Algorithms

Muchang Bahng

Spring 2024

Contents

A course on the study of data structures and algorithms. We can start off by defining what these two terms mean.

Definition 0.1 (Data Structure)

A **data structure** $\mathcal{D}$ is an object that can store some amount of data. The **size** $|\mathcal{D}|$ of the data structure is a finite collection of nonnegative integers used to characterize the number of values that it stores.[a]

[a]For example, the size of an array is N, but the size of a graph $G = (V, E)$ is characterized by two numbers (N, M) representing the number of vertices and edges in the graph.

Definition 0.2 (Algorithm)

Given a data structure $\mathcal{D}$, an **algorithm** $\mathcal{A}(\mathcal{D})$ is a procedure on the data structure for correctly solving a mathematical problem in a finite number of steps. There are generally two paradigms on how algorithms work:

1. *Iterative.* Where $\mathcal{A}(\mathcal{D})$ is called that processes $\mathcal{D}$.
2. *Recursive.* $\mathcal{A}(\mathcal{D})$ calls the same algorithm on a finite number of processed (usually smaller) data structures $\mathcal{A}(\mathcal{D}_i)$. After these recursive calls, the base algorithm may then postprocess the outputs to generate a new output. The general way to do this is to consider the base case of $\mathcal{D}$ when $\mathcal{A}(\mathcal{D})$ should not make any more recursive calls. Then assuming that the algorithm runs correctly for some input, determine what we should do with the recursive call.[a]

The **runtime** of an algorithm is a function $T(|\mathcal{D}|)$ with respect to the size of the dataset that outputs the number of steps it takes to complete this algorithm.[b][c]

[a]We must ensure that every recursive call gets closer to the base case, or this will never end in a finite number of steps and is thus not an algorithm.

[b]The reason I don't put time as the output is that time can refer to many things, such as physical time or the number of flops, which may differ depending on what computer you run an algorithm on. The number of steps is constant, and informally these refer to *atomic* steps that cannot be decomposed further into substeps.

[c]There are two notions of runtime here. We can compare f and g with respect to the *value* N of the input, or we can compare them with respect to the *number of bits* n in the input. While we mostly use the complexity w.r.t. the value in these notes, we should be aware for certain (especially low-level operations), the bit complexity is also important, which will be focused on in my advanced algorithms notes.

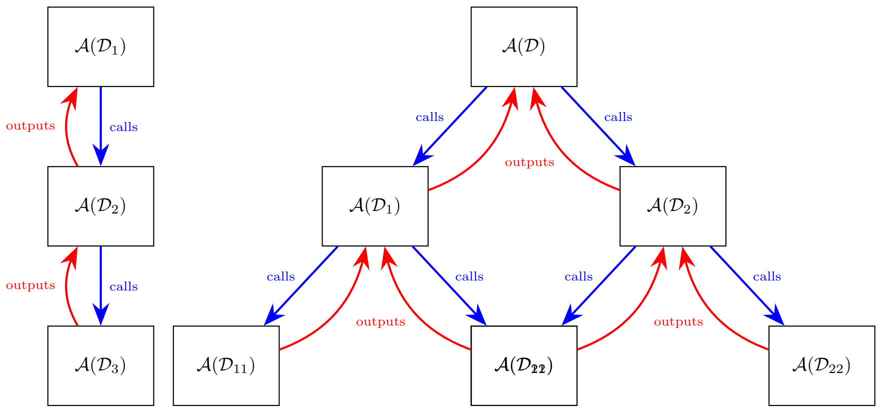

$\mathcal{A}(\mathcal{D}_1)$
outputs
calls
$\mathcal{A}(\mathcal{D}_2)$
outputs
calls
$\mathcal{A}(\mathcal{D}_3)$
$\mathcal{A}(\mathcal{D})$
calls
calls
outputs
$\mathcal{A}(\mathcal{D}_1)$
$\mathcal{A}(\mathcal{D}_2)$
calls
calls
calls
calls
outputs
outputs
$\mathcal{A}(\mathcal{D}_{11})$
$\mathcal{A}(\mathcal{D}_{22})$
$\mathcal{A}(\mathcal{D}_{22})$

Figure 1: Linear recursion on the left. Binary tree recursion on the right.

This is a pretty loose definition of data structure and algorithms, but the point I want to make is that data structures are a fundamental object of study, and to interact with these data structures, we can use algorithms (hence I put algorithms as taking a data structure as an input). The whole purpose of having different data structures is that we can construct efficient algorithms that can interact with them in different ways, e.g. getter/setter algorithms. Therefore, this is sort of like the chicken and egg situation. Algorithms act on data structures, but data structures are inspired by algorithms.

Clearly, the runtime is extremely important when comparing the efficiency of different algorithms. To formalize this comparison, we need the power of mathematical analysis to analyze the asymptotic behavior between two functions.

Definition 0.3 (Complexity)

Given two positive functions f, g,

1. $f = O(g)$ if f/g is bounded.[a]
2. $f = \Omega(g)$ if g/f is bounded, i.e. $g = O(f)$.
3. $f = \Theta(g)$ if $f = O(g)$ and $g = O(f)$.

Recalling analysis, if we want to figure out the complexity of two positive functions f, g,[b] we can simply take the limit.

$$\lim_{x\to+\infty} \frac{f(x)}{g(x)} = \begin{cases} 0 & \implies f = O(g) \\ 0 < x < +\infty & \implies f = \Theta(g) \\ +\infty & \implies f = \Omega(g) \end{cases} \tag{1}$$

[a]Note that it is more accurate to write $f \in O(g)$, since we consider $O(g)$ a class of functions for which the property holds.

[b]These will be positive since the runtime must be positive.

Most of the time, we will have to use L'Hopital's rule to derive these actual limits, but the general trend is $\log n$ is small, poly(n) grows faster, $\exp(n)$ grows even faster, $n!$ even faster, and n^n even faster. Some nice properties follow immediately.

Theorem 0.1 (Properties)

Some basic properties, which shows very similar properties to a vector space.

1. Transitivity.

$$f = O(g), g = O(h) \implies f = O(h) \tag{2}$$
$$f = \Omega(g), g = \Omega(h) \implies f = \Omega(h) \tag{3}$$
$$f = \Theta(g), g = \Theta(h) \implies f = \Theta(h) \tag{4}$$
$$\tag{5}$$

2. Linearity.

$$f = O(h), g = O(h) \implies f + g = O(h) \tag{6}$$
$$f = \Omega(h), g = \Omega(h) \implies f + g = \Omega(h) \tag{7}$$
$$f = \Theta(h), g = \Theta(h) \implies f + g = \Theta(h) \tag{8}$$
$$\tag{9}$$

Example 0.1 (Comparing Runtimes)

Compare the following functions.

1. $f(n) = \log_{10}(n), g(n) = \log_2(n)$. Since they are different bases, we can write $f(n) = \log(n)/\log(10)$ and $g(n) = \log(n)/\log(2)$. They differ by a constant factor, so $f = \Theta(g)$.
2. $f(n) = (\log n)^{20}, g(n) = n$. We have

$$\lim_{n\to\infty} \frac{(\log n)^2 0}{n} = \lim_{n\to\infty} \frac{20 \cdot (\log n)^{19} \cdot \frac{1}{n}}{1} = \ldots = \lim_{n\to\infty} \frac{20!}{n} = 0 \implies f = O(g) \tag{10}$$

3. $f(n) = n^{100}, g(n) = 1.01^n$. We have

$$\lim_{n\to\infty} \frac{n^{100}}{1.01^n} = \lim_{n\to\infty} \frac{100n^{99}}{1.01^n \cdot \log(1.01)} = \ldots = \lim_{n\to\infty} \frac{100!}{1.01^n \cdot (\log 1.01)^100} = 0 \implies f = O(g) \tag{11}$$

Let's do a slightly more nontrivial example.

Example 0.2 (Complexity of a Recursive Function)

Given the following algorithm, what is the runtime?

```
for i in range(1, n+1):
  j = 1
  while j <= i:
    j = 2 * j
```

Now we can see that for each i, we will double up to $\log_2(i)$ times. Therefore summing this all over i is

$$\sum_{i=1}^{n} \log_2(i) = \log_2(n!) \leq \log_2(n^n) = n\log_2(n) \tag{12}$$

and so we can see that the runtime is $O(n \log n)$. Other ways to do this is to just replace the summation with an integral.[a]

$$\int_1^n \log_2(x)\,dx = x\log(x) - x\Big|_1^n = n\log(n) - n + 1 = O(n\log n) \tag{13}$$

[a]Need more justification on why this is the case. Mentioned in lecture.

1 Array-Based Data Structures

1.1 Arrays and Lists

We want to define a list that has the following methods given some list `lst`.

1. `Object y = lst.get(0)` outputs the 0th element.
2. `boolean y = lst.contains(elem)` checks if `elem`
3. `lst.add(Object elem)` adds the element to the end of the list.
4. `lst.remove(Object elem)` removes the element.
5. `lst.size()` returns size of the list.

Definition 1.1 (ArrayList)

An ArrayList just implements an array in the backend but with some extra systematic way to dynamically grow. If we add to an array, we either have space and can do it, or we don't and can't. If we add to an ArrayList, we can

1. simply add to the first open position if there's space left, $O(1)$
2. we grow the size of the ArrayList by creating a new larger array, copying everything, and then adding to the first open position. (linear time $O(n)$), since we have to add all the elements to the new array.

Starting with a length 1 array, if we add N elements one at a time and when full, create a new array that is

1. is twice as large (geometric growth: 1, 2, 4, 8, 16, ...). Then, we must copy at sizes 1, 2, 4, 8, ... and the total values copied looks like
$$1 + 2 + 4 + \ldots + (N/4) + (N/2) = N - 1$$
This is what the Java.util.ArrayList implements, and you can see the performance of adding is $O(N)$.
2. has 1 more position (arithmetic growth: 1, 2, 3, ...). Then, we must copy at sizes 1, 2, 3, 4, ... and the total values copied looks like
$$1 + 2 + 3 + \ldots + (N - 1) = N(N - 1)/2$$

If the arithmetic pattern is adding, say 1,000,000 elements, then we are wasteful of memory. This geometric growth is a good tradeoff between performance and memory usage. It never uses more than twice the memory of an array in order to store it. Furthermore, the runtime of a geometric growth pattern is amortized constant time, which means that it is constant when averaged over a long time. This is because the vast majority of these operations are constant time, with a few add operations which require resizing to be longer. But these few ones happen less and less frequently that when averaged over a long period, we can treat it as constant.

One thing to note is that while adding to the end of an ArrayList can be efficient, adding to the front is not since it must shift the entire Array, even if there is space left.

Theorem 1.1 (ArrayList Runtime Complexity)

The following are true for ArrayList `lst`:

1. Getting and Contains
   (a) `lst.get(int index)` is $O(1)$.
   (b) Getting every element is $O(n)$
   (c) `lst.contains(Object elem)` is $O(n)$

2. Adding
   (a) `lst.add(Object elem)` is amortized $O(1)$.
   (b) `lst.add(0, Object elem)` is $O(n)$.
   (c) `lst.add(int index, Object elem)` is on average $O(n)$.
3. Removing
   (a) `lst.remove(0)` is $O(n)$
   (b) `lst.remove(int index)` is $O(n)$
   (c) `lst.remove(lst.size() - 1)` is $O(1)$
   (d) `lst.remove(Object elem)`

Proof.

Listed.

1. Getting the element at index `index` requires us to just look at the same index in the underlying array, which is $O(1)$.
2. We loop through each element of the ArrayList and call `.equals(elem)` at each step, which results in $O(1)$.
3. Since the geometric growth of the ArrayList happens exponentially less frequently, it averages out to be $O(1)$, so amortized.
4. Adding at a specific index requires $O(n)$ since we create a new ArrayList and copy over all the elements with the added element.
5. Removing an object requires us to shift the indices of the remaining elements by 1, so this is $O(n)$.

Example 1.1 (String)

The string type is just an ArrayList of characters. It has the following attributes and methods. Let `x = "I love CS201"`

1. `int y = x.length()` outputs the length and is $O(1)$
2. `char y = x.charAt(0)` outputs a character and is $O(1)$
3. `String y = x.substring(0, 4)`
4. `boolean y = x.equals("I love CS201")`
5. `String y = x + "!!"`
6. `String[] y = x.split(" ")`
7. `String y = String.join(" ", words)`

Definition 1.2 (Linked List)

A linked list contains a sequence of nodes that each contain an object for its element and a reference to the next node. More specifically, it can be divided up into 3 parts:

1. The variable which points to the *first* node. This can be confusing since this variable, which represents an *entire list* is just a pointer to the first node.
2. A sequence of nodes containing the element and a reference to the next node.
3. The final node containing the element and a reference to `null`.

We can implement these functionalities in the `ListNode` class, which are used to build a linked list of integers.

```java
public class ListNode {
    int info;           // value i.e. element
    ListNode next;      // reference to next ListNode

    ListNode(int x) {
        info = x;
    }
    ListNode(int x, ListNode node) {
        info = x;
        next = node;
    }
}
```

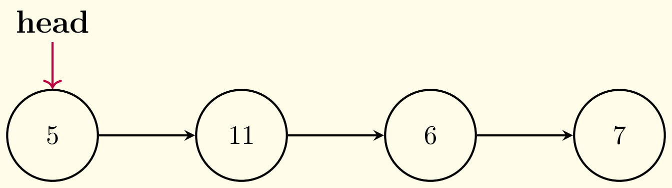
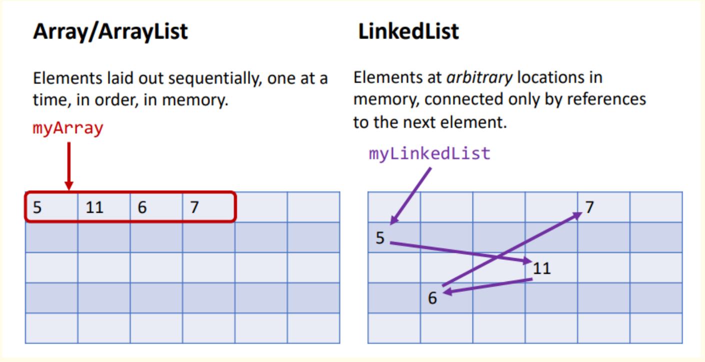


Figure 2: The following diagram represents a linked list.

But in reality, the elements are all located random in memory and can only be found by references.


Array/ArrayList
Elements laid out sequentially, one at a time, in order, in memory.
myArray
5
11
6
7
LinkedList
Elements at *arbitrary* locations in memory, connected only by references to the next element.
myLinkedList
7
5
11
6

To print everything in a linked list, we just loop over the nodes as long as the nodes are not null.

```java
public static void printList(ListNode list) {
    while (list != null) {          // common conditional for traversing
        System.out.println(list.info);
        list = list.next;
    }
}
```

Theorem 1.2 (Linked List Runtime Complexity)

The following are true for a basic LinkedList `lst`:

1. Getting and contains
   (a) `lst.get(int index)` is $O(n)$ on average (unless you get the first index, which is fast).
   (b) Getting every element in the list is $O(n^2)$.
   (c) `lst.contains(Object elem)` is $O(n)$
2. Adding
   (a) Start: `lst.add(0, Object elem)` is $O(1)$
   (b) Middle: `lst.add(int index, Object elem)` is on average $O(n)$.
   (c) End: `lst.add(Object elem)` is $O(n)$
3. Removing
   (a) Start: `lst.remove(0)` or `lst.remove()` is $O(1)$

(b) Middle: `lst.remove(int index)` is on average $O(n)$
(c) End: `lst.remove(lst.size() - 1)` is $O(n)$

Proof.

Listed.
1. We must traverse from the beginning of the linked list, and so it is $O(n)$. If we just pay attention to the first (or last, for doubly-linked list) element, then this is just $O(1)$.
2. Getting every element is just looping an $O(n)$ operation n times, so $O(n^2)$.
3. You need to iterate through each element and call `.equals(elem)`, so it is $O(n)$.
4. We can simply take the reference
5. Adding

Even though our basic LinkedList solves the problem of adding in the beginning, in order to add in the middle or end, we must get to that position (which is $O(n)$ time) before we are able to utilize our $O(1)$ add. This is quite inefficient, especially when we do repeated adding, so we should keep track of certain "markers" that indicate where our current node is. **Iterators** do this naturally, so we would like to implement some current notion of position.Below we implement a new linked list (of integers).

Definition 1.3 (Iterator)

An **iterator** is a Java interface that has the two methods:
1. `.hasnext()` checks if there is an element after the current one.
2. `.next()` prints out the next element.

We want to implement iterators to any collections or whatever custom class if we want to be able to use enhanced for loops over them.

Definition 1.4 (DIYLinkedList)

Note the following:
1. Adding to the end (`add`) and to the front (`addtoFront`) are both $O(1)$, since we always have access to the dynamic attributes `first` and `last`.

```java
public class DIYLinkedList implements Iterable<Integer> {
    private class ListNode {
        int value;
        ListNode next;
        public ListNode(int value) {
            this.value = value;
        }
        public ListNode(int value, ListNode next) {
            this.value = value;
            this.next = next;
        }
    }

    private ListNode first;
    private ListNode last;
    private int size;

    public DIYLinkedList() {
        size = 0;
    }

    public int size() {
        return size;
    }

    public int get(int index) {
        if (index < 0 || index >= size) {
            throw new IndexOutOfBoundsException();
        }
        current = first;
        for (int i = 0; i < index; i++) {
            current = current.next;
        }
        return current.value;
    }

    public void add(int elem) {
        // add to end
        if (last == null) {
            last = new ListNode(elem);
            first = last;
        }
        else {
            last.next = new ListNode(elem);
            last = last.next;
        }
        size++;
    }

    public void addToFront(int element) {
        // add to front
        first = new ListNode(element, first);
        size++;
    }

    private class DIYListIterator implements Iterator<Integer> {
        ListNode current = first;

        @Override
        public boolean hasNext() {
            return current != null;
        }

        @Override
        public Integer next() {
            int value = current.value;
            current = current.next;
            return value;
        }
    }

    @Override
    public Iterator<Integer> iterator() {
        return new DIYListIterator();
    }
}
```

Theorem 1.3 (Appending)

Appending two ListNodes (of size n and m) is $O(n)$ time.

```java
public static ListNode append(ListNode listA, ListNode listB) {
    ListNode first = listA;
    while (listA.next != null) {
        listA = listA.next;
    }
    listA.next = listB;

    return first;
}
```

Theorem 1.4 (Reversing)

When we reverse a linked list, we want to work with it one step at a time by establishing a **loop invariant**, which is just some condition that we want to be true every iteration. In this case, our invariant is "after k iterations, `rev` points to the reverse of the first k nodes."

```java
public ListNode reverse(ListNode front) {
    ListNode rev = null;
    ListNode list = front;
    while (list != null) {
        ListNode temp = list.next;
        list.next = rev;
        rev = list;
        list = temp;
    }
    return rev;
}
```

Example 1.2 ()

Here are three reversing examples, in increasing difficulty:

1. If `front` is a ListNode with `front.next == null`, then `reverse(front)` will return

$$\texttt{reverse(front)} \mapsto \texttt{front} \mapsto \texttt{null}$$

2. If we have a linked list `list` $\mapsto 1 \mapsto 2 \mapsto 3 \mapsto$ `null`, then `reverse(list.next)` will return

$$\texttt{reverse(list.next)} \mapsto 3 \mapsto 2 \mapsto \texttt{null}$$

3. If we have a linked list `list` $\mapsto 1 \mapsto 2 \mapsto 3 \mapsto$ `null`, then after running `reverse(list.next)`, the original list variable will be

$$\texttt{list} \mapsto 1 \mapsto 2 \mapsto \texttt{null}$$

   This is because after the method call, we have $3 \mapsto 2 \mapsto$ `null`, but the 1 still points to 2! Therefore, the original list, which points to 1, will point to 2, which points to `null`.

1.2 Linked List

Example 1.3 (Reverse)

We can reverse a LinkedList with the following recursive algorithm.

```java
public static ListNode reverse(ListNode list) {
    if (list == null || list.next == null) {
        return list;
    }
    ListNode reversedLast = list.next;
    ListNode reversedFirst = reverse(list.next);
    reversedLast.next = list;
    list.next = null;
    return reversedList;
}
```

Example 1.4 ()

The following algorithm

```java
public static ListNode rec(ListNode list) {
    if (list == null || list.next == null) {
        return list;
    }
    ListNode after = rec(list.next);
    if (list.info <= after.info) {
        list.next = after;
        return list;
    }
    return after;
}
```

1.3 Stacks

Definition 1.5 (Stack)

A **stack** is an abstract data structure represented as a **Last-In-First-Out (LIFO) list**, which implements the following methods given stack `st`, which we can initialize with

1. `st.add(Object element)` adds to the top of the stack, which is $O(1)$
2. `st.pop()` removes the element that is at the top of the stack, which is $O(1)$, and returns whatever is popped out.

Remember that this is just a list and so anything we can do with a stack we can do with a list. What makes the stack so useful is the way the list is implemented. We can literally imagine the elements of this list as "stack." If you want to remove something from the stack, of course you have to remove the top element.

1.4 Queues

Definition 1.6 (Queue)

A **queue** is an abstract data structure represented as a **First-In-First-Out (FIFO) list**, which implements the following methods given queue `q`, which we can initialize with Note that LinkedList implements the Queue interface.

1. `q.add(Object element)` adds to the top of the queue, referred to as **enqueue**.
2. `q.remove()` removes the first element in the queue, referred to as **dequeue**.

This is just like how a queue works. Whatever has been waiting in the queue the longest is the one that is removed first.

2 Hash-Based Data Structures

2.1 Hashing

One of the most important applications of hashing and probabilistic algorithms is in cryptography. For example, when you generate a RSA keypair, you must choose two 128-bit prime numbers which serve as your private key. State of the art methods simply generate a random 128-bit prime number perform *independent* randomized probabilistic tests with error rate ϵ. Given that this is independent, simply running this tests k times reduces our error exponentially to ϵ^k, practically guaranteeing primality. In Bitcoin and Ethereum wallets, hardened child key derivation functions in hierarchical determinstic wallets hash subsets of the parent key to generate both the child keys and their seeds.[1]

Assume we want to map keys from some **universe** U into a smaller set $[N] = \{1, \ldots, N\}$ of N bins. The algorithm will have to handle some dataset $S \subset U$ of $|S| = M$ keys.[2] We want to store S with efficient performance in

1. `Find(x)`, which returns true if $x \in S$
2. `Insert(x)`
3. `Delete(x)`

This is essentially a set. If one is new to hashing, we could try to use a balanced binary search tree or a heap (implemented as a red-black tree or some other variant), which can do all three operations in $O(\log(N))$ time. If we had access to a randomized function $h : U \to [N]$, with the property that for a fixed $x \in U$, $h(x) \sim \text{Multinomial}(N)$, then by linearity of expectation[3]

$$\mathbb{P}(h(x) = h(y)) = \mathbb{E}[\mathbb{1}_{h(x)=h(y)}] = \frac{N}{N^2} = \frac{1}{N} \tag{14}$$

This is great, but this is also random, which means that we are not guaranteed to map to the same bin across time. If we add time-invariance, this is pretty much a hash function, which is really deterministic, but we like to call it psuedo-random.

So what if there are collisions? They are inevitable anyways even with a completely random function, but minimizing them will give us the best runtime performance. There are a two main ways to approach this.

1. At each bin, store the head of a linked list, BST, or another hash table. You would incur an additional linear, logarithmic, or constant cost of traversing this data structure for each operation.
2. Look for another bin. We can just look at $h(x) + 1$ or $h(h(x))$ (or really any deterministic function $f(h(x))$) if $h(x)$ is occupied.

This is what we have to work with here, so we would like to modify our assumptions. Therefore, we can achieve good performance by reducing the collisions and by improving how we deal with collisions. We will focus on the first part, and we would like for our hash functions to have some nice properties, which we will define. The most intuitive way is to treat x, y as random variables uniformly sampled from U, and then we would like $\mathbb{P}(h(x) = H(y)) \approx 1/N$. However in practice, this is not the case since our subdomain S is already fixed, yet may be unknown. Therefore given any function h, we can construct an adversarial sequences of inputs that will all map to the same bin. We must therefore consider a *family* of hash functions.

Definition 2.1 (Universal Hashing)

A family of hash functions $H = \{h : U \to [N]\}$ is **universal** if for any fixed $x, y \in U$,

$$\mathbb{P}_h\big(h(x) = h(y)\big) \leq \frac{1}{N} \tag{15}$$

[1]For an implementation of both, look at my blockchain implementation here
[2]For example, think of all possible strings as U and all 256-bit numbers as $N = 2^{256}$. S in this case S may be the set of all addresses on a blockchain. This is SHA256, and to date there is no known hash collisions, though there are a few for SHA1.
[3]Note that this is not $\mathbb{P}_{x,y}$ since x, y are not random variables. They are fixed, and $h(x), h(y)$ are the random variables.

where h is drawn uniformly. If the bound is $2/N$ then it is called **near-universal**, and what's important is the N in the denominator. This polynomial decay in collision probabilities is exactly what we need for constant operation time. That is, given query x, let the random variable X be the number of operations we must do to find X, which consists of 1 pointer traversal plus some other amount of traversals (e.g. through a linked list) if collision, say which is C_n. So we have

$$X = 1 + \sum_{y \in S} Q_y \tag{16}$$

where 1 comes from the initial pointer traversal in hashing, and Bernoulli $Q_y = 1$ if $h(y) = h(x)$.[a] Then, by linearity of expectation combined with the bound on probability of collision we have

$$\mathbb{E}[X] = 1 + \sum_{y \in S} \mathbb{E}[Q_y] = 1 + |S| \cdot \mathbb{P}(h(x) = h(y)) = 1 + \frac{|S|}{N} \tag{17}$$

which is constant.

[a]This may not be 1 since we may have to do more than 1 additional traversal, e.g. in linked lists. But we assume the extra cost is bounded by a constant.

If H is the set of all functions from U to $[N]$, then this universal property is trivially satisfied since

$$\mathbb{P}_h\big(h(x) = h(y)\big) = \frac{1}{N} \tag{18}$$

But this is quite a complex function class and h may not be easy to store, i.e. may not be analytic. The question now becomes whether we can achieve universality with a smaller class. It turns out yes, and the obvious use is to use cyclic groups.

Lemma 2.1 (Carter-Wegman 71)

Suppose $U = \{0, 1, \ldots, M-1\}$. We choose a prime $p \geq M$[a] and construct the family

$$H = \{h_a(x) = (ax \text{mod} p) \text{mod} N \mid a = 0, \ldots, p-1\} \tag{19}$$

We claim this is near-universal.

[a]It is guaranteed that a prime exists between m and $2m$.

Proof.

We treat x as the generator of this group, and then $ax\text{mod}p = a'xmodp$ iff $a = a'$. Therefore,

$$\mathbb{P}_a(h_a(x) = h_a(y)) = \mathbb{P}_a\big((ax - ay)\text{mod}p = Nk\big) \tag{20}$$

for some k. If $a = 0$, then everything collides, but if $a \neq 0$ and $x \neq y$, then $ax - ay$ must have a difference of exactly a multiply of N, which happens with probability $1/N$. Therefore, we have

$$\mathbb{P}_a(h_a(x) = h_a(y)) = \mathbb{P}(a = 0) + \mathbb{P}(a \neq 0) \cdot \frac{1}{N} = \frac{1}{p} + \frac{p-1}{p} \cdot \frac{1}{N} < \frac{2}{N} \tag{21}$$

since $p > N$.

Therefore, to hash,

1. we choose a prime $p \geq m = |U|$.
2. choose $a \in \{0, 1, \ldots, p-1\}$ at random.

3. Use $h_a(x) = (ax \bmod p) \bmod N$ to hash everything.[4]

Example 2.1 ()

Let n be a fixed prime number and

$$H = \{h_{a,b}(x, y) = ax + by \bmod n : a, b \in [n]\} \tag{22}$$

We claim that this is universal. Let $x = (x_1, x_2)$ and $y = (y_1, y_2)$ be arbitrary pairs of integers where $x \neq y$. WLOG, assume $x_2 \neq y_2$. Then

$$\mathbb{P}(h_{a,b}(x) = h_{a,b}(y)) = \mathbb{P}(ax_1 + bx_2 = ay_1 + by_2) = \mathbb{P}\big(a(x_1 - y_1) = b(y_2 - x_2)\big) \tag{23}$$

Suppose that we pick a first and we compute the conditional probability. Then the LHS is a constant, and for this to hold in a cyclic group, b must equal $c(y_2 - x_2)^{-1}$, which occurs with probability $1/N$. Using LOTP the joint probability is also $1/N$.

2.2 Set

Sets are conceptually the same as mathematical sets.

Definition 2.2 (Set)

A **set** is an unordered data structure that stores unique elements.

1. Add is $O(1)$.
2. Remove is $O(1)$.
3. Retrieval and contains is $O(1)$.

2.3 Map

A hash table is an array of key value pairs. But rather than adding to positions in order from 0, 1, 2, ..., we will calculate the hash of the key, which would return an int that specifies where we store this key-value pair. So to store, `<"ok", 8>`, we will calculate `hash("ok") == 5` and store it in the 5th index.

```
0
1
2 <"hi", 5>
3
4
5 <"ok", 8>
6
7
```

We can immediately see how this makes search easier, since if we want to find the value associated with the key "ok", then we can calculate the hash of it to find the index and look it up on the array.

Definition 2.3 (Maps)

A **map**, also called a **hash table**, is an unordered data data structure that stores key-value pairs.

1. Add is $O(1)$.
2. Remove is $O(1)$.
3. Retrieval and contains is $O(1)$.

[4]We do not change a every time we hash.

So running `get(key)` on a HashMap looks up position `hash(key)` in the hash table and returns the value there. Immediately, we see that if `hash` is not injective (which it isn't), then we can run into collisions. This is solved using chaining or bucketing. Bucketing basically takes each index in the array and stores not just one key-value pair, but a list of key-value pairs. So basically, when we want to search for the value of a key, we compute the index of it with `hash(key)`, which would return a list of key-value pairs.

Obviously, if we create a custom `hashCode()` method that trivially maps to 0, then we would just have one giant list in the bucket at index 0, which is no more efficient than a list search. So, we should ideally assume that given N pairs with M buckets, our hashing function is built so that the probability of two random (unequal keys) hash to the same bucket is 1/M. Note that this hash function is completely deterministic. We should talk about runtime/memory tradeoff. Given N pairs and M buckets (with SUHA):

1. $N >> M$ means too many pairs in too few buckets, so runtime inefficient
2. $M >> N$ means too many buckets for too few pairs, so memory wasteful
3. M slightly larger than N is the sweet spot.

To maintain an ideal ratio, we basically create a new larger table (with geometric resizing) and rehash/copy everything until we reach it.

3 Tree-Based Data Structures

3.1 Binary Trees

Let us compare the HashSet/Map and the TreeSet/Map. The purpose of Hashing is to "find" and add elements quickly.

1. This means that add, contains, put, and get are all amortized $O(1)$ (under Simple Uniform Hashing Assumption). The TreeSet/Map have all operations add, contains, put, get are $O(\log(N))$, which is slower, but is not amortized.

2. Trees are sorted, while Hashes are not, and so we can get a range of Tree values in sorted order efficiently, but not for Hashes.

Definition 3.1 (Binary Tree)

A **binary tree** is a recursive data structure in which every node has up to two children, which we call the *left child* and *right child*.

1. The **root** of the tree is the top node, which is 4
2. The **leaf** of the tree are nodes that do not have a left nor right subchild.
3. A **path** is any path from one node to another node. A simple path is a path that doesn't cross the same edge twice
4. The **height** of a node is the length of the longest downward path to a leaf from that node.
5. The **depth** of a node is the number of edges from the root to the node.

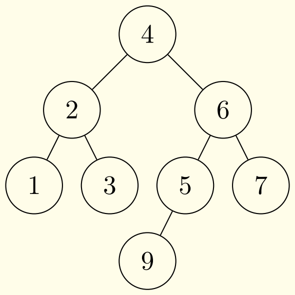


Figure 3: An example of a binary tree. The root is 4, with a max depth of 3.

Theorem 3.1 (Print All Nodes in A Binary Tree)

Here are three ways to recursively traverse a tree. The difference is in where the nonrecursive part is. Let us have a binary tree from above.

1. This tells us to print everything on the left of the node, then print the node, and then print everything on the right.

```
void inOrder(TreeNode t) {
    if (t != null) {
        inOrder(t.left);
        System.out.println(t.info);
        inOrder(t.right);
    }
}
// 1, 2, 3, 4, 9, 5, 6, 7
```

2. This tells us to print the node itself first, then print all the ones on the left, and then print all the ones on the right.

```java
void preOrder(TreeNode t) {
    if (t != null) {
        System.out.println(t.info);
        preOrder(t.left);
        preOrder(t.right);
    }
}
// 4, 2, 1, 3, 6, 5, 9, 7
```

3. This tells us to print all the nodes on the left, then all ones on the right, and then the node itself.

```java
void postOrder(TreeNode t) {
    if (t != null) {
        postOrder(t.left);
        postOrder(t.right);
        System.out.println(t.info);
    }
}
// 1, 3, 2, 9, 5, 7, 6
```

Theorem 3.2 (Storing All Nodes in a List)

Now if we want to store them all in a list, then this recursive strategy will not work, since if we create a list inside the function body, then we will have a bunch of lists floating around in memory. Therefore, we want to initialize a list outside of the entire function, and store that entire thing within a wrapper function. The `inOrder` takes in also a reference to a list that it will be adding to.

```java
public ArrayList<String> visit(TreeNode root) {
    ArrayList<String> list = new ArrayList<>();
    inOrder(root, list);
    return list;
}

private void inOrder(TreeNode root, ArrayList<String> list) {
    if (root != null) {
        inOrder(root.left, list);
        list.add(root.info);
        inOrder(root.right, list);
    }
}
```

Definition 3.2 (Finding Height of Node)

The height of a node is the longest downward path to a leaf from that node, so its height would be the maximum of the two heights of its children. A null node would have height -1, which is our base case.

```java
public int getHeight(TreeNode root) {
    if (root == null) { return -1; }
    return 1 + Math.max(getHeight(root.left), getHeight(root.right));
}
```

Definition 3.3 (Finding Depth of Node)

The depth is quite hard to find recursively, but if we have a reference to the parent, then we can write

```java
int depth(TreeNode node) {
    if (node == null) {
        return -1;
    } else {
        return 1 + depth(node.parent);
    }
}
```

3.2 Heaps

Definition 3.4 (Binary Heap)

A **binary heap** is a binary tree satisfying the following structural invariants:

1. Maintain the **heap property** that every node is less than or equal to its successors, and
2. The **shape property** that the tree is complete (full except perhaps last level, in which case it should be filled from left to right.

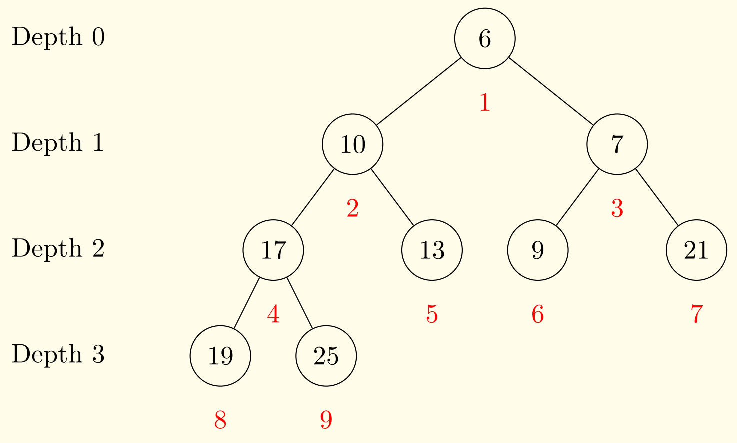


Figure 4: Tree structure of a binary heap, with red labels representing the indices of the array structure mentioned below.

We should conceptually think of a binary heap as an underlying binary tree, but it is actually usually implemented with an array, and we can create a map from the heap to the array with the following indices.

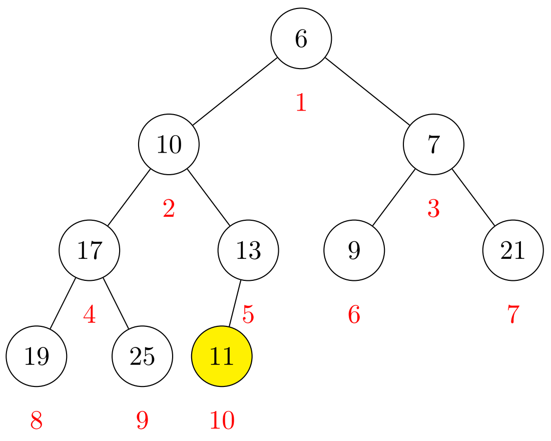
	6	10	7	17	13	9	21	19	25	
0	1	2	3	4	5	6	7	8	9	10

Figure 5: When 1-indexing, for node with index k, the left child is index $2k$, the right child is index $2k + 1$, and the parent is index $k/2$ (where this is integer division).

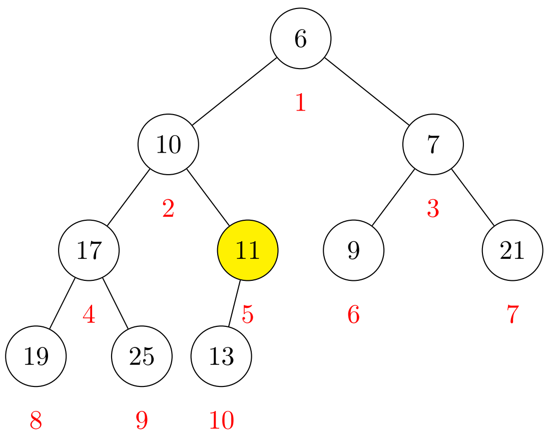
Implementing peek is easy, since we just return the first index, but it can be quite tricky to maintain this invariant after an arbitrary sequence of add/remove operations.

1. To add values to a heap, we add to the first open position in the last level of the tree (to maintain the

shape property), and then swap with the parent is the heap property is violated. If we are swapping with the parent at most $\log(N)$ times, then the add property has $O(\log(N))$ complexity.



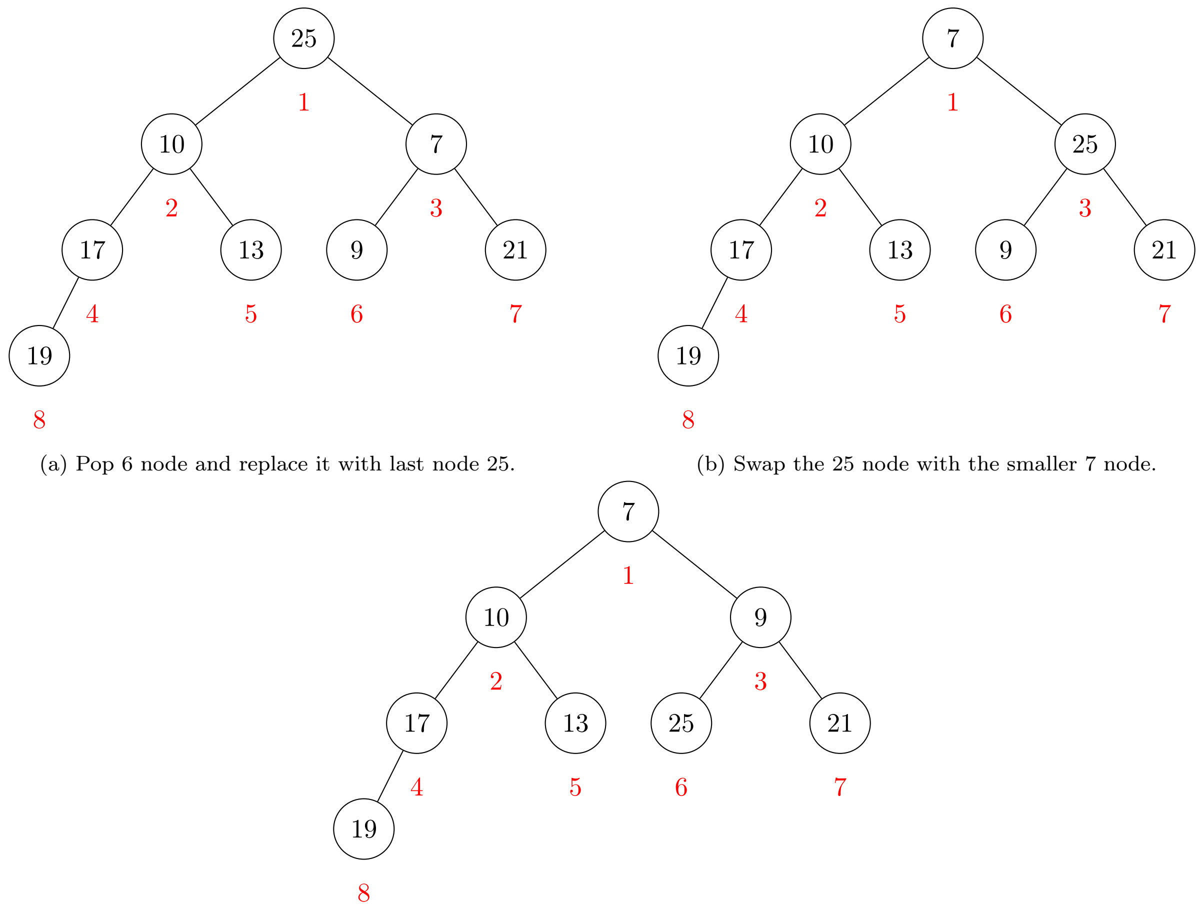
(a) Add the 11 node to the tree by inserting in the lowest depth, from left to right.



(b) Keep swapping the added node with its ancestor if the node is smaller.

Figure 6: Visual of how to add nodes to a heap.

2. We remove the first (minimal) value, we first replace the root with the last node in the heap, and while the heap property is violated, we swap with the smaller child. There are two choices, the left or right child, in which we can swap. But we must always swap with the **smaller child**, since we swapped with the bigger child, then this bigger child would be larger than the smaller one, violating the heap property. Since a complete binary tree always has height $O(\log(N))$, remove also "traverses" one root-leaf path, and so its runtime complexity is $O(\log(N))$, too. hi



(a) Pop 6 node and replace it with last node 25.

(b) Swap the 25 node with the smaller 7 node.

(c) Swap the 25 node with the smaller 9 node.

Figure 7: Binary heap restructuring during deletion operation

3. The decreaseKey operation just takes an arbitrary node and decreases its value to some other integer. In this case, it wouldn't violate the shape property, and to restore the heap property, we just put swap it with its parent if the new value is smaller than its parent, making this operation $O(\log(N))$.

Definition 3.5 (Priority Queues)

A priority queue simply adds things according to their priority. Every time we add an element, it looks at where the element should go to keep the list sorted. If we want to dequeue, then we just remove the first element of the list. If we implement a priority queue with a heap, we have

1. Adding a new value is $O(\log N)$.
2. Removing a specific value is $O(\log N)$.
3. Peek returns the minimal element and is $O(1)$.
4. Checking if a value is contained in a priority queue is ?.

3.3 Binary Search Trees

Definition 3.6 (Binary Search Tree)

A binary tree is a **binary search tree** if for every node, the left subtree values are all less than the node's value, and the right subtree values are all greater than the node's value. That is, the nodes are in order, and if we called `inOrder(root)` on the tree, then we would get a sorted list, which allows for efficient search.

Adding elements to a binary search tree is also very similar. But note that the order in which we add elements to the binary search tree will matter, since it can either make the tree **balanced** or **unbalanced**.

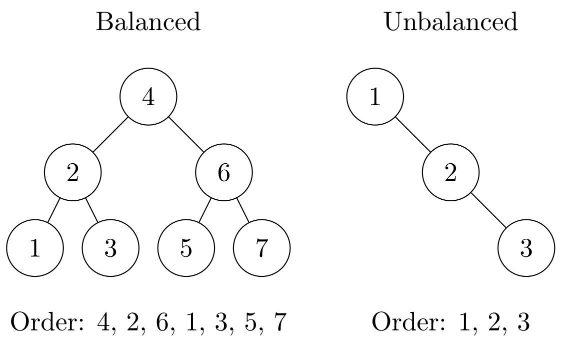


Figure 8: Comparison of balanced and unbalanced binary trees. In a balanced case, contains/add will be $O(\log(N))$, while in an unbalanced case, we will have $O(N)$.

Definition 3.7 (Binary Search)

Given that we have a sorted list (this is important!), we can search for the index of an element in $O(\log n)$ time. We want the loop invariant "if the target is in the array/list, it is in the range [low, high]." Let us have a list of N elements, and at every step, we either

1. get our desired element and its index, or
2. cut down our search space by half

Now, we have learned how we can implement a priority queue using a binary heap. This is also possible to use a binary search tree, since it's easy to get the minimal element for adding and removing, but there are three things that make it difficult:

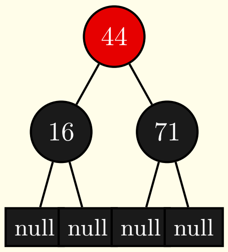
1. all elements must be unique
2. it is not array-based, and so uses more memory and higher constant factors on runtime
3. it is much harder to implement with guarantees that the tree will be balanced. This makes it difficult since if we want to search through a balanced BST, it is $O(\log(N))$, but if it turns out to be unbalanced, then it is $O(N)$.

Therefore, while a balanced tree may be efficient on average, in the worst case the linear complexity is not tolerable. Therefore, we must implement a binary search tree that will do extra work to ensure that they are approximately balanced. This is where *red-black trees*, which are a special type of binary search trees, come in.

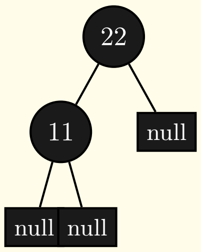
Definition 3.8 (Red-Black Tree)

Red-Black Trees are binary search trees consisting of nodes that are labeled either red or black that satisfy the following properties:

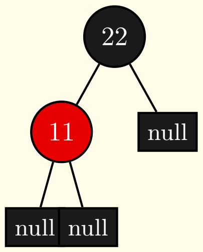
1. The root is black
2. A red node cannot have red children
3. From any node, all paths to null descendants must have the same number of black nodes. (Null is considered to be a black node)



(a) Invalid: Root node is red, violating the property that the root must be black.



(b) Invalid: Path (22,null) has 2 black nodes, but path (22,11,null) has 3, violating the black-depth property.



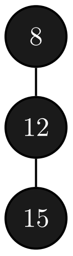
(c) Valid: Root is black, reds have no red children, all paths from node to null have same black counts.

Figure 9: Examples and non-examples of red-black trees.

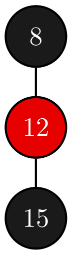
Remember that red black trees are also just binary search trees, and so some of the operations are the same.

1. contains (search) method is the exact same thing as BST
2. The add method needs to be slightly modified, since after we add, we need to make sure that the resulting tree is a red-black tree. This is done in three steps:
   (a) Run the regular BST add
   (b) Color the new node red
   (c) Fix the tree to reestablish red-black tree properties. This is extremely complicated with different cases, but it all essentially uses some sort of recoloring and a (right or left) rotation of the tree.

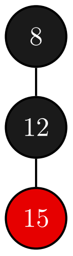
Note that there are binary search trees that cannot be turned into a red-black tree.



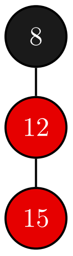
(a) Too many black nodes on right compared to left paths



(b) Too many black nodes on right compared to left paths



(c) Too many black nodes on right compared to left paths



(d) Red node with red child not allowed

Figure 10: Examples of Red-Black Tree violations.

This is intentional because red-black tree properties guarantee approximate balance. If we can turn a binary search tree into a red-black tree, then it logically follows that the original BST was approximately balanced. Note that a red black tree does not make searching asymptotically faster in any way; it just takes care of the worst-case.

3.4 Tries

4 Brute Force Algorithms

4.1 Basic Arithmetic

In here, we use basic deductions from elementary algebra to give us a starting point at which we analyze fundamental arithmetic algorithms.

Theorem 4.1 (Complexity of Addition)

The complexity of addition of two $O(N)$ values with n bits is
1. $O(n)$ bit complexity.
2. $O(\log N)$ complexity.
3. $O(1)$ memory complexity.

By the same logic, the complexity of subtraction is
1. $O(n)$ bit complexity.
2. $O(\log N)$ complexity.
3. $O(1)$ memory complexity.

Proof.

To see bit complexity, we are really taking each bit of each number and adding them together, plus a potential carry operation. Therefore, we are doing a bounded number of computations per bit, which is $O(1)$, but we must at least read through all of the bits, making this $O(\max\{n, m\})$.

Theorem 4.2 (Complexity of Multiplication)

The complexity of multiplication of two values N, M with bits n, m is
1. $O(n^2)$ bit complexity.[a]
2. $O((\log n)^2)$ complexity.
3.

[a] It turns out we can do better, which we will learn later.

Theorem 4.3 (Complexity of Division)

The complexity of multiplication of two values N, M with bits n, m is
1. $O(n^2)$ bit complexity.
2.

Theorem 4.4 (Complexity of Modulus)

The complexity of multiplication of two values N, M with bits n, m is
1. $O(n^2)$ bit complexity.
2.

Theorem 4.5 (Complexity of Exponentiation)

The complexity of multiplication of two values N, M with bits n, m is
1. $O(n^2)$ bit complexity.
2.

Theorem 4.6 (Complexity of Square Root)

The complexity of multiplication of two values N, M with bits n, m is

1. $O(n^2)$ bit complexity.
2.

Definition 4.1 (Factorial)

4.2 Lists

Definition 4.2 (Max and Min of List)

Definition 4.3 (Bubble Sort)

Definition 4.4 (Binary Search)

4.3 Stack, Queues, Heaps

A heap is sort of in between a sorted array and an unsorted array.

4.4 Cryptography

Example 4.1 (GCD of Two Numbers)

Take a look at the following algorithm.

```python
def gcd(a, b):
  if a == b:
    return a
  elif a > b:
    return gcd(a - b, b)
  else:
    return gcd(a, b - a)

print(gcd(63, 210))
```

Definition 4.5 (Primality Testing)

Definition 4.6 (Integer Factorization)

4.5 Matrix Operations

Definition 4.7 (Matrix Multiplication)

Definition 4.8 (Singular Value Decomposition)

Definition 4.9 (QR Decomposition)

Definition 4.10 (LU Decomposition)

Definition 4.11 (Matrix Inversion)

5 Divide and Conquer

Definition 5.1 (Divide and Conquer Algorithms)

The general idea is two steps:

1. Divide an input into smaller instances of the same problem. The simplest form is merge sort.
2. Conquer/solve these smaller instances, which takes less time.
3. Merge/Combine the smaller solutions into the original, bigger solution.

This is usually recursive, but does not need to be.

It has its applications in the most elementary operations, in sorting and multiplication. To prove correctness, we use induction by proving the correctness of the base case and then the inductive step to show that there is an invariant.

5.1 Recursive Algorithms and Recurrence Relation

I assume that the reader is familiar with recursive algorithms. Now to evaluate the runtime of a recursive algorithm, one must implicitly solve for the runtime of its recursive calls, and we can visualize it.

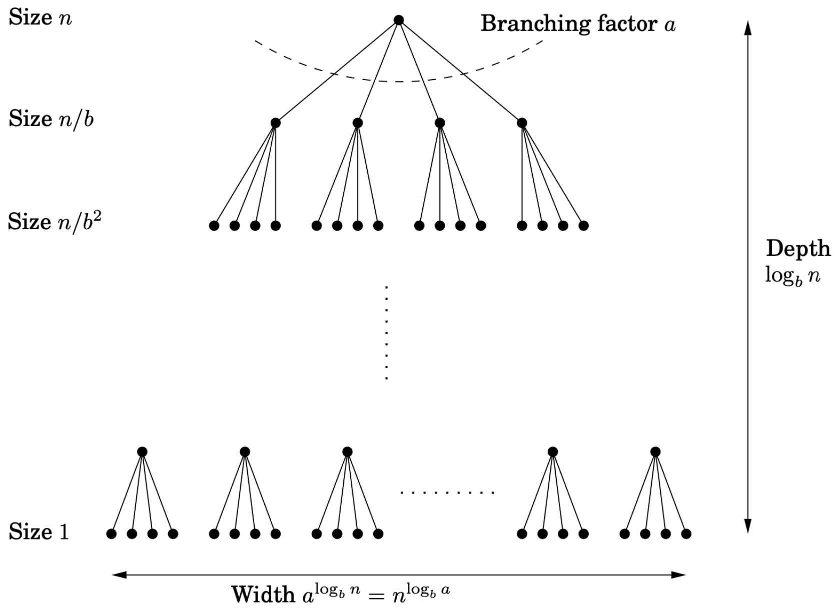


Figure 11

An important theorem for divide-and-conquer algorithms is

Theorem 5.1 (Master Theorem)

Given a recurrence relation of form

$$T(N) = aT(N/b) + O(N^c) \tag{24}$$

then the following holds

$$a > b^c \implies T(N) = O(N^{\log_b a}) \tag{25}$$
$$a = b^c \implies T(N) = O(N^c \log N) \tag{26}$$
$$a < b^c \implies T(N) = O(N^c) \tag{27}$$

Proof.

To intuit this, see that if $a > b^c$, then there arises a lot of subproblems, so our complexity is greater. If $a < b^c$, then we have few subproblems and can get a better runtime. If $a = b^c$, we get somewhere in between. To actually solve this, we can just unravel the recurrence to get the infinite series

$$\begin{aligned} T(N) &= aT(N/b) + O(N^c) && (28)\\ &= a^2 T(N/b^2) + N^c\left(\frac{a}{b^c} + 1\right) && (29)\\ &= N^c\left(1 + \frac{a}{b^c} + \frac{a^2}{b^{2c}} + \dots\right) && (30) \end{aligned}$$

So, if $a < b^c$, then even as $N \to \infty$, the sum is finite, so it is of order $O(N^c)$. If $a = b^c$, then the series is just $1 + \ldots + 1$, which scales on the order of $O(\log_2 N)$. If > 1, then we have to calculate the last term, which contributes to our runtime and overpowers c.
Therefore, if a is large, our algorithm will have an exponential number of subproblems and will be bottlenecked by the.

5.2 Merge Sort and Counting Inversions

Algorithm 5.1 (Merge Sort)

Merge sort is the first instance.

Algorithm 1 Merge Sort

```
Require: Array nums
  function MergeSort(nums)
    n = len(nums)
    if n < 2 then                                         ▷base case
      return nums
    end if
    mid = n // 2
    left_sorted = MergeSort(nums[:mid])                   ▷Divide into left half
    right_sorted = MergeSort(nums[mid:])                  ▷Divide into right half
    res = [0] * n
    i = j = k = 0
    while k < n do                                        ▷Merge the sorted subarrays
      if j == len(right_sorted) or left_sorted[i] < right_sorted[j] then
        res[k] = left_sorted[i]          ▷We should add the next element from left array
        i += 1                           ▷if it is smaller or if right array is filled already
      else
        res[k] = right_sorted[j]
        j += 1
      end if
      k += 1
    end while
    return res
  end function
```

The recurrence relation for the runtime is as follows. Let $T(n)$ represent the worst-case runtime of MergeSort of size n. Then, we have

$$T(n) = 2 \cdot T(n/2) + O(n) \tag{31}$$

Consisting of two recursive calls with input size $n/2$ and then the merge step which is $O(n)$. But if we take a look at this, we have

$$\begin{aligned} T(n) &= 2 \cdot T(n/2) + O(n) && (32)\\ &= 2 \cdot \big(2 \cdot T(n/4) + O(n/2)\big) + O(n) && (33)\\ &= 4 \cdot T(n/4) + O(n) + O(n) && (34) \end{aligned}$$

and the number of times $O(n)$ is added up is $\log n$, meaning that this recurrence relation turns into $O(n \log n)$.

Definition 5.2 (Inversions)

Given two lists of ranked items, say

$$\begin{aligned} \text{Alice :} a > b > c > d > e && (35)\\ \text{Bob :} b > d > a > e > c && (36) \end{aligned}$$

We want to measure the dissimilarity between two rankings by counting the number of *inversions*, which are pairs of items for which open persons orders the opposite of the other (e.g. a, b for above).[a] So how many inversions are there? We can do this in $\Theta(n^2)$ by explicitly looking at all n pairs. Without loss of generality, we can assume that the first list is sorted simply by bijectively relabeling these elements for both lists. Therefore, the set of inversions is defined to be

$$\{(i, j) \text{ s.t. } i < j \text{ and } \texttt{A[i] > A[j]}\} \tag{37}$$

[a]Also known as Kendall-Tau distance in statistics.

Algorithm 5.2 (Counting Inversions)

The idea is very similar. By assuming that the first list is sorted, we can simply count the number of inversions in a single list A.

```
A = 5 6 1 3 4 8 2 7
```

1. In the divide step, we count all inversions in A_l, A_r, which are the left and right sides of A, *and* we sort A_l, A_r to add additional structure.

   ```
   1 3 5 6 | 2 4 7 8
   ```

2. In the conquer step, we merge them linearly but every time we add an element from A_r into our result, this reveals that there are k additional inversions added where k is the number of elements left in A_l to add.

Algorithm 2 Counting Inversions

```
Require: Array nums
  function INVERSIONS(nums)
    n = len(nums)
    if n < 2 then                                                      ▷base case
      return nums
    end if
    mid = n // 2
    left_sorted, left_invs = Inversions(nums[:mid])          ▷Divide into left half
    right_sorted, right_invs = Inversions(nums[mid:])       ▷Divide into right half
    res = [0] * n
    i = j = k = 0                                 ▷left, right, and combined index
    inv = 0                                                ▷number of inversions
    while k < n do                                    ▷Merge the sorted subarrays
      if j == len(right_sorted) or left_sorted[i] < right_sorted[j] then
        res[k] = left_sorted[i]          ▷We should add the next element from left array
        invs += len(left_sorted - i)     ▷Increment inversions by # of elems in left array
        i += 1                           ▷if it is smaller or if right array is filled already
      else
        res[k] = right_sorted[j]
        j += 1
      end if
      k += 1
    end while
    return res, inv + left_invs + right_invs
  end function
```

The recursion relation is still

$$T(n) = 2 \cdot T(n/2) + O(1) + O(n) \implies O(n \log n) \tag{38}$$

5.3 Selection and Quick Sort

The next problem is a generalization of finding the median of an array, which we present can be done in $O(n)$ time *on average*. It turns out that that this idea of choosing a pivot and then dividing and conquering happens often in general selection and sort algorithms.

Algorithm 5.3 (Select kth Largest Element from Unsorted Array)

Algorithm 3 Find Kth Largest Element

```
Require: Array of numbers nums, integer k where 1 <= k <= length(nums)

  function FindKthLargest(nums, k)
      if length(nums) = 1 then                          ▷Base case: if array has only one element
          return nums[0]
      end if
      pivot ← $\lfloor$length(nums)/2$\rfloor$                        ▷Select middle element as pivot
      left ← [ ]                                  ▷Unsorted array for elements smaller than pivot
      right ← [ ]                                  ▷Unsorted array for elements larger than pivot
      n_pivots ← 0                                        ▷Count of elements equal to pivot
      for each n in nums do                                         ▷Start filling in the arrays
          if n = nums[pivot] then
              n_pivots ← n_pivots + 1
          else if n $\leq$ nums[pivot] then
              left.append(n)
          else
              right.append(n)
          end if
      end for
      if length(right) $\leq$ k - 1 and length(left) $\leq$ length(nums) - k then
          return nums[pivot]              ▷Found the $k$th element which was in the middle in n_pivots
      else if length(right) $\geq$ k then
          return FindKthLargest(right, k)               ▷Largest is in the array of bigger numbers.
      else
          return FindKthLargest(left, k - length(right) - n_pivots)      ▷Largest is in the array of
  smaller numbers.
      end if
  end function
```

Algorithm 5.4 (Quick Sort)

Algorithm 4 Quicksort Algorithm

```
Require: Array A of comparable elements

  function QUICKSORT(A, low, high)
      if low < high then
          p ← PARTITION(A, low, high)          ▷Get pivot position
          QUICKSORT(A, low, p - 1)             ▷Sort left subarray
          QUICKSORT(A, p + 1, high)            ▷Sort right subarray
      end if
  end function

  function PARTITION(A, low, high)
      pivot ← A[high]                          ▷Choose rightmost element as pivot
      i ← low - 1                              ▷Index of smaller element
      for j ← low to high - 1 do               ▷Scan through array
          if A[j] ≤ pivot then
              i ← i + 1                        ▷Increment index of smaller element
              swap A[i] and A[j]               ▷Swap current element with pivot
          end if
      end for
      swap A[i + 1] and A[high] ▷Put pivot in its correct position return i + 1 ▷Return pivot's final
  position
  end function
```

5.4 Closest Pair of Points

The next problem simply takes a series of points and calculates the closest pair of points. This can be done trivially in $O(N^2)$ by taking all combinations, but with clever divide and conquer, we can reduce this down. The idea is that we want to divide them into a left and a right side, which we can sort in $O(N \log N)$ and find the median in $O(1)$. This now reduces to computing the closest pair of points in each $N/2$ half. If we can find the closest pair in each half, then we must compare it to all pairs of points across the halves, meaning that we must do $(N/2)^2$ comparisons again, leading to

$$T(N) = 2T(N/2) + O(N^2) \tag{39}$$

which isn't any better than $O(N^2)$. But imagine that we found that the smallest distance of the left and right were δ_1, δ_2, then for each point in the left side, we don't have to check all $N/2$ points on the right.

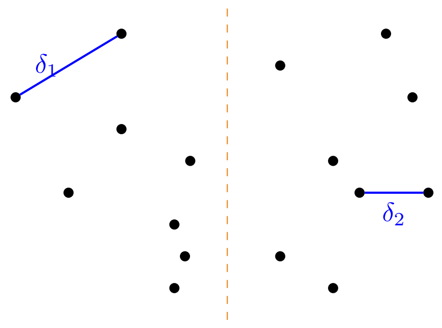


Figure 12: Set of points with two measured distances δ_1 and δ_2.

We just have to check those with distance at most $\delta = \min\{\delta_1, \delta_2\}$ from each point. Furthermore, we can discard all points that are too far away from the boundary. However, all N points could like in the relevant space, leading to $O(N^2)$ computations of distance.

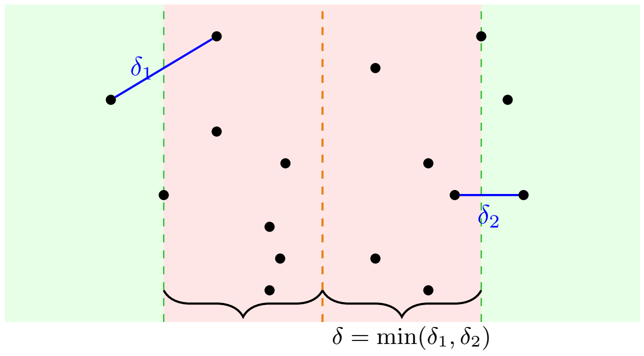


Figure 13: Closest pair algorithm with divide-and-conquer approach: only points within δ distance of the dividing line need to be checked for cross-boundary minimum distances. Note that if the closest pair consists of one point being in LHS and the other in the RHS of the red line, then it must lie in between the two green lines.

Therefore, we want to incorporate vertical distances and tile our relevant space into square of side length $\delta/2$.

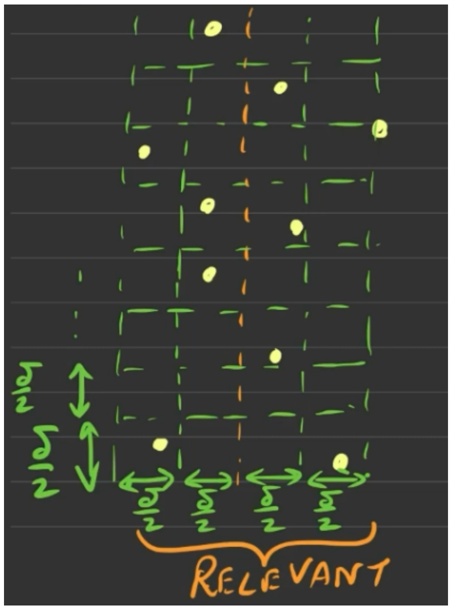


Figure 14

We claim that each square has at most 1 point, since if there were two, then their distance would be less than $\delta/\sqrt{2}$, which contradicts the distance between the two points being greater than δ. Therefore given a point, we only need to check a bounded constant.

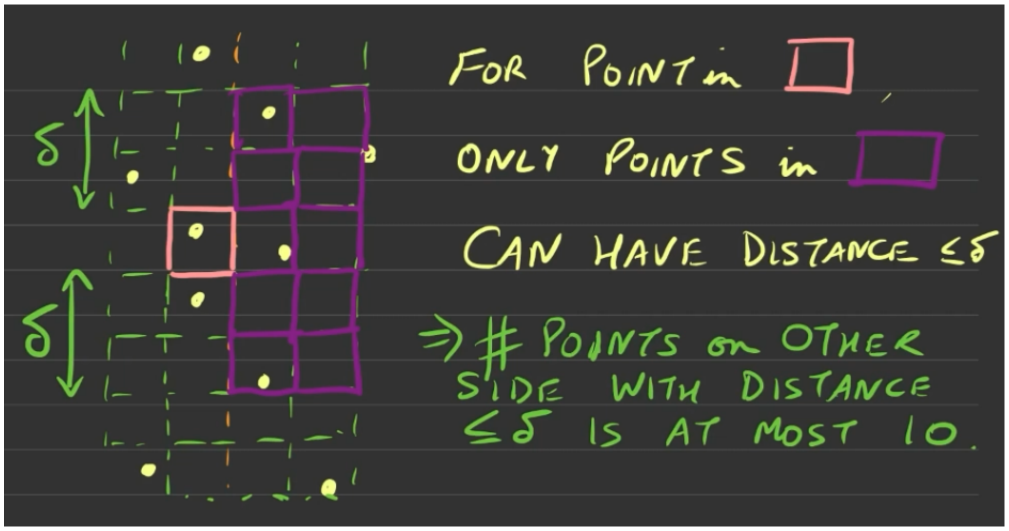


Figure 15: The number of points on the other side with distance $\leq \delta$ is at most 10. If we are more careful, we can reduce the number down to 7.

It turns out that we just need to compute $5N = (N/2) \cdot 10$ distances at most, and now our question reduces to how do we find these 5 points? Well we can first sort the points on the left and right by their y-coordinates, so we can just take a sliding window that encapsulates these points on the right for every on the left. The pointers of the sliding window should point to a point where the y-coordinate is at most δ away, and this can be done in constant time. Therefore, our recurrence relation is

$$T(N) = 2T(N/2) + O(N \log N) \implies T(N) = O(N \log^2 N) \tag{40}$$

This sorting along the y-coordinates is a bottleneck, but we can just shove this into the recursion step by telling the left and right to not just return the δ_1, δ_2, but also the points sorted in the y-coordinate. Then at the end of the merge, we also merge the lists L, R so that $S = L \cup R$ is sorted on y, which also takes $O(N)$ and doesn't add extra runtime on the merge step. This reduces the runtime to $O(N \log N)$.

Algorithm 5.5 (Closest Pair of Points)

The next problem simply takes a series of points and calculates the closest pair of points. This can be done trivially in $O(N^2)$ by taking all combinations, but with clever divide and conquer, we can reduce this down.

Algorithm 5 Closest Pair of Points

```
Require: $N$ points $\{(x_i, y_i)\}$.

  function ClosestPair(P)
      Sort points by x-coordinate.                          ▷Bottleneck of $O(N \log N)$
      if len(P) = 2 then
          return $d(p_1, p_2)$, sorted $P$ by y-coord
      end if
      $\delta_1, L \leftarrow$ ClosestPair($P_L$)
      $\delta_2, R \leftarrow$ ClosestPair($P_R$)
      $\delta = \min\{\delta_1, \delta_2\}$
      min $\leftarrow \delta$
      for $l \in L$ s.t. distance to boundary $\leq \delta$ do              ▷$O(N)$ iterations
          $W_l \leftarrow \delta$-window of points in $R$ around $l$.  ▷Can be done in $O(1)$ using sliding window.
          for $r \in W_l$ do                                  ▷This is bounded by $O(10)$
              if $d(p, l) < \delta$ then
                  min $\leftarrow d(p, l)$
              end if
          end for
      end for
      merge $L$ and $R$ into sorted $S$                                  ▷O(N)
      return min, S
  end function
```

5.5 Multiplication

5.5.1 Karatsuba Algorithm

5.5.2 Strassen Algorithm

We can solve matrix multiplication of two $N \times N$ matrices in a slightly more clever way than $O(N^3)$. Note that we can take the 2×2 block form of matrices A, B and multiply them to get $C = AB$, where

$$C_{11} = A_{11}B_{11} \cdot A_{12}B_{21} \tag{41}$$
$$C_{12} = A_{11}B_{12} \cdot A_{12}B_{22} \tag{42}$$
$$C_{21} = A_{21}B_{11} \cdot A_{22}B_{21} \tag{43}$$
$$C_{22} = A_{21}B_{12} \cdot A_{22}B_{22} \tag{44}$$

This requires us to compute a total of 8 $N/2 \times N/2$ multiplications and 4 additions, each of which is $O(N^2)$. Therefore, our recurrence relation is

$$T(N) = 8T(N/2) + O(N^2) \tag{45}$$

Using the master theorem, we find that $a = 8 > 2^2 = b^c$, so our runtime is $O(N^{\log_b a}) = O(N^3)$, which brings us right back to where we started. The problem with this is that $a = 8$, which is large. If we could get $a = 7$, then this would be an improvement. We want to reduce this number of multiplications, and we can do this using the Strassen algorithm, which uses the following values.

$$\begin{aligned}
P_1 &= (a_{11} + a_{22})(b_{11} + b_{22}) \\
P_2 &= (a_{21} + a_{22})b_{11} \\
P_3 &= a_{11}(b_{12} - b_{22}) \\
P_4 &= a_{22}(b_{21} - b_{11} \\
P_5 &= (a_{11} + a_{12})b_{22} \\
P_6 &= (a_{21} - a_{11})(b_{11} + b_{12}) \\
P_7 &= (a_{12} - a_{22})(b_{21} + b_{22})
\end{aligned}$$

Then, we claim that we the entries of C are

$$\begin{aligned}
c_{11} &= P_1 + P_4 - P_5 + P_7 \\
c_{12} &= P_3 + P_5 \\
c_{21} &= P_2 + P_4 \\
c_{22} &= P_1 + P_3 - P_2 + P_6
\end{aligned}$$

So we have reduced 8, 4 mult/add to 7, 18 mult/add. Addition is cheap and the number of additions is bounded, so now we have decreased a to 7.[5] We can then solve the new recurrence relation

$$T(N) = 7T(N/2) + O(N^2) \implies O(N^{\log_2 7}) \approx O(N^{2.81}) \tag{46}$$

5.6 Polynomial Multiplication with Fast Fourier Transform

Given as inputs 2 degree N polynomials,

$$A(x) = a_0 + a_1 x + a_2 x^2 + \ldots + a_{n-1} x^{n-1} \tag{47}$$

$$B(x) = b_0 + b_1 x + b_2 x^2 + \ldots + b_{n-1} x^{n-1} \tag{48}$$

we want to multiply them to $C(x) = A(x)B(x)$ defined

$$C(x) = c_0 + c_1 x + \cdots + c_{2n-2} x^{2n-2} \tag{49}$$

Clearly, we must multiply every coefficient in A with B, which takes $O(N^2)$ time. It is also called the convolution operation.

Convolution of $(a_0, a_1, \ldots, a_{n-1})$ and $(b_0, b_1, \ldots, b_{n-1})$

$$\begin{aligned}
c_0 &= a_0 b_0 \\
c_1 &= a_0 b_1 + a_1 b_0 \\
c_2 &= a_0 b_2 + a_1 b_1 + a_2 b_0 \\
c_3 &= a_0 b_3 + a_1 b_2 + a_2 b_1 + a_3 b_0 \\
&\vdots \\
c_{2n-2} &= a_{n-1} b_{n-1}
\end{aligned}$$

We can actually compute this convolution of two vectors in $O(N \log N)$ time using the FFT algorithm. Let's ease into this idea.

[5]We can reduce it even further, down to 2.37. Whether $O(N^2)$ is possible is an open problem.

Lemma 5.1 (Evaluating a Polynomial at x)

If we are given an x and want to evaluate $A(x)$, we can just incrementally evaluate the terms up each degree in $O(N)$ time.

Algorithm 6 Evaluate polynomial $A(x)$ at $x = p$

```
1: $S \leftarrow a_0$
2: $R \leftarrow x$
3: for $i = 1, 2, \ldots, n-1$ do
4:     $S \leftarrow S + a_i \cdot R$
5:     $R \leftarrow R \cdot x$
6: end for
```

Great, we can make some progress, but what does this have to do with finding the actual polynomial? Recall that from the fundamental theorem of algebra, a set of $n+1$ points will uniquely determine a nth degree polynomial. This at first glance doesn't help, since evaluating all $n+1$ points is $O(n^2)$, and even if we did, this doesn't really tell us how to reconstruct the polynomial in some fast time (e.g. matrix inversion won't work). But note that if we have a 1st degree polynomial, then evaluating it at ± 1 will retrieve the whole polynomial back

$$f(x) = a_0 + a_1 x \implies \begin{cases} f(+1) & = a_0 + a_1 \\ f(-1) & = a_0 - a_1 \end{cases} \implies \begin{cases} a_0 & = \frac{1}{2}\big(f(+1) + f(-1)\big) \\ a_1 = \frac{1}{2}\big(f(+1) - f(-1)\big) \end{cases} \tag{50}$$

We can think of this as sort of our base case. For Nth degree polynomials, we can divide it into a even and odd powers part.

$$\begin{aligned} A(x) &= a_0 + a_1 x + a_2 x^2 + \ldots && (51) \\ &= (a_0 + a_2 x^2 + a_4 x^4) + x(a_1 + a_3 x^2 + a_5 x^4 + \ldots) && (52) \\ &= A_{\text{even}}(x^2) + x A_{\text{odd}}(x^2) && (53) \end{aligned}$$

where each of the splits have degree $N/2$. Then we want to evaluate the even and odd parts.

Let's jump ahead and focus on the problem of evaluating $A(x)$ of degree N at the Nth roots of unity.

Example 5.1 ()

For $N = 4$, we evaluate at $\pm 1, \pm i$.

$$A(x) = (a_0 + a_2 x^2) + x(a_1 + a_3 x^2) \tag{54}$$

which gives us

$$\begin{aligned} A(+1) &= A_e(+1) + A_o(+1) && (55) \\ A(-1) &= A_e(+1) - A_o(1) && (56) \\ A(+i) &= A_e(-1) + iA_o(-1) && (57) \\ A(-i) &= A_e(-1) - iA_o(-1) && (58) \end{aligned}$$

Note that even though we had $\pm i$ evaluated on A, they were all squared in each split so evaluating the 4th of unity, which we denote $U(4)$, has been reduced to finding $U(2)$ for each of the left and right polynomials.

Therefore, to evaluate A of degree N at $U(N)$, it suffices to evaluate A_e, A_o each at $U(N/2)$, followed by combing them using addition and multiplication, which turns out to be $O(N)$.

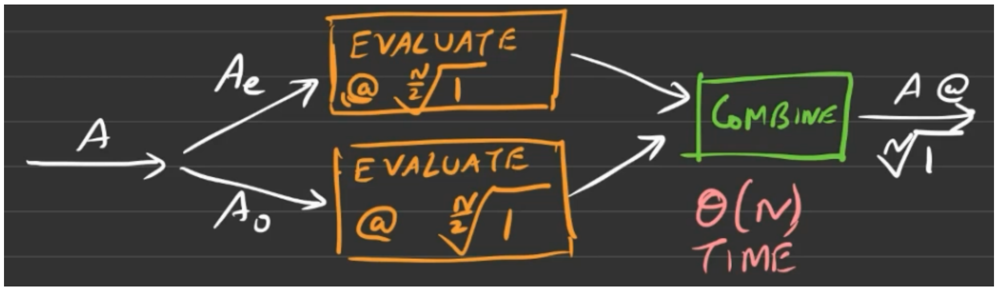

A
A_e
A_o
EVALUATE @ $\sqrt[\frac{N}{2}]{1}$
EVALUATE @ $\sqrt[\frac{N}{2}]{1}$
COMBINE
$\Theta(N)$ TIME
A @ $\sqrt[N]{1}$

Figure 16

Algorithm 5.6 (Evaluate Nth Degree Polynomial at Nth Roots of Unity)

Require:

```
function FUNC(x)
end function
```

Therefore, we have divided the problem of evaluating over $U(N)$ to be

$$T(N) = 2T(N/2) + O(N) \implies T(N) = O(N \log N) \tag{59}$$

So we have shown that in general, evaluating N points of a polynomial takes $O(N^2)$ time, but if you're clever about what points to evaluate, you can get $O(N \log N)$.

Now going back to the original problem, we can evaluate $A(u), B(u)$ for $u \in U(N)$, and then multiply them to get $C(u)$. Great. Now to reconstruct the polynomial using the roots of unity, it turns out that there is a method in $O(N \log N)$ time as well.

6 Greedy Algorithms

Greedy algorithms are easy to implement and easy to conceptualize, but they are often not correct. So the only interest in them is proving if they are correct. We have done greedy algorithms like Dijkstra and finding MSTs. Every step, greedy algorithms make an irrevocable decision of what to do and cannot undo it. That is, we cannot backtrack. In fact, some algorithms like Kruskal's algorithm for MSTs is precisely a greedy algorithm.

Example 6.1 (Interval Scheduling)

Let there be 1 classroom and n courses, each with a start and end time (s_i, e_i). We want to find the largest subset of courses that can be scheduled in a day such that they don't overlap. There are two things we want to do. How do we code this up? How do we show that a greedy algorithm will give a correct answer? Two greedy approaches are as follows.

1. Sort them by the start times, and add them as long as they do not overlap with what you have. This will not work.
2. Sort them by the time interval lengths and keep adding them as long as they do not overlap with what you have. This will not work either.

It turns out that if we sort based on finish time, this will work. So why is this correct? Assume that the first k decisions of this greedy algorithm are correct. Then, to find the next interval that we will include, the optimal algorithm must choose one that does not overlap with what we have. This is good since we do the same. The second part is that the next added interval must have the smallest end time t from all viable left intervals. Assume that it was not and that the optimal next interval had end time $s > t$. Then, the next interval must start after s, but this also starts after t, so we are sacrificing unnecessary extra space. An optimal solution with the endpoint at s can also be constructed with the same interval ending at t without anything to lose.

Algorithm 7 Find Max Classes to Fit into 1 Room

```
Require: classes $C = \{(s_i, e_i)\}_i$ of type List[tuple(int, int)]
  function SCHEDULE(C)
      res ← []
      sort $C$ by increasing end time
      for $s, e \in C$ do
          if $s <$ res[-1][-1] then                ▷If start overlaps with previous end time
              continue
          end if
          add $(s, e)$ to res                      ▷Otherwise add class to final schedule
      end for
      return res
  end function
```

To get the run time, the sorting takes $O(n \log n)$ time and iterating is $O(n)$, so a total of $O(n \log n)$ time.

Example 6.2 (Classroom Scheduling)

A slightly more practical example is that you have n classes and you want to minimize the number of classrooms m. For this, you can really use any order. The general idea is

1. Consider intervals in increasing start time s_i.
2. If it can be scheduled in an existing classroom, then schedule it there.
3. If not, then open a new classroom.

Even then, we can conduct a line search through all the time periods, incrementing the current

number of classes if an incoming time is a start time and decrementing if it is an end time. We report the max Δ over all times.
This algorithm is correct since by definition, Δ is the lower bound for the number of classrooms needed and the greedy algorithm attains it. The greedy algorithm reports Δ since if it reported $\Delta+1$ classroom, then at some time t it opened the $\Delta+1$th classroom. But this is impossible since this must mean that there are $\Delta+1$ classrooms concurrently in use, contradicting our assumption that Δ is the optimal solution.

Algorithm 8 Minimizing Number of Rooms With n Classes (Leetcode 253, Meeting Rooms II)

```
Require: classes $C = \{(s_i, e_i)\}_i$ of type List[tuple(int, int)]
  function MeetingRooms($C$)
      start ← all start times sorted increasing
      end ← all end times sorted increasing
      res ← 0                                   ▷Our max classes count
      count ← 0                                 ▷Our curr. classes count
      s, e ← 0                                  ▷pointers for start, end
      while s < len(start) do                   ▷Don't need to check for end since
          if start[s] == end[e] then            ▷start will always finish faster.
              s += 1, e += 1                    ▷One started and one ended, so don't change
          else if start[s] > end[e] then
              e += 1, count -= 1                ▷One ended, so decrement count
          else if start[s] < end[e] then
              s += 1, count += 1                ▷One started, so increment count
              res = max(res, count)             ▷Update res if we got past it
          end if
      end while
      return res
  end function
```

The runtime is $O(n \log n)$, which is to sort. However, you don't even need to sort since you can go through all intervals in any order and place it in an existing classroom or open a new classroom.

The next one isn't as trivial, but requires us to devise a custom sorting method by comparing two sequences with a swap difference.

Example 6.3 (Quiz Problem)

Suppose you have n questions with question i having reward r_i, but the probability that you get it correct is p_i. You keep collecting the reward until you get one question wrong, and the game terminates. What order would you answer the questions in?
Suppose we have $q = (10, 5)$ and $r = (0.1, 0.2)$.

1. If you choose the first and then second, then the expected return is
$$10 \cdot 0.1 + 5 \cdot 0.1 \cdot 0.2 = 1.1 \tag{60}$$
2. If you choose the second and then first, then the expected return is
$$5 \cdot 0.2 + 10 \cdot 0.1 \cdot 0.2 = 1.2 \tag{61}$$

Clearly, finding all possibilities is too computationally expensive since it increases exponentially w.r.t. n. Intuitively, if I have a question with a high reward, I want to answer it first, but if I have a low probability of getting it correctly, then I can't answer future questions if I get it wrong. So I have to balance these two forces. So we want to sort the score of each question with some function $f(r_i, p_i)$, which is increasing in both r_i and p_i. It is not $r_i p_i$, but this is a good start.

Rather, we can take a different approach. Assume that the tuples were sorted in some order, but this was not the optimal order. Then this indicates that we can swap to adjacent elements and it will get a better order. This is really like bubble sort, and now we need to find out the conditions on which it can be improved.

1. If we answer $q_1 \to q_2 \to q_3 \ldots$, our expected reward is
$$\mathbb{E}[R] = r_1 p_1 + r_2 p_1 p_2 + r_3 p_1 p_2 p_3 \tag{62}$$
2. If we swap q_2 and q_3 and answer $q_1 \to q_3 \to q_2 \ldots$, then our expected reward is
$$\mathbb{E}[R] = r_1 p_1 + r_3 p_1 p_3 + r_2 p_1 p_2 p_3 \tag{63}$$
Note that the swap does not affect higher order terms. Doing some algebra, the swap is better iff
$$r_2 p_2 (1 - p_3) < r_3 p_3 (1 - p_2) \implies \frac{r_2 p_2}{1 - p_2} < \frac{r_3 p_3}{1 - p_3} \tag{64}$$
where the final swap saves computational time since we can compute for a single element rather than comparing pairwise elements. So, we sort it (in descending order!) according to these values to maximize this value. Note that the numerator measures your expected reward, while the denominator measures the probability of you screwing up.

Not all greedy problems admit to scoring functions however. This was just one example.

Example 6.4 (Minimizing Max Lateness)

Suppose there are n jobs that all come in at once at time 0, with job j having length p_j and deadline d_j. We want to schedule the order of jobs on one machine that can handle one job at a time, where you must minimize the maximum lateness, where the lateness of job i is $\max\{f_i - d_i, 0\} = [f_i - d_i]^+$. For example, given jobs $(p, d) = \{(10, 9), (8, 11)\}$, we can run it in two ways.

1. Running 1 then 2. The first job finishes at 10 and the second at 18. The lateness is $10 - 9 = 1$ and $18 - 11 = 7$ respectively, so the maximum lateness is 7.
2. Running 2 then 1. The first job finishes at 8 and the second at 18. The lateness is 0 and $18 - 9 = 9$, so the maximum lateness is 9.

Therefore, $1 \to 2$ beats $2 \to 1$. Clearly, brute forcing this over $N!$ jobs orderings is unfeasible, but there is a greedy approach to this. We can

1. schedule jobs in increasing order of deadlines.
2. try to exchange by taking a sequence, swapping it, and computing the score like we did for previous examples.

Both lead to the same score/principle that we should schedule jobs in increasing order of deadlines. Let's prove this by taking jobs i and j, where $d_j \leq d_i$. Then, if we schedule $i \to j$, then we have
$$l_i = [t + p_i - d_i]^+ \tag{65}$$
$$l_j = [t + p_i + p_j - d_j]^+ \tag{66}$$
with l_j being the greatest late time. If we did $j \to i$, then we have
$$\hat{l}_j = [t + p_j - d_j]^+ \tag{67}$$
$$\hat{l}_i = [t + p_i + p_j - d_i]^+ \tag{68}$$
with $\hat{l}_i$ being the greatest late time. But this is a good sacrifice since we can see that both
$$l_j > \hat{l}_j \text{ and } l_j > \hat{l}_i \tag{69}$$
meaning that if we swap, then we will decrease the lateness of j at the smaller cost of increasing that of i. Therefore, $j \to i$ is better. Note that even though the end times of the second job will always

be the same between the two choices, starting with j will give a later deadline time. So this means that in the optimal solution, we have to place them in this order of deadlines since we can always improve them by swapping.

Example 6.5 (Gift Wrapping a Convex Hull)

Given n points in $\mathbb{R}^2$, we want to find the subset of points on the boundary of its convex hull. We can intuit this by taking the bottom most point and imagine "wrapping" a string around the entire region.[a] Here's a basic fact. If we have a point $p_1 = (x_1, y_1)$ and we're looking at $p_2 = (x_2, y_2)$, and then $p_3 = (x_3, y_3)$ shows up, then we can look at the cross product

$$P = (x_2 - x_1)(y_3 - y_1) - (y_2 - y_1)(x_3 - x_1) \tag{70}$$

and if $P > 0$, then p_3 is counterclockwise from p_2. The algorithm is as such.

1. You choose the bottommost point and label it p_1.
2. You construct a vector pointing to the right and start turning it counterclockwise until it hits the first point. This can be done by iterating through all of the n points and computing the angle.
$$\tan\theta = \frac{y_i - y_1}{x_i - x_1} \tag{71}$$
and the points with the smallest such tangent value will be the desired point, which will be $\Theta(n)$. We set this point to be p_2 and claim that p_1, p_2 is in the convex hull. This is true since all point must lie above the line $p_1 p_2$. Now you do the same thing with p_2 starting with a vector facing to the right and rotating it left until it hits the first peg. Once you get to the topmost point, you can start the vector from the right and rotate it counterclockwise, but this is an implementation detail.
If your convex hull has L vertices, then this algorithm will have $O(nL)$, which will be $O(n^2)$ worst case but if we sample uniformly, then we have $L \approx \log n$ and on average it is $O(n \log n)$.

A lot of geometric algorithms is output sensitive.

[a]This is called the gift wrapping algorithm.

Example 6.6 (Convex Hull with Graham Scan)

In fact, we can take a different approach and get $O(n \log n)$ time.

1. You still start with p_1 with the minimum y-coordinate and we sort in increasing $\theta_i = \arctan \frac{y_i - y_1}{x_i - x_1}$.
2. Then it will walk from p_i to p_{i+1} and check at p_{i+1} if it is on the convex hull or not.
   (a) If you turned counterclockwise to go to p_{i+1} then it's good and you're on the convex hull.
   (b) If you turned clockwise it is bad, and p_i is not on the convex hull, so remove p_i. However, p_{i-1} could also be clockwise as well, so you must backtrack and keep on removing points until you find a point p_j such that the previous turn is counterclockwise.
   Note that this is implemented as a stack.

This can be calculated in $O(1)$ by the previous cross product formula.

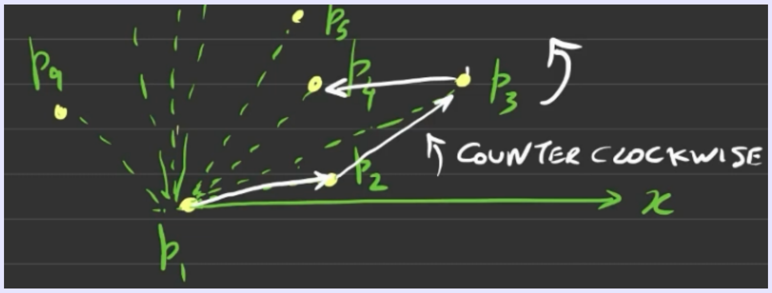


Figure 17: Overview of the steps mentioned above.

Algorithm 9 Finding Convex Hull with Graham Scan

```
Require: $p_1, \ldots, p_n \in \mathbb{R}^2$
  function GRAHAM($\mathbf{p}$)
      s ← stack()
      push $p_1, p_2$ onto s
      for $i = 3, 4, \ldots, n$ do
          push $p_i$ onto stack
          while top 3 on stack are clockwise do
              pop second from top point
          end while
      end for
      return s
  end function
```

Note that once we have 3 points, the first three in the stack will always be counterclockwise, so we will never need to check if there are enough points in the stack. Even in the beginning, the next point (third) added is guaranteed to be counterclockwise because of ordering. Note that even though we might do a bunch of pushes and pops, the maximum number of pushes we can do is n and pops is n, so this is $O(n)$. In fact, the bottleneck is the sorting, which is $O(n \log n)$, so the total runtime is $O(n \log n)$.

6.1 Huffman Coding

Now, let's talk about **Huffman encoding**. We already know about the ASCII encoding that encodes characters in 7 bits, for a total of 2^7 possibilities. The extended ASCII uses 8 bits, but all of these things use something called **fixed-length encoding** which uses a constant number of bits to encode any character. To compress something, we want to use **variable-length encoding**.

To decode something, the mapping from the characters to the bits must be injective, so we can define an inverse over its image. It turns out that if we an encoding for 3 characters, say

$$a \mapsto 1, b \mapsto 10, c \mapsto 11$$

then decoding 1011 is ambiguous since 1 is a prefix of the encoding for c. Therefore, we do not want one encoding to be the prefix of another. It turns out that we can avoid this conflict by encoding everything as a binary tree and setting all encodings as leaf nodes.

Prefix property encoding as a tree

char	**binary**
‘s’	000
‘_’	001
‘p’	010
‘h’	0110
‘o’	0111
‘g’	10
‘e’	110
‘r’	111

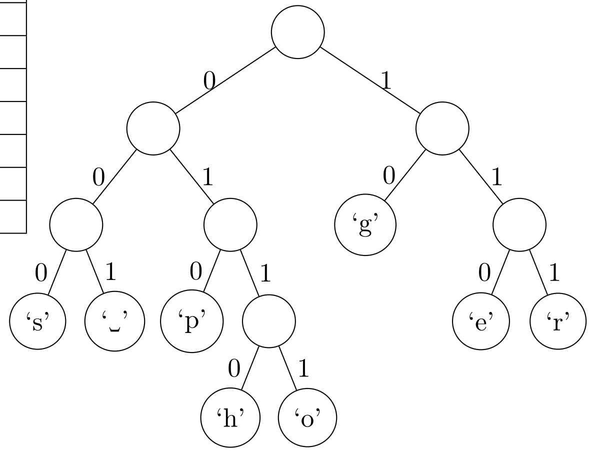


Figure 18: Tree-based representation of prefix codes showing how different characters are encoded with variable-length binary sequences. The convention uses 0 for left branches and 1 for right branches. The encoding for each character is the sequence of 0’s and 1’s on the path from root to leaf. Characters are placed at leaf nodes to ensure the prefix property, with values deeper in the tree requiring more bits than those higher up.

This makes sense since an encoding will be a prefix of another if and only if it is a parent of another. Furthermore, a greater depth of a character in the tree corresponds to a longer encoding. So, Huffman encoding tries to convert shorter characters to longer leaves and less recurrent characters into longer encodings. To decode a string of bits using a tree, we read the bit at a time to traverse left or right edge. When we reach a leaf, we decode the character and restart at root.

Now, we describe the greedy algorithm for building an optimal variable length encoding tree.

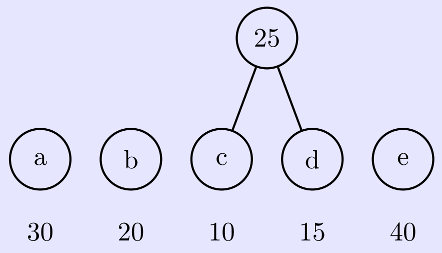
1. We take the document and compute the frequencies of all characters that we want to encode. We want to less frequent characters to be lower on the tree, and so we will build the tree up from the leaves.
2. We iteratively choose the lowest weight nodes to connect up to a new node with weight = sum of children.

We implement this using a priority queue.

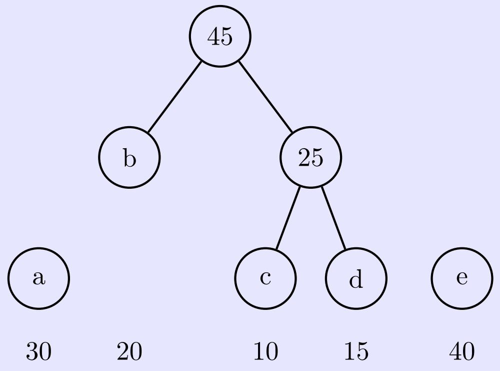
Example 6.7 ()

Let us go do an example of where we have the characters and frequencies

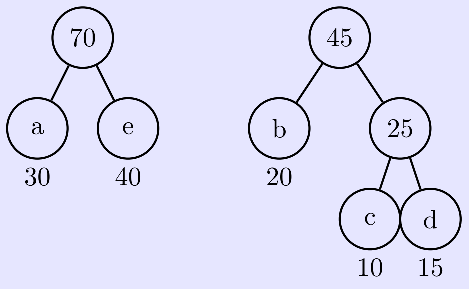
$$a \mapsto 30,\ b \mapsto 20,\ c \mapsto 10,\ d \mapsto 15,\ e \mapsto 40 \tag{72}$$



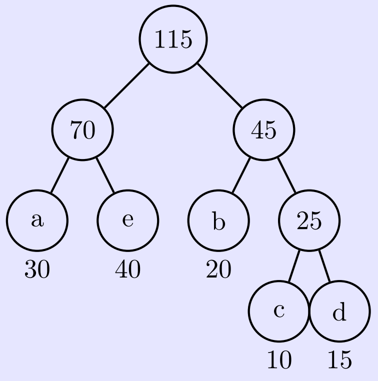
(a) We write out them as leaf nodes with the values $30, 20, 10, 15, 40$. We take the smallest of the frequencies and sum them up: $10 + 15 = 25$.



(b) We have the values $30, 20, 25, 40$. We sum the smallest two frequencies: $20 + 25 = 45$.



(c) We have the values $30, 45, 40$. We sum the smallest two frequencies: $30 + 40 = 70$.



(d) We have the values $70, 45$. We sum them up to get the complete tree.

Figure 19: Building a Huffman encoding tree step by step

Theorem 6.1 (Number of Nodes in Huffman Tree)

If we have a document of N total characters and M unique characters, the number of nodes in the Huffman tree, in complexity notation, is

$$O(M) \tag{73}$$

Clearly, this has nothing to do with N. Note that we have M leaf nodes, and in each iteration, we connect 2 nodes up to a parent. Therefore, the number of nodes to connect up decreases by 1 per iteration, and we create a new node per iteration. Since there are $M - 1$ iterations, we add one node, so there will be $M + M - 1 = O(n)$ nodes in the binary tree.

7 Graph Algorithms

A huge portion of problems can be solved by representing as a *graph* data structure. In here, we will explore various problems that can be solved through *graph algorithms*.

7.1 Representations and Properties

All graphs consist of a set of vertices/nodes V and edges E. This tuple is what makes up a graph. We denote $|V| = n, |E| = m$.

Definition 7.1 (Undirected Graphs)

An **undirected graph** $G(V, E)$ is a tuple, where $V = \{v_1, \ldots, v_n\}$ is the vertex set and $E = \{\{v_i, v_j\}\}$ is the edge set (note that it is a set of sets!).

1. The **degree** d_v of a vertex v is the number of edges incident to it.
2. A **path** is a sequence of vertices where adjacent vertices are connected by a path in E. It's **length** is the number of edges in the path.
3. A **cycle/circuit** is a path that has the same start and end.
4. A graph is **connected** if for every pair of vertices $e_i, e_j \in E$, there is a path from e_i to e_j.
5. A **connected component** is a maximal subset of connected vertices.

Definition 7.2 (Directed Graph)

A **directed graph** $G(V, E)$ is a tuple, where $V = \{v_1, \ldots, v_n\}$ is the vertex set and $E = \{(v_i, v_j)\}$ is the edge set (note that it is a set of tuples, so $(i, j) \neq (j, i)$).

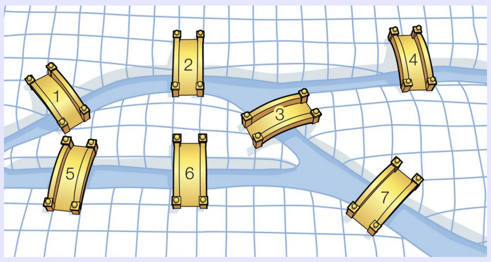
1. The **in/out degree** $d_{v,i}, d_{v,o}$ of a vertex v is the number of edges going in to or out from v.
2. A **path** is a sequence of vertices where adjacent vertices are connected by a path in E. It's **length** is the number of edges in the path.
3. A **cycle/circuit** is a path that has the same start and end.
4. A directed graph is **strongly connected** if for every pair of vertices $e_i, e_j \in E$, there is a path from e_i to e_j.[a]
5. A **strongly connected component** is a maximal subset of connected vertices.

[a]Obviously, a connected undirected graph is also strongly connected.

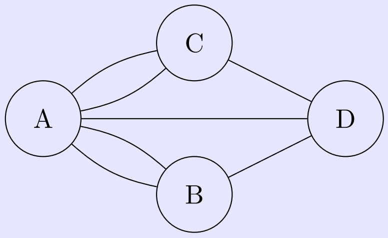
In fact, from these definitions alone, we can solve an ancient puzzle called *the Bridges of Konigsberg*. Euler, in trying to solve this problem, had invented graph theory.

Example 7.1 (Bridges of Konigsberg)

Is there a way to walk that crosses each bridge *exactly* once?



(a) Figure of the bridges of Konigsberg.



(b) Graph representation.

Figure 20: It can be decomposed into this undirected graph.

Euler's observation is that except for start and end points, a talk leaves any vertex by different edge that the incoming edge. Therefore, the degree (number of edges incident on it) must have an even number, so all but 2 vertices must have an even degree. Since every vertex has an odd degree, there is no way of doing it.

In addition to the *adjacency list* representation, another way in which we represent a directed graph is through *adjacency matrices*.

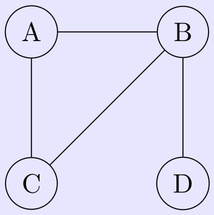
Definition 7.3 (Adjacency Matrix)

In a finite directed graph (V, E), we can construct a bijection from V to the natural numbers and so we label each element in V with $i \in \mathbb{N}$. Then, we can construct a matrix A such that

$$A_{ij} = \begin{cases} 1 & \text{if } (i,j) \in E \\ 0 & \text{if } (i,j) \notin E \end{cases} \tag{74}$$

Example 7.2 (Adjacency List vs Matrix)

Given a graph, we can completely represent it with a list of adjacent vertices for each vertex or an adjacency matrix.



An adjacency list would look something like this

$$\begin{aligned} A &: B, C \\ B &: A, C, D \\ C &: A, B \\ D &: B \end{aligned}$$

and the adjacency matrix looks like this:

$$\begin{array}{ccccc} & A & B & C & D \\ A & 0 & 1 & 1 & 0 \\ B & 1 & 0 & 1 & 1 \\ C & 1 & 1 & 0 & 0 \\ D & 0 & 1 & 0 & 0 \end{array}$$

While the adjacency matrix does have its advantages and has a cleaner form, usually in sparse graphs this is memory inefficient due to there being an overwhelming number of 0s.

Definition 7.4 (Trees)

An undirected graph $G(V, E)$ is a **tree** if

1. G is connected.

2. G has no cycles.[a]

Removing the first requirement gives us the definition of a **forest**, which is a collection of trees. Conversely, if $G(V, E)$ is connected and $|E| = n - 1$, then G is a tree.

[a] This makes sense, since to get back to a previous vertex you must backtrack.

Theorem 7.1 (Properties of Trees)

If $G(V, E)$ is a tree, then

1. There exists a $v \in V$ s.t. $d_v = 1$, called a **leaf node**.
2. $|E| = |V| - 1 = n - 1$.

Proof.

The outlines are quite intuitive.

1. There must be some leaf node since if there wasn't, then we would have a cycle. We can use proof by contradiction.
2. We can use proof by induction. We start off with one vertex and to construct a tree, we must add one edge and one vertex at every step, keeping this invariant.

7.2 Exploration

Given two $v, s \in V$ either directed or undirected, how can we find the shortest path from v to s? We can do with either with DFS or BFS.

Now, in order to traverse this graph, we basically want to make an algorithm that starts at a node, prints it value, and then goes to all of its neighbors (which we can access through the adjacency list) to print them out. Thus, this is by nature recursive. We don't want the algorithm to loop around printing nodes infinitely often, so we must create a base case that tells the algorithm to not print out a node. It makes sense to create a set of visited nodes, which we can add to whenever we reach a new node. So, if we ever come onto a node that we have visited, we can just tell the function to do nothing.

Algorithm 7.1 (Recursive Depth-First Search)

The recursive implementation of Depth-First Search explores a graph by recursively visiting each unvisited neighbor, going as deep as possible along each branch before backtracking.

```
Require: Graph $G(V, E)$, start vertex $s$
function DFS-Recursive($G, s$)
    visited ← $\emptyset$                    ▷Initialize empty set of visited vertices
    DFS-Visit($G, s$, visited)
end function
function DFS-Visit($G, u$, visited)
    Add $u$ to visited
    /* Process vertex $u$ here */
    for each neighbor $v$ of $u$ in $G$ do
        if $v \notin$ visited then
            DFS-Visit($G, v$, visited)
        end if
    end for
end function
```

Though recursion really makes this simple, we can construct an iterative approach that uses stacks. Note that in recursion, we are really making a call stack of different functions. We can be explicit about this by actually implementing a stack, which would store all the nodes that we have discovered, but not yet explored from)i.e. all the current nodes). At each iteration, we would pick a node to continue exploring, and since this is a DFS, we would want to implement a LIFO stack so that the last element we input in is the first thing that we should explore from, i.e. we always explore from the last node discovered.

Algorithm 7.2 (Iterative Depth-First Search)

The iterative implementation of Depth-First Search uses a stack to mimic the function call stack of the recursive implementation. It explores vertices in the same order as the recursive version.

```
Require: Graph $G(V, E)$, start vertex $s$
function DFS-ITERATIVE($G, s$)
    visited $\leftarrow \emptyset$                    ▷Initialize empty set of visited vertices
    stack $\leftarrow$ empty stack
    Push $s$ onto stack
    Add $s$ to visited
    while stack is not empty do
        $u \leftarrow$ Pop from stack
        /* Process vertex $u$ here */
        for each neighbor $v$ of $u$ in $G$ do
            if $v \notin$ visited then
                Add $v$ to visited
                Push $v$ onto stack
            end if
        end for
    end while
end function
```

Theorem 7.2 (Runtime of DFS)

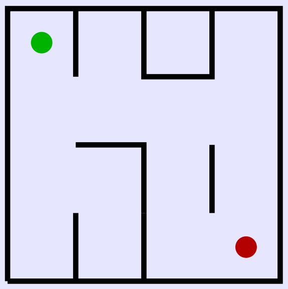
The runtime of DFS is $O(n + m)$.

Proof.

The runtime complexity of this search is $O(N + M)$ because first, the while loop loops at most over the N nodes. The for loop may loop over M edges, but this is a bit pessemistic in bound. Rather, we can view it as looping over neighbors of each node at most exactly once, and so it considers every edge twice, meaning that the for loop will get called $2M$ times in the entire algorithm. So $N + 2M = O(N + M)$.

Example 7.3 (DFS in a Maze)

We can represent a grid graph, like a maze, with a two dimensional array that stores whether it is connected north, east, south, and west, where boolean of true represents that there is a wall, and false means there isn't a wall (so connected).


(a) 4×4 maze representation

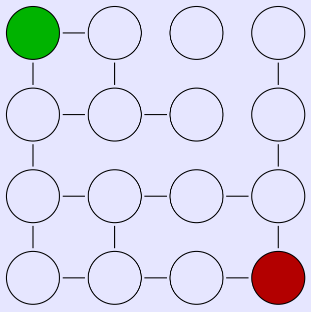

(b) Graph representation of the maze.

Figure 21: A 4×4 maze (left) and its corresponding graph representation (right). Start and end points are shown in green and red respectively.

But remember that in a tree traversal, we recursively searched down and down until we hit a null node, in which case we backtrack up to look in another branch. For graphs, this is a bit more complicated, since we could go in loops. Therefore, we want to keep track of all the visited nodes to avoid infinite recursion. We have three base cases:

1. If we search off the grid, then this is not a valid path
2. If we already explored here, then we don't want to repeat it
3. If we reached the goal of the maze, then we output the length of the path.

The recursive case would take each node and recurse on its 4 adjacent neighbors, if they are connected. Note that this algorithm recurses on each of the N nodes 4 times (for each direction, and each recursive call is $O(1)$), so the complexity is $O(N)$.

Note that the main idea of DFS is to always explore a new adjacent vertex if possible, and if not, then backtrack to the most recent vertex adjacent to an unvisited vertex and continue. On the contrary, the main idea of BFS is to explore *all* your neighbors before you visit any of your neighbors' neighbors. It exhaustively searches for the closest regions of your search space before you look any further. Unlike DFS, which finds the some arbitrary path to a node, BFS finds the shortest (perhaps non-unique) path to a node. This can be simply done with a queue.

Algorithm 7.3 (Iterative Breadth-First Search)

Breadth-First Search (BFS) explores a graph by visiting all neighbors at the current depth before moving to nodes at the next depth level. It uses a queue to process nodes in the order they are discovered.

```
Require: Graph $G(V, E)$, start vertex $s$
function BFS($G, s$)
    visited ← $\emptyset$                          ▷Initialize empty set of visited vertices
    queue ← empty queue
    Enqueue $s$ onto queue
    Add $s$ to visited
    while queue is not empty do
        $u$ ← Dequeue from queue                   ▷Get the next vertex to process
        /* Process vertex $u$ here */
        for each neighbor $v$ of $u$ in $G$ do
            if $v \notin$ visited then
                Add $v$ to visited
                Enqueue $v$ onto queue
            end if
        end for
    end while
end function
```

Theorem 7.3 (Runtime of BFS)

The runtime of BFS is $O(n + m)$.

Proof.

To get the running time, we know that each vertex is popped only once from the queue, giving us $O(n)$. For each pop, we are exploring all the neighbors of V.

$$O\left(\sum_{v \in V} |\text{neighbors of } v| + 1\right) = O\left(\sum_{v \in V} d_v + 1\right) \tag{75}$$

$$= O(2|E| + |V|) = O(m + n) \tag{76}$$

which is linear in input size!

The more straightforward application is in reachability.

Example 7.4 (Reachability)

Given a directed graph and a node v, find all nodes that are reachable from v.

Exercise 7.1 ()

Prove that in any connected undirected graph $G = (V, E)$ there is a vertex $v \in V$ s.t. G remains connected after removing v.

Proof.

Let u be such a leaf node of T, and let G' be the subgraph of G resulting by removing u and its incident edges from G. For sake of contradiction,[a] suppose G' has more than one connected component. Let C be a connected component in G' that does not contain s, the root of the BFS tree T. Since G was connected before the removal of u, it must be that every path from s to any vertex v in C includes

u (otherwise there would remain a path from s to v in G' and s would be in C). Then u is the only vertex not in S with edges to vertices in S, so all vertices in C must be "visited" during BFS only after visiting u. Furthermore, the vertices of S must be in the subtree of T rooted at u. But u is a leaf, which is a contradiction.

[a] We provide an alternative direct proof as follows: Since G is given to be connected, T contains all vertices of G. Let T' be the BFS tree minus u and its single incident edge connecting it to its parent in T. Since u is a leaf, T' remains a connected tree with all other vertices of G. The edges of T' exist in G', so G' is connected.

Exercise 7.2 ()

Two parts.

1. Give an example of a strongly connected directed graph $G = (V, E)$ s.t. that every $v \in V$, removing v from G gives a directed graph that is not strongly connected.
2. In an undirected graph with exactly two connected components, it is always possible to make the graph connected by adding only one edge. Give an example of a directed graph with two strongly connected components such that no addition of one edge can make the graph strongly connected.

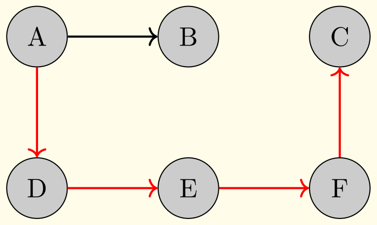
Proof.

Listed.

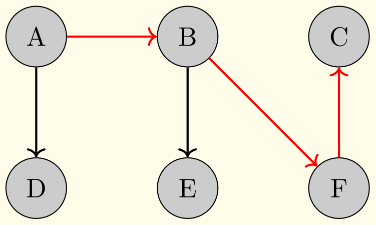
1. A graph whose edges form a cycle, having at least three nodes.
2. Two strongly connected components with no edges between them.

Definition 7.5 (Search Trees)

Once we have traversed a graph using BFS or DFS, we can label the directed path that this traversal algorithm takes into a **search tree**.



(a) After DFS traversal, we can store the previous nodes in a hashmap $\{B \mapsto A, D \mapsto A, E \mapsto D, F \mapsto E, C \mapsto F\}$. From this we can see the path to get to C is $A \mapsto D \mapsto E \mapsto F \mapsto C$ of length 4.



(b) After BFS traversal, we can store the previous nodes in a hashmap $\{B \mapsto A, D \mapsto A, E \mapsto B, F \mapsto B, C \mapsto F\}$. From this we can see the path to get to C is $A \mapsto B \mapsto F \mapsto C$ of length 3.

Figure 22: Comparison of two directed graphs with different path lengths from A to C.

By construction, we can see that the path from A to C is always shorter for BFS than for DFS.

7.3 Directed Acyclic Graphs and Topological Sorting

Definition 7.6 (Directed Acyclic Graph)

A DAG is a directed graph that has no cycles. Note that a DAG must have a node that has no in-edges.

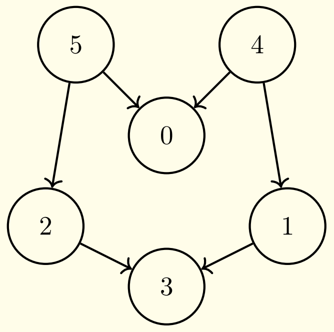
To determine if a graph is a DAG, then we can brute force it by taking a node $s \in V$, running DFS/BFS, and if a neighbor is already in visited, return False. Then go through this for all starting nodes $s \in V$. This again has quadratic runtime. Can we do better? This introduces us to topological sorting.

It may be helpful to take a graph $G(V, E)$ and induce some partial order on the set of nodes V based off of E. It turns out that we can do this for a specific type of graph.

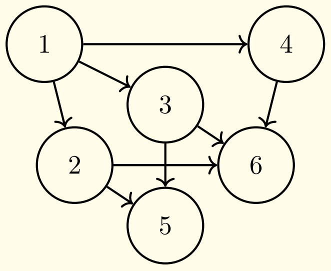
Definition 7.7 (Topological Sort)

Given a directed acyclic graph (DAG), a linear ordering of vertices such that for every directed edge $u - v$, vertex u comes before v in the ordering is called a **topological sort**. It satisfies the facts:

1. The first vertex must have an in-degree of 0.
2. A topological sort is not unique.



(a) This graph can have the two (not exhaustive) topological sortings $[5, 4, 2, 3, 1, 0]$ and $[4, 5, 2, 3, 1, 0]$.



(b) This graph can have the two (not exhaustive) topological sortings $[1, 4, 2, 3, 6, 5]$ and $[1, 4, 3, 2, 6, 5]$.

Figure 23: Examples of topological sortings of different graphs.

To determine if a graph is a DAG, note the following theorem.

Theorem 7.4 (Topological Order and DAGs)

G has a topological order if and only if it is a DAG.

Proof.

To prove that a DAG has a topological order, we use induction. Pick a v such that its indegree is 0. Then, delete v, and therefore $G \setminus v$ is also a DAG with a topological order since we are only deleting edges. We keep going.

Therefore, if we can successfully topologically sort, we know it is a DAG. So we can kill two birds with one stone. Let's see how this is implemented. We can do it iteratively and recursively (the proof above should hint that this can be recursive).

Algorithm 7.4 (Iterative Topological Sort, Determine If Graph is DAG)

The general idea is that we first find the node that as 0 in-degree. From here, we can do DFS, and when we run out of neighbors to explore, then we push this into a queue. This is essentially a post-order traversal, where at the end are going to be the end nodes with no more neighbors, and the node we started from will be added last. Then we loop through and do this for all start nodes. We first need a slightly modified form of DFS.

```
Require: Nodes $V$, adjacency list $E$
visited ← set()
res ← stack()
is_acyclic ← True
function DFS($v \in V$)
    if $v \neq$ visited then
        add $v$ to visited
        $N_v$ ← neighbors of $v$
        for $n \in N_v$ do
            if $n \in$ visited then
                is_acyclic ← False
            end if
            DFS($n$)
        end for
        push $v$ onto res
    end if
end function

function TopologicalSort(V, E)
    for $v \in V$ do
        DFS($v$)
    end for
    if ! is_acyclic then
        return False
    end if
    return reversed of res
end function
```

Note that this runtime is $O(|V| + |E|)$ since we are just running DFS with a constant amount of work on top of each call.

Algorithm 7.5 (Recursive Topological Sort)

We want to see that while G is nonempty, we want to find the $v \in V$ such that it has indegree 0. Then place v next in order, and then delete v and all edges out of v. The problem is finding which vertex has indegree 0 (if we brute force it by looking through all remaining nodes and edges, you have quadratic runtime). To do this fast, the idea is

1. initially scan over all edges to store the indegrees of every node to a list `indeg`.
2. store all nodes with indegree 0 to a queue.
3. Run through the queue, and during each loop, when we remove a node, we look at all of its out-nodes s and decrement `indeg[s]`. If `indeg[s] = 0`, then add it to the queue.

```
Require: Nodes $V$, Edges $E$
$q \leftarrow$ queue()
indeg $\leftarrow$ list()
visited $\leftarrow 0$
function RECUR(x)
    initialize the indeg and $q$
    while q is nonempty do
        $v \leftarrow$ pop(q)
        visited $+= 1$
        for each $w \in E[v]$ do
            indeg[w] -= 1
            if indeg[w] $= 0$ then
                push w into q
            end if
        end for
    end while
    if visited != |V| then
        return False
    end if
    return True
end function
```

Notice that the inner for loop is $O(d(v) + 1)$, while we run over all n. So really, we are doing $O(n(d(v) + 1)) = O(m + n)$, where the plus n comes from the constant work we are doing for each node. Note that if we have a non-DAG, then at some point the queue will be empty but we haven't processed all the vertices, at which point we can declare failure.

To end this, we can make a general statement about all directed graphs.

Theorem 7.5 ()

Every directed graph is a DAG of strongly connected components (SCC).

This gives us a way to represent a directed graph with a collection of DAGs.[6] An extension of topological sort is making a *BFS tree*, which partitions a graph into layers that represent the number of steps required to go from a source vertex to a node.

Algorithm 7.6 (BFS Tree)

To construct a BFS tree, we just need to slightly modify the original BFS code.

[6]In fact, this Kosaraju's algorithm, can be done in linear time, though it is highly nontrivial.

```
Require: Nodes $V$, adjacency list $E$
visited = set()
layers = $\{v : 0 \mid v \in V\}$
function BFS(start)
    layer ← 0
    toExplore ← queue()
    add (start, layer) to toExplore
    add start to visited
    while toExplore do
        curr, layer = pop from toExplore
        layers[curr] = layer
        for $n \in$ neighbors of curr do
            if $n \notin$ visited then
                add $n$ to visited
                add $(n, layer + 1)$ to visited
            end if
        end for
    end while
end function
```

This is simply BFS with constant extra work so it is $O(n + m)$.

So, a BFS tree is really just another way to topologically sort. Note the following properties.

1. In a directed graph, no nodes can jump from layer i to layers $j > i + 1$, since if it could, then it would be in layer $i + 1$. However, nodes can jump from layer j back to any layer $i < j$, even skipping layers.
2. In a directed graph, going forward is the same as going back, so nodes can jump at most one layer forwards or backwards.

Exercise 7.3 (DPV 3.16)

Suppose a CS curriculum consists of n courses, all of them mandatory. The prerequisite graph $G = (V, E)$ has a node for each course, and an edge from course v to course w if and only if v is a prerequisite for w. Note this is a directed acyclic graph (DAG). In order for a student to take a course w with prerequisite v, they must take v in an earlier semester. Find an algorithm that works directly with this graph representation, and computes the minimum number of semesters necessary to complete the curriculum, under the assumption any number of courses can be taken in one semester. The running time of your algorithm should be $O(n + m)$, where n and m are the numbers of vertices and edges in G, respectively.

Proof.

For each vertex, we want to find the longest path leading to it: if there is a path leading to a node, then all of the courses in the path should be taken sequentially. Perform a topological sort of G's nodes and label them 1 through n. Then, we go through the nodes in the resulting topological order. For each vertex, we assign the minimum number of semesters required to take it: if there are no prerequisites, we assign 1, and if there are prerequisites, we assign 1 plus the maximum value assigned to its prerequisite nodes.
Implementation details: If the input is in adjacency list format, then we do not have access to the *incoming* edges to a node (its prerequisites). By exploring the entire graph with BFS calls, we can compute the list of incoming edges to every vertex in $O(n + m)$ time. These details are not required.

If the input is in adjacency matrix format, for each node it takes $O(n)$ time to find its incoming edges, so the total running time is $O(n^2)$.

Exercise 7.4 (DPV 3.22)

Give an efficient algorithm that takes as input a directed graph $G = (V, E)$, and determines whether or not there is a vertex $s \in V$ from which all other vertices are reachable.

Proof.

We first build the DAG representation of the SCCs of G in $O(m + n)$ time, as described in lecture. This graph is a DAG where each SCC of G is represented by a single node. We return true if this DAG has exactly one node with no incoming edges (i.e., exactly one source node), and return false otherwise.
Correctness. Let u be a vertex in an SCC that is a source node in the DAG representation. If there is a path in G from a vertex v not in the SCC of u to u, then there must be an edge (corresponding to an edge in this path) into the SCC of u in the DAG, which contradicts that the node has no incoming edges. Thus, if there are two source SCCs in the DAG, no vertex of G can reach all vertices; in particular, no vertex can reach the vertices in both of the source SCCs. Thus we correctly return false if there are multiple source SCCs.
On the other hand, if there is a single source SCC in the DAG, we claim that every vertex in the SCC can reach every other vertex in G, in which case our algorithm correctly returns true. Every other SCC in the DAG is not a source, so it has an incoming edge.[a] Consider starting at an SCC in the DAG, picking an incoming edge to the SCC, and then repeating this process from the SCC from which the edge was leaving. This process stops when we reach an SCC without incoming edges. In this case there is exactly one source SCC, so this process will arrive at the single source SCC when starting from any SCC in the DAG. This implies there is a path from the unique source SCC to every other SCC in the DAG, and thus every vertex of the source SCC can reach all vertices in all SCCs of G; that is, all vertices of G.

[a]The following argument can be made formal with induction.

Exercise 7.5 (DPV 3.19)

You are given a binary tree $T = (V, E)$ with designated root node with $n = |V|$ vertices which are the integers from 1 to n. In addition, there is an array $x[1..n]$ of values where $x[u]$ is the value of vertex $u \in V$. Define a new array $z[1..n]$ where, for each $u \in V$,

$$z[u] = \max\{x[v] \mid v \text{ is a descendant of } u\} \tag{77}$$

That is, $z[u]$ is the maximum x-value of all vertices in the subtree of T rooted at u. Note that, by definition, any node is a descendant of itself. Describe an $O(n)$-time algorithm which calculates the entire z-array.

Proof.

We propose the following recursive algorithm performs a *postorder* traversal of the tree and populates the values of the z-array in the process:

```
computeZ(u):
  maxVal = x[u]
  if u.left is not null:
    computeZ(u.left) # compute z for all descendants of u.left
```

```
      maxVal = max(z[u.left], maxVal)
    if u.right is not null:
      computeZ(u.right) # computes z for all descendants of u.right
      maxVal = max(z[u.right], maxVal)
    z[u] = maxVal
```

We initially call `computeZ` on the root node of T. The algorithm takes $O(1)$ time per node, which is $O(n)$ overall[a]

[a] This algorithm can also be described as a modified version of the so-called *depth-first search* (DFS) graph traversal algorithm, which is different from BFS.

Exercise 7.6 ()

Your data center needs to run a number of jobs (compute requests) numbered $1, 2, \ldots, n$. These are specified in a list of m tuples where (i, k) means that job i must be completed before job k can run. A given job may have multiple dependencies; for example, you might have constraints $(1, 4), (2, 4), (3, 4)$ that all of jobs $1, 2,$ and 3 must be completed before 4 can run.

1. Describe an $O(n + m)$ runtime algorithm that determines whether it is possible to execute all of the jobs, and if so, determines a valid order in which to execute the jobs one at a time. *Hint. How to relate SCCs to cycles?*
2. Suppose you have n identical servers (so that if there were no constraints you could simply run each job on a separate server). Suppose every job has the same runtime R. Describe an $O(n + m)$ runtime algorithm to compute the total runtime that will be necessary to run all of the jobs in a valid order.

Proof.

For this question we define a graph $G = (V, E)$ where there is a vertex for every job $1, \ldots, n$ and an edge from i to k for every listed dependency (where k depends on i).

1. We note that a sequence of jobs can be executed if and only if there is no circular dependency. In the language of graphs, this requires that G be free of cycles. To this end, it suffices to propose an algorithm that runs in $\mathcal{O}(m+n)$. We will use Kosaraju's SCC algorithm. We prove the following claim:

   $$\text{Any SCC with} \geqslant 2 \text{ vertices contain a cycle.}$$

   To see this, consider any distinct u, v in this SCC. Let $p_{u,v}$ and $p_{v,u}$ be the paths from u to v and backwards, respectively. Now concatenate the paths and get a walk that starts from u and ends at u. Note that each vertex appears at most once in $p_{u,v}$ and in $p_{v,u}$, so in the combined walk, it appears at most twice. Consider the set of vertices that are revisited in this walk — clearly, u is one of them and is the latest one to be revisited. There must exist a vertex w that was the *first* to be revisited. Then, the section of the walk between the two visits of w form a cycle by definition: it starts from w, ends at w, and does not repeat any other vertices. This proves the claim. And to go back to our problem, the following are equivalent:
   (a) jobs can be executed
   (b) no cycles in G
   (c) each SCC obtained from Kosaraju is a singleton
2. We first run the algorithm from part (a) to check if it is possible and to find a valid order of the jobs if so. Then define an array L of length n. We will compute $L[k]$ as the length of the longest dependency chain prior to k. Loop over the k jobs in topological order. For each, compute $L[k] = 0$ if k has no dependencies, or $L[k] = 1 + \max_{(i,k)} L[i]$ otherwise. Finally, return $\max_k L[k]$.

7.4 Bipartite Graphs

Now we shall see a further application of BFS trees.

Definition 7.8 (Bipartite Graph)

A **bipartite graph** is an undirected graph $G(V, L)$ where we can partition $V = L \sqcup R$ such that for all $e = \{u, v\} \in E$, we have $u \in L, v \in R$.

We would like to devise some method to determine if an arbitrary graph is bipartite.

Theorem 7.6 ()

G is bipartite if and only if all cycles in G are even length.

Proof.

Proving ($\implies$) is quite easy since if we suppose G has an odd length cycle, then we start packing vertices of a cycle into L, R, but by the time we came back to the start, we are forced to pack it into the wrong partition!
The converse is quite hard to prove, and we'll take it at face value.

Now in practice, how would we determine if all cycles are even length? This is where BFS shines.

Algorithm 7.7 (Determine Bipartite On All Cycles of Even Length)

The general idea is we first run BFS on the graph starting at $s \in V$, which divides it up into layers $L_1, \ldots, L_l$ representing the shortest path from s. Then for each layer $L_i \subset V$, we check if there are connections between two vertices $x, y \in L_i$. If there are connections, then this is not bipartite. If there are none, then this is bipartite since we can then color it.

```
Require: Nodes $V$, adjacency list $E$
visited = set()
layers = $\{v : 0 \mid v \in V\}$
function BFS(start)
    layer ← 0
    toExplore ← queue()
    add (start, layer) to toExplore
    add start to visited
    while toExplore do
        curr, layer = pop from toExplore
        layers[curr] = layer
        for $n \in$ neighbors of curr do
            if $n \notin$ visited then
                add $n$ to visited
                add $(n, layer + 1)$ to visited
            end if
        end for
    end while
end function

function Bipartite(V, E)
    BFS(v) for some $v \in V$
    for $(u, v) \in E$ do
        if layers[u] == layers[v] then
            return False
        end if
    end for
    return True
end function
```

Therefore, we run BFS, which is $O(n + m)$, and then to compare the edges, it is $O(m)$.

Bipartiteness is actually a special case of *coloring problems*. Given a graph with k colors, can I color it so that every neighbor has a different color than the original node? It may seem like at first glance that we can do the same method and look at the layers again, but it turns out that 3-coloring is hard. More specifically it is an NP-complete problem, which colloquially means that there isn't much of a better way than a brute-force solution. However, it turns out that according to the *4 color theorem*, any map can be colored with 4 colors.

7.5 Strongly Connected Graphs

Now how do we find out if a directed graph is strongly connected? The straightforward solution would be to take each vertex $v \in V$, run BFS to find the set of vertices reachable from v, and do this for every vertex. The total running time is $O(n(n + m))$, which is quadratic. Note that for an undirected graph this is trivial since we just run DFS/BFS once.

Theorem 7.7 ()

G is strongly connected if and only if for any $v \in V$,

1. all of V is reachable from v.
2. v is reachable from any $s \in V$

Algorithm 7.8 (Determine if Graph is Strongly Connected)

Using the theorem above, we can run BFS/DFS twice: one on the original graph and one on the reversed graph, consisting of all edges directed in the opposite direction.

```
Require: Nodes $V$, Adjacency list $E$
function StronglyConnected($s \in V$)
    visited ← set()
    BFS(s)
    if visited != $V$ then
        return False
    end if
    visited ← set()
    reverse all edges in $E$
    BFS(s)
    if visited != $V$ then
        return False
    end if
    return True
end function
```

The running time is just running BFS twice plus the time to reverse the edges, so it is $O(n + m)$.

7.6 Shortest Path

In the shortest path, you are given a *weighted* (positive integer) directed graph and your goal is to find a path from s to t with the smallest length. This is where we use Dijkstra's. What we can do to brute force is just replace a edge with length k to k edges of length 1, and we run BFS on this. However, this is not efficient since the weights can be unbounded. This is where we introduce Dijkstra. The following is how it is introduced in class.

Algorithm 7.9 (General Dijkstra)

We can keep a temporarily list π of the shortest path we have found so far, and a permanent list `dist` keeping track of all paths we know are for sure the shortest path.

```
Require: Graph $G(E, V)$
1: $S \leftarrow \{s\}$                                   ▷set of nodes that we know for sure is shortest
2: dist[s] = 0 and the rest very large numbers          ▷our final list
3: $\pi[v] = w_{sv}$ for all $v \in V$                    ▷initialize list with neighboring nodes from start $s$
4: function Dijkstra(s)
5:     $u = \text{argmin}_{v \notin S} \pi[v]$           ▷Find node having minimum accum weight so far
6:     add $u$ to $S$                                    ▷This node must be shortest so add to $S$
7:     dist$[u] = \pi[u]$                                ▷Now we can update its shortest path in dist
8:     for $v \notin S$ do                               ▷Look at all neighbors of $u$ and update those not in
9:         $\pi[v] \leftarrow \min\{\pi[v], d[u] + w_{uv}\}$   ▷$S$ since those in $S$ are all guaranteed to be shortest
10:    end for
11: end function
```

The problem is line 5 above. We don't want this linear search time since it makes the whole thing quadratic, so rather than a list, we can implement a heap, resulting in the code below.

Algorithm 7.10 (Dijkstra's Algorithm)

The general idea is to run a graph traversal like BFS but when you reach a new vertex v, you can store the accumulated time it took to get to v and store for all neighbors the accumulated time it will take to get to each of those neighbors. If it is less than what we have currently, then we have found a shorter path and we should update this.

```
Require: Nodes $V$, Edges $E$
1: function DIJKSTRA(s)
2:     dist ← list of size $|V|$ with $+\infty$       ▷Initialize list of big nums representing shortest distances
3:     dist[$s$] ← 0                                   ▷The starting node has dist 0
4:     predecessors ← $\{v : None \mid v \in V\}$      ▷predecessors of each node for path tracking
5:     toExplore ← minheap()                           ▷A priority queue of (weight, node)
6:     add $(0, s)$ to toExplore                       ▷You want to explore this first
7:     while toExplore do
8:         (curr_dist, curr_node) ← pop from toExplore ▷pop from toExplore
9:         if curr_dist > dist[curr_node] then         ▷If this distance is greater than what
10:            continue                                ▷I already have then not worth exploring
11:        end if
12:        for neighbor, weight ∈ E[curr_node] do      ▷Look at each neighbor
13:            new_dist ← curr_dist + weight           ▷The distance to getting to neighbor from now
14:            if new_dist < dist[neighbor] then       ▷If this new dist is shorter than what we have
15:                dist[neighbor] = new_dist           ▷Update best distance
16:                predecessors[neighbor] = curr_node  ▷Update its predecessor
17:                push (new_dist, neighbor) onto toExplore ▷Should prob explore from here
18:            end if
19:        end for
20:    end while
21:    return distances, predecessors
22: end function
```

You essentially push n times and pop m times, and the time per push and pop is $\log_2(n)$. Therefore, the total time to push is $n \log(n)$ and to pop is $m \log(n)$, making the total runtime $O(log(n)(n+m))$.

The first example gotten in class ignores the distances and just attempts to modify the distances in the heap itself (through the decrease key operation). This takes $2\log_2(n)$, but if we use a heap with d children, we can modify the runtime to $d \log_d(n)$. Therefore, the total runtime with tunable parameter d is

$$O\big((m + nd) \log_d(n)\big) \tag{78}$$

which can be minimized if we set $d = m/n$, so $O(m \log_{m/n} n)$, where for dense graphs m/n is large and so it can behave roughly in linear time $\Theta(m)$.

Exercise 7.7 ()

Let $G = (V, E)$ be a weighted strongly connected directed graph with positive edge weights. Let v_0 be a specific vertex. Describe an algorithm that computes the ***cost*** of the shortest walk between every pair of vertices of G, with the restriction that each of these walks must pass through v_0 (that is, for every distinct pair $u, v \in V$, among all walks from u to v that pass through v_0, compute the cost of the shortest walk). Describe the algorithm, analyze its runtime complexity, and briefly explain (not a formal proof) why it is correct. Try to give an algorithm that runs in $O(|E| \log(|V|) + |V|^2)$ time. As usual, you may use any algorithm as described in lecture without restating it or arguing for its correctness.

Proof.

The high level idea is to decompose any qualifying $u \to v$ walk into the combination of two paths $u \to v_0 \to v$, where we try to minimize the cost of both subpaths. It's easy to compute the minimum cost of $v_0 \to v$ for all v: running Dijkstra once over the graph suffices. The first half, $u \to v_0$, is the nuisance since we need to calculate this quantity for every $u \in V$. Solution? Observe that the destination node v_0 is fixed! We flip the direction, define a "reverse graph" G^{-1} where each edge carries its original weight but points in the other direction. Then, any cheapest $v_0 \to u$ path in G^{-1} would correspond to the cheapest $u \to v_0$ path in G, with matching total costs.

Exercise 7.8 ()

Let $G = (V, E)$ be a directed, weighted graph with $|V| = n$ and $|E| = O(n)$ (that is, the graph is sparse). Let s be a vertex in V. How quickly can the cost of the following shortest paths be computed under the given conditions? Just note the runtime and be prepared to explain. All of these can be solved using a single call to a shortest-path algorithm if provided the correct input graph (not necessarily the given one).

1. Compute the shortest path distance from some s to all other vertices in G under the condition that the weight of every edge is a positive integer ≤ 10.
2. Compute the shortest path distance *to* a target t from all possible source vertices s in a graph with positive edge weights.

Proof.

Listed.

1. Since all weights are integer and uniformly bounded, we convert G into an unweighted graph and apply BFS. Construct unweighted $G' = (V', E')$ as follows: for each directed edge $(u \to v \in E$, put a series of dummy nodes between u, v in G' so that the distance from u to v in G' is precisely the integer weight $w(u, v)$ of $u \to v$ in E. Now G has at most $10n$ nodes and $10n$ edges. So BFS runs in $O(|V'| + |E'|) = O(n)$.
2. Construct the reversed graph G^{-1} and run Dijkstra(G^{-1}, t). This finishes in $\mathcal{O}((m+n)\log n) = O(n \log n)$ time since G is sparse.

Exercise 7.9 ()

Let $G = (V, E)$ be an undirected, weighted graph with non-negative edge weights. Let vertices $s, t \in V$ be given. Describe an algorithm that efficiently solves the following questions.

1. Find the shortest/cheapest $s - t$ walk with an even number of edges.
2. Find the shortest/cheapest $u - v$ walk with a number of edges of form $6k + 1, k \in \mathbb{N}$.

Proof.

Listed.

1. The key observation is that as we travel on G, the number of edges we have travelled along alternates between being odd and even. Furthermore, the very same vertex may correspond to both even and odd: for example if we walked along $u \to v \to w \to u$, then initially we travelled for 0 edges, but upon return we travelled a total of 3 edges. We need a way to distinguish them. The solution? Duplicate each vertex into two categories: "odd" and "even."
   We construct a new graph $G' = (V', E')$ by duplicating every vertex $v \in V$, labeling one of them as v_{odd} and the other v_{even}. For each edge $(u, v) \in E$, add two edges $(u_{\text{odd}}, v_{\text{even}})$ and $(u_{\text{even}}, v_{\text{odd}})$ to E', both with the same as $(u, v) \in E$.
   Clearly, $|V'| = 2|V|$ and $|E'| = 2|E|$. What would edges look like in G'? By construction, the

two endpoints of an edge in G' have different subscripts, one with "odd," the other "even." This agrees with our previous observation on the original G that as we walk along the graph, the distance we have so far travelled alternates between even and odd. It follows that, starting from s_{even}, a vertex $v_{\text{even}} \in V'$ (resp. v_{odd}) is only reachable via even (resp. odd) number of edges. On the other hand, also notice that there is a natural correspondence between edges in G' and G: $(u_{\text{odd}}, v_{\text{even}}) \in E'$ corresponds to $(u, v) \in E$. This means a *path* in G' naturally corresponds to a walk in G, e.g.:

$$u_{\text{even}} \to v_{\text{odd}} \to w_{\text{even}} \to u_{\text{odd}} \to t_{\text{even}} \qquad \text{corresponds to} \qquad u \to v \to w \to u \to t.$$

Combining both observations above, there exists an $s - t$ walk in G with an even number of edges if and only if there is a path in G' from s_{even} to t_{even}. The rest is simple: run a pathfinding algorithm on G'. The weights are non-negative, so we use Dijkstra's algorithm.
Total runtime? Time to construct G' involves $|V'| = 2|V|$ vertices and $|E'| = 2|E|$ edges. This is dominated by running Dijkstra on G', which takes $O((|V'|+|E'|)\log|V'|) = O((|V|+|E|)\log|V|)$ time. Finally, transforming the path in G' back to a walk in G takes linear time w.r.t. the path length (one step for each edge), which is bounded by $O(|E'|)$. So overall most work is dominated by Dijkstra's algorithm and the overall algorithm runs in $O((|V| + |E|)\log|V|)$.

2. Same idea but make 6 copies of the graph.

Exercise 7.10 ()

Suppose that in addition to having edge costs $\{l_e : e \in E\}$, a graph also has vertex costs $\{c_v : v \in V\}$. Now define the cost of a path to be the sum of its edge lengths, *plus* the costs of all vertices on the path. Give an efficient algorithm for finding the minimum cost path from s to t. You may assume edge costs and vertex costs are all nonnegative.

Proof.

Using the generic approach, we can use $\text{cost}_u(v) = \text{cost}(u) + w(u, v) + c_v$ to solve this problem. Alternatively, for each edge (u, v) we can update its weight to $w(u, v) + c_v$ and run Dijkstra on this updated graph, which gives an equivalent mathematical formulation.

7.7 Negative Weighted Graphs

Now let's extend this problem to find the shortest path in negative weighted graphs. Before we think of a solution, we must make sure that there is no cycle that has a negative accumulated path. Otherwise, this problem becomes ill-defined, so we first assume that such a shortest path exists.

At first glance, we may just think of adding $\min(v)$, the minimum value to every node so that this now just becomes a regular positive graph and run BFS on it. However, this does not work since we are not adding a constant number over all paths (it is proportional to the number of nodes in the path).

Another way we can think of is just run Dijkstra. However, if it is looking at two paths. We can have $s \xrightarrow{2} b$ and $s \xrightarrow{5} a \xrightarrow{-4} b$. Dijkstra will immediately go to b thinking that it is the shortest path, since that's how far it see. So we need to look far into the future. Therefore, after an arbitrarily long path length, you could get a negative length that just kills your accumulator.

We use the Bellman equations, which could be solved using dynamic programming like we've seen before.

Theorem 7.8 (Bellman Equations)

We write the **Bellman equations**.

$$d[v] = \min_w \{d[w] + l_{wv}\} \tag{79}$$

with $d[s] = 0$ for the starting vertex. The solution has a unique solution that finds the shortest path from s to any $v \in V$.

Proof.

Note that $d[w] + l_{wv}$ is the length from some path from $s \mapsto v$ that goes through w. The minimum of it must be the shortest path over all $w \in V$. Suppose the shortest path goes through fixed x. If there exists a shorter path from $s \mapsto x$, then replace $d[x]$ by this shortest path. Therefore,

$$d[v] = d[x] + l_{xv} \leq d[w] + l_{wv} \implies d[v] = \min_w \{d[w] + l_{wv}\} \tag{80}$$

To prove uniqueness, suppose there are some other solutions π where $\pi[v] \neq d[v]$ for some v. But this cannot be the case by definition since $d[v] <= \pi[v]$ for all v.

Theorem 7.9 ()

Given the shortest paths, we can lay out this graph like a tree where $l_{ab} = l_{aa_1} + l_{a_1 a_2} + \ldots + l_{a_l b}$.

So how do we actually implement this?

Algorithm 7.11 (Shortest Path in Possibly Negative Weighted Graph)

```
Require: Nodes $V$, Edges $E$
1: function ShortPath(V, E)
2:     res ← list(0) of large numbers of size $|V|$.
3:     res[$s$] = 0
4:     predecessors ← $\{v : None | v \in V\}$
5:     while $\exists (u, v)$ s.t. res[v] > res[u] + $l_{uv}$ do
6:         res[$v$] ← res[u] + $l_{uv}$
7:         predecessor[v] ← u
8:     end while
9: end function
```

This is guaranteed to converge and stop after a finite number of steps since at every iteration, a path will either

1. get updated from infinity to a path length
2. get reduced from a path length to a shorter path length

And we will have to reach the shortest path length at which point we can't reduce it further.[a] Computing the runtime is a bit tricky, since we can look at the same edge twice since minimum paths may have been updated in the middle. Therefore this list `res` may reduce very slowly. For example, let the length of each edge $|l_e| \leq L$. Then in the worst case, res[s] can be initialized to $(n-1)L$ representing the max path across all nodes, and we can decrease by 1 in each step. So over all nodes, we can decrease so that each res[s] becomes $-(n-1)L$, meaning that we are doing on the order of $2n^2 L$ iterations. This is too slow, especially for non-distributed settings.

[a] This algorithm is also called *policy iteration* in reinforcement learning and is analogous to gradient descent.

A better way is to not be so random about how we choose the (u, v) in the while loop. Notice how we can lay out the shortest paths like a tree, so we can work in layers. The next algorithm implements this.

Algorithm 7.12 (Bellman-Ford Algorithm)

We think of going in rounds indexed by t, and at every round, we are iterating through all the nodes and updating the shortest path of v using the shortest path of w included in all in-neighbors of v. At most, we will need to update this at most n times, which will guarantee convergence.

```
Require: Nodes $V$, Edges $E$
1: $\pi \leftarrow$ list(0) of large numbers of size $|V|$.
2: $\pi[s] = 0$
3: function BellmanFord(x)
4:     for $t = 1, \ldots, n-1$ do
5:         for $v \in V$ do
6:             $\pi^{(t)}[v] \leftarrow \min_w \{\pi^{(k-1)}[v], \pi^{(k-1)}[w] + l_{wv}\}$
7:         end for
8:     end for
9: end function
```

The runtime is easier to see.

1. The step in the inner loop looks over the set of nodes of size indeg(v).
2. Looping over all the nodes in the inner for loop means that we are going over all edges, so $O(m)$.
3. The outer for loop goes through $n-1$ times, so the total runtime is $O(nm)$.

At first glance, this problem seems like it isn't too different from Dijkstra, but there is a 50-year conjecture that this cannot be improved to linear time.

Exercise 7.11 ()

Let $G = (V, E)$ be a directed graph with real-valued edge weights, where each vertex is colored in either red or green. Find the shortest/cheapest $s - t$ walk such that, not counting s, the walk visits red vertices for an even number of times and green vertices at least thrice. (Duplicates allowed and will be counted more than once.)

Proof.

Similar to the last problem in the previous recitation, the key insight lies in constructing a directed graph $G' = (V', E')$ that captures some additional structures. Based on the constraints, as we walk along a path in G, there are two things we need to take care of:

- The number of (not necessarily distinct) red vertices we have walked past, and whether this number even or odd (this is called the *parity* of that number), and
- The number of (not necessarily distinct) distinct green vertices we have walked past.

To encode all of the information above, each vertex in G' will be represented by a "state", or a tuple (v, p, g) where

- $v \in V$ corresponds to an original vertex in G,
- $p \in \{0, 1\}$ (or "even", "odd") represents the parity of the count of red vertices (not necessarily distinct) visited so far, and
- $g \in \{0, 1, 2, 3+\}$ represents the number of times green vertices (not necessarily distinct) have been visited.

Now we will need to consider the conditions under which each of the tuple variable updates. For example, every time we visit a red vertex, the value p should alternate, and every time we visit a green vertex, the value g should increase until it becomes 3+. Formally, the state transitions (i.e. edges in E') can be formulated as follows. For each edge (u, v) in the original graph G, depending on the colors of u and v, we add the following edges, all with the same weight as (u, v), to E':

(v red) For every state (u, p, g) [a total of 8 such thates because $p \in \{0, 1\}$ and $g \in \{0, 1, 2, 3+\}$], add an edge to the corresponding state $(v, 1 - p, g)$. In other words, we flip the parity because we visited one more red vertex, but this does not affect the value of g.

(v green)

- For every state (u, p, g) with $g \in \{0, 1, 2\}$, add a (directed) edge to $(v, p, g+1)$ because our green counter increases given v is green. (Define $2 + 1$ to be "3+.")
- For states of form $(u, p, 3+)$, add a (directed) edge to $(v, p, 3+)$ because we still fall under the "$g \geqslant 3$" category after visiting an additional green vertex.

All of our observations on the Recitation #2 graph modeling problem still hold: if we have a path in G', we can uniquely recover a well-defined walk in G. Initially, we want to start from state $(s, 0, 0)$ because we start from vertex $s \in V$ and, per the problem, the starting point does not contribute to the red and green count. Our goal is to reach the state $(t, 0, 3+)$, which means (i) we arrive at t, and along the course we have (ii) visited an even number of red vertices and (iii) green vertices $\geqslant 3$ times. This is exactly what we want.

How about the runtime? The construction of G' involves defining $8|V|$ vertices since p has 2 possible values and g has 4, and we need to construct one state for each pair of p and g. Similarly, for each (u, v), regardless of the color of v, in both cases we add a total of 8 edges. Therefore $|E'| = 8|E|$. Since G, G' are directed graphs with real-valued weights, we need to run Bellman-Ford, which takes $\mathcal{O}(|V||E|)$. Like shown before, other costs (e.g. the one to recover a walk in G from a path in G') are linear and hence dominated by the pathfinding runtime. So the final complexity is $\mathcal{O}(|V||E|)$.

7.8 All Pairs Shortest Paths

Now what if we want to find the minimum distance between all $u, v \in V$? We can just use $|V|$ Dijkstras or Bellman-Fords to get the appropriate runtimes of $O(EV + V^2 \log V)$ or $O(EV^2)$, respectively, but for negative weighted graphs, there is a way to do this in $O(V^3)$.[7]

7.9 Minimum Spanning Trees

Definition 7.9 (Spanning Tree)

Given an undirected graph $G(V, E)$, a **spanning tree** is a subgraph $T(V, E' \subset E)$ that is

1. a tree, and
2. spans the graph, i.e. is connected

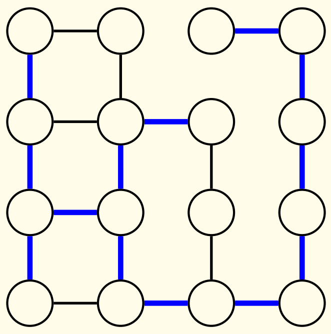


Figure 24: A spanning tree on a 4×4 grid graph.

[7]Note that if the graph is sparse, then $|E| < |V|$ and just running $|V|$ Bellman Fords may be optimal.

Note that an unconnected graph will never have a spanning tree, but what about a connected graph?

Theorem 7.10 (Spanning Trees of Connected Graphs)

A connected graph will always have at least one spanning tree, not necessarily unique.

Given a connected undirected weighted graph, we may want to find the **minimum spanning tree (MST)**, i.e. the spanning tree with edges E' such that the sum of the weights of all $e \in E'$ is minimized.[8] How do we do this? There are two well-known algorithms to solve this. Prim's and Kruskal's algorithm.

7.9.1 Prim's Algorithm with Cuts

Let's try to apply what we already know: Dijkstra. If we run Dijkstra on the graph starting at $s \in V$, we can get the shortest path from s to every other node in the graph. This will give us a tree, but it may not be minimum.

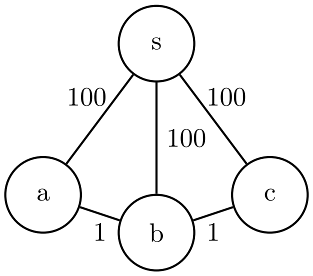


Figure 25: If we run Dijkstra on s, then our output will be a tree of cost 300, even when the actual MST can be of cost 102 starting from a.

It may seem like this is just a problem of where we start, but even this is not the case.

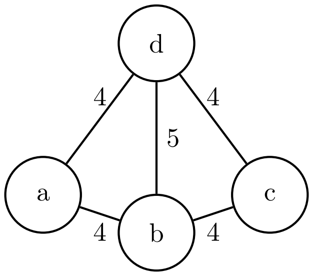


Figure 26: No matter where we start from, we will never output the MST. The MST has cost 12. If we start from b or d, we will get a tree of cost 13. If we start from a or c, we will get a tree of cost 16.

Definition 7.10 (Cuts)

Given graph $G(V, E)$, a **cut** is a partitioning of V into $(S, V \setminus S)$. Furthermore, let $\text{Cut}(S)$ be the number of edges with exactly one endpoint in S and the other in $V \setminus S$.

[8] An application of this is when we generally want to make sparse graphs. In a datacenter, wires can be expensive, so how I can minimize the length of wires to buy to construct a spanning subgraph?

Theorem 7.11 (Cycles and Cuts)

Given cycle $C \subset E$ in a graph and a cut $S \subset V$,

$$|C \cap \text{Cut}(S)| \tag{81}$$

is even. We can intuit this by visualizing the cycle as a long piece of looped string and a cut is a circle. The intersection between this circle and the string must be even since every time the cycle crosses through the cut, it must return back across the cut to the initial point.

Now time for a bizarre theorem.

Theorem 7.12 (Cut Property of MSTs)

For all cuts $S \subset V$ of an undirected graph, the minimum cost edge in $\text{Cut}(S)$ belongs to the MST. Furthermore, the converse is true: if we take all cuts and find all their minimum cost edges, these edges is precisely the MST! Therefore, an edge $e \in \text{MST}$ iff e is a min-cost edge for some cut.

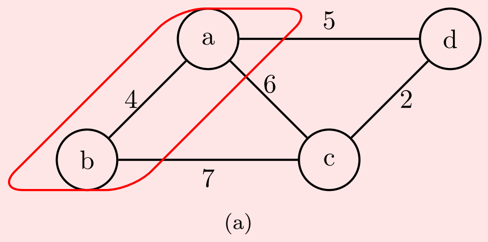


Figure 27: In the left cut, the edges are $(a, d), (b, c), (a, c)$. The minimum weight is 5 on the (a, d) edge, so it must be in the MST. For the right cut, (c, d) must be in the MST.

The final part is that if we have all edge costs different, then we will have a *unique* MST.

Proof.

We use a greedy approach and prove by contradiction. Suppose that this is not true, i.e. there exists a cut S with minimum cost edge e, and $e \notin \text{MST}$. Then, there exists some other edge $e' \in \text{Cut}(S)$ that is in the MST, since the MST is spanning and it must cross over to connect the whole graph. Well if we just put e in and take e' out, we will still have a spanning tree since it connects the left spanning tree to the right spanning tree, and we now have a cheaper tree. So the original cannot be the MST in the first place.
To prove the converse, consider some edge e in the MST and we must prove that it is the minimum cost edge in some cut. Note that if we take e out, then it divides the MST into two connected components, and we can just define the cut as these subsets of nodes. So this is in $\text{Cut}(S)$ for some $S \subset V$. We can also prove that this is minimal since if it wasn't the minimum cost edge for some cut, we could have taken it out and inserted a cheaper edge e' to begin with, getting a cheaper spanning tree.

We can just brute force this logic into an algorithm by going through all possible cuts and adding the minimum cost edge to our MST set. It is clear that a cut is defined by a subset of S, so really the number of cuts a graph can have is $2^{|S|-1}$, which is exponential in n. However, the minimum spanning tree isn't exponential since it must have $n-1$ edges, so there must be many cuts with the same minimum edge.

One way is to start with one vertex a that contains the minimum cost edge (a, b) across all edges. This edge must be minimal and must be in the MST. Then we can look at the cut $S = \{a, b\}$ and look at that cut. We keep doing this, keeping track of the set of edges we need to look at after adding a new node to our cut.

So the number of cuts I consider is equal to the number of edges in the spanning tree.

Algorithm 7.13 (Prim's Algorithm to Find MST)

It turns out that we can modify Dijkstra to solve it.

```
Require: Graph $G(E, V)$
 1: function PRIM(s)
 2:     $S \leftarrow \{s\}$                                   ▷Our initial cut
 3:     $\pi[v] \leftarrow$ list of size $|V|$ of $+\infty$      ▷$\pi[u]$ is min cost of getting from $u$ into the $S$
 4:     $\pi[v] = w_{sv}$ for all $v \in V$                  ▷initialize list with neighboring nodes from start $s$
 5:     while $S \neq V$ do
 6:         $u = \text{argmin}_{v \notin S} \pi[v]$            ▷Find node having minimum cost to reach from $S$
 7:         $S \leftarrow S \cup \{u\}$                        ▷Adding this node to $S$ to expand our cut
 8:         for $u \notin S$ do                                ▷Since we expanded $S$, our min reach distances
 9:             $\pi[u] \leftarrow \min\{\pi[u], w_{wu}\}$       ▷must be updated. It can only get shorter
10:         end for                                            ▷through a path from new $u$, so compare them
11:     end while
12: end function
```

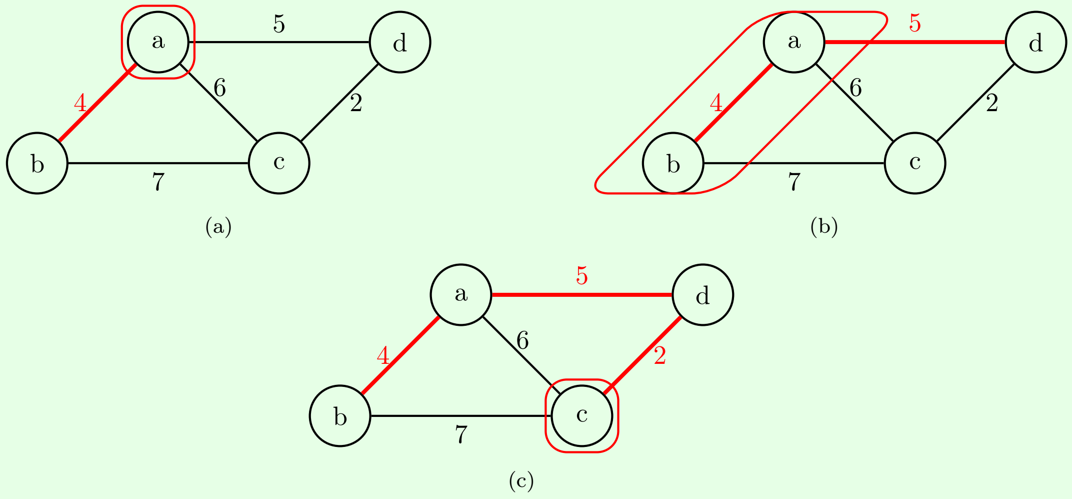


Figure 28: Step by step process of the method we mention above.

We are really going through two loops. We add to the cut S n times and for each time we add, we must compute the argmin, which is also $O(n)$, so our total time complexity of $O(n^2)$.

However, this is not efficient, so we can introduce a minheap to get the argmin step faster.

Algorithm 7.14 (Prim's Algorithm with MinHeap)

```
Require: Nodes $V$, Edges $E$
 1: function Prim(V, E)
 2:     mst ← []                                            ▷Initialize mst array to return
 3:     s ← 0                                               ▷Choose any starting node
 4:     visited ← set()                                     ▷Our expanding set of cuts.
 5:     edges ← minheap()                   ▷The set of low-weight edges that we can explore from
 6:     add (weight, s, next_node) for edges in E    ▷Look for edges from s to expand our cut from.
 7:     while |visited| < |V| do                            ▷Until we have visited all cuts,
 8:         weight, frm, to ← pop from edges                ▷Get the cheapest edge to explore
 9:         if to ∉ visited then                            ▷If this isn't already in our cut,
10:             add to → visited                            ▷Add it to our cut. From cut
11:             add (frm, to, weight) to mst                ▷property, this must be added to mst
12:             for next_to, next_weight ∈ E[to] do         ▷After expanding, add newly discovered
13:                 if next_to ∉ visited then               ▷edges for future exploration
14:                     push (next_weight, to, next_to) to edges
15:                 end if
16:             end for
17:         end if
18:     end while
19:     return mst                                          ▷of form (from, to, weight)
20: end function
```

You essentially push n times and pop m times, and the time per push and pop is $\log_2(n)$. Therefore, the total time to push is $n\log(n)$ and to pop is $m\log(n)$, making the total runtime $O((n+m)log(n)) = O(m\log n)$.

This can be sped up even faster if we use Fibonacci heaps or assume extra structure on the graph.

7.9.2 Kruskal's Algorithm

If we were to try and construct this algorithm from scratch, we may take a greedy approach by incrementally adding the minimum cost edge from your cut. However, there is one thing to check: have we entered a cycle? Checking whether the next added node a completes a cycle in $S \cup \{a\}$ is nontrivial.

Theorem 7.13 (Cycle Property)

For all cycles C, the max cost edge of C does not belong to MST.

Therefore, you can take S and either add to it using the cut property or delete candidates from it using the cycle property. What is the best order to do this in? Kruskal's algorithm answers this question, which takes a greedy approach.

Algorithm 7.15 (General Kruskal's Algorithm)

The general idea is that we sort $e \in E$ in increasing cost, and for each $e \in E$, we use either the cut or cycle property to decide whether e goes in or out.

```
Require: Graph $G(V, E)$
function Kruskal(V, E)
    sort $E$ in increasing cost
    $T \leftarrow \{\}$
    for $e \in E$ do
        if $T \cup \{e\}$ does not have cycle then
            $T \leftarrow T \cup \{e\}$                              ▷Cut property
        else                                                        ▷Cycle property
            continue          ▷discard $e$ since from sorting, this edge is heaviest in cycle
        end if
    end for
end function
```

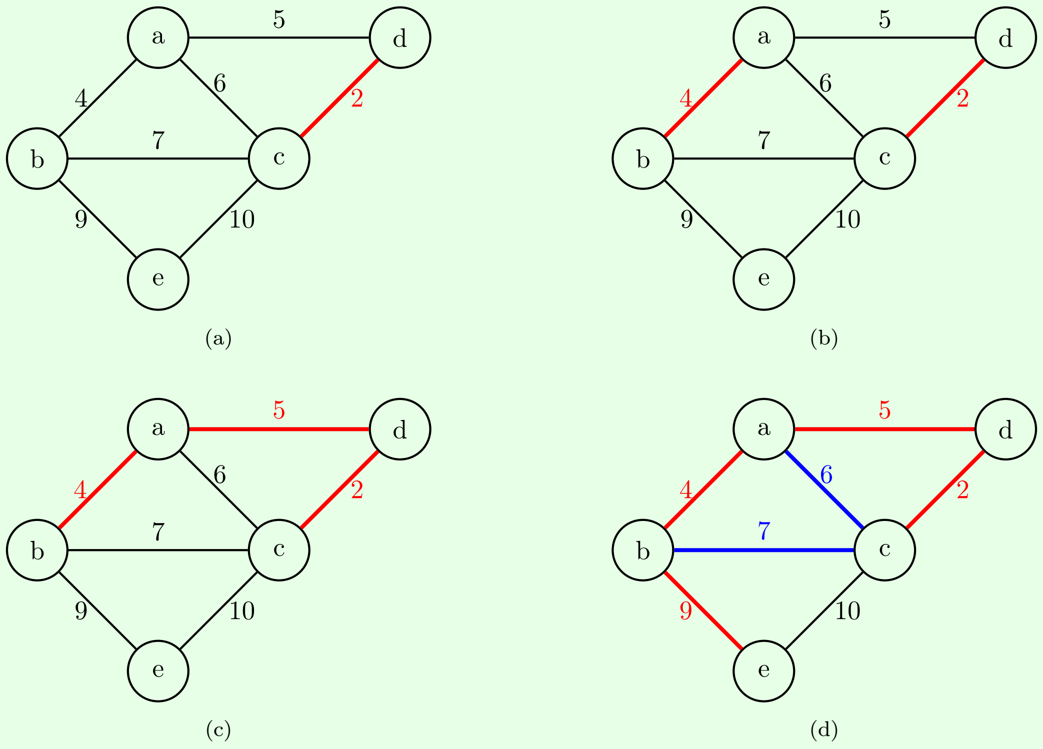


Figure 29: Kruskal's algorithm. In the last step, we see that the next minimum cost edge of 6 and 7 forms a cycle, so we add the edge of length 9.

The sorting of edges take $O(m \log m)$ time, and after sorting, we iterate through all the edges and apply m find-union algorithm, which each take at most $O(\log n)$ time. Therefore, the overall complexity is $O(m \log m + m \log n)$. However, the value of m can be at most $O(n^2)$, so the two logarithms are essentially the same, arriving at the final runtime of $O(m \log n)$.

The way to prove that this is correct is to show that every step you do is correct, known as *structural induction*, either because of one of the two properties. Say that so far, we have some edges in V which forms a partition of $V = \sqcup_i T_i$ of disjoint trees (can be trees, one edge, or just single nodes). We are looking at the next biggest edge $e = (a, b)$. There are two possibilities.

1. If a, b are both in a single T_i, then this forms a cycle and can be thrown away since this is the max cost edge in the cycle by the cycle property.

2. If a, b connect T_i and T_j for $i \neq j$, then this edge is in $\text{Cut}(T_i)$ and is the minimum cost edge since the rest of the edges in $\text{Cut}(T_i)$ come next in the sorted E. Therefore this must be included by the cut property.

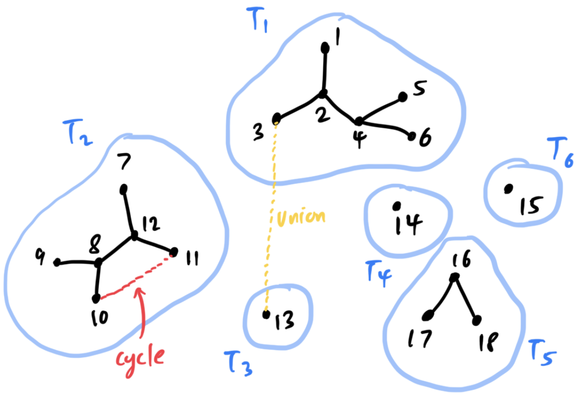


Figure 30: Each T_i is a component formed by the edges chosen so far. For example, $T_1 = \{1, 2, 3, 4, 5, 6\}, \ldots$. We can either discard an edge (red) or include an edge (yellow).

The only bottleneck in here is line 5, where we check if e does not complete a cycle in T. It will obviously not be efficient to do BFS, construct a tree, and see if there is a loop by checking if two points in the same layer are connected. It may help to decompose this algorithm into two steps: what is being stored and what is being checked?

Note that from our visual, we are really just keeping a set of these points that each make a subtree and connecting them together. How do we efficiently search for which cluster a point is a part of and efficiently merge two clusters? We could use a hashmap but this wouldn't work. We need something like a doubly linked list.

Algorithm 7.16 (Kruskal's Algorithm)

The implementation uses the Union-Find data structure. For clarity, we will not elaborate it but will show the full pseudocode.

```
Require: Nodes V. Edges $E = \{(u, v, w)\}$ where $u, v \in V$ and $w > 0$ is a weight.

  function KRUSKAL(V, E)
      $n \leftarrow |V|$
      sort $E$ in increasing order of weights.                    ▷Needed for Kruskal
      parent ← $[0, \ldots, n-1]$              ▷Initialize the disjoint cluster each node is in
      rank ← list of 0s of size $n$
      function FIND(x)
          if parent[x] $\neq$ x then
              parent[x] ← Find(parent[x])                          ▷Path compression
          end if
          return parent[x]
      end function
      function UNION(x, y)
          px, py = find(x), find(y)
          if px = py then
              return False
          end if
          if rank[px] < rank[py] then
              parent[px] ← py
          else if rank[px] > rank[py] then
              parent[py] ← px
          else
              parent[py] ← px
              rank[px] ← rank[px] + 1
          end if
          return True
      end function
      mst ← []
      for u, v, weight ∈ edges do
          if union(u, v) then
              add (u, v, weight) to mst
          end if
          if len(mst) = n - 1 then
              break
          end if
      end for
      return mst
  end function
```

To analyze the runtime of this, we define the function.

Definition 7.11 (Ackerman Function)

The **Ackerman function** is one of the fastest growing functions known. It is practically infinity.

$$A(m, n) = \begin{cases} n+1 & \text{if } m = 0 \\ A(m-1, 1) & \text{if } m > 0 \text{ and } n = 0 \\ A(m-1, A(m, n-1)) & \text{if } m > 0 \text{ and } n > 0 \end{cases} \tag{82}$$

The inverse Ackerman function therefore grows extremely slowly.

If we optimize the steps in Kruskal's algorithm, we can get its runtime to

$$O((m+n)\log^*(n)) \tag{83}$$

which is practically linear.

7.9.3 Applications

Here is a way to cluster data, a surprising way to apply MSTs. It is the most widely used application, especially in data science. The problem is that given n data points $x_i \in \mathbb{R}^d$ and an integer k, we want to partition the points into k groups $\mathbf{C} = (C_1, \ldots, C_k)$ where $\mathbf{x} = \sqcup_i C_i$. You want to distances between the points within a group to be small and the distances between groups to be large. We can think of finding the objective which takes every pair of clusters and computes the minimum distance between these clusters, and we want to maximize this distance over all pairs of clusters.

Algorithm 7.17 (Single Linkage/Hierarchical Clustering)

The general idea is to take this dense graph, find the MST, and cut off the largest edges from this MST, which will give you k components. This is the answer. Or really, you can use Kruskal's algorithm and terminate earlier when T has k sets/components. Note that as we add edges as we construct our MST, we are merging two clusters into one. So that all you are doing is finding the next pair of closest points and merging the clusters that they are a part of.

```
Require: Nodes $V = \{v_i\} \subset \mathbb{R}^n$
1: function CLUSTER(V)
2:     Run Kruskal and at each iteration, check if you have $K$ clusters.
3:     If so, terminate and return the parent list.
4: end function
```

Theorem 7.14 ()

The algorithm above minimizes the objective function.

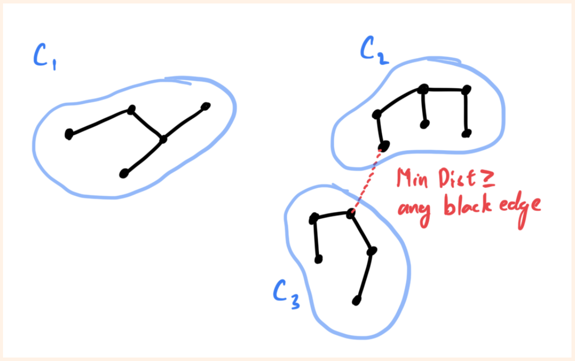
$$\operatorname*{argmax}_{\mathbf{C}} \min_{p \in C_i, q \in C_j} \{d(p,q)\} \tag{84}$$

Proof.

Let $\mathbf{C}^*$ be the MST clustering. We claim that for any other clustering $\mathbf{C} = \{C_1, \ldots, C_k\}$,

$$\text{mindist}(\mathbf{C}) \leq \text{mindist}(\mathbf{C}^*) \tag{85}$$

Assume that this was not the case, so mindist$(\mathbf{C}) >$ mindist$(\mathbf{C}^*)$ and therefore there exists a $p, q \in C_i, C_j$ such that $d(p,q) = \text{mindist}(\mathbf{C}) > \text{mindist}(\mathbf{C}^*)$. Since this is a different clustering, p, q must have been in the same cluster C_i^*. But note that since Kruskal adds edges in increasing length, all edges within a cluster must have length less edges that go across two clusters.



So $d(p, q)$ must be the less than the length of all edges within a cluster in $\mathbf{C}^*$. But all within-cluster edges must be smaller than mindist($\mathbf{C}^*$), meaning that mindist($\mathbf{C}^*$) $> d(p, q)$, contradicting the fact that is is greater, and we are done.

If you define the distance between two clusters to be the distance between the centroids (mean point), then this is called *average linkage* (min avg distance). If we define the cluster distance as the maximum distance between two points, then it is called *complete linkage* (min max distance). Kruskal's algorithm only worked for the single linkage case but may not work for these additional definitions. This is why there is usually a whole suite of clustering algorithms for a particular problem and we just find out which one fits the data the best. Furthermore, we have done *bottom-up clustering*, where we took individual points to make clusters. In *top-down clustering*, we take the whole set and cut it up into clusters.

8 Dynamic Programming

Let's take a look at a motivating example.

Example 8.1 (Computing Fibonacci Numbers)

To compute the Nth Fibonacci number, we can use a recursive method.

Algorithm 10

```
Require: $N$

  function RecFib($N$)
      if $N = 0$ then
          return 0
      else if $N = 1$ then
          return 1
      else
          return RecFib($N - 1$) + RecFib($N - 2$)
      end if
  end function
```

This is exponential, in fact $O(\varphi^N)$ where φ is the golden ratio. The reason is that we are repeatedly computing the same subproblem (e.g. RecFib($N - 2$ is computed twice)), leading to inefficiency. It would be great if we could store these intermediate values rather than recomputing them. This introduces us to the concept of memoization.

Definition 8.1 (Memoization)

Memoization refers to storing intermediate values for reuse rather than computing them again (e.g. in future recursive calls).
Therefore, the term **dynamic programming** just means *to deliberately evaluate and memoize in the correct order.*

Algorithm 8.1 (Redursive Memoized Fibonacci)

This leads us to a memoized version of computing Fibonacci Numbers, which is linear runtime. In fact, we can have constant space complexity since we don't need more than the last 2 previous Fibonacci numbers to compute the next one.

Algorithm 11 Memoized Fibonacci

```
Require: $N$
  Initialize memo array $F[0..N]$ with -1
  $F[0] \leftarrow 0$
  $F[1] \leftarrow 1$
  function MemFib($N$)
    if $N < 0$ then
      return 0
    else if $F[N] \neq -1$ then
      return $F[N]$
    else
      $F[N] \leftarrow$ MemFib($N-1$) + MemFib($N-2$)
      return $F[N]$
    end if
  end function
```

The runtime is computed, by taking the number of distinct problems (i.e. the number of calls to MemFib with distinct inputs) multiplied by the time per subproblem (constant since lookup is constant and adding is constant). Note that this assumes that arbitrary arithmetic operations take constant time, but this is not true if we look at the bit complexity, which can scale quite fast as these numbers grow.

Note that this does not really explicitly show the order in which the memoized list is being filled. It is implicit but hard to see in the recursive calls. Therefore, it may help to write it iteratively.

Algorithm 8.2 (Iterative Memoized Fibonacci)

In here, we can explicitly see that the nth Fibonacci number is explicitly dependent on the $n-2$ and $n-1$.

Algorithm 12 Iterative Fibonacci

```
Require: $N$
  Initialize array $F[0..N]$
  $F[0] \leftarrow 0$
  $F[1] \leftarrow 1$
  for $i \leftarrow 2$ to $N$ do
    $F[i] \leftarrow F[i-1] + F[i-2]$
  end for
  return $F[N]$
```

8.1 Longest Increasing Subsequence

Definition 8.2 (Longest Increasing Subsequence)

Given a sequence of numbers $A = \{a_1, a_2, ..., a_n\}$, a **longest increasing subsequence** is a subsequence $\{a_{i_1}, a_{i_2}, ..., a_{i_k}\}$ of A such that:

1. $1 \leq i_1 < i_2 < ... < i_k \leq n$ (maintains original order)
2. $a_{i_1} < a_{i_2} < ... < a_{i_k}$ (strictly increasing)
3. k is maximized (longest such subsequence)

Example 8.2 ()

For the sequence $A = \{3, 10, 2, 1, 20, 4, 25\}$:

- $\{3, 10, 20, 25\}$ is an increasing subsequence of length 4
- $\{2, 4, 25\}$ is an increasing subsequence of length 3
- $\{3, 10, 20, 25\}$ is a longest increasing subsequence as no increasing subsequence of length 5 or greater exists

For the actual problem of calculating the length of the LIS, dynamic programming gives us an efficient approach. First, let's do a brute force algorithm.

Algorithm 8.3 (Recursive Brute Force LIS)

At each step, we consider whether to include the current element in our subsequence. We can only include it if it's larger than the previous element we chose, maintaining the increasing property. We explore both possibilities (including and excluding) recursively to find the longest possible subsequence.

Algorithm 13 Recursive Brute Force Longest Increasing Subsequence

```
Require: Array $A[1..n]$
  function LIS($A, i, prev$)                ▷$i$ is current position, $prev$ is last element we took
    if $i = n + 1$ then                     ▷If we've processed all elements
      return 0                              ▷Return 0 as we can't add more elements
    end if
    // First choice: skip current element
    skip ← LIS($A, i + 1, prev$)            ▷Keep same prev, move to next element
    // Second choice: try to take current element
    take ← 0                                ▷Initialize take option to 0 in case we can't take it
    if $A[i] > prev$ then                   ▷Only if current element maintains increasing sequence
      take ← 1+ LIS($A, i + 1, A[i]$)       ▷Add 1 for taking current element
    end if                                  ▷Recursively find best sequence starting at i+1 with A[i] as previous
    // Return best option between taking and skipping
    return max(skip, take)                  ▷Choose the better of our two options
  end function
  // Initial call with sentinel value to allow taking any first element
  return LIS($A, 1, -\infty$)               ▷Start at first element, no previous restrictions
```

The runtime can be used by the recurrence relation. At every call, we may at most have to compute 2 calls on the subarray not including the current element, and we compute the max of them which takes $O(1)$, so

$$T(N) = 2T(N-1) + O(1) \implies T(N) = O(2^N) \tag{86}$$

which is not good.

Now let's move to our DP solution.

Algorithm 8.4 (Dynamic Programming LIS)

The key insight is that LIS[i] (the length of the LIS within the input array ending at i, inclusive) depends on all previous LIS[j] where j < i and A[j] < A[i]. For each position i, we can extend any previous increasing subsequence that ends with a smaller value. In other words, we are solving

$$\text{LIS}[i] = 1 + \max\{\text{LIS}[j] \mid j < i \text{ and } A[i] > A[j]\} \tag{87}$$

Algorithm 14 Dynamic Programming Longest Increasing Subsequence

```
Require: Array $A[1..n]$
  function DPLIS($A$)
      // Initialize LIS array - each single element is an increasing sequence
      Initialize array $LIS[1..n]$ with 1        ▷Base case: each element forms LIS of length 1
      for $i \leftarrow 2$ to $n$ do        ▷Consider each element as end of sequence
          for $j \leftarrow 1$ to $i - 1$ do        ▷Look at all previous elements
              if $A[i] > A[j]$ then        ▷Can we extend sequence ending at j?
                  $LIS[i] \leftarrow \max(LIS[i], LIS[j] + 1)$        ▷Take best possible extension
              end if
          end for
      end for
      return LIS[-1]        ▷Find the maximum value in LIS array
  end function
```

The runtime is $O(n^2)$ since we have two nested loops, and the space complexity is $O(n)$ since we store one value for each position in the array. Note that we could also have filled this DP array backwards by considering all arrays that start at $A[i]$.

Example 8.3 (DPLIS for Small Array)

For array $A = [3, 1, 4, 1, 5]$:
- Initially: $LIS = [1, 1, 1, 1, 1]$
- After processing $i = 2$: $LIS = [1, 1, 1, 1, 1]$
- After $i = 3$: $LIS = [1, 1, 2, 1, 1]$ (4 can extend sequence from 3)
- After $i = 4$: $LIS = [1, 1, 2, 2, 1]$
- After $i = 5$: $LIS = [1, 1, 2, 2, 3]$ (5 can extend sequence from 4)

Final answer is 3, corresponding to subsequence $[3, 4, 5]$

8.2 0/1 Knapsack

Another application of DP is in the following problem.

Definition 8.3 (0/1 Knapsack)

Given a knapsack with maximum weight capacity W, along with a set of n items, each with:
- Weight/Cost c_i
- Value v_i

You want to know the size of the subset of items that maximizes total value while keeping total weight/Cost $\leq W$. The constraint is that each item can be picked at most once (hence 0/1).

Example 8.4 ()

For $n = 3$ items and capacity $W = 4$:
- Item 1: $(c_1 = 2, v_1 = 3)$
- Item 2: $(c_2 = 1, v_2 = 2)$
- Item 3: $(c_3 = 3, v_3 = 4)$

Optimal solution: Take items 1 and 2
- Total weight: $2 + 1 = 3 \leq 4$
- Total value: $3 + 2 = 5$ (maximum possible)

The most natural way to approach this would be greedy, but this does not exactly work.

Example 8.5 (Counter-Example for Greedy Knapsack)

Let's consider a knapsack with capacity $W = 10$ and:

- Values $V = [100, 48, 60, 11]$
- Weights $C = [10, 6, 4, 1]$

Then the value/weight ratios would be

$$V/C = [10, 8, 15, 11] \tag{88}$$

and so we would choose to gain 60 for cost 4, then gain 11 for cost of 1. We do not have enough to buy any more and have a cost of 71, when we could have gotten a cost of 108 by buying 60 and 48 for a total of 10.

Just like as always, we just solve this using recursive brute force and then apply optimization with DP.

Algorithm 8.5 (Recursive Brute Force Knapsack)

The key idea is that for each item, we have two choices: either include it (if we have enough capacity) or exclude it. We try both possibilities recursively to find the maximum value possible. In here, i represents the current item we're considering and r is the remaining weight we can still use.

Algorithm 15 Recursive Knapsack

```
Require: Values $V[1..n]$, Weights $W[1..n]$, Capacity $C$
  V, W
  function KNAPSACK($i, r$)
      if $i = n + 1$ or $r = 0$ then                    ▷Base case
          return 0          ▷Either we've considered all items or filled the knapsack
      end if                                  ▷No more value can be added
      skip ← KNAPSACK($i + 1, r$)    ▷1st Op: Skip current item and maintain same $r$
      take ← 0                            ▷2nd Op: Try to include curr item
      if $W[i] \leq r$ then      ▷Only try taking if item's weight fits in remaining $r$
          take ← $V[i]$+ KNAPSACK($i + 1, r - W[i]$)    ▷Add current item's value + best value from
  remaining items
      end if                   ▷Subtract current item's weight from remaining $r$
      return max($take, skip$)          ▷Return best possible value between two choices
  end function
  // Start considering from first item with full $r$
  return KNAPSACK($1, C$)
```

Now let's apply memoization.

Algorithm 8.6 (Dynamic Programming Knapsack)

The key insight is that $K[i, r]$ (best value possible using items up to i exclusive with remaining capacity r) can be built from $K[i-1, r]$ and $K[i-1, r - W[i]]$ using the formula

$$K[i, r] = \begin{cases} K[i-1, r] & \text{if } W[i] > r \text{ (can't take item i)} \\ \max(K[i-1, r], V[i] + K[i-1, r - W[i]]) & \text{if } W[i] \leq r \text{ (can take item i)} \end{cases} \tag{89}$$

In here, i represents the items 1..i we're considering and r represents the remaining capacity.

Algorithm 16 Dynamic Programming Knapsack

```
Require: Values $V[1..n]$, Weights $W[1..n]$, Capacity $C$
  V, W
  function DPKnapsack($C$)
      Create table $K[0..n, 0..C]$          ▷K[i,r] = max value using items V[:i] with remaining r
      Initialize all entries to 0
      for $i \leftarrow 1$ to $n$ do          ▷Consider each item
          for $r \leftarrow 0$ to $C$ do          ▷Consider each possible remaining capacity
              $K[i, r] \leftarrow K[i-1, r]$          ▷Default: inherit value from excluding item i
              if $r \geq W[i]$ then          ▷If current item fits in remaining capacity
                  take $\leftarrow V[i] + K[i-1, r-W[i]]$          ▷Value of item i + best value with remaining r
                  $K[i, r] \leftarrow \max(K[i, r], \text{take})$          ▷Take better of including or excluding
              end if
          end for
      end for
      return $K[n, C]$          ▷Best value possible using all items
  end function
```

The memory complexity is obviously $\Theta(nC)$. The number of subproblems is nC, and the processing for each step is constant time (possibly addition and max), so $O(1)$. Therefore the total runtime is $O(nC)$ also. Note that if we compare this in terms of the bit runtime, then this is $O(n \log_2 C)$, which is psuedopolynomial since C is described by $\log_2 C$ bits. However, the n will scale linearly since it is the size of the array and not the size of each integer in V, W.

Note that if C can be very big, and this can be problematic.

8.3 Line Breaking

Definition 8.4 (Line Breaking)

Line breaking is used whenever you compile a tex document. Given

- a sequence of words $w_1, w_2, ..., w_n$ where w_i has length l_i
- a maximum line length L
- each line must contain whole words in order, separated by spaces

Our goal is to break words into lines to minimize the sum of squares of empty spaces at the end of each line, i.e. our cost function for a sequence of words $W[i : j+1]$ is

$$\left(L - \sum_{k=i}^{j} w_k \right)^2 \tag{90}$$

If we used the absolute value, there exists a greedy solution.

Example 8.6 (Line Breaking Example)

Given $L = 20$ and $W = [12, 8, 9]$ (note that spaces won't count in length in this problem), we have 3 possible arrangements.

1. Option 1: First two words on first line
   - Line 1: $[12 + 8 = 20]$ (0 spaces remain)
   - Line 2: $[9]$ (11 spaces remain)
   - Total cost $= 0^2 + 11^2 = 121$
2. Option 2: Words one per line

- Line 1: [12] (8 spaces remain)
- Line 2: [8] (12 spaces remain)
- Line 3: [9] (11 spaces remain)
- Total cost $= 8^2 + 12^2 + 11^2 = 64 + 144 + 121 = 329$

3. Option 3: Word 2 and 3 together
   - Line 1: [12] (8 spaces remain)
   - Line 2: $[8 + 9 = 17]$ (3 spaces remain)
   - Total cost $= 8^2 + 3^2 = 64 + 9 = 73$

Therefore, Option 3 is optimal with cost 73. This also demonstrates that the greedy strategy, which gives Option 1, will not work.

We start off with recursive brute force. Given any word w_i, we can either end the line there $w_i|$ or add another word $w_i w_{i+1}$.

Algorithm 8.7 (Recursive Brute Force Line Breaking)

The key idea is at each position i, we try placing different numbers of words on the current line and recursively solve for the rest. For each word i, we try all possible ways to break the line starting at word i. Let $MinCost(i)$ be the minimum cost to arrange words $[i..n]$. Then:

$$MinCost(i) = \begin{cases} 0 & \text{if } i > n \\ \min_{i \leq j \leq n} \left\{ (L - \sum_{k=i}^{j} W[k])^2 + MinCost(j+1) \right\} & \text{if else} \end{cases} \tag{91}$$

where $(L - \sum_{k=i}^{j} W[k])^2$ is the cost to put the words $i...j$ on a new line and then ending. So whenever we add the new ith word at the end of the line, we are looking at all words j at which we can break the line, and taking the minimum cost given this line break. The line break from $j+1$ should be computed as well (this is what we will store in our DP array later).

Algorithm 17 Recursive Line Breaking

```
Require: Word lengths $W[1..n]$, Line length $L$
  W
  function MinCost($i$)                                  ▷Returns min cost for words[i..n]
      if $i = n + 1$ then                                    ▷Base case: no words left
          return 0
      end if
      min_cost ← $\infty$
      lineLen ← 0
      for $j \leftarrow i$ to $n$ do                     ▷Try placing words i through j on current line
          lineLen ← lineLen $+W[j]$
          if lineLen $\leq L$ then                          ▷If these words fit on the line
              spaces ← $L-$ lineLen                          ▷Extra spaces at end of line
              cost ← spaces$^2$+ MinCost($j + 1$)                 ▷Cost of this line + rest
              min_cost ← min(min_cost, cost)
          end if
      end for
      return min_cost
  end function
  return MinCost(1)                                          ▷Start with first word
```

Example 8.7 (Recursive Brute Force Line Breaking)

For example, with $L = 20$ and $W = [12, 8, 9]$:

- At $i = 1$: Try
  12 alone (8 spaces) + solve for [8,9]
  12,8 together (0 spaces) + solve for [9]
- At $i = 2$: Try
  8 alone (12 spaces) + solve for [9]
  8,9 together (3 spaces) + solve for []
- At $i = 3$: Try
  9 alone (11 spaces) + solve for []

Algorithm 8.8 (Dynamic Programming Line Breaking)

The key insight is that $DP[i]$ represents the minimum cost of optimally arranging words $[i..n]$. For each word i, we try placing words $[i..j]$ on a line and add the cost of optimally arranging the remaining words $[j + 1..n]$. We do the same logic but rather than computing it we just retreive it from the DP array.

$$DP[i] = \begin{cases} 0 & \text{if } i > n \\ \min_{i \leq j \leq n} \left\{ (L - \sum_{k=i}^{j} W[k])^2 + DP[j+1] \right\} & \text{if else} \end{cases} \tag{92}$$

Algorithm 18 Dynamic Programming Line Breaking

```
Require: Word lengths $W[1..n]$, Line length $L$
  W
  function LineBreak($L$)
      Create array $DP[0..n]$                          ▷DP[i] = min cost for words[i..n]
      Initialize all entries to $\infty$
      $DP[n+1] \leftarrow 0$                           ▷Base case: no words left
      for $i \leftarrow n$ downto 1 do                 ▷Consider each starting word
          lineLen $\leftarrow 0$
          for $j \leftarrow i$ to $n$ do               ▷Try placing words i through j on a line
              lineLen $\leftarrow$ lineLen $+W[j]$
              if lineLen $\leq L$ then                 ▷If these words fit on the line
                  spaces $\leftarrow L-$ lineLen       ▷Extra spaces at end
                  cost $\leftarrow$ spaces$^2 + DP[j+1]$   ▷Cost of this line + rest
                  $DP[i] \leftarrow \min(DP[i]$, cost)     ▷Update if better
              end if
          end for
      end for
      return $DP[1]$                                   ▷Cost of optimally arranging all words
  end function
```

The memory complexity is clearly $O(n)$. To add each element i, we must iterate over all the possible j's, making this $O(n^2)$ total iterations. However, computing the cost is also $O(n)$, making the total runtime $O(n^3)$. However, if we also store another DP array `sums`, which can be computed in linear time and stores

$$\texttt{sums}[i] = \sum_{k=i}^{n} w_k \implies \sum_{k=i}^{j} = \sum_{k=i}^{n} w_k - a \sum_{k=j+1}^{n} w_k = \texttt{sums[i]} - \texttt{sums[j+1]} \tag{93}$$

which can be accessed and computed in $O(1)$ time, bringing us down to $O(n^2)$.

Example 8.8 (DP Line Breaking)

For example, with $L = 20$ and $W = [12, 8, 9]$:

- $DP[4] = 0$ (base case)
- $DP[3] = 11^2 = 121$ (only option for last word)
- $DP[2] = \min(12^2 + 121, 3^2) = \min(265, 9) = 9$ (alone or with word 3)
- $DP[1] = \min(8^2 + 9, 0^2 + 121) = \min(73, 121) = 73$ (alone or with word 2)

8.4 Bellman Ford Revisited

Recall the Bellman equations that we must solve using DP. That is, given some $s \in V$,

$$d[v] = \min_w \{d[w] + l_{wv}\} \tag{94}$$

with $d[s] = 0$. Note that the problem was not well defined if there are negative cycles in the graph, since we can just loop an infinite number of times. However, we can modify the bellman equations to get a better sense.

Theorem 8.1 (Modified Bellman Equations)

For paths of length at most i edges, the Bellman equations become:

$$d(v, i) = \begin{cases} 0 & \text{if } v = s \\ \min\{d(v, i-1), \min_{(u,v)\in E}\{d(u, i-1) + w_{uv}\}\} & \text{otherwise} \end{cases} \tag{95}$$

where $d(v, i)$ represents the shortest path from source s to vertex v using at most i edges, and w_{uv} represents the weight of edge (u, v). Note that the inner minimum takes the minimum over all paths with the final edge connecting from some other node u to target v, and the outer minimum compares this minimum path to what we already have to see if it's an improvement.

This indicates that we should use a 2D DP array to memoize. This allows you to have a more flexible representation in case there are negative cycles since we are also limiting the number of edges a path could have.

Algorithm 8.9 (2D DP Bellman-Ford)

The implementation of the modified equations gives us the following algorithm.

```
Require: Nodes $V$, Edges $E$, source $s$
 1: $d \leftarrow$ 2D array of size $|V| \times |V|$ initialized to $\infty$
 2: $d[s, 0] \leftarrow 0$                          ▷Base case: Can reach source with 0 edges
 3: for $i = 1, \ldots, n-1$ do
 4:     for $v \in V$ do
 5:         $d[v, i] \leftarrow d[v, i-1]$                ▷Keep best path seen so far
 6:         for $(u, v) \in E$ do
 7:             $d[v, i] \leftarrow \min(d[v, i], d[u, i-1] + l_{uv})$
 8:         end for
 9:     end for
10: end for
```

Note that this algorithm still has time complexity $O(nm)$ because the outer loop runs $n - 1$ times and for each iteration, we examine each edge once. The space complexity is $O(n^2)$, and finally, note the important properties.

1. *Monotonicity*: For all vertices v and indices i:
$$d(v, i) \leq d(v, i-1) \tag{96}$$
This is because any path using $\leq (i-1)$ edges is also a valid path using $\leq i$ edges.
2. *Convergence*: The algorithm will stabilize after at most $n-1$ iterations since:
    - Any shortest path without negative cycles can use at most $n-1$ edges
    - Therefore, $d(v, n-1) = d(v, n)$ for all v if no negative cycles exist
3. *Negative Cycle Detection*: A negative cycle exists if and only if:
$$\exists v \in V : d(v, n) < d(v, n-1) \tag{97}$$
This is because any improvement after $n-1$ edges must use a negative cycle.

9 Probabilistic Algorithms

9.1 Primality Testing

10 Flows and Cuts

10.1 Max-Flow Min-Cut Theorem

Definition 10.1 (Flow Network)

Given a nonnegative weighted directed graph $G(V, E)$, we can define the weights to be the **capacity** $C : E \to \mathbb{R}^+$ of some commodity that can travel through the edge. Say that we have a source node $s \in V$ and a sink/destination node $t \in V$. This is called a **flow network**.

Definition 10.2 (Flow)

Given a flow network, a **flow** is a function $f : E \to \mathbb{R}^+$ satisfying

1. *Conservation.* What flows in must always flow out. For all $v \neq s, t$, we have

$$\sum_{(u \to v) \in E} f(u \to v) = \sum_{(v \to w) \in E} f(v \to w) \tag{98}$$

2. *Feasibility.* We cannot exceed the capacity. For all $e \in E$,

$$0 \leq f(e) \leq C(e) \tag{99}$$

The **value** of a flow represents the actual/realized amount of the commodity that we can push through the graph

$$|f| = \sum_{(s \to u) \in E} f(s \to u) - \sum_{(u \to s) \in E} f(u \to s) \tag{100}$$

where we need to subtract the right expression to avoid loops.

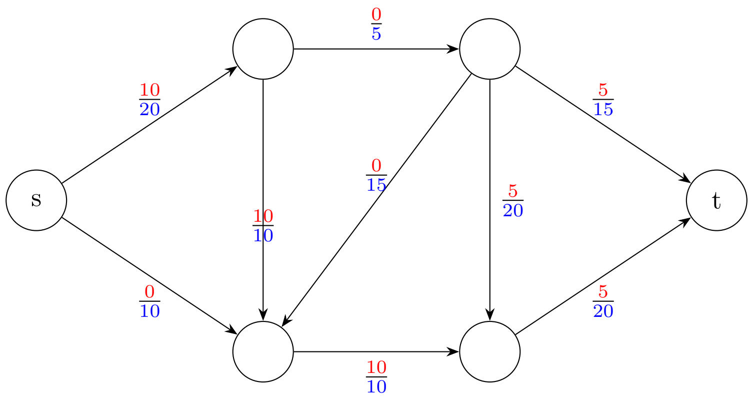


Figure 31: The red values represent the flow. The blue values represent the capacity. The value of the above flow is 10.

Now let's revisit cuts.

Definition 10.3 (Cuts)

Given a flow network, a s-t **cut** is a partition of $V = S \sqcup T$ s.t. $s \in S, t \in T$. The **value**, or **capacity**, of a cut is defined as the capacity of its edges

$$||S,T|| = \sum_{u in S} \sum_{v \in T} C(u \to v) \tag{101}$$

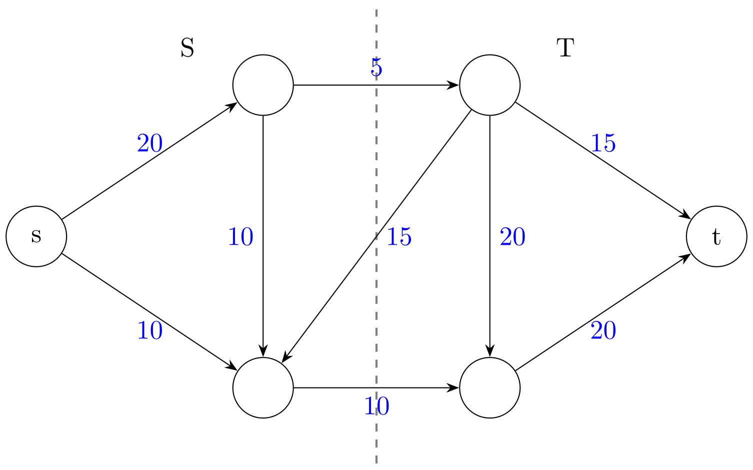


Figure 32: This cut has capacity 15. Note that the cut going from right to left is not included since this is a S-T cut.

We now introduce two related problems.

1. Max Flow. How do we find the flow with the maximum value? This is analogous to maximizing the commodity transport from s to t.
2. Min Cut. How do we find the cut with the minimum capacity? This is analogous to removing the smallest total edges that will disconnect s to t.

Theorem 10.1 (Max-Flow Min-Cut Theorem)

For any valid (s,t) flow f on a flow network $G = (V, E)$ and any valid (s,t)-cut S, T, we have

$$|f| \leq ||S,T|| \tag{102}$$

It also turns out that equality is achieved, which immediately implies that the max-flow is the min-cut.

Proof.

We can see that in order for a flow to have a value, given some cut the flow value cannot exceed the capacity through this cut from S to T. Mathematically, we can see that the flow value is invariant due to conservation, and so

$$\begin{aligned} |f| &= \sum_{s \to v} f(s \to v) - \sum_{v \to s} f(v \to s) && (103) \\ &= \sum_{u \in S} \sum_{v \in T} f(u \to v) - \sum_{v \in T} \sum_{u \in S} f(v \to u) && (104) \\ &\leq \sum_{u \in S} \sum_{v \in T} c(u \to v) = ||S,T|| && (105) \end{aligned}$$

The fact that equality is achieved is highly nontrivial, and we will not cover it here.

10.2 Maximum Flows

Now to compute the maximum flow itself, we use a residual graph. For now, we will assume that a flow network G does not have 2-cycles (cycles of length 2), and if we do find a graph with such a 2-cycle, we modify it by adding an intermediate node.

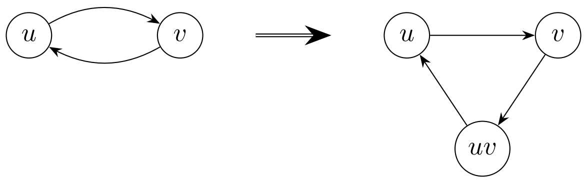


Figure 33: How we modify our 2-cycle.

Definition 10.4 (Residual Network)

Given a flow network G and a flow f, the **residual graph** G_f is a new flow network that has edges $u \to v$ and $v \to u$, where

$$c_f(u \to v) = \begin{cases} c(u \to v) - f(u \to v) & \text{if } u \to v \in E \\ f(u \to v) & \text{if } v \to u \in E \end{cases} \tag{106}$$

Basically, if there is some flow in an edge, we replace that edge's capacity by the residual capacity, giving us our *forward edge*, and then take this flow and add it to a new opposite pointing edge, called the *backwards edge*.

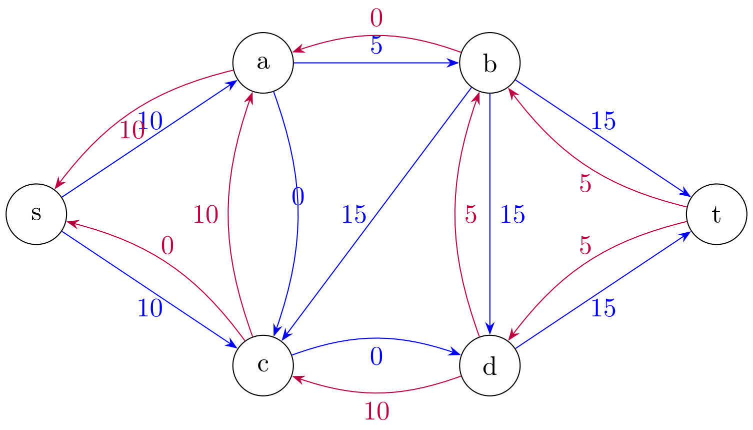


Figure 34: Residual network.

Theorem 10.2 ()

If we have a flow f, its value $|f|$ can be improved/augmented up to the bottleneck capacity on any residual path. That is, if we find any $s \to t$ path π on G_f, with a positive bottleneck capacity (minimum capacity over its edges), we can improve the flow.[a] Given this residual path π, it may have both forward or backwards edges. To augment the original path on G by value b, we look at each edge $(u \to v) \in E$ and

1. If it corresponds to a forward edge (i.e. if $(u \to v)$ is in the residual path π), we add b. This corresponds to how much more flow you can put into this edge.

2. If it corresponds to a backward edge (i.e. if $(v \to u)$ is in the residual path π), we subtract b. The corresponds to how much flow could you reroute in the backwards direction and have it reach t through another path.
3. If it is not in π, then we don't change it.

In summary, the new flow f' is defined

$$f'(u \to v) = \begin{cases} f(u \to v) + b & \text{if } (u \to v) \in \pi \\ f(u \to v) - b & \text{if } (v \to u) \in \pi \\ f(u \to v) & \text{else} \end{cases} \tag{107}$$

[a]For example, the capacity of the path S, C, A, B, T is 5, since $A \to B$ has weight 5.

Proof.

We first show that f' is a valid flow. For feasibility,

1. *Forward.* If $(u \to v) \in \pi$, then

$$f'(u \to v) = f(u \to v) + b \leq f(u \to v) + c_f(u \to v) = c(u \to v) \tag{108}$$

2. *Backward.* If $(v \to u) \in \pi$, then

$$f'(u \to v) = f(u \to v) - b \geq f(u \to v) - c_f(v \to u) = 0 \tag{109}$$

For conservation, there are 4 cases.

1. Forward into node, forward out of node. $\Delta_{in} = +b, \Delta_{out} = +b$. Therefore the flows cancel out.
2. Forward into node, backward out of node. $\Delta_{in} = +b - b = 0$.
3. Backward into node, forward out of node. $\Delta_{out} = +b - b = 0$.
4. Backward into node, backward out of node. $\Delta_{in} = -b, \Delta_{out} = -b$.

At this point, we're pretty much ready to give our algorithm on max-flow.

Algorithm 10.1 (Ford-Fulkerson)

We've reduced this problem to a problem of finding a positive path π in the residual graph, which we have a lot of algorithms built for. We find such a path and augment the flow until we cannot find such a path. If we cannot find a path, then we can use BFS to look at all vertices $S \subset V$ reachable from s in G_f, and then let $T = V \setminus S$. These two disjoint sets will give the min cut.

```
Require: Flow network $G(V, E)$, source/sink $s, t \in V$. Capacity $c$.

  function MaxFlow(G)
      Start with the trivial flow $f$, maybe stored as array $[0, \dots, 0]$ of length $E$.
      Construct residual graph $G_f$ for this trivial flow.
      while $G_f$ has an augmenting path in $G_f$ do          ▷Maybe use BFS to find this.
          Pick $\pi$ as any augmenting path
          augment $f$ by bottleneck capacity
          update $G_f$.
      end while
      return f
      Use BFS to find all nodes in $G_f$ reachable from $s$. Call this $S$.
      Let $T = V \setminus S$.
      $S, T$ is our min cut, and $f$ is your max flow.
  end function
```

The inner loop is just BFS, which is $O(N + M)$, but how many iterations do we do? Well for integer capacities, we can bound it by the actual value of the max flow $|f^*|$ since we must improve by at least $+1$ every loop.[a] Unfortunately, $O((N + M)|f^*|)$ is the best we can do in generality, which is pseudopolynomial.

[a]This is not a trivial result that if all capacities are integers, then there is an integer max-flow, despite the actual flow being dispersed as fractions across edges.

There are indeed polynomial time algorithms, e.g. Orlin's algorithm with runtime $O(NM)$, which is guaranteed to be polynomial but in practice may be slower.

Example 10.1 (Max Flow and Residual Graph)

Here is the max flow.

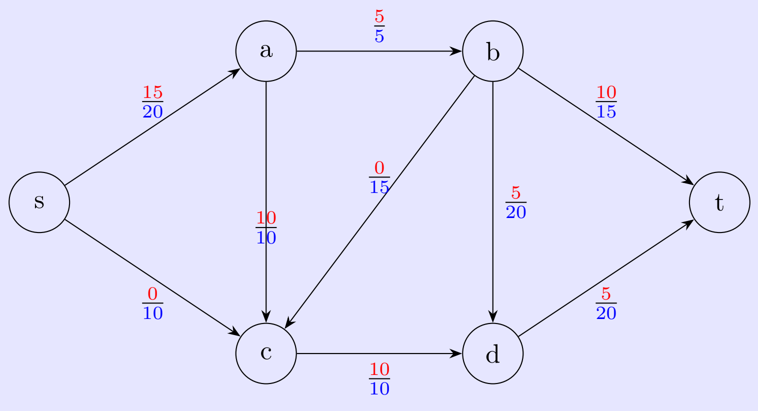


Figure 35: The max flow of the above graph.

Here is the residual graph, which shows indeed that this the max flow. The min cut is therefore $S = \{s, a, c\}$ and $T = \{t, b, d\}$.

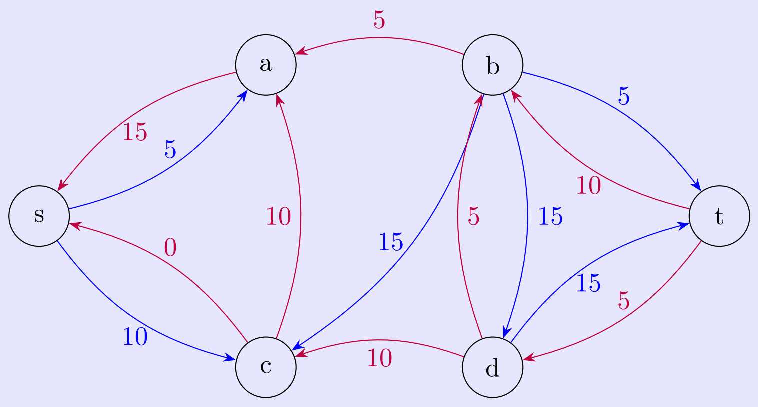
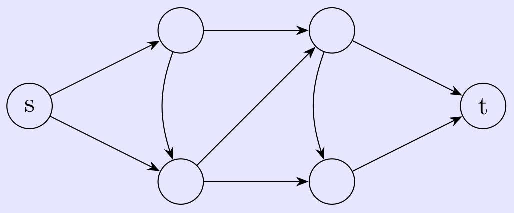


Figure 36: You can see that there are no paths from s to t anymore. The edges $a \to b$ and $c \to d$ in the original graph are saturated and so have weight 0 in the residual graph.

Example 10.2 (Edge Disjoint Paths)

Given a directed graph $G(V, E)$ with start and end nodes $s, t \in V$, we want to find the maximum number of paths from $s \to t$ s.t. they do not share any edge, i.e. are *edge-disjoint*.

Figure 37: This graph has two edge-disjoint paths: the top and bottom paths.



The idea is to reduce the maximum flow by first assigning capacity 1 to every edge. Then we compute the maximum flow f^*, and then return $|f^*|$. It turns out that even if we chose a path π that went through the center node, the augmentation would push us to redirect the flows to the top and bottom outer edges. Since $|f^*| \leq n - 1 < m$ in this example (since it can be at most the maximum indegree or outdegree of s, t), we should probably use Ford-Fulkerson rather than Orlin's algorithm.

This leads to the lemma.

Theorem 10.3 (Edge-Disjoint Paths)

Given directed graph G and its corresponding flow network G' of capacity 1, the following are true:

1. If there exists k edge-disjoint paths in G, then there exists a flow with value $\geq k$ in G'.
2. If there exists a flow of value k in G', then there exists k edge-disjoint paths in G.

Proof.

The first is trivially feasible and conservation is satisfied since for any node $u \in V \setminus \{s, t\}$, every path in P (the collection of k paths) that enters u must exit u, so the total flow in equals the total flow out $\implies f$ is a valid (s, t)-flow. Therefore, we have k paths leaving s, so $|f| = k$.
For the other direction, let f be the flow with value k, and let $P = \{\}$. Let H be the subgraph of G' of edges with positive flow, and for every loop, we use BFS from s to t to get a path π. We then add

π to P and remove π from H, repeat this k times until we get k paths in P.

The way that we have constructed a collection of paths from a flow function is very useful, and is called *flow decomposition*. We can go back and forth between a flow and a collection of paths by doing what we have just done in the previous proof.

Definition 10.5 (Flow Decomposition)

Given a general flow f on flow network G, we initialize our set of paths $P = \{\}$ and loop the following:

1. Use BFS from s to t to get a path π.
2. Add π to P and rather than strictly removing the path from H, we need to be mindful of how much flow is left on it after removing only the bottleneck flow on the path.

This can be done in $O(NM)$ time, which loosely is because we are running BFS $O(M)$ times.

Example 10.3 (Undirected Edge-Disjoint Paths)

What if G is undirected? In this case, we want to construct a flow network and then convert its 2 cycles into a 3 cycles by adding artificial points, taking linear time.

Example 10.4 (Undirected Edge-Disjoint Paths)

Another problem is about vertex-disjoint paths, which is a stronger restriction than edge-disjoint ones. Each node does not have a concept of a capacity, so the general idea is for each vertex v, replace it with an "in-out gadget" with an edge of weight 1. The rest of the details follow similarly from edge-disjoint paths.

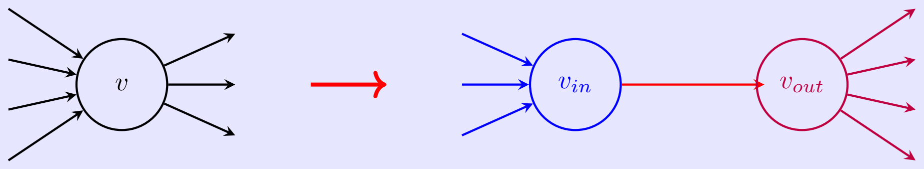


Figure 38

Example 10.5 (Maximum Bipartite Matching)

Another application is in **bipartite matching**, which for an undirected bipartite graph $G(V, E)$ with $V = A \sqcup B$, is a subset of the edges $M \subset E$ s.t. no 2 edges in M share an endpoint. The goal is to find the maximum bipartite matching, i.e. the matching with the greatest total weight.

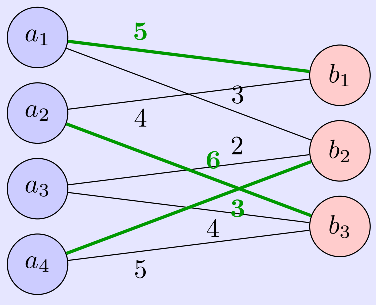
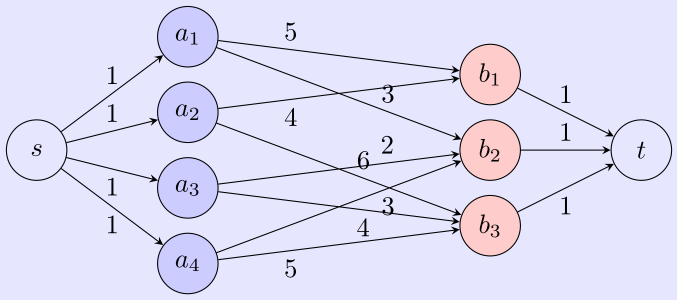


Figure 39: A bipartite matching

The general idea is to take an undirected graph, convert it to a flow network, compute the max flow, i.e. min cut, and then translate it back into the undirected graph solution.

1. Construct the flow network $G' = (V', E')$, where $V' = V \cup \{s, t\}$, with additional edges going from s to A and B to t all with weights 1. Replace the middle edges with directed edges towards the flow with the same weight (though it can be any weight at least 1).



2. Compute max flow f in G'.
3. Return $M = \{(u \to v) \in E \mid f(u \to v) = 1\}$. This can be done using the flow decomposition or just by looking at the edges in our flow with value 1 and mapping it to the corresponding undirected bipartite graph.

To see correctness, we claim that there exists a matching M in G of size k iff there exists a flow f in G' with $|f| = k$, and therefore the max matching corresponds to the max flow. Constructing the flow network is $O(N + M)$ since $N' = N + 2$ and $M' = M + N$. Computing the max-flow is $O(N'M')$ or $O(M'|f|)$ depending on if we use Orlin's or Ford-Fulkerson, but either one is $O(NM)$. Then mapping it back to M might take $O(M)$, so the total time is $O(NM)$.

11 Efficiency

11.1 Classification of Decision Problems

We have talked about *optimization problems* that choose the best solution from a class of solutions. Now we will talk about *decision problems*, which seek to prove or disprove the existence of a solution satisfying a certain property.

Lemma 11.1 (Conversion between Optimization and Decision Problems)

It turns out we can convert one into the other easily by rephrasing the problem. The following conversion will be used frequently.

1. Given a problem P, we want to find a solution that minimizes some variable K.
2. Given a problem P and some constant K', does there exist a solution that is less than K'?

Voila.

Now that this is established, we can freely consider the classes P and NP of all decision problems and classify them. There are equivalent definitions for optimization problems, but it is more popular to pose these classes as decision problems.

Definition 11.1 (P)

P (Polynomial Time) is the set of all decision problems that can be solved by polynomial-time algorithms.

Definition 11.2 (NP)

NP (Nondeterministic Polynomial Time) is the set of all decision problems for which if it returns 1 (i.e. a solution exists), this can be *verified* in polynomial time.[a]

[a]The fact that a *proposed* solution can be verified is important! Proving or disproving a proposed solution is much easier than proving the nonexistence of a solution.

The other way around $NP \subset P$ is an unsolved millenium conjecture. It attempts to answer what the runtime difference or difficulty is between solving and verifying a problem? The conjecture states that solving is exponentially harder that verifying.

Theorem 11.1 ()

$P \subset NP$

Proof.

Given a problem π with a proposed solution p, we can run p in polynomial time to verify it.

Colloquially, the set of problems which are at least as hard as NP is called *NP-hard*, which can be defined for both optimization and decision problems. These are the absolute hardest problems. The formal definition of NP-Hard problems is shown below.

Definition 11.3 (NP-Hard)

A (decision or optimization) problem π is NP-hard iff the existence of a polynomial time algorithm for π implies $P = NP$.

Definition 11.4 (NP-Complete)

The intersection of NP-hard and NP is **NP-complete**, which can be thought of as the hardest problems existing in NP. The worst case runtime complexity of an NP-complete problem is exponential.

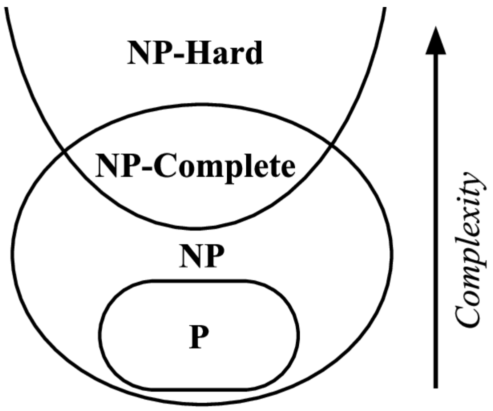


Figure 40: Diagram of the 4 classes. The complexity rises as we go up.

11.2 Proving NP-Hardness

To show that a problem is in P, all we have to do is find a polynomial time algorithm, which is what we've been doing this entire course. To show whether a problem is NP, we should show that it is verifiable in polynomial time, which we have also done so far. Proving NP-hardness (which then allows us to prove NP-completeness) is a bit more tricky. The pattern is always three things.

1. *Reduction.* Describe a polynomial time algorithm f to transform an *arbitrary instance* x of a problem β into a *special instance* $y = f(x)$ of a problem π.
2. *Equivalence.* Prove that if arbitrary x is a true instance of π, then y is a true instance of π.
3. *Equivalence.* Prove that if arbitrary y is a true instance of π, then x is a true instance of π.[9]

Definition 11.5 (Reduction)

In other words, to prove a problem π is NP-hard, reduce[a] a known NP-hard problem β to π in polynomial time (not the other way around!). If this reduction is possible, then we claim that π must be at least as hard as β: $\beta \leq_p \pi$.

1. Assume that we can solve π in polytime.
2. Then we can solve β in polytime by first converting any instance of β to π in polytime, and then polytime solving π.[b]
3. Therefore, $\pi \in P \implies \beta \in P$.

But since we already know β is NP-hard, this creates a contradiction. Therefore, $\pi \notin P$.

[a]The fact that we say reduce here is misleading, since we are showing that the reduction is at least as hard as the original.

[b]In other words, there exists an injective (but not necessarily surjective) mapping from the set of solutions of π to the set of solutions of β.

The reduction serves to compare the two problems, and the equivalence then serves to compare the set of solutions to these two problems.

[9]This is the hard part.

Definition 11.6 (Equivalence of Solutions)

The next step is to show two things:

1. A solution to π implies a solution to β, and
2. A solution to β implies a solution to π.

As a general overview of popular problems and their relation to each other, we'll mention this now.

Theorem 11.2 (Reduction Tree of NP-Hard Problems)

For convenience.

1. 3SAT is NP-hard by Cook-Levin
2. CircuitSAT is NP-hard by reducing from 3SAT
3. Max Independent Set is NP-hard by reducing from 3SAT
4. Min Vertex Cover is NP-hard by reducing from MIS
5. Max Clique is NP-hard by reducing from MIS
6. Hamiltonian Path is NP-hard
7. Hamiltonian Cycle is NP-hard by reducing from MVC, 3SAT, or Hamiltonian Path
8. Graph Coloring is NP-hard by reducing from 3SAT
9. Subset Sum is NP-hard by reducing from MVC
10. Planar CircuitSAT is NP-hard by reducing from CircuitSAT
11. Paritioning[a] is NP-hard by reducing from Subset Sum.
12. Set Cover[b] is NP-hard.
13. Longest Path is NP-hard.
14. MaxCut[c] is NP-hard.

[a]Given a set S of n integers, are there disjoint subsets A, B s.t. their sums are equal?

[b]Given a collection of sets S, find the smallest sub-collection of sets that contains all the elements in $\cup S$.

[c]Find the max cut of a graph.

11.3 Logic and Satisfiability

Definition 11.7 (CircuitSAT)

Consider a black box circuit f, with input $\mathbf{x} \in \{0,1\}^n$. We want to answer the question of whether there exists some $\mathbf{x}$ s.t. $f(\mathbf{x}) = 1$, i.e. whether the circuit is **satisfiable**?

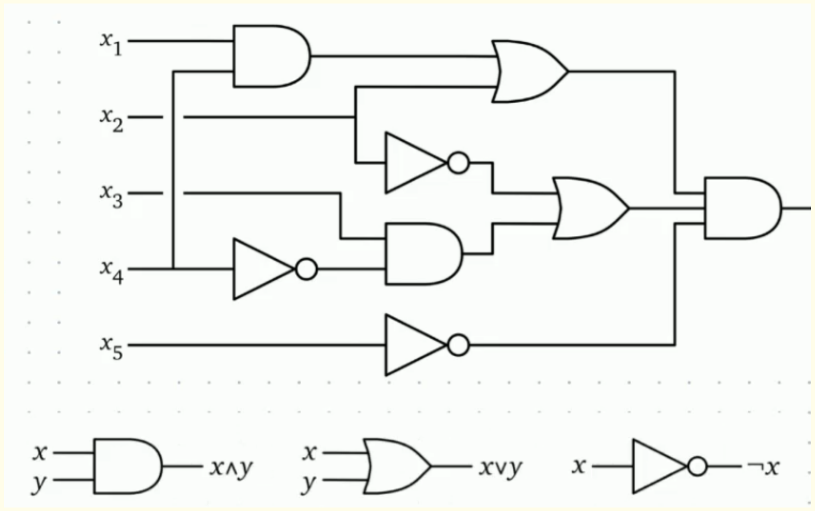


Figure 41: Some random circuit where $n = 5$.

The obvious way is to brute force is an check all 2^n combinations, and this is the only way since this is black-box. However, even if we knew the circuit itself, the amount of computations that we need to solve this is conjectured to be still $O(2^n)$, though we may be able to cleverly shave off some calculations here and there.[a]

[a]It is not proven that there exists no exponential algorithm.

Definition 11.8 (3SAT)

Given a boolean formula ϕ of N variables where each **clause** contains exactly 3 literals, determine if there exists a satisfying assignment to the variables that makes ϕ true. Formally:

$$\begin{aligned} \phi &= C_1 \wedge C_2 \wedge \cdots \wedge C_m \\ C_i &= (l_{i1} \vee l_{i2} \vee l_{i3}) \\ l_{ij} &\in \{x_k, \neg x_k\} \text{ for some } k \in \{1, \ldots, n\} \end{aligned}$$

where x_k are boolean variables and $\neg x_k$ their negations. For example:

$$\phi = (x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee x_2 \vee x_4) \wedge (x_2 \vee \neg x_3 \vee \neg x_4)$$

Like CircuitSAT, this problem is also believed to require $O(2^n)$ time, and in fact these problems are polynomially reducible to each other (CircuitSAT $\leq_p$ 3SAT and 3SAT $\leq_p$ CircuitSAT). The key difference is that 3SAT has a very specific structure that makes it easier to analyze and reduce other problems to, making it a canonical NP-complete problem.[a]

[a]The restriction to exactly 3 literals per clause is not arbitrary - 2SAT is solvable in polynomial time, while 3SAT is NP-complete.

In fact, there is a polynomial time algorithm to translate between 3SAT and CircuitSAT, so the two are theoretically equivalent.

Theorem 11.3 (P, NP Classification of SAT)

For CircuitSAT,

$$\texttt{CircuitSAT} \not\in P, \texttt{CircuitSAT} \in NP \tag{110}$$

That is, it cannot be solved in polynomial time, but given a solution circuit and a satisfying assignment, the assignment can be verified in polynomial time. For 3SAT,

$$\texttt{3SAT} \not\in P, \texttt{3SAT} \in NP \tag{111}$$

That is, it cannot be solved in polynomial time, but given a solution formula and a satisfying assignment, the assignment can be verified in $O(N^3)$ time (N^3 comes from the size of the expression).

Theorem 11.4 (Cook-Levin)

3SAT is NP-complete.

Proof.

The proof is extremely long, and we use this as the initial point from which we start proving NP-hardness of other problems using reduction.

For example, to prove that CircuitSAT is NP-hard, we should reduce 3-SAT to CircuitSAT in polynomial time and then we are done. Reduction is quite complicated, so we omit this for now.

11.4 Graph Problems

11.4.1 Maximum Independent Set

Definition 11.9 (MIS)

Given an undirected graph $G(V, E)$, an **independent set** is a subset $S \subseteq V$ such that no two vertices in S are connected by an edge in E. The **maximum independent set** problem asks
1. Given G, find an independent set of maximum cardinality.
2. Given G and constant K, does there exist an independent set of size $\geq K$?

Theorem 11.5 (Maximum Independent Set (MIS))

MIS is NP-hard.

Proof.

We prove NP-hardness by reducing from 3SAT, which is more structured than CircuitSAT. Given a 3SAT instance ϕ, we construct a graph G such that ϕ is satisfiable if and only if G has an independent set of size m, where m is the number of clauses in ϕ. The reduction works as follows: For a 3SAT formula $\phi = C_1 \wedge C_2 \wedge \cdots \wedge C_m$ where each clause $C_i = (l_{i1} \vee l_{i2} \vee l_{i3})$, construct graph $G(V, E)$:
1. For each clause C_i, create a **clause gadget**: a triangle with vertices labeled l_{i1}, l_{i2}, and l_{i3} with undirected edges between them.
2. Add **consistency edges**: for any two literals l_{ij} and $l_{k\ell}$ from different triangles, add an edge if they are complementary (i.e., one is x and the other is $\neg x$)

This reduction has the following properties:
- Each triangle forces us to choose at most one literal from each clause
- Consistency edges ensure we can't choose contradictory literals

- Choosing m vertices (one per triangle) corresponds to making each clause true

For equivalence, it turns out that ϕ is satisfiable iff the size of the IS in G is exactly the number of clauses in ϕ.

- ($\Rightarrow$) If ϕ is satisfiable, choose one true literal from each clause to form an independent set of size m
- ($\Leftarrow$) If G has an independent set of size m, it must contain exactly one vertex per triangle (by pigeonhole principle), giving us a valid satisfying assignment

Therefore, now that we have established the reduction, since 3SAT is NP-complete and we have a polynomial-time reduction to MIS, MIS is NP-hard.

Example 11.1 (3SAT Reduction of MIS)

Consider the 3SAT formula

$$\phi = (a \vee b \vee c) \wedge (b \vee \neg c \vee \neg d) \wedge (\neg a \vee c \vee d) \wedge (a \vee \neg b \vee \neg d) \tag{112}$$

Following our reduction, we construct graph G as follows:

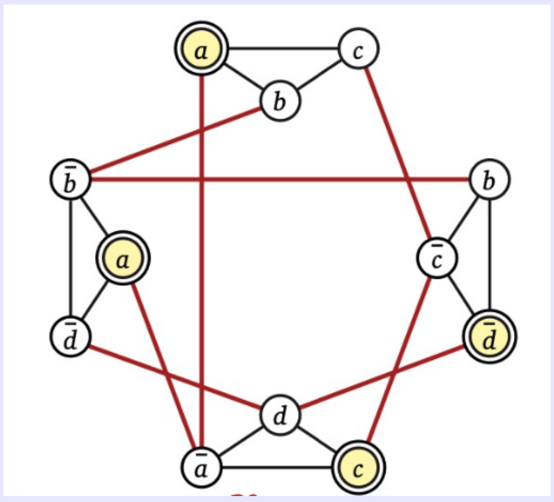


Figure 42: Reduction graph from 3SAT to MIS.

The graph contains:

1. Four triangles, one for each clause of ϕ
2. Red edges connecting complementary literals (e.g., b with $\bar{b}$)
3. A maximum independent set highlighted in yellow ($\{a, d\}$) of size 4

This independent set corresponds to the satisfying assignment $(a, b, c, d) = (1, *, 1, 0)$.

- $(a \vee b \vee c)$ is satisfied by $a = 1$
- $(b \vee \neg c \vee \neg d)$ is satisfied by $d = 0$.
- $(\neg a \vee c \vee d)$ is satisfied by $c = 1$.
- $(a \vee \neg b \vee \neg d)$ is satisfied by $a = 1$.

To demonstrate the proof, we assume that this MIS can be solved in polynomial time, so $MIS \in P$. But since $3SAT \leq_p MIS$, this means that $3SAT \in P$, which is a contradiction since $3SAT \in NP - hard$.

11.4.2 Minimum Vertex Cover

Definition 11.10 (Min Vertex Cover)

Given an undirected graph $G(V, E)$, a **vertex cover** is a subset $S \subset V$ where for all $(u, v) \in E$, either $u \in S$ or $v \in S$ (or both). The **minimum vertex cover** problem asks

1. Given G, find a vertex cover of minimum cardinality.
2. Given G and constant K, does there exist a vertex cover of size $\leq K$?

Theorem 11.6 (MVC)

MVC is NP-hard.

Proof.

Reduction from MIS. Recall that an independent set is a subset $S \subset V$ s.t. for all $(u, v) \in E$, $u \in S$ or $v \in S$ (but not both). By definition, we can see that $u \in S \iff u \notin V \setminus S$, and so IS and VCs are related in the following way:

$$\begin{aligned} S \subset V \text{ is a VC} &\iff \forall (u, v) \in E, u \in S \text{ or } v \in S \text{ or both} && (113) \\ &\iff \forall (u, v) \in E, u \notin S \setminus V \text{ or } v \notin S \setminus V \text{ or both} && (114) \\ &\iff V \setminus S \text{ is an IS} && (115) \end{aligned}$$

Therefore, if S has size K, then $V \setminus S$ will have size $N - K$. We assume that a polytime algorithm exists for MVC, so on input $G(V, E)$, we calculate the MVC, call it M, and then we return the corresponding MIS $S = V \setminus M$.
Equivalence.

1. $(\Rightarrow)$ Assuming we have the solution $|S|$ for MVC, we can return true iff $|V| - |S| \geq K$.
2. $(\Leftarrow)$ Assuming we have the solution $|S|$ for MIS, we can return true iff $|V| - |S| \leq K$.

But this would imply that there exists such a polytime algorithm, implying that $P = NP$, and therefore MVC must be NP-hard by definition.

Example 11.2 (Reduction from Independent Set to Vertex Cover)

Just take the complement.

11.4.3 Maximum Clique

Definition 11.11 (Max Clique)

Given an undirected graph $G(V, E)$, a **clique** is a subset of vertices $S \subset V$ s.t. there is an edge between each pair in S, i.e. a *complete subgraph*. The **maximum clique** problem asks

1. Given G, find the clique of maximum cardinality.
2. Given G and constant K, does there exist a vertex cover of size $\geq K$?

Theorem 11.7 (MC)

Finding the max clique is NP-hard.

Proof.

Reduction from MIS. Observe that
Let's define the **edge-complement** of a graph G as $\bar{G} = (V, \bar{E})$ where $\bar{E}$ is the set of edges not in G. Then, note that

$$\begin{aligned} S \subset V \text{ is an IS of } G &\iff \forall u, v \in S, (u, v) \notin E && (116) \\ &\iff \forall u, v \in S, (u, v) \in \bar{E} && (117) \\ &\iff S \text{ is a clique of } \bar{G} && (118) \end{aligned}$$

Therefore, we can reduce this from MIS by taking input G, getting $\bar{G}$ in $O(N + M)$ time, solving the MIS of $\bar{G}$, and returning the size of the max clique of $\bar{G}$ in our assumed polytime algorithm. Therefore, MC is NP-hard. In fact, we have already proved the equivalence at the same time due to the iff conditions, but we state it explicitly for completeness.
Equivalence.
1. ($\Rightarrow$) Assuming we have the MIS solution $S \subset G$, then S is the max clique of $\bar{G}$.
2. ($\Leftarrow$) Assuming we have the max clique solution $S \subset G$, then S is the MIS of $\bar{G}$.

Example 11.3 (Reduction of Independent Set to Clique)

Just take the edge complement.

11.4.4 Hamiltonian Paths

Definition 11.12 (Hamiltonian Path)

A **Hamiltonian path** in a directed graph G is a path that visits every vertex exactly once.

Theorem 11.8 ()

Finding the existence of a Hamiltonian cycle of a graph G is NP-hard.

Definition 11.13 (Hamiltonian Cycle)

A **Hamiltonian cycle** in a directed graph G is a cycle that visits every vertex exactly once.

Theorem 11.9 ()

Finding the existence of a Hamiltonian cycle of a graph G is NP-hard.

Proof.

Reduction from VC. Given an undirected graph $G(V, E)$ and integer K, we want to construct a directed graph $H(V', E')$, such that H has a Hamiltonian cycle iff G has a vertex cover of size K. We construct H in the following.
1. For each undirected edge $(u, v) \in E$, the directed graph H contains an *edge gadget* containing 4 vertices
$$(u, v, \text{in}), (u, v, \text{out}), (v, u, \text{in}), (v, u, \text{out}) \tag{119}$$

and 6 directed edges

$$(u,v,\text{in}) \to (u,v,\text{out}) \qquad (u,v,\text{in}) \to (v,u,\text{in}) \qquad (v,u,\text{in}) \to (u,v,\text{in})$$
$$(v,u,\text{in}) \to (v,u,\text{out}) \qquad (u,v,\text{out}) \to (v,u,\text{out}) \qquad (v,u,\text{out}) \to (u,v,\text{out})$$

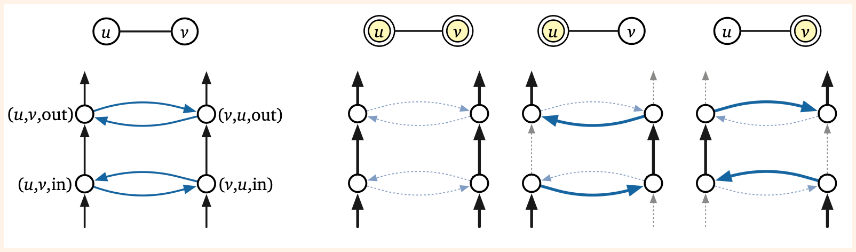


Figure 43: An edge gadget and its only possible interactions with a Hamiltonian cycle. Any Hamiltonian cycle in H must pass through an edge gadget in one of 3 ways: straight through on both sides, or with a detour from one side to the other and back. These options will correspond to both u and v, only u, or only v belonging in the vertex cover.

2. For each vertex $u \in V$, all the edge gadgets for incident edges (u, v) are connected in H into a single directed path, called a *vertex chain*. Specifically, suppose vertex u has d neighbors $v_1, \ldots, v_d$. Then H has $d-1$ additional edges $(u, v_i, \text{out}) \to (u, v_{i+1}, \text{in})$ for each $i = 1, \ldots, d-1$.
3. Finally, H also contains k *cover vertices* $x_1, \ldots, x_k$. Each cover vertex has a directed edge to the first vertex in the vertex chain, and a directed edge from the last vertex in each vertex chain.

Equivalence.

1. Suppose that $C = \{u_1, \ldots, u_k\}$ is a vertex cover of G of size k. We can construct a Hamiltonian cycle that encodes C as follows.
2. Suppose that H contains a Hamiltonian cycle C. This cycle must contain an edge from each cover vertex to the start of some vertex chain.

Example 11.4 (Reduction from Vertex Cover to Directed Hamiltonian Cycle)

We can see that each double arrowed blue segment represents a pair of directed edges, each part of the gadget.

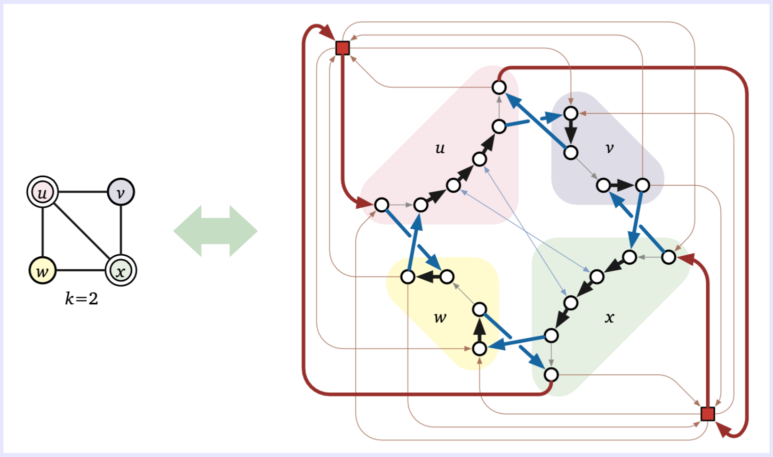


Figure 44: Reduction of a simple undirected graph to Hamiltonian cycle.

11.4.5 Euler Paths

11.4.6 Longest Path

Theorem 11.10 (Longest Path)

Finding the longest path in a graph is NP-complete.

11.4.7 Graph Coloring

Definition 11.14 (Proper K-coloring)

A **proper k-coloring** of a graph $G(V, E)$ is a function $C : V \to \{1, \dots k\}$ that assigns one of k colors to each vertex, so that every edge has two different colors at its endpoints.

Theorem 11.11 ()

Graph coloring is NP-hard.

Proof.

Reduction from 3SAT.

11.5 Subset Sum

Definition 11.15 (Subset Sum)

Given a multiset (allowing duplicates) X of positive integers and a target value T, we want to find whether there exists a subset $S \subset X$ that sum up to T.

Theorem 11.12 ()

Finding the subset sum is NP-hard.

Proof.

Reduction from MVC. Given an arbitrary graph G and integer K, we need to compute a set X of positive integers and integer T, such that X has a subset of sum T iff G has a vertex cover of size K. The general idea is that we label each edge with a number in $\{0, 1, 2, 3\}$, where 2 means that the edge is in the vertex cover.

1. We label each edge of G from $e = 0, 1, \dots, E - 1$.
2. Construct $X = \{\}$:
3. for each $e = 0, \dots, E - 1$, add $b_e = 4^e$ to X.
4. for each vertex v, add

$$a_v = 4^E + \sum_{e \in \Delta(v)} 4^e \tag{120}$$

   to X, where $\Delta(u)$ are the incident edges of u. What we have just done is represented each integer in X as an $(E + 1)$ digit number written in base 4. The Eth digit is 1 if the integer represents a vertex and 0 otherwise. For each $e < E$, the eth digit is 1 if the integer represents edge e or one of its endpoints, and 0 otherwise.

5. We set our target sum to be

$$T = K \cdot 4^E = \sum_{e=0}^{E-1} 2 \cdot 4^e \tag{121}$$

where the first term covers all vertices and the summation covers all edges.

Equivalence.

1. Suppose G has a vertex cover C of size k. Consider the subset

$$X' = \{a_v \mid v \in C\} \cup \{b_e \mid \text{edge } e \text{ has exactly 1 endpoint in } C\} \tag{122}$$

The sum of the elements in X', written in base 4, has most significant digit k and all other digits equal to 2. Thus the elements of X' sum exactly to T.

2. Suppose that there is a subset $X' \subset X$ that sums to T. We must have

$$\sum_{v \in V'} a_v + \sum_{e \in E'} b_e = T \tag{123}$$

for some subset $V' \subset V, E' \subset E$. Then each b_e contributes exactly one digit of 1 (in base 4). Each edge is incident to 2 vertices, so the non-significant digits contribute two digits of 1. Therefore, if we sum these base-4 numbers, there are no carries in the first non-sigf E digits since for each e there are only 3 numbers in X whose eth digit is 1. Each edge number b_e contribute only one 1 to the eth digit of the sum, but the eth digit of T is 2. Thus, for each edge in G, at least one of its endpoints must be in V'. In other words, V' is a vertex cover.

Example 11.5 (Reduction from Vertex Cover to Subset Sum)

Given the 4 vertex graph G with $V = \{u, v, w, x\}$ and $E = \{uv, uw, vw, vx, wx\}$, our set X will contain the following base-4 integers.

$a_u := 111000_4 = 1344$	$b_{uv} := 010000_4 = 256$
$a_v := 110110_4 = 1300$	$b_{uw} := 001000_4 = 64$
$a_w := 101101_4 = 1105$	$b_{vw} := 000100_4 = 16$
$a_x := 100011_4 = 1029$	$b_{vx} := 000010_4 = 4$
	$b_{wx} := 000001_4 = 1$

We can see that all the a's have most significant digit 1, meaning that it represents a vertex, and all the b's that represent an edge are simply stored with a single 1 and rest 0s. For the a's, the non-significant digits represent the incident edges. Vertex u has connections to v, w, so the next 2 bits, representing the 1 bits for edges uv, uw are flipped to 1. If we are looking for a vertex cover of size $K = 2$, we have

$$T = 222222_4 = 2730 \tag{124}$$

Indeed the vertex cover $\{v, w\}$ corresponds to the subset $\{a_v, a_w, b_{uv}, b_{uw}, b_{vx}, b_{wx}\}$, whose sum is $1300 + 1105 + 256 + 64 + 4 + 1 = 2730$.

12 Streaming Algorithms

Algorithm 12.1 (Track Median From Data Stream)

Algorithm 19 Track Median From Data Stream

```
Require: Stream of numbers                                  ▷Input is a continuous stream of numbers
Ensure: Current median after each number                    ▷Returns median as each number arrives
maxHeap ← empty max heap                                    ▷Stores lower half of numbers
minHeap ← empty min heap                                    ▷Stores upper half of numbers
function AddNumber(num)
    if maxHeap.isEmpty() OR num < maxHeap.peek() then       ▷Number belongs in lower half
        maxHeap.add(num)                                    ▷Add to max heap
    else
        minHeap.add(num)                                    ▷Add to min heap
    end if
    balance ← maxHeap.size() - minHeap.size()               ▷Check heap balance
    if balance > 1 then                                     ▷Max heap has too many elements
        minHeap.add(maxHeap.poll())                         ▷Move largest from max heap to min heap
    else if balance < -1 then                               ▷Min heap has too many elements
        maxHeap.add(minHeap.poll())                         ▷Move smallest from min heap to max heap
    end if
end function
function FindMedian
    if maxHeap.size() > minHeap.size() then                 ▷Odd number of elements
        return maxHeap.peek()                               ▷Return top of max heap
    else if minHeap.size() > maxHeap.size() then            ▷Odd number of elements
        return minHeap.peek()                               ▷Return top of min heap
    else                                                    ▷Even number of elements
        return (maxHeap.peek() + minHeap.peek()) / 2        ▷Average of both tops
    end if
end function
```

13 Linear Programming