_L denote the first and second halves of x, respectively.
    Let y_M and y_L denote the first and second halves of y, respectively.
    Compute $u = (x_M + x_L)$ using the grade school addition algorithm.
    Compute $v = (y_M + y_L)$ using the grade school addition algorithm.
    $a =$ KARATSUBA(x_M, y_M)
    $b =$ KARATSUBA(x_L, y_L)
    $c =$ KARATSUBA(u, v)
    Compute $z = c - a - b$ using the grade school subtraction algorithm.
    Compute $a \cdot 10^n + z \cdot 10^{n/2} + b$ using left shifts and the grade school addition algorithm, and return the result.

This is the algorithm used in Python to multiply large numbers. Don't underestimate the difference between n^2 (blue line in Figure 2.2) and $n^{1.59}$ (green line)!

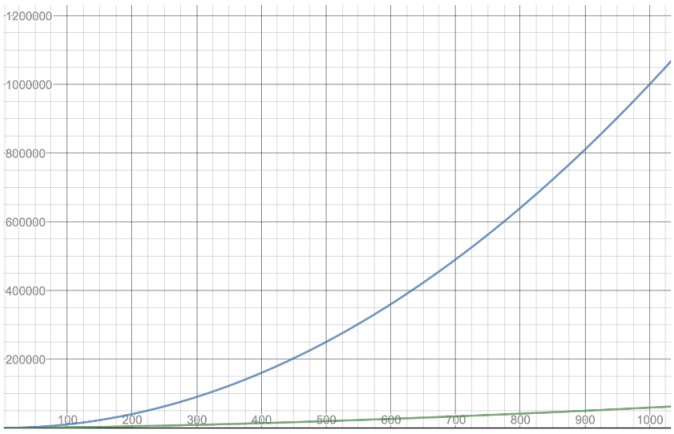


Figure 2.2: Growth of n^2 and $n^{\log_2 3}$.

2.2 Merge Sort (Revision)

Merge sort is a divide-and-conquer sorting algorithm that sorts a given sequence by dividing it into two halves, sorting those halves and then merging the sorted halves back together. How does merge sort sort the two halves? Using merge sort of course! Merge sort is an example of a recursive sorting algorithm, where each half of the sequence is sorted recursively until we reach a base case (a sequence of only one element). The key part of merge sort is the merge routine, which takes two sorted sequences and interleaves them together to obtain a single sorted sequence. The merge algorithm is shown in Algorithm 5.

Observe that this merge routine is stable (since we write $A[i] \leq B[j]$, rather than $<$), and hence

Algorithm 5 Merge

```
1: function MERGE(A[i..n_1], B[j..n_2])
2:     result = empty array
3:     while i ≤ n_1 or j ≤ n_2 do
4:         if j > n_2 or i ≤ n_1 and A[i] ≤ B[j] then
5:             result.append(A[i])
6:             i += 1
7:         else
8:             result.append(B[j])
9:             j += 1
10:    return result
```

our merge sort implementation will also be stable. Using this merge routine, an implementation of merge sort can be expressed like this. The recursive merge sort pseudocode is shown in Algorithm 6.

Algorithm 6 Merge sort

```
1: function MERGE_SORT(array[lo..hi])
2:     if hi > lo then
3:         mid = ⌊(lo + hi)/2⌋
4:         MERGE_SORT(array[lo..mid])
5:         MERGE_SORT(array[mid + 1..hi])
6:         array[lo..hi] = MERGE(array[lo..mid], array[mid + 1..hi])
```

Time and space complexity of merge sort

Merge sort repeatedly splits the input sequence in half until it can no longer do so any more. The number of times that this can occur is $\log_2(n)$, so the depth of the recurrence is $O(\log(n))$. Merging n elements takes $O(n)$ time, so doing this at each level of recursion results in a total of $O(n\log(n))$ time. This happens regardless of the input, hence the best, average and worst-case performances for merge sort are equal. Although the MERGE_SORT routine itself uses no extra space, the MERGE routine uses $O(n)$ extra space, hence the auxiliary space complexity of merge sort is $O(n)$.

	Best case	**Average Case**	**Worst Case**
Time	$O(n\log(n))$	$O(n\log(n))$	$O(n\log(n))$
Auxiliary Space	$O(n)$	$O(n)$	$O(n)$

Enhancements of merge sort

Merge sort can also be implemented *bottom-up*, in which we eliminate the recursion all-together and simply iteratively merge together the sub-arrays of size 1, 2, 4, 8, ... n. This still requires $O(n)$ auxiliary memory, but eliminates the use of the program stack (for recursion) and hence should be slightly faster in practice. There are also in-place implementations of merge sort that require

only $O(1)$ space, but they are much more complicated[1].

2.3 Counting Inversions

Recommender systems are widely employed nowadays to suggest new books, movies, restaurants, etc that a user is likely to enjoy based on his past ratings. One commonly used technique is collaborative filtering, in which the recommender system tries to match your preferences with those of other users, and suggests items that got high ratings from users with similar tastes. A distance measure that can be used to analyse how similar the rankings of different users are is counting the number of inversions. The counting inversions problem is the following:

- **Input:** An array A of n distinct integers.
- **Output:** The number of inversions of A, i.e., the number of pairs of indices (i, j) such that $i < j$ and $A[i] > A[j]$.

The exhaustive search algorithm for solving this problem has time complexity $O(n^2)$, but we want to improve on that and obtain an algorithm with time complexity $O(n \log n)$ for solving this problem. Note that potentially there are $\Theta(n^2)$ inversions, so an algorithm for solving the problem with time complexity $O(n \log n)$ cannot look individually at each possible inversion.

The basic idea to obtain a $O(n \log n)$ solution is to adapt the Merge Sort algorithm to solve this problem. We split the array in the middle and invoke recursive calls on the first half and the second half. Each recursive call will count the number of inversions in that subarray *and also sort the elements of that subarray.* Getting the elements of the subarrays sorted is key to allowing us to count, during the merging procedure, in time $O(n)$ the number of "split inversions", i.e., inversions in which i belongs to the left subarray and j to the right subarray.

When we are performing the merging procedure, at each step the smallest remaining element is selected (and it will be either the first remaining element of the left subarray or of the right subarray, as the subarrays are sorted). If that smallest element comes from the left subarray, then there are no split inversions to be counted (as the index of this element is smaller than the indices of all elements in the right subarray). On the other hand, if that smallest element comes from the right subarray, then the number of split inversions should be increased by the amount of elements still to be processed in the left subarray (as all those elements have smaller indices than the selected one).

The pseudocode of the algorithm is presented below:

[1] See Nicholas Pippenger, Sorting and Selecting in Rounds, *SIAM Journal on Computing* 1987.

Algorithm 7 Sort-and-CountInv

```
1:  function SORT-AND-COUNTINV(array[lo..hi])
2:      if lo = hi then
3:          return (array[lo], 0)
4:      else
5:          mid = ⌊(lo + hi)/2⌋
6:          (array[lo..mid], Inv_L) = SORT-AND-COUNTINV(array[lo..mid])
7:          (array[mid + 1..hi], Inv_H) = SORT-AND-COUNTINV(array[mid + 1..hi])
8:          (array[lo..hi], Inv_S) = MERGE-AND-COUNTSPLITINV(array[lo..mid], array[mid + 1..hi])
9:          Inv = Inv_L + Inv_H + Inv_S
10:         return (array[lo..hi], Inv)
```

Algorithm 8 Merge-and-CountSplitInv

```
1:  function MERGE-AND-COUNTSPLITINV(A[i..n_1], B[j..n_2])
2:      result = empty array
3:      splitInversions = 0
4:      while i ≤ n_1 or j ≤ n_2 do
5:          if j > n_2 or (i ≤ n_1 and A[i] ≤ B[j]) then
6:              result.append(A[i])
7:              i += 1
8:          else
9:              result.append(B[j])
10:             j += 1
11:             splitInversions = splitInversions + n_1 - i + 1
12:     return (result, splitInversions)
```

Chapter 3

Fast Sorting Algorithms

Sorting is a fundamental and interesting problem in computer science. Many sorting algorithms exist, each having their own advantages and disadvantages. Sorting algorithms like selection sort and insertion sort - analysed in Chapter 1 - take $O(n^2)$ time but are very easy to implement and run very fast for very small input sizes. Faster sorting algorithms exists which only take $O(n\log(n))$ time, but which may be more difficult to implement and may run slower for small lists. Merge sort was revised on Chapter 2 and we will now analyse some other algorithms and also explore some interesting theoretical results regarding sorting. We will also take a look at how in many cases, integers can be sorted in just $O(n)$ time.

Summary: Fast Sorting Algorithms

In this chapter, we cover:

- The heapsort algorithm (revision).
- Complexity lower bounds for sorting.
- The Quicksort algorithm.
- Sorting integers in linear time: counting sort and radix sort.

3.1 Heapsort (Revision)

You should be familiar with the heap data structure from your previous units. If you've forgotten, here is a quick reminder:

Definition: Binary Heap Data Structure

The binary heap data structure can be described as follows.

- A binary heap is a complete binary tree (all levels except for the last are completely filled).
- Every element in a max heap is no smaller than its children (i.e. the maximum element is at the top).

Property: Binary Heap Data Structure

The binary heap data structure has the following properties.

- Due to its structure, a binary heap can be represented as a flat array $array[1..n]$ where the root node is array[1] and for each node $array[i]$, its children (if they exist) are elements $array[2i]$ and $array[2i+1]$.
- An existing array can be converted into a heap in place in $O(n)$ time.
- A new item can be inserted into a binary heap in $O(\log(n))$ time.
- The maximum element can be removed from a binary heap in $O(\log(n))$ time.

Making use of the heap data structure, one can easily piece together the heapsort algorithm, which simply involves converting the given sequence into a max heap (*heapifying* it) and then successively removing the maximum element and placing it at the back of the array.

Algorithm 9 Heapsort

```
1: function HEAPSORT(array[1..n])
2:     HEAPIFY(array[1..n])
3:     for i = n to 1 do
4:         array[i] = EXTRACT_MAX(array[1..i])
```

That's it! Of course you'll need to recall how to perform the HEAPIFY and EXTRACT_MAX operations. They are provided below for convenience.

Time and space complexity of heapsort

Since heapsort performs one HEAPIFY which takes $O(n)$ time, and n invocations of EXTRACT_MAX taking up to $O(\log(n))$ time each, the total time spent by heapsort is $O(n\log(n))$. Note that the behaviour of heapsort is independent of the structure of the input array, so its average and worst-case performances are asymptotically the same. Notably however, since HEAPIFY is done in place and each element extracted is placed at the end of the array, heapsort requires only constant extra memory and hence its auxillary space complexity is $O(1)$.

There is one rare situation in which heapsort can run in just $O(n)$, when the entire input array consists of identical elements. In this case, each call to EXTRACT_MAX will take constant time since nothing has to be swapped to maintain the heap property. Heapsort is not an online algorithm since we begin by performing heapify which requires knowing the entire array. It is also not stable since the swaps made when heapifying and removing elements may move equal elements out of their relative order. It is however an in-place algorithm since heapify is in place and each call to EXTRACT_MAX places the element back into the array.

	Best case	**Average Case**	**Worst Case**
Time	$O(n)$	$O(n\log(n))$	$O(n\log(n))$
Auxiliary Space	$O(1)$	$O(1)$	$O(1)$

Binary Heap Operations (Revision)

The standard operations on a min/max heap, implementing the priority queue abstract data type are:

1. HEAPIFY: Convert a given (not necessarily sorted) array into a min/max heap.
2. INSERT: Insert an element into the heap.
3. REMOVE: Remove the min/max value from the heap.

These operations are supported by two internal procedures:

1. RISE: Move an element higher up the heap until it satisfies the heap property.
2. FALL: Move an element down the heap until it satisfies the heap property.

The following pseudocode implements a max heap, i.e. a heap that pops the maximum value.

Algorithm 10 Heapify

```
1: function HEAPIFY(array[1..n])
2:     for i = n/2 to 1 do
3:         FALL(array[1..n], i)
```

Algorithm 11 Heap: Insert

```
1: function INSERT(array[1..n], x)
2:     array.append(x)
3:     n += 1
4:     RISE(array[1..n], n)
```

Algorithm 12 Heap: Delete

```
1: function EXTRACT_MAX(array[1..n])
2:     swap(array[1], array[n])
3:     n = n - 1
4:     FALL(array[1..n], 1)
5:     return array.pop_back()
```

Algorithm 13 Heap: Rise

```
1: function RISE(array[1..n], i)
2:     parent = ⌊i/2⌋
3:     while parent ≥ 1 do
4:         if array[parent] < array[i] then
5:             swap(array[parent], array[i])
6:             i = parent
7:             parent = ⌊i/2⌋
8:         else
9:             break
```

Algorithm 14 Heap: Fall

```
1: function FALL(array[1..n], i)
2:     child = 2i
3:     while child ≤ n do
4:         if child < n and array[child+1] > array[child] then
5:             child += 1
6:         if array[i] < array[child] then
7:             swap(array[i], array[child])
8:             i = child
9:             child = 2i
10:        else
11:            break
```

3.2 Quicksort

Quicksort is perhaps the most well-known divide-and-conquer algorithm for sorting. Quicksort follows these key ideas to sort an array.

Key Ideas: Quicksort

- Select some element of the array, which we will call the *pivot*.
- Partition the array so that all items less then the pivot are to its left, all elements equal to the pivot are in the middle, and all elements greater than the pivot are to its right.
- Quicksort the left part of the array (elements less than the pivot).
- Quicksort the right part of the array (elements greater than the pivot).

The key ideas are very simple. The two major components to correctly (and efficiently) implementing Quicksort are:

- the partitioning algorithm (how does the algorithm move elements lesser/greater than the pivot to the left/right efficiently?)
- how does the algorithm select the pivot element?

The partitioning problem

The partitioning problem is central to Quicksort. The choice of partitioning algorithm affects whether the algorithm is in-place and whether it is stable, and can also influence the worst-case time complexity.

Naive partition algorithm

The naive partitioning algorithm simply partitions the array into three temporary arrays, one for elements that are less than the pivot, one for elements that are greater than the pivot, and one for elements equal to the pivot. Naive partitioning is shown as Algorithm 15

Algorithm 15 Naive partitioning

```
1:  function PARTITION(array[lo..hi], pivot)
2:      left = empty array                          // Store elements less than the pivot
3:      pivots = empty array                        // Store elements equal to the pivot
4:      right = empty array                         // Store elements greater than the pivot
5:      for i = lo to hi do
6:          if array[i] < pivot then left.append(array[i])
7:          else if array[i] = pivot then pivots.append(array[i])
8:          else right.append(array[i])
9:      array[lo..hi] = left + pivots + right       // Concatenate the arrays
10:     return length(left) + ⌈length(pivots)/2⌉    // Return the position of the middle pivot
```

Naive partitioning takes $O(n)$ time and is stable, which are both nice properties. However, naive partitioning is not in-place since it stores all n elements in new arrays, using $O(n)$ space. Creating extra arrays means that the algorithm will be slower in practice than alternatives.

Hoare partition algorithm

The Hoare partition algorithm is the original partitioning algorithm described by C.A.R. Hoare, the inventor of Quicksort. Hoare's partitioning scheme maintains two indices that begin at the start and end of the subarray, which move towards each other while swapping elements that are on the wrong side of the pivot. An implementation of Hoare's partitioning scheme is shown in Algorithm 16. Note that the Hoare partitioning scheme assumes that the pivot is the first element of the range, so our first step is to swap the pivot to the first location so that we can handle more general pivot selection rules later.

The Hoare partitioning scheme is in-place and can be shown to be very efficient in practice. Its main disadvantage compared to the naive scheme is that it is no longer stable.

Dutch National Flag partition algorithm

One disadvantage of the Hoare scheme described above is that it can perform very badly when there are many elements that are equal to the pivot. All elements equal to the pivot will end up in the left side of the partition, which will be very imbalanced if many (or possible even all) elements are equal to the pivot. One way to fix this is to add a post-processing step to the partitioning algorithms that scans the partition and identifies the interval of elements equal to the pivot. These elements can then all be ignored in the subsequent recursive phases of Quicksort.

Algorithm 16 Hoare partitioning

```
1: function PARTITION(array[lo..hi], array[p])   // array[p] is a reference to the pivot element
2:     swap(array[lo], array[p])                 // Move the pivot to the front of the array
3:     i = lo, j = hi
4:     while i ≤ j do
5:         while i ≤ j and array[i] ≤ array[p] do i = i + 1
6:         while i ≤ j and array[j] > array[lo] do j = j − 1
7:         if i ≤ j then swap(array[i], array[j])   // swap elements on the wrong side
8:     swap(array[lo], array[j])                    // swap the pivot into the correct position
9:     return j                                     // Return the position of the pivot
```

An alternative partitioning scheme that accounts for this is the so called *Dutch National Flag* (DNF) scheme proposed by Edsger Dijkstra. The DNF scheme partitions the array into three sections $< p$, $= p$, and $> p$ in-place. Only the $<$ and $>$ subarrays then need to be recursed on. Dijkstra described the partitioning scheme by phrasing it in terms of the following problem.

Problem Statement

Given an array of n objects ($array[1..n]$) coloured **red**, **white** or **blue**, sort them so that all objects of the same colour are together, with the colours in the order **red**, **white** and **blue**.

The connection to the Quicksort partitioning problem should be quite clear: The **red** items correspond to elements less than the pivot, the **white** items correspond to elements equal to the pivot, and the **blue** items correspond to elements greater than the pivot. To solve the DNF problem, we keep three pointers, *lo*, *mid*, and *hi*, such that the following invariant is maintained.

Invariant: Dutch National Flag Partitioning Problem

Maintain three pointers *lo*, mid, hi such that:

- $array[1..lo-1]$ contains the **red** items
- $array[lo..mid-1]$ contains the **white** items
- $array[mid..hi]$ contains the currently unknown items
- $array[hi+1..n]$ contains the **blue** items

Initially, we set $lo = 1$, $mid = 1$, $hi = n$. This means that initially the entire array is the unknown section. We then iteratively move items from the unknown section into their corresponding sections while updating the pointers lo, mid, hi. At each iteration, we move the item at $array[mid]$ (i.e. the first unknown item). There are three cases to consider:

Case 1: *array*[*mid*] is Red:

If $array[mid]$ is **red**, we'd like to move it into the first section. So, we will swap $array[mid]$ with $array[lo]$. This means that the **red** section is extended by one element, so we increment the lo pointer. The item originally at $array[lo]$ was either **white**, in which case it has been moved to

the end of the white section, so we increment mid, or the white section was empty, meaning we simply swapped *array*[*lo*] with itself, then we increment mid. In both situations, we see that the invariant is maintained, and the unknown section has shrunk in size by one element.

Case 2: *array*[*mid*] is White:

If *array*[*mid*] is white, then it is already where we want it to be (right at the end of white section) so we simply have to increment the mid pointer. This also shrinks the unknown section by one element.

Case 3: *array*[*mid*] is Blue:

If *array*[*mid*] is **Blue**, we want to move it into the final section, so we will swap *array*[*mid*] with *array*[*hi*]. The **Blue** section is therefore extended so we decrement the pointer. We do not move the mid pointer in this case since the *array*[*mid*] now contains what was previously the final element in the unknown section.

Pseudocode implementing the DNF partitioning scheme is shown in Algorithm 17.

Algorithm 17 Dutch national flag partitioning

```
1:  function PARTITION(array[1..n], pivot)
2:      lo = 1, mid = 1, hi = n
3:      while mid ≤ hi do
4:          if array[mid] < pivot then                    // Red case
5:              swap(array[mid], array[lo])
6:              lo += 1, mid += 1
7:          else if array[mid] = pivot then               // White case
8:              mid += 1
9:          else                                          // Blue case
10:             swap(array[mid], array[hi])
11:             hi -= 1
12:     return lo, mid
```

Note that we return the final position of the *lo* and *mid* pointers so that we can quickly identify the locations of the three sections after the partitioning. Like Hoare, also notice that this partitioning process is unfortunately not stable.

Implementing Quicksort

We now have the tools to implement Quicksort. Assume for now that we select the first element in the range [*lo*..*hi*] to be our pivot. We will explore more complicated pivot selection rules later. Using our partitioning functions and the fact that they return the resulting location of the pivot, Quicksort can be implemented as shown in Algorithm 18.

Time and space complexity of Quicksort

The running time of Quicksort is entirely dependent on the quality of the pivot that is selected at each stage. Ideally, the resulting partition will be balanced (contain roughly equal amounts of elements on each side) which will minimise the depth of recursion required.

Algorithm 18 Quicksort

```
1: function QUICKSORT(array[lo..hi])
2:     if hi > lo then
3:         pivot = array[lo]
4:         mid = PARTITION(array[lo..hi], pivot)
5:         QUICKSORT(array[lo..mid−1])
6:         QUICKSORT(array[mid+1..hi])
```

Best-case partition

In the ideal case, the pivot turns out to be the median of the array. Recall that the median is the element such that half of the array is less than it and half of the array is greater than it. This will result in us only having $\log_2(n)$ levels of recursion since the size of the array would be halved at each level. At each level, the partitioning takes $O(n)$ total time, and hence the total runtime in the best case for Quicksort is $O(n\log(n))$.

Worst-case partition

In the worst case, the pivot element might be the smallest or largest element of the array. Suppose we select the smallest element of the array as our pivot, then the size of the subproblems solved recursively will be 0 and $n-1$. The size 0 problem requires no effort, but we will have only reduced the size of our remaining subproblem by one. We will therefore have to endure $O(n)$ levels of recursion, on each of which we will perform $O(n)$ work to do the partition. In total, this yields a worst-case run time of $O(n^2)$ for Quicksort.

Average case partition

On average, the partitions obtained by an arbitrary pivot will be neither a perfect median nor the smallest or largest element. This means that on average, the splits should be "okay." Our intuition should tell us that in the average case, Quicksort will take a $O(n\log(n))$ running time, since selecting a terrible pivot at every single level of recursion is extremely unlikely. Let's prove that this intuition is correct.

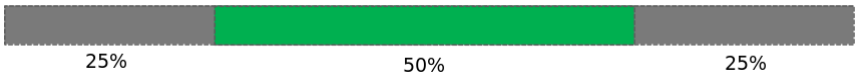
Theorem: Average case run time for Quicksort

The average-case run time for Quicksort is $O(n\log(n))$.

We will present two proofs of this fact. First, an intuitive proof based on coin flips and probability, then a more rigorous proof using recurrence relations.

Proof: Using a coin-flip argument

Consider an arbitrary array of n elements. Ideally, we would like to select a pivot that partitions the array fairly evenly. Let's define a "good pivot" to be one that lies in the middle 50% of the array, i.e. a good pivot is one such that at least 25% of the array is less than it, and 25% of the array is greater than it.



Suppose we are lucky and we select a good pivot every single time. The worst thing that can happen is that the pivot might be on the edge of the good range, i.e. we will select a pivot right on the 25^{th} or 75^{th} percentile. In this case, our recursive calls will have arrays of size $0.25n$ and $0.75n$ to deal with. The longest branch of the recursion tree will therefore consist of the later calls. The number of elements left after d levels of recursion will be given by $0.75^d \times n = \left(\frac{3}{4}\right)^d \times n$, and hence the depth of the tree will be

$$\left(\frac{3}{4}\right)^d \times n = 1 \qquad \Longrightarrow \qquad d = \log_{\frac{4}{3}}(n)$$

Since $\log_{\frac{4}{3}}(n) = O(\log(n))$, the tree has logarithmic height. At each level of recursion we perform $O(n)$ work for the partitioning, and hence the time taken will be $O(n\log(n))$. This should not be surprising, as we already know that the best-case performance for Quicksort is $O(n\log(n))$.

We know that we can not expect to select a good pivot every time, but since the good pivots make up 50% of the array, we have a 50% probability of selecting a good pivot. Therefore we should expect that on average, every second pivot that we select will be good. In other words, we expect to need two flips of an unbiased coin before seeing heads. This means that even if every other pivot is bad and barely improves the sub-problem size, we expect that after $2 \times \log_{\frac{4}{3}}(n)$ levels of recursion to hit the base case. This means that the expected amount of work to Quicksort a random array is just twice the amount of work required in the best case, which was $O(n\log(n))$. Double $O(n\log(n))$ is still $O(n\log(n))$, hence from this we can conclude that the average case complexity of Quicksort is $O(n\log(n))$

Proof: Using recurrence relations (Not examinable in Semester Two, 2024)

Let us denote the time taken to Quicksort an array of size n by $T(n)$. If the k'th smallest element is selected as the pivot, then the running time $T(n)$ will be given by

$$T(n) = n + 1 + T(k-1) + T(n-k),$$

where the terms $T(k-1)$ and $T(n-k)$ correspond to the time taken to perform the recursive calls after the pivot operation. Now observe that over all possible inputs, there are equally many that partition the input into any given sizes $k-1$ and $n-k$. Given this, the average running time over all possible inputs is equal to the average over all possible partitions, and hence

$$\mathbb{E}[T(n)] = n + 1 + \frac{1}{n}\sum_{k=1}^{n}(\mathbb{E}[T(k-1)] + \mathbb{E}[T(n-k)]),$$

where the expectation is taken over all possible inputs. Observe that each term $T(k-1)$

appears a total of twice for each k in the sum, so we have

$$\mathbb{E}[T(n)] = n+1+\frac{2}{n}\sum_{k=1}^{n}\mathbb{E}[T(k-1)].$$

Multiply this equation by n to obtain

$$n\,\mathbb{E}[T(n)] = n^2+n+2\sum_{k=1}^{n}\mathbb{E}[T(k-1)].$$

Now substitute $n-1$ for n and subtract the resulting equation from the above to find

$$n\,\mathbb{E}[T(n)]-(n-1)\mathbb{E}[T(n-1)] = n^2+n-(n-1)^2-(n-1)+2\,\mathbb{E}[T(n-1)].$$

Clean up with some algebra and we will get

$$n\,\mathbb{E}[T(n)] = (n+1)\mathbb{E}[T(n-1)]+2n.$$

Divide by $(n+1)$ and n to find the recurrence

$$\frac{\mathbb{E}[T(n)]}{n+1} = \frac{\mathbb{E}[T(n-1)]}{n}+\frac{2}{n+1}.$$

Telescoping the right hand side, we obtain

$$\begin{aligned}\frac{\mathbb{E}[T(n)]}{n+1} &= \frac{\mathbb{E}[T(n-1)]}{n}+\frac{2}{n+1}.\\ &= \frac{\mathbb{E}[T(n-2)]}{n-1}+\frac{2}{n}+\frac{2}{n+1}\\ &= \quad\vdots\\ &= \sum_{k=1}^{n}\frac{2}{k+1}\end{aligned}$$

where we have stopped at the base case $T(0)=0$. Finally, using the fact that

$$\sum_{k=1}^{n}\frac{2}{k+1} = O(\log(n)),$$

(this is the asymptotic behaviour of the harmonic numbers) we obtain that

$$\mathbb{E}[T(n)] = (n+1)\sum_{k=1}^{n}\frac{2}{k+1} = O(n\log(n)).$$

Space complexity

In the worst case, due to the linear number of recursive calls, we would expect that the auxiliary space required by Quicksort would be $O(n)$. This is true, but can actually be improved by mak-

ing a very simple optimisation. Instead of recursively sorting the left half of the array and then the right half of the array, we can always sort the smallest half first, followed by the largest half. By sorting the smallest half first, we encounter only $O(\log(n))$ levels of recursion on the program stack, while the larger half of the partition is sorted by a tail-recursive call, which adds no overhead to the program stack. If you work in a language without tail-call optimisation, you can replace the second recursion with a loop to achieve the same effect. The complexities written below assume that such an optimisation is made.

	Best case	**Average Case**	**Worst Case**
Time	$O(n\log(n))$	$O(n\log(n))$	$O(n^2)$
Auxiliary Space	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$

For the strict $O(1)$ space definition of in-place, Quicksort is not in-place since it requires at least $O(\log(n))$ space for recursion. Using the other definition, if we use one of the in-place partitioning schemes (Hoare or DNF) then Quicksort will also be in-place, but not stable. If we use a stable partitioning scheme (such as the naive partitioning scheme), then Quicksort will be stable, but not in-place.

3.3 Complexity Lower Bounds for Sorting

Selection sort and insertion sort take $O(n^2)$ time, which is fine for very small lists, but completely impractical when n grows large. Faster sorting algorithms such as merge sort, heapsort and Quicksort run in $O(n\log(n))$ time, which is asymptotically a significant improvement. The next question on your mind should now be, can we do even better than this? An interesting result actually shows us that if we work in the *comparison model*, that is to say that the only valid operations that we can perform on elements are comparisons ($<, >, \geq, \leq, =, \neq$), then sorting can not possibly be done faster than $O(n\log(n))$.

Theorem: Lower bound on comparison-based sorting

Sorting in the comparison model takes $\Omega(n\log(n))$ in the worst case. In other words, any comparison-based sorting algorithm can not run faster than $O(n\log(n))$ in the worst case. Recall that Ω notation denotes an asymptotic lower bound, i.e. the algorithm must take **at least** this long.

To prove the lower bound, we consider a *decision tree* that models the knowledge we have about the order of a particular sequence of data after comparing individual elements. A decision tree that represents the sorting process of a sequence of three elements in shown below. Each node corresponds to a set of potential sorted orders based on the comparisons performed so far. The leaf nodes of the tree correspond to states where we have enough information to know the fully sorted order of the sequence. An example of such a tree is shown in Figure 3.1.

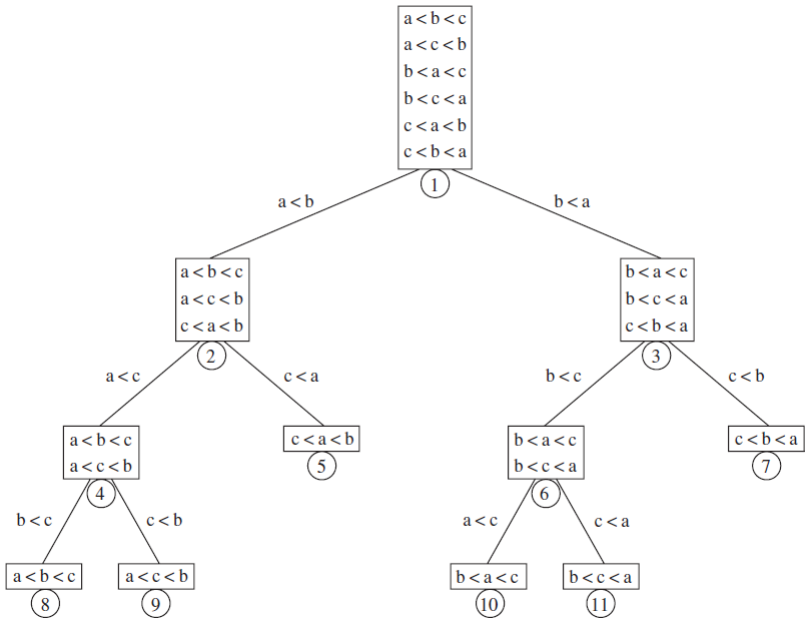


Figure 3.1: A decision tree for comparison-based sorting. Taken from Weiss, Data Structures and Algorithm Analysis in Java, 3rd ed, page 303.

Proof: (Not examinable in Semester Two, 2024)

We appeal to the structure of the decision tree depicted above and observe that comparing elements until we deduce the fully sorted order of a sequence is equivalent to traversing the tree until reaching a leaf node. The worst-case behaviour of any comparison based sorting algorithm therefore corresponds to the depth of the deepest leaf in the tree, i.e. the height of the tree. There are $n!$ total possible sorted orders for a sequence of n elements, each of which must appear as a leaf in the tree. Let's denote the height of the tree by h and observe that a binary tree of height h can not have more than 2^h leaves. Putting this together, we have that

$$n! \leq 2^h.$$

Taking the logarithm of both sides yields

$$h \geq \log_2(n!).$$

To get a lower bound, let's use log laws to rewrite the quantity in question as

$$\log_2(n!) = \log_2(1 \times 2 \times 3 \times \ldots \times n) = \log_2(1) + \log_2(2) + \log_2(3) + \ldots + \log_2(n).$$

Considering just the second half of the terms, since the log function is increasing, we

can write

$$\begin{aligned}\log_2(n!) &\geq \log_2\left(\frac{n}{2}\right)+\log_2\left(\frac{n}{2}+1\right)+\ldots+\log_2(n),\\ &\geq \frac{n}{2}\log_2\left(\frac{n}{2}\right),\\ &= \frac{n}{2}(\log_2(n)-1),\\ &= \Omega(n\log_2(n))\end{aligned}$$

which establishes the desired lower bound. Hence the height of the tree is at least

$$h = \Omega(n\log(n)),$$

which shows that sorting in the comparison model takes $\Omega(n\log(n))$ time.

3.4 Sorting Integers Fast

For arbitrary data in the comparison model, we saw before that there is a $\Omega(n\log(n))$ lower bounded on the time complexity. However, this assumes that the data being sorted has no additional properties that we can exploited. If we make assumptions about the specific kind of data that we are sorting, we can do better. In this section, we will focus specifically on integers, although many of these techniques can also be applied to other kinds of data, such as strings.

As a simple but illustrative example, suppose that we want to sort an array that consists only of 0s and 1s. This can be solved quite easily in linear time by simply counting the number of 0s and 1s in the array and then writing a new array consisting of that many 0s followed by that many 1s. Let's see how this idea can be generalised to arrays of more than just 0s and 1s.

3.4.1 Counting Sort

Suppose we wish to sort an array that contains only integers in some fixed universe U. For example, if all elements are integers in the range 0 to $u-1$. One way to achieve this is to generalise the counting idea that we just saw, and do a pass over the input to count the number of occurrences of each number. If we know that there are for example, one 0, two 1's, one 2 and three 3's, then we can immediately deduce that the sorted array is $0,1,1,2,3,3,3$ in linear time. Of course, simple naively writing out the output would not be a stable sort (indeed, any satellite data attached to the input is lost entirely), so we should instead do a second pass over the input and place each element into its correct position. Computing the position of each element is simple. Let count$[x]$ denote the number of occurrences of x, then the start position of the element y is simply

$$\text{position}[y] = 1+\sum_{x=0}^{y-1}\text{count}[x],$$

which can be computed in linear time in the size of the universe. Each time we place an element y into its correct position, we increment position$[y]$ to prepare for the next occurrence. This algorithm is called counting sort, and is shown below in Algorithm 19. Note that counting sort

can be similarly applied in other domains, e.g. alphabet characters, integers in the range from $-n/2$ to $n/2$, and so on.

Algorithm 19 Counting sort

```
1:  function COUNTING_SORT(array[1..n], u)
2:      counter[0..u−1] = [0,0,...]
3:      for i = 1 to n do
4:          counter[array[i]] += 1
5:      position[0..u−1] = [1,0,...]
6:      for v = 1 to u−1 do
7:          position[v] = position[v−1] + counter[v−1]
8:      temp[1..n] = [0,0,...]
9:      for i = 1 to n do
10:         temp[position[array[i]] = array[i]
11:         position[array[i]] += 1
12:     swap(array, temp)                    // Place the result into the input array
```

This sort is stable but is not in place since we construct the answer in a separate temp array.

Complexity of counting sort

What is the complexity of counting sort? We are required to create and maintain the counter array of size u, and we do two passes over the input. This yields a time and space complexity of $O(n+u)$ in all cases.

	Best case	**Average Case**	**Worst Case**
Time	$O(n+u)$	$O(n+u)$	$O(n+u)$
Auxiliary Space	$O(n+u)$	$O(n+u)$	$O(n+u)$

This tells us that counting sort takes linear time in n if and only if $u = O(n)$. An alternative way to analyse integer sorting algorithms is to consider the *width* of the integers, rather than the maximum value u. The width of an integer is the number of bits required to represent it in binary. If we are sorting w-bit integers, then our universe size is $u = 2^w - 1$. The time complexity of counting sort on w-bit integers is therefore $O(n+2^w)$, which is linear in n if and only if

$$w = \log(n) + O(1).$$

We will improve on this in the next section with an algorithm that can sort integers with a greater number of bits.

3.4.2 Radix Sort

Radix sort is a more general, non-comparison-based sort that achieves linear time for a wider class of inputs than counting sort. There are many variants of radix sort, but we will look at arguably the simplest one, *least significant digit (LSD) radix sort*. In essence, LSD radix sort works

by sorting an array of elements one digit at a time, from the least significant to the most significant. Each digit must be sorted in a stable manner in order to maintain the relative ordering of the previously sorted digits.

Key Ideas: Least significant digit radix sort

- Sort the array one digit at a time, from least significant (rightmost) to most significant (leftmost).
- For each digit: Sort using a stable sorting algorithm.

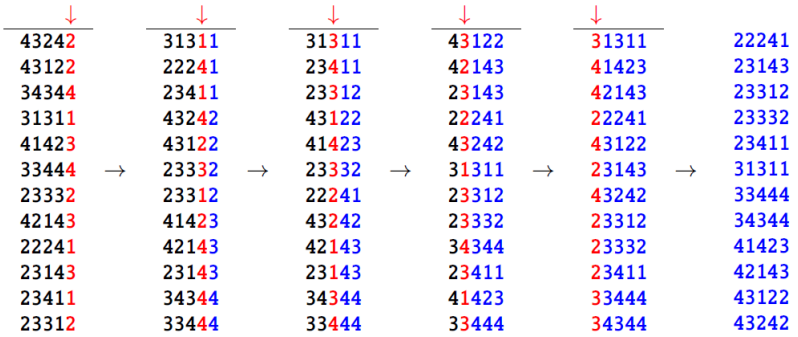


Figure 3.2: An illustration of LSD radix sort. First the numbers are sorted by their least significant digit, then their second, etc. until the most significant digit is sorted and we are finished.

Since individual digits have a very small number of possible values, we will sort the individual digits using counting sort. A visualisation of an LSD radix sort in base-10 is shown in Figure 3.2. It is important to realise that we do not necessarily have to use their *decimal* digits to sort the elements of the array. We could have instead done radix sort with a radix (base) of 100, which would mean sorting consecutive groups of two decimal digits at a time. Using a larger base means fewer passes over the array must be made, but that more work must be done on each pass. It is therefore a tradeoff to find the best base for a given input.

An implementation of LSD radix sort is shown in Algorithm 20. Note that the RADIX_PASS function is simply performing counting sort on a particular digit of the input. The function call GET_DIGIT(x, b, d) computes the value of the d^{th} digit in base b of the integer x.

Complexity of radix sort

Let's suppose that we are sorting n integers that have k digits in base-b. Radix sort will perform k passes over the array, one for each digit, and perform a counting sort each time. Since we are interpreting the integers in base-b, the counting sort will sort over a universe of size b, and hence have complexity $O(n+b)$. The total time complexity of radix sort is therefore $O(k(n+b))$, and the space complexity is just that required of counting sort, so $O(n+b)$.

Algorithm 20 Radix sort

```
1:  function RADIX_PASS(array[1..n], base, digit)
2:      counter[0..base-1] = [0,0,...],
3:      for i = 1 to n do
4:          counter[GET_DIGIT(array[i], base, digit)] += 1
5:      position[0..base-1] = [1,0,...]
6:      for v = 1 to base−1 do
7:          position[value] = position[v−1] + counter[v−1]
8:      temp[1..n] = [0,0,...]
9:      for i = 1 to n do
10:         digit = GET_DIGIT(array[i],base,digit)
11:         temp[position[digit]] = array[i]
12:         position[digit] += 1
13:     swap(array, temp)
14:
15: function RADIX_SORT(array[1..n], base, digits)
16:     for digit = 1 to digits do
17:         RADIX_PASS(array[1..n], base, digit)
```

	Best case	**Average Case**	**Worst Case**
Time	$O(k(n+b))$	$O(k(n+b))$	$O(k(n+b))$
Auxiliary Space	$O(n+b)$	$O(n+b)$	$O(n+b)$

The implementation of radix sort described above is stable but not in place since we use a temporary array for counting sort. Alternate implementations of radix sort can be written that are in place but not stable. In fact, it is even possible to write a stable, in-place radix sort, but it is extremely complicated and well beyond the scope of this course[1].

Improving radix sort

So how does radix sort compare to counting sort? Recall that counting sort can sort a sequence of n integers of maximum value $O(n)$ in linear time. An important face to notice is that for radix sort, the number of digits and the base are not independent, they are related. A larger base will mean fewer digits to sort. Suppose that we wish to sort integers that are w bits wide, i.e. they have maximum value 2^w. Then k, the number of digits, is related to b, the base by

$$k = \frac{w}{\log(b)}.$$

Therefore we can write the complexity of radix sort more precisely as

$$O\left(\frac{w}{\log(b)}(n+b)\right).$$

[1] See G Franceschini, S Muthukrishnan, M Pătraşcu, Radix Sorting with No Extra Space, *ESA 2007*

Comparing this to counting sort, let's say we choose a constant base b, and wish to sort integers of maximum value $O(n)$, i.e. integers with $\log(n)+O(1)$ bits. Then from the above complexity, we see that radix sort will take $O(n\log(n))$ time, since b is a constant, which is worse than counting sort! In order to make radix sort faster, we must therefore choose the base b more cleverly.

Notice that we can make radix sort perform fewer loop iterations by lowering the number of digits, which can be achieved by increasing the base. Of course, increasing the base too much would make the counting sort subroutine too slow, so we need to find a balance point. Since the counting sort subroutine takes $O(n+b)$ time, the largest base that we can pick without slowing it down is $b = O(n)$. If we pick base-n, we then find that the complexity of radix sort will be

$$O\left(\frac{w}{\log(n)}(n+n)\right) = O\left(\frac{w}{\log(n)}n\right).$$

We can now see that for integers with at most $w = O(\log(n))$ bits, the complexity of radix sort will be $O(n)$, which is an improvement over counting sort! In particular, notice that integers with $c\log(n)$ bits have maximum size $O(2^{c\log(n)}) = O(n^c)$, which is better than counting sort, which can only handle integers of size $O(n)$. The space complexity will still be the space complexity of counting sort, which is $O(n+u)$, where $u = n$, hence it will be $O(n)$.

	Best case	**Average Case**	**Worst Case**
Time	$O\left(\frac{w}{\log(n)}n\right)$	$O\left(\frac{w}{\log(n)}n\right)$	$O\left(\frac{w}{\log(n)}n\right)$
Auxiliary Space	$O(n)$	$O(n)$	$O(n)$

Fun (Non-examinable) Fact: We showed here two algorithms that sort integers in linear time provided that they have a bounded number of bits. Surprisingly, linear time sorting is not only possible for integers with a bounded width w. For integers with very large width, specifically those that have at least $w = \Omega(\log^{2+\varepsilon}(n))$ bits for some $\varepsilon > 0$, an algorithm called signature sort can sort them in linear time. It is an open research problem whether integers of any width w can be sorted in linear time. The best known algorithms can sort n integers of any width w in $O(n\sqrt{\log(\log(n))})$ randomised, or $O(n\log(\log(n)))$ deterministic in the worst case[2].

[2]If interested, you should watch this lecture from MIT OpenCourseware, `https://youtu.be/pOKy3RZbSws`

Chapter 4

Order Statistics and Selection

Given a sequence of numbers, many statistical applications are interested in computing the minimum, maximum or median element of the sequence. These are all special cases of a general problem called order statistics, which asks for the k^{th} smallest number in the sequence. In this chapter we will consider several different cases of this problem and explore some algorithms for solving them.

Summary: Order Statistics and Selection Algorithms

In this chapter, we cover:

- The order statistics problem.
- Algorithms for finding minimum and maximums.
- The Quickselect algorithm.
- The median of medians technique.

4.1 Order Statistics and the Selection Problem

The k^{th} order statistic of a sequence of n elements is defined to be the k^{th} smallest element. For example, the first order statistic is the simply the smallest element of the sequence and the n^{th} order statistic is simply the maximum. The median element of a sequence is an element that is smaller than and larger than one half of the other elements of the sequence. In terms of order statistics, the median is the $k = (n+1)/2$ order statistic, rounding either up or down if n is even. The selection problem is the problem of finding for a given sequence and a given value of k, the k^{th} order statistic, i.e. the k^{th} smallest element of the sequence. The easy way to solve the selection problem is of course to simply sort the sequence and then select the k^{th} element that results. Using a fast ($O(n\log(n))$) sorting algorithm would yield an $O(n\log(n))$ solution to the selection problem. We aim to explore faster algorithms for solving the selection problem.

The easy case: finding minimums and maximums

Minimums and maximums are the simplest order statistics to find. Rather than sorting the entire sequence in $O(n\log(n))$, an obvious linear time algorithm to find the minimum and/or maximum is to simply iterate over the sequence and track the best found minimum and/or maximum so far. A possible implementation is shown in Algorithm 21.

That was rather easy, but it raises an interesting question: can we solve this problem any faster or with fewer comparisons in the general case (assuming arbitrary input?) It should not be hard

Algorithm 21 Select minimum

```
1: function SELECT_MIN(array[1..n])
2:     min = array[1]
3:     for i = 2 to n do
4:         if array[i] < min then
5:             min = array[i]
6:     return min
```

to convince yourself that we can not determine the minimum with any fewer than $n-1$ comparisons, since each comparison can only rule out one element as not been the minimum/-maximum.

A more interesting problem arises if we consider the case of finding both the minimum and maximum element at the same time. Clearly, our first algorithm for finding the minimum in $n-1$ comparisons could easily be adapted to find both the minimum and maximum in $2n-2$ comparisons, but can we do better? The answer is no longer obvious since we are doing more comparisons than necessary to find either one of the minimum or maximum on its own. It turns out that we can in fact select both the minimum and maximum element of a sequence in just approximately $1.5n$ comparisons. The clever trick is to just consider each pair of numbers in the input, and for each pair, to only compare the higher of the two with the current maximum, and the lower of the two with the current minimum. This results in cutting out $\frac{1}{4}$ of the comparisons which were unnecessary. The implementation shown in Algorithm 22 demonstrates this trick.

Algorithm 22 Select minimum and maximum

```
1: function SELECT_MINMAX(array[1..n])
2:     min = array[1], max = array[1]
3:     for i = 1 + n mod 2 to n, step 2 do        // start at 1 if n is even, 2 if n is odd
4:         if array[i] < array[i+1] then
5:             if array[i] < min then min = array[i]
6:             if array[i+1] > max then max = array[i+1]
7:         else
8:             if array[i] > max then max = array[i]
9:             if array[i+1] < min then min = array[i+1]
10:    return min, max
```

Interestingly, it is possible to prove that $1.5n$ is the lower bound on the number of comparisons that are required in the worst case to select the minimum and maximum element of a sequence, so this algorithm is in fact performing the optimal number of comparisons.

4.2 The Quickselect Algorithm

Okay, minimums and maximums are easy to find, let's make it harder by trying to find the median, or more generally, the k^{th} order statistic. Remarkably, although minimums and maximums are "easy" problems that require only linear time to solve, the general k^{th} order statistic problem can also be solved in linear time in the worst case. The most well known algorithm for

the order statistics problem is the *Quickselect* algorithm.

The Quickselect algorithm, if the name was not a massive hint already, is based on the Quicksort algorithm. It is based on the observation that if we use Quicksort to find the k^{th} order statistic, then at every level of recursion, sorting the half of the array that does not contain the answer is unnecessary, and can therefore be skipped. As was the case for Quicksort, pivot selection is the crucial step of the algorithm that determines whether or not the run time will be fast or slow. Using the same partition function from Quicksort, a simple realisation of Quickselect can be implemented like so.

Algorithm 23 Quickselect

```
1:  function QUICKSELECT(array[lo..hi], k)
2:      if hi > lo then
3:          pivot = array[lo]
4:          mid = PARTITION(array[lo..hi], pivot)
5:          if k < mid then
6:              return QUICKSELECT(array[lo..mid−1], k)
7:          else if k > mid then
8:              return QUICKSELECT(array[mid+1..hi], k)
9:          else
10:             return array[k]
11:     else
12:         return array[k]
```

Just like Quicksort though, this function has a worst-case run time of $O(n^2)$ since we might select the minimum or maximum element as the pivot and hence perform $O(n)$ levels of recursion. Luckily for us, there are strategies to avoid this sort of pathological behaviour. We will explore two such strategies: random pivot selection which results in an expected $O(n)$ run time and is empirically the fastest strategy, and the median of medians of fives technique, which yields a guaranteed worst-case linear performance, although is not as efficient in practice.

4.3 Randomised Pivot Selection

One of the simplest and empirically the fastest solution to avoid pathological pivot choices is to simply select the pivot randomly.

Theorem: Randomised pivot selection

Using randomised pivot selection, the Quickselect algorithm has expected run time $O(n)$.

Proof: (Not examinable in Semester Two, 2024)

Assume without loss of generality that the array contains no duplicate elements and denote by $T(n)$ the time taken by Quickselect to select the k^{th} order statistic of an array

of size n. If the pivot selected is the i^{th} order statistic of the array, then the time taken by Quickselect will be

$$T(n,i) = n+1+\begin{cases} T(n-i) & \text{if } i<k, \\ 0 & \text{if } i=k, . \\ T(i-1) & \text{if } i>k \end{cases}$$

Averaging out over all possible pivot selections, we obtain an expected run time of

$$\mathbb{E}[T(n)] = n+1+\frac{1}{n}\left(\sum_{i=1}^{k-1}\mathbb{E}[T(n-i)]+\sum_{i=k+1}^{n}\mathbb{E}[T(i-1)]+1\right).$$

Rearranging the indices of the sums, we find that this is the same as

$$\mathbb{E}[T(n)] = n+1+\frac{1}{n}\left(\sum_{i=n-k+1}^{n-1}\mathbb{E}[T(i)]+\sum_{i=k}^{n-1}\mathbb{E}[T(i)]+1\right).$$

Since both of the sums appearing above in the bracketed expression contain a consecutive sequence of terms ending at $T(n-1)$ and there are a total of n terms, they must be bounded above by the largest possible terms, so we have the inequality

$$\mathbb{E}[T(n)] \leq n+1+\frac{2}{n}\sum_{i=n/2}^{n-1}\mathbb{E}[T(i)].$$

We can now prove by induction that $\mathbb{E}[T(n)] \leq Cn = O(n)$ for some constant C. First, we have that $T(0) = T(0) = 1 \leq Cn$ for any choice of $C \geq 1$. Now suppose that $\mathbb{E}[T(n)] \leq Ci$ for some constant C for all $i < n$ for some value of n. We have that

$$\mathbb{E}[T(n)] \leq n+1+\frac{2}{n}\sum_{i=n/2}^{n-1}\mathbb{E}[T(i)],$$

which by our inductive hypothesis satisfies

$$\mathbb{E}[T(n)] \leq n+1+\frac{2}{n}\sum_{i=n/2}^{n-1}Ci.$$

Evaluating the sum using the fact that $\sum_{i=0}^{n} i = \frac{n(n+1)}{2}$, we find

$$\begin{aligned}\mathbb{E}[T(n)] &\leq n+1+\frac{2C}{n}\left(\frac{n(n-1)}{2}-\frac{\frac{n}{2}(\frac{n}{2}-1)}{2}\right) \\ &\leq n+1+C(n-1)-\frac{C}{2}(\frac{n}{2}-1) \\ &\leq \left(\frac{3}{4}C+1\right)n+1-\frac{C}{2}.\end{aligned}$$

If we choose $C = 4$ then we will have

$$T(n) \leq 4n-1 \leq 4n = Cn$$

as desired. Hence by induction on n, we have that $\mathbb{E}[T(n)] = O(n)$, completing the proof.

4.4 Median of Medians

The randomised pivot selection technique yields expected linear time complexity and has excellent empirical performance. From a theoretical standpoint however, it is not deterministic, so there is still room for improvement. Despite the absolute unlikelihood of random pivot choices yielding slow behaviour in practice, it is theoretically satisfying to come up with an algorithm that is deterministically linear in the worst case. One such linear time selection algorithm is the median of medians algorithm, which is simply a special pivot selection strategy for Quickselect. We know that if Quickselect could select the median as the pivot, then it would yield guaranteed linear run time, but of course, finding the median is precisely the problem (or one specific case of the problem) that we are trying to solve. Instead, the median of medians technique involves finding an approximate median that is good enough to guarantee linear time performance.

Key Ideas: Median of medians

The median of medians algorithm selects an approximate median by:

- Dividing the original list of n elements into $\lceil \frac{n}{5} \rceil$ groups of size at most 5.
- Finding the median of each of these groups (just sort them since they are small).
- Finding the true median of the $\lceil \frac{n}{5} \rceil$ medians recursively using Quickselect.

Using these ideas, an implementation of the median of medians technique is shown in Algorithm 24. The function MEDIAN_OF_FIVE selects the true median of a small list using insertion sort.

Algorithm 24 Median of medians

```
1:  function MEDIAN_OF_MEDIANS(array[lo..hi])
2:      if hi - lo < 5 then
3:          return MEDIAN_OF_FIVE(array[lo..hi])
4:      else
5:          medians = empty array
6:          for i = lo to hi, step 5 do
7:              j = min(i + 4, hi)
8:              median = MEDIAN_OF_FIVE(array[i..j])
9:              medians.append(median)
10:         n = length(medians)
11:         return QUICKSELECT(medians[1..n], |(n + 1)/2|)
12:
13: function MEDIAN_OF_FIVE(array[lo..hi])
14:     INSERTION_SORT(array[lo..hi])
15:     return array[(lo + hi)/2]
```

The recursive call made to QUICKSELECT in the median of medians algorithm must then also use MEDIAN_OF_MEDIANS to select its pivot. We call these two functions *mutually recursive* since they do not recurse directly on themselves but rather back and forth on each other.

It remains for us to prove that this technique does indeed result in guaranteed linear time per-

formance. On the one hand, we must ensure that the pivot selected is good enough to yield low recursion depth, and on the other, we need to be convinced that finding the approximate median does not take too long and kill the performance of the algorithm.

Theorem: Worst-case performance of the median of medians algorithm

Quickselect using the median of medians strategy has worst case $O(n)$ performance.

Proof

First, we note that out of the $\frac{n}{5}$ groups, half of them must have their median less than the pivot (by definition of the median). Similarly, half of the $\frac{n}{5}$ groups must have their medians greater than the pivot. Together, we have $\frac{n}{10}$ groups whose median is less than the pivot, and $\frac{n}{10}$ groups whose median is greater than the pivot.

Since within each group of five, two elements are greater than its median and two elements are less than its median, we have transitively that in each of the $\frac{n}{10}$ groups whose median is less than the pivot, three elements that are less than the pivot, and similarly in each of the $\frac{n}{10}$ groups whose median is greater than the pivot, three elements that are greater than the pivot. This yields a grand total of $\frac{3n}{10}$ elements that are below and above the pivot respectively.

Consequently, the median of medians lies in the 30th to 70th percentile of the data. From this, we know that the worst case of the recursive call made by Quickselect will be on a sequence of size $0.7n$. In addition to the recursive call, we also make an additional recursive call of size $0.2n$ to compute the exact median of the original $\frac{n}{5}$ medians. Adding in the linear number operations required to perform the partitioning step and to find the median of each group of 5, a recurrence relation for the amount of work done by Quickselect using the median of medians strategy is

$$T(n) = an + T(0.2n) + T(0.7n), \qquad T(0) = 0,$$

for some constant a. We can show that this recurrence relation has solution $T(n) \leq Cn$ for some constant C by induction. First let C be a constant such that $C \geq 10a$. Consider the base case

$$T(0) = 0 <= C \times 0 = 0,$$

which trivially holds. Now suppose that $T(n) \leq Cn$ for $n < N$, then we have, by substituting this inductive hypothesis

$$\begin{aligned} T(n) &= an + T(0.2n) + T(0.7n) \\ &\leq an + 0.2Cn + 0.7Cn \quad . \\ &= an + 0.9Cn \end{aligned}$$

Since we chose the constant $C \geq 10a$, we have that

$$T(n) \leq an + 0.9Cn \leq 0.1Cn + 0.9Cn = Cn,$$

and hence by induction, for all $n > 0$,

$$T(n) \leq Cn = O(n).$$

Improving Quicksort

Finally, we note that we can use this technique to improve Quicksort. If we use the median as Quicksort's pivot, which is sometimes referred to as Balanced Quicksort, we will get the minimum possible recursion depth, resulting in a guaranteed worst-case $O(n \log(n))$ performance! In practice, this strategy is outperformed by random pivot selection, but it is satisfying to see Quicksort achieve the same worst-case behaviour as heapsort and merge sort.

Chapter 5

Graphs Basics

Graphs are a simple way of modelling sets of objects and encoding relationships between those objects. Graphs have an enormous number of applications in a wide range of fields. Indeed, two of the biggest tech companies in existence, Google and Facebook, base their entire business around two huge graphs! Additionally, graph problems are ubiquitous not only in computer science, but in modelling complex processes and situations in all disciplines of science and business. In this chapter we begin our study of graphs.

Summary: Graphs Basics

In this chapter, we cover:

- Modelling real-world problems with graphs and formal descriptions of graphs.
- Representation and storage techniques for graphs.
- Graph traversal algorithms with applications.
- The concept of shortest paths.
- The topological sorting problem of directed acyclic graphs (DAGs).
- The incremental connectivity problem.
- The union-find linked list data structure.

5.1 Modelling with Graphs

At the most basic level, graphs tell us about relationships between pairs of objects. Graphs that model objects with particular attributes are often referred to as *networks*, although the terms can usually be used interchangeably, so don't worry about the distinction. For example, consider a social network where friends connect with each other. We can represent this information in a graph, where each person is represented by a *node* or *vertex* of the graph, and a connection between friends is denoted by an *edge* or *arc* connecting them. Given this graph, several interesting questions might immediately come to mind:

- What are the groups of mutual friends? I.e. friends that have a friend in common, or a friend-of-a-friend etc. These are called the *connected components* of the graph.
- What is the largest degree of separation between two people in the network? This would correspond to the longest *distance* between any people in the graph - also called its *diameter*.

- Are there any people in the network whom if removed would cause a pair of mutual friends to no longer be mutual friends? Such a vertex in a graph is called an *articulation point* or *cut point*.

Graph and network algorithms can help us solve each of these problems.

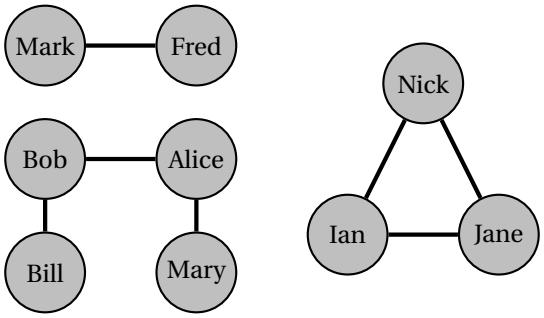


Figure 5.1: A network representing a set of people who have connections with each other.

Graphs can also be *weighted*. As another example, consider a road map of towns in the country, where each town is represented by a vertex, and roads between towns are represented by edges with *weights* corresponding to the distance between them.

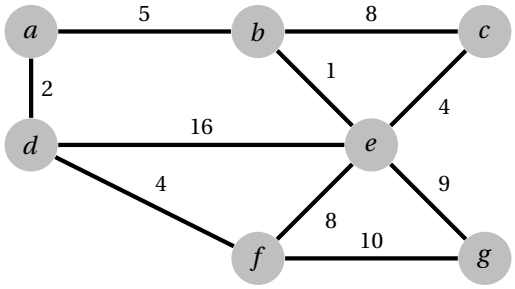


Figure 5.2: A network representing a road map of country towns.

Several interesting questions can be asked about such a graph:

- Given a pair of towns, what is the shortest route connecting them? This is called the *shortest path problem*.
- What is the overall shortest subset of roads that we could keep, removing all others while still keeping every town connected? This is called the *minimum spanning tree problem*.

Edges in a graph can also be directed, that is the relation only holds in one direction but not both. For example, consider your courses that you are taking for your degree, which have prerequisites that must be completed first. We can model this with a graph, in which a course has a directed edge to another course if the first course is a prerequisite of the second course and hence must be completed first.

Some more interesting questions now arise:

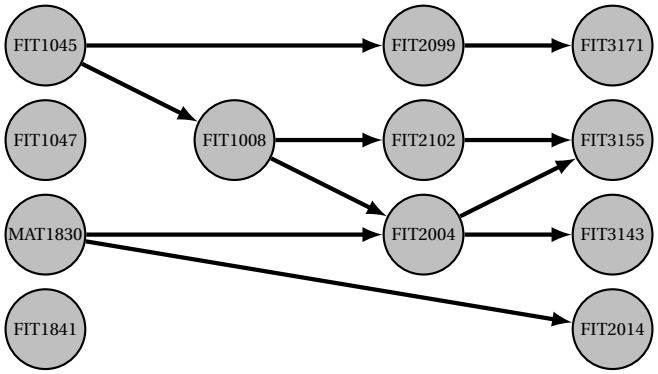


Figure 5.3: A network representing a course progression for a Computer Science degree.

- Are there any cycles in the course graph? If so, the degree is impossible to complete! Ooops.

- If there aren't any cycles, what is a valid order to complete the courses in that satisfies all of the prerequisites? Such an order is called a *topological ordering* of the directed graph.

- If there aren't any cycles, what is the longest chain of prerequisites? I.e. assuming we could handle as many courses at a time as we wanted, how many semesters would it take us to complete the degree? This is called the *critical path problem*.

Once again, graph and network algorithms can help us solve all of these and many more problems!

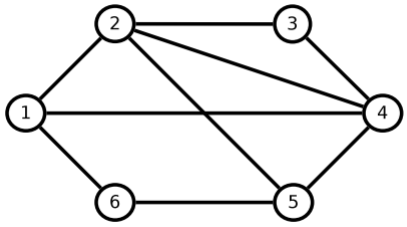
Formal graph notation

Formally, a graph G is defined by a pair of sets V and E, where:

- V is the the set of vertices/nodes.

- E is the set of edges/arcs, where an edge e connects two nodes u and v.

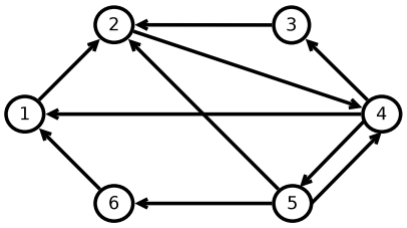
The number of vertices in a graph is usually either denoted by $|V|$, or as n or N. Sometimes when the distinction is obvious and the writer is lazy, people will simply write V to refer to the number of vertices. Similarly, we denote the number of edges by $|E|$, m or M, or occasionally when being lazy and not totally accurate, by E.



An unweighted, undirected graph.

Direction

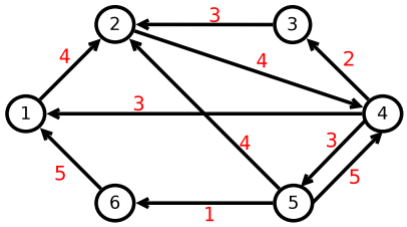
The edges of a graph may be *directed* or *undirected*. In an undirected graph, the edges (u, v) and (v, u) are equivalent, but in a directed graph, they represent different things: for example, a one-way street in a road map, or a course prerequisite, which are not the same if you reverse them!



An unweighted, directed graph.

Weights

The edges may also be *unweighted* or *weighted,* in which case they have an associated quantity, which may represent for example, the distance between two towns in a road map, the bandwidth of a connection in an internet network, the amount of money that it would cost to connect two houses via fibre-optic cable, or anything else you can imagine. We will usually denote the weight of the edge (u, v) by $w(u, v)$.



A weighted, directed graph.

Multigraphs and loops

Depending on the particular problem, it may be allowed or disallowed for a graph to contain multiple edges between the same pair of vertices. A graph that does contain multiple edges between the same pair of vertices is usually referred to as a *multigraph*.

Similarly, for a particular purpose, a graph may or may not be allowed to contain edges that connect a vertex to itself. Such edges are usually referred to as *loops*. A graph that contains no loops or multiple edges between the same pair of vertices is called a *simple graph*. If we do not specify otherwise, we will assume that the graphs we consider in this unit are simple by default.

Directed acyclic graph

A directed acyclic graph (DAG) is a directed graph that contains no cycles. DAGs are useful in particular for representing tasks that have prerequisites. For example, the units in your degree and their prerequisites form a DAG, where each edge denotes a unit that has another as a requirement. In project management, the tasks in a project might form a DAG which indicates which tasks must be performed before others. The two most fundamental and interesting problems on DAGs are the topological sorting problem and the critical path problem.

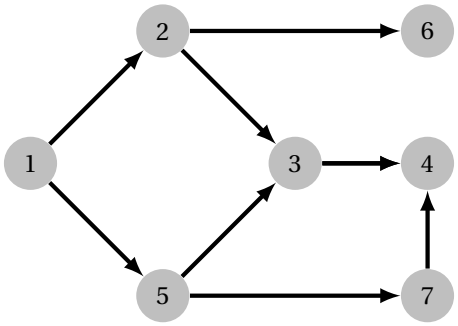


Figure 5.4: A directed acyclic graph.

5.2 Representation and Storage of Graphs

There are several options that we may choose from when it comes to actually storing a graph for use in a computer program. The choice that we make will depend on the particular variety of graph and on the problem that we plan on solving.

Considerations - Density of the graph

The biggest factor that will influence our decision of how to store our graph will be its *density,* which informally means does it have few edges or lots of edges. If we allow multiple edges between vertices, our graph can have an unbounded number of edges, so assume for now that our graphs do not have multiple edges.

- In a directed graph, the maximum number of possible edges that we can have is therefore $|V|^2$ if we allow loops, or $|V|(|V|-1)$ otherwise.
- In an undirected graph, the maximum number of possible edges is $\binom{|V|+1}{2}$ if we allow loops, or $\binom{|V|}{2}$ otherwise.

We say that a graph is *dense* if $|E| \approx |V|^2$, that is informally, the graph has a lot of edges. Conversely, we call a graph *sparse* if $|E| \ll |V|^2$, i.e. the graph has a relatively small number of edges.

Representation strategy – Adjacency Matrix

One way to represent a graph is using an *adjacency matrix*. The adjacency matrix of a graph $G = (V, E)$ is a matrix A of size $|V| \times |V|$. The space requirement of storing an adjacency matrix is therefore $O(|V|^2)$. When the edges are unweighted, the entries of the adjacency matrix are defined by

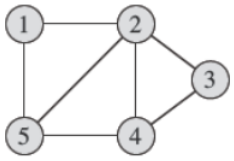
$$a_{i,j} = \begin{cases} 1, & \text{if there is an edge } (i,j) \\ 0, & \text{otherwise} \end{cases}.$$

In the case of multigraphs, adjacency matrices can be generalised such that $a_{i,j} = k$ where k is the number of edges between vertices i and j.

In a weighted graph, the adjacency matrix stores the weights of the edges, such that

$$a_{i,j} = \begin{cases} w(i,j), & \text{if there is an edge } (i,j) \\ 0 \text{ or } \infty, & \text{otherwise} \end{cases}.$$

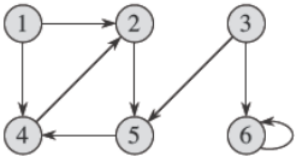
The choice of whether to use 0 or ∞ as an indicator that a particular edge is absent will depend completely on the problem at hand. The choice must be made such that the inclusion of the absent edge with the given weight does not change the solution to the problem being considered. Alternatively, if this is not feasible, one could store two matrices, one indicating adjacency without weights, and one storing the weights.



	1	2	3	4	5
1	0	1	0	0	1
2	1	0	1	1	1
3	0	1	0	1	0
4	0	1	1	0	1
5	1	1	0	1	0

An example of an adjacency matrix for an unweighted, undirected graph. Source: CLRS

In an undirected graph, the adjacency matrix will clearly be symmetric, so if we wish to, we can save memory by only storing the upper triangular entries. Given an adjacency matrix, checking whether two vertices have an edge between them can be done in constant $O(1)$ time by simply checking the corresponding entry in the adjacency matrix.

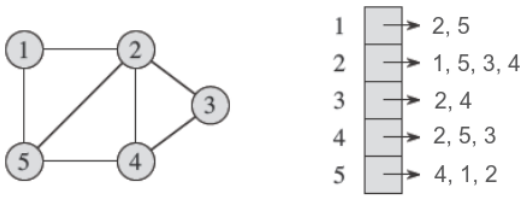


	1	2	3	4	5	6
1	0	1	0	1	0	0
2	0	0	0	0	1	0
3	0	0	0	0	1	1
4	0	1	0	0	0	0
5	0	0	0	1	0	0
6	0	0	0	0	0	1

An example of an adjacency matrix for an unweighted, directed graph. Source: CLRS

Representation strategy – Adjacency List

Adjacency matrices are good for representing dense graphs since they use a fixed amount of memory that is proportional to the total number of possible edges. However, when a graph is sparse, the corresponding adjacency matrix will contain a very large number of 0 entries, using the same $O(|V|^2)$ space regardless of the actual number of edges in the graph. In this case, a strong alternative is to store the graph in the form of an *adjacency list.* An adjacency list is simply a list of lists, where each list corresponding to a particular vertex stores the vertices that the given vertex is adjacent to. In a weighted graph, the adjacency list will store the weights of the edges alongside the vertices corresponding to those edges.



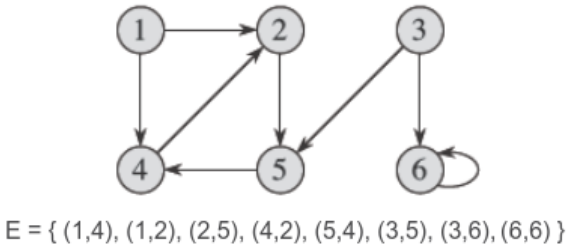
An example of an adjacency list for an unweighted, undirected graph. Source: CLRS

Since an adjacency list only uses memory for edges that actually exist in the graph, the space required for this representation is $O(|V|+|E|)$, which is a huge improvement over adjacency matrices for sparse graphs, and about the same for dense graphs.

Unlike with adjacency matrices, we can no longer check whether two given vertices are adjacent in constant time using an adjacency list. Adjacency lists are however, particularly convenient when we only need to iterate over the existing edges since we will never need to examine a non-existent edge. We could instead store our adjacency list for each vertex in a balanced binary search tree, which would allow for $O(\log(|E|))$ adjacency checks while retaining the ability to iterate efficiently. Such a strategy would use more memory however and is not particularly useful for most applications.

Representation strategy – Edge List

Although only rarely, sometimes we will opt for the *edge list* strategy. In this case, we simply store a list that contains all of the edges of the graph in no particular order.



An example of an edge list for an unweighted, directed graph. Source: CLRS

The edge list uses only $O(|E|)$ space, but affords us neither the ability to check adjacency quickly or iterate over the adjacent vertices of a particular vertex efficiently. Their primary use is in Kruskal's minimum spanning tree algorithm where we need to sort all of the edges in descending order of weight, which cannot be done effectively with the other two representation strategies.

5.3 Graph Traversal and Applications

A huge number of graph algorithms are built on a few small key ideas. The simplest and most wide reaching archetype of graph algorithm are the depth-first and breadth-first traversals. *Traversing* a graph simply means to explore it and figure out some of its properties.

5.3.1 Depth-First Search

A depth-first traversal as the name implies, searches a graph by following a path as deep as possible from some starting vertex before reaching a dead end, where there are no more adjacent and unvisited vertices, then backtracking up the path, continuing the search for vertices that have yet to be explored. In a depth-first traversal, we maintain a flag on each node that indicates whether or not we have visited that node yet. When a node is visited, we mark its flag as such in order to avoid visiting the same node multiple times and causing infinite repetition. An example graph is shown in Figure 5.5, with its vertices labelled in the order that a depth-first search might visit them when starting from node s. We note that depth-first search always visits every node that is reachable from the source vertex, and since it cannot visit a node twice, it never creates any cycles in its path. This implies that the edges travelled by the search form a tree, which we call a *depth-first search tree*.

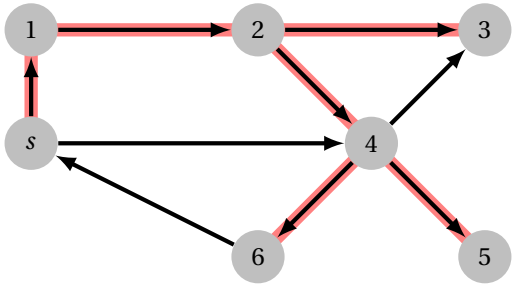


Figure 5.5: A graph with it's nodes labelled in the order that they might be visited by a depth-first search. The highlighted edges are those traversed by the search, forming a depth-first search tree.

Depth-first search can be implemented very succinctly using recursion. We will assume that the graph is represented as an adjacency list, so that we can access adjacent vertices in constant time. A depth-first search is shown in Algorithm 25. Note that since the graph might not be connected, we loop over all vertices and make a call to DFS if that vertex has not been visited. Otherwise we may only visit one component of the graph (although for some applications, this may be all that is needed). This algorithm is applicable to both directed and undirected graphs.

We make the following assumptions to simplify the code:

1. Vertices are interchangeable with their ID number. That is, we will frequently loop over all vertices by looping over $1..n$, and we will read from and assign to arrays using vertices as indices.

2. Properties of the graph, for example the adjacency list, are visible to the DFS function. This avoids us having to pass around lots of extra arguments.

Algorithm 25 Generic depth-first search

```
1:  // Driver function that calls DFS until everything has been visited
2:  function TRAVERSE(G = (V, E))
3:      visited[1..n] = false
4:      for each vertex u = 1 to n do
5:          if not visited[u] then
6:              DFS(u)
7:
8:  function DFS(u)
9:      visited[u] = true
10:     for each vertex v adjacent to u do
11:         if not visited[v] then
12:             DFS(v)
```

3. Quantities associated with the graph that we are working on are visible to the DFS function. For example, we assume that the DFS function can see and use the visited array even though it was declared in a different scope. This too saves on passing extra arguments around.

Since the depth-first search algorithm visits every vertex only once and examines every edge at most once (or twice for an undirected graph), its time complexity is $O(|V|+|E|)$. Despite its simplicity, the skeleton DFS algorithm forms the basis of a huge number of graph algorithms with wide-ranging applications. We will now explore a few of these applications.

Some applications of depth-first search

- Finding the connected components of a graph – the connected components of a graph are the maximal connected subgraphs, that is, they are the subsets of vertices that are mutually connected, directly or indirectly.
- Two-colouring a graph, or deciding that the graph cannot be two-coloured – a graph can be two-coloured if every vertex can be assigned a colour, white or black, such that no two neighbouring vertices share the same colour. This is equivalent to the graph being bipartite.
- Finding a cycle in a graph – a cycle is a path consisting of a non-empty sequence of distinct, adjacent edges that begins and ends at the same vertex.
- Finding a topological ordering of a directed, acyclic graph (DAG) – a topological ordering is an ordering of the vertices of DAG such that all vertices v that are reachable from the vertex u come after it in the topological ordering, in other words, they are orderings that satisfy prerequisite relationships.
- Finding bridges and cut-points of a graph – these are edges and vertices that if removed from the graph would disconnect some connected component. In a sense they are "non-redundant" connections.

5.3.2 Finding Connected Components

One of the simplest and most useful applications of graph traversal and depth-first search is finding the *connected components* of an undirected graph. The connected components of an undirected graph are the maximal connected subgraphs, that is, they are the subsets of vertices that are mutually connected, directly or indirectly.

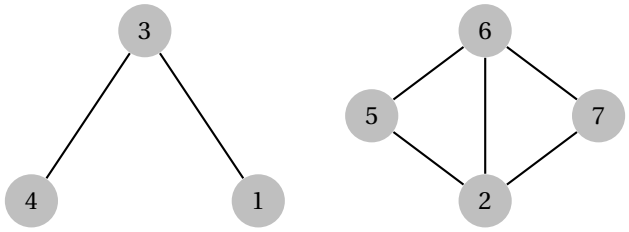


Figure 5.6: An unweighted, undirected graph with two connected components

We can find the connected components by observing that a single call to the depth-first search function visits precisely the connected component of the starting vertex, and nothing else. So we can simply run a depth-first search from each vertex that has not yet been visited by a previous one, and each time, we will discover one more component of the graph. An implementation is shown in Algorithm 26. Since the depth-first search runs in $O(|V|+|E|)$ time, this algorithm for determining the connected components also runs in $O(|V|+|E|)$.

Algorithm 26 Finding connected components using depth-first search

```
1:  // Driver function that finds each connected component
2:  function CONNECTED_COMPONENTS(G = (V, E))
3:      component[1..n] = null
4:      num_components = 0
5:      for each vertex u = 1 to n do
6:          if component[u] = null then
7:              num_components += 1
8:              DFS(u, num_components)
9:      return num_components, component[1..n]
10:
11: // One DFS will visit a single connected component
12: function DFS(u, comp_num)
13:     component[u] = comp_num
14:     for each vertex v adjacent to u do
15:         if component[v] = null then
16:             DFS(v, comp_num)
```

After running CONNECTED_COMPONENTS, num_components will contain the number of connected components, each of which is identified by an integer from 1 to num_components. Each vertex $u \in V$ will have the ID number of the connected component that contains it stored in component[u]. Note that we do not need an explicit visited array, since we know whether or not a vertex has been visited by whether it has been assigned a component number yet. It is very common in depth-first search algorithms for the visited array to become redundant like this.

5.3.3 Cycle Finding

Another useful application of depth-first search is locating cycles in graphs. Recall that a cycle is a path consisting of a non-empty sequence of distinct, adjacent edges that begins and ends at the same vertex. Consider an undirected, unweighted graph $G = (V, E)$. Suppose without loss of generality that G is connected. Performing a depth-first search on G will form a depth-first search tree T that covers every vertex $v \in V$. If at any point the search finds an edge that leads to a vertex that it has already visited, then this edge connects two vertices u, v that are already connected in T and hence forms part of a cycle. We must be careful not to accidentally interpret the edge we just came from as a cycle though (since this "cycle" would use the same edge twice). To account for this, we'll have our DFS keep an extra parameter p, the vertex that we just came from, so that we know not to interpret (u, p) as part of a cycle. Putting all of this together, an algorithm to find cycles in undirected graphs is shown in Algorithm 27.

Algorithm 27 Cycle detection in an undirected graph using depth-first search

```
1:  // Driver function that calls DFS to look for a cycle
2:  function HAS_CYCLE(G = (V, E))
3:      visited[1..n] = false
4:      for each vertex u = 1 to n do
5:          if not visited[u] and DFS(u, null) = true then
6:              return true
7:      return false
8:
9:  // Returns True if a cycle was detected
10: function DFS(u, p)
11:     visited[u] = true
12:     for each vertex v adjacent to u do
13:         if visited[v] and v ≠ p then                    // We found a cycle!
14:             return true
15:         else if v ≠ p and DFS(v, u) = true then
16:             return true
17:     return false
```

5.3.4 Breadth-First Search

Breadth-first search, like depth-first search traverses a graph one vertex at a time, never visiting a vertex more than once, until all reachable vertices have been visited. The only difference between the two is the order in which the vertices are visited. Many problems that can be solved with a depth-first search can also be solved with a breadth-first search. While a depth-first search explores paths as deep as possible immediately, a breadth-first search visits nearby vertices first and further away vertices later. Formally, breadth-first search always visits every vertex that is a distance k from the starting point before visiting any vertices that are a distance $k+1$. Like depth-first search, breadth-first search also produces a tree, but a much more special tree. Specifically, the tree produced by breadth-first search is a *shortest path tree*, a tree in which every path from the source s is a shortest path in the original graph.

Typically we use breadth-first search to explore a graph that is assumed to be connected, and hence we do not initiate multiple searches like we did in the depth-first search. This is com-

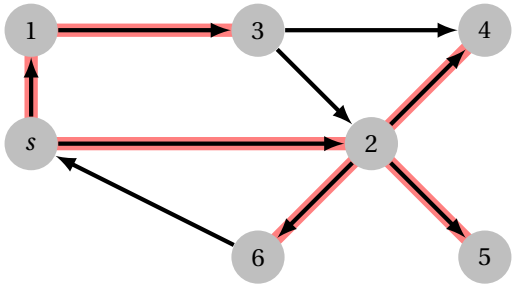


Figure 5.7: A graph with it's nodes labelled in the order that they might be visited by a breadth-first search. The traversed edges form a shortest path tree from s. Observe that all paths from s in the tree use the fewest edges possible to reach their respective destinations.

pletely a matter of convention. You can perform breadth-first search from multiple components if you want, and you may even search from multiple starting locations simultaneously. An implementation of breadth-first search is shown in Algorithm 28.

Algorithm 28 Generic breadth-first search

```
1:  function BFS(G = (V, E), s)
2:      visited[1..n] = false
3:      visited[s] = true
4:      queue = Queue()
5:      queue.push(s)
6:      while queue is not empty do
7:          u = queue.pop()
8:          for each vertex v adjacent to u do
9:              if not visited[v] then
10:                 visited[v] = true
11:                 queue.push(v)
```

In order to visit vertices that are nearby sooner, the algorithm maintains a queue of vertices that need to be visited, appending progressively further vertices to the end of the queue as it traverses. Finally, note that just like the depth-first search, breadth-first search also visits each vertex at most once and examines each edge at most once (or twice for an undirected graph), hence its time complexity is $O(|V|+|E|)$.

5.4 Shortest Paths

Shortest path problems are one of the most natural and useful kinds of graph problems. If you've ever used Google maps to find your way to a destination, then you've seen shortest paths problems in action.

The notion of a shortest path is intuitive, and may be stated informally as a path between vertices of a graph that minimises the total weight of the edges used in the path. One of the most obvious examples would be the idea of finding the shortest route on a road map from one location to another, in which we must decide which roads to take in order to minimise either the

total distance travelled, or the time taken, or indeed any other quantity that we are interested in, such as toll costs. Note that we usually refer to **a** shortest path rather than **the** shortest path since there may be multiple equally short paths between two vertices.

Before diving straight in to the algorithms, we'll first take a look at some useful properties that shortest paths have which will help us in understanding the algorithms and how they work.

5.4.1 Properties of Shortest Paths

Shortest paths have optimal sub-structure

The first and most important observation that we can make about shortest paths is that they contain massive amounts of sub-structure. Consider the graph in Figure 5.8. We can see that the shortest path from $1 \rightsquigarrow 7$ is through vertices $1 \to 2 \to 4 \to 7$ with a total weight of 6. Given that this is true, we know for sure that a shortest path from vertex 1 to vertex 4 must be $1 \to 2 \to 4$. Why? Since $1 \to 2 \to 4$ is part of the shortest path from $1 \rightsquigarrow 7$ (it is a *sub-path*), if it were not itself a shortest path, then we could replace this part of the original path with a shorter one, making our supposed shortest path even shorter. (Think about how you would prove this formally. **Hint:** Use contradiction.)

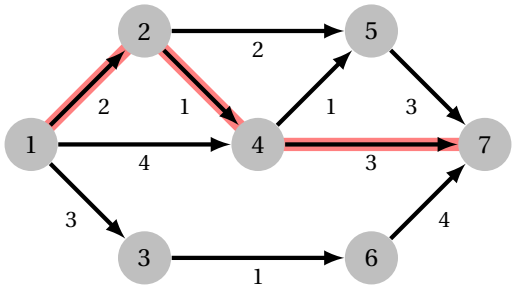


Figure 5.8: An weighted, directed graph. A shortest path from vertex 1 to vertex 7 has a total weight of 6, passing through the vertices $1 \to 2 \to 4 \to 7$.

Shortest paths satisfy the "triangle inequality"

Let us denote by $\delta(s,v)$ the length of a shortest path from vertex s to vertex v. The triangle inequality says that for any edge $(u,v) \in E$ with weight $w(u,v)$, it is true that

$$\delta(s,v) \leq \delta(s,u) + w(u,v).$$

Behind these fancy symbols is a simple and intuitive but very critical fact that underpins most shortest path algorithms. All that this says in plain English is that if we have a shortest path $s \rightsquigarrow v$, then we can't find a path $s \rightsquigarrow u$, followed by an edge $u \to v$ with a shorter overall length, because this would mean that $s \rightsquigarrow u \to v$ would be an even shorter path. Expressed even more simply, you can't find a path that is shorter than your shortest path!

Why is this true? If the shortest path from vertex s to vertex u has total length $\delta(s,u)$ and the edge (u,v) has weight $w(u,v)$ then we can form a path to vertex v by taking a shortest path from s to u and then traversing the edge (u,v), forming a path from s to v. The total length of this

path is $\delta(s,u)+w(u,v)$, so this length cannot be shorter than the length of the shortest path, since it is itself a valid path. Try to formalise this argument using contradiction.

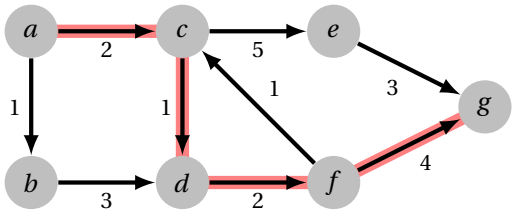


Figure 5.9: A weighted, directed graph. A shortest path from vertex a to vertex g has length 9. $\delta(a,e)=7$, so the triangle inequality tells us that $\delta(a,g)\leq\delta(a,e)+w(e,g)=7+3=10$, which is indeed satisfied. Since the bound is not tight, i.e. $\delta(a,g)<\delta(a,e)+w(e,g)$, this implies that the edge (e,g) is not part of a shortest path from a to g.

We can disregard cycles

Consider a path that contains a cycle in it. If this cycle has a positive total weight, then we can remove it from the path and the total weight decreases, so this path was not a shortest path. If the cycle has a negative total weight, then we may travel around the cycle over and over for as long as we wish and accumulate arbitrarily short paths, so the notion of a shortest path is not well defined in this case. Finally, if the path contains a cycle of zero total weight, then we may remove it and obtain a path with the same total weight. Therefore without loss of generality, we may assume that shortest paths are simple paths (contain no cycles) when looking for them.

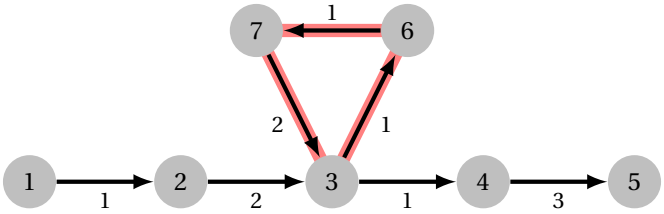


Figure 5.10: An edge weighted, directed graph containing a directed cycle. The cycle can not possibly be a part of a shortest path since it only makes the path length longer without going anywhere!.

5.4.2 Shortest Path Problem Variants

There are three main variants of the shortest path problems that we might wish to consider.

Problem Statement: The single-pair shortest path problem

Given a weighted, directed graph $G=(V,E)$ and a pair of vertices $u,v\in V$, find a shortest path between u and v. Formally, find a sequence of adjacent vertices $(v_1,v_2,...,v_k)$

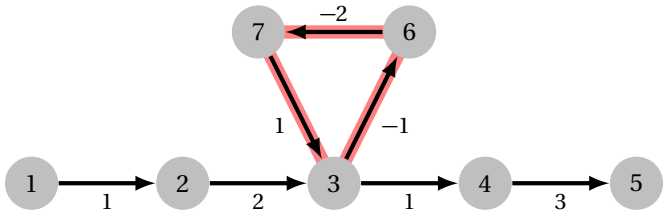


Figure 5.11: An edge weighted, directed graph containing a directed cycle with a negative total weight. The shortest path between vertex 1 and 5 is now **undefined**. For any path that you give me, I can always make an even shorter one by adding in another traversal of the cycle.

with $v_1 = u$ and $v_k = v$ such that the sum

$$\sum_{i=1}^{k-1} w(v_i, v_{i+1})$$

is minimised. In an unweighted graph, we consider $w(v_i, v_{i+1}) \equiv 1$, which is equivalent to asking for a path from u to v containing the fewest possible edges.

Although single-pair is seemingly the simplest possible shortest path problem, it turns out that the more general variants are typically simpler and asymptotically no more difficult to solve.

Problem Statement: The single-source shortest path problem

Given a weighted, directed graph $G = (V, E)$ and a starting vertex $s \in V$, find for every vertex $v \in V$ a shortest path from s to v.

Given the copious amounts of substructure that is present in shortest paths, it should be intuitive that we can solve the single-source problem much faster than we could solve a single-pair problem separately for every possible vertex v.

Problem Statement: The all-pairs shortest path problem

Given a weighted, directed graph $G = (V, E)$, find a shortest path between every pair of vertices $u, v \in V$.

There is much overlap between these problems, and indeed each of them can be used to solve the former one. We will only explore algorithms for the last two of them since in general, it is usually the same difficulty to solve the first problem as the second, except in some special cases.

In this chapter we will show an algorithm that finds the shortest path in unweighted graphs. In Chapter 6 we present a greedy algorithm to solve the shortest path problem in graphs with non-negative weights, and in Chapter 8 we present dynamic programming algorithms that can compute shortest paths on graphs with negative weights.

5.4.3 Unweighted Shortest Paths

Due to the fact that it visits vertices in distance order, breadth-first search can be used for finding the shortest paths in an unweighted graph from a given starting vertex s. We will track information about shortest paths by maintaining two arrays, dist[1..n] and pred[1..n], where dist[u] denotes the distance of the vertex u from our starting vertex s, and pred[u] denotes the vertex that proceeded u on the shortest path from s (the edge that the breadth-first search tree used to get to u). By maintaining these two arrays, we will be able to reconstruct the shortest paths from s to every other vertex. An implementation is illustrated in Algorithm 29.

Algorithm 29 Single-source shortest paths in an unweighted graph

```
1:  function BFS(G = (V, E), s)
2:      dist[1..n] = ∞
3:      pred[1..n] = null
4:      queue = Queue()
5:      queue.push(s)
6:      dist[s] = 0
7:      while queue is not empty do
8:          u = queue.pop()
9:          for each vertex v adjacent to u do
10:             if dist[v] = ∞ then
11:                 dist[v] = dist[u] + 1
12:                 pred[v] = u
13:                 queue.push(v)
```

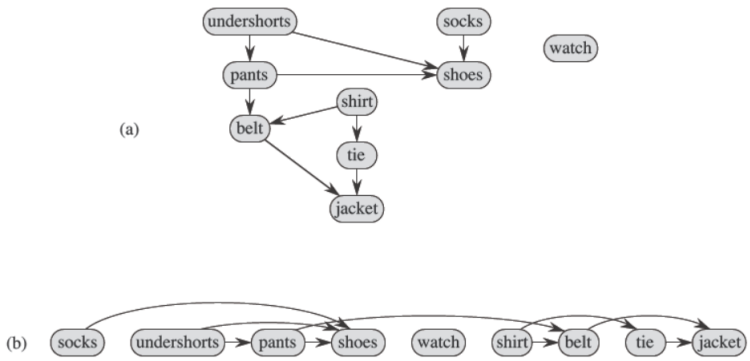
Note that this algorithm does not only compute the shortest path between one pair of vertices, it computes the shortest paths from the source vertex s to **every other** reachable vertex in the graph. Once we have the shortest path information, we can reconstruct the sequence of vertices that make up the shortest path from s to the vertex u by backtracking through the pred array until we reach s. An implementation is depicted in Algorithm 30.

Algorithm 30 Reconstruct shortest path

```
1:  function GET_PATH(s, u, pred[1..n])
2:      path = [u]
3:      while u ≠ s do
4:          path.append(pred[u])
5:          u = pred[u]
6:      return reverse(path)
```

5.5 The Topological Sorting Problem

Given a DAG $G = (V, E)$, a topological ordering is a permutation of the vertices such that for any directed edge $e = (u, v)$, the vertex u occurs before the vertex v. In other words, if the edges of the graph represent prerequisites, then a topological ordering represents a valid order in which to complete the tasks such that every prerequisite is satisfied. The topological sorting problem is the problem of producing a valid topological ordering for a given DAG.



A DAG (a) representing prerequisites for getting dressed and a correct topological ordering of the DAG (b). Image source: CLRS

5.5.1 Kahn's Algorithm for Topological Sorting

The most well-known algorithm for topological sorting is Kahn's algorithm, which maintains a queue of vertices that are ready to be completed and inserts them one by one into the topological ordering. A vertex is considered ready to be completed if it has no incoming edges (prerequisites) remaining. Once a vertex is taken and inserted into the ordering, any outgoing edges that it has are removed and its descendants are checked to see whether they are now ready too. An implementation is shown in Algorithm 31.

Algorithm 31 Topological sorting using Kahn's algorithm

```
1:  function TOPOLOGICAL_SORT(G = (V, E))
2:      order = empty array
3:      ready = queue of all vertices with no incoming edges
4:      while ready is not empty do
5:          u = ready.pop()
6:          order.append(u)
7:          for each edge (u, v) adjacent to u do
8:              Remove (u, v) from G
9:              if v has no remaining incoming edges then
10:                 ready.push(v)
11:     return order
```

Since Kahn's algorithm visits each vertex once and removes every edge once, its time complexity is $O(|V|+|E|)$ if implemented with the appropriate data structures. In practice, for efficiency, we do not actually delete edges from the graph during Kahn's algorithm. What we usually do is maintain an array that remembers the in-degree of every vertex (the number of incoming edges). Whenever we pop a vertex u, we decrease by one the in-degree of all vertices v such that there exists an edge (u, v). When this number hits zero, we know that the vertex has no more incoming edges and hence is ready to be added to the queue.

5.5.2 Topological Sorting Using DFS

Another useful algorithm for computing topological orderings is via a depth-first search. The key idea is that by performing a depth first search from some vertex v, we necessarily visit all of the vertices that depend on v. Therefore if during depth-first search we add each vertex to an array after we have finished visiting all of its descendants, its descendants will have already been added to the array before it, so the array will contain a reverse topological order. Since we are just performing a depth-first search and appending $|V|$ items to an array, the time complexity of this algorithm is also $O(|V|+|E|)$. An implementation is shown in Algorithm 32.

Algorithm 32 Topological sorting using DFS

```
1:  function TOPOLOGICAL_SORT(G = (V, E))
2:      order = empty array
3:      visited[1..n] = false
4:      for each vertex v = 1 to n do
5:          if not visited[v] then
6:              DFS(v)
7:      return reverse(order)
8:
9:  function DFS(u)
10:     visited[u] = true
11:     for each vertex v adjacent to u do
12:         if not visited[v] then
13:             DFS(v)
14:     order.append(u)                    // Add to order after visiting descendants
```

5.6 Incremental Graph Connectivity

Connectivity in undirected graphs is a simple but widely applicable problem. We've seen how to find the connected components of a graph using depth-first search. Once the components have been identified, the problem of determining whether various pairs of nodes are connected can then be solved in constant time by checking whether the two vertices have the same component ID. In some applications that we will see later, it is useful to be able to determine the connectivity of a graph that is changing, i.e. has edges being added to it in between queries. This problem is called the incremental connectivity problem. The incremental connectivity problem can be solved by reducing it to the disjoint set problem, which can be solved fast using a data structure called union-find.

Two vertices u, v in a graph G are considered to be *connected* if there exists a path between them. This can be determined in linear time by running a search from u or v and checking whether the other vertex gets visited. Of course, if we wish to check large numbers of pairs of vertices, doing a search each and every time would be redundant and inefficient. A better solution is to first find the connected components of the graph, which can be done in linear time with a depth-first or breadth-first search. Recall Algorithm 26.

With the connected components known, each connectivity query can be answered in constant time by simply checking whether the two vertices are assigned to the same component or not.

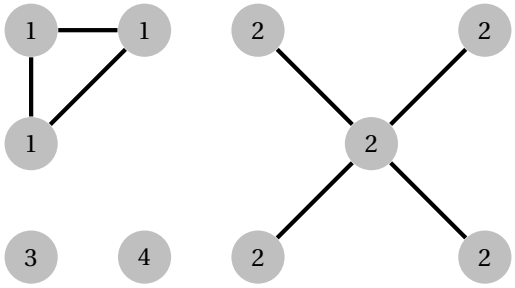


Figure 5.12: An undirected graph where each vertex is labelled by its connected component number. Note that two vertices are connected if and only if they have the same number.

Algorithm 33 Connectivity check

```
1: function CONNECTED(u, v)
2:     return (component[u] = component[v])
```

The problem becomes much more interesting (and difficult) if we want to modify the graph in between queries. One solution is to simply recompute the connected components each time we make a modification, but this will get expensive if we begin to make lots of them. This problem is called the *dynamic connectivity* problem. If we restrict ourselves to the case in which edges are only added, and not deleted, this is called the *incremental connectivity* problem.

The incremental connectivity problem is the problem of determining whether two vertices u, v in a graph G are connected, while allowing the graph G to be modified, that is, have new edges inserted in between queries. Simply rerunning the connected components algorithm every time we make a modification would work, but is very inefficient, so we'd like to find a better way to approach the problem.

Formally, the incremental connectivity problem is the problem of finding a data structure that can support the following two operations:

1. CONNECTED(u, v): Check whether the vertices u and v are connected.
2. LINK(u, v): Add an edge between the vertices u and v.

We'd like to perform these operations fast, so doing a fresh search per CONNECTED query or recomputing the entire connected components per LINK operation is too slow. This problem can be solved efficiently using a data structure called a *union-find* or *disjoint-set* data structure.

5.6.1 The Union-Find Disjoint-Set Data Structure

The *union-find* or *disjoint-set* data structure is a data structure for maintaining a collection of n elements, each of which belongs to a single set, that allows us to merge the contents of some pair of sets together. Each disjoint set is identified by a *representative*, which is some arbitrary element of that set.

Formally, the operations supported by a union-find structure are:

1. FIND(u): Determine which set the element u is contained in. This means returning the identity of the representative of the set containing u.
2. UNION(u, v): Join the contents of the sets containing u and v into a single set. This may change the representatives of the sets.

These two operations can be seen as equivalent to maintaining the connected components of an undirected graph, where each item in a set represents a vertex, and each union operation represents the addition of an edge which might connect two components. Checking whether two vertices are in the same connected component then reduces to checking whether they have the same representative, i.e. whether FIND(u) = FIND(v).

Disjoint set forests

Disjoint set forests represent each set by a rooted tree, where each node represents one element and the root node of the tree is taken to be the representative element. For each node in the forest, all that we need to store is a pointer to its parent in its tree. If a node is a root node, then we traditionally indicate this by setting its parent pointer to itself. The disjoint set operations are then supported like this:

1. FIND(u): To find the representative of u, we simply follow the parent pointers until we reach the root of the tree containing it.
2. UNION(u,v): To merge the sets containing u and v, we first check that they are not already contained in the same set, and if not, point the root node of one of the trees to the root node of the other, making one tree a subtree of the other.

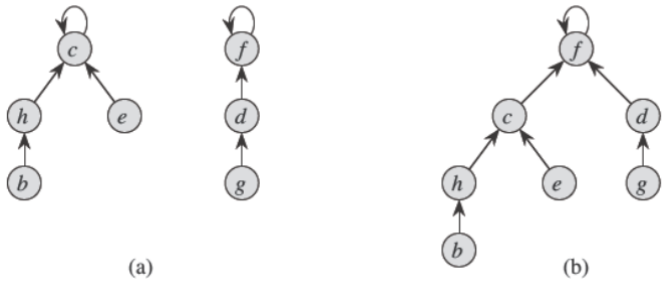


Figure 5.13: A disjoint set forest. The result of merging the two sets shown in (a) is depicted by (b). The left tree has become a subtree of the right tree. Figure source: CLRS

An implementation is depicted in Algorithm 34. So far, this is not very efficient, since a sequence of union operations may result in a long chain, which will make FIND operations take $O(n)$ time per query. To improve this, we make two optimisations.

The path compression technique

The first technique that we can apply to speed up union-find is *path compression*. The effect of path compression is very simple, whenever we perform the FIND operation to locate the root

Algorithm 34 Union-find using disjoint-set forests (without optimisations)

```
1: function INITIALISE(n)
2:     parent[1..n] = 1..n
3:
4: function FIND(x)
5:     if parent[x] = x then return x
6:     else return FIND(parent[x])
7:
8: function UNION(x, y)
9:     parent[FIND(x)] = FIND(y)
```

of a particular node's tree, we have to look at all of the nodes on the path to the root. We can therefore modify the parent pointer for each node on the path to point directly at the root, so that long paths are compressed and will never have to be traversed again. This has the effect of flattening the tree and removing long chains, which significantly speeds up future FIND queries. It can be shown that with path compression, the worst-case cost of performing m operations is $O(m\log(n))$, i.e. at most $O(\log(n))$ per operation in total.

An implementation of FIND with path compression is shown in Algorithm 35. By using recursion, path compression can be implemented cleverly in just one line!

Algorithm 35 The FIND operation using path compression

```
1: function FIND(x)
2:     if parent[x] ≠ x then parent[x] = FIND(parent[x])
3:     return parent[x]
```

The union-by-rank technique

In union by rank, we maintain a *rank* for each tree, which is an upper bound on the height of the tree. When merging two trees together, we choose to always make the tree with a smaller rank a child of the tree with the larger rank. This results in more balanced trees since it encourages shorter tree to become the subtree of the larger one. It can be shown that with union by rank, the worst-case cost of performing m operations is also $O(m\log(n))$, i.e. at most $O(\log(n))$ per operation in total.

An implementation of UNION with union by rank is shown in Algorithm 36.

Combining path compression union by rank

Combining both path compression and union-by-rank speeds up the data structure even more. It can be shown that in the worst case, a sequence of m operations will cost $O(m\alpha(n))$, where $\alpha(n)$ is the *inverse Ackermann function*, an extremely slowly growing function. To put it in perspective, $\alpha(n) \leq 4$ for all values of n that can be written in the universe, so for all practical purposes (but not theoretical purposes) it is a constant. This is a significant improvement over the $O(\log(n))$ complexity of path compression and union by rank individually. This complexity

Algorithm 36 The UNION operation using union by rank

```
1:  function INITIALISE(n)
2:      parent[1..n] = 1..n
3:      rank[1..n] = 0
4:
5:  function UNION(x, y)
6:      x = FIND(x)
7:      y = FIND(y)
8:      if rank[x] < rank[y] then
9:          parent[x] = y
10:     else
11:         parent[y] = x
12:         if rank[x] = rank[y] then
13:             rank[x] += 1
```

bound also turns out to be provably optimal, in other words, no disjoint-set data structure can possibly do better than this.

Chapter 6

Greedy Algorithms

In this chapter, we will study greedy algorithms. Greedy algorithms make a locally optimal choice at each stage and commit to it. Often it is easy to come up with ideas to solve a problem using a greedy strategies, but in many cases the proposed greedy algorithm does not find the overall optimal solution for the problem (thus correctness proofs are very important). Some of the most important greedy algorithms solve problems on graphs and will be studied in this chapter.

Summary: Greedy Algorithms

In this chapter, we cover:

- Dijkstra's algorithm for finding shortest paths in graphs with non-negative weights.
- The concept of a minimum spanning tree.
- Prim's algorithm for minimum spanning trees.
- Kruskal's algorithm for minimum spanning trees.

6.1 Shortest Path in Graphs with Non-negative Weights

Most of the algorithms that we will study for shortest paths hinge on the technique of **relaxation**. For each of these algorithms, we are going to maintain a *distance estimate* for each vertex. That is, we will maintain for each vertex, the length of the shortest path to that vertex that we have found so far. Throughout the algorithms, we will continuously update these estimates and improve them as we find better paths. The notion of *relaxation* simply means to update a current shortest path to an even shorter path found by **enforcing the triangle inequality.** In other words, if we find an edge that violates the triangle inequality, then it means that we have found a path that is shorter than one of our current estimates, so we should update our estimate to the newer, shorter path. Edge relaxation is depicted in Algorithm 37.

Algorithm 37 Edge relaxation

```
1: function RELAX(e = (u, v))
2:     if dist[v] > dist[u] + w(u, v) then
3:         dist[v] = dist[u] + w(u, v)
4:         pred[v] = u
```

The predecessor array, just like in breadth-first search, indicates for each vertex its parent in the shortest path tree from the source. It is maintained in the same way as it was in a breadth-first

search and can be used in the same way to reconstruct the shortest paths once we have found all of the correct distances (see Algorithm 30).

Dijkstra's algorithm for finding shortest paths in graphs with non-negative weights is based off the following key ideas:

Key Ideas: Dijkstra's Algorithm

1. If all edge weights are non-negative, i.e. $w(u,v) \geq 0$ for all edges (u,v), then any sub-path $s \rightsquigarrow u$ of some longer path $s \rightsquigarrow v$ necessarily has a length that is no greater than the entire path. This is not true when negative weights appear, as a path may grow in number of edges but decrease in total length due to the negative weights.
2. This means that if we know the minimum distance to all vertices $u \in S$ for some set S and we relax all edges that leave S, then we can guarantee that the distance estimate to the vertex $v \notin S$ with the minimum estimate $\delta(s,u) + w(u,v)$ is optimal. This is because even though we do not know the distances to any other vertex $v' \notin S$, none of them can provide a shorter path to v since all edge weights are nonnegative and hence the paths can only get longer.
3. Dijkstra's algorithm therefore employs a **greedy** approach, a similar idea to breadth-first search, by visiting nodes in order of distance from the source.

Dijkstra's algorithm works like a weighted version of breadth-first search, i.e. vertices are still visited in order of their distance from the source, but distance is now computed in terms of edge weights, rather than number of edges. The primary difference in implementation is that because the edge weights are no longer all 1, we cannot simply use an ordinary queue to store the vertices that we need to visit, since a path consisting of many low weight edges may have a shorter length than a path of fewer large weight edges (see Figure 6.1 for an example). Instead, Dijkstra's algorithm employs a priority queue to ensure that vertices are visited in the correct order of their distance from the source[1]. An implementation is given in Algorithm 38.

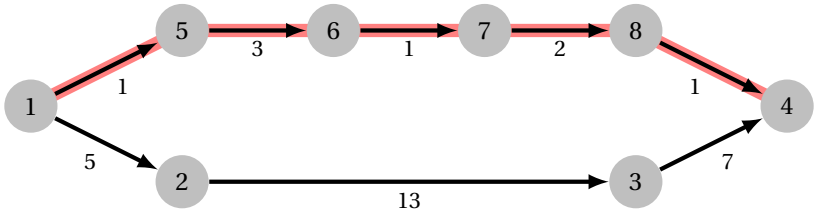


Figure 6.1: A graph where the shortest path from $1 \rightsquigarrow 4$ has far more edges than the path with the fewest edges. In this case, breadth-first search will not find the shortest path.

The priority queue Q is a minimum priority queue initially containing all vertices. It serves each vertex u in order of their current distance estimate, $dist[u]$. At each iteration, the next vertex

[1] Actually, Dijkstra's original implementation of his algorithm did not use a priority queue, but rather simply looped over the vertices to find which one was the next closest. This is very inefficient for sparse graphs however, and was improved by Fredman and Tarjan, who described the now standard min-heap-based priority queue implementation.

Algorithm 38 Dijkstra's algorithm

```
1:  function DIJKSTRA(G = (V, E), s)
2:      dist[1..n] = ∞
3:      pred[1..n] = 0
4:      dist[s] = 0
5:      Q = priority_queue(V[1..n], key(v) = dist[v])
6:      while Q is not empty do
7:          u = Q.pop_min()
8:          for each edge e that goes from u to a node still in the queue do
9:              // Priority queue keys must be updated if relax improves a distance estimate!
10:             RELAX(e)
11:     return dist[1..n], pred[1..n]
```

u that has not been visited yet and which has the smallest current distance estimate is processed. The intuitive idea behind the correctness of Dijkstra's is that since vertices are removed in distance order, once a vertex has been removed from the queue, its distance estimate must be correct and will never be further decreased.

Complexity of Dijkstra's algorithm

The time complexity of Dijkstra's algorithm depends on the way in which we choose to implement the priority queue of vertices. In general, the time taken depends on two things, the amount of time required to find the next minimum (the complexity of pop_min()), and the amount of time required to update the distance estimates when we relax them. If we denote by T_{update} and T_{find}, the time taken to update the distance estimates and to find the minimum respectively, then the total time complexity of Dijkstra's algorithm will be

$$O(|E| \cdot T_{\text{update}} + |V| \cdot T_{\text{find}}).$$

This is due to the fact that we relax every edge exactly once, performing an update operation each time in the worst-case, and we find and extract each vertex exactly once when it is the current minimum. If we implement Dijkstra's in the simplest way possible, using an array and looping over every vertex to find the minimum, then it takes constant $O(1)$ time to update the distances (we just update the dist array and nothing else) and linear $O(|V|)$ time to find the next vertex. Therefore the total running time would be

$$O(|E| + |V| \cdot |V|) = O(|V|^2).$$

For dense graphs, this is great, since this is $O(|E|)$, but this can be improved significantly for sparse graphs by using a min-heap-based priority queue. If we use a standard binary heap data structure, we achieve $O(\log(|V|))$ time per operation for both find and update, and hence the time complexity of Dijkstra's algorithm becomes

$$O(|E|\log(|V|) + |V|\log(|V|)) = O(|E|\log(|V|)),$$

assuming the graph is connected. For dense graphs, this is actually slightly worse than the simple, array-based implementation, but for sparse graphs, this is a huge performance increase. Theoretically better asymptotic complexities can be achieved by using fancier heap data structures such as the Fibonacci heap, which results in a complexity of $O(|E| + |V|\log(|V|))$, but these often turn out to be less efficient in practice than their simpler counterparts.

Correctness of Dijkstra's algorithm

We discussed the intuitive reasoning behind the correctness of Dijkstra's algorithm earlier. Since all edge weights are nonnegative, whenever we visit a closest vertex that hasn't been visited before, we can guarantee that no shorter path exists since any other paths we find can only be longer, since the only way to make a path shorter would be to add a negative weight somewhere. We prove this now formally using induction.

Theorem: Correctness of Dijkstra's algorithm

Given a graph $G = (V, E)$ with non-negative weights and a source vertex s, Dijkstra's algorithm correctly finds shortest paths to each vertex $v \in V$.

Proof

Let's use induction on the set of vertices S that have been removed from the queue, i.e. the set $S = V \setminus Q$. We will show for any vertex $v \in S$, that $dist[v]$ is correct.

For the base case, we will always remove the source vertex s first which has $dist[s] = 0$, which is correct since the graph contains no negative weights.

Suppose for the purpose of induction that at some point it is true that for all $v \in S$, $dist[v]$ is correct. Let u be the next vertex removed from the priority queue, we will show that $dist[u]$ must be correct.

Suppose for contradiction that there exists a shorter path for $s \rightsquigarrow u$ with a shorter distance $\delta(s \rightsquigarrow u) < dist[u]$. Let x be the furthest vertex along the correct $s \rightsquigarrow u$ path that is in S. Since $x \in S$, by the inductive hypothesis, its distance estimate is correct. Let y be the next vertex on the shortest path after x. Since $\delta(s \rightsquigarrow u) < dist[u]$ **and all edge weights are non-negative**, it must be true that $\delta(s \rightsquigarrow y) \leq \delta(s \rightsquigarrow u) < dist[u]$. But y is adjacent to x on a shortest path, which means the edge (x, y) was relaxed when x was removed from Q. This means that $dist[y] = \delta(s \rightsquigarrow y) < dist[u]$. If $dist[y] < dist[u]$ and $y \neq u$, then Dijkstra's algorithm would have popped y from the priority queue instead of u, a contradiction. Alternatively if $y = u$ and $dist[y] < dist[u]$, this is also a contradiction.

By contradiction, we conclude that $dist[u]$ must be correct and hence by induction on the set S, when Dijkstra's algorithm terminates, $dist[v]$ is correct for all $v \in V$.

Notice how the key step of the proof requires us to invoke the fact that the edge weights are non-negative. Without this restriction, this step of the proof would not be true, since the subpath $s \rightsquigarrow y$ might actually have a higher total weight than the total path $s \rightsquigarrow u$ due to negative weights later on.

Practical considerations: Implementing Dijkstra's algorithm

When implementing Dijkstra's algorithm, careful thought needs to be put into the way in which the priority queue is going to be implemented. The most common description of the priority queue used for Dijkstra's algorithm is a min-heap that supports the following operations:

1. Insert the initial items into the priority queue.

2. Remove the item with the minimum key.
3. Decrease the key of an item already in the priority queue (whenever an edge is relaxed).

In practice however, most programming language's standard library priority queues do not support operation 3, and implementing this operation ourselves can be annoying and tricky. An alternate formulation of Dijkstra's algorithm which does not require operation 3 is therefore commonly used instead. It works like this:

1. Begin with only the source vertex s in the priority queue.
2. Whenever an edge is relaxed, insert the target vertex in the priority queue keyed by its current distance estimate.
3. When a key is removed from the priority queue, check whether it is out of date, and if so, ignore it. A key is out of date if $\text{key}(u) > dist[u]$.

The main idea here is that instead of updating the keys of vertices that are in the priority queue, we simply allow it to contain multiple entries for the same vertex with different distance estimates. If a vertex u that has already been processed is popped from the queue, the key will be greater than $dist[u]$ and hence we can ignore it since it is an out-of-date entry. This does not hurt the asymptotic complexity of the algorithm, since the only difference encountered is that the priority queue may grow to a worst-case size of $|E|$ instead of $|V|$, leading to operations that cost $O(\log(|E|))$ instead of $O(\log(|V|))$. But since in a simple graph

$$O(\log(|E|)) = O(\log(|V|^2)) = O(2\log(|V|)) = O(\log(|V|)),$$

this is asymptotically equivalent (and we can always preprocess a graph to make it simple by taking just the shortest edge between every pair of vertices). In practice, the number of entries in the priority queue at any given time is likely to be much smaller than $|E|$, so this method is almost always a speed-up too, except in pathological cases. An example implementation is depicted in Algorithm 39.

Algorithm 39 Improved Dijkstra's algorithm

```
1:  function DIJKSTRA(G = (V, E), s)
2:      dist[1..n] = ∞
3:      pred[1..n] = 0
4:      dist[s] = 0
5:      Q = priority_queue()
6:      Q.push(s, key = 0)
7:      while Q is not empty do
8:          u, key = Q.pop_min()
9:          if dist[u] = key then                    // Do not process an out-of-date entry
10:             for each edge e that is adjacent to u do
11:                 if dist[v] > dist[u] + w(u, v) then
12:                     dist[v] = dist[u] + w(u, v)
13:                     pred[v] = u
14:                     Q.push(v, key = dist[v])
15:     return dist[1..n], pred[1..n]
```

6.2 Minimum Spanning Trees

The minimum spanning tree problem is informally, the problem of finding the cheapest way to connect a set of nodes in a network. Minimum spanning trees arise in applications such as connecting cables in a network, where several locations need to be connected together at a minimum possible cost.

A spanning tree of a graph $G = (V, E)$ is a subgraph T consisting of all of the same vertices V and some subset $E' \subset E$ of the edges such that T is a tree. Informally, it is a subset of the edges in E that connect all of the vertices in V. Since spanning trees are trees, they are acyclic (they do not contain cycles). A spanning tree of an edge-weighted graph is called a weighted spanning tree, and a weighted spanning tree with minimum possible total weight is called a minimum spanning tree. In other words, a minimum spanning tree is a connected subgraph that minimises the total weight

$$w(T) = \sum_{(u,v) \in E(T)} w(u, v).$$

Although the definition of a minimum spanning tree is a global one, that is we are required to minimise the total weight of the tree, minimum spanning trees have local structure which allows us to find them using greedy algorithms. We will see two such greedy algorithms, Prim's algorithm which is almost identical to Dijkstra's algorithm, and Kruskal's algorithm which makes use of the union-find disjoint-set data structure. For both algorithms presented below, we assume that the graph G is connected (otherwise no spanning trees exist).

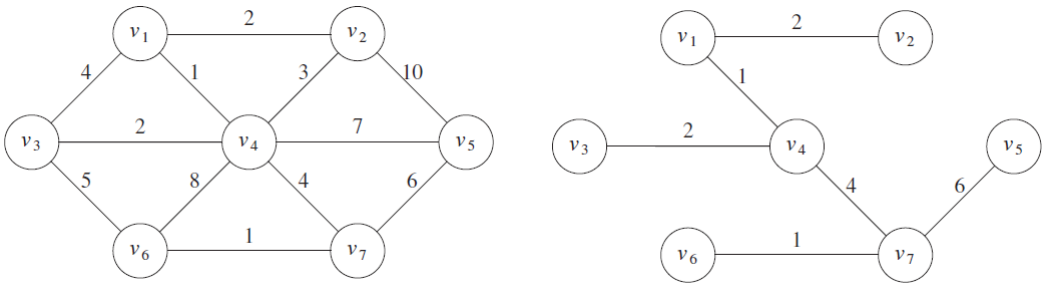


Figure 6.2: A weighted graph G and a minimum spanning tree of G. Image source: Weiss

6.2.1 Prim's Algorithm

Prim's algorithm for minimum spanning trees is very similar to Dijkstra's algorithm for single-source shortest paths. Prim's algorithms begins with an arbitrarily chosen source vertex and builds a tree T, adding edges one at a time to form a spanning tree. The edge to add at each iteration is chosen such that it is a lightest weight edge that connects the current partial spanning tree T to some new vertex v not yet in the tree. To contrast, while Dijkstra's algorithm forms a shortest path tree by selecting a vertex with the smallest total distance from the source, Prim's selects a vertex with minimum possible distance to T. An implementation is given in Algorithm 40. Observe the similarity to Dijkstra's algorithm. Note that we may choose the root node r to be any node in the graph.

Algorithm 40 Prim's algorithm

```
1:  function PRIM(G = (V, E), r)
2:      dist[1..n] = ∞
3:      parent[1..n] = null
4:      T = ({r}, ∅)
5:      dist[r] = 0
6:      Q = priority_queue(V[1..n], key(v) = dist[v])
7:      while Q is not empty do
8:          u = Q.pop_min()
9:          T.add_vertex(u)
10:         T.add_edge(parent[u], u)
11:         for each edge e = (u, v) adjacent to u do
12:             if not v ∈ T and dist[v] > w(u, v) then
13:                 // Remember to update the key of v in the priority queue!
14:                 dist[v] = w(u, v)
15:                 parent[v] = u
16:     return T
```

Complexity of Prim's algorithm

Compare the code for Prim's to that of Dijkstra's algorithm. The main difference is that the key used to greedily select the next vertex v is taken to be the weight of the lightest edge $w(u,v)$ connecting v to the current tree, rather than the total distance from v to the source node. Otherwise, the two algorithms are almost identical. Like Dijkstra's algorithm, the complexity of Prim's algorithm using a binary heap to implement the priority queue is therefore $O(|E|\log(|V|))$.

Correctness of Prim's algorithm

We prove the correctness of Prim's algorithm by describing an invariant that it maintains.

Theorem: Invariant of Prim's algorithm

In every iteration of Prim's algorithm, the current set of selected edges in T is a subset of some minimum spanning tree of G.

Proof

Initially T is empty, so T is a subset of some MST of G. Suppose that at some iteration, T is a subset of some MST. Denote this MST by M. Let $e = (u,v)$ denote the lightest edge that connects some $u \in T$ to some $v \notin T$. If M contains e then $T \cup \{e\}$ is a subset of some MST, so the invariant holds. Otherwise, suppose that M does not contain e. We need to prove that there exists some other MST such that $T \cup \{e\}$ is a subset of it.

Since M is a tree, is contains a path p from $u \rightsquigarrow v$. Since u and v are not connected in T, at least one of the edges on the path p is not contained in T. Let (x,y) be the first edge on the path p from u to v that is not contained in T. Since (x,y) is the first edge not in T, and T is a subset of M, $x \in T$ and $y \notin T$.

Since M is a tree, removing the edge (x, y) disconnects M into two components M_1 and M_2. Connecting the edge (u, v) to M_1 and M_2 reconnects them to form a new spanning tree $M' = M_1 \cup M_2 \cup \{(u, v)\} = M \cup \{(u, v)\} \setminus \{(x, y)\}$. Since (u, v) is the lightest edge connecting T to a new vertex, it is no heavier than (x, y), otherwise Prim's would have picked (x, y) instead since $x \in T$ and $y \notin T$. In other words, $w(u, v) \leq w(x, y)$, which implies that $w(u, v) - w(x, y) \leq 0$. So the total weight of M' is

$$\begin{aligned} w(M') &= w(M) + w(u, v) - w(x, y) \\ &\leq w(M) \end{aligned}$$

But since M is a minimum spanning tree $w(M) \leq w(M')$ and hence $w(M) = w(M')$, from which we conclude that M' is also a minimum spanning tree. Since $T \cup \{e\}$ is a subset of M', we have shown that the invariant is maintained.

Corollary: Correctness of Prim's algorithm

Prim's algorithm correctly produces a minimum spanning tree.

Proof

First, we argue that Prim's algorithm produces a spanning tree. Whenever Prim's adds an edge to the MST, it does so to a newly added vertex, which means it can never create a cycle. Since the graph is connected, Prim's will visit every vertex and hence it will produce a spanning tree. Therefore when the algorithm terminates, T is a spanning tree, and by the invariant above, it is a subset of some minimum spanning tree. Since all spanning trees contain the same number of edges, T itself must be a minimum spanning tree. Hence Prim's algorithm is correct.

6.2.2 Kruskal's Algorithm

Kruskal's algorithm like Prim's also greedily selects edges, but using a different strategy. Instead of building a single tree and growing it larger each iteration, Kruskal's algorithm instead maintains a forest F of minimum weight subtrees. At each iteration, Kruskal's algorithm selects the lightest weight edge that connects two currently disconnected subtrees. In order words, Kruskal's selects the minimum weight edge that does not induce a cycle in the current forest. In order to quickly check whether or not an edge will connect two currently disconnected vertices, we utilise the union-find disjoint sets data structure. See the implementation in Algorithm 41.

Complexity of Kruskal's algorithm

Sorting the edges of the graph takes $O(|E|\log(|E|))$ time, and the sequence of UNION and FIND operations takes $O(|E|\log(|V|))$. The total time complexity of Kruskal's algorithm is therefore $O(|E|\log(|E|))$, or equivalently $O(|E|\log(|V|))$ since $\log(E) = O(\log(|V|^2)) = O(|\log(|V|))$ in a simple graph.

Algorithm 41 Kruskal's algorithm

```
1: function KRUSKAL(G = (V, E))
2:     sort(E, key((u, v)) = w(u, v))          // Sort edges in ascending order of weight
3:     forest = UnionFind.initialise(n)
4:     T = (V, ∅)
5:     for each edge (u, v) in E do
6:         if forest.FIND(u) ≠ forest.FIND(v) then      // Ignore edges that would create a cycle
7:             forest.UNION(u, v)
8:             T.add_edge(u, v)
9:     return T
```

Correctness of Kruskal's algorithm

We prove the correctness of Kruskal's algorithm by using an invariant. In fact, it is the same invariant as maintained by Prim's, and the proof is almost identical.

Theorem: Invariant of Kruskal's algorithm

Every iteration of Kruskal's algorithm, the current set of selected edges T is a subset of some minimum spanning tree of G.

Proof

Initially T is empty, so T is a subset of some MST of G. Suppose that at some iteration, T is a subset of some MST. Denote this MST by M. Let $e = (u, v)$ denote the lightest edge that connects two vertices that are not yet connected in T. If M contains e then $T \cup \{e\}$ is a subset of some MST, so the invariant holds. Otherwise, suppose that M does not contain e. We need to prove that there exists some other MST such that $T \cup \{e\}$ is a subset of it.

Since M is a tree, it contains a path p from $u \rightsquigarrow v$. Since u and v are not connected in T, at least one of the edges on the path p is not contained in T. Let (x, y) be any edge on the path from u to v that is not contained in T. Since T is a subset of M, the vertices x and y cannot be connected in T or M would contain a cycle.

Since M is a tree, removing the edge (x, y) disconnects M into two components M_1 and M_2. Connecting the edge (u, v) to M_1 and M_2 reconnects them to form a new spanning tree $M' = M_1 \cup M_2 \cup \{(u, v)\} = M \cup \{(u, v)\} \setminus \{(x, y)\}$. Since (u, v) is the lightest edge connecting two disconnected vertices in T, it is no heavier than (x, y), otherwise Kruskal's would have picked (x, y) instead since x and y are not connected in T. In other words, $w(u, v) \leq w(x, y)$, which implies that $w(u, v) - w(x, y) \leq 0$. So the total weight of M' is

$$\begin{aligned} w(M') &= w(M) + w(u, v) - w(x, y) \\ &\leq w(M) \end{aligned}$$

But since M is a minimum spanning tree $w(M) \leq w(M')$ and hence $w(M) = w(M')$, from which we conclude that M' is also a minimum spanning tree. Since $T \cup \{e\}$ is a subset of M', we have shown that the invariant is maintained.

Corollary: Correctness of Kruskal's algorithm

Kruskal's algorithm correctly produces a minimum spanning tree.

Proof

First, we argue that Kruskal's algorithm produces a spanning tree. T will never contain a cycle since the use of the union-find data structure explicitly prevents it. T must also be connected since G is connected, and if there were two components that were not connected in T, Kruskal's would select the edge connecting them. Therefore when the algorithm terminates, T is a spanning tree, and by the invariant above, it is a subset of some minimum spanning tree. Since all spanning trees contain the same number of edges, T itself must be a minimum spanning tree. Hence Kruskal's algorithm is correct.

Chapter 7

Dynamic Programming

A powerful technique that we have already encountered for solving problems is to reduce a problem into smaller problems, and to then combine the solutions to those smaller problems into a solution to the larger problem. We have seen this for example in the recursive computation of Fibonacci numbers, and in the divide-and-conquer algorithms for sorting (Merge Sort and Quicksort). In some situations, the smaller problems that we have to solve might have to be solved multiple times, perhaps even exponentially many times (for example, recursive computation of the Fibonacci numbers). Dynamic programming involves employing a technique called *memoisation*, where we store the solutions to previously computed subproblems in order to ensure that repeated problems are solved only once. Dynamic programming very frequently arises in optimisation problems (minimise or maximise some quantity) and counting problems (count how many ways we can do a particular thing).

Summary: Dynamic Programming

In this chapter, we cover:

- The key ideas behind dynamic programming.
- Memoisation applied to computing Fibonacci numbers.
- The coin change problem.
- The top-down approach vs. the bottom-up approach.
- Reconstructing solutions to optimisation problems.
- The unbounded knapsack problem.
- The 0-1 knapsack problem.
- Optimal matrix multiplication.
- The edit distance problem.
- The space-saving trick.

7.1 The Key Elements of Dynamic Programming

Memoisation - Computing Fibonacci numbers

The essence of dynamic programming is to solve a problem by breaking it into smaller subproblems and combining the solutions of those subproblems into a solution to the greater problem. In contrast with the divide-and-conquer method which also employs the same idea, dynamic

programming applies when the subproblems **overlap** and are repeated multiple times. Recall the example of computing Fibonacci numbers recursively. Fibonacci numbers are defined by recurrence relation

$$F(n) = F(n-1) + F(n-2), \qquad F(0) = 0, \quad F(1) = 1,$$

and can be computed recursively with a very simple algorithm shown in Algorithm 42.

Algorithm 42 Recursive Fibonacci numbers

```
1: function FIBONACCI(n)
2:     if n ≤ 1 then
3:         return n
4:     else
5:         return FIBONACCI(n-1) + FIBONACCI(n-2)
```

Although simple, this algorithm is extremely inefficient. What is its time complexity? Well, the amount of work taken $T(n)$ to compute the value of $F(n)$ is the sum of the times taken to compute the values of $T(n-1)$ and $T(n-2)$, so the time complexity is given by the solution to the recurrence

$$T(n) = T(n-1) + T(n-2) + c.$$

Hopefully we recognise the main piece of this recurrence, its just the Fibonacci recurrence with an added constant! The time complexity to compute $F(n)$ can therefore be shown to be $O(F(n))$. As cool as this fact is, hopefully you remember from your mathematics study, that the Fibonacci numbers grow exponentially, so this is an extremely slow algorithm. Specifically they grow asymptotically like φ^n where φ is the golden ratio. How does it get so bad? What exactly are we doing so wrong?

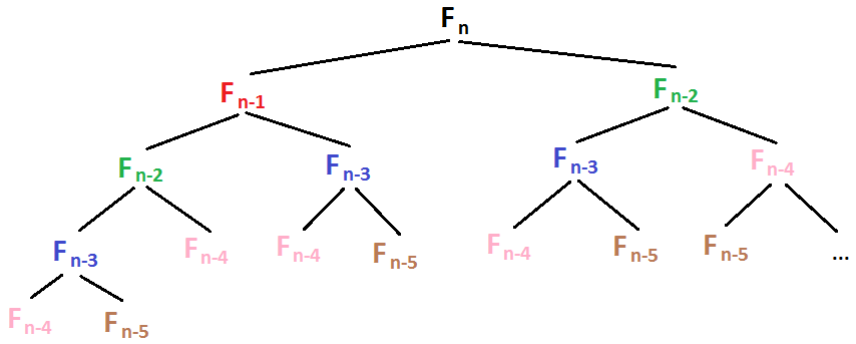


Figure 7.1: The call tree resulting from the recursive Fibonacci function for computing F_n.

Figure 7.1 depicts what happens when we try to compute $F(n)$. $F(n)$ and $F(n-1)$ are computed once, $F(n-2)$ is computed twice, $F(n-3)$ is computed three times, $F(n-4)$ is computed five times... In general, $F(n-k)$ is computed $F(k+1)$ times, so the leaves of the call tree are computed exponentially many times, which explains why the algorithm is so slow. The solution to this problem is simple but demonstrates the most powerful, core idea of dynamic programming. Instead of allowing ourselves to compute the same thing over and over again, we

will simply store the result of each computation in a table, and if we are required to compute it again, we can look it up and return it straight from the table without any recomputation. This technique is called **memoisation** (no, that is not a typo, it is actually spelled and pronounced *memo-isation*). A memoised implementation of the recursive Fibonacci number calculation is shown in Algorithm 43.

Key Ideas: Memoisation of overlapping subproblems

Memoisation is the process of remembering the solutions to previously computed subproblems and storing them in a table for later lookup. If the same subproblem needs to be computed multiple times, it need only be computed once, and then can be subsequently looked up from the memo table at no additional cost.

Dynamic programming solves problems that exhibit overlapping or repeated subproblems by using memoisation to avoid doing redundant work.

Algorithm 43 Memoised Fibonacci numbers

```
1: function FIBONACCI_MEMO(n)
2:     if n ≤ 1 then return n                          // The base case
3:     if memo[n] = null then                          // Compute it for the first time
4:         memo[n] = FIBONACCI_MEMO(n−1) + FIBONACCI_MEMO(n−2)
5:     return memo[n]
6:
7: function FIBONACCI(n)
8:     memo[0..n] = null
9:     return FIBONACCI_MEMO(n)
```

We store the Fibonacci numbers that have been computed in the memoisation table *memo*, and return straight from the table whenever we encounter a problem that we already have the solution to. The initial value of **null** in the table is used to indicate that we have not computed that number yet. If we were planning on computing lots of Fibonacci numbers, we could even keep the memo table between calls. Since each call to FIBONACCI_MEMO takes constant time, each problem is never computed more than once, and there are n subproblems, the total time complexity of this implementation is $O(n)$.

The approach that we have implemented above is an example of "top-down" dynamic programming since we first ask for the solution to the largest problem and recursively ask for the solutions to smaller problems while filling in the memoisation table. An alternative implementation would be to start from the smaller problems and work our way up to the big problems. This would be the "bottom-up" approach. A bottom-up version of computing Fibonacci numbers is shown in Algorithm 44.

We will explore the differences between the top-down and bottom-up approaches soon.

Algorithm 44 Bottom-up Fibonacci numbers

```
1: function FIBONACCI(n)
2:     fib[0..n] = 0
3:     fib[1] = 1
4:     for i = 2 to n do
5:         fib[i] = fib[i−1] + fib[i−2]
6:     return fib[n]
```

Optimal substructure - The coin change problem

A classic and very illustrative example of dynamic programming is the coin change problem. Suppose we have currency with denominations $c_1, c_2, \ldots c_n$ and we wish to make exactly $\$V$. There might be many ways to make this dollar amount. The coin change problem is to find the one that uses the fewest coins possible. For example, if the denominations are $\$1, \$5, \$6, \$9$ then the minimum amount of coins required to make \$13 is three, using two \$6 coins and one \$1 coin.

Identifying the subproblems

In order for dynamic programming to be applicable, the problem must exhibit *optimal substructure*. This means that we can break the problem down into subproblems such that the subproblems can be recombined to form a solution to the original problem. In contrast with divide and conquer, the subproblems of a dynamic programming solution are repeated or overlapping, that is, their values are required many times (for example, computation of Fibonacci numbers). One often useful way to think about a dynamic programming problem is to try to break the problem up into a sequence of choices which lead us between subproblems. For example, in the coin change problem, our choices are the coins that we select. Every time we select a coin, our subproblem reduces in value by the dollar amount of the coin we just selected.

Key Ideas: Optimal substructure

A combinatorial problem exhibits optimal substructure if it can be broken down into smaller subproblems, such that the solutions to the subproblems can be combined to obtain a solution to the original problem.

Optimal substructure is not only useful for DP algorithms. Divide and conquer algorithms also use optimal substructure (for example, Quicksort), but do not rely on subproblems being repeated.

If we need to make $\$V$ and we choose to use a coin with value $\$x$, then we are now required to make change for $\$(V-x)$. The essence of dynamic programming is then to try every possible choice and take the best one, while memoising the answers to avoid repeated computation. Using this idea, the subproblems for coin change can be expressed as

$$\text{MinCoins[v]} = \{\text{The fewest coins required to make exactly \$v}\},$$

for all values of v in $0 \leq v \leq V$. These subproblems exhibit optimal substructure since if the optimal solution for $\$V$ includes a coin of value x, then the remaining coins must necessarily

be optimal for the value \$($V - x$).

Deriving the recurrence

Now that we have our subproblems, we write a recurrence relation that expresses the optimal substructure and encompasses all of the possible choices we could make, taking the best one. If we use a coin of value \$$c_i$ when trying to make \$$v$, then we must recursively try to make \$($v - c_i$). The following recurrence tries all possible coins and selects the best option.

$$\text{MinCoins[v]} = \begin{cases} 0 & \text{if } v = 0, \\ \infty & \text{if } v > 0 \text{ and } v < c[i] \text{ for all } i, \\ \min\limits_{\substack{1 \le i \le n \\ c[i] \le v}} (1 + \text{MinCoins[v - c[i]]}) & \text{otherwise} \end{cases}$$

We include the base case MinCoins[0] = 0 since the recurrence must terminate, and it takes zero coins to make zero dollars. We also write the infeasible case (returning ∞) in case it is not possible to make some values, since there might not necessarily be a \$1 coin.

Implementing a solution

We can implement a top-down solution to coin change by implementing the recurrence relation above as a recursive function. Your implementation might look like the pseudocode shown in Algorithm 45. Note that we use memoisation to avoid computing a solution to a subproblem twice! Assume that the memo table is appropriately initialised to **null**.

Algorithm 45 Top-down coin change

```
1: function COIN_CHANGE(c[1..n], v)
2:     if v = 0 then return 0
3:     if memo[v] = null then
4:         min_coins = ∞
5:         for i = 1 to n do
6:             if c[i] ≤ v then
7:                 min_coins = min(min_coins, 1 + COIN_CHANGE(v − c[i]))
8:         memo[v] = min_coins
9:     return memo[v]
```

Since we solve $V + 1$ subproblems and each one tries all n coins, the time complexity of this solution is $O(nV)$. The space complexity of storing the solutions to V subproblems is $O(V)$. Just as we did for Fibonacci numbers, we could also convert this into a bottom-up solution. Pseudocode for a bottom-up coin change algorithm is given in Algorithm 46.

7.2 Top-down vs. Bottom-up Dynamic Programming

Both styles of dynamic programming, top down and bottom up involve using a table to remember the solutions to previously computed subproblems. Their behaviour is quite different however. In the top-down approach, the memoisation table is filled on demand as the particular subproblems that are required are computed. In the bottom-up approach, we fill the solution

Algorithm 46 Bottom-up coin change

```
1: function COIN_CHANGE(c[1..n], V)
2:     MinCoins[0..V] = ∞
3:     MinCoins[0] = 0
4:     for v = 1 to V do
5:         for i = 1 to n do
6:             if c[i] ≤ v then
7:                 MinCoins[v] = min(MinCoins[v], 1 + MinCoins[v − c[i]])
8:     return MinCoins[V]
```

table one problem at a time in an order which ensures that any dependent subproblems have already been computed before they are needed. The order in which we fill the table in the bottom-up approach is the *reverse topological order* of the dependency graph of the subproblems. Most of the time, figuring out the order in which the subproblems must be computed is simple, but in rare cases it might be more difficult.

The majority of the time, the two dynamic programming styles are equivalent and equally as good, but there are certain situations in which it may be preferable to favour one over the other. As eluded to above, one reason that the top-down approach might be preferable in some cases is that we do not need to know a-priori the order in which the subproblems need to be computed, as the recursion takes care of ensuring that subproblems are made available when required. The top-down approach also boasts the advantage that subproblems are only computed if they are actually required, while the bottom-up approach strictly computes the solution to every possible subproblem. If the solutions to only a small number of the possible subproblems are required, then the top-down approach will avoid computing the ones that are not needed.

On the contrary, the bottom-up approach avoids recursion altogether and hence should be faster if subproblems are frequently revisited since we drop the overhead on the program's call stack required to make the recursive calls. The bottom-up approach also allows us to more easily exploit the structure of specific dynamic programs in situations where we can improve the time and space complexity of the resulting program. We will see one such example, the space-saving trick, which allows us to reduce the space complexity of some dynamic programs, which is not possible with the top-down approach. The following table summarises the pros and cons of both approaches.

Pros: Top-down Approach	**Pros: Bottom-up Approach**
1. No need to know the topological ordering of the subproblem dependencies. 2. Avoids computing the solution to subproblems that are not needed. 3. More intuitive for programmers who like to think recursively.	1. Avoids recursion, which is typically slower than iteration. 2. Allows for clever optimisations (e.g. space-saving trick in the next section). 3. Might be more intuitive for programmers who don't like recursion.

7.3 Reconstructing Optimal Solutions to Optimisation Problems

When we solved the coin change problem, we left out something rather important. Although we computed the fewest number of coins required, we did not actually produce a list of those coins. We can extend our dynamic programming algorithm to also produce an optimal list of coins. In general, there are two ways to approach constructing an optimal solution.

Method 1: Backtracking through the subproblems

The first way to reconstruct solutions to optimisation problems is to *backtrack* through the subproblems and figure out which choices were made that lead to the given solution. For example, in the coin change example given earlier, the fewest coins needed to make $13 was three, so we could backtrack through the entries of the dynamic programming table over values of i = 1 to n until we found a coin c_i such that MinCoins[13 - c_i] = 2. When we find such a coin, the optimal substructure of the problem tells us that this coin is optimal. We then subtract c_i from 13 and repeat until we reach $V = 0$ at which point we know we have constructed the entire solution. An example implementation using the bottom-up approach is shown in Algorithm 47.

Algorithm 47 Coin change solution reconstruction using backtracking

```
1: function GET_COINS(c[1..n], V, MinCoins[0..V])
2:     if MinCoins[V] = ∞ then return error
3:     coins = empty list
4:     while V > 0 do
5:         for i = 1 to n do
6:             if c[i] ≤ V and MinCoins[V] = 1 + MinCoins[V − c[i]] then      // c[i] is optimal
7:                 coins.append(c[i])
8:                 V = V − c[i]
9:     return coins
```

Method 2: Using a decision table

The second approach often used for reconstructing solutions is to keep a second table in addition to the memoisation or bottom-up dynamic programming table which is used to remember the optimal decisions that were made by each subproblem. Once the optimal value is found, the solution can then be reconstructed by looking up the choices that were made in this table. An implementation of coin change using this approach is shown in Algorithm 48. In this implementation, we store an array OptCoin in addition to the MinCoins array, which remembers for each value of v, a coin that would lead to the optimal solution.

Both approaches are very similar. The first is advantaged by the fact that it uses less memory, since a second table is not required to remember the optimal coins. It should also usually be faster as a consequence. The second approach however is simpler to understand, and also allows us to do some more advanced optimisations that involve dynamically adjusting the search space for a particular subproblem based on the optimal choices of previous subproblems.

Algorithm 48 Bottom-up coin change with solution reconstruction using decision table

```
1:  function COIN_CHANGE(c[1..n], V)
2:      MinCoins[0..V] = ∞
3:      OptCoin[0..V] = null
4:      MinCoins[0] = 0
5:      for v = 1 to V do
6:          for i = 1 to n do
7:              if c[i] ≤ v then
8:                  if 1 + MinCoins[v − c[i]] < MinCoins[v] then
9:                      MinCoins[v] = 1 + MinCoins[v − c[i]]
10:                     OptCoin[v] = c[i]                    // Remember the best coin
11:     return MinCoins[v]
12:
13: function GET_COINS(c[1..n], V, MinCoins[0..V], OptCoin[0..V])
14:     if MinCoins[V] = ∞ then return error
15:     coins = empty list
16:     while V > 0 do
17:         coins.append(OptCoin[V])                         // Take the best coin
18:         V = V - OptCoin[V]
19:     return coins
```

7.4 The Unbounded Knapsack Problem

Suppose you find yourself in a room full of n heavy but valuable kinds of items! You have with you a backpack that you can use to carry up to C kilograms of items. You have figured out for each kind of item i how valuable it is v_i and how much it weighs w_i. Which items should you take in order to maximise the total value in your backpack? We can consider two different versions of this problem. In the first one, we can take as many of each kind of item as we want. This is called the *unbounded* knapsack problem. In the second version, which we will cover next, we can only take one of each kind.

We can solve the unbounded knapsack problem using dynamic programming. Let's walk through the process.

Identifying the optimal subproblems

The unbounded knapsack problem is actually rather similar to the coin change problem. In coin change, we were selecting coins such that made exactly $\$V$ using the fewest coins possible. Now we are selecting items that weigh at most Ckg that have the highest value possible. Both problems have a capacity constraint, although unlike coin change, for the knapsack problem it is not required that we leave no empty space. Both of them are also optimising an objective based on the items selected, except that for coin change we are minimising and for knapsack we are maximising. The decisions that we can make in the knapsack problem are the items that we select, analogous to the coins that we select for the coin change problem.

[1] Taken from `https://en.wikipedia.org/wiki/File:Knapsack.svg` by Dake. Shared under a creative commons licence.

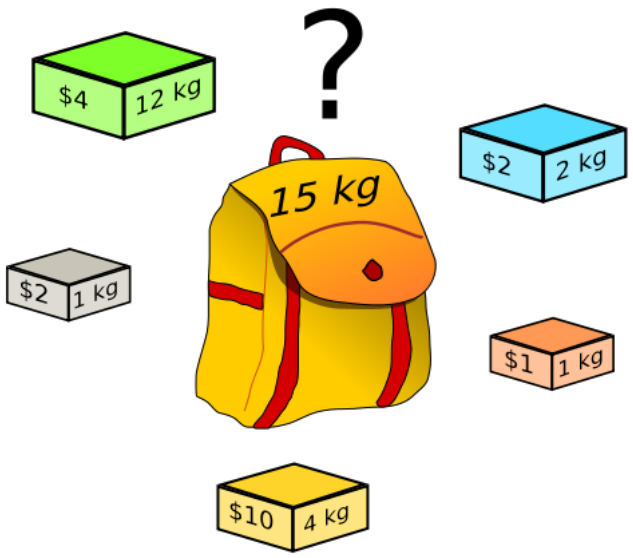


Figure 7.2: An example instance of a knapsack problem. The optimal solution is to take all but the green box for a total value of $15. Image taken from Wikimedia Commons[1].

Suppose that we are trying to fill a bag with capacity Ckg. If we choose an item with weight $x$kg, we are then required to fill the remaining $(C-x)$kg of space with the most valuable items. This substructure is optimal since we always want to use the remaining $(C-x)$kg optimally. This suggests that our subproblems should be the following:

$$\text{MaxValue}[c] = \{\text{The maximum value that we can fit in a capacity of } c\}$$

for values of c such that $0 \leq c \leq C$.

Deriving a recurrence

Using the substructure that we derived above, we can write a recurrence for the unbounded knapsack problem like so:

$$\text{MaxValue}[c] = \begin{cases} 0 & \text{if } c < w_i \text{ for all } i, \\ \max\limits_{\substack{1 \leq i \leq n \\ w_i \leq c}} (v_i + \text{MaxValue}[c - w_i]) & \text{otherwise.} \end{cases}$$

Notice that we do not need an infeasible case like we did for coin change since we are not requiring that the knapsack be completely full.

Implementation of the unbounded knapsack problem

A bottom-up implementation of the unbounded knapsack dynamic program is shown in Algorithm 49. Note how extremely similar it ends up being to the bottom-up solution for coin change.

Algorithm 49 Bottom-up unbounded knapsack

```
1: function UNBOUNDED_KNAPSACK(v[1..n], w[1..n], C)
2:     MaxValue[0..C] = 0
3:     for c = 1 to C do
4:         for i = 1 to n do
5:             if w[i] ≤ c then
6:                 MaxValue[c] = max(MaxValue[c], v[i] + MaxValue[c − w[i]])
7:     return MaxValue[C]
```

7.5 The 0-1 Knapsack Problem

We now consider a variant of the knapsack problem in which we are only allowed to take one of each kind of item. This problem is known in general as the 0-1 knapsack problem (0-1 because we can either take 0 of the item or 1 of it) One method to solve the knapsack problem is to simply try all possibilities and see which one is the best feasible solution. If we have n items to choose from, then we would have to try all possible subsets of n items. The total number of subsets of a set of n items is 2^n, so trying all of them would lead to an exponential time complexity algorithm! We can use dynamic programming to derive a much more efficient and elegant solution to the knapsack problem.

Identifying the optimal subproblems

Consider an instance of the knapsack problem, for example, the one shown in Figure 7.2. The optimal solution for the five items shown given a backpack that carry a total weight of 15kg is to take the blue, silver, yellow and orange items for a total of $15 value. The optimal substructure for 0-1 Knapsack is almost identical to that of the unbounded version, but we need to account for the fact that an item can not be used twice.

Suppose we try the blue item. Since it takes up precisely 2kg of the 15kg backpack, there is 13kg of space left for other items. We are then left with solving the subproblem of finding the most value that we can accrue using 13kg of spacing **using the remaining items**. This is the optimal substructure of the 0-1 knapsack problem. Note the emphasis highlighting the difference compared to the unbounded version. In the 0-1 case, we need to never consider an item again if it has already been used. One way to do this would be to keep track of the subset of items that have been used so far, but this does not improve on the naive solution since we would need to account for 2^n different subsets, which is far too large for moderate values of n. Instead, the key observation to make is that the order in which we consider items is completely irrelevant since taking the 2kg item followed by the 4kg item is the same as taking the 4kg item followed by taking the 2kg item. Therefore instead of keeping track of all subsets, we will just remember that we have taken some items from the first i of them, for all values of i from $1 \leq i \leq n$.

$$\text{MaxValue}[i, c] = \{\text{The maximum value that we can fit in a capacity of } c \text{ using items 1 to } i\}$$

for all $0 \leq c \leq C$ and $0 \leq i \leq n$. This way, if we choose to take item i, then we know that we can subsequently only consider items 1 to $i-1$. Note that by $i = 0$, we denote the empty set of items to give us a simple base case.

Deriving a recurrence

Consider an instance of the knapsack problem consisting of n items of weight w_i and value v_i for $1 \leq i \leq n$ and a backpack of capacity C kg. Since we are avoiding duplicate items, we should not try using every item in the backpack, but rather, in a subproblem considering items 1 to i, we should simply try taking item i or not taking item i. This gives us two options, both of which lead to recursively trying to fill the knapsack with the remaining items 1 to $i-1$.

$$\text{MaxValue}[i,c] = \begin{cases} 0 & \text{if } i = 0, \\ \text{MaxValue}[i-1,c] & \text{if } w_i > c, \\ \max(\text{MaxValue}[i-1,c], v_i + \text{MaxValue}[i-1,c-w_i]) & \text{otherwise.} \end{cases}$$

In the second case of the recurrence, we account for the fact that item i may not fit in the knapsack, in which case there is no decision to make, we simply can not take that item and should consider the remaining $i-1$ items.

Implementation of the 0-1 knapsack problem

Here is a bottom-up implementation of the dynamic programming algorithm corresponding to the recurrence derived above. Each row of the recurrence depends only on the previous row, so we will solve the subproblems in row order. See Algorithm 50. Since there are $n \times C$ subproblems and solving each one requires looking at no more than two previous subproblems, the time and space complexities of this dynamic programming algorithm are $O(nC)$.

Algorithm 50 Bottom-up 0-1 knapsack

```
1: function KNAPSACK(w[1..n], v[1..n], C)
2:     MaxValue[0..n][0..C] = 0
3:     for i = 1 to n do
4:         for c = 1 to C do
5:             if w[i] ≤ c then
6:                 MaxValue[i][c] = max(MaxValue[i−1][c], v[i] + MaxValue[i−1][c − w[i]])
7:             else
8:                 MaxValue[i][c] = MaxValue[i−1][c]
9:     return MaxValue[n][C]
```

7.6 The Edit Distance Problem

The edit distance problem is a way to formally classify how similar or dissimilar two strings of text are. For example, the words "computer" and "commuter" are similar, as we can change one into the other by just modifying one letter, 'p' to 'm.' The words "sport" and "sort" are similar, as one can be changed into the other by deleting one letter from the first (or equivalently, inserting a new letter into the second).

This notion of the number of "edits" required to transform one string into another form the basis of the edit distance metric. A single edit operation consists of one of the following three operations:

1. Insert a new symbol anywhere in the string.
2. Delete one of the symbols from the string.
3. Replace one of the symbols in the string with any other symbol.

Definition: Edit Distance

The edit distance between two strings is the minimum number of edit operations required to convert one of the strings into the other.

We can also define an optimal alignment of two strings which shows where the optimal number of insertions, deletions and substitutions occur. An optimal alignment for "computer" and "commuter" would be

c	o	m	**p**	u	t	e	r
c	o	m	**m**	u	t	e	r
			*				

which contains one edit operation (substitute 'p' for 'm.') An optimal alignment for the words "kitten" and "sitting" is given by

k	i	t	t	**e**	n	-
s	i	t	t	**i**	n	**g**
*				*		*

which contains two substitutions and one insertion, so the edit distance between them is 3.

Identifying the optimal subproblems

Consider the problem of finding an optimal alignment for the strings "ACAATCC" and "AGCATCG." Considering just the final column of the alignment, we have three choices:

C		C		-
G	or	-	or	G
*		*		*

The first choice corresponds to substituting the last 'C' for 'G.' The second choice corresponds to deleting the last 'C,' and the third choice corresponds to inserting the character 'G' at the end of the string. Once we have made a choice of one of these three options, we are then left to find an optimal alignment of all of the remaining characters. In other words, the optimal alignment of "ACAATCC" and "AGCATCG" is one of:

$$\text{OPTIMAL\_ALIGNMENT}(\text{"ACAATC"}, \text{"AGCATC"}) + \begin{bmatrix} \texttt{C} \\ \texttt{G} \\ * \end{bmatrix}$$

$$\text{OPTIMAL\_ALIGNMENT}(\text{"ACAATC"}, \text{"AGCATCG"}) + \begin{bmatrix} \texttt{C} \\ \texttt{-} \\ * \end{bmatrix}$$

$$\text{OPTIMAL\_ALIGNMENT}(\text{“ACAATCC”}, \text{“AGCATC”}) + \begin{bmatrix} - \\ \texttt{G} \\ * \end{bmatrix}$$

The alignment of the remaining prefixes must be optimal because if it were not, we could substitute it for a more optimal one and achieve a better overall alignment. Let us therefore define our subproblems as follows:

$$\text{Dist}[i, j] = \{\text{The edit distance between the prefixes } S_1[1..\text{i}] \text{ and } S_2[1..\text{j}]\}$$

for $0 \leq i \leq n$, $0 \leq j \leq m$.

Deriving a recurrence

Using the optimal subproblems observed above, we can derive a recurrence for the edit distance between two strings $S_1[1..n]$ and $S_2[1..m]$ like so. Consider the computation of the subproblem $\text{Dist}[i, j]$, where $i, j > 0$. We have three choices:

1. Align $S_1[i]$ with $S_2[j]$, which has a cost of zero if $S_1[i] = S_2[j]$, or one otherwise.
2. Delete the character $S_1[i]$, which has a cost of one.
3. Insert the character $S_2[j]$, which has a cost of one.

If we choose to align $S_1[i]$ with $S_2[j]$, then the subproblem that remains to be solved is to align the remaining prefixes i.e. to minimise $\text{Dist}[i-1, j-1]$. If we delete the character $S_1[i]$, then we need to align the remaining prefix of S_1 with S_2, i.e. we are interested in minimising the subproblem $\text{Dist}[i-1, j]$. Finally, if we chose to insert the character $S_2[j]$, then we are left to align the remaining prefix of S_2 with S_1, i.e. we want to optimise the subproblem $\text{Dist}[i, j-1]$. Trying all three possibilities and selecting the best one leads to the following general recurrence for $\text{Dist}[i, j]$.

$$\text{Dist}[i, j] = \begin{cases} i & \text{if } j = 0, \\ j & \text{if } i = 0, \\ \min \begin{cases} \text{Dist}[i-1, j-1] + 1_{S_1[i] \neq S_2[j]} \\ \text{Dist}[i-1, j] + 1 \\ \text{Dist}[i, j-1] + 1 \end{cases} & \text{otherwise} \end{cases}$$

We have used the notation $1_{S_1[i] \neq S_2[j]}$ to denote an indicator variable for $S_1[i] \neq S_2[j]$. In other words, it equals one if $S_1[i] \neq S_2[j]$, or zero otherwise. The base cases should be easy to understand. If one of the two strings is empty, then an optimal alignment simply consists of inserting all of the characters of the other string.

Implementation of edit distance

A bottom-up implementation of the dynamic programming solution for the edit distance problem is shown in Algorithm 51. We note that the order in which we must compute the subproblems corresponds to prefixes of increasing length since each subproblem depends only on the subproblems of length one shorter. There are a total of $n \times m$ subproblems, and computing the solution to each one requires us to look back at just three previous ones, hence the time and space complexity of the edit distance dynamic programming algorithm is $O(nm)$.

Algorithm 51 Bottom-up edit distance

```
1:  function EDIT_DISTANCE(S1[1..n], S2[1..m])
2:      Dist[0..n][0..m] = 0
3:      Dist[1..n][0] = 1..n
4:      Dist[0][1..m] = 1..m
5:      for i = 1 to n do
6:          for j = 1 to m do
7:              if S1[i] = S2[j] then
8:                  Dist[i][j] = Dist[i-1][j-1]
9:              else
10:                 Dist[i][j] = min(Dist[i-1][j-1], Dist[i-1][j], Dist[i][j-1]) + 1
11:     return Dist[n][m]
```

Constructing an optimal alignment

In order to construct the alignment for the strings S_1 and S_2, we need to backtrack through the dynamic programming subproblems from $Dist[n,m]$ to $Dist[0,0]$, examining the choices that were made along the way. The following function shown in Algorithm 52 takes the filled dynamic programming table $Dist[0..n][0..m]$ and produces the alignment. Note that we build the alignment in reverse since we backtrack from the ends of the strings.

Algorithm 52 Optimal sequence alignment

```
1:  function ALIGNMENT(S1[1..n], S2[1..m], Dist[0..n][0..m])
2:      A1 = "", A2 = ""
3:      i = n, j = m
4:      while i + j > 0 do
5:          if i = 0 then
6:              A1 += "-", A2 += S2[j]
7:              j = j − 1
8:          else if j = 0 then
9:              A1 += S1[i], A2 += "-"
10:             i = i − 1
11:         else
12:             best = min(Dist[i − 1][j − 1], Dist[i][j − 1], Dist[i − 1][j])
13:             if S1[i] = S2[j] or Dist[i − 1][j − 1] = best then        // Substitution
14:                 A1 += S1[i], A2 += S2[j]
15:                 i = i − 1, j = j − 1
16:             else if Dist[i − 1][j] = best then                       // Deletion from S1
17:                 A1 += S1[i], A2 += "-"
18:                 i = i − 1
19:             else                                                     // Insertion into S1
20:                 A1 += "-", A2 += S2[j]
21:                 j = j − 1
22:     return reverse(A1), reverse(A2)
```

7.7 Matrix Multiplication

Recall the problem of multiplying two matrices A and B. If A is an $n \times m$ matrix and B is an $m \times o$ matrix, then their product will be an $n \times o$ matrix.

$$\underbrace{\begin{bmatrix} \\ A \\ \\ \end{bmatrix}}_{n \times m} \times \underbrace{\begin{bmatrix} \\ B \\ \\ \end{bmatrix}}_{m \times o} = \underbrace{\begin{bmatrix} \\ A \times B \\ \\ \end{bmatrix}}_{n \times o}$$

In general, the amount of work required to multiply an $n \times m$ matrix and an $m \times o$ matrix is $n \times m \times o$ scalar multiplications. Consider now the problem of multiplying a sequence of matrices of differing sizes. For example, suppose we want to compute the product of a 4×2, a 2×5 and a 5×3 matrix.

$$\underbrace{\begin{bmatrix} \\ A \\ \\ \end{bmatrix}}_{4 \times 2} \times \underbrace{\begin{bmatrix} \\ B \\ \end{bmatrix}}_{2 \times 5} \times \underbrace{\begin{bmatrix} \\ C \\ \\ \end{bmatrix}}_{5 \times 3}$$

There are several ways that we could compute this product, owing to the fact that matrix multiplication is associative. This means that we can bracket the product in any way we want and we will still obtain the same answer. We can compute this product in either the order $(A \times B) \times C$ or $A \times (B \times C)$. Surprisingly, the order that choose to do the multiplications actually has a substantial effect on the amount of computational time required. If we choose the first order $(A \times B) \times C$, then we perform $(4 \times 2 \times 5) + (4 \times 5 \times 3) = 100$ scalar multiplications to obtain the answer. If we use the second order $A \times (B \times C)$, then we only perform $(2 \times 5 \times 3) + (4 \times 2 \times 3) = 54$ scalar multiplications. That's almost half the amount of work! The general optimal matrix multiplication problem is the problem of deciding for a sequence of n matrices, the best way to multiply them to minimise the number of scalar multiplications required.

For a sequence of four matrices, there are five different ways to parenthesise them as follows: $A \times ((B \times C) \times D)$, $A \times (B \times (C \times D))$, $(A \times B) \times (C \times D)$, $(((A \times B) \times C) \times D)$ and $(A \times (B \times C)) \times D$. In general, the number of ways to parenthesise a sequence of n matrices is exponential in n (it is related to the famous Catalan number sequence), so trying all of them would be way too slow. Dynamic programming can be used to quickly find a solution to the optimal matrix multiplication problem.

Identifying the optimal subproblems

Consider a sequence of n matrices $A_1, A_2, A_3, A_4, ...A_n$ to be multiplied (assume that they are of valid dimensions to be multiplied.) Think about the final multiplication that happens in the

sequence. This multiplication must be the product of two matrices

$$\underbrace{(A_1 \times A_2 ... \times A_k)}_{\substack{\text{Result of multiplying} \\ \text{matrices 1 to } k}} \times \underbrace{(A_{k+1} \times A_{k+2} \times ... \times A_n)}_{\substack{\text{Result of multiplying} \\ \text{matrices } k+1 \text{ to } n}},$$

where $(A_1 \times A_2 ... \times A_k)$ and $(A_{k+1} \times A_{k+2} \times ... \times A_n)$ were multiplied earlier. If this is the optimal order in which to multiply the matrices, then the two subproblems of multiplying $(A_1 \times A_2 ... \times A_k)$ and $(A_{k+1} \times A_{k+2} \times ... \times A_n)$ must also be multiplied in optimal order. If they were not, we could substitute them for a better order and obtain a better solution. This is the optimal substructure for the optimal matrix multiplication problem.

The dynamic programming solution for the optimal matrix multiplication problem therefore involves trying every possible "split point" k at which to perform the final multiplication. The problem is then recursively solved by asking for the optimal order in which to multiply matrices 1 to k and the remaining matrices $k+1$ to n. Our subproblems should therefore be:

$$\text{DP}[i, j] = \{\text{The minimum number of multiplications required to multiply } (A_i, A_{i+1}, ... A_j)\}$$

for all $1 \leq i \leq j \leq n$.

Deriving a recurrence

Let us denote the number of columns in the matrices $A_1, A_2, ... A_n$ by the sequence $c_1, c_2, ..., c_n$. Recalling that two adjacent matrices can only be multiplied if the number of columns of the first is the same as the number of rows of the second, we know that if the sequence of matrices (A_i) is valid, then the number of rows in $A_2, ... A_n$ must be $c_1, ... c_{n-1}$. For completeness, we will further denote the number of rows of A_1 as c_0. A base case for our problems is easy to identify. It takes zero multiplications to multiply a sequence of just one matrix. So $\text{DP}[i, i] = 0$ for all i.

For the general case, to find the optimal number of multiplications for the sequence $A_i, ... A_j$ where $i < j$, we must try every possible "split point" and recurse. If we perform the split after matrix k, where $i \leq k < j$, then the total number of required multiplications is $\text{DP}[i, k] + \text{DP}[k+1, j]$ plus the number of multiplications required to multiply $(A_i \times ... \times A_k)$ and $(A_{k+1} \times ... \times A_j)$. The matrix $(A_i \times ... \times A_k)$ has exactly c_{i-1} rows and c_k columns, and the matrix $(A_i \times ... \times A_k)$ and $(A_{k+1} \times ... \times A_j)$ has exactly c_k rows and c_j columns. Hence the number of scalar multiplications required to multiply them is $c_{i-1} \times c_k \times c_j$. Therefore the total number of required multiplications to multiply matrices i to j if we split after matrix k is given by

$$\text{DP}[i, j] = \text{DP}[i, k] + \text{DP}[k+1, j] + c_{i-1} \times c_k \times c_j.$$

Combining everything together and trying all possible split points, we end up with the recurrence

$$\text{DP}[i, j] = \begin{cases} 0 & \text{if } i = j \\ \min\limits_{i \leq k < j} (\text{DP}[i, k] + \text{DP}[k+1, j] + c_{i-1} \times c_k \times c_j) & \text{if } i < j. \end{cases}$$

for $1 \leq i \leq j \leq n$. Note that we do not care about defining $\text{DP}[i, j]$ for $j < i$ since it does not make sense to have reversed intervals. With this recurrence, the answer to the entire problem is $\text{DP}[1, n]$, the optimal number of scalar multiplications required to multiply the entire sequence $A_1 \times ... \times A_n$.

Implementation of optimal matrix multiplication

Let's write a bottom-up solution to optimal matrix multiplication. First, we must consider the order in which the subproblems depend on each-other. For each subproblem DP$[i, j]$, we need to have already computed the values of DP$[i, k]$ and DP$[k+1, j]$ for all $i \leq k < j$. This means that it would be incorrect to simply loop over all $1 \leq j \leq n$ and $1 \leq i \leq j$ as we might be naturally inclined to. The essential observation is that the subproblems depend on subproblems consisting of a shorter sequence of matrices. A valid order in which to compute the solutions is therefore in order of length, i.e. we compute the answer for all contiguous subsequences of 2 matrices, then all contiguous subsequences of 3 matrices and so on. The algorithm is depicted in Algorithm 53.

Algorithm 53 Optimal matrix multiplication

```
1:  function MATRIX_ORDER(c[0..n])
2:      DP[1..n][1..n] = 0
3:      for length = 2 to n do
4:          for i = 1 to n - length + 1 do
5:              j = i + length - 1
6:              best = ∞
7:              for k = i to j − 1 do
8:                  best = min(best, DP[i][k] + DP[k + 1][j] + c[i − 1] × c[k] × c[j])
9:              DP[i][j] = best
10:     return DP[1][n]
```

The initialisation of DP takes time $O(n^2)$ and already sets the correct values of the subproblems with $i = j$. For the subproblems with $j > i$, there are $n(n-1)/2$ subproblems and the computation of each subproblem i, j requires looping over $j - i$ previous subproblems (using constant time for each iteration of the inner loop). Therefore to obtain the time complexity of computing all those subproblems we analyse the corresponding sum:

$$\begin{aligned}
\sum_{i=1}^{n}\sum_{j=i+1}^{n}(j-i) &= \sum_{i=1}^{n}\left(\sum_{j=i+1}^{n} j - \sum_{j=i+1}^{n} i\right), \\
&= \sum_{i=1}^{n}\sum_{j=i+1}^{n} j - \sum_{i=1}^{n}(n-i)i, \\
&= \sum_{i=1}^{n}\left(\frac{n(n+1)}{2} - \frac{i(i+1)}{2}\right) - n\sum_{i=1}^{n} i + \sum_{i=1}^{n} i^2, \\
&= \frac{n^2(n+1)}{2} - \frac{1}{2}\left(\sum_{i=1}^{n} i^2 + \sum_{i=1}^{n} i\right) - \frac{n^2(n+1)}{2} + \sum_{i=1}^{n} i^2, \\
&= \frac{1}{2}\sum_{i=1}^{n} i^2 - \frac{n(n+1)}{4}.
\end{aligned}$$

Finally, using the sum of squares formula, $\sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{12}$, we obtain

$$\begin{aligned}\sum_{i=1}^{n}\sum_{j=i+1}^{n}(j-i) &= \frac{1}{2}\sum_{i=1}^{n} i^2 - \frac{n(n+1)}{4} \\ &= \frac{n(n+1)(2n+1)}{12} - \frac{n(n+1)}{4},\end{aligned}$$

which is $O(n^3)$. Therefore the time complexity of the dynamic programming solution for optimal matrix multiplication is also $O(n^3)$. Since we have $O(n^2)$ subproblems, the space complexity is $O(n^2)$.

7.8 The Space-Saving Trick

You may have noticed that the recurrences for the edit distance problem and the 0-1 knapsack problem both had a very interesting property. Despite having a 2 dimensional table of subproblems, each subproblem only ever depended on subproblems from the previous row in order to compute it's optimal solution. Because of this, we do not need to keep the entire table in memory all at once. It is enough to simply store two rows of the table at a time, and re-use the same memory for every second row of subproblems that we compute. This reduces the space complexity of the edit distance and knapsack problems to $O(\min(n,m))$ and $O(C)$ respectively. Of course, we have to remember that if we employ the space-saving trick, we can no longer backtrack through the subproblems to construct the optimal solution.

Chapter 8

Dynamic Programming Graph Algorithms

In this section, we will study dynamic programming graph algorithms. Dynamic programming can, for instance, be used to solve the shortest path problem with negative weights.

Summary: Dynamic Programming Graph Algorithms

In this chapter, we cover:

- The Bellman-Ford algorithm for finding shortest paths in graphs with negative weights.
- The Floyd-Warshall algorithm to solve the all-pairs shortest path problem (and its relation to the transitive closure problem).

Throughout this chapter, we will assume that we are dealing only with directed graphs.

8.1 Shortest Paths with Negative Weights

The Bellman-Ford algorithm was one of the first applications of the dynamic programming paradigm. Given a source s, the Bellman-Ford algorithm returns the shortest distances from s to all vertices in the graph if there are no negative cycles that are reachable from s. If there is a negative cycle reachable from s, the algorithm returns an error (alternatively, it can be modified to return all valid shortest distances, and $-\infty$ for vertices which are affected by the negative cycle.

Key Ideas: Bellman-Ford Algorithm

- If no negative cycles are reachable from node s, then for every node t that is reachable from s there is a shortest path from s to t that is simple (i.e., no nodes are repeated).
- Cycles with positive weight cannot be part of a shortest path.
- Given a shortest path that contains cycles of weight 0, the cycles can be removed to obtain an alternative shortest path that is simple.
- Note that any simple path has at most $|V|-1$ edges.

For a source node s, let $\mathsf{OPT}[i,v]$ denote the minimum weight of a $s \rightsquigarrow v$ path with at most i edges. Let P be an optimal path with at most i edges that achieves total weight $\mathsf{OPT}[i,v]$:

- If P has at most $i-1$ edges, then $\mathsf{OPT}[i,v] = \mathsf{OPT}[i-1,v]$.
- If P has exactly i edges and (u,v) is the last edge of P, then $\mathsf{OPT}[i,v] = \mathsf{OPT}[i-1,u] + w(u,v)$.

The recursive formula for dynamic programming is then

$$\mathsf{OPT}[i,v] = \min\Big(\mathsf{OPT}[i-1,v], \min_{u:(u,v)\in E}(\mathsf{OPT}[i-1,u] + w(u,v))\Big).$$

To obtain the shortest paths we just need to compute $\mathsf{OPT}[|V|-1,v]$ for all vertices $v \in V$.

Commonly, a more space-efficient version of Bellman-Ford algorithm is implemented (in some cases, this version also provides a speed-up, but no improvement in the worst-case time complexity). The idea is that $|V|-1$ iterations of a loop are executed and in each iteration all edges are sequentially relaxed to obtain the new distance estimates. A simple implementation is shown in Algorithm 54. The time complexity of Bellman-Ford is easy to establish: the outer loop runs for $|V|-1$ iterations, the inner loop runs for $|E|$ iterations, and each relaxation takes $O(1)$, so the total time taken is $O(|V||E|)$.

Algorithm 54 Bellman-Ford

```
1: function BELLMAN_FORD(G = (V, E), s)
2:     dist[1..n] = ∞
3:     pred[1..n] = null
4:     dist[s] = 0
5:     for k = 1 to n − 1 do
6:         for each edge e in E do
7:             RELAX(e)
8:     return dist[1..n], pred[1..n]
```

Correctness of Bellman-Ford

The key to establishing the correctness of Bellman-Ford lies in a simple property about paths in graphs. We write a formal proof of correctness using induction.

Theorem: Correctness of Bellman-Ford

The Bellman-Ford algorithm terminates with the correct distance estimates to all vertices whose shortest path is well defined after at most $|V|-1$ iterations.

Proof

In a well defined shortest path, the number of edges on the path cannot be greater than $|V|-1$, why? If a path contained more than $|V|-1$ edges, then it must visit some vertex multiple times and hence contain a cycle, which we know we from our discussion before

that we can ignore. We claim that after k iterations of the Bellman-Ford algorithm, that all valid shortest paths consisting of at most k edges are correct.

We can prove this formally by induction on the number of edges in each shortest path. For the base case, we initialise dist[s] = 0, which is correct if there is no negative cycle containing s. Hence all valid shortest paths containing zero edges are correct.

Inductively, suppose that all valid shortest paths containing at most k edges are correct after k iterations. Let's argue that all valid shortest paths containing at most $k+1$ edges are correct after one more iteration. Consider some shortest path from $s \rightsquigarrow v$ consisting of at most $k+1$ edges. By the substructure of shortest paths, the sub-path $s \rightsquigarrow u$ consisting all but the final edge (u,v) is a shortest path, and by our inductive hypothesis, is correctly estimated at the current iteration of the algorithm. If the current distance estimate of v is incorrect, it will be relaxed in this iteration by the edge (u,v) and hence be made correct since $\delta(s,v) = \delta(s,u) + w(u,v) = dist[u] + w(u,v)$.

Hence by induction on the number of edges in the shortest paths, all valid shortest paths consisting of at most k edges are correctly estimated after k iterations. Therefore the Bellman-Ford algorithm terminates correctly after at most $|V|-1$ iterations.

Dealing with negative cycles

An important problem that needs to be addressed is how to deal with the existence of negative cycles. As mentioned earlier, the presence of a negative cycle may cause the shortest path to a particular vertex to become undefined since there may exist paths of arbitrarily short length. In this case, the algorithm above will terminate with an arbitrary distance estimate to those vertices that are reachable from a negative cycle. Checking if there is a negative cycle can be done by just running one additional iteration of Bellman-Ford and checking if any distance estimate would be improved. If $|V|$ additional iterations of Bellman-Ford are executed, then the influence of the negative cycles are guaranteed to propagate to all vertices that are reachable from the negative cycles, and any vertex whose distance estimate can be improved in those $|V|$ additional iterations of Bellman-Ford can simply have its distance estimate updated to $-\infty$. After running this post-processing procedure, all of the distances given in the *dist* array will be:

1. ∞ if the vertex in question was never reached (it is not connected to the source s).
2. $-\infty$ if the vertex in question is reachable via a negative cycle and hence does not have a shortest path.
3. The correct distance if the vertex has a well-defined shortest path.

Note that the *pred* array may contain cycles if there are negative cycles present, so do not attempt to reconstruct shortest paths to vertices whose distance is undefined!

Optimising Bellman-Ford

There are some simple optimisations that can be made to Bellman-Ford that improve its running time:

1. During the iterations of the main k loop, if no relaxations are successful, then the shortest paths have already been found and we can terminate early.
2. If such an early termination occurs, then we are guaranteed that no negatives cycles exist and hence can skip the post-processing step that checks for undefined shortest paths.
3. Rather than iterating over every single edge, we could instead maintain a queue of edges that need to be relaxed. If an edge is not successfully relaxed in one phase, then it clearly need not be relaxed in the next phase either unless its source vertex has its distance estimate improved. For sparse graphs, this improves the empirical average case complexity of the algorithm to just $O(|E|)$, although the worst-case behaviour is unfortunately still $O(|V||E|)$. This variant is often referred to as the "Shortest Path Faster Algorithm."

8.2 The All-Pairs Shortest Path Problem

Finally, we consider the all-pairs shortest path problem, in which we seek a shortest path between every pair of vertices in the given graph. The most obvious solution to this problem is to simply solve the single-source shortest path problem using every possible vertex as the source. This might sound inefficient, but in some situations, this is actually the optimal solution!

- Consider an unweighted graph. If we perform a breadth-first search from every possible source, each taking $O(|V|+|E|)$ time, then the complexity of finding all-pairs shortest paths is $O(|V|(|V|+|E|))$.
- In graph with nonnegative weights, we can run Dijkstra from every possible source and achieve a runtime of $O(|V||E|\log(|V|))$ assuming that our priority queue is a binary heap.
- In a graph with negative weights, multiple invocations of Bellman-Ford from every source would yield a time complexity of $O(|V|^2|E|)$.
- For graphs with negative weights, multiple invocations of Bellman-Ford is extremely slow. However, a clever trick exists called the *potential method* that allows us to re-weight the graph in such a way that we get rid of all of the negative weight edges without changing the shortest paths. Multiple invocations of Dijkstra's algorithm can then be made on the modified graph before reverting the weights and recovering the shortest paths. This algorithm is called Johnson's algorithm and has a runtime of $O(|V||E|\log(|V|))$, assuming Dijkstra's is implemented with a binary heap.

8.2.1 The Floyd-Warshall algorithm

The Floyd-Warshall algorithm is a very simple all-pairs shortest path algorithm that is fast for dense graphs, but can be beaten in performance on sparse graphs by multiple invocations of the single-source algorithms as described above. Floyd-Warshall relies on the observation that a shortest path from $u \rightsquigarrow v$ that has at least two edges can be decomposed into two shortest paths $u \rightsquigarrow k$ and $k \rightsquigarrow v$ for some intermediate vertex k. The dynamic programming algorithm simply iteratively increases the pool of possible intermediate vertices by adding one vertice k at a time (k going from 1 to n) and updating all paths that can be improved by visiting vertex k as an intermediate node.

We initialise the all-pairs distance estimate as a 2D matrix $dist[1..n][1..n]$ such that $dist[v][\text{v}]$

$= 0$ for all vertices v and $dist[u][\text{v}] = w(u,v)$ for all edges $e = (u,v)$. Observe that if the graph is represented as an adjacency matrix, then initialising the distance matrix dist is easy as this is nothing but a copy of the adjacency matrix! See the implementation in Algorithm 55. The time complexity of Floyd-Warshall is obvious. There are three loops from 1 to n and a constant time update each iteration, so the total runtime is $O(|V|^3)$. The space complexity is $O(|V|^2)$ since we store the $|V| \times |V|$ *dist* matrix (note that the algorithm implicitly already uses dynamic programming space savings techniques).

Algorithm 55 Floyd-Warshall

```
1: function FLOYD_WARSHALL(G = (V, E))
2:     dist[1..n][1..n] = ∞
3:     dist[v][v] = 0 for all vertices v
4:     dist[u][v] = w(u, v) for all edges e = (u, v) in E
5:     for each vertex k = 1 to n do
6:         for each vertex u = 1 to n do
7:             for each vertex v = 1 to n do
8:                 dist[u][v] = min(dist[u][v], dist[u][k] + dist[k][v])
9:     return dist[1..n][1..n]
```

Comparing this against the running times of the repeated single-source solution, we can see that for dense graphs, where $|E| \approx |V|^2$, Floyd-Warshall is the best or equal best in all cases. For sparse graphs however ($|E| \approx |V|$), Floyd-Warshall is beaten by $O(|V|^2)$ and $O(|V|^2 \log(|V|))$.

All-pairs shortest path algorithms	Sparse	Dense
Repeated BFS	$O(\|V\|^2)$	$O(\|V\|^3)$
Repeated Dijkstra	$O(\|V\|^2 \log(\|V\|))$	$O(\|V\|^3 \log(\|V\|))$
Repeated Bellman-Ford	$O(\|V\|^3)$	$O(\|V\|^4)$
Johnson	$O(\|V\|^2 \log(\|V\|))$	$O(\|V\|^3 \log(\|V\|))$
Floyd-Warshall	$O(\|V\|^3)$	$O(\|V\|^3)$

Finally, note that just like Bellman-Ford, Floyd-Warshall can handle negative weights just fine, and can also detect the presence of negative weight cycles. To detect negative weight cycles, simply inspect the diagonal entries of the dist matrix. If a vertex v has $dist[v][v] < 0$, then it must be contained in a negative weight cycle.

Correctness of Floyd-Warshall

Theorem: Correctness of Floyd-Warshall

Floyd-Warshall produces the correct distances for all pairs of vertices (u,v) such that there exists a shortest path between u and v.

Proof

We will establish the following invariant: After k iterations of Floyd-Warshall, $dist[u][v]$ contains the length of a shortest path $u \rightsquigarrow v$ consisting only of intermediate vertices from 1 to k.

For $k = 0$, we have performed no iterations yet. $dist[u][v]$ is therefore equal to $w(u, v)$ if an edge exists between u and v. Using no intermediate vertices at all, this the only path possible, and hence it is the shortest path possible. If $u = v$, then $dist[u][v] = 0$ which is optimal using no other vertices. If there is no edge (u, v), then $dist[u][v] = \infty$ which is correct since no paths are possible yet.

Suppose that after iteration k, $dist[u][v]$ contains the length of a shortest path $u \rightsquigarrow v$ consisting only of intermediate vertices from 1 to k. We will show that after iteration $k+1$, $dist[u][v]$ contains the length of a shortest path $u \rightsquigarrow v$ consisting only of intermediate vertices from 1 to $k+1$. Consider a shortest path between some vertices u and v consisting of intermediate vertices in 1 to $k+1$. If vertex $k+1$ is not a part of the path then by the inductive hypothesis we already have the optimal length. Otherwise, this path contains only one occurrence of vertex $k+1$, since if it contained multiple, it would contain a cycle. Therefore there exists two sub-paths $u \rightsquigarrow k+1$ and $k+1 \rightsquigarrow v$ which do not contain vertex $k+1$ as an intermediate vertex. By the inductive hypothesis, $dist[u][k+1]$ and $dist[k+1][v]$ are optimal, and since together they form the shortest path, the update will set $dist[u][v] = dist[u][k+1] + dist[k+1][v]$ which is now optimal. Therefore after iteration $k+1$, $dist[u][v]$ contains the shortest path $u \rightsquigarrow v$ consisting of intermediate vertices from 1 to $k+1$.

By induction on k, after iteration n, $dist[u][v]$ contains the length of a shortest path $u \rightsquigarrow v$ consisting of any intermediate vertices, hence the distances are optimal.

8.3 Transitive Closure

Definition: Transitive Closure

The transitive closure of a graph G is a new graph G' on the same vertices such that there is an edge (u, v) in G' if and only if there is a path between u and v in G.

Finding the transitive closure of a graph can be reduced to the problem of finding all-pairs shortest paths by noting that there is an edge between u and v in the transitive closure if and only if $dist[u][v] < \infty$. We can save some space by observing that we only ever care about whether or not two vertices are connected, not what their distance value is. Given this, we can store our connectivity matrix using single bits for each entry (a data structure called a *bitset* or *bitvector* in some languages), where 0 means disconnected and 1 means connected, and update them by noting that vertex u is connected to vertex v via vertex k if and only if $connected[i][k]$ and $connected[k][j]$ are both true. The space complexity of the algorithm therefore becomes $O\left(|V|^2/w\right)$, where w is the number of bits in a machine word. The time complexity can also be improved to $O\left(|V|^3/w\right)$ by using bitwise operations, but we won't discuss that for now.

Algorithm 56 Warshall's transitive closure algorithm

```
1: function TRANSITIVE_CLOSURE(G = (V, E))
2:     connected[1..n][1..n] = false                    // Store using a bitvector
3:     connected[v][v] = true for all vertices v
4:     connected[u][v] = true for all edges e = (u, v) in E
5:     for each vertex k = 1 to n do
6:         for each vertex u = 1 to n do
7:             for each vertex v = 1 to n do
8:                 connected[u][v] = connected[u][v] or (connected[u][k] and connected[k][v])
9:     return connected[1..n][1..n]
```

8.4 The Critical Path Problem

The critical path problem is the problem of finding the **longest** path in a directed acyclic graph. This problem arises frequently in project management and operations research, where the vertices of a graph represent task completions and the edges represent tasks that must be completed, weighted by the amount of time that it will take to do so. The edges of the graph dictate the prerequisites of the tasks in the project. The longest path in the DAG therefore corresponds to the minimum amount of time in which the project can be completed if all prerequisites are obeyed.

The critical path problem can be formulated as a simple dynamic programming problem. On a DAG, just like shortest paths, longest paths also admit substructure, i.e. any sub-path of a longest path must also be a longest path. Note that this is not true for a general graph (a graph that may potentially have cycles), which is why longest path is only efficiently solvable on DAGs.

For all vertices $u \in V$, let $longest[u]$ denote the length of the longest path that starts at u. Our base cases are all of the vertices with no outgoing edges, where $longest[v] = 0$. To determine the longest path that begins at some arbitrary vertex u, we simply consider all of the outgoing edges of u, and see which one yields the longest total path. In other words, we compute the dynamic program

$$longest[u] = \max_{v \in \text{adj[u]}} \big(w(u,v) + longest[v]\big).$$

The dependencies of the subproblems are clear from the graph, we must compute the longest path for all descendants of a node before we can compute the result for that node. In other words, the subproblems are dependant in a reverse topological order. Computing a topological order and iterating over every vertex and edge in the graph takes $O(|V|+|E|)$ time, hence we can solve the critical path problem in $O(|V|+|E|)$ time.

Since we are required to compute the topological order, we might as well solve the problem top-down recursively which removes the need to know the order of dependencies of the subproblems. This results in a very short and elegant solution which also runs in $O(|V|+|E|)$ time.

Relationship between topological orderings and dynamic programming

The recursive solution to the critical path problems reveals a very subtle but interesting fact about the relationship between depth-first search, topological orders, and dynamic programming. Take a look at the pseudocode in Algorithm 58. It looks a lot like the pseudocode for

Algorithm 57 Bottom-up longest path in a DAG

```
1: function CRITICAL_PATH(G = (V, E))
2:     longest[1..n] = 0
3:     order = reverse(TOPOLOGICAL_SORT(G))
4:     for each vertex u in order do
5:         for each edge (u, v) adjacent to u do
6:             longest[u] = max(longest[u], w(u, v) + longest[v])
7:     return longest[1..n]
```

Algorithm 58 Recursive longest path in a DAG

```
1: function CRITICAL_PATH(u)
2:     if longest[u] = null then          // longest[1..n] contains memoised subproblems
3:         longest[u] = 0
4:         for each edge (u, v) adjacent to u do
5:             longest[u] = max(longest[u], w(u, v) + CRITICAL_PATH(v))
6:     return longest[u]
```

depth-first search (it is doing one after all), and it looks a lot like our top-down dynamic programming algorithms (which it is also).

Although it might not be immediately obvious, this problem actually illuminates a very interesting fact about dynamic programming that we saw when comparing top-down vs. bottom-up solutions. When computing a dynamic program bottom-up, it is necessary to know the order in which the subproblems are dependent on each other so that all of the necessary values are available when needed to compute each subsequent subproblem. When implementing a solution top-down however, we did not need to know the order of the subproblems since they would be computed precisely when needed.

The subproblems cannot depend on each other cyclically, so their dependencies on each other actually form a DAG, where each subproblem is a vertex, and each dependency is a directed edge. The order in which we compute the subproblems must therefore be a valid topological order of that DAG. Whenever we compute a solution to a dynamic program using top-down recursion, we are actually performing a **depth-first search on the subproblem graph**, which means that we are actually producing a reverse topological order as a by-product! This explains why top-down solutions do not need to know a valid order in advance, because they are performing precisely the algorithm that computes one, even if we did not realise it.

This realisation allows us to rephrase many dynamic programming problems as path problems on DAGs. For example, recall the coin change problem from Chapter 7. If we construct a graph G where each node corresponds to a dollar amount $\$V$, and add edges between the dollar amounts $(V, V - c_i)$ for all coin denominations c_i and all dollar amounts V, then the coin change problem is precisely the problem of finding the shortest path from the desired dollar amount in this DAG to the \$0 vertex!

Chapter 9

Network Flow

Network flow models describe the flow of some material through a network with fixed capacities, with applications arising in several areas of combinatorial optimisation. On the surface, network flow can be used to model fluid in pipes, telecommunication networks, transport networks, financial networks and more. Beyond its obvious applications however, network flow turns out to be useful and applicable to solving a large range of other combinatorial problems, including graph matchings, scheduling, image segmentation, project management, and many more.

Summary: Network Flow

In this chapter, we cover:

- Network flow and the maximum flow problem.
- The Ford-Fulkerson algorithm for maximum flow.
- The minimum cut problem.
- The min-cut max-flow theorem.
- The bipartite matching problem and its solution using max flow.
- Circulations with demands and lower bounds.

Network Flow Problems

Consider a road network consisting of towns connected by roads, along each of which a company can send up to a certain number of trucks per day. Our trucking company wants to know the maximum number of trucks that they can send per day from Vancouver to Winnipeg if they distribute them optimally. See the example in Figure 9.1.

In this case, an optimal solution turns out to be to send:

- 12 trucks from Vancouver to Edmonton,
- 12 trucks from Edmonton to Saskatoon,
- 19 trucks from Saskatoon to Winnipeg,
- 11 trucks from Vancouver to Calgary,
- 11 trucks from Calgary to Regina,
- 7 trucks from Regina to Saskatoon,
- 4 trucks from Regina to Winnipeg,

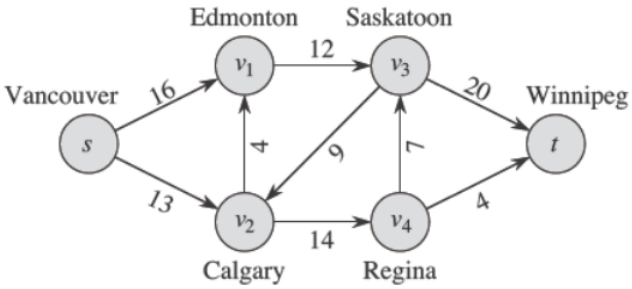


Figure 9.1: A road network of major cities in Canada. The edge weights represent the capacity of the roads, say the number of trucks that can be sent along that particular road per day. Image source: CLRS

which results in a total flow of 23 trucks per day. See Figure 9.2.

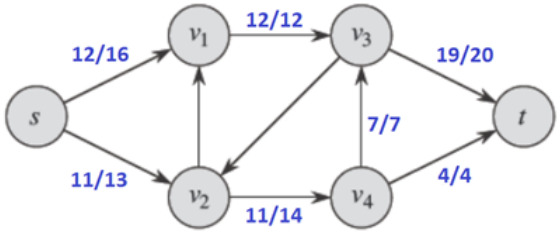


Figure 9.2: An optimal solution to the road network problem in Figure 9.1. A label of x/y indicates that we send x trucks along the road which has capacity y. Image source: Adapted from CLRS

A similar problem might be to consider a computer network, consisting of computers (nodes) that are connected via a wired network. Each pair of connected computers can send messages to each other up to some fixed capacity, called their bandwidth. Given the bandwidths of each connection, we might want to find the total bandwidth between two computers given that we can send data along multiple different paths at the same time.

All of these are examples of **network flow** problems. In short, we have a network consisting of nodes, one of which is the "source", the location at which the flow is produced, and the "sink", the location at which the flow is consumed. The problem that we will consider is the **maximum network flow** problem, or max-flow for short, where we are interesting in maximising the total amount of flow that can be sent from the source node to the sink node.

Mathematical formulation as a linear program

The intuitive definition and examples above are more than enough to understand the maximum flow problem and the algorithms that we'll cover to solve it, but for completeness, here is the formal definition.

Problem Statement: Maximum network flow problem

Let $G = (V, E)$ be a directed, edge-weighted network with two special vertices s and t, the source and sink respectively, such that no edge enters the source s and no edge leaves the sink t. Denote the capacity (edge weight) of an edge by $c(u, v)$. We will assume for now that all capacities are integers.[a] A $s-t$ flow in G is a function f that maps each pair of vertices to a non-negative real number

$$f : V \times V \to \mathbb{R}^+$$

that satisfies the following conditions:

- **Capacity constraint:** The flow along an edge must not exceed the capacity of that edge. Formally,
$$0 \leq f(u, v) \leq c(u, v).$$
- **Flow conservation:** The amount of flow entering a node must be equal to the amount of flow leaving that node, with the exception of the source and sink. That is for all $u \in V \setminus \{s, t\}$,
$$\sum_{v \in V} f(v, u) = \sum_{v \in V} f(u, v)$$
where $f(u, v) = 0$ if there is no edge (u, v), i.e. if $(u, v) \notin E$.

The **value** of a flow $|f|$ is given by

$$|f| = \sum_{v \in V} f(s, v).$$

The maximum flow problem then seeks to find a flow f that maximises the value:

$$\max_{\text{flow} f} |f| \quad \text{subject to the } \textbf{capacity} \text{ and } \textbf{flow conservation} \text{ constraints.}$$

[a] The maximum flow problem can in fact be generalised to consider non-integer capacities, although the Ford-Fulkerson algorithm is no longer guaranteed to work.

9.1 The Ford-Fulkerson Algorithm

The Ford-Fulkerson algorithm, often referred to as the Ford-Fulkerson method since it encompasses a variety of potential implementations is one of the first and most intuitive algorithm for solving the maximum flow problem. Stated as briefly as possible, the algorithm is effectively:

While more flow can be pushed through the network, push more flow through the network.

Of course, we have to do some work to define what we really mean by pushing flow through a network, and to figure out how to find paths through which we are able to push flow. For example, simple naively pushing flow through the network until there is no longer a capacitated path from the source to the sink won't necessarily result in an optimal solution.

To understand the Ford-Fulkerson method, we first need to define and explore the notion of the **residual network.**

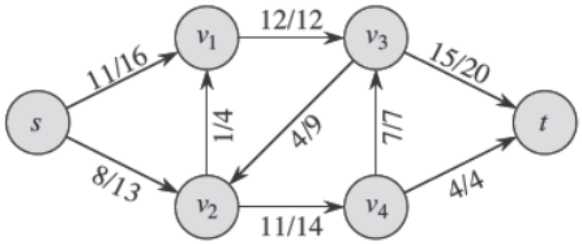


Figure 9.3: A flow on the road network from Figure 9.1. No more capacitated paths exist from the source to the sink, but the total flow of 19 is not optimal. In an optimal solution, we'd like to redirect the 4 flow that is going from v_3 to v_2 to go to t instead. In other words, we need a way to send suboptimal flow choices back and redirect them. Image source: CLRS

9.1.1 The Residual Network

Given a flow network $G = (V, E)$ and a flow f, the residual network G_f essentially represents how much more flow can be sent through each edge. For example, an edge with capacity 9 and flow 4 has residual capacity 5, as 5 additional units of flow can be pushed along it. The residual capacity of an edge (u, v) is given by

$$c_f(u, v) = c(u, v) - f(u, v).$$

Additionally, the residual network contains *back edges* or *reverse edges*, which are edges that flow in the opposite direction to those in E. The back edges are what allow the algorithm to "undo" flow by pushing flow back along the edge that it came from, and along a new direction instead. For each edge (u, v) with positive flow $f(u, v)$, the residual network contains an edge (v, u) with residual capacity

$$c_f(v, u) = f(u, v).$$

In other words, the back edge (v, u) has the ability to cancel out the flow that currently exists on the edge (u, v) by redirecting it down a different path.

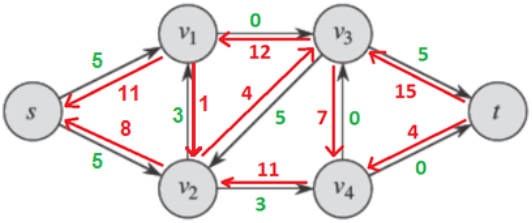


Figure 9.4: The residual network G_f corresponding to the network and flow in Figure 9.3. The green labelled edges are the forward edges from E with capacity $c(u, v) - f(u, v)$. The red edges are the back edges with capacity $f(v, u)$. Image source: Adapted from CLRS

9.1.2 Augmenting Paths

The example of the suboptimal flow in Figure 9.3 demonstrates that we cannot always improve a flow by seeking a capacitated path in G. In order to increase the flow in a network, we instead seek *augmenting paths*, which are capacitated paths in the residual network G_f. An augmenting path may contain forward edges or back edges, which correspond to two situations:

- Augmenting along a forward edge simply corresponds to pushing more flow through that edge on the way from the source to the sink.
- Augmenting along a back edge corresponds to redirecting flow along that edge down a different path. The flow is reduced on the respective forward edge, and is moved to the remainder of the augmenting path.

Although the flow on the road network depicted in Figure 9.3 has no more capacitated paths from s to t, the residual network does indeed have a capacitated path given by $s \to v_2 \to v_3 \to t$. The total capacity of the path is the minimum capacity of the edges encountered along the path (that is the bottleneck of the path), as clearly we can send no more than this amount of additional flow along the path (or we would violate a capacity constraint.) Filling an edge to full capacity is often referred to as "saturating" the edge. Similarly, an edge is called saturated if the flow along it is at capacity.

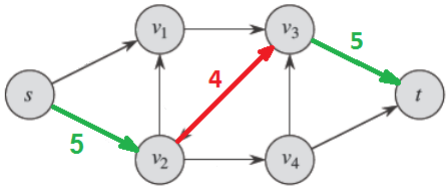


Figure 9.5: An augmenting path p the in residual network G_f in Figure 9.4. The total capacity of the augmenting path is 4, the bottleneck of the edge capacities in p. Image source: Adapted from CLRS

Augmenting 4 units of flow along the path p depicted in Figure 9.5 will result in a total flow of 23 units which is now optimal. The key thing to understand here is that by augmenting along a back edge, we have redirected the flow that was once flowing along $v_3 \to v_2 \to v_4 \to t$ to flow directly from $v_3 \to t$ instead. As this flow is redirected away, the 4 units of flow that was being supplied to v_2 by v_3 is now supplied by s, so that the flow along $v_2 \to v_4 \to t$ is not reduced. Since any flow that is reduced by augmenting along a back edge always gets replaced, an augmentation always increases the total value of the flow.

The method

We now have everything we need to describe the Ford-Fulkerson method. See the implementation in Algorithm 59. Of course, the Ford-Fulkerson method is really just a skeleton. The method tells us what to do, but does not specifically tell us **how** to do it. There are many possible strategies for actually finding augmenting paths, each of which will result in a different time complexity.

Algorithm 59 The Ford-Fulkerson method

```
1: function MAX_FLOW(G = (V, E), s, t)
2:     Set initial flow f to 0 on all edges
3:     while there exists an augmenting path p in the residual network G_f do
4:         Augment the flow f along the augmenting path p as much as possible
5:     return f
```

Note that Ford-Fulkerson method will always find integer-valued flows (i.e., flows in which the flow on every edge has an integer value). This is due to the facts that initial flow is 0 (invariant at the beginning) and all edges capacities are integers. Therefore in each step of the loop the flow will be augmented along the augmenting path by an integer amount, thus keeping the invariant.

9.1.3 Implementation Using Depth-First Search

The simplest way to find augmenting paths in the residual network is to simply run a depth-first search on G_f. Before we present the implementation, lets just look at a few tricks that we use to make the algorithm easier to write.

Use zero capacity edges to effectively represent back edges: In a naive implementation, we might try to write separate code for dealing with forward edges and back edges, but we can actually use a very simple trick to make them work the same, eliminating repeated code. All we need to do is define the capacity of the back edges to be 0, and to maintain that their flow value is always equal to the negative of the flow on the corresponding forward edge. This works because the formula for residual capacity on an edge

$$c_f(u,v) = c(u,v) - f(u,v),$$

when applied to a back edge will yield

$$c_f(v,u) = c(v,u) - f(v,u) = 0 - (-f(u,v)) = f(u,v),$$

which is the definition of the residual capacity on a back edge. In other words, this formulation allows us to treat forward and back edges the same since the formula for their residual capacities is now the same.

Augment flow in the same routine that finds an augmenting path: Once an augmenting path has been found, we know that we are going to augment along it, so we might as well make our depth-first search that finds augmenting paths also perform the augmentation at the same time. Otherwise we would simply have to write twice the length of code where we would have to locate an augmenting path and then separately traverse it again to add flow.

Edge structure: Let's assume that we have an object structure representing an edge of the network, which contains the fields `capacity`, indicating the edge's capacity (remember that this will be zero for back edges), `flow`, which will indicate the current flow on that edge, and `reverse` which contains a reference to its reverse edge (the back-edge corresponding to the forward edge or vice versa).

Throwing all of these tricks together yields the pseudocode depicted in Algorithm 60.

Algorithm 60 Ford-Fulkerson implemented using depth-first search

```
1:  // DFS returns the capacity of the augmenting path found (or 0 if there are none left)
2:  function DFS(u, t, bottleneck)
3:      if u = t then return bottleneck          // We hit the sink, so we have an augmenting path
4:      visited[u] = true
5:      for each edge e = (u, v) adjacent to u do
6:          residual = e.capacity - e.flow
7:          if residual > 0 and not visited[v] then
8:              augment = DFS(v, t, min(bottleneck, residual))
9:              if augment > 0 then              // We found an augmenting path - add the flow
10:                 e.flow += augment
11:                 e.reverse.flow -= augment
12:                 return augment
13:     return 0                                 // We could not find an augmenting path
14:
15: function MAX_FLOW(G = (V, E), s, t)
16:     flow = 0
17:     do
18:         visited[1..n] = false
19:         augment = DFS(s, t, ∞)
20:         flow += augment
21:     loop while augment > 0
22:     return flow
```

Time Complexity

The time complexity of the Ford-Fulkerson algorithm depends on the strategy selected to find augmenting paths. Let's consider the simplest variant in which we use a depth-first search to find **any** augmenting path.

- Finding an augmenting path if one exists using a depth-first search from the source s takes $O(|E|)$ time.
- In the worst case, each augmentation only increases the amount of flow by one unit, and hence it could take up to $|f_{max}|$ augmentations, where $|f_{max}|$ is the value of a maximum flow.

Therefore, the total time complexity of the Ford-Fulkerson algorithm is $O(|E| \times |f_{max}|)$.

Better strategies for selecting augmenting paths

We can improve on the time complexity bound of $O(|E| \times |f_{max}|)$ by selecting better augmenting paths. We won't explore the details, but just summarise some of the common strategies:

- **Select shortest augmenting paths:** If we use a breadth-first search instead of a depth-first search, then we will find augmenting paths with the fewest number of edges. This is called the **Edmonds-Karp algorithm** and has a runtime complexity of $O(|V||E|^2)$.
- **Select augmenting paths with the largest capacity:** Finding a path with the maximum

possible capacity can be achieved by finding a **maximum** spanning tree[1] in the residual graph G_f with Prim's algorithm and then augmenting along the resulting path from s to t. This is called the **fattest augmenting path algorithm** and results in a runtime of

$$O\left(|E|^2 \log(|V|) \log\left(|E| \max_{u,v \in V} c(u,v)\right)\right),$$

which is better, although much trickier to read than the complexity of Edmonds-Karp!

9.2 The Minimum Cut Problem

Given a flow network $G = (V, E)$ with edge capacities and a designated source vertex s and sink vertex t, the minimum cut problem is the problem of finding a set of edges of minimum total capacity that if removed, would disconnect s from t.

Let's illustrate this with a realistic example. Let's say you have a road network consisting of directed roads. Suppose that your tutor lives at vertex s and that Monash University is at vertex t. You have a test coming up in your next lab, so you want to sabotage your tutor's chances of getting to the lab on time to avoid taking the test! For each directed road, you know the dollar amount that it would cost to have that road temporarily shut down and blocked off. What is the minimum amount of money that you'll have to spend in order to ensure that there is no way for your tutor to get to the university?

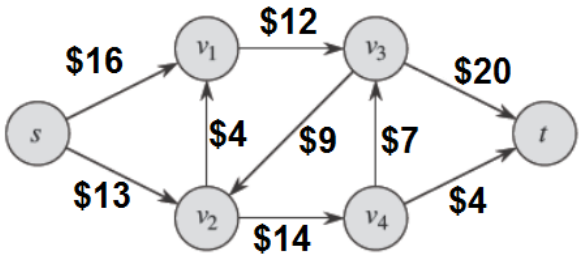


Figure 9.6: A road network, where edges are labelled by the cost of blocking off that particular road. Image source: Adapted from CLRS

This is precisely an example of a minimum cut problem. If we interpret the costs of blocking each road as the edges capacity, then the minimum cut is the cheapest way to disconnect s from t. Consider the problem in Figure 9.6. By inspection, we can see that the minimum cut for the network above corresponds to blocking off the roads $v_1 \to v_3$, $v_4 \to v_3$ and $v_4 \to t$ for a total cost of $23.

The minimum cut problem is intimately related to the maximum flow problem. Notice that this road network is actually the same network that we used as an example for maximum flow in the previous section, whose maximum flow turned out to be exactly 23 units. This is not a coincidence.

[1] Exactly the same as a minimum spanning tree except selecting the heaviest edges instead of the lightest.

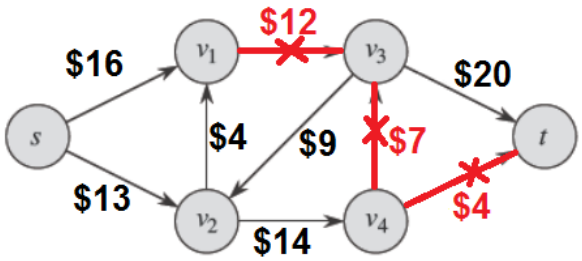


Figure 9.7: An optimal solution to the minimum cut problem. Image source: Adapted from CLRS

9.2.1 The Min-cut Max-flow Theorem

One of the most interesting theorems in network theory is the **min-cut max-flow theorem.** Stated formally, the minimum cut problem is as follows.

Problem Statement: Minimum cut problem

Let $G = (V, E)$ be a directed, edge-weighted network with two special vertices s and t, the source and sink respectively. Denote the capacity (edge weight) of an edge by $c(u, v)$.

An **s-t cut** $C = (S, T)$ is a partition of the vertices V into two disjoint subsets S and T such that $s \in S$ and $t \in T$. The **capacity** of the cut C is defined to be the total capacity of all edges that begin in S and end in T, in other words, all edges that **cross** the cut. Formally,

$$c(S, T) = \sum_{u \in S} \sum_{v \in T} c(u, v)$$

where as usual, we take $c(u, v) = 0$ if there does not exist an edge (u, v). The **minimum cut** problem seeks an s-t cut $C = (S, T)$ such that $c(S, T)$ is as small as possible.

The minimum cut problem and the maximum flow problem are related by the min-cut max-flow theorem. Informally, in a given flow network, the minimum cut and maximum flow problems have the same solution.

Theorem: Min-cut max-flow theorem

Given a flow network $G = (V, E)$ with a source vertex s and sink vertex t, the value of a maximum s-t flow in G is equal to the minimum capacity s-t cut in G. Mathematically:

$$\max_f |f| = \min_{(S,T)} c(S, T).$$

Proof

For a rigorous mathematical proof of the theorem, see CLRS, Theorem 26.6. We will consider a more intuitive version of the proof, which follows roughly the same structure

as the formal proof, but using intuitive arguments in place of rigorous mathematical arguments.

If we consider a flow network G, some flow f and some cut (S,T), the net amount of flow that crosses the cut, i.e. moves through edges from S into T must be equal to the value of the flow $|f|$. This has to be true because if $|f|$ total units of flow are moving from s (contained in S) to t (contained in T), then they must cross through the cut (S,T). Given this, the value of the maximum flow cannot be greater than the capacity of any (S,T) cut, since it must travel through it, so in particular, the value of the maximum flow must be less than or equal to the capacity of the minimum cut.

Consider a maximum flow f and the particular s-t cut (S,T) such that S contains all vertices reachable from s in the residual graph G_f, and T contains all remaining vertices. (S,T) is a valid cut since t is not reachable from s, otherwise there would be an augmenting path, which would imply that the flow f was not in fact maximum. Suppose that the value of f was less than the capacity of (S,T), i.e. that $|f| < c(S,T)$, then either there is an edge crossing (S,T) that is not saturated, or there is an edge bringing flow from T into S. If there is an unsaturated edge (u,v) such that $u \in S$ and $v \in T$, then there is a path from S to T in the residual graph, which is a contradiction. Similarly, if there is an edge bringing flow from T into S, then the corresponding back-edge from S into T has non-zero residual capacity, and hence there is a path from S to T in the residual graph, which is also a contradiction. Therefore, the value of the flow f must be equal to the capacity of the cut (S,T).

Since the value of the maximum flow f cannot exceed the capacity of any cut, and we have found a cut such that $|f| = c(S,T)$, we can conclude that the value of the maximum flow is equal to the capacity of the minimum cut.

Not only does the proof above illuminate why the theorem is true, it also shows us how to construct a minimum cut if we have found a maximum flow. If $G = (V,E)$ is a network and f is the maximum flow, then the minimum cut (S,T) can be found by taking S equal to all of the vertices that are reachable from s via the residual graph G_f, and taking T as all remaining vertices. The proof shows that this will indeed be a minimum s-t cut in G. The min-cut max-flow theorem also gives us an easy tool to prove that the Ford-Fulkerson algorithm is correct.

Corollary: Correctness of Ford-Fulkerson

Ford-Fulkerson terminates when there are no augmenting paths left. Since each augmenting path increases the flow by an integer, and the maximum flow is finite, the algorithm eventually terminates. When there does not exist an augmenting path from s to t, the flow across the cut $(S = \{v : v \text{ reachable from } s \text{ in } G_f\},\ T = V \setminus S)$ is equal to the capacity of the cut. By the min-cut max-flow theorem, this flow is therefore maximum.

9.3 Bipartite Matching

Recall that a bipartite graph is one where the vertices can be separated into two disjoint subsets L and R such that every edge connects a vertex $u \in L$ to a vertex $v \in R$.

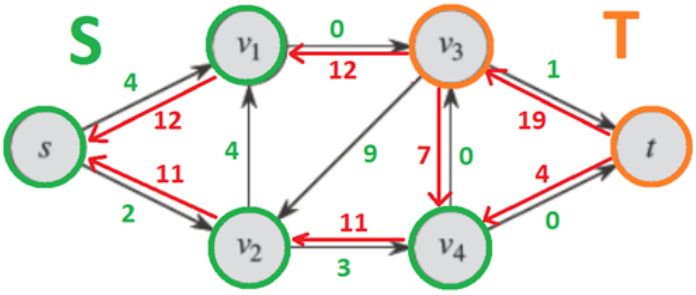


Figure 9.8: The residual network G_f for the maximum flow in the network in Figure 9.6. The vertices reachable from s are $\{s, v_1, v_2, v_4\}$, leaving $T = \{v_3, t\}$, across which, the capacity can be seen to equal to $12+7+4=23$, which agrees with the value of the maximum flow f. Image source: Adapted from CLRS

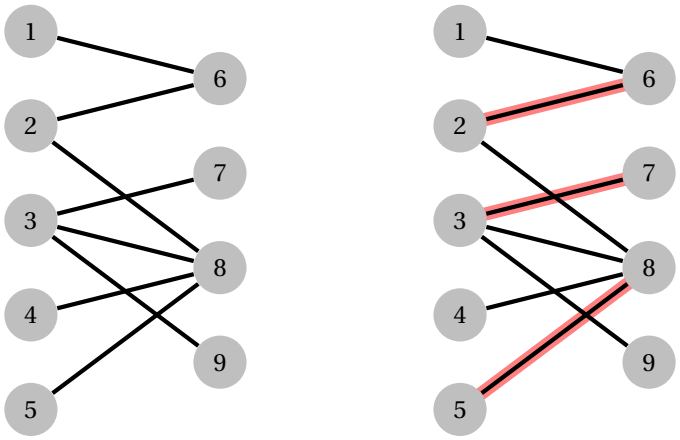


Figure 9.9: A bipartite graph composed of the two sides, $L = \{1,2,3,4,5\}$ and $R = \{6,7,8,9\}$, and a maximum matching which has size three.

Consider the example in Figure 9.9. Suppose that the set L represents job applicants, and the set R are available jobs. There is an edge between the applicant u and the job v if person u is qualified for job v. The bipartite matching problem asks us to match qualified applicants to jobs such that no applicant gets multiple jobs, and no job is taken by multiple people. For this example, the greatest number of applicants that can be successfully matched is three.

Problem Statement: Maximum bipartite matching

Given a bipartite graph $B = (V, E)$, a **matching** is a subset M of the edges E such that no vertex $v \in V$ is incident to multiple edges in M. The maximum bipartite matching problem seeks a matching of the graph with the largest possible number of edges.

Although the maximum bipartite matching problem does not sound very similar to the maxi-

mum flow problem, it can actually be solved by constructing a corresponding flow network and finding a maximum flow. The key idea is to use units of flow to represent matches. We construct the corresponding flow network by taking the bipartite graph $B = (V, E)$ and adding new source (s) and sink (t) vertices. We then connect the source vertex s to every vertex in the left subset L and connect every vertex in the right subset R to the sink vertex t, and direct all of the original edges in E to point from L to R. We then finally assign every edge a capacity of one.

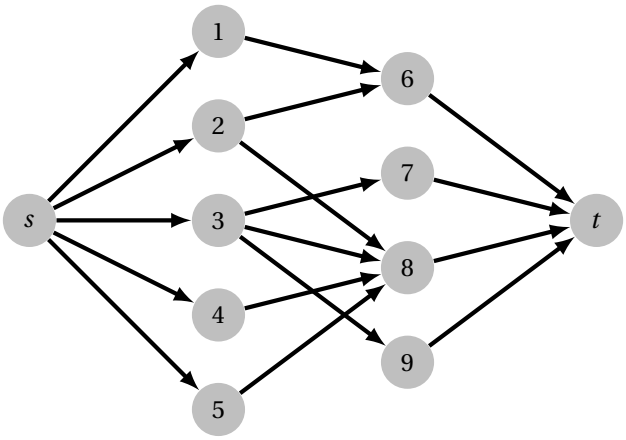


Figure 9.10: The corresponding flow network for the bipartite graph in Figure 9.9. Capacities are omitted since all edges have capacity 1.

Since every edge has a capacity of one, at most one unit of flow can be sent from s through any vertex $u \in L$. For an integer-value flow, this ensures that each applicant can get at most one job. Also, at most one unit of flow can leave any $v \in R$ into t, which ensures that each job gets at most one applicant. After finding a maximum flow, matches will correspond to edges from L to R that are saturated. The flow being a maximum flow ensures that the maximum possible number of matches are obtained.

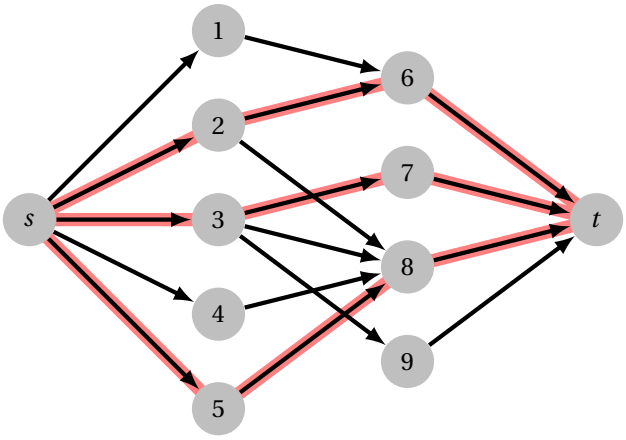


Figure 9.11: A maximum flow in the corresponding flow network. The highlighted edge are saturated with a flow of one, and form a maximum matching with size 3.

Given a bipartite graph $B = (V, E)$, the maximum possible number of matches cannot be greater than the number of vertices, so the value of the flow in the corresponding flow network is bounded above by $|f| \leq |V|$. Using the Ford-Fulkerson algorithm to find the maximum flow therefore results in a runtime of $O(|V||E|)$ to solve the maximum bipartite matching problem. A faster algorithm called Hopcroft-Karp solves the maximum bipartite matching problem in $O(\sqrt{|V|}|E|)$ by using the same flow network but a faster maximum flow algorithm.

Using Ford-Fulkerson to find maximum matchings is sometimes called the *alternating paths algorithm* for bipartite matching, since the augmenting paths that will be found by Ford-Fulkerson will strictly alternate between the left and right parts of the graph using forward edges and back edges sequentially. In practical implementations of the algorithm, constructing the flow network explicitly is not required, and much unnecessary code can be avoided by simply keeping track of the matches such that augmenting paths can be found by beginning at an arbitrary unmatched vertex and traversing the graph, alternating between unmatched vertices on the left and matched vertices on the right until an unmatched vertex is discovered on the right.

9.4 Circulations with Demands and Lower Bounds

The real power of network flow algorithms is that they allow us to efficiently solve nontrivial combinatorial problems that in principle have nothing to do with the flow of a material/data in a network. This can be done by reducing a combinatorial problem to either a max-flow or a min-cut problem in an appropriate directed graph. A first example was given in the last section, with the problem of bipartite matching.

It is often the case that these combinatorial problems can be tackled more easily by using two generalisations of max-flow problem that we will introduce in this section:

- Circulation with demands.
- Lower bounds on the flows of each edge.

We will show that these generalisations can be reduced to the standard max-flow problem, and therefore the Ford-Fulkerson method can still be used.

In the problem of circulation with demands, there is no single source or sink. Instead, all vertices can potentially have both incoming and outgoing edges, and each vertex u has an associated number d_u that denotes its condition:

- If $d_u > 0$, then the vertex u has a demand of d_u units of flow, i.e., it wants to receive in its incoming edges d_u units of flow more than what it sends out in its outgoing edges.
- If $d_u < 0$, then the vertex u has a supply of $-d_u$ units of flow, i.e., it wants to receive in its incoming edges d_u units of flow less than what it sends out in its outgoing edges.
- If $d_u = 0$, then the vertex u wants to keep the balance between the incoming and outgoing flow in its edges (similar to all non-sink non-source vertices in the standard max-flow problem).

The circulation with demands problem is to determine the feasibility of satisfying these constraints. A necessary condition for obtaining a feasible solution is that $\sum_{u \in V} d_u = 0$ (as otherwise considering the global situation there would be either an over-supply or over-demand), but this is not a sufficient condition. The problem is formally stated below.

Problem Statement: Circulation with demands problem

Let $G = (V, E)$ be a directed, edge-weighted graph. Denote the capacity (edge weight) of an edge by $c(u, v)$. We will assume for now that all capacities are integers. Associated with each vertex $u \in V$ there is an integer d_u that denotes its demand. A circulation with demands $\{d_u\}$ is a function f that maps each pair of vertices to a non-negative real number

$$f : V \times V \to \mathbb{R}^+$$

that satisfies the following conditions:

- **Capacity constraint:** The flow along an edge must not exceed the capacity of that edge. Formally,
$$0 \leq f(u, v) \leq c(u, v).$$
- **Demand constraint:** For each vertex $u \in V$
$$\sum_{v \in V} f(v, u) - \sum_{v \in V} f(u, v) = d_u$$
where $f(u, v) = 0$ if there is no edge (u, v), i.e. if $(u, v) \notin E$.

The circulation with demands problem is a feasibility problem concerned with the question of whether there is a solution satisfying both constraints or not.

Below we show an example of the circulation with demands problem in Figure 9.12, and a solution to this instance of the problem in Figure 9.13.

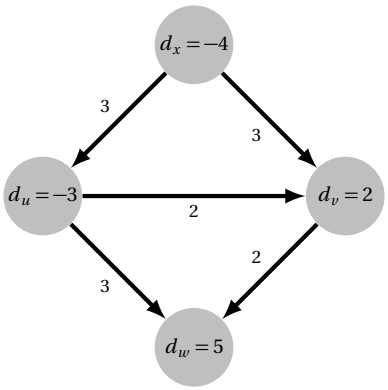


Figure 9.12: An instance of the circulation with demands problem. The demand is indicated in each vertex, and the capacity in each edge.

But how can we reduce the circulation with demands problem to a standard max-flow problem so that we can use Ford-Fulkerson to solve it?

The idea is to create a supergraph G' of G, solve a max-flow problem in G' to obtain a solution f'_{max}, and translate the solution f'_{max} of the max-flow problem in G' to a solution to the circulation with demands $\{d_u\}$ problem in G. The supergraph G' is created from G as follows:

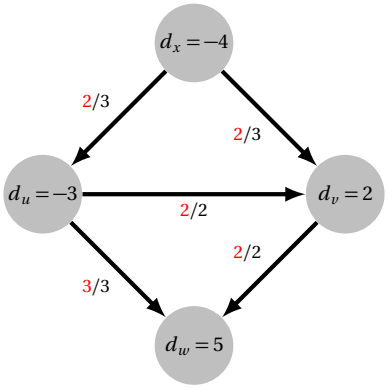


Figure 9.13: A solution for the circulation with demands instance above, with the flows along each edge indicated in red.

- Add a super-source vertex s.
- For each vertex u such that $d_u < 0$, add an edge (s, u) with capacity $-d_u$.
- Add a super-sink vertex t.
- For each vertex u such that $d_u > 0$, add an edge (u, t) with capacity d_u.

Figure 9.14 presents the supergraph G' obtained from G presented in Figure 9.12.

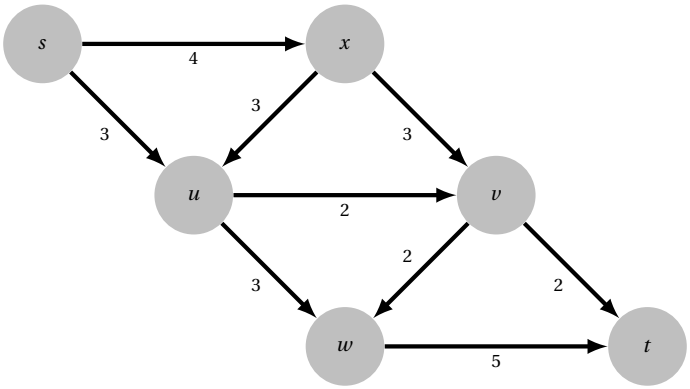


Figure 9.14: The supergraph G' obtained from G presented in Figure 9.12 so that a max-flow problem can be solved.

We now solve the max-flow problem on G' to obtain the maximum flow f'_{max} between s and t in G'. A solution of the max-flow problem in our example is presented in Figure 9.15.

Let $D = \sum_{u \in V, d_u > 0} d_u = \sum_{u \in V, d_u < 0} -d_u$. Remember that a necessary condition for the existence of a feasible solution f to the circulation with demands $\{d_u\}$ in G is that $\sum_{u \in V} d_u = 0$.

Note that $|f'_{max}|$ cannot be bigger than D, as the cut $C = (S, T)$ with $S = \{s\}$ has capacity equal to D by construction.

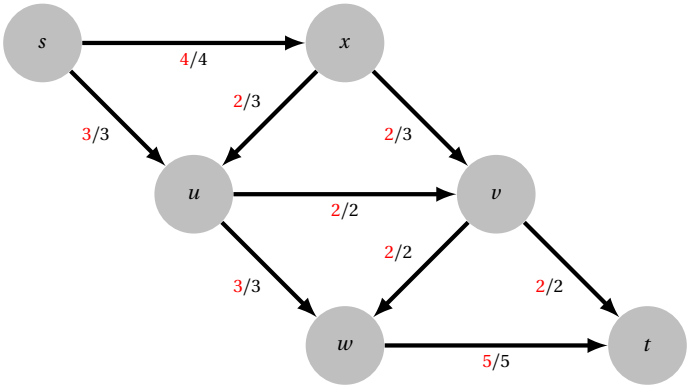


Figure 9.15: A solution of the max-flow problem presented in Figure 9.14.

If there is a feasible circulation f with demands $\{d_u\}$ in G, then by setting

$$\begin{array}{ll} f'(s,u) = -d_u, & \text{for all } u \text{ such that } d_u < 0; \\ f'(u,t) = d_u, & \text{for all } u \text{ such that } d_u > 0; \\ f'(u,v) = f(u,v), & \text{for all other edges;} \end{array}$$

we obtain a flow f' in G' of value D (and so a max-flow).

On the other hand, suppose that there is a (max-)flow f' in G' with value D, then it must be the case that all outgoing edges of s are saturated (as by design the sum of their capacities is D). Similarly, all incoming edges of t must be saturated. Therefore to obtain a feasible solution f to the circulation with demands $\{d_u\}$ in G, we simply need to delete these edges of f' (note that each vertex $v \neq s,t$ met the flow conservation constraint in f', so after deleting the mentioned edges, it will now meet the demand constraint $\sum_{v \in V} f(v,u) - \sum_{v \in V} f(u,v) = d_u$).

The method for solving the circulation with demands problem is presented below.

Algorithm 61 The method to solve the circulation with demands problem

1: **function** CIRCULATION($G = (V,E), \{d_u\}$)
2: If $\sum_{u \in V, d_u > 0} d_u \neq \sum_{u \in V, d_u < 0} -d_u$, **return** unfeasible.
3: Create the supergraph G' from G.
4: Solve the max-flow problem in G' using Ford-Fulkerson to obtain f'_{max}.
5: If $|f'_{max}| \neq \sum_{u \in V, d_u > 0} d_u$, **return** unfeasible.
6: Given f'_{max}, delete all outgoing edges of s and all incoming edges of t to obtain f.
7: **return** f

For many applications of network flow to combinatorial problems it is also useful to consider lower bounds on the edges (i.e., we will enforce that at least a specified amount of flow go through the edge). Therefore we introduce circulation with demands and lower bounds, as formally specified below.

Problem Statement: Circulation with demands and lower bounds problem

Let $G = (V, E)$ be a directed, edge-weighted graph. Denote the capacity of an edge by $c(u, v)$ and its lower bound by $\ell(u, v)$, where $0 \leq \ell(u, v) \leq c(u, v)$. We will assume for now that all capacities and lower bounds are integers. Associated with each vertex $u \in V$ there is an integer d_u that denotes its demand. A circulation with demands $\{d_u\}$ and lower bounds is a function f that maps each pair of vertices to a non-negative real number

$$f : V \times V \rightarrow \mathbb{R}^+$$

that satisfies the following conditions:

- **Capacity constraint:** The flow along an edge must be at least equal to its lower bound and must not exceed the capacity of that edge. Formally,

$$\ell(u, v) \leq f(u, v) \leq c(u, v).$$

- **Demand constraint:** For each vertex $u \in V$

$$\sum_{v \in V} f(v, u) - \sum_{v \in V} f(u, v) = d_u$$

where $f(u, v) = 0$ if there is no edge (u, v), i.e. if $(u, v) \notin E$.

The circulation with demands and lower bounds problem is a feasibility problem concerned with the question of whether there is a solution satisfying both constraints or not.

Let's go back to our previous example and now impose a lower bound of 1 on all edges, as depicted in Figure 9.16. From our previous solution to the circulation with demands problem, we already know that it is feasible to solve this example even when a lower bound of 1 is put on all edges, but how can the circulation with demands and lower bounds problem be solved in general?

The idea is that we will reduce this problem to the problem of circulation with demands but no lower bounds.

We first define the flow f^ℓ by setting, for each edge, $f^\ell(u, v) = \ell(u, v)$. And then we figure out if there is exists a flow f^* such that, for $f = f^\ell + f^*$, f is a feasible solution to the circulation with demands $\{d_u\}$ and lower bounds problem on G. Note that f^ℓ already takes care that the flow f is at least equal to the lower bound on each edge. Now we just have to consider the related graph G^* with adjusted capacities and demands:

- the vertices of G^* are the same as in G, but they now have demand $d^*_u = d_u - \sum_{v \in V} f^\ell(v, u) + \sum_{v \in V} f^\ell(u, v)$;
- the edges of G^* are the same as in G, but they now have capacities $c^*(u, v) = c(u, v) - \ell(u, v)$ and no lower bounds;

and solve the problem of circulation with demands $\{d^*_u\}$ - without lower bounds - in G^*.

Following our example, we now consider the flow f^ℓ in Figure 9.17, the adjusted graph G^* with

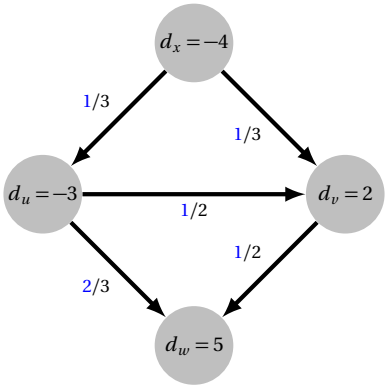


Figure 9.16: An instance of the circulation with demands and lower bound problem. The demand is indicated in each vertex, and for each edge the capacity is indicated in black and the lower bound in blue.

its solution f^* in Figure 9.18, and the final solution $f = f^\ell + f^*$ for the circulation with demands $\{d_u\}$ and lower bounds problem in G in Figure 9.19.

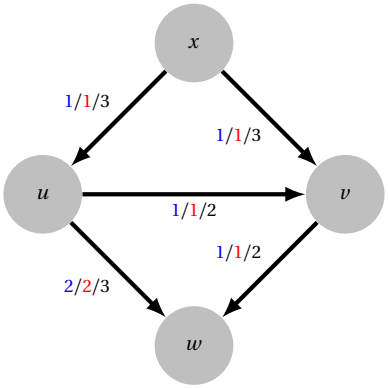


Figure 9.17: The flow f^ℓ (in red) corresponding to the circulation with demands and lower bound example in Figure 9.16.

Equipped with these extremely powerful tools, you will be able to solve many nontrivial combinatorial problems, as you can see from the examples given in the lectures and tutorials.

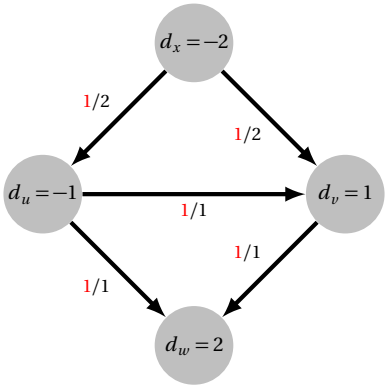


Figure 9.18: The adjusted graph G^* for the circulation with demands $\{d_u^*\}$ problem (without lower bounds) corresponding to the example in Figure 9.16, and its solution f^* in red.

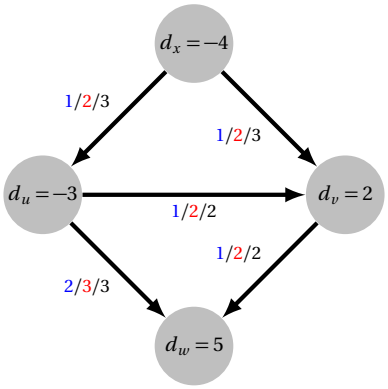


Figure 9.19: The final solution $f = f^\ell + f^*$ for the circulation with demands $\{d_u\}$ and lower bounds problem in G (lower bounds in blue, flow in red and capacities in black).

Chapter 10

Hashing and Hashtables

Storing and retrieving data is one of the most common scenarios in which we employ advanced data structures. We will first look at arguably one of the most well-known data structures in computer science, hashing. Hashing aims to solve the "dictionary" problem, where we need to store and search for items associated with keys. Using hashing, we can usually support these operations in just $O(1)$ expected time! We will briefly revise the basic hashing schemes that you should be familiar with, hashing with chaining and hashing with open addressing (probing). Then we will cover a selection of good hash functions for random integer keys and string keys, and briefly discuss the roll of randomness in produces good hashes. We will then discuss a more advanced hashtable strategy, cuckoo hashing.

Summary: Hashing and Hashtables

In this chapter, we cover:

- The dictionary problem and hashtables (Revision).
- Collision resolution strategies (Revision).
- Good hash functions for integers.
- Hash functions for strings.
- Universal hashing.
- Cuckoo hashing.

The Dictionary Problem

The dictionary problem is arguably the most frequently applicable problem in computer science. The dictionary abstract data type (ADT) supports the following operations:

1. INSERT(item): insert an item with a given key into the dictionary.
2. DELETE(item): delete an item with a given key from the dictionary.
3. LOOKUP(key): find the item with the given key if it exists.

Note that the items can contain other data in addition to the keys. Often the main data structure used to solve the dictionary problems will store only the keys together with links to the associated data. In this chapter we focus only on the keys to simplify the explanation. We assume that all items have unique keys, and that all keys come from a universe $\mathscr{U}$, typically integers

$\{0,1,...,u-1\}$. Non-integer keys should be converted to integers. We will see in the next chapter that this problem can be solved using balanced binary search trees in $O(\log(n))$ time per operation, but we want to do better than this. The goal of hashtables is to solve the dictionary problem in just $O(1)$ expected time per operation.

10.1 Hashtables (Revision)

Direct addressing

The simplest way to solve the dictionary problem is with *direct addressing*. A direct address table is a table of size $|\mathscr{U}|$, where we insert key k into slot k of the table. This allows for worst-case $O(1)$ insertion, deletion, and lookup! Of course, this scheme is only practical when the universe $\mathscr{U}$ is small, otherwise we will not have enough memory to store a direct address table.

Hashing

The solution to the prohibitive memory usage of direct addressing is to employ *hashing*. We reduce the universe $\mathscr{U}$ down to a size that we can manage, say integers $\{0,1,...,m-1\}$ and use a *hash function*

$$h : \mathscr{U} \to \{0,1,...,m-1\}$$

to map keys onto the reduced universe. Items are then stored in a table of size m, with key k being stored at position $h(k)$. This allows us to control the memory usage, but introduces a new problem. Since it is likely that $m \ll |\mathscr{U}|$, it will be the case that there are distinct pairs of keys k_i, k_j such that $h(k_i) = h(k_j)$, i.e. they will map to the same value. We call such a pair of keys a *collision*. We will explore several strategies for resolving collisions. First, we revise chaining and open addressing. Later, we will explore an entirely different strategy, Cuckoo hashing.

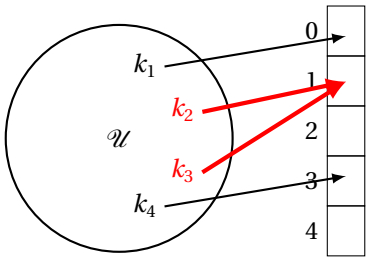


Figure 10.1: A hashtable with collisions

Chaining

Collision resolution via chaining involves maintaining a linked list at every position $0,1,...,m-1$ in the table and inserting all elements with $h(k) = i$ into the linked list at position i. If we assume that the hash function maps keys evenly to each slot in the hash table, then for a random set of keys, the expected number of items in each chain is n/m, where n is the number of items in the hashtable. We call the quantity $\alpha = n/m$ the *load factor*. Given that the expected chain length is α, the average-case time complexity for chaining is therefore $O(1+\alpha)$ per operation. If we keep $m \geq n$, which implies $\alpha \leq 1$, this yields average-case $O(1)$ performance per operation!

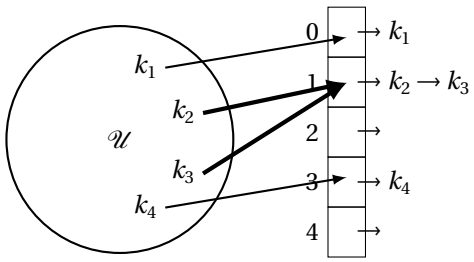


Figure 10.2: Hashing with chaining

Open Addressing (Probing)

Open addressing involves storing all of the items in the table directly. This means that we strictly need $m \geq n$, i.e. $\alpha \leq 1$. In open addressing, whenever a collision occurs, we seek an alternate location to place the key. We do so by *probing*. We adjust our hash function so that instead of suggesting a single location, it suggests a sequence of locations for us to try

$$h(k,i) : \mathscr{U} \times \mathbb{N} \to \{0,1,...,m-1\}.$$

Deletion must be done with caution when using open addressing. Simply removing an item from the table may make other items unreachable if they were placed there via probing. To account for this, we should mark deleted slots with a flag that insertion is free to ignore and overwrite, but lookup should interpret as an occupied slot.

If we once again assume that our hash functions maps keys evenly into the available table slots, and additionally assume that for every key, each of the possible $m!$ probe sequences are equally likely[1], then the expected number of operations performed by probing is $O\left(\frac{1}{1-\alpha}\right)$ since we have probability at least $\frac{m-n}{m}$ of finding an empty slot each probe, which implies an expected number of trials at most equal to

$$\frac{1}{\frac{m-n}{m}} = \frac{m}{m-n} = \frac{\frac{m}{m}}{\frac{m}{m} - \frac{n}{m}} = \frac{1}{1-\alpha}.$$

e.g. if $\alpha = 0.5$, we expect at most two probes, for $\alpha = 0.75$, four probes, etc. Keeping α below any fixed constant therefore implies that we can perform operations in $O(1)$ average-case time. In practice, this means that we must increase the size of the table and select a new hash function whenever the load factor exceeds this selected threshold. Common thresholds used are 0.25, 0.5, 2/3, and 0.75 depending on the probing method used.

We will look at three probing strategies that are commonly employed.

Linear Probing

In linear probing, we probe adjacent locations in the table until we find an empty slot:

$$h(k,i) = (h'(k) + i) \mod m,$$

where $h'(k)$ is the original hash function. In theory, linear probing performs quite poorly since it is quite susceptible to *primary clustering*. If many keys end up around the same location,

[1] This is a very unrealistic assumption, but it is commonly made since it makes for an easier analysis.

then it only further increases the probability that even more keys end up around that location. However in practice, linear probing is incredibly efficient due to cache locality.

Quadratic Probing

In quadratic probing, we follow the probe sequence:

$$h(k,i) = (h'(k) + c_1 i + c_2 i^2) \mod m,$$

where $h'(k)$ is the original hash function and c_1 and c_2 are chosen constants. Quadratic probing avoids the problem of primary clustering since keys jump further away, but we are still susceptible to *secondary clustering*, keys with the same hash values will always probe the same elements.

Double Hashing

Double hashing aims to avoid the problem of secondary clustering. In double hashing, we use a second hash function to determine the distance between probed elements.

$$h(k,i) = (h_1(k) + i\,h_2(k)) \mod m,$$

where $h_1(k)$ is the original hash function, and $h_2(k)$ is the secondary hash function. Although it eliminates secondary clustering, double hashing is sometimes less efficient in practice since two hash functions have to be computed

10.2 What is a Good Hash Function?

Good hash functions are those which minimise the probability of collisions occurring. This is very easy to say, but very difficult, and sometimes impossible to achieve in practice. Assuming that the keys to be hashed are uniformly distributed implies that an ideal hash function maps keys uniformly onto the range of potential hash values. For example, to hash keys which are floating point values in the range $0 \leq k \leq 1$, a hash function that maps onto the integers 0 to $m-1$ which satisfies this ideal is

$$h(k) = \lfloor k\,m \rfloor.$$

While this might be ideal under the assumption that the keys k are uniformly distributed in $0 \leq k \leq 1$, this is a terrible hash function if it turns out that 99% of the keys k turn out in practice to be very close together, as this hash function maps keys that are very close to the same hash value. In this scenario, an ideal hash function would be one that maps keys that are very close together to hash values that are very far apart.

What this really means is that unfortunately, it is not possible in general to design a hash function that is perfect for all situations. In fact in theory, the ideal hash function is actually the totally random hash function! A totally random hash function would map each potential input key to each potential slot in the hashtable with equal probability uniformly at random. Unfortunately, implementing a totally random hash function would use a prohibitive amount of time and memory to compute, since we would need to store a hash value for every possible input key in advance.

In the following sections, we will study some hash functions which are good under the assumption that the input keys are random. In practice, this is a bad assumption, since real-world data usually has biases! To account for this, we often introduce randomness into our hash functions, which we discuss later when we meet universal hashing.

How likely are collisions?

Although collisions might seem rare, they are unintuitively common. One way to intuit this is to consider the famous *birthday problem*.

Problem Statement: The Birthday problem

Given a group of n people, find the probability that there is at least one pair of people with the same birthday.

Very unintuitively, the probability reaches 50% at just 23 people. At 367 people, the probability is 100% since there are only 366 possible birthdays. If we consider the analogy with respect to a hash table, we have a table of size 366, and with just 23 keys to insert, it is already more likely than not that a collision will occur. In general, with n keys and a table of size m where $n < m$, the probability that there exists a colliding pair is

$$p(n,m) = 1 - \frac{m!}{(m-n)!m^n}$$

which rapidly approaches 1 as $n \to m$. In practice, the best we can do is to minimise the probability of a collision by designing a good hash function. It is not realistic to try to avoid collisions 100% of the time, so the design of good hashtable data structures is therefore often heavily focused not only on avoiding collisions, but handling them in an efficient way when they do inevitably occur.

For the mathematically inclined, the rigorous analysis of hash collisions and complexity bounds on hashtables is analysed using tools from probabilistic combinatorics. Interested students should read *Probability and Computing*, Mitzenmacher & Upfal.

10.3 Hashing Integers

The most common types of keys that we need to hash when implementing hashtables are integer types. Even when the types that we are dealing with might not be explicitly integers, there is often a simple and convenient way to interpret them as integers. For example, characters from a finite alphabet can be interpreted as integers corresponding to their position in the alphabet. A generalisation of this allows us to consider strings as integers in which we consider the characters as place values in some fixed radix system (covered in the next section).

The divisional method

The first and simplest method for producing a hash function for a set of integral keys onto a table of m slots is to simply take the value of the key mod m. That is, for each key k, the hash function is

$$h(k) = k \mod m.$$

This hash function produces hash values that are usually well distributed provided that m is selected well. In particular, one should take care to avoid using powers of two, since taking k mod 2^p simply extracts the p least significant bits of k. A good hash function should depend on the values of all of the bits in the key to increase the likelihood that the hashes are well distributed. Prime values of m, particularly those which are not close to a power of two are strong

choices since some number theory will tell us that consecutive multiples of a key will continuously result in different hash values provided that the key is not a multiple of m itself.

The multiplicative method

The multiplicative hash works as follows. We select a constant A where $0 < A < 1$ and take the fractional part of kA, which is $kA - \lfloor kA \rfloor$. We then scale m by the resulting value to produce a hash like so

$$h(k) = \lfloor m(kA - \lfloor kA \rfloor) \rfloor.$$

Unlike the divisional method, the value of m does not heavily influence the quality of the hash, so taking m as a power of two is fine and even preferable since the computation of the hash can then be sped up by taking advantage of bitwise operations.

The main decision to make is the value of A, whose optimal choice is heavily influenced by the characteristics of the keys being hashed. According to Knuth[2], a good choice is

$$A = \varphi^{-1} = \frac{\sqrt{5}-1}{2} \approx 0.61803...$$

10.4 Hashing Strings

The polynomial hashing technique is a common and effective method for hashing strings. Given a string $S[1..n]$, if we associate each character of S with a numeric value (its position in the alphabet for example), then the polynomial hash is given by

$$h(S) = ((S[1]x^{n-1} + S[2]x^{n-2} + ... + S[n-1]x + S[n]) \mod p) \mod m,$$

where p is a prime such that $p > m$. To reduce the chance of collisions, the value used for the base x should be greater than the maximum attainable value of $S[i]$. If this is the case, then it is easy to see that the value of the polynomial hash for any string ignoring the modulo is unique, suggesting that this is a good hash function. Evaluating the polynomial hash for a given string naively term-by-term will lead to a time complexity of $O(n^2)$ or $O(n \log(n))$ if fast exponentiation is used, but this can be easily improved by using *Horner's Method*.

[2]Donald E. Knuth. Sorting and Searching, volume 3 of *The Art of Computer Programming*. Addison-Wesley, 1973. Second edition, 1998

Definition: Horner's Method

Given the polynomial

$$p(x) = a_0 + a_1 x + a_2 x^2 + \ldots + a_n x^n$$

we can evaluate it in $O(n)$ time by computing the sequence of values

$$\begin{cases} b_n = a_n \\ b_{n-1} = a_{n-1} + b_n x \\ \vdots \\ b_0 = a_0 + b_1 x \end{cases}$$

where b_0 is the value of $p(x)$.

Horner's Method works because we can notice that a polynomial

$$p(x) = a_0 + a_1 x + a_2 x^2 + \ldots + a_n x^n$$

can be rewritten as

$$p(x) = a_0 + x(a_1 + x(a_2 + \ldots + x(a_{n-1} + a_n x))).$$

Horner's method then simply evaluates this expression from the innermost bracketed part outwards.

Rolling the Polynomial Hash Function

One of the brilliant properties of the polynomial hash function that makes it so useful is that if we know the hash value of a particular substring of a given string, then we can compute the hash value of an adjacent substring (one differing in just one character) in just $O(1)$ time. Suppose that we first pre-compute the value of x^{n-1} and we know the polynomial hash for some substring $S[i..j]$. Then the polynomial hash for the substring $S[i..j+1]$ is given by

$$h(S[i..j+1]) = ((x \times h(S[i..j]) + S[j+1]) \mod p) \mod m,$$

and the polynomial hash for the substring $S[i+1..j]$ can be computed as

$$h(S[i+1..j]) = ((h(S[i..j]) - S[i]x^{n-1}) \mod p) \mod m.$$

This allows us to "roll" the polynomial hash across a string and compute the hash values for all substrings of a fixed length in just linear time, rather than the quadratic time that would be required to do each substring separately.

10.5 Universal Hashing

All of our analysis above has assumed that our input keys are hashed uniformly at random to the table slots. This requires making among other things, the bold assumption that our input data is totally random. We should recall from our discussion of sorting that real world data is

never random, so such assumptions are incorrect and foolish. When discussing Quicksort and Quickselect, a remarkably simple solution to combat the fact that our input data is not random was to choose our pivots randomly. Indeed, the simplest and most effective way to combat the fact that real-world data is not random is to introduce randomness into our algorithms instead! This ensures that no adversary can produce input that will cause our algorithms to run at worst-case performance. An ideal hash function would be *totally random*, that is, it would satisfy for all $x \in \mathcal{U}$, independent of all other elements of $\mathcal{U}$,

$$\Pr(h(x) = t) = \frac{1}{m}.$$

Using a totally random hash function, chaining and linear probing both achieve $O(1)$ expected worst-case performance. Unfortunately, generating a totally random hash function would require us to store a random hash value for every possible input key $x \in \mathcal{U}$, which would use just as much space as direct addressing! Instead, we try to approximate totally random hashing with hash functions that are somewhat random but easy to compute. One such approach is *universal hashing*.

In universal hashing, we select our hash function h **randomly** from a *universal family* of hash functions. A family $\mathcal{H}$ is called universal if it satisfies

$$\Pr_{h \in \mathcal{H}}\{h(k) = h(k')\} \leq \frac{1}{m}, \qquad \text{for all } k \neq k' \in \mathcal{U}.$$

With universal hashing, the expected time per operation for chaining can be shown to still be $O(1+\alpha)$. Unfortunately, unlike totally random, universal hashing is not strong enough to guarantee that probing achieves $O(1)$ expected complexity. Here are three examples of universal families:

- The set $\mathcal{H}$ of all hash functions. This is clearly universal but it is completely useless since selecting from this family is equivalent to generating a totally random hash function which as we discussed is infeasible.
- $\mathcal{H} = \{h_{a,b}(k) = \big((ak+b) \mod p\big) \mod m\}$ for all $0 < a < p$, and all $0 \leq b < p$, for a fixed prime number such that $p \geq u$. This family has the disadvantage that table slots above $p \mod m$ may be less likely to be occupied. It is advantaged by the fact that it only needs to store three integers, a, b, and p and hence uses very little space.
- $\mathcal{H} = \{h_{a,b}(k) = ((ak+b) \mod u) >> (\log_2(u) - \log_2(m))\}$ for all odd $0 < a < u$, and all $0 \leq b < u/m$. This family works when u and m, the universe size and the table size, are powers of two. This hash function then amounts to simply taking the highest $\log_2(m)$ bits from the product $(ak+b) \mod u$, and is very fast to compute since when m and u are powers of two, no modular arithmetic is required, only bitwise operations and a single multiplication, which are much faster.

Even stronger families of hash functions exist that provide stronger mathematical guarantees, but we won't discuss these for now[3].

[3] If interested, you should watch this lecture from MIT OpenCourseware, `https://youtu.be/Mf9Nn9PbGsE`.

10.6 Cuckoo Hashing

Cuckoo hashing is a hashing scheme for hashtables that guarantees worst-case $O(1)$ lookup and deletion, while supporting $O(1)$ expected insertion assuming sufficiently good hash functions. The basic ideas are very simple. The tricky part is the analysis of whether insertion is fast.

Key Ideas: Cuckoo Hashing

1. Maintain two hashtables T_1 and T_2.
2. Use two hash functions $f(k)$ and $g(k)$.
3. If a collisions occurs, move the collided item to the other table.

To lookup a key k:

1. Look at $T_1[f(k)]$. If found, return it.
2. Look at $T_2[g(k)]$. If found, return it.
3. Otherwise, element not found.

Algorithm 62 Cuckoo hashing: Lookup

```
1: function LOOKUP(k)
2:     if T1[f(k)].key = k then return T1[f(k)]
3:     if T2[g(k)].key = k then return T2[g(k)]
4:     return null                                  // Not found
```

To delete an item with key k:

1. Look at $T_1[f(k)]$. If found, delete it.
2. Look at $T_2[g(k)]$. If found, delete it.
3. Otherwise, element not found.

Algorithm 63 Cuckoo hashing: Deletion

```
1: function DELETE(item)
2:     if T1[f(item.key)] = item then T1[f(item.key)] = null
3:     else if T2[g(item.key)] = item then T2[g(item.key)] = null
```

To insert a key k:

1. Attempt to insert the key k into T_1 at position $f(k)$.
2. If k collides with an existing key in T_1, then move $k' = T_1[f(k)]$ into T_2 at position $g(k')$ and insert key k into T_1 at position $f(k)$.
3. If moving k' into T_2 caused a collision, take $k'' = T_2[g(k')]$ and attempt to insert it into T_1.

4. Continue until successful or until too many attempts have been made.
5. If the process fails, rebuild the tables larger with new hash functions.

Algorithm 64 Cuckoo hashing: Insertion

```
1: function INSERT(item)
2:     if not LOOKUP(item.key) then                    // Ignore if the key is already present
3:         for i = 1 to MaxLoop do
4:             swap(item, T1[f(item.key)])
5:             if item = null then return
6:             swap(item, T2[g(item.key)])
7:             if item = null then return
8:         REHASH()                                    // Resize and start from scratch
9:         INSERT(item)                                // Try again
```

Lookup and deletion from a hashtable using cuckoo hashing clearly has a worst-case $O(1)$ complexity since it suffices to simply check two addresses. Analysis of insertion on the other hand is tricky. Selecting the appropriate threshold *MaxLoop* for insertion is important to ensure that rebuilds do not happen too frequently, but happen frequently enough such that insertions do not encounter too many probes. In theory, a good threshold to use is $MaxLoop = 6\log(n)$ with appropriate hash functions.

If the hash functions are assumed to be totally random and we maintain that the table sizes are $m \geq 2n$, then it can be shown that any given insertion will cause a failure with probability $O(1/n^2)$. Consequently, Cuckoo hashing will fail to insert n keys with probability $O(1/n)$, which implies that n keys can be inserted in expected $O(n)$ time. Unfortunately, Cuckoo hashing's downside is that it requires rather strong hash functions to achieve this complexity. Cuckoo hashing does not achieve $O(1)$ expected time with universal hashing, or even with some much stronger hash families[4].

[4]For the details and analysis, see Pagh & Rodler, Cuckoo Hashing, `http://www.it-c.dk/people/pagh/papers/cuckoo-jour.pdf` and/or this lecture from MIT OpenCourseware, `https://youtu.be/Mf9Nn9PbGsE`.

Chapter 11

Balanced Binary Search Trees

Storing and retrieving data is one of the most common scenarios in which we employ advanced data structures. We just covered hashtables which solve the dynamic dictionary problem in $O(1)$ expected time. You should already be familiar with binary search trees as a powerful alternative lookup data structure. Binary search trees are slower than hashtables in expectation, but they provide worst-case guarantees and can also support operations that hashtables are incapable of, like finding the elements closest to a given key. Ordinary binary search trees can, in the worst case degrade to $O(n)$ performance, so this chapter focuses on overcoming this weakness.

Summary: Balanced Binary Search Trees

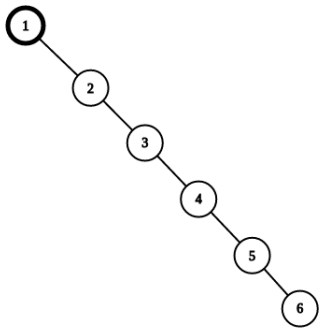
In this chapter, we cover:

- Binary search trees.
- AVL trees - self-balancing binary search trees.

11.1 AVL Trees

Recall from your previous studies that a binary search tree is a rooted binary tree such that for each node, the keys in its left subtree compare less than its own and the keys in its right subtree compare greater than its own. Binary search trees are an efficient data structure for storing and retrieving elements of an ordered set in $O(\log(n))$ time per operation. Pathological cases can occur however when the tree becomes imbalanced, where performance degrades in the worst case to linear time. We will study one variety of improved binary search trees called an AVL[1] tree which self-adjusts to prevent imbalance and keeps all operations running in guaranteed worst case $O(\log(n))$ time.

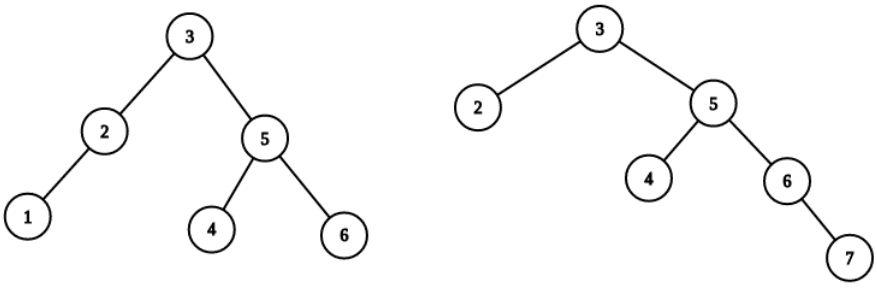
[1] AVL trees are named after Adelson, Velskii, and Landis, the original inventors of the data structure.



An example of a very imbalanced binary search tree resulting from inserting the keys $1,2,3,4,5,6$ in sorted order. Performance in such a tree is equivalent to a linked list with worst-case $O(n)$ lookup and insertion.

11.1.1 Definitions

In general, a tree is considered balanced if it has a height of $O(\log(n))$. There are many ways to achieve this with different kinds of trees. AVL trees use a very strict definition of balance which they maintain throughout insertions and deletions. In the context of an AVL tree, a tree is considered to be balanced if for any node in the tree, the heights of their left and right subtrees differ by at most one. This implies that its height is worst-case $O(\log(n))$, as required. Since lookup, insertion and deletion all take time proportional to the height of the tree, the tree being balanced implies that lookup, insertion and deletion can all be performed in worst-case $O(\log(n))$ time.



(a) Balanced

(b) Imbalanced

A balanced binary search tree (a) and an imbalanced binary search tree (b). (b) is imbalanced because the heights of the root node's left and right children are 1 and 3 respectively.

11.1.2 Rebalancing

AVL trees maintain balance by performing *rotations* whenever an insertion or deletion operation violates the balance property. Rotations move some of the elements of the tree around in

order to restore balance. We define the *balance factor* of a node to be the difference between the heights of its left and right subtrees, i.e.

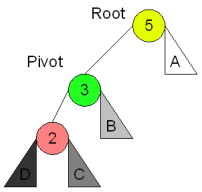
$$\text{balance\_factor(u)} = \text{height(u.left) - height(u.right)},$$

where height(**null**) = 0. A tree is therefore imbalanced if there is a node u such that

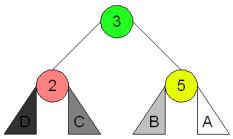
$$|\text{balance\_factor}(u)| \geq 2$$

There are four separate cases that can occur when a tree is imbalanced.

Left-left imbalance



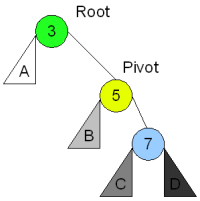
A left-left imbalance occurs when a node's balance factor is 2 (its left subtree is taller than its right subtree) and the left node's left subtree is at least as tall as the left node's right subtree (in other words, the balance factor of the left node is non-negative). To remedy a left-left imbalance, we perform a rightwards rotation, in which the left node becomes the new root, and the children are shifted accordingly. (Image source: Wikimedia Commons)



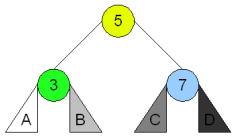
Algorithm 65 AVL tree: Right rotation

```
1: function ROTATE_RIGHT(root)
2:     par = root.parent
3:     pivot = root.left
4:     temp = pivot.right
5:     pivot.set_right_child(root)
6:     root.set_left_child(temp)
7:     par.swap_child(root, pivot)
```

Right-right imbalance



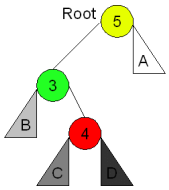
A right-right imbalance occurs when a node's balance factor is −2 (its right subtree is taller than its left subtree) and the right node's right subtree is at least as tall as the right node's left subtree (in other words, the balance factor of the right node is non-positive). To remedy a right-right imbalance, we perform a leftwards rotation, in which the right node becomes the new root, and the children are shifted accordingly. (Image source: Wikimedia Commons)



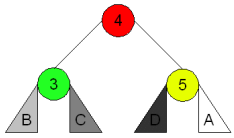
Algorithm 66 AVL tree: Left rotation

```
1: function ROTATE_LEFT(root)
2:     par = root.parent
3:     pivot = root.right
4:     temp = pivot.left
5:     pivot.set_left_child(root)
6:     root.set_right_child(temp)
7:     par.swap_child(root, pivot)
```

Left-right imbalance



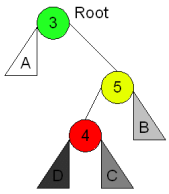
A left-right imbalance occurs when a node's balance factor is 2 (its left subtree is taller than its right subtree) and the left node's right subtree is taller than the left node's left subtree (in other words, the balance factor of the left node is negative). To remedy a left-right imbalance, we first perform a leftward rotation on the left node, which converts the imbalance into the left-left situation. We then perform a rightward rotation on the root to balance it. (Image source: Wikimedia Commons)



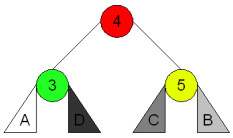
Algorithm 67 AVL tree: Double-right rotation

```
1: function DOUBLE_ROTATE_RIGHT(root)
2:     rotate_left(root.left)
3:     rotate_right(root)
```

Right-left imbalance



A right-left imbalance occurs when a node's balance factor is −2 (its right subtree is taller than its left subtree) and the right node's left subtree is taller than the right node's right subtree (in other words, the balance factor of the right node is positive). To remedy a right-left imbalance, we first perform a rightward rotation on the right node, which converts the imbalance into the right-right situation. We then perform a leftward rotation on the root node to balance it. (Image source: Wikimedia Commons)



Algorithm 68 AVL tree: Double-left rotation

```
1: function DOUBLE_ROTATE_LEFT(root)
2:     rotate_right(root.right)
3:     rotate_left(root)
```

Note that in the above algorithms, it is crucial that the functions for manipulating the children also correctly update the node's parent pointers! Failing to do so is a very common bug that programmers encounter when attempting to implement self-balancing binary search trees. We should also make sure that we handle the **null** cases correctly when the children or the parent nodes are null. In implementations of self-balancing binary search trees, it is common to use special dummy nodes to represent null nodes rather than using the language's actual null pointer to avoid having to include special cases all throughout the code. We also have to store and maintain the heights of each node in the tree, since computing them online would be too slow.

Combining each of these together, the entire rebalancing procedure can be written as in Algorithm 69. After performing any modification to the tree, we should call rebalance on all ancestors of the modified nodes, starting with the deepest ones first. It is important to rebalance the deepest nodes first since fixing an imbalance at a low level of the tree may correct an imbalance higher up in the tree, eliminating the need for redundant rotations.

Since the tree is assumed to be balanced before we modify it, the heights of at most $O(\log(n))$ nodes are affected by any one modification operation and hence we require at most $O(\log(n))$ rebalances, each of which can be executed in constant time. Therefore the total time complexity of insertion and deletion for an AVL tree is worst-case $O(\log(n))$. Since the tree is kept balanced by the rebalances, lookup in the tree is also worst-case $O(\log(n))$.

Algorithm 69 AVL tree: Rebalance

```
1:  function REBALANCE(node)
2:      if balance_factor(node) = 2 then
3:          if balance_factor(node.left) < 0 then
4:              DOUBLE_ROTATE_RIGHT(node)
5:          else
6:              ROTATE_RIGHT(node)
7:      else if balance_factor(node) = -2 then
8:          if balance_factor(node.right) > 0 then
9:              DOUBLE_ROTATE_LEFT(node)
10:         else
11:             ROTATE_LEFT(node)
```

Chapter 12

Prefix Tries and Suffix Trees

Today, huge amounts of the world's data is in the form of text data or data that can be interpreted as textual data. From classic literature, your favourite algorithms and data structures textbook to DNA sequences, text data is everywhere, and it needs to be stored, processed and analysed. We will introduce and study one special data structure for working with text strings, the suffix tree, as well as a related data structure, the prefix trie / retrieval tree data structure which allows for fast storage and lookup of strings.

Summary: Prefix Tries and Suffix Trees

In this chapter, we cover:

- Retrieval Trees / Prefix Tries.
- Suffix trees.
- Applications of suffix trees.

String Terminology

Recall that a *string* $S[1..n]$ is a sequence of characters, i.e. some textual data.

1. A *substring* of S is a string $S[i..j]$ where $1 \leq i \leq j \leq n$.
2. A *prefix* of S is a substring $S[1..j]$ where $1 \leq j \leq n$.
3. A *suffix* of S is a substring $S[i..n]$ where $1 \leq i \leq n$.

A useful observation to make is that a substring of S is always a prefix of some suffix of S.

12.1 The Prefix Trie / Retrieval Tree Data Structure

A retrieval tree or prefix trie is a data structure that stores a set of strings arranged in a tree such that words with a shared prefix are contained within the same subtree.

Key Ideas: Prefix Trie

- The prefix trie is a rooted tree (not necessarily binary).
- Each edge of the tree is labelled with a character. Alternatively, each node (except the root) can be labelled, which is equivalent to labelling its parent edge.

- A path from the root to a node in the trie corresponds to a prefix of a word in the trie.
- A path from the root to an internal node of the trie corresponds to a common prefix of multiple strings.

Prefix tries can be implemented in a variety of ways. The most important decision is how to index the children of a particular node. There are a several strategies, and we will consider three of them. We will denote the length of a query string by n, the total length of all words in the trie by T, and the alphabet by Σ.

- Use an array to store a child pointer for each character in the alphabet:

  This method allows us to perform lookup and insert in $O(n)$ time since we can follow each child pointer in constant time. However, the space requirement is quite high, since we are storing a lot of pointers to null children. Specifically, we will use $O(|\Sigma|T)$ space in the worse case, since every node has a pointer for every character in the alphabet.

- Use a balanced binary search tree for storing child pointers:

  This method uses the minimal amount of space, as we only need to store pointers to children that actually exist. For large alphabets, this is advantageous. However, we lose in lookup time since it now requires up to $O(n\log(|\Sigma|))$ time per lookup since at each node we must search a BST to find the child pointer. The total space requirement however is minimal, at just $O(T)$.

- Use a hashtable for storing child pointers:

  This method allows us to only store pointers to the children that exist, hence we will only use the minimal $O(T)$ space. It also allows lookup in $O(n)$ expected time, since at each node we perform a hashtable lookup in expected constant time. This method seems to be superior to the first two since it has optimal speed and optimal space usage. However, this implementation is less versatile, and prohibits us from doing some more advanced tricks with the trie. For example, you might want to implement the ability to lookup the alphabetically closest word to a given word. This is possible with the first two approaches, but not (efficiently) with a hashtable since there is no way in a hashtable to quickly lookup the alphabetically nearest key. It also does not provide deterministic worst-case guarantees which might be important to us.

It is common to terminate each word with a special character to allow us to distinguish between full words and prefixes. This special character is usually denoted by \$. This ensures that a node is a leaf if and only if it is the end of a word. In actual code, you should use the null character (`'\0'` in ASCII) rather than an actual \$ since \$ may be a character in one of the strings.

12.1.1 Applications of Tries

Prefix matching / exact string lookup

Prefix tries are designed to solve the prefix matching problem, i.e. determining whether a list of strings contains a word that begins with a particular prefix. If we terminate all words with a $\$ \notin \Sigma$, then we also gain the ability to do exact searches by appending \$ to our query strings, since \$

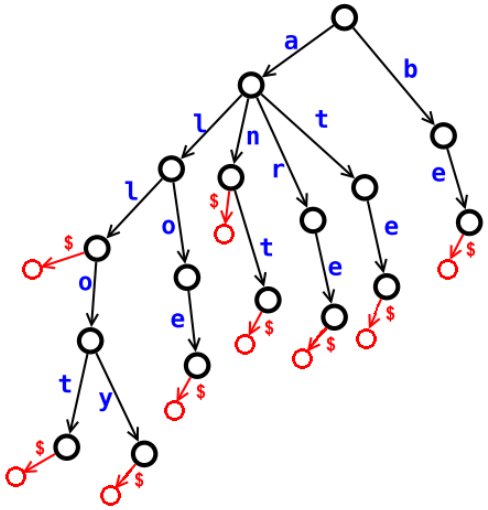


Figure 12.1: An example of a prefix trie containing the strings `be`, `ant`, `alloy`, `ate`, `are`, `aloe`, `an`, `allot`, `all`, with terminating $ characters added to indicate the ends of words.

will not be contained in any proper prefix of a word. This makes tries a potential substitute for a hashtable for storing a dictionary of strings. Implementations of insertion and lookup for prefix tries are shown in Algorithms 70 and 71 respectively. If we use the array-based implementation, building a prefix trie takes $O(|\Sigma|T)$ time, and lookup takes $O(n)$ time, which are optimal if $|\Sigma|$ is constant.

Algorithm 70 Prefix trie: Insertion

```
1: function INSERT(S[1..n])
2:     node = root
3:     for each character c in S[1..n] do
4:         if node has an edge for character c then
5:             node = node.get_child(c)
6:         else
7:             node = node.create_child(c)
```

Sorting a list of strings

We can use a prefix trie to quickly sort a list of strings coming from a fixed size alphabet. Simply insert all of the strings into a prefix trie, and then traverse the trie in lexicographical order. Using the array-based implementation, the complexity of this algorithm is $O(|\Sigma|T)$. If we used the balanced-binary-search-tree-based implementation instead, we could reduce the complexity to $O(T\log(|\Sigma|))$. Assuming that all of the input strings are roughly the same length, the complexity of sorting the strings using a typical fast sorting algorithm would instead be

$$O(n\log(n)) \times O\left(\frac{T}{n}\right) = O(T\log(n)),$$

Algorithm 71 Prefix trie: Lookup

```
1: // Returns True if the trie contains a word that has S as a prefix
2: function LOOKUP(S[1..n])
3:     node = root
4:     for each character c in S[1..n] do
5:         if node has an edge for character c then
6:             node = node.get_child(c)
7:         else
8:             return false
9:     return true
```

where n is the number of words in the list being sorted. We can therefore see that if the alphabet size is constant or sufficiently small, the prefix trie method will be faster. Lastly, note that this can actually be interpreted as a form of radix sort, specifically a most significant digit (MSD) radix sort, since strings are divided up based on their characters, first to last as we go down the tree. An implementation of sorting using a trie is given in Algorithm 72.

Algorithm 72 Prefix trie: String sorting

```
1: function SORT_STRINGS(strings[1..n])
2:     for each string s in strings[1..n] do
3:         INSERT(s + '$')
4:     TRAVERSE(root, "")
5:
6: function TRAVERSE(node, cur_string)
7:     if cur_strings ends with '$' then                    // We are at the end of a string
8:         print (cur_string)                               // Can omit printing the '$' character
9:     else
10:        for each child character c of node in alphabetical order do
11:            cur_string.append(c)
12:            TRAVERSE(node.get_child(c), cur_string)
13:            cur_string.pop_back()                        // Remove the last character of cur_string
```

Compact Tries

One major drawback to tries that we have observed is that they can potentially waste a lot of memory, particularly if they contain long non-branching paths (paths where each node only has one child). In cases like these, we can make prefix tries more efficient by combining the edges along a non-branching path into a single edge. The *Radix Trie* and the *PATRICIA Trie* are particular kinds of compact prefix tries. Figure 12.2 shows an example of a compact trie.

12.2 Suffix Trees

A suffix tree is a compact trie containing all of the suffixes of a string. We motivate suffix trees as a solution to the *pattern matching* problem.

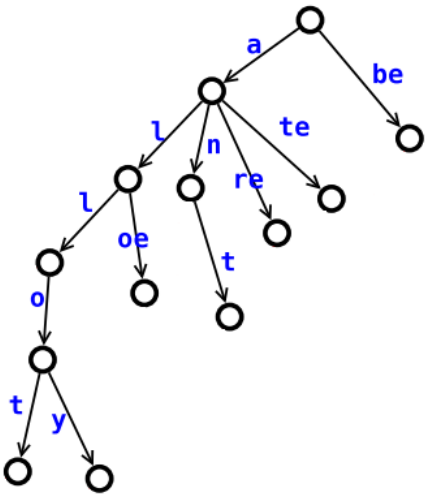


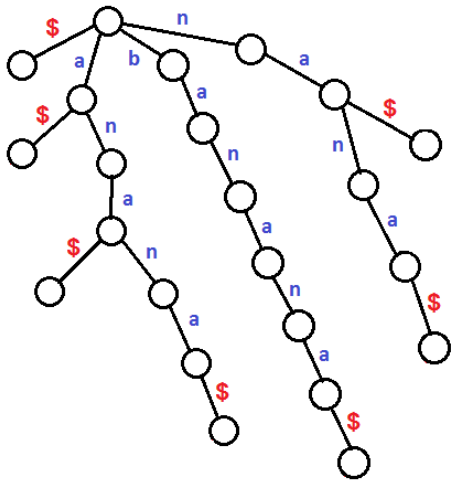
Figure 12.2: A compact trie containing `be`, `ant`, `alloy`, `ate`, `are`, `aloe`, `an`, `allot`, `all`. No terminating character $ has been added in this example.

Problem Statement: Pattern matching

Given a text string $T[1..n]$ and a pattern $P[1..m]$, find all occurrences of P as a substring of T.

Many well known algorithms exist that will solve the pattern matching problem in $O(n+m)$ per query, but what if we wish to keep the same text string T and search for lots of small patterns? Doing the $O(n)$ work per query might turn out to be extremely wasteful if the text string has length $n = 1,000,000$ and the patterns only have length $m = 20$.

We can start with a *suffix trie* by inserting all of the suffixes into a prefix trie. In many applications, we need to be able to distinguish suffixes from substrings so we add the terminating character $.



A suffix trie for the string “banana$”.

A suffix trie of the string T can be used to solve the pattern matching problem by checking whether P is a prefix in the trie. This will take $O(m)$ time per query, which is optimal. Since a string of length n has n suffixes, and each has length $O(n)$, a suffix trie will take $O(n^2)$ space, which is extremely inefficient. This is why the suffix tree is stored as a compact trie, which will take just $O(n)$ space. A suffix tree of the string `banana$` is shown in Figure 12.3.

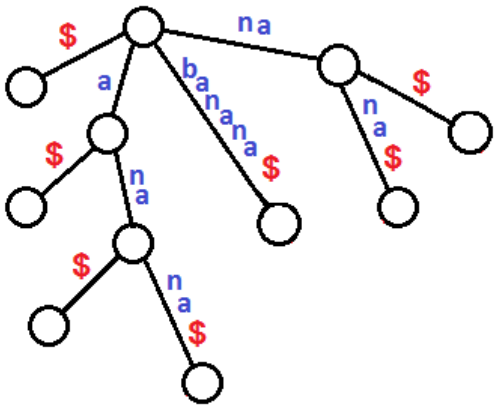


Figure 12.3: A suffix tree for the string “banana$”.

It is important to note that if we store all of the edge labels explicitly then the memory used by the suffix tree will still be $O(n^2)$. In order to mitigate this and bring the memory required down to $O(n)$, we instead simply refer to each substring by its position in the original string. For example, the substring “na” in “banana$” would simply be represented as $[3,4]$, meaning that it is the substring spanning the positions 3 to 4 in the string “banana$”.

12.2.1 Building a Suffix Tree

The naive approach

The simplest way to build a suffix tree is to build the suffix trie in $O(n^2)$ and then compress it into a suffix tree. This is simple to implement but is unfortunately useless in practice due to the poor time complexity.

Ukkonen's algorithm (Not examinable in Semester Two, 2024)

A very elegant algorithm for constructing suffix trees was given by Ukkonen[1], who produced an algorithm that was not only $O(n)$ time but also **online**, meaning that you can continue to add characters to the string while updating the suffix tree without having to start over from scratch. In its essence, Ukkonen's algorithm works by extending the length of each leaf edge of the suffix tree by one for each new character inserted, and appropriately splitting existing edges into two whenever a common prefix diverges.

12.2.2 Applications of Suffix Trees

Pattern matching

Since we could use the suffix trie to perform pattern matching in $O(m)$ time, we can also use the suffix tree for pattern matching in exactly the same way. The implementation is a little more involved since traversing the suffix tree along the edges requires more work (to check that each character along the edge is a match), but the idea and complexity are the same.

The longest repeated substring problem

Problem Statement: Longest repeated substring

Given a string $S[1..n]$, find the longest substring of S that occurs at least two times.

To solve this problem, we recall that within a prefix trie and equivalently a suffix tree, internal nodes correspond precisely to common prefixes, and hence in the case of suffix trees, substrings that occur multiple times. Finding the longest repeated substring is therefore simply a matter of traversing the suffix tree and looking for the deepest internal node (non-leaf node). An example is depicted in Figure 12.4.

[1] If interested, see E Ukkonen, On-line construction of suffix trees, *Algorithmica* 1995, and this Stackoverflow post `http://stackoverflow.com/questions/9452701` which describes the algorithm in a very accessible way.

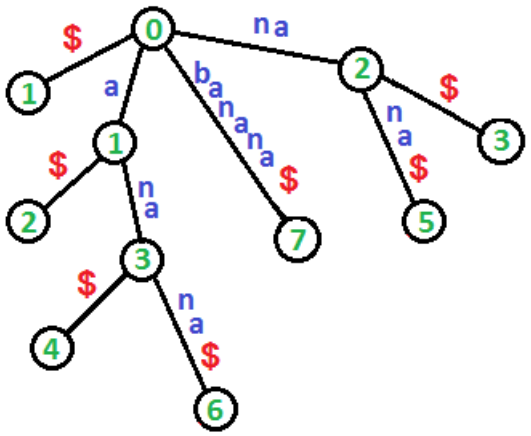


Figure 12.4: A suffix tree for the string “banana$” where each node is labelled with its depth (distance from the root node as measured by the number of characters on each edge). The deepest internal node has depth 3, which corresponds to the substring ”ana.”

Chapter 13

Suffix Arrays

We just saw the suffix tree data structure, a compact structure for processing and answering questions about strings. In addition to the suffix tree, there are many other data structures that utilise the suffixes of a string to perform efficient string related queries. We will study one more such structure as a more space efficient alternative to the suffix tree, the suffix array.

Summary: Suffix Arrays

In this chapter, we cover:

- Suffix arrays.
- How to build a suffix array.
- Applications of suffix arrays.

Suffix Arrays

Suffix arrays are a compact data structure that index all of the suffixes of a particular string in sorted order. While this may not sound immediately useful at first sight, the suffix array has a mountain of applications in string processing which carry over to applications in fields ranging from the study of natural languages to bioinformatics.

Consider as a first example, the string `banana`. Recall that when working with suffixes, we often mark the end of the string with a special character, denoted by $. The suffixes of `banana$` are then given by

```
banana$
anana$
nana$
ana$
na$
a$
$
```

In general, we can see that a string of length n has $n+1$ suffixes (including the empty suffix containing only $.) The suffix array of a string is a sorted array of its suffixes, so the suffix array of banana looks like

```
$
a$
ana$
anana$
banana$
na$
nana$
```

Note that the special character ($) is considered to be lexicographically less than all other suffixes, so it appears at the beginning of the suffix array. This is useful since it conceptually signifies the *empty string* (the suffix of length zero of the original string.)

Since actually writing down all of the suffixes of a string would take $O(n^2)$ space, suffix arrays are represented in a more compact form, where each element simply refers to an index $1 \leq i \leq n$ into S at which the corresponding suffix begins. For example, the suffix array for `banana` would be stored as:

$$\text{SA("banana\$")} = [7,6,4,2,1,5,3]$$

where the i^{th} index corresponds to the position in `banana` at which the i^{th} sorted suffix starts as shown in the table bellow.

Index	**Suffix**
7	`$`
6	`a$`
4	`ana$`
2	`anana$`
1	`banana$`
5	`na$`
3	`nana$`

In practice, the first entry (7 in this case) is sometimes omitted since it is guaranteed that the empty suffix is always the first one.

13.1 Building a Suffix Array

13.1.1 The Naive Approach

The simplest and most obvious way to build a suffix array is to simply build a list containing the indices $1..n$ and to sort them by comparing suffixes. Although short and simple, this implementation is useless in practice since its time complexity is $O(n^2 \log(n))$, owing to the fact that sorting performs $O(n \log(n))$ comparisons and each pair of suffixes takes $O(n)$ time to compare, which means that this algorithm can not be used on large strings. An implementation is show in Algorithm 73.

Algorithm 73 Naive suffix array construction

```
1: function SUFFIX_ARRAY(S[1..n])
2:     SA[1..n] = [1..n]
3:     sort(SA[1..n], SUFFIX_COMPARE(S,...)) // Use suffix_compare as the comparison operator
4:     return SA
5:
6: // Compare the suffixes at position i and j
7: function SUFFIX_COMPARE(S[1..n], i, j)
8:     return (S[i..n] < S[j..n])
```

13.1.2 The Prefix Doubling Algorithm

The naive algorithm for constructing suffix arrays is simply too slow to work on large strings. A much faster approach which is still conceptually simple is the *prefix doubling* algorithm:

Key Ideas: Prefix Doubling Algorithm

To construct the suffix array of the string $S[1..n]$

1. Sort the suffixes by their first character.
2. Sort the suffixes by their first two characters.
3. Sort the suffixes by their first four characters.
4. ...
5. Sort the suffixes by their first k characters (k = a power of two).

Note: If a suffix beginning at a particular position has length less than k at a particular iteration, it is considered to have empty characters coming after it, remembering that the empty character compares less than all others, e.g. "cat" < "cathode."

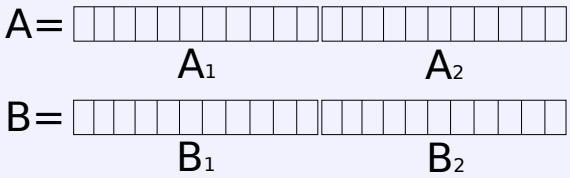
What we are really doing here is sorting consecutively larger prefixes of the suffixes, hence the name prefix doubling. If we perform prefix doubling naively by simply comparing substrings at each iteration, this will be no faster than the naive algorithm (in fact it will be even slower!) We therefore need a trick to perform the comparisons faster.

Fast suffix comparison

Prefix doubling iteratively sorts longer and longer substrings each round. The key idea is that by knowing the sorted order of the shorter substrings, we can compare the longer substrings fast!

Key Ideas: Fast suffix comparison

Suppose we wish to compare two strings A and B, each of which has the same even length. We can imagine these two strings as being composed of two halves. Call these halves A_1, A_2 and B_1, B_2.



When we compare A and B naively, we are comparing them from character 1 through to character l in order until we hit a pair of letters that differ. What if we already know that $A_1 < B_1$? This means that the first pair of characters that differ between A and B occurs in the first half in favour of A, and therefore $A < B$. We don't even have to look at the second halves. If instead we knew that $A_1 > B_1$, then we would similarly know that $A > B$.

If instead $A_1 = B_1$, then there is no point comparing the characters in the first halves of A and B since we already know that they must be the same. We can then apply the same logic to A_2 and B_2 and deduce that if $A_2 < B_2$ then $A < B$, or if $A_2 > B_2$ that $A > B$, or finally if $A_2 = B_2$ that $A = B$. What this tells us is that knowing the order of the halves of the strings we are comparing allows us to skip comparing all of the characters. In fact, all we had to do was compare the halves, so we only had to do at most 2 comparisons, regardless of the lengths of A and B! This means we can perform comparisons in $O(1)$ if we already know the sorted order of the halves.

For example, say we wish to compare the long strings "supercalafragalisticexpialadoshs" and "supercalafragalisticsuffixarrays". Comparing them naively, we would take until character 21 to find the answer. However, if we were to look at the strings as made up of two halves:

```
''supercalafragali''+''sticexpialadoshs''
''supercalafragali''+''sticsuffixarrays''
```

and I were to tell you that the first halves are the same, and that the second half of the first string "sticexpialadoshs" is less than the second half of the second string "sticsuffixarrays", you could immediately tell me that the first string is less than the second.

This explains why prefix doubling sorts the first 1 character, then 2 characters, then 4 and so on. What is important to realise here is that if I am looking at the first $2k$ characters of the suffix at position p, then what I really have are two halves: the first k characters of the suffix at position p and the first k characters of the suffix at position $p+k$. At each iteration, knowing the order of the previous halves allows all of the comparisons of the current suffixes to be performed in $O(1)$ time.

Maintaining the sorted order

The last issue that we must discuss before presenting the entire prefix doubling algorithm is keeping track of the sorted order of the suffixes. We know that the key piece of information we need is to be able to compare two halves quickly. If we have all of the suffixes partially sorted, then we can compare two of them by searching the array and seeing which one comes first, but this would take $O(n)$ time and negate all of the hard work that prefix doubling is doing for us. Instead, we need to maintain a second array, the *rank* array. The rank array remembers for each

suffix, its current position in the suffix array. Two suffixes that are equal up to the current length must be assigned the same rank so that we know that they are equal. For example, say we are computing the suffix array of "banana" and we have already sorted on the first two characters of each suffix and are now about to compare on the first four characters of each suffix. The partially sorted suffixes and their ranks are:

Index	Suffix	Rank
7	**$**	1
6	**a$**	2
2	**an**ana$	3
4	**an**a$	3
1	**ba**nana$	4
3	**na**na$	5
5	**na**$	5

If we know the ranks of two suffixes, we can compare them by simply comparing their ranks, since a lower rank means it comes earlier in the suffix array. We now have all of the ingredients to implement prefix doubling.

Implementation of Prefix Doubling

An implementation of prefix doubling is given in Algorithm 74. The value `ord(c)` simply refers to the order or rank of the character `c` in the alphabet. e.g. `ord('a') = 1`, `ord('b') = 2`, etc. The SUFFIX_COMPARE function compares two prefixes of length k in $O(1)$ by looking at the ranks of the two halves of the prefix, i.e. it compares by the pairs $(\text{rank}[i], \text{rank}[i+k])$. After sorting, we compute the new ranks by going through the suffixes in the current order, and adding 1 to the rank if the current suffix is greater than the previous one. This ensures that we correctly assign equal ranks to suffixes that are currently equal. Sorting at each iteration of the algorithm takes $O(n\log(n))$ time since the comparisons are now performed in $O(1)$, and we have to perform $\log(n)$ iterations to fully sort the suffixes, so the total time complexity of the prefix doubling algorithm is $O(n\log^2(n))$.

13.1.3 Faster Suffix Array Algorithms

Improved prefix doubling with radix sort

The prefix doubling algorithm is already significantly faster than the naive suffix array construction method, but there is still lots of room for improvement. Since the ranks are bounded above by n, we can speed up prefix doubling by replacing the $O(n\log(n))$ sort with an $O(n)$ radix sort instead. To take advantage of radix sort, we interpret prefix doubling as sorting suffixes using the pairs $(\text{rank}[i], \text{rank}[i+k])$ as the keys, since these correspond to the two halves of the i^{th} suffix of length $2k$. We can therefore sort the array by first sorting on $\text{rank}[i+k]$ and then stably sorting on $\text{rank}[i]$. Using a counting sort for each of these, we perform $O(n)$ work to sort the array. Using this method, the complexity of the prefix doubling algorithm drops to $O(n\log(n))$ since we still perform $O(\log(n))$ rounds, but each round now takes just $O(n)$.

Algorithm 74 Prefix-doubling suffix array construction

```
1:  function SUFFIX_ARRAY(S[1..n])
2:      SA[1..n] = [1..n]
3:      rank[1..n] = [ord(S[1..n])]
4:      for k = 1 to n, stepping k ×= 2 do
5:          sort(SA[1..n], suffix_compare(rank, k, ...))
6:          // Update the rank array to account for the new sorted order
7:          temp[1..n] = 0
8:          for i = 1 to n − 1 do
9:              temp[SA[i+1]] = temp[SA[i]] + suffix_compare(rank[1..n], k, SA[i], SA[i + 1])
10:         swap(temp, rank)
11:     return SA
12:
13: // Compare the suffixes of length 2k beginning at positions i and j
14: function SUFFIX_COMPARE(rank[1..n], k, i, j)
15:     if rank[i] ≠ rank[j] then                          // Compare by first halves
16:         return rank[i] < rank[j]
17:     else if i + k ≤ n and j + k ≤ n then               // Compare by second halves
18:         return rank[i + k] < rank[j + k]
19:     else                                               // Second half is empty
20:         return j < i
```

Linear time suffix arrays

Prefix doubling with radix sort achieves reasonably good performance on very large strings in practice, but we can still do even better! Several algorithms exist for constructing suffix arrays in just $O(n)$ time for a fixed size alphabet. Probably the most well known of these algorithms is the DC3 algorithm[1], which works by sorting a set of "sample suffixes" which are the suffixes at positions in the string that are not divisible by three. DC3 operates by radix sorting the first three characters of each suffix (stably in reverse order), then recursively sorting sets of suffixes that are still equal. Using the sample suffixes, one can then quickly sort the remaining 1/3 of the suffixes and merge them with the sample suffixes since every non-sample suffix is just one character prepended to a sample suffix. The running time is then given by

$$T(n) = \begin{cases} T\left(\frac{2}{3}n\right) + O(n) & \text{if } n > 1, \\ \Theta(1) & \text{if } n = 1. \end{cases}$$

which can be solved to reveal a time complexity of $O(n)$.

13.2 Applications of Suffix Arrays

As fun as suffix arrays are to think about, they also admit a wide range of useful applications to string processing problems. Let's look at the most important one, pattern matching.

[1] See *Linear Work Suffix Array Construction* , Juha Kärkkäinen, Peter Sanders, Stefan Burkhardt

13.2.1 Pattern Matching with Suffix Arrays

Using the suffix array of the text string $T[1..n]$, we can perform pattern searches fast, saving time if a large number of searches on the same string are desired. The idea is very simple: since the suffix array gives the suffixes in sorted order, all occurrences of the pattern that we are searching for will exist in some contiguous range. Since they are sorted, we can find this range with a pair of binary searches, one to locate the first occurrence and one to locate the final occurrence. The idea is illustrated in Algorithm 75. All of the occurrences of the pattern are found at the positions given by $SA[begin..end]$. If this range is empty, then no matches were found. Since each string comparison takes $O(m)$ time and the binary searches must perform $O(\log(n))$ iterations, the total time complexity of pattern matching using a suffix array is $O(m\log(n))$ which is a significant improvement over $O(n+m)$ when $n \gg m$. It is possible to speed this up to just $O(m)$ using the *longest common prefix array*, a companion to the suffix array which we will not cover for now.

Algorithm 75 Suffix array: Pattern matching

```
1:  function FIND_PATTERN(SA[1..n], T[1..n], P[1..m])
2:      // Binary search to find the first occurrence of P
3:      // Invariant: T[SA[lo]..n] < P and T[SA[hi]..n] ≥ P
4:      lo = 0, hi = n
5:      while lo < hi - 1 do
6:          mid = ⌊(lo + hi)/2⌋
7:          if T[SA[mid]..n] < P[1..m] then
8:              lo = mid
9:          else
10:             hi = mid
11:     begin = hi
12:     // Binary search to find the final occurrence of P
13:     // Invariant: T[SA[lo]..n] ≤ P and T[SA[hi]..n] < P
14:     lo = 1, hi = n + 1
15:     while lo < hi - 1 do
16:         mid = ⌊(lo + hi)/2⌋
17:         if T[SA[mid]..n] ≤ P[1..m] then
18:             lo = mid
19:         else
20:             hi = mid
21:     end = lo
22:     return begin, end
```