6.006 – Lecture Notes

Sanjana Das

Spring 2022

Notes for the MIT class **6.006** (Introduction to Algorithms), taught by Sam Hopkins, Mauricio Karchmer, and Nir Shavit. All errors are my responsibility.

Contents

1 Algorithmic Thinking **5**
- 1.1 The Celebrity Problem . . . 5
- 1.2 Dutch National Flag Problem . . . 5

2 Efficient Algorithms **7**
- 2.1 Asymptotics . . . 7
- 2.2 Counting Time . . . 7
  - 2.2.1 Word-RAM Model of Computation . . . 7
- 2.3 Peak Finding . . . 8

3 Data Structures **9**
- 3.1 Sequences . . . 9
  - 3.1.1 Interfaces . . . 9
  - 3.1.2 Implementations . . . 9
  - 3.1.3 Memory Overhead . . . 10
- 3.2 Sets . . . 11
  - 3.2.1 Interface . . . 11

4 Sorting and Divide and Conquer **11**
- 4.1 Permutation Sort . . . 12
- 4.2 Insertion Sort . . . 12
- 4.3 Merge Sort . . . 12
- 4.4 More Divide and Conquer . . . 13

5 Implementing Set **14**
- 5.1 Direct Access Array . . . 15
- 5.2 Hash Table . . . 15
  - 5.2.1 Choosing a Hash Function . . . 16
  - 5.2.2 Analysis . . . 16

6 Linear Time Sorting **17**
- 6.1 Direct Access Array Sort . . . 18
- 6.2 Counting Sort . . . 18
- 6.3 Tuple Sort . . . 18
- 6.4 Radix Sort . . . 19

7 Binary Search Trees **20**
7.1 Some Operations . . . 20
7.1.1 Finding . . . 20
7.1.2 Traversal . . . 20
7.1.3 Growing BSTs . . . 21
7.2 BST as a Data Structure . . . 21
7.2.1 Set Operations . . . 21
7.2.2 Sequence Operations . . . 22
7.3 AVL Trees . . . 22
7.3.1 Importance of Balance . . . 22
7.3.2 Strategy for Balance . . . 22
7.3.3 Maintaining the Invariant . . . 23
7.3.4 Inserting . . . 23
7.3.5 Conclusion . . . 24
7.3.6 AVL Sort . . . 24

8 Priority Queues and Heaps **24**
8.1 Priority Queues . . . 24
8.2 Heaps . . . 25
8.2.1 Heap Operations . . . 25
8.2.2 Max-Heapify . . . 25
8.2.3 Build-Max-Heap . . . 26
8.2.4 Heap Sort . . . 26

9 Graphs **27**
9.1 Definitions . . . 27
9.1.1 Graphs as Models . . . 27
9.1.2 Computer Representations . . . 27
9.1.3 Basic Graph Notions . . . 28
9.2 Breadth-First Search . . . 28
9.2.1 Basic Graph Problems . . . 28
9.2.2 Algorithm . . . 28
9.2.3 Naive Implementation . . . 29
9.2.4 Improved Implementation . . . 29
9.2.5 BFS Tree . . . 30

10 Depth-First Search **30**
10.1 Complexity . . . 31
10.2 Directed DFS . . . 32
10.3 Uses of DFS Tree . . . 33

11 DFS Application **33**
11.1 Topological Sort . . . 33
11.2 Strongly Connected Components . . . 34

12 Dijkstra's Algorithm **36**
12.1 Weighted Graphs . . . 36
12.2 Structure of the SP Problem . . . 37
12.2.1 Optimal Subproblems . . . 37
12.2.2 Scaffolding for SP Algorithms . . . 37
12.2.3 The Triangle Inequality . . . 37

12.3 Relaxation Algorithms . . . 38
12.3.1 Relaxation . . . 38
12.3.2 Setup . . . 38
12.4 Dijkstra's Algorithm . . . 39
12.4.1 Correctness . . . 40
12.4.2 Runtime . . . 40

13 Bellman-Ford Algorithm **41**
13.1 Bad Relaxation Orders . . . 41
13.2 Path Relaxation Lemma . . . 41
13.3 Baby Bellman-Ford . . . 42
13.4 Negative Cycles . . . 43
13.5 Shortest paths on DAGs . . . 43

14 All-pairs Shortest Paths **44**
14.1 Nonnegative Weights . . . 44
14.2 Reweighting . . . 44
14.3 A Beautiful Symmetry . . . 44
14.4 Finding a Potential . . . 45
14.5 Overall Algorithm . . . 45
14.6 Other Algorithms . . . 46

15 Dynamic Programming **46**
15.1 The Row Coin Problem . . . 46
15.2 Memoization . . . 47
15.3 Pure DP . . . 48
15.4 Framework . . . 48
15.5 Use Cases . . . 49

16 Dynamic Programming II **49**
16.1 Longest Common Subsequence (LCS) . . . 50

17 Dynamic Programing and Shortest Paths **50**
17.1 SSSP on DAGs . . . 51
17.2 SSSP For General Graphs . . . 51
17.3 Floyd-Warshall Algorithm for APSP . . . 52
17.4 The Other Direction . . . 52

18 Greedy Algorithms **52**
18.1 Activity Selection Problem . . . 53
18.2 Activity Scheduling Problem . . . 54

19 Huffman Codes **55**
19.1 The Problem . . . 55
19.2 Huffman's Algorithm . . . 56
19.3 Runtime . . . 58

20 Subset Sum and Pseudo-Polynomial Time **58**
20.1 Sandwich Cutting . . . 58
20.2 Runtimes . . . 59
20.3 Subset Sums . . . 59
20.4 Pseudopolynomial Time . . . 60

20.5 More About Runtime 61

21 Complexity **61**
21.1 Decision Problems 62
21.2 Complexity Classes 62
21.3 Exponentially Large Search Spaces 63
21.4 Concrete Questions 65
21.5 The Punch Line 66

22 Computability **67**
22.1 Final Remarks 71

23 Parallel and Asynchronous Computation **71**
23.1 Problems 71
23.2 Parallel Primality Testing 72
23.3 Mutual Exclusion 73
23.4 Parallel Computing and Amdahl's Law 75

§1 Algorithmic Thinking

"If you cannot solve the proposed problem, try to solve first some related problem." – George Polya, *How To Solve It*

§1.1 The Celebrity Problem

Example 1.1

A *celebrity* among a group of n people who knows nobody but is known by everybody. You can ask questions of the form "do you know this person?" and you want to figure out whether there is a celebrity.

We'll solve a simple question first: if we know there is a celebrity, how can we find them?

Algorithm 1.2 (Simpler Celebrity Problem) — In the base case $n = 1$, the person is a celebrity.

In the recursive step, suppose S is a n-person group. Arbitrarily pick A and B from S, and ask if A knows B. If A does know B, then perform the same algorithm on $S - A$, since A cannot be a celebrity. If A doesn't know B, then perform the same algorithm on $S - B$, since B cannot be a celebrity.

We have $T_1 = 0$ and $T_n = T_{n-1} + 1$ for $n \geq 2$, so $T_n = n - 1$. So this is an $O(n)$ algorithm.

Definition 1.3. An algorithm is $O(n)$ if there exists $c \geq 0$ such that $T_n \leq cn$ for all sufficiently large n.

Now we can use this to solve the problem in the case where we don't know if there's a celebrity. The previous algorithm fails: for example, suppose we have a group of 2, where A and B both know each other. Then the algorithm should return there isn't a celebrity – but we ask whether A knows B, get the answer yes, and then we look at a group of one and return that B is a celebrity – because we fail to ask the question "does B know A?" So we can use this understanding to devise an algorithm for the new case. (Similarly, if neither knew each other it would fail and return A.)

Algorithm 1.4 (Harder Celebrity Problem) — Similarly to before, ask if A knows B. If A does know B then recurse without A – then we get either the name of a celebrity in $S - A$, or none.

If it returns B, then ask whether B knows A. If B knows A then he isn't a celebrity so there is none, while if B doesn't know A then he is a celebrity.

If it returns some C (other than B), then check *both* whether C knows A and A knows C (the answers should be no and yes, respectively).

Finally, if there's no celebrity in $S - A$, then there's no celebrity in S.

The other case, where we recurse on $S - B$, works symmetrically.

This has $T_n = T_{n-1} + 3$ for $n \geq 3$, so $T_n = 3n - 4$ for $n \geq 2$. This is still $O(n)$.

§1.2 Dutch National Flag Problem

Example 1.5

Suppose we have red, white, and blue pegs in an arbitrary order. We want to, using a minimum number of operations, move the pegs so that all the red pegs are to the left of the white pegs, which are all to the left of the blue pegs.

We'll solve a simpler problem first: the Polish national flag problem, where we only have red and white pegs, and want to sort using the minimum number of swaps.

Algorithm 1.6 — Search for the leftmost white peg and the rightmost red peg. If the white peg is left of the red peg, swap them; otherwise we're done.

In the worst case, where all the white pegs are left of all the red pegs (and there's an equal number of both), we swap the leftmost and rightmost pegs and reduce to $n-2$. So $T_n = T_{n-2} + 1$, which means $T_n = \frac{n}{2}$.

However, here we just counted swaps, but in the actual algorithm searching for the leftmost white and rightmost red peg takes time as well. Consider the situation where we have $\frac{n}{4}$ red, white, red, white pegs – then we can only get an $O(n)$ bound, so

$$T_n = T_{n-2} + O(n) + O(n) + 1 = T_{n-2} + O(n).$$

This means T_n is $O(n^2)$. An n^2 algorithm is not that great, so we want to improve on it.

Our problem here wasn't really the comparison, but the searching. So what we can do is actually add pointers – have a left pointer and a right pointer doing the search, and every swap, move the pointers forward. (So everything left of the left pointer is known to be red.) Move these pointers until they cross each other. Then we do an $O(n)$ number of swaps, and each pointer moves an $O(n)$ amount as well, so this entire algorithm is $O(n)$.

Remark 1.7. There's always a tradeoff between memory and computation. We replaced the computation of needing to search from the sides repeatedly, by keeping track of where we were before – so we've traded memory for time. This is common in computer science.

Now we return to the Dutch National Flag problem. This algorithm is by Dijkstra.

Algorithm 1.8 — Keep a left, right, and middle pointer. Start with the left and middle pointers on the left end, and the right pointer on the right end. (Move the left and mid until the left pointer is no longer red, and then the mid until the mid is no longer white; similarly move the right pointer until it's no longer blue.)

Then we swap (similar to the Polish flag) depending on the color of the middle pointer:

- If the middle pointer is red, swap the left and middle pointer's pegs, and move both the left and middle pointer one step right.
- If the middle pointer is white, then just advance it.
- If the middle pointer points to blue, then swap its peg with the peg at the right pointer, move the right pointer one left, but don't move the middle pointer.

This isn't *exactly* the Polish national flag, but it's similar.

All pointers move $O(n)$ distance, and there are at most $O(n)$ swaps, so this is an $O(n)$ algorithm. (The algorithm ends when the middle and right pointers cross over.)

In general, you want to try out *small cases* and small examples – start with a simpler thing to solve the general thing.

Here's another solution to the harder celebrity problem:

Algorithm 1.9 (Harder Celebrity Problem) **—** Run the algorithm that assumes a celebrity exists. It returns a person. Then check that person against all other people, in both directions. If all checks pass then this person is a celebrity, otherwise there is none.

§2 Efficient Algorithms

We want our algorithms to be correct on all inputs. (This isn't necessarily obvious – in some cases you'll want algorithms that are correct *most* of the time, in exchange for being more efficient).

For an algorithm to be efficient, it should be faster than other algorithms for the same problem. We'll compare algorithms and try to make ours as fast as possible.

Most of the time, runtime is our main resource. Memory is another resource – in that case efficient algorithms would use as little memory as possible.

§2.1 Asymptotics

We'll measure time *asymptotically* as a function of the input size.

This means we'll ignore constant factors, for example.

If the input is an array of numbers, the input size is the length of that array. But for a given input size there are many such arrays – we'll look at the *worst case*, the input that makes the algorithm run the slowest (because we want a guarantee that our algorithm is fast). Again this is not an obvious choice – we could look at the average case, for example.

We'll write that an algorithm runs in worst case $O(f(n))$. This is shorthand – we mean that there exists a constant $c > 0$ and input size n_0, such that for all $n \geq n_0$ and all inputs of size n, the algorithm runs in time at most $cf(n)$.

Notation 2.1. We use the following asymptotic notation:

- $g(n) \in O(f(n))$ means that there is a constant c such that $g(n) < cf(n)$ for all sufficiently large n.
- $g(n) \in \Omega(f(n))$ means that there is a constant c such that $g(n) > cf(n)$ for all sufficiently large n.
- $g(n) \in \Theta(f(n))$ means it is in both $O(f(n))$ and $\Omega(f(n))$.

If we can get a Θ bound then our analysis is as tight as possible. We can prove the lower bound Ω by showing that some input is bad. To prove an upper bound you'd have to show that it works within that time for all input.

§2.2 Counting Time

We count time in the number of steps that the computer makes.

§2.2.1 Word-RAM Model of Computation

In this model, we have **machine words** of size w – the computer manipulates words, where each word has w bits. Each word can store integers between 0 and $2^w - 1$, for example, or characters in an alphabet of size at most 2^w. (You can think of w as 64, for current machines.)

Then we have a **processor**, which can operate on words in $O(1)$ time. This means you can add, multiply, subtract, divide, compare, bitwise shift, and so on.

Then we have **memory**, where we have a big array M. We allow for **indirect addressing** – if we have a word i, we can access memory location $M[i]$. So we can store pointers in words and look at the memory location pointed at by our pointer. This gives the upper bound $|M| \leq 2^w - 1$.

We also assume that the input at the beginning is in $M[0], \ldots, M[n-1]$ – it starts at hte beginning of the memory. This means $n \leq 2^w - 1$. So we're assuming $w \geq \log n$. (Since 2^{64} is a very big number, this usually isn't a problem.)

There is a bit of duality because w is fixed, n is upper bounded by a function of w, and we're taking n to infinity. We'll have to live with that inconsistency (imagine w going to ∞ along with n).

> **Remark 2.2.** The RAM model can be thought of as Python without lists, dictionaries, strings. If you want one then you'd have to implement them from scratch.

§2.3 Peak Finding

Suppose we have an array of numbers, which we can think of as representing heights. A **peak** is an index which is no smaller than its neighors. We want to find a peak.

First, every array has at least one peak, because the maximal element must be a peak.

> **Algorithm 2.3 —** Check every index to see whether it is a peak.

```
for i = 0 to n - 1
    if i is a peak
        return i
```

Analysis. This algorithm is obviously correct. The outer loop runs n times, and checking whether an index is a peak takes constant time, so this algorithm takes $O(n)$ time.

Meanwhile we can lower bound at $\Omega(n)$ as well – take the array to be strictly increasing. Then the only peak is at the end, so the algorithm has to run through every index.

So then this algorithm is $\Theta(n)$. □

But we can do better. Imagine we pick some index and it's not a peak. Intuitively, if you want a peak then you should climb uphill (since a peak is a local max). So then we should go uphill.

> **Algorithm 2.4 —** Do binary search – start at the midpoint of the array. If it's a peak then we're done. Otherwise, one neighbor is larger than it; then go in the direction of that larger neighbor, and disregard everything on the other side.

```
if n = 1 return 0
m = n/2
if m is a peak return m
else if A[j - 1] > A[j] recurse left
else recurse right
```

Analysis. To prove correctness, use strong induction. This clearly works for the base case $n = 1$. Now assume it works for all $m < n$ – so we can assume that the recursive calls produce a peak k of the sub-array.

So we want to show that if k is a peak of the sub-array, then it's a peak of the actual array. If k is in the middle of the sub-array – meaning that it's not next to the midpoint – then it's already been compared to both its neighbors in the sub-array, so k is a peak of the whole array.

Meanwhile, if k is next to the midpoint, then again it's been compared to its other neighbor by the recursive call, while when we did the recursion we guaranteed that k is greater than the midpont. So this works.

This takes $O(\log n)$ time because it splits the array in half each time, and $\Omega(\log n)$ time by again taking a strictly increasing array – then the only peak is the endpoint, so none of the midpoints will be a peak and the algorithm will only end when the search space goes from n to 1. We can prove this via recurrence relations: if $T(n)$ is the worst-case time, then we can write

$$T(n) \leq T(\frac{n}{2}) + O(1).$$

This has the answer $T(n) = O(\log n)$.

So therefore our algorithm is $\Theta(\log n)$, which is much better than our linear-time algorithm. □

§3 Data Structures

Data structures use algorithms, and are at the core of many algorithms.

> **Definition 3.1.** A **data structure** is an object we use to store data, together with a collection of operations that allow us to access and manipulate the data.

A data structure can be specified using a **interface** (or API, or Abstract Data Type), which is a specification of the operations of the data structure.

The **implementation** is the actual code and design that implements the operations from the interface. We can have one interface with many implementations.

§3.1 Sequences

A sequence maintains a sequence of elements $x_0, x_1, x_2, \ldots, x_{n-1}$. For example, Python lists.

§3.1.1 Interfaces

The interface has container methods – commands `build(A)`, which, given an iterable, builds a sequence from its items); and `len(A)`, which gives its number of items.

We also have static methods – `iter_seq()`, which returns the stored items one-by-one; `get_at(i)`, which returns the ith item; and `set_at(i, x)`, which replaces the ith element with x.

We also have the dynamic mehods – `insert_at(i, x)`, which adds x as the ith item (and therefore shifts everything after it); `delete_at(i)`, which removes and returns the ith item; and `insert_first(x)` and `delete_first()`, and `insert_last(x)` and `delete_last()`.

There are a few special cases of sequences. The **stack** implements a last-in, first-out ordering, and has two operations – *push*, which is `insert_last(x)`, and *pop*, which is `delete_last()`. The **queue** is a first-in, first-out ordering, and has two operations – *queue*, which is `insert_last(x)`, and *dequeue*, which is `delete_first()`.

§3.1.2 Implementations

> **Definition 3.2.** A **linked list** is an implementation of a sequence where you store an element and a pointer to the next element (as well as a head pointing to the beginning, and its length).

The `build(A)` operation takes $O(n)$ time. The static `get_at(i)` and `set_at(i, x)` operations take $O(n)$ time as well, since you have to walk to the point where the ith element is. The dynamic `insert_first(x)` and `delete_first()` take $O(1)$ time, since all you have to do is change the pointers at the beginning (and fix the length); meanwhile `insert_last(x)` and `delete_last()`, as well as `insert_at(i, x)` and `delete_at(i)`, take $O(n)$ time because we have to walk to that position – you have to position your pointer to the place where you do the surgery, and moving to that position takes $O(n)$ time, although the surgery itself takes constant time.

We could potentially keep a tail, and have pointers in *both* directions – a **double linked list**, where we have a pointer to the next element *and* the previous element (and a head and a tail). Then inserting or deleting at the end becomes constant time as well, although inserting and deleting at position i remains $O(n)$.

> **Definition 3.3.** A **array** is a contiguous part of memory (also storing its length).

Again `build(A)` takes $O(n)$ time (since you allocate memory and copy all elements into your array). Now `get_at(i)` and `set_at(i, x)` take constant time, since you can just get to that location in memory in a constant time. Meanwhile `insert_first(x)` and `delete_first()` take $O(n)$ time because we'd have to shift every element one to the right.

With `insert_last` and `delete_last`, if we have space then it's easy because we know n and therefore know where to add. In the case where we don't have space, we'd have to allocate a new array of larger size and move everything. This is something that seems like it should be easy but isn't because of the case where the array is full, which makes this $O(n)$ – we can use a **dynamic array** to fix this.

In **amortized analysis**, when we're doing worst-case analysis looking at data structures within a given operation, we care about the sequence of operations. For example, even though allocation into a larger array is expensive, the idea is to just not do it too often – if in between expensive operations you have many cheap ones, then the total span is not that bad.

> **Definition 3.4.** An operation has **amortized runtime** $T(n)$ if any sequence of k operations takes at most $kT(n)$ time.

So this is the worst case over all collections of k operations. We don't need every single one of these operations to take $T(n)$ time, only that this is the aggregate (so that on average, the operations take $T(n)$ time). So if one of them is expensive but the others are cheap, then the expensive one gets amortized over the cheap one.

In a **dynamic array**, since growing an array is expensive, we grow it smartly. Every time the array of size n gets full, we grow it by doubling its size, so that we can insert many elements before we're forced to do another reallocation. This is called **array doubling** – if an array of size n is full, then reallocate to an array of size $2n$. The doubling gets amortized, and then inserting and deleting at the end becomes $O(1)$.

More formally, assume we have $k = 2^m$ operations, and we want to count how much time we spend inserting k elements ito an array. We have doublings after 2, 4, 8, $\ldots$, 2^{m-1} elements. Each doubling takes lilnear time $\Theta(2^i)$. So then their sum is $c\sum 2^i = c \cdot 2^m = O(k)$. So to insert k operations, we spend an $O(k)$ amount of time expanding the array. This means that (for large k), one insert is $O(1)$ using amortized runtime.

§3.1.3 Memory Overhead

> **Definition 3.5. Memory overhead** is the amount of memory you use not included in the memory used for storing the data itself.

For example, the overhead for a linked list is $O(n)$, since we have to keep pointers. For a dynamic array it's still $k = O(n)$ – since if we have n elements, we have at most n extra slots.

If we do n inserts and then n deletes, we end up with no elements in the list, but we have an array of size n. This means infinite overhead, since we have n elements used with no elements stored. So then we have to release some of the memory that we're not using. We use **array halving** for that. Whenever we have less than $\frac{n}{4}$ elements stored (where n is the starting amount),we reduce to an array of size $\frac{n}{2}$ – we're deleting extra cells, while still preserving a bunch of empty space in order to be inexpensive via amortized cells.

Then any sequence of k inserts or deletes can be done in $O(k)$ time.

If you know in advance how many elements you'll store, it may be better to use a static array; if you don't, then you may want to use a dynamic array.

Dynamic arrays also deal with the problem of inserting and deleting first by slightly modifying so that you can insert or delete from the beginning.

§3.2 Sets

§3.2.1 Interface

A set contains elements $x_1, \ldots, x_n$, each of which hjas a key. The keys are unique (and are usually integers – for example, you could imagine storing a set of students, with keys their IDs).

The set interface has the container methods – `build(A)`, which builds from an iterable, and `len()`. It has the static method `find(k)`, which finds the object with key `k`.

It has the dynamic methods `insert(x)`, which inserts an element `x` into the set (if its key is not already present), and `delete(k)`, which deletes the element with key `k`.

If the keys come from an ordered set, this induces an ordering on the elements. So you can ask the set to spit out an iterable of the set in sorted order, you can find the minimum and maximum element, and you can find the first element before or after a given key `k`.

Sets are an extremely important data structure, and for the next few weeks we're going to build progressively better implementations of sets.

§4 Sorting and Divide and Conquer

To build good data structures, we need efficient algorithms; and we want fast data structures in an algorithm. You can use sorting to build a nontrivial implementation of a set structure that has a fast find operation.

> **Question 4.1.** Suppose we are given an array $A = [a_1, a_2, \ldots, a_n]$ of integers (or anything comparable). The goal is to output $B = [b_1, b_2, \ldots, b_n]$ which is equal to A in sorted order.

The elements don't have to be integers; for example, you can have tuples of a number and name, with comparison operation to compare the numbers.

> **Definition 4.2.** A sorting algorithm is called **in place** if it uses $O(1)$ additional memory.

The algorithm needs n memory to receive the input, but we want to see how much extra memory it has to allocate outside of the input.

> **Definition 4.3.** A sorting algorithm is called **stable** if it preserves the order of equivalent elements.

For example, if we have the elements `(15, Sam)` and `(15, Nir)` in the array (which are equal in the comparison), a stable algorithm should preserve their original order when it sorts.

There are many cases where we might want to sort.

§4.1 Permutation Sort

Suppose you try every permutation of the input array, and for each permutation, check if it is sorted.

This is a very inefficient algorithm – it takes $\Omega(n!)$ time.

§4.2 Insertion Sort

The idea of insertion sort is that after the ith round, we want the array from 0 to $i-1$ to be sorted.

```
    for i = 0 to len(A) - 1:
        insert(A, i)
    return A
```

We want `insert` to insert the ith element into its correct position in the first i: suppose that up to index $i-1$, the array is sorted. Then `insert(A, i)` will stick the ith element into wherever it goes in that sorted prefix, so that we get a sorted prefix of length $i+1$.

There are a ton of ways to do this. One way is to swap it with adjacent elements until it's greater than the element to its left.

Example 4.4

Suppose the input is 3, 5, 1.

The length-1 prefix is already sorted. Then we try to insert 5, which is already in place, so nothing happens. Then we try to insert 1. We swap with 5 to get 3, 1, 5, and then we swap with 3 to get 1, 3, 5.

In an algorithm we care about correctness and running time. You can prove correctness by induction on i (that you get a sorted prefix of length i).

The running time is $O(n^2)$ – the insert operation takes $O(i)$ time, since in the worst case you have to push the ith element down to the start. So then the algorithm is $O(1+2+\cdots+n) = O(n^2)$.

The worst-case input here is when the array is entirely in reverse order. Then every insert operation must take the full i turns, so equality holds.

So then the algorithm is $\Theta(n^2)$.

This is a **incremental algorithm**; selection sort is as well.

§4.3 Merge Sort

Cutting things in half is often a very good idea (we saw this with peak finding) – *divide and conquer*.

First we slice the array in half (taking a left half and a right half). Then we want to recursively sort both the halves. Now given the solutions to the subproblems, we want to put them together into a solution to the overall problem.

So we want an operation to merge two sorted arrays – `merge(B, C)`. The first element of the new array should be either the first element of B or C. Then the next element should be the new least element from either B or C, and so on. So you start with a pointer at the beginnings of B and C, and every step you take the element with the smaller pointer, stick it in the array, and move that pointer.

Example 4.5

Merge the arrays $[1, 3, 8]$ and $[2, 4, 7]$.

At the start, we have our pointers at 1 and 2. Then we fill in 1 and advance that pointer. Then $2 < 3$ so we fill in 2 and advance that pointer. Then $3 < 4$ so we fill in 3, then $4 < 8$ so we fill in 4, then $7 < 8$ so we fill in 7, then we fill in 8.

Merge takes length $O(n)$, where n is the total number of elements – since you only look at each element once.

```
mergesort(A):
    B = mergesort(A[0:n/2])
    C = mergesort(A[n/2:n])
    return merge(B, C)
```

The algorithm is correct by induction. Meanwhile, for time, we have

$$T(n) = 2T(n/2) + O(n).$$

We can draw a recursion tree and we will see that every level has $O(n)$ contribution (because there are 2^k nodes contributing $\frac{n}{2^k}$ each), and there are $O(\log n)$ nodes, so then the total work done is $O(n \log n)$.

In real life $\log n$ is small so this is an extremely fast algorithm.

The thing about mergesort is that it builds the whole recursion tree without looking at the input, so it runs in $n \log n$ time for *any* array. So it's also $\Omega(n \log n)$, which means it's $\Theta(n \log n)$.

§4.4 More Divide and Conquer

In 1-dimensional peak finding, we used the "go uphill" algorithm – you take the middle, and recurse in the direction that's uphill.

Example 4.6 (2D Peak Finding)

Suppose we have a 2-dimensional array with dimensions $n \times m$. A **peak** in this array is an element that is greater than or equal to all its orthogonal neighbors (north, south, east, and west). Find a peak.

A peak still exists (the maximum of the array is a peak).

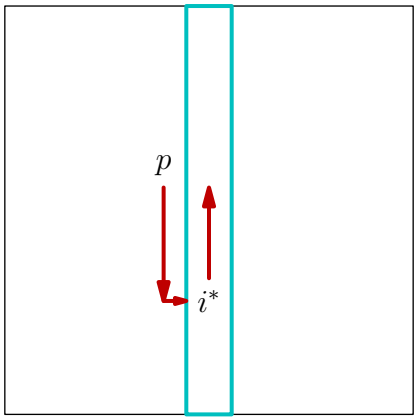
Algorithm 4.7 — Pick the middle column. Find the maximum element of that column; call that index i^*. If $A[i^*, m/2]$ is a peak then we're done. Otherwise, we look left and right, and go in the uphill direction (and call the peak-finding algorithm on that half of the array) – call that the uphill subarray, so we recursively return a peak on the uphill subarray.

Proof of correctness. Induct; the base case is when there is only one column, in which case we're done because the max is a peak.

Now use strong induction. If our element $A[i^*, m/2]$ is a peak we're done. Otherwise by the inductive hypothesis, we can assume that we correctly got a peak of the subarray.

If that peak in the subarray is not adjacent to our middle column, then it must be a peak of the actual array as well.

Meanwhile, if it *is* adjacent to the middle column, the only way that this fails is if the element in the middle column next to it is strictly greater than it.



But note that the algorithm is always returning a peak which is the maximum of the column. So then our peak is the maximum of that adjacent column. But this is at least the element next to $A[i^*, m/2]$ in that column which is uphill, *that* is at least $A[i^*, m/2]$, and *that* is at least any element in its column. So then the maximum of that adjacent column is at least as great as the element next to it in our column, and this failure doesn't happen. □

Proof of runtime. Finding a max takes $O(n)$ time, so then this is

$$T(n, m) = T(n, m/2) + O(n) = O(n \log m).$$

□

§5 Implementing Set

In a sorted array implementation of set, we can do `find` in $O(\log n)$, and `insert` and `delete` in linear time. Today we'll look at whether we can beat $O(\log n)$ for `find`. The answer is both no and yes.

Definition 5.1. In the **Comparison Model**, a set stores $x_1, \ldots, x_n$ with keys, and the only way that the implementation can access the keys is through comparisons.

The sorted array implementation fits in this model, since we just binary search.

Theorem 5.2

`find(k)` takes $\Omega(\log n)$ time in the comparison model.

Proof. Introduce a *decision tree*, which is a tree that represents the behaviour of the `find` operation. Represent the first query as the first node, where we ask "is $a < b$?" That query comes back either true or false, giving two possible routes. Then we have another comparison query at each child, and so on. Eventually the algorithm returns an answer – either one of the elements, or none. Represent outputs as the leaf nodes of the tree.

The running time is at least the number of comparisons, which is at least the length of the root to leaf path. So the worst-case running time is at least the *height* h of the decision tree.

But the decision tree has to have a leaf for every possible element in the set (and one for `None`). So there are at least $n + 1$ leaves.

But we double the number of nodes per layer, so then we need $\Omega(\log n)$ levels. □

Remark 5.3. This proof makes sense if we only have a true-false comparison algorithm. In real life you can make greater-than and equal-to queries. But each time, as long as there's a constant number of query types, with a constant number of possible answers, then the proof still holds – you get a number of outcomes that's at most exponential in the number of queries.

So if we're going to beat $\log n$, then we need an operation that has a lot more outcomes.

§5.1 Direct Access Array

The idea is that there *is* an operation that has more than a finite number of possible outputs, and that operation is random memory access. We can take a memory address and look at what's in that memory address in constant time, and there's a ton of memory addresses.

Imagine that the keys to the elements x_1, x_2, ..., x_n are integers from 0 to $U-1$. Then allocate a big array of size U, and when you want to insert x, look at its key and *treat that as a memory address* – to do `insert(x)`, we let `A[x.key] = x`.

Assuming that all the lookups are constant time operations (meaning everything fits into one word), `insert` is $O(1)$ time. Similarly `delete` is also $O(1)$ time, since we just go to that memory address and delete what's there. Similarly `find` is also $O(1)$, since if x has key k, you just go to memory address k and see what's there.

Question 5.4. What are some drawbacks?

A drawback is that if U is really big, then this array is really big – probably much bigger than the number of things we need to store.

Suppose we want to store keys which are names. Then there's a *ton* of possible keys (if a name is at most 30 characters, there are $26^30 \approx 10^{42}$ different names).

The operations `min` and `max`, as well as `next` and `previous`, are linear in U – because you start somewhere and start scanning, and you may have to scan through a huge number of stuff in order to reach an element that actually exists.

So we want to fix this. The fix is *hashing*.

§5.2 Hash Table

We saw the problem that we don't want to use so much memory, so the fix is... to use less memory.

Instead of allocating an array of size equal to the number of keys, allocate an array A of size m, where $m = \Theta(n)$.

Definition 5.5. A **hash function** is a function $h : [U-1] \to [m-1]$, which maps the universe of keys to the universe of indices.

So we have a hash function. Then when we do `insert`, we store x at the index where we hash its key – we do `A[h(x.key)] = x`. Similarly we can do insertion and deletion easily.

The issue is that multiple keys will map to the same hash value, since the universe of keys is much bigger than the universe of indices. So there's the potential of **hash collisions** – when we try to store x and y with `h(x.key) = h(y.key)`.

A big part of hash tables is designing an intelligent way to deal with hash collisions. We'll use an approach called **chaining**.

Algorithm 5.6 — Suppose that we have inserted x at some point, and we're trying to insert y at the same point. Then instead of inserting either there, have a pointer from that index of the hash table pointing to a linked list (or any other reasonable set implementation), where we put x and then y.

Then when we insert, we need to go to the array at index given by the hashed value of the key, and call insert on the auxiliary structure stored there. The same is true for delete and find.

Remark 5.7. Another way to deal with hash collisions is called **open addressing**. Chaining will work out well for us in theory, but it turns out that allocating these extra lists is kind of slow (it's a constant, but a big constant). So the idea is to store collisions in another place in the hash table, with rules for where you go when you have a collision.

Now for running times:

- For `insert`, we go to the hashed key, and then insert into the chain stored there. This takes time $O(c)$, where c is the length of the chain stored at that key. The same is true for `delete` and `find`.
- For `min` and `max`, as well as `next` and `prev`, these are $O(m)$, which by our choice of m is still $O(n)$.

§5.2.1 Choosing a Hash Function

We see that collisions aren't just bad for correctness, they're bad for running time as well (because the relevant operations are linear in chain size). So a good hash function should minimize collisions. There's also a hidden assumption here, that the hash function can be computed in constant time – so this should also be true.

One possible function is taking mod m: $h(k) = k \mod m$.

Consider what happens when m is a power of 2, and we think of k as a string of bits. Then this extracts the low-order bits of k. This gives you a bad hash function when you're trying to hash html tags, for example, where the last characters are all brackets so the lower order bits are all the same.

So we see that if we have a modulo hash function, we want m to not be a power of 2, or something similar – in fact it would be good to take m prime.

§5.2.2 Analysis

The worst possible input is one that generates a bunch of collisions – colliding every element. Then we get one gigantic chain and everything is back to linear time. This is bad. So we need a better analytical framework than worst case analysis (since that will almost never happen, as long as your function is unrelated to the things you're hashing).

So we'll use a different assumption:

Definition 5.8. The **Simple Uniform Hashing Assumption** is the assumption that the keys are random enough to have the following property: that $h(k)$ is distributed uniformly in $\{0, \ldots, m-1\}$, and is independent of all other $h(k')$.

Remark 5.9. Intuitively, this captures the assumption that the hash function and keys are pretty unrelated. In 6.046 we'll see a more sophisticated analysis that avoids the random keys assumption – you can bake the randomness into the hash function itself.

Fact 5.10 — Even with this assumption, if we insert $O(\sqrt{m})$ elements we're likely to see a collision.

Now for `find`, suppose we've already stored $x_1, \ldots, x_n$. Then the time for `find` is O of the length of the chain at the hashed key. We're doing *average case analysis*, so we want to find the *expected length of the chain*.

There are m possible hashes, and the hashes are uniformly random. So then the average case runtime is

$$\mathbb{E}(\text{length of chain}) = \frac{1}{m}\sum_{i=0}^{m-1} \text{length of chain at } \texttt{A[i]} = \frac{n}{m},$$

since we've stored n items. This is called the **load factor**.

If the load factor is tiny there's some other operations we have to do – so then the runtime is $O(1+\alpha)$. If α gets very large, then we can rebuild to make m larger – similarly to the dynamic array from earlier. The same amortized analysis from there still holds – so then the *averaged and amortized* runtime for `find`, `insert`, and `delete` are all $O(1)$.

Previously we had $\log n$ `find` and linear `insert` and `delete` – now we've made these constant, at the cost of screwing up the order operations (`next` and `prev` are now linear).

§6 Linear Time Sorting

Now that we've seen how to break the $O(\log n)$ search on a set by using direct access, we'll try to do something somewhat similar for sorting.

We've seen Insertion Sort, which is $\Theta(n^2)$, and Merge Sort, which is $\Theta(n \log n)$. These fit into the *Comparison Model* from last class – that you can only look at the input by asking questions with true-false answers (by asking $\leq$, $\geq$, and $=$).

Theorem 6.1
Sorting takes $\Omega(n \log n)$ time in the comparison model.

Proof. Make a similar decision tree as before. Then there are $n!$ possible outputs – each of the permutations of n numbers – so if the tree has height h, then we must have

$$2^h \geq n! \implies h \geq \log_2 n! = \log_2 1 + \log_2 2 + \cdots + \log_2 n.$$

You can bound this expression as $\Theta(\log n)$ – for example, this expression is at least

$$\frac{n}{2}\log_2\left(\frac{n}{2}\right) = \frac{n(\log_2 n - 1)}{2} = \Theta(n \log n).$$

(You can also use the Stirling approximation $n! \approx (\frac{n}{e})^n$, which gives the same bound.) □

So this tells us we cannot do better in the Comparison Model than Merge Sort. Now we will use direct memory access to improve on the sort algorithm as well.

§6.1 Direct Access Array Sort

This is the analog to a direct access array in sets.

Assume that the input $x_1, \ldots, x_n$ each have integer keys from 0 to $U-1$. Also assume that the keys are unique. (We will work on getting rid of these assumptions later.)

> **Algorithm 6.2 —** First, allocate a giant array of size U, with indices 0 to $U-1$. Then scan through the input, and when you read x_i, treat the key as a memory address, and stick x_i at the location indicated by its key. Then scan from left to right and read out the x_i.

This is $\Theta(U)$ runtime (because it takes $\Theta(n)$ to scan the x_i, $\Theta(U)$ to scan the big array, and $U \geq n$).

If $U = O(n)$ then this actually beats Merge Sort. But if U is large, then this is very slow.

Now we'd like to delete the assumptions.

§6.2 Counting Sort

Maintain assumption 1 (that keys are from 0 to $U-1$) and drop assumption 2.

> **Algorithm 6.3 —** As before, allocate an array of size U. As we scan through the inputs, store them at the indices indicated by their keys. If x_i and x_j have the same key, then set up a chain at that key's slot – this chain should be a **queue**. Then scan through and read everything out (by repeatedly dequeuing until the queue is empty).

> **Remark 6.4.** This algorithm is stable – items with the same key maintain their order because queues are first-in, first-out. If we used a stack instead (first-in, last-out) then we would have reversed equivalent elements.

This takes the same time $O(n+U)$. But we've done something nontrivial.

Now we'll try to do something about the fact that this running time is gigantic if U is large.

§6.3 Tuple Sort

Here's yet another sorting algorithm, that's more of an outside algorithm that requires you to plug in another sorting algorithm on the inside.

Here, assume that the keys are actually tuples of other keys: `x.key = (x.k1, x.k2, ..., x.kc)`. Suppose the little keys come from an ordered set (so they have an ordering on them), and you have some sorting algorithm for the little keys. Assume that this sorting algorithm is *stable*.

Then we want to sort by the big keys, where their ordering is lexicographic. (For example, $(1,5), (2,1), (2,3)$ is in sorted order.)

The idea is to use our auxiliary sorting algorithm to sort according to each of the digits in turn.

> **Algorithm 6.5 —** Sort the elements by each key, starting from the *last* (least significant) key and moving upwards.

```
for i = 0 to c - 1:
    sort by k(c - i)
```

Example 6.6

Suppose our input is 32, 03, 44, 42, 22.

Solution. We first sort according to the least significant digit. Because of stability, we end up with

$$32, 42, 22, 03, 44.$$

Then we sort by the next digit, and because of stability we end up with

$$03, 22, 32, 42, 44.$$

So this worked. □

We can see here why stability matters – because we've already sorted by least-significant digit to get 42 before 44, and we want to maintain this when we sort by first digit.

So stability *remembers* the previous rounds of sorting.

Remark 6.7. The reason you start with the least significant digit is because your previous sorts get overwritten every time you do a new one – unless the elements are equal, in which case they don't. So you want to first sort by the least significant, so that gets overwritten unless all the remaining digits are equal (in which case this order is maintained), and so on.

We want to find the runtime. All we're doing is calling the other sorting algorithm c times, so this is $O(cT)$, where T is the runtime of the auxiliary sorting algorithm.

§6.4 Radix Sort

Now we have Counting Sort, which is $O(n+U)$ and is stable, and we have Tuple Sort, which runs in $O(cT)$ time if the keys are $k = (k_1, \ldots, k_c)$. We'd like to combine these.

We again want to sort $x_1, \ldots, x_n$ assuming that the keys are numbers 0 to $U-1$. We have an algorithm that works well when keys are small, so we can expand our keys in some base. We'll use base n – write

$$k = k_0 n^c + k_1 n^{c-1} + \cdots + k_c.$$

Here we have $c = \log_n U$.

Now the order on $\{0, \ldots, U-1\}$ is lexicographic order on the tuples $(k_0, k_1, \ldots, k_c)$.

Algorithm 6.8 — Now run Tuple Sort, with Counting Sort as the auxiliary sorting algorithm.

So then the runtime is $\log_n U \cdot O(n)$, since when we did Counting Sort we had $U = O(n)$. So the time that Radix Sort takes is $O(n \log_n U)$.

Remark 6.9. This is linear time under the condition that $\log_n U$ is a constant – if $U < n^{O(1)}$. So if our keys are polynomially bounded then this is linear time.

§7 Binary Search Trees

Hash tables are good ifi you want to find a key exactly, but not for finding the next or previous key.

In a binary tree, every node should have a left child (smaller than it) and a right child (larger than it). We want to build a tree where all vertices descended from the left child are less than this vertex, and all vertices descended from the right child are larger.

> **Definition 7.1.** The **ancestors** of a node are all nodes above it, and **descendants** are all nodes below it. The **root** has no ancestors, and **leaves** have no descendants.

An actual tree is organized with a root at the bottom, but we draw trees with the root at the top.

You can implement this as a bunch of nodes, each with a key, a left pointer, and a right pointer.

This is a variation of a linked list. You can look at paths in the tree which are essentially linked lists.

§7.1 Some Operations

§7.1.1 Finding

Binary search trees are very good at searching recursively.

```
    def search_recursively(key, node):
2   if node == none or node.key == key:
        return node
4   if key < node.key:
        return search_recursively(key, node.leftChild)
6   if key > node.key:
        return search_recursively(key, node.rightChild);
```

§7.1.2 Traversal

We can also traverse the tree.

```
1   def traverse_binary_tree(node):
        if node == none:
3           return
    traverse_binary_tree(node.leftChild)
5   print(node.value)
    traverse_binary_tree(ndoe.rightChild)
```

This prints out the tree in left-to-right order (it prints the left child, then the vertex, then the right child).

If the binary tree is sorted correctly – so that for every vertex, all vertices in the left subtree are less than that vertex, and all vertices in the right subtree are greater – then this prints in increasing order.

This is called a **in-order traversal**. We're traversing the tree in ascending order of the keys, similarly to what would happen if using a linked list.

The beautiful thing is that we can do this all in recursion.

§7.1.3 Growing BSTs

When we build a tree, we start with the root 10. Then we want to insert 12. We have $12 > 10$, so we add it on the right. If we want to insert 4, then $4 < 10$, so we add it on the left. If we want to insert 1, then $1 < 10$ and $1 < 4$, so we add it as the left child of 4. If we want to insert 8, then $8 < 10$ but $8 > 4$, so we insert it as the right child of 4. If we want to insert 9, then $9 < 10$, $9 > 4$, and $9 > 8$, so we insert it as the right subtree of 8.

We can essentially keep on doing this.

The complexity of inserting and deleting is related to the **height** – the distance from the leaves to the root. For example, the tree we described earlier has height 3.

Our goal is to make every operation $O(\log n)$. Today we'll show that we can make all operations $O(h)$ (and next lecture we'll see how to keep $h = O(\log n)$ in order to achieve this).

§7.2 BST as a Data Structure

§7.2.1 Set Operations

We have the operations `insert` and `find`, which we already saw. These are $\Theta(h)$.

We also want an operation `findmin`. This is the leftmost element. So you just have to keep on going down to the left child until there is no left child, and this gives a min. This is $\Theta(h)$.

We can perform operation `deletemin` similarly. This is also $\Theta(h)$.

We can also have the operation `successor`, which finds the node with the next key. This is essentially the next element in the in-order traversal – it's the analog to `find_next` in a linked list.

In order to be able to do this, we'll modify the implementation of the tree so that we have links that point to the parents as well (so we have arrows in both directions).

If x is 10 in our tree, then its successor is 12. But if it's 9, then we have to go all the way back up to 10.

Case 1 (x has a right child). Then the successor is the smallest descendant of the right child. So we can just return `find_min(x.right)`.

Case 2 (x has no right child). Then we need to go back up until we take a step right.

If x has no parent, then it doesn't have a successor. If x is a left child, then its successor is its parent – because the parent is the next thing we visit in the in-order traversal.

If x is a right child, then if its parent is a left child of some node, that ancestor is the successor. This keeps on happening.

So you just keep walking upwards from this node until you take a step right, at which point that ancestor is your successor. (If you never take a step right, then you are the maximal element.)

```
    if x.right is not None:
2       return find_min(x.right)
    y = x.parent
4   while (y is not None and x = y.right):
        x = y
6       y = y.parent
    return y
```

This is also $\Theta(h)$. Similarly `predecessor` is symmetric.

So we've shown that we can implement all the set operations just traversing down the tree.

§7.2.2 Sequence Operations

We can actually implement sequence properties in $\Theta(h)$ time as well.

The way we'll do this is by **tree augmentation** – on each node, keep track of properties of the subtree rooted at that node – essentially we're adding information.

The augmentation value of a node should be efficiently computable (in $O(1)$ time) from the augmentation values of the children. If so, then we can update the tree in $O(h)$ time – when we modify something in the tree, we only have to walk a path up from that node to to the root in order to fix things.

In particular we can augment by size. For every vertex we want to keep track of how many vertices are in the subtree rooted at that node (including itself). Then we have the rule:

```
x.size = 1 + x.left.size + x.right.size
```

(The empty children are just 0.)

If we maintain this property – the value of a node is a function of the values of its children – then if we insert some new node into the tree, we only have to fix the augmentations of the nodes on the path from our node to the root – none of the other nodes are affected. So that's a $O(h)$ number of things we need to update.

Now if we have size augmentation, we can find teh ith element in $O(h)$ time using binary search! We need to have $i-1$ elements below it, and the way we do this is by recursively binary searching on the left and right subtree, with the remaining value. (You may need to modify these numbers a bit.)

So the fact that we've augmented with size now lets us do indexing operations. Now this lets us do all the sequence operations in $\Theta(h)$ time as well: we have a data structure that allows us to maintain a sequence where we can get to the ith element in $\Theta(h)$ time, which lets us do all the sequence operations.

§7.3 AVL Trees

§7.3.1 Importance of Balance

The worst case h is if we start with 1, then get 5, then 6, and so on, where we're always adding to the right child at the bottom. This ends up with $h = n-1$ (and our tree is a linked list). So we need a way to prevent this, or else we end up with $O(n)$ time for everything.

So it's important to *balance* the binary tree so that we have $h = O(\log n)$ – getting asymptotically close to perfect balance.

§7.3.2 Strategy for Balance

We're going to augment each node with useful information. Then we're going to define a local invariant on this information, and we're going to maintain this invariant when we manipulate the data structure. (If it's not maintained, then we fix it.) Then we'll show this invariant guarantees $h = \Theta(\log n)$. Finally, we'll show algorithms that actually maintain the information and invariant.

We'll discuss **AVL Trees** (Adelson-Velskii and Landis from 1962), which is probably the simplest implementation.

> **Algorithm 7.2 —** For every node, store its height augmentation – the longest path from the vertex to a leaf. (Leaves have height 0.)
>
> Our invariant will be that for every node, the **skew**, or the height difference between its left child and right child, should be at most 1.

Tree height augmentation is easy to implement:

```
X.height = 1 + max(X.left.height, X.right.height)
```

This gives us an $O(1)$ way to calculate the augmentation for a given node, which is what we need in order to compute everything in terms of $O(h)$ every time we modify something. (We use the convention that $h = -1$ for an empty child.) If we want to insert, similarly to before, we update the nodes on the path from that vertex to the root.

§7.3.3 Maintaining the Invariant

We're going to be *rotating an area in the tree* via `left_rotate(x)`. If x has left subtree A and right child y, where y has subtrees B and C, then we want to rotate it so that y is the new top node, x is the child of y with subtrees A and B, and C remains the right child of y.

This rotation preserves the traversal order (which is what matters).

Similarly, we can perform `right_rotate(y)` to do the opposite operation.

> **Claim 7.3 —** If we maintain the AVL Property (that the skew of each node is at most 1), then the height of the tree is $\Theta(\log n)$.

Proof. The lower bound is obvious; we'll prove the upper bound.

Let n_h be the minimum number of nodes of an AVL tree of height h. Then

$$n_h \geq 1 + n_{h-1} + n_{h-2}.$$

(This is because n_h is nondecreasing, and one of the subtrees needs height $h-1$, and the other has height at least $h-2$.) Since $n_{h-1} \geq n_{h-2}$ we can write

$$n_h \geq 1 + 2n_{h-2} > 2n_{h-2}.$$

So this gives $n_h \geq 2^{h/2}$, which means $h < 2\log_2 n$. So we get $h = O(\log n)$. □

So now what we want to do is actually figure out how to maintain this invariant.

§7.3.4 Inserting

If we insert a new node into a simple BST, this can create an imbalance. So we have to fix that, and we'll do that by working our way up the tree (from the insertion to the node), fixing things.

Example 7.4

Take a tree with root 41, next layer 20 and 65, next layer 11, 29, 30, and below that, 26. Then suppose we insert 23.

So 23 is inserted as the left child of 26. Now we look up from the insertion to the first vertex which got imbalanced, which here is the 29 (since its subtrees have 1 and -1).

So now we do a *right rotation* (we choose what to do based on what the skew is, and the local vicinity). Now we have $20 \to 26 \to 23, 29$ instead.

In the abstract, what happens in a right rotation is that we switch which of x and y is on top, and move B over.

Algorithm 7.5 — Let x be the lowest violating node. WLOG x is *right-heavy*, meaning `x.right > x.left`. Then there are three cases. Let y be the right child of x, and as usual let A be the left subtree of A and B and C the left and right subtrees of y.

Case 1 (y is right-heavy). Then A has height $k-1$, y has height $k+1$, B has height $k-1$, and C has height k. In this case, we do a left rotation of x – now x has height k (as both its subtrees A and B are $k-1$), and y is $k=1$. So we've done two things: we've moved the heavy subtree up one, and we've moved B over.

Case 2 (y is balanced). Then A has height $k-1$, and B and C both have height k.

Then we do the same thing – we still left-rotate x. Then A and B are still $k-1$ and k, so x is $k+1$. Meanwhile C is still k, so now the difference is 1 (and y is $k+2$).

Case 3 (y is left-heavy). Then A has height $k-1$, B has height k, and C has height $k-1$. Now the rotation isn't going to save us because B is the thing we need to bring up.

Instead, what we'll do is look at this problematic subtree B; call its root z. We analyze whether *it* is left-heavy, right-heavy, or balanced.

If z is left-heavy, then we first do `right_rotate(y)`, and then `left_rotate(x)`. If z has left subtree B and right subtree D, then B should be $k-1$ and D should be $k-2$. Right-rotating y puts z on top of y and gives y children D and C, with $k-2$ and $k-1$. *Then* left rotating at x moves x down and brings z up, leaving the tree balanced. (We end up with z having children x and y, with x having A and B, and y having D and C.)

The other cases of z work similarly.

So this fixes the lowest place where there was an imbalance. So then we have to go up the tree again to check. We only have to look at nodes along the path from the inserted node to the root, so we can keep going up the tree.

Because each rotation is $O(1)$ and we perform a constant number of these per node, the entire thing is $O(h) = O(\log n)$.

§7.3.5 Conclusion

So with an AVL tree, we've made $h = \Theta(\log n)$. So we've succeeded in making all of our data structure operations take $\log n$ time.

§7.3.6 AVL Sort

The cool thing about having a sorted tree that takes logarithmic time to insert means that we can use it to sort. The way you do this is by putting in all your elements into the AVL tree (which rebalances), and this gives you a $n \log n$ sorting algorithm by taking its in-order traversal.

As you push values into the AVL tree, what's important is that there's a logarithmic number of manipulations that keeps the tree balanced, so that all the time you're paying $\log n$ to put things in.

So this is an $O(n \log n)$ sort.

§8 Priority Queues and Heaps

§8.1 Priority Queues

Definition 8.1. A **priority queue** is a data structure implementing a set S of elements, each associated with a key, supporting the following operations: `insert(S, x)`, `max(S)`, and `delete_max(S)`.

For a dynamic array, we can insert quickly but can't delete quickly; in a sorted dynamic array we can delete the max quickly but can't insert quickly. In an AVL tree, both become $\log n$, but we're using a dynamic data structure that requires pointers and is complicated.

Our goal is to do `build` in linear time, and `insert` and `delete_max` in amortized $\log n$ time.

§8.2 Heaps

A **heap** is an implementation fo a binary queue.

Definition 8.2. A **heap** is an array visualized as a nearly complete binary tree, with the **Max Heap Property**: the key of a node is at least the keys of its children.

The root of the tree is the first element in the array, corresponding to $i = 1$ (where i is the index into the array). The parent of i is $\left\lfloor \frac{i}{2} \right\rfloor$. The left child of i is $2i$, and the right child of i is $2i + 1$. (So we're essentially reading each level left to right.)

The beautiful thing about the heap is that we don't need any pointers – we know where in the array the children are sitting. The height of the binary tree is $O(\log n)$.

Remark 8.3. The heap property is not the same as the search tree property – a heap is not an ordered set. The heap gives us an efficient pointerless implementation to delete a minimum or maximum – but it *doesn't* give us a traversal order. We can't use a heap to get an efficient predecessor or successor.

In some sense this is the opposite of a binary search tree, where we have traversal order, but a bunch of pointers. So we're relaxing our requirements by limiting what we're trying to achieve with the data structure, and that gives us better properties for our specific tasks.

§8.2.1 Heap Operations

The operations we want to think about are `build_max_heap`, which produces a max heap from an unordered array; and `max_heapify`, which corrects a single violation of the heap property. Then we want `insert`, `delete_max`, and `heapsort`.

§8.2.2 Max-Heapify

Algorithm 8.4 — Suppose the trees rooted at `left(i)` and `right(i)` are max-heaps. If $A[i]$ iolates theh max-heap property, correct the violation by "trickling" element $A[i]$ down teh tree, making the subtree rooted at index i a max-heap.

In every trickling step, bring up $A[i]$'s largest child (the bigger one among the left child and right child).

The fact that we're moving up the bigger child maintains the property. Then we knew the children of 14 were okay (less than 14), so even if we bring them up, we're fine.

This takes time $O(\log n)$, since we're trickling down the tree whose height is $O(\log n)$.

Here is the pseudocode:

```
    l = left(i)
    r = right(i)
    if (l <= heap-size(A) and A[l] > A[i])
        largest = l
    else
        largest = i
    if (r <= heap-size(A) and A[r] > A[largest])
        largest = r
    if largest != i
        exchange A[i] and A[largest]
        max_heapify(A, largest)
```

§8.2.3 Build-Max-Heap

Given an array A, we want to turn it into a max heap.

```
    Build_Max_Heap(A):
        for i = n/2 downto 1
            do Max_heapify(A, i)
```

So we're starting one step above the leaves, and applying `max_heapify` every time. (The leaves are already sorted because they don't have any children, so we don't have to do those.)

So we're doing one layer at a time and fixing things. This guarantees that we end up with the max heap property.

This takes $O(k)$ time for a node k levels above the leaves. There are approximately $\frac{n}{2^k}$ such nodes. So then the amount of time taken is approximately

$$\frac{n}{2} \cdot 1 + \frac{n}{4} \cdot 2 + \frac{n}{8} \cdot 3 + \cdots = n \sum \frac{k}{2^k}.$$

The sum on the right is a constant (since we can just calculate the infinite sum), which means this is $O(n)$.

So `build_max_heap` is $O(n)$.

Example 8.5

Perform `build_max_heap` on the array $[4, 1, 3, 2, 16, 9, 10, 14, 8, 7]$.

Solution. Start with the vertices which are not leaves. The first is node 5. It's not broken because $16 > 7$. Then we go one back and see that 2 is a violation (at node 4). We then move the larger child 14 up. Then we keep doing this, going up the tree. □

So we've built our heap in linear time.

§8.2.4 Heap Sort

Algorithm 8.6 — First, build a max heap from an unordered array. Then we get the maximum element $A[1]$.

Now swap elements $A[n]$ and $A[1]$, so that the maximum element is at the end of the array. Discard node n from the heap (by decrementing the size).

The new root may violate the max heap property, but its children are max heaps. So then we can run `max_heapify(A, 1)` on the root to fix this.

Then we keep repeating this (from step 2) until the tree becomes empty.

This takes $O(n \log n)$ time, since fixing the heap takes $O(\log n)$ time and this is done n times.

So now we have a sort algorithm using a data structure that gives us logarithmic (amortized) delete and insert. (This is amortized because we're doing it in an array.) This is a $n \log n$ sorting algorithm that's *in-place* – it doesn't use extra memory (unlike the other $n \log n$ sorting operations we've seen so far) – which is pretty useful.

§9 Graphs

§9.1 Definitions

Definition 9.1. A **graph** $G = (V, E)$ is a set of vertices V, and a set of edges $E \subset V \times V$. We denote $|V| = n$ and $|E| = m$.

Definition 9.2. A graph is **undirected** if we define the edges as sets (so we don't care about the order), and **directed** if we do care about the order.

Definition 9.3. The **degree** of a vertex x is the number of nodes adjacent to x.

In an undirected graph, we have $\sum \deg(v) = 2m$, since we count each edge twice.

Definition 9.4. In a directed graph, the **outdegree** is the number of nodes $y \in V$ with $(x, y) \in E$, and the **indegree** is the number with $(y, x) \in E$.

So the outdegree is the number of edges out, and the indegree is the number of edges in. The sum of the indegrees and outdegrees are both m.

§9.1.1 Graphs as Models

We use graphs to model a lot of things: friendship, power grids, maps, a Rubik cube.

Graphs are visual – we can picture them. This is very beneficial for us as algorithm designers.

§9.1.2 Computer Representations

One way to represent a graph is as an **adjacency list**: have an array with the names of the vertices, and for each vertex, have a list of the edges adjacent to it.

Another is as an **adjacency matrix**: have a matrix where a 1 in (x, y) represents $x \to y$. (If the graph is undirected, then theh adjacent matrix is symmetric.)

These both have pros and cons.

Definition 9.5. A **sparse graph** is a graph where $|E| = O(n)$. A **dense graph** is a graph where $|E| = O(n^2)$.

In a sparse graph, the adjacency list is more economical – you're keeping something that's $O(n)$.

A matrix takes $O(n^2)$ space. If the graph is dense, then the adjacency matrix is faster.

In general, we typically use the adjacency list – there are other representations as well, but today we'll use the adjacency list, which is most popular.

Today we'll only look at undirected graphs.

§9.1.3 Basic Graph Notions

Definition 9.6. A **subgraph** of $G = (V, E)$ is a graph $G' = (V', E')$ with $V' \subseteq V$ and $E' \subseteq E$.

Note that you don't need to take all edges between vertices of V' that were in the original graph – but if you have an edge, you need both vertices of that edge.

Definition 9.7. A **path** P is a sequence of (possibly repeating) vertices V_0, V_1, ..., V_k such that $(V_i, v_{i+1}) \in E$. A path is **simple** if none of its vertices are equal. Its **length** is k (the number of edges).

Definition 9.8. u and v are **connected** if G contains a path from u to v. A graph G is connected if each pair of vertices are connected.

Definition 9.9. A **connected component** is a maximal connected subgraph of the graph.

Definition 9.10. The **distance** between s and t is the length of the minimum-length path from s to t.

§9.2 Breadth-First Search

§9.2.1 Basic Graph Problems

There are many questions we can ask about a graph:

- Find the distance from s to every other vertex.
- Find whether G is connected.
- Find the connected component to which a vertex s belongs.
- Find the connected components of G.

We'll do this via **Breadth-First Search**. We'll implement an algorithm in a bunch of stages – the point is that we have to be careful.

§9.2.2 Algorithm

Algorithm 9.11 — Initially, all the vertices are unmarked, except s. Until no new neighbors are marked, mark all neighbors of currenty marked vertices.

So we start at s, and then we mark all its neighbors, then all their neighbors, and so on.

Note that if we have another connected component, then the algorithm won't reach it.

§9.2.3 Naive Implementation

The complexity of this algorithm depends on how we implement it.

Suppose we implement it with an adjacency list: add a marked array, whose initialization takes $O(n)$. Then we can iterate: look for the marked vertices, go through their neighbors, and mark those neighbors. So you'll pay $O(n)$ in order to go through the array at each step – going through the marked array and marking its neighbors.

If the graph is a single path, for example, then this algorithm takes $\Theta(n^2)$ time. Every time we're progressing by 1 vertex, but paying $O(n)$.

If the graph is dense, then the number of edges is approximately n^2, so this is fine. But if the graph is not dense, then this complexity is not great.

§9.2.4 Improved Implementation

Algorithm 9.12 — Initially, at $i = 0$, s is marked with 0 and all other vertices are unmarked (marked ∞).

In each step, for each unmarked neighbor of a vertex labelled i, mark it with $i + 1$ (if there are none, then stop). Then increment i by 1, and repeat.

So now we're only taking care of the latest ones. Note that we're marking vertices with their distance from s.

So our BFS algorithm is kind of layering the graph. Our labels tell us how far our vertices are from s.

The complexity depends on how we find the unmarked neighbors of vertices marked i. We need to be careful with how we implement this.

One possible implementation is to have our adjacency list and a marked list, but for every layer, we also have a *list of nodes in that layer*. We start at layer 0 (which only has s). Then we move through that list and process all the vertices in it.

When we process v, we go to the adjacency list, look at the list of edges, and every time we see some vertex w with an edge, we go to w in the marked array. If w is already marked then we do nothing. If w si not marked, then we mark it with $i + 1$ and insert it to the $i + 1$ list.

To analyze the complexity of this implementation: initializing the algorithm takes $O(n)$, since we need to initialize lists of size n.

Then marking a node is $O(1)$, and inserting a node into a marked list is $O(1)$. Every node gets marked once, and added once, so that's $O(n)$.

Meanwhile, in the processing, every edge is processed at most once.

So the total complexity is $\Theta(n + m)$.

Remark 9.13. If the graph is connected, then $n \leq m - 1$. This is a breadth-first search from s. But we might have a disconnected graph where the number of vertices is larger than the number of edges, and some of the vertices are sitting on the side. Then we'd have to implement this for each of those components. So when we write that BFS is $\Theta(n + m)$, this is because we'd have to go through all of the vertices to make sure that all of them are marked.

§9.2.5 BFS Tree

We can take the graph and keep track of what the order is when we get to a node: record the first edge at which we arrived at that node. For example, if we start at s and then we arrive at a and x, we mark the edges sa and sx. If then we arrive at z from a, we mark the edge az.

So as we do the BFS, we keep track of the first edge on which we arrive at a given node. That defines a spanning tree.

> **Definition 9.14.** A **spanning tree** of G is a subgraph that's a tree containing all vertices of G.

We will see a lot of spanning trees; one specific one is the BFS spanning tree.

> **Definition 9.15.** The **BFS Spanning Tree** is a level structure with the level indicating the shortest path from the root.

So each level indicates where we are in terms of distance from s.

We can have edges between consecutive levels, as well as edges within a level. But we can't have edges between vertices in levels that differ by at least 2: then we'd have a shorter path to the later vertex.

The layers of the spanning tree tells us the actual distances in the graph.

Spanning trees are extremely useful because we can use them to actually move information along the edges of the spanning tree.

For example, the Bacon Number forms a spanning tree based on "u acted with v". These are originally derived from Erdos numbers, which describe the "collaborative distance" between Erdos and another person, with edges in terms of "who wrote a paper with whom".

There's also the Erdos-Bacon number – Natali Portman's is 7 and Elon Musk's is 6. The lowest Erdos-Bacon number is 3: Prof. Daniel Kleitman coauthored papers with Erdos and has a Bacon number of 2 by acting with Minnie Driver in *Good Will Hunting*. (This is the sum of the Erdos and Bacon numbers.)

§10 Depth-First Search

BFS looks somewhat like a queue: we keep a front that moves through the graph. Its complexity is $O(n+m)$. It gives us a spanning tree that layers the graph – each layer tells us the distance from s, and this gives us additional information: edges can only exist between vertices in the same or consecutive layers.

Today we will see a different way of traversing the graph – which will *also* generate a spanning tree, but a different one.

Instead of going breadth-wise, we'll go depth-wise. Depth-first search is a new way to explore a graph and number its nodes. It works for both directed and undirected graphs, and generates an informative DFS tree.

DFS has a very physical interpretation (more than BFS, since we can't split into multiple entities and go layer after layer). Imagine that we walk on a graph with a pen (used to number nodes and mark edges) and a flag (representing our center of activity). We can think of this as an impatient visit to a museum – the nodes are the rooms, edges are doors, and we try always to see new rooms. If it's possible to see a new room we go there, otherwise we backtrack to the room we initially came from. Rather than backtracking from the initial room, we halt.

Start by labelling our initial node s as 0. We have a red dot representing our **center of activity**, as well as you.

When we start the algorithm, we pick an edge from s and traverse it; we then mark that edge as traversed. We then go over to the vertex on the other side of that edge, and check whether it's marked or not. If it's

not marked, then we mark it with the next index (which here is a 1), and we mark this edge to be in the DFS tree. Now we move our center of activity to that vertex.

Now we repeat the same thing from the new center of activity: pick an edge, mark it, and traverse it to the next vertex. Check if that vertex is marked; it isn't, so we mark it, mark the edge as part of the DFS tree, and move our center of activity.

Now we again pick an edge, mark it, and move ourselves to the next vertex. It's not marked, so we mark that vertex as 3, mark the edge for the DFS tree, and move our center of activity.

Now suppose we choose an edge that leads to a marked vertex. We mark the edge we take, and then notice our vertex is already marked: so we go back to 3. We check whehter there's anything else at 3 which we can go to: there isn't, so we backtrack to 1. Then we go to another outgoing edge from 2, and continue.

We always backtrack along the *tree* edge that allowed us to arrive at the node. We repeat this until we end up at node 0 with no unused edges, in which case we halt (the ultimate form of backtrack).

(Note: the label should start at 1 instead.)

Algorithm 10.1 — First mark all edges unused, and initialize $i = 0$. Perform $i \to i+1$, mark the center of activity with i. If the CoA has an unused edge, then:

- Choose one unused edge e. Mark e used, and traverse e to u.
- If u is marked, go back.
- If it's marked, then mark this as a tree edge, move the CoA here, and go back to (1).

If there are no unused edges, then if this is the first vertex then halt; otherwise backtrack along the edge from here marked D.

To prove this works :

Lemma 10.2

If you number a node, you'll eventually backtrack from it.

This follows directly: eventually we will run out of outgoing edges.

Theorem 10.3

If the graph is connected, we will number all its vertices.

Proof. Use contradiction: assume it isn't true, so there is a subgraph of nodes which are numbered S. There's also an un-numbered node somewhere. So there is an edge from a numbered to an unnumbered node. (Consider the first unnumbered node along the path.) So then i has an edge to our unnumbered node v. This edge is unmarked. But we backtracked from i, and that's impossible because we only backtrack when we mark all edges. □

So then we eventually create a *spanning tree* of the graph.

§10.1 Complexity

Intuitively, per edge we do one marking, two traversals (one where we move the center of activity along the edge, and one when we go back). And there's oen backtracking. So each edge contributes a constant amount of work, giving $O(m)$. Meanwhile, per node, we number it once, bring the CoA there once, and remove the

CoA once. So that's also $O(1)$ per node, giving $O(n)$. (At every node we also make $\deg u$ visits, but we've already counted that.) So DFS is $O(n+m)$.

> **Remark 10.4.** BFS and DFS are both $O(n+m)$.

Now we can think about our graph, having both tree edges and non-tree edges. We can look at the edges in terms of the numbering that they lead to: edges that we call *back edges* lead to *ancestors*. (We don't want to just say they lead to an old node – because if 5 has two children 6 and 7, then we don't want to consider the edge between 6 and 7 a back edge.)

So we have tree edges in the tree, and back edges leading to ancestors.

We *can't* have any edges other than tree edges or back edges.

An ancestor is a vertex on your path to the root (which is 1).

§10.2 Directed DFS

We can do the same algorithm; we just have directed edges (so we make sure to traverse in the direction of the edge – we only look at the outgoing edges from a node when we're trying to leave that node).

The same proof as before still works: now we take a directed path from s to an unnumbered node, and do the exact same proof.

Now we get a directed DFS tree.

Our tree edges are directed from the parent to the children. (As we go deeper into the tree, numbers get bigger.)

> **Remark 10.5.** The thing about DFS trees is that the numbers are indicative of this "depthwise" move – larger numbers were traversed later. This tree will actually have a lot more information than the BFS one.

Again we have back edges leading to an ancestor.

We also have forward edges, leading to your descendants.

Finally, we can have *cross edges* between "unrelated" nodes.

Suppose we want to determine edge types while doing DFS. Use three colors: mark nodes white (undiscovered), gray (visited but not backtracked from – unfinished), and black (backtracked from – finished). So each node goes from white to gray to black.

We also associate time intervals with the process, represented by parentheses.

So we start at 1, which is gray. We then go to 2, mark it gray, then go to 3, and 4. Then we go down to 2. We see that 2 is marked gray – it's being processed right now. (These are the nodes that are in the tree and we haven't finished processing their outgoing edges.) So then this must be a back edge. Now we backtrack to 4 and mark it black, and we go from 4 to 3 (which is still being processed). Now we open 5.

So a back edge is when the target is tray.

Now suppose when we reach 7, we have one outgoing edge, and it's to 6, which has been marked black. This is a cross edge: we know this because its target is black – which tells us that the processing of that black target doesn't overlap with ours. Everything along the path to our vertex should be gray, because these guys are not done – so since that guy is black, and has a lower number, this must be a cross edge.

So a cross edge is when our vertex is black and has a lower number.

When we leave 5, we see a time interval when we processed everything in 5's subtree:

$$(1(2(3(44)(5(66)(77)5)3)\dots 2)\dots 1).$$

So the two 3 parentheses represent enclosing the subtree rooted at 3.

Now from 9 we see an edge to 2. That edge is similar to the edge $7 \to 6$: it points from a gray vertex, to a black vertex with a smaller number. So this is also a cross edge. Notice that 2 must have finished processing before we started here, so this node can't be our ancestor – it finished processing while we're still here.

So we finish 9, then 10, then 8, and we go back to 1. Now we see the edge $1 \to 9$. It leads form a gray vertex to a black vertex, but the numbering goes up: 9 is bigger than 1. So this must be a forwards edge.

A forwards edge has its interval overlapping – forwards edges are when the target is black and the interval overlaps, cross edges are when the target is black and the interval doesn't overlap. In terms of numbering, for cross edges, you must have been discovered after the target; for forwards edges you must have been discovered before.

So we have a way of figuring out what is a back edge, a cross edge, and a forwards edge. So based on the numbering, we know exactly what's going on.

This describes every edge in the graph.

§10.3 Uses of DFS Tree

So we've developed a way of marking the edges of a spanning tree, that allows us to detect which edges are tree, forwards, back, and cross.

Now we can detect *cycles* in a graph: a directed graph is acyclic if and only if a DFS yields no back edges. The algorithm detects cycles by finding back edges.

§11 DFS Application

Last lecture, we used DFS to generate a spanning tree, and we characterized all edges in the graph as:

- Tree edges, which are in the tree;
- Back edges, which lead to ancestors;
- Forward edges, which go to descendants (lower number to a finished higher number);
- Cross edges, which point to an unrelated subtree.

§11.1 Topological Sort

A **topological sort** is a numbering of the vertices of a **directed acyclic graph** such that all edges go from low to high – if $u \to v$ then $TS(u) < TS(v)$.

So we have to in some sense identify the "latest" things in the search. We can see that 1 and 2 are the roots of the graph, and 6 and 7 are sinks. So we're following the edges in some sense.

This always exists.

A naive algorithm checking every possibility would be $O(n!\,m)$.

A **topological reverse sort** is the opposite – $TR(u) > TR(v)$. This is a symmetric problem.

Consider the DFS numbering. This isn't exactly a topological sort (or a reverse topological sort) – cross edges go big to small.

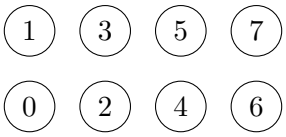
But DFS *backtrack numbering* gives us a reverse topological sort. If we place the numbers *when we backtrack* instead of when we first visit the vertex, then we get a reverse topological sort. We can turn this into a topological sort by flipping all the numbers.

So that takes us from $O(n!\,m)$ to $O(n+m)$.

So we can use DFS and how it works to get a solution to our algorithm. We're not exactly using DFS as a black box here – we're understanding a little bit about it and using that to get the answer.

§11.2 Strongly Connected Components

Definition 11.1. A **strongly connected component** in an acyclic graph is a subset where for any (u, v), there is a path from u to v and from v to u.



Breaking in to strongly connected components is super useful. For example, understanding networks.

Definition 11.2. The **condensation graph** is the graph created by collapsing strongly connected components.

So now the edges indicate how the SCCs are connected.

What we want to do today is find a graph algorithm for finding SCCs. The algorithm we'll see is by Robert Tarjan, one of the leaders in the field of graph algorithms. It's from the 1970s, when people had just started doing these algorithms.

You can solve SCCs kind of using DFS as a black box, without looking into its internals. But this is an algorithm that actually digs into the depths of DFS and uses the actual structure of the traversal, and also does one more thing: it combines the use of a DFS algorithm on the graph, together with a stack on the side. So we're going to use a combination of these two data structures to find our SCCs.

Have a vector `in` where `in[u]` is the *discovery time* of u – the time when we discover it and give it a label. We'll also use the same three colors as before (white is undiscovered, gray discovered, and black finished). We'll also keep `low`, where `low[u]` is the minimal discovery time for a path starting at u.

We now go and label each vertex with $(0, 0)$ for both `in` and `low`.

Now we start a DFS at the root. Label the root 1; so it's labeled $(1, 1)$. Then we go on to the next vertex and label it $(2, 2)$; then $(3, 3)$, $(4, 4)$, $(5, 5)$ – going down the DFS.

Now we bump into a back edge: suppose $5 \to 3$. Now when we bump into this back edge, we update `low`: there's a path from 5 to 3, so we update 5 to $(5, 3)$. Now suppose we see another back edge $5 \to 4$; since the current low of 5 is $3 < 4$, we don't update it.

The idea is that we're trying to look for the entrypoint into the SCC.

Now we finalize 5 because we've traversed all its outgoing edges.

So now we back up to 4. When we backtrack, we know there's a path from 4 to 5, and 5 has a 3, so we update 4's `low` to 3 as well. What we've discovered is $4 \to 5 \to 3$. So `low` is telling us that there's a path from both 4 and 5 to 3.

Now we keep going – we label $(6, 6)$. It doesn't have any outgoing edges, so we *finalize* that node 6 as a SCC. When we finalize a node, if it has `low = in`, then it's the root of a strongly connected componenent (call that SCC 1).

Now we backtrack to 4. When we finalize 4, we see its label is $3 < 6$, so we don't need to update the low. So we finalize 4 and backtrack.

Now we backtarck to 3. Then we go to $(7, 7)$. Now when we're at 7, we have a path $7 \to 6$, which is a cross edge. But we *don't* update because 6 already belongs to another SCC that we've already declared.

So if we have an edge that leads to a node that has a smaller low number, but is already part of another SCC, then we don't update. This is a change to the rule.

Now we see $7 = 7$, so this is the root of another SCC, call that SCC2.

Now we backtrack to 3 and finalize it. We see $3 = 3$, so it's the root of a SCC, which consists of $\{3, 4, 5\}$.

Now we backtrack to 2, and discover $(8, 8)$, $(9, 9)$, $(10, 10)$. Now from 10, we see the edge to 9, so we update $(10, 9)$. Then we finalize 10; because $10 \neq 9$, we backtrack and get to 9. Now $9 = 9$, so this is the root of a SCC consisting of $\{9, 10\}$, which is SCC4.

Now from 8 we see $8 \to 10$, a forwards edge, which does nothing. Then we see an edge to 1, so we update 8 with 1. Then 2 gets updated with that 1.

So we made a DFS traversal and detected all the strongly connected componenets.

```
in = {0, ..., 0}
low = {0, ..., 0}
scc_num = 1
time = 0

Stack = [] # SCC accumulation stack (not the DFS stack) -- accumulates SCC being constructed

instack = {false, ..., false} # which nodes are in the stack

DFS(u):
    low[u] = in[u] = time++
    stack.push(u)
    instack(u) = true
    for v adjacent to u:
        if in[v] == 0: # if undiscovered, call DFS
            DFS(v)
            low[u] = min(low[u], low[v]) # if the DFS found a path to a lower number, then update low
        else if instack[v]: # if v is in the current SCC being accumulated
            low[u] = min(low[u], low[v])
        # otherwise v belongs to a different SCC so we do nothing
    if low[u] == in[u]: # u is a root of the SCC
        # pop everything in the stack and declare it a SCC
        while true:
            w = stack.op()
            scc(w) = scc_num
            instack[w] = false
            if w == u: break
        scc_num++
```

In the end, we need to go look for other possible roots – stuff that didn't get reached from our vertex. So we need to start a DFS traversal from every unmarked vertex.

```
for each node u in G:
    if in[u] == 0:
        DFS(u);
```

Suppose we had the edge $3 \to 1$ as well. This is a back edge, which closes a cycle: so what that's doing is that now the connected components get combined.

Note that 4 and 5 will not actually get their `low` updated to 1 – the root of the monster is 1, bur 4 and 5 will not get labelled 1 (since we've already left them by that point). So we update 3 to 1, but 4 and 5 are not going to get updated because they're already finished in the DFS.

But fortunately, 4 and 5 are still in the stack. And when we finish we opp everything in the stack.

So the key is that a node remains on the stack after it has been visited iff it has a path to some node earlier on the stack. Node 3 won't be finalized because $3 \neq 1$. So 4 and 5 stay on the stack, which is all we need. Even though 4 and 5 have the "wrong" label, there *is* a path to the root, and when we do the popping, they're still in the stack.

So the stack is actually necessary – the low labels don't really tell you the component labels.

This is $O(n + m)$.

So what we saw today was two DFS algorithms for popular applications (topological sort and SCCs) in $O(n + n)$. DFS reduces something potentially exponential to linear, by showing us how to traverse the graph.

The beauty of Tarjan's algorithm is that it gives you a feel of how DFS actually works – by adding another data structure to make the whole thing detect connected components.

§12 Dijkstra's Algorithm

Question 12.1. Given vertices u and v, how can we find a *shortest path* from u to v?

Note that the shortest path doesn't have to be unique: there may be two routes from u to v with the same length.

WLOG we will always assume the graph is directed – if we have an undirected graph, we can replace each undirected edge with a directed edge.

All algorithms we build will actually solve the **single source shortest paths** problem – given u, find a shortest path from u to *every other vertex* in the graph.

Notation 12.2. Define $\delta(v)$ to be the length of a shortest path from u to v (or the *distance* from u to v).

BFS explores the graph starting from a single vertex, and it first looks at the vertices of distance 1, then 2, then 3, and so on. In fact the BFS tree is actually a shortest path tree – the path from the root of the tree to any vertex is a shortest path.

So BFS just solves the shortest paths problem in terms of number of edges traversed. But sometimes not all roads have the same length (or time to traverse). So we want a richer model to handle more interesting shortest paths questions.

§12.1 Weighted Graphs

Definition 12.3. For every directed edge $u \to v$, associate it with a **weight** $w(u, v) \in \mathbb{R}$.

For today, since we're thinking of the weights as distances or travel times, we'll assume $w(u, v) \geq 0$.

Definition 12.4. A **path** from u_1 to u_n is a series of directed edges (u_1, u_2), (u_2, u_3), $\ldots$, (u_{n-1}, u_n). The total **cost** or **length** of a path is the sum of the weights of the edges.

For example, in Google Maps, edge weights are travel times.

Now we can't use BFS directly – a sequence of two hops can have smaller length than a sequence of one hop (since BFS as discussed doesn't pay attention to the edge weights).

If our weights are integers, we can expand the edge into a path of the appropriate length.

But this only makes sense for integer weights. And if the numbers are big, then we blow up the graph tremendously – the size of the graph depends on the size of the weights. We want to avoid this.

§12.2 Structure of the SP Problem

We'll solve this with Dijkstra's Algorithm. This is pretty hard; to build it up, we'll look at a few properties of the SP problem.

§12.2.1 Optimal Subproblems

Fact 12.5 — If a shortest path from u to v goes through a, then it contains a shortest path from u to a.

Proof. If not, replace the $u \to a$ segment with the shortest path; this gives a shorter path from u to v. □

This tells us that if we've solved the shortest paths problem from u to v, we've also solved it for u to a. So we want to try to solve the subproblems first, since they're only easier – so we want to try to find shortest paths to *intermediate vertices*.

But we don't know ahead of time which vertices go into the shortest path from u to v. So we don't know which subproblems to solve, in what order.

§12.2.2 Scaffolding for SP Algorithms

Definition 12.6. Our algorithms will define:

- `d[v]`, our *best guess* of distance from u to v (which we'll maintain to be at least $\delta(v)$);
- A series of pointers `parent[v]`, where the parent of v comes right before v in a shortest path from u to v.

Then if we successfully build these parent pointers, we can follow them backwards to reconstruct the shortest path.

So now we have the idea that we're going to iteratively refine our guesses. To get an idea of how to do that refinement, we'll think more about the structure of shortest paths.

§12.2.3 The Triangle Inequality

Proposition 12.7 (Triangle Inequality)
For any u, v, and a,

$$\delta(u, v) \leq \delta(u, a) + \delta(a, v).$$

Proof. Assume not; then replace the shortest path with the shortest paths $u \to a \to v$. □

Corollary 12.8

If $(a, v) \in E$, then

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

We can use this to *improve our guesses*.

§12.3 Relaxation Algorithms

§12.3.1 Relaxation

The key building block for SP algorithms is *relaxation*. Suppose we have some guesses, and the invariant that our guesses are upper bounds on the true distances.

Whenever `d[a]` violates the triangle inequality, we feel stressed, because we know that our guesses can't be correct.

So then we fix the guesses so that the triangle inequality is satisfied:

```
    def try_to_relax(a, v):
        if d[v] > d[a] + w(a, v):
            d[v] = d[a] + w(a, v)
            parent[v] = a
```

So if we get a violation, then we update our guess – now our best guess for a shortest path through v is that it goes through a. And since our guess is that the path goes through a, we update the parent pointer to be a.

We call this **relaxing the edge** (a, v).

Clearly this fixes the violation of the triangle inequality.

Lemma 12.9 (Safety Lemma)

Relaxation is safe – it preserves the invariant that our guesses are upper bounds, meaning

$$\delta[v] \leq d[v].$$

Proof. This is true from the triangle inequality. □

§12.3.2 Setup

So now we have a framework for a shortest path algorithm: keep on relaxing until we can't relax anymore. (We'll have to make intelligent choices for how to do this.)

```
    def sssp(G, u):
        for v in V:
            d[v] = infty
            parent[v] = None
        d[u] = 0
        while we are not done:
            (v1, v2) = pick an edge somehow
            try_to_relax(v1, v2)
        return d, parent
```

To describe an algorithm, we need a way to describe how to pick the edges and to decide when we're done.

§12.4 Dijkstra's Algorithm

We build up a set S of vertices with *correct* distances, by starting with u and growing S outwards by *greedily* relaxing edges. So we're building this frontier outwards into the graph, of having correct distances.

```
    def dijkstra(G, u):
        for v in V:
            d[v] = infty
            parent[v] = None
        d[u] = 0
        S = {u}
        pq = PriorityQueue.build(d)
        # Stick all vertices into the (min) priority queue
        # priority is our current guess for distance
        while pq:
            v1 = pq.pop_min()
            S.add(v1)
            for (v1, v2) in E:
                try_to_relax(v1, v2)
                pq.update_key(v2)
```

So we pop the min from the priority queue into S, and then we try to relax all edges from that vertex. Relaxing updates the distance estimates, so we then update the priorities in our priority queue.

> **Remark 12.10.** Updating the keys in our priority queue isn't something we discussed earlier – we'll discuss how to do it later, but we really can do this.

> **Example 12.11**
> Perform Dijkstra's Algorithm on the graph $u \to a$ 5, $u \to b$ 15, $a \to b$ 2, $a \to c$ 5, $b \to c$ 2.

First, u is the min in the queue, so we pop it and add u to S. Then we relax the edges from u; so we update a to 5 and b to 15. Our parent pointers for both a and b now go back to u.

We've done a good job at a; so far, it hasn't seen the edge $a \to b$ that gives a shorter path to b.

Now we pop the min again, which is a. There's two edges leaving it – $a \to b$ and $a \to c$. Now we update our guesses, so we update b to 7 and c to 10.

Now we pop the min again, which is b. We then relax the edge $b \to c$, and this gives a path of length $7 + 2 = 9$ to c.

So we've got a shortest path from u to c. Along the way, a lotnf nontrivial things happened.

> **Remark 12.12.** We said that the algorithm was making greedy choices. Here that means it's choosing to process vertices in a greedy order – the next vertex we process is the one that seems closest to u among all remaining vertices. Greedy choices don't always work, but when they do, they're pretty powerful.

> **Remark 12.13.** Note that greediness doesn't mean leaving every vertex out of its cheapest edge – that wouldn't work.

§12.4.1 Correctness

> **Claim 12.14 —** When we add a vertex v to S, $d[v] = \delta[v]$.

Proof. Assume not, and take the first vertex v where this fails. Now take a shortest path from u to v, where y is the *first* vertex outside S in this path, and x is the vertex right before y. So we have a path

$$u \to \cdots \to x \to y \to \cdots \to v.$$

Since x was already in S, and we assumed v was the first vertex which failed, we know $d[x] = \delta[x]$.

Note also that $u \to \cdots \to x \to y$ is a shortest path. So

$$\delta[y] = \delta[x] + w(x, y) = d[x] + w(x, y).$$

The choice of relaxation order means that we've already relaxed the edge $x \to y$. So that means we have

$$d[y] \leq d[x] + w(x, y) = \delta[y].$$

So we must actually have equality: we've already computed the distance to y correctly as well.

So now we've correctly computed the distance to y.

When we decided to process v: by greediness, we must have $d[v] \leq d[y] = \delta[y]$. But by the optimal subproblem property (since weights are nonnegative), $\delta[y] \leq \delta[v] \leq d[v]$.

So we must have $d[v] = \delta[v]$, contradiction. □

§12.4.2 Runtime

The runtime is dominated by the priority queue operations – since `try_to_relax` takes constant time.

We call `pop_min` $|V|$ times – every time we process a vertex. Meanwhile, `update_key` is called $|E|$ times, since every edge gets processed once.

There are different priority queue operations we could use, and those take different amounts of time.

If we use a Direct Access Array, then to do `pop_min` we'd have to scan through the entire array. So this gives $O(V^2 + E) = O(V^2)$.

But a better implementation of priority queues is using heaps. We can actually update a heap key in logarithmic time, so this gives $O((V + E) \log V)$ time. This is better if E is significantly less than V^2, but worse if $E = O(V^2)$.

It turns out tha tthere's another data structure, called a **Fibonacci heap**, which can do this in $O(V \log V + E)$ amortized. We won't learn this, but it's citeable.

So now we need to look at how to update keys in a heap. In addition to the heap, store a set S mapping priority queue elements to heap locations. Assume that vertex IDs are in 0 through $|V|-1$, so we can use a direct access array to implement the set.

In order to update the key, look up the location of v and then perform swaps with the parents and children to recover the binary heap property, updating S accordingly.

§13 Bellman-Ford Algorithm

Last time we saw Dijkstra's algorithm for finding the shortest path when edge weights are nonnegative. But we may want to consider graphs with negative weights as well: for example, if weights measure energy gain or loss.

Dijkstra's algorithm will not work here: if there's a really negative edge, Dijkstra may not see it in the correct order.

So we have to design a new algorithm that works for graphs with negative edge weights. We'll still use the relaxation framework:

```
def sssp(G, u):
    for v in V:
        d[v] = infty
        parent[v] = None
    d[u] = 0
    while we are not done:
        (v1, v2) = pick an edge somehow
        try_to_relax(v1, v2)
    return d, parent
```

§13.1 Bad Relaxation Orders

Dijkstra chooses edges greedily and relaxes every edge exactly once. It seems too optimistic that we can get away with relaxing edges only once, if edge weights are negative.

So we can ask what happens if we try repeatedly relaxing edges – will the algorithm necessarily converge?

> **Question 13.1.** Suppose we choose edges to relax arbitrarily (allowing repeats), where every relaxation is nontrivial. Does the algorithm converge quickly?

The answer is no; there is an order that takes $\Omega(2^{n/2})$ steps to converge. Take a graph with edges $v_0 \to v_1 \to v_2 \to \cdots \to v_n$, with shortcut edges $v_2 \to v_4 \to v_6 \to \cdots$. On the first copy $v_0 \to v_1 \to v_2$ and $v_0 \to v_2$, the weights are $2^{n/2}$; then in the next copy $v_2 \to v_3 \to v_4$ and $v_2 \to v_4$, the weights are $2^{n/2-1}$, and so on.

The relaxation order relaxes the left and right edges in the gadget (the edges $v_0 \to v_1$ and $v_1 \to v_2$), then recursively relaxes the right subgraph, then relax the top edge, then relax the right subgraph again. This is because the first time we do the right subgraph, we think that the distsance is 2^{n+1}, and now we know it's 2^n.

We can see that $T(n) \geq 3 + 2T(n-2)$, which gives $T(n) \geq \Omega(2^{n/2})$.

So what this tells us is that we can't be totally naive in choosing which edges to relax.

§13.2 Path Relaxation Lemma

This is the key content for today: we're going to think about the structure of the shortest path algorithm again.

> **Lemma 13.2**
>
> If $u \to u_1 \to u_2 \to \cdots \to u_{k-1} \to u_k$ is a shortest path, and we relax the edges *in this order* (possibly with other relaxations in between), then $\delta[u_i] = d[u_i]$ for all i along the path, no matter what other relaxations we do.

This is intuitively obvious – you see the edge $u \to u_1$, then $u_1 \to u_2$, and so on.

Proof. Induct on path length; the base case (a path of length 0) is vacuous.

Assume it's true up to $k-1$, so $\delta[u_{k-1}] = d[u_{k-1}]$. Then after the relaxation of (u_{k-1}, u_k), we have

$$d[u_k] \leq d[u_{k-1}] + w(u_{k-1}, u_k) = \delta[u_{k-1}] + w(u_{k-1}, u_k) = \delta[u_k]$$

(this is the true distance because this is a shortest path). Since $d[u_k] \geq \delta[u_k]$ at all times, they must be equal. □

What this tells us is that if we design a relaxation order such that for every shortest path, we relax the edges on the path in order – then we're good. But of course, the problem is that we don't know the order in advance, since we don't know the shortest path.

Fact 13.3 — On a graph with V vertices, a shortest path has at most $V-1$ edges.

Algorithm 13.4 (Bellman-Ford) **—** Number the edges from 1 to E. Relax them in this order $V-1$ times.

This is very different from Dijkstra's – which relaxes every edge at most once and looks at the graph very carefully to decide the order to process and relax. This algorithm doesn't look at the graph at all! This is slower; it's like a big hammer.

Remark 13.5 (Big Lie)**.** The fact above isn't true if we have a negative-weight cycle. But then there is no well-defined shortest path to vertices on the cycle (or vertices reachable from the cycle), since we can go around the cycle arbitrarily many times.

So the shortest path problem isn't well-defined if there's a negative weight cycle. This means we need to do more: given a graph G and source u, output whether there exists a negative-weight cycle. If not, compute shortest paths from u to all other vertices.

§13.3 Baby Bellman-Ford

We will bracket this issue of negative-weight cycles for now, and study Bellman-Ford under the assumption there are no negative weight cycles.

```
def BellmanFord(G, u):
    for v in V:
        d[v] = infty
        parent[v] = None
    d[u] = 0
    for i in range(|V| - 1):
    for (v1, v2) in E:
        try_to_relax(v1, v2)
```

This has runtime $O(VE)$.

Correctness. This is almost automatic from the path relaxation lemma. Suppose there is a shortest path $(u, v_1, \ldots, v_k)$, where $k \leq |V|-1$.

In the first pass, we relax (u, v_1) (together with all other edges), in the second pass we relax (v_1, v_2) (along with all other edges), and so on. So in the $|V|-1$ rounds, we relax the edges of the path in the correct order. □

Remark 13.6. After k iterations, the distances are correct for any vertex v with a shortest path of k edges from u.

Remark 13.7. It's kind of magical that we don't look at the graph *at all* – this is the opposite of Dijkstra. The runtime is significantly slower, though.

§13.4 Negative Cycles

We want our algorithm to detect negative cycles as well.

```
def BellmanFord(G, u):
    for v in V:
        d[v] = infty
        parent[v] = None
    d[u] = 0
    for i in range(|V| - 1):
    for (v1, v2) in E:
        try_to_relax(v1, v2)
    # Check for negative cycles
    for (v1, v2) in E:
        if d[v2] > d[v1] + w(v1, v2):
            return False
    return True, d, parent
```

This is because we know that if there are no negative cycles, we've correctly computed all distances, and the triangle inequality is satisfied. Meanwhile, if there is a negative cycle, then the triangle inequality must fail for two vertices inside that cycle: this is because if we have a negative weight cycle, then

$$\sum_{i=1}^{k} w(v_i, v_{i+1}) < 0.$$

So then we have

$$\sum_{i=1}^{k} (d[v_i] + w(v_i, v_{i+1})) < \sum_{i=1}^{k} d[v_{i+1}]$$

(since the two sums of the $d[v_j]$ are the same). So at least one edge must violate the triangle inequality.

Remark 13.8. It's not necessarily true that *all* violations of the triangle inequality are on the cycle.

§13.5 Shortest paths on DAGs

For directed acyclic graphs, we can actually find a $O(V + E)$ algorithm. This is significantly faster than Bellman-Ford, and even faster than Dijkstra's (while allowing negative edge weights).

Algorithm 13.9 — First, run topological sort. Now for each v in the topological order (in that order), relax all edges outgoing from v.

This works because all paths in this graph go forward in the topological ordering. So we're guaranteed to relax the edges of every path in the order which they appear in, and by the Path Relaxation Lemma we're done.

§14 All-pairs Shortest Paths

So far we've discussed the single-source shortest path algorithm, and seen four algorithms: BFS (which only works on unweighted graphs, and has linear runtime) and three relaxation-based algorithms – DAG Relaxation, Dijkstra's ALgorithm, and Bellman-Ford.

Today we will discuss a generalization:

> **Question 14.1.** Given a weighted directed graph G, for *every* pair of vertices (u, v), compute the shortest path from u to v.

For example, paper atlases used to come with the shortest path distances between every pair of cities. We want to output, for every source u and target v, the distance `d[u, v]` and the parent pointer `parent[u, v]`. Note that the output size is $O(V^2)$ – for dense graphs this is comparable to the size of the input, but for sparse graphs it's bigger.

§14.1 Nonnegative Weights

First we'll think about how to solve the problem for graphs with nonnegative edge weights.

We could try running Dijkstra from each vertex; this takes $O(V^2 \log V + VE)$ time. For dense graphs, this is $O(V^3)$.

For sparse graphs, this is $O(V^2 \log V)$ – it's almost linear in the output size, since $\log V$ is usually small. So this is pretty good.

We're not going to be able to beat V^3 here; later we'll discuss why.

But when there's negative edges, suppose we ran Bellman-Ford starting at every vertex. Then the runtime is $O(V^2 E)$. For dense graphs, this is $O(V^4)$, which is terrible. For sparse graphs, it's $O(V^3)$, which is not great either.

The key thing we'll do today is show how to match the runtimes in the case of nonnegative edge weights. (Dijkstra from every vertex is close to optimal in the nonnegative case; we'll discuss this later.)

§14.2 Reweighting

The key idea is reweighting. Given a graph, we want to construct a **reweighting** G', which may have added vertices and edges, and changed weights, such that a path is a shortest path in G iff it is one in G'. (The weight may be different, but this is fine – if we can get the shortest paths in G', we can get the shortest paths in G and compute their weights.)

The goal is to create G' which has nonnegative weights.

We can try adding a large constant to all the edges; but this doesn't work because it favors paths with fewer edges. So we need another approach.

§14.3 A Beautiful Symmetry

Let $h : V \to \mathbb{R}$ be an arbitrary function, called a *potential function*. Now take $G' = G$ with weights

$$w'(u, v) = w(u, v) + h(u) - h(v).$$

Claim 14.2 — This is a reweighing.

Proof. We'll prove that for *any* path from u_0 to u_k,

$$\ell_{G'}(p) = \ell_G(p) + h(u_0) - h(u_k).$$

In particular, the amount the path length changed only depends on the endpoints.

The proof is just that the sum telescopes: we get

$$\sum_{i=0}^{k-1} w'(u_i, u_{i+1}) = \sum_{i=0}^{k-1} w(u_i, u_{i+1}) + (h(u_0) - h(u_1)) + (h(u_1) - h(u_2)) + \cdots + (h(u_{k-1}) - h(u_k)).$$

□

§14.4 Finding a Potential

Now we saw that any choice of h leads to a reweighing; but we want to make the edge weights nonnegative, so we want to find a potential that does so.

We want to find a function $h(v)$ such that

$$w(u, v) + h(u) \geq h(v).$$

Recall the Triangle Inequality

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

where δ are the distances from s. These are the same inequality! So we can pick an arbitrary source vertex s, compute the distances from the source with SSSP (using Bellman-Ford, since we have negative edges), and reweigh the graph using the distances from that vertex.

There's a problem, though: what do we do about vertices not reachable from the source vertex? We need a source vertex which can reach every other vertex, and we don't know that one exists.

So what we do is we *declare* one. Add a new vertex s, with weight-0 directed edges from s to every vertex. This is just a tool to allow us to use Bellman-Ford to compute the potential function. Here s will not be part of the reweighed graph; it's a modification we make before we feed the graph to Bellman-Ford.

But it's possible that there's a negative cycle. If there's a negative cycle, APSP is not well-defined; Bellman-Ford detects negative cycles, so this is fine.

Adding s can't create negative cycles, because it can't create any cycles – it has no incoming edges.

§14.5 Overall Algorithm

This is **Johnson's Algorithm**.

```
def Johnson(G, w):
G0, w) = add_supernode(G, w)
neg_cycle, h = BellmanFod(G0, w0)
if neg_cycle:
    return none
for (u, v) in E:
    w2[u, v] = w[u, v] + h[u] - h[v]
for u in V:
    d2 = Dijkstra(G, w2, u)
    for v in V:
        d[u, v] = d2[v] + h[v] - h[u]
return d
```

The runtime is dominated by the calls to Dijkstra and Bellman-Ford. Bellman-Ford is $O(VE)$ and Dijstra V times is $O(V^2 \log V + VE)$, so the runtime is $O(V^2 \log V + VE)$.

So we've achieved the same runtime as in the nonnegative weight case.

> **Remark 14.3.** All distances from s are *at most* 0; but there may be negative distances.

§14.6 Other Algorithms

For dense graphs, we'll see another classic algorithm that achieves $O(V^3)$. There are many improved algorithms in the literature for special cases – APSP is a heavily studied problem. The fastest known algorithm for APSP on dense graphs has the runtime

$$\frac{V^3}{2^{\Omega(\sqrt{\log V})}}.$$

If we had $2^{\log V}$ this would be a polynomial improvement; but because of the square root, it isn't. This is a teeny bit faster. It's not practical; the purpose is to understand the fundamental process of APSP.

> **Question 14.4.** Is there an algorithm for APSP that runs in time $O(V^{3-\epsilon})$ for dense graphs (where $\epsilon > 0$)?

This has been intensely studied for the last 10-15 years; we don't know the answer.

One of the themes from today was that you could take algorithms for shortest paths, and convert them to algorithms for APSP. In general, we call that a **reduction** – reducing one problem to another. This is a fundamental tool in algorithms research, and we'll revisit it later in the course.

But you can also use reductions to understnad wehther or not there might be faster algorithms. APSP itself is a really useful target for reductions – there's lots of other problems we can solve if we can solve APSP. Tons of these reductions are known; so if there's a faster algorithm for APSP, then it would give faster algorithms for tons of other problems.

You can think of this either as evidence that we should study APSP really hard; or as evidence that there isn't a better algorithm for APSP (since we haven't found better algorithms for those problems either).

§15 Dynamic Programming

Dynamic programming is a very powerful technique for designing efficient algorithms. It was invented by Richard Bellman in the 1950s. It has tons of applications – to computer science, control theory, economics, computational biology, finance and trading, and so on.

§15.1 The Row Coin Problem

> **Example 15.1** (Row Coin Problem)
> We are given an array A representing values of n coins $a[0], \ldots, a[n-1]$. We want to maximize the value of the coins picked, subject to the constraint that no two consecutive coins can be chosen.

For example, if the coins are

$$5, 1, 2, 10, 6, 2,$$

then we can pick $5 + 2 + 6 = 13$ or $1 + 10 + 2 = 13$, but the optimal solution is $5 + 10 + 2 = 17$.

The idea of dynamic programming is to think about a process where we make decisions one at a time – do we include 5? Do we include 1? After each decision, we want to reduce the problem to a smaller instance of the same problem, and use recursion – which lets us compute our decisions.

In this problem, suppose we want to decide whether we want to pick 5 or not. If we don't pick 5, then we're solving the subproblem on 1, 2, 10, 6, 2. Meanwhile, if we pick 5, we can't pick 1; and then we have to solve the problem on 2, 10, 6, 2.

So if we know the answers for those two subproblems, then we can make the decision of whether to include 5 or not. An initial implementation:

```
    def coins(B):
        if len(B) == 1:
            return B[0]
        if len(B) == 2:
            return max(B)
        without = coins(B[1:]) # optimal if you don't choose first coin
        with = B[0] + coins(B[2:]) # optimal if you do choose first coin
        return max(with, without)
    coins(A)
```

§15.2 Memoization

Now we want to understand the structure of the input. If we call `coins(A)`, then we call `coins` on `A[1:n]` and `A[2:n]`. Then when processing `A[1:n]` we call it on `A[2:n]` and `A[3:n]`, and then when processing `A[2:n]` we call it on `A[3:n]` and `A[4:n]`. Note that all the inputs to `coins` are subarrays which are *suffixes* of the original array.

Instead of using the arrays as input, we can use the indices as input to `coins`.

```
n = len(A)
def coins(i):
    if i == n - 1:
        return A[i]
    if i == n - 2:
        return max(A[n - 2], A[n - 1])
    without = coins(i + 1) # optimal if you don't choose first coin
    with = A[i] + coins(i + 2) # optimal if you do choose first coin
    return max(with, without)
coins(0)
```

Let $T(n)$ be the time the algorithm takes on arrays of length n. Then we have

$$T(n) = T(n-1) + T(n-2) + \Theta(1).$$

This gives $T(n) = \Omega(2^{n/2})$, which is not good.

It is strange that the algorithm takes exponential time, when all of the inputs are integers between 0 and $n-1$. This means for some indices, we are computing `coins(i)` an exponential number of times.

A solution is **memoization**: a trick where we store the results of computations we've already done, so that we don't recompute things we've already computed.

```
n = len(A)
T = dict() # memoization table
def coins(i):
    if i in T: # have we already computed coins(i)?
        return T[i]
```

```
        value = None
        if i == n - 1:
            value = A[i]
        if i == n - 2:
            return max(A[n - 2], A[n - 1])
        without = coins(i + 1)
        with = A[i] + coins(i + 2)
        value = max(with, without)

        T[i] = value # store value of coins(i)
        return value
```

Now we will compute `coins(i)` once for every index (n times), and it takes $\Theta(1)$ per index. So the runtime is $\Theta(n)$. This is much better than the exponential solution.

§15.3 Pure DP

> **Definition 15.2. Dynamic programming** is recursion plus memoization, minus recursion.

The point is that recursion is nice, but can be inefficient – you don't want a recursive stack of length a million.

So we can actually get rid of the recursive stack. Instead of having T be a supporting factor, it'll be the main thing we work with. What we'll do is unwind the recursion.

When we call `coins(i)`, we have to call `coins(i + 1)` and `coins(i + 2)`. So we want to have `T[i + 1]` and `T[i + 2]`, and this value goes to `T[i]`.

So we can just unwind the recursion and get to the base cases, and then come back.

```
    def coins(A):
        n = len(A)
        T = [0] * n # initialize T; T[i] is the max value on A[i:n]
        T[n - 1] = A[n - 1]
        T[n - 2] = max(A[n - 2], T[n - 1]) # base cases
        for i in n - 3, ..., 0: # fill in T
            T[i] = max(T[i + 1], A[i] + T[i + 2])
        return T[0]
    coins(A)
```

> **Example 15.3**
> Run the algorithm on $A = [5, 1, 2, 10, 6, 2]$.

Solution. We start from the end, and fill in 2; then $\max(6, 2) = 6$, then $\max(10 + 2, 6) = 12$, then $\max(2 + 6, 12) = 12$, then $\max(1 + 12, 12) = 13$, then $\max(5 + 12, 13) = 17$. So the answer is $\boxed{17}$. □

So this is a very concise, beautiful algorithm.

§15.4 Framework

When discussing a dynamic program algorithm in problem sets (or exams), you should use the SRBTOT framework:

- Subproblems: define T – its meaning and dimensions. For example, $T[i]$ represents the maximum value of coins in $A[i:n]$, for $0 \leq i \leq n-1$.
- Relationship: describe the relationship between the entries of T – how the entries can be defined as a function of other entries. For example,
$$T[i] = \max(T[i+1], A[i] + T[i+2]).$$
- Base cases: identify the base cases and show how to initialize them (the entries that don't depend on other entries of T). For example,
$$T[n-1] = A[n-1] \text{ and } T[n-2] = \max(A[n-2], A[n-1]).$$
- Topological sort: argue that the dependency graph defined by the relations described is an acyclic graph, so that T can be filled iteratively according to a topological sort. For example, here $T[i]$ only depends on $T[j]$ with $j > i$, so we can compute backwards.
- Output: show how the output depends on the entries of T. Here the output is $T[0]$.
- Time: analyze the runtime of the algorithm. Here the table has n entries, and each takes $O(1)$ work, so the total work is $O(n)$.

§15.5 Use Cases

The key to know whether DP is a good strategy to use is the optimal sub-structure property:

> **Definition 15.4.** The **optimal sub-structure property** (OSSP) is that optimal solutions *contain* optimal solutions to sub-problems.

What this means is that it's okay to optimize sub-problems: there's no need to conisder suboptimal solutions to the sub-problems. This is why it was fine to call `coins` to the remaining array, and so on. Many optimization problems aren't like that: sometimes it's better to get a non-optimal solution to the sub-problem, rather than taking an optimal one and getting stuck.

For the coins problem, an optimal solution S that does not pick $A[0]$ contains an optimal solution to $A[1:n]$, while an optimal solution S that does pick $A[0]$ contains an optimal solution to $A[2:n]$.

Proof. Cut and paste: if we have a solution that doesn't pick $A[0]$, and it doesn't contain an optimal solution for the rest, then just get an optimal solution, and that's better. □

§16 Dynamic Programming II

As discussed last time,
$$\text{DP} = \text{recursion} + \text{memoization} - \text{recursive calls}.$$
Problems must have the optimal sub-structure property, which guarantees that we can use recursive calls to solve the problem.

Last time we discussed a thought process, where we first came up with a recursive solution – think about a sequence of decisions you have to make, and look at the recursive calls that give you the information needed to make the correct decisions. The next step was to recognize the structure of the inputs to the recursive calls (for example, that the inputs are always suffixes of the array, or consecutive pieces, or so on). Then you can use indices instead of the entire array. Once this is done, you can use memoization – this yields an efficient algorithm, but an inefficient implementation, and is impure DP. (Recursive calls are a bit inefficient.) Finally, the pure DP is when we remove recursive calls by unwinding the recursion, and filling the table T bottom-up – it's still defined recursively, but we fill it bottom-up and use it as the main actor.

Today we will do a couple of problems.

§16.1 Longest Common Subsequence (LCS)

Example 16.1

The input is two strings A and B, of lengths n and m respectively.

Find the length of the longest common subsequence of the two strings.

For example, take `A = HIEROGLYPHOLOGY` and `B = MICHAELANGELO`. Then we can pick `HELLO`, so the answer is 5.

Imagine walking through the strings from left to right. If the two first characters match, then we may decide to include the character and delete it from each of the strings. We may also decide to disregard the first character of A, or disregard the first character of B. If they do't match, then we can only disregard.

```
def lcs(A, B):
    n, m = len(A), len(B)
    if n == 0 or m == 0:
        return 0
    if A[0] == B[0]:
        # can include or not
        incl = 1 + lcs(A[1:], B[1:])
        not_incl_a = lcs(A[1:], B)
        not_incl_b = lcs(A, B[1:])
        return max(incl, not_incl_a, not_incl_b)
    else:
        not_incl_a = lcs(A[1:], B)
        not_incl_b = lcs(A, B[1:])
        return max(not_incl_a, not_incl_b)
```

You can actually argue that disregarding a match is never optimal. But here this only adds a constant factor of time so we won't make this argument.

Now note that all inputs to the recursive calls are suffixes of the strings. We can use that to replace the strings with indices in our solution – the same argument, but using indices instead of strings as input.

```
def lcs(i, j):
    if i >= n - 1 or j >= m - 1:
        return 0
    if A[i] == B[j]:
        incl = 1 + lcs(i + 1, j + 1)
        not_incl_a = lcs(i + 1, j)
        not_incl_b = lcs(i, j + 1)
        return max(incl, not_incl_a, not_incl_b)
    else:
        not_incl_a = lcs(i + 1, j)
        not_incl_b = lcs(i, j + 1)
        return max(not_incl_a, not_incl_b)
```

§17 Dynamic Programing and Shortest Paths

Shortest paths have the optimal subpath property – if we have any shortest path, it also gives a shortest path to every intermediate node. So it should be unsurprising that shortest paths problems can be solved using dynamic programming.

§17.1 SSSP on DAGs

Suppose we have a source node s, and some node v; we want the shortest path from s to v. The first decision we make is where to go from s. Then the subproblems are the shortest paths from each of those neighbors to v. The problem with this approach is that then the subproblems become from a node u to v. So we'd end up having to compute distances between every pair of nodes, which is more difficult.

But we can do this by making the *last* choice instead, meaning from which vertex we enter v. Then if v has neighbors u_i, the answer is $\min(d(s, u_i) + w(u_i, v))$.

S: The subproblems are that $T(v)$ is the shortest path from s to v, for all v in the graph.

R: We have

$$T(v) = \min_{uv \in E} d(s, u) + w(u, v).$$

(If this set is empty, then the distance is ∞.)

B: We have $T(s) = 0$.

T: Fill vertices in terms of their topological sort order on the DAG. Then by definition of the topological order, if there's an edge from u to v then we've already filled u by the time we get to v.

O: The output is all of the entries $T(v)$.

T: We fill every node in the graph; this takes $O(\sum 1 + |\text{Adj}(v)|) = O(V + E)$.

> **Remark 17.1.** This is similar to DAG relaxation, but instead of processing all outgoing edges, we're processing all incoming edges. But other than this, it's the same thing: all we're doing in DAG relaxation (every time we relax edges to a vertex) is finding the minimum. So these are essentially the same algorithm in principle.

§17.2 SSSP For General Graphs

The problem with general graphs is that there's cycles, so the dependency graph of our table would have cycles. The idea is to remember how many edges we have used as a second parameter in our table, so that we don't have cycles in the dependency graph – because if there's a cycle in the graph, when you return you've used more edges.

S: $T(v, k)$ for all $v \in V$ and $0 \le k \le n$ is the minimum weight of a shortest path from s to v using at most k edges.

R: We have

$$T(v, k) = \min(\min(T(u, k-1) + w(u, v)), T(v, k-1))$$

over all u with edges to v.

B: $T(s, 0) = 0$ and $T(v, 0) = \infty$ for all $v \neq s$.

T: Fill the table in increasing order of k.

Note that for every k, we're looking for every edge in the graph and relaxing it – computing the minimum is the same thing as relaxing all edges to that vertex. So this is exactly the same thing as Bellman-Ford, except that the natural way of implementing this is processing the edges in order by vertex, rather than in a random order.

O: For the same reason as Bellman-Ford, we output $T(v, n-1)$ for all v if there are no cycles; if there are negative cycles, we can detect them by seeing whether anything improves when $k = n$. (This is the same negative cycle detection as in Bellman-Ford.)

§17.3 Floyd-Warshall Algorithm for APSP

Previously we saw Johnson's Algorithm, which has a runtime of $O(V^2 \log V + VE)$. For dense graphs this is $O(V^3)$. Today we'll discover a *different* algorithm which matches this runtime – this algorithm was discovered separately by Floyd and Warshall.

Similarly to Bellman-Ford we'll look at restricted paths, but rather than restricting by number of edges, we'll restrict by which nodes are used as intermediary nodes.

So if we have u and v, we can't visit any node outside our subset, apart from u and v. We're going to start with the subset being small, and growing it to the entire set V.

Start by assuming there are no negative cycles; also number the nodes 1, 2, ..., n.

S: Our table stores $T(u, v, k)$, the minimum weight of a path from u to v using only the nodes u, v, and nodes in $[k]$ (for all u, v, and k).

R: We make a decision on whether to use k or not. If we don't use k, this is $T(u, v, k-1)$. Meanwhile, if we use k, then this is

$$T(u, k, k-1) + T(k, v, k-1).$$

So the answer is their minimum.

B: The base cases are when $k = 0$; then $T(u, u, 0) = 0$ and $T(u, v, 0)$ is $w(u, v)$ if $u \to v$ is an edge, and ∞ otherwise.

T: Entries for k only depend on entries for $k-1$, so fill the table in order of k.

O: $S(u, v) = T(u, v, n)$, since this corresponds to being able to use any node.

T: The table has size V^3, and each entry takes constant work. So the runtime is $O(V^3)$.

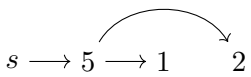
§17.4 The Other Direction

Here we saw that shortest paths can be solved with dynamic programming. But problems which can be solved with dynamic programming can also often be modelled with shortest paths.

Consider the row coin problem, where we have a set of coins, and we're trying to maximize the value of coins picked, with the constraint that we can't pick two coins which are consecutive.

Consider $[5, 1, 2, 10, 6, 2]$.

Draw a graph with two nodes s and t, and six nodes – each representing a decision on whether to pick a coin.



Here the $5 \to 2$ edge corresponds to picking 5, and it should have weight -5. The straight edges have weight 0.

§18 Greedy Algorithms

Greedy algorithms can be thought of as the naive cousin of dynamic programming.

§18.1 Activity Selection Problem

Example 18.1

The input is n activities $a_1, \ldots, a_n$, representing an interval $a_i = [s_i, f_i)$.

Our goal is to find the maximum number of non-overlapping activities.

This problem exhibits the optimal sub-structure property: if we have an optimal solution $S = \{b_1, \ldots, b_k\}$ and we define A' to be the subset of intervals which don't overlap b_1, then the rest of our intervals are an optimal solution for A'.

This means we can solve the problem by dynamic programming:

```
def asp(A):
    if |A| == 0:
        return 0
    a = A.pop()
    H = non_overlapping(A, a)
    incl = 1 + asp(B)
    not_incl = asp(A)
    return max(incl, not_incl)
```

The idea for greedy is that instead of analyzing all of the decisions and using all information, we're going to hope for the best: we order the activities in some way and choose the first one.

We'll order the activities by *end times*.

```
def asp(A):
    # assume A is sorted by end time
    if |A| == 0:
        return 0
    a = A.pop()
    B = non_overlapping(A, a)
    return 1 + asp(B)
```

In order to prove that a greedy algorithm works, we need the **greedy choice property**.

Definition 18.2. A problem has the **greedy choice property** if there is an optimal solution that includes the greedy choice.

(Note that this doesn't mean every optimal solution contains the greedy choice.)

In other words, the greedy choice can be extended to an optimal solution – or equivalently, we will never regret starting with the greedy choice.

For our problem, we want to show that there is an optimal solution which contains the interval ending first.

Proof. The idea is to start with an optimal solution and transform it to include the greedy choice.

Start with an optimal solution $S = \{a_1, a_2, \ldots, a_k\}$, where these are nonoverlapping and occur in that order. Now form S' by replacing a_1 with a, the interval which ends first.

Then we have $|S'| = |S|$. But S' consists of nonoverlapping intervals – since $a_2, \ldots$ start after a_1 finishes, and a finishes before a_1, then a finishes before any other interval starts. □

```
def greedu(A):
    if |A| == 0:
        return {}
```

```
        a = A.pop()
        B = non_overlapping(A, a)
        return {a} + greedy(B)
```

Claim 18.3 — Greedy is optimal.

Proof. Use induction; assume greedy works for sets of less than n intervals, then we want to show it works for n as well. So assume greedy works for B.

Start with an optimal solution S; we want to show the one returned by greedy is at least as good.

By the greedy choice property, choose an optimal solution containing the greedy choice.

Now S and the greedy solution both contain a, so we can delete all overlapping intervals with a to get B. By induction, we know greedy is optimal for B, so greedy without a is at least S without a; and by induction we are done. □

Remark 18.4. When you prove correctness for greedy algorithms, you should do something similar (use induction).

We can unwind the recursion here as well.

```
    def greedy(A):
        n = len(A)
        S = {}
        last = -1
        for (s, e) in A:
            if n >= last:
                S = S + {(s, e)}
            last = e
```

§18.2 Activity Scheduling Problem

Example 18.5

We have activities a_i with a duration t_i and a due time d_i.

Schedule all the activities to minimize the maximal lateness.

If something scheduled to finish at 3 now finishes at 6 (because we came to it three hours late), its lateness is 3.

If a_i is scheduled at time t, then

$$\ell(a_i) = \max(0, t + t_i - d_i).$$

For example, suppose we have $(1, 1)$, $(2, 2)$, $(3, 3)$.

If we schedule in reverse order, then $(3, 3)$ has lateness 0. We then schedule 2. Its duration is 2 so it finishes at 5; then its lateness is 3. Then 1 finishes at 6 instead of 1, so its lateness is 5. So the total lateness is

If we do it in correct order, we get 0, 1, and 3.

Claim 18.6 — There is an optimal schedule which schedules the activity with smallest due time first.

Proof. Start with an optimal schedule S. Let a be the activity with smallest due time.

Assume that a is not the first one. Suppose b is before a. Now swap a and b.

It suffices to show $\max(\ell(a), \ell(b))$ doesn't increase when we do this swap. Clearly a's lateness doesn't increase. Meanwhile, b's lateness is at most a's previous lateness, since b's due time is after a's. So $\ell'(a) \leq \ell(a)$ and $\ell'(b) \leq \ell(a)$, as desired. □

§19 Huffman Codes

§19.1 The Problem

Question 19.1. Suppose we want to compress Shakespeare. Suppose our symbols are $\{a, b, \ldots, z\}$ as well as a few other symbols – so we have at most 32 symbols. We want to encode each symbol as a binary string, so that our compression comes out as short as possible.

We can encode every symbol as a binary string of length 5. This is not much of a compression, since we're not doing much. But some symbols appear more than others – a, e, s, and t appear more than q or z. So it would be nice to give shorter codes to the more common symbols, and longer codes to the rarer ones – instead of using fixed-length codes, we're going to use variable-length codes.

More formally:

Problem 19.2. Our input is a string w over an alphabet Σ, where $|w| = k$ and $|\Sigma| = n$. We want to find codes $c(\sigma)$ for every $\sigma \in \Sigma$, and the code for w is the concatenation

$$c(w) = c(w_1)c(w_2)\cdots c(w_k).$$

But it's not enough to just find the lengths. For example, consider the case where we have eight letters. We could use the coes

$$a = 000, b = 0001, c = 1000, d = 1100, e = 11.$$

But now if we have the encoded string 00011000, we don't know whether it's aea or bc. This is bad – we want to disallow ambiguous codes, since those aren't useful.

To solve this, we'll use **prefix-free codes**: no codeword can be a prefix of another. (Here a is a prefix of b.) We can view these as trees, where the elements are the symbols to be encoded: we have a binary tree where all vertices have a 0 edge and a 1 edge; then to get our codes, we place our alphabet at the leaves, and read the sequence on the path to each letter.

Question 19.3. What's the best prefix-free code for w?

Notation 19.4. Given $x \in \Sigma$, we use f_x to denote the frequency of x in w, and $d_x = |c(x)|$ the length of the code word for x.

The code is specific to the string w – the string w is fixed.

We're going to look at prefix-free codes as trees: given a PFC T, we'll define the cost

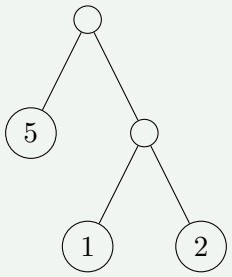
$$L(T) = \sum_{x \in \Sigma} f_x d_x = |c(w)|.$$

Note that we don't care about the order of symbols in w, only their frequencies.

So now we really want to find a tree T with lowest cost $L(T)$.

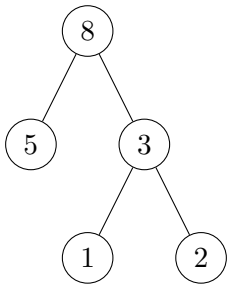
Example 19.5

Consider the frequencies



This corresponds to $5 \cdot 1 + 1 \cdot 2 + 2 \cdot 2 = 11$.

We can take the frequencies of the leaves, and propagate them upwards.



Lemma 19.6

$L(T)$ is the sum of f_v for all v other than the root.

Proof. Every leaf appears d_x times in the expression, since it contributes to every vertex on its path upwards. So every letter appears in d_x inner frequencies (including itself); □

Call this the **sum-tree view**. So now we can view the problem as:

Question 19.7. The input is the given n frequencies $f_1, \ldots, f_n$, and we want to find a binary tree that minimizes $\sum_v f_v$ over all non-root vertices of the tree.

Remark 19.8. We've ended up with a problem that looks very different from the original, by abstracting away everything irrelevant. So we end up with a combinatorial interpretation that's hopefully easier to work with. This is very common.

§19.2 Huffman's Algorithm

Given the frequencies, we're going to form the tree bottom-up: taking two frequencies and joining them together, and replacing with the sum; and continuing recursively.

So the meta-algorithm is:

```
until done:
    take f_x, f_y
    replace with f_x + f_y
    recurse
```

When we replace f_x and f_y with the sum, this corresponds to joining them.

With greedy algorithms, it's good to get some intuition about how optimal solutions look like; so we'll make a few simple observations.

> **Claim 19.9 —** Every optimal tree is complete.

(A complete tree is one where every inner node has two children.)

This is clear because if we have a tree where a node has one child only, we can just remove it and bypass it, giving a better code.

> **Claim 19.10 —** In any optimal tree, more frequent symbols should appear closer to the root.

This is intuitively clear: if $f_a < f_b$, then we should have $d_b \leq d_a$. If not, we could swap them and get something better.

The greedy choice is to choose the two smallest frequencies, and combine them. Intuitively, what we join at the beginning will appear deeper in the tree; and we want symbols which appear less frequently to appear deeper in the tree.

So now we have Hoffman's algorithm:

```
until done:
    take smallest f_x and f_y
    replace them with f_x + f_y
    recurse
```

For example, suppose we have (a, b, c, d, e, f, g, h) of frequencies $(3, 4, 4, 4, 2, 4, 3, 2)$. Now we combine $eh = 4$, then $ag = 6$, then $(eh)f = 8$, then $bc = 8$, and so on. By joining frequencies, we create an inner node, whose frequency is the sum and whose children are those two frequencies. We'll end up with a tree where all our symbols are leaves.

The first thing we want to do is prove the **Greedy Choice Property**: that there exists an optimal tree where the two smallest frequencies are siblings.

Start with an optimal tree, and we'll transform it into an optimal tree where the two smallest frequencies are siblings. Let's assume the two are $f_x \leq f_y$. So we have x and y somewhere in our tree.

Let a be the deepest symbol in the tree. Then a must have a sibling b – this is because if our tree is complete, and if b were an inner node then there would exist a lower node. WLOG we can assume $f_a \leq f_b$.

We know $f_x \leq f_a$ and $f_y \leq f_b$. Meanwhile, a and b are both the deepest, so $d_a \geq d_x$ and $d_b \geq d_y$. By our second observation, we can now swap a with x, and b with y.

So we end up with a tree where x and y, the two smallest frequencies, are siblings.

Proof of correctness. This is a recursive algorithm, so we can use induction – start with an optimal tree T, and we want to show Huffman produces a tree H with $L(H) \leq L(T)$. By the GCP we can assume in T that the two smallest frequencies are siblings. The Huffman algorithm also joins x and y; those are siblings there as well. Now we can look at T', the tree where we delete x and y; we do the same for H'. By induction we know $L(H') \leq L(T')$ (taking the sum-tree view). Then we add $f_x + f_y$ to both sides, and get $L(H) \leq L(T)$. So we are done. □

Remark 19.11. The proof of correctness once we have the GCP is easy – there are no ideas.

§19.3 Runtime

What we're doing is picking the two smallest things, joining them, and inserting a new frequency. So we should use a priority queue – because we want to find and delete the minimum (twice), and insert.

In the algorithm, we can also unfold the recursion. We do this $n-1$ times – we take the two smallest frequencies, join them, and insert the node. In the end we end up with a unique element, which is the root of the tree; that's what we return.

Building the heap is $O(n)$; deleting and inserting are both $O(\log n)$; and we do this n times. So the total is $O(n \log n)$.

Remark 19.12. This is an example of how we were able to describe the algorithm without thinking about implementation, but then we have nice data structures that we can use to implement efficiently.

§20 Subset Sum and Pseudo-Polynomial Time

§20.1 Sandwich Cutting

Problem 20.1. Suppose we have a long sandwich, and soneone has told you prices that they're willing to pay for various lengths of that sandwich (where everything is in integer intervals). You want to maximize the amount you can sell the sandwich for by cutting it up.

Given L (the length of your overall sandwich) and a table of values $v_\ell \in \mathbb{Z}_{>0}$ for $\ell < L$, the goal is to partition L into $\ell_1, \ldots, \ell_n$ such that $\ell_1 + \cdots + \ell_n = L$ and

$$v_{\ell_1} + \cdots + v_{\ell_n}$$

is maximized.

We can start by trying a greedy algorithm. You could try taking the most valuable sandwich length, but of course you can imagine that the most valuable sandwich length is the entire sandwich, worth 10, while the length-1 sandwich is worth 9.

The next thing we can try is to make a greedy choice based on value per unit length: the natural greedy choice is to choose the most valuable per unit length. Unfortunately, this doesn't work:

Example 20.2

Take $L = 7$, and the values $[0, 1, 10, 13, 18, 20, 31, 32]$ (for lengths 0 to 7).

The maximal $\frac{v_\ell}{\ell}$ is 6; the partition $6+1$ would give us $31+1=32$. Meanwhile, the partition $2+2+3$ gives $20+13=33$. So this greedy choice doesn't work.

So at this point, we give up and use the big hammer, which is dynamic programming.

In dynamic programming, we think about writing a recursive algorithm, and then removing the recursion.

The crucial decision we're making is whether or not to cut a sandwich of a given length. To know that, we want to know how much we'd get for the smaller sandwich.

We can describe this algorithm in SRBTOT:

S: We have a table of length L, where for $\ell \leq L$, $t(\ell)$ represents the maximum value from a sandwich of length ℓ.

R: We have

$$t(\ell) = \max_{1 \leq \ell' \leq \ell} v_{\ell'} + t(\ell - \ell').$$

B: The base case is $\ell = 0$, where $t(0) = 0$.

T: We can fill out the table in increasing order of ℓ (since we're only looking at sandwiches of shorter length).

O: Our output is $t(L)$.

T: The runtime is $O(L^2)$, since we have $L + 1$ entries to fill and each takes $O(\ell)$ time.

§20.2 Runtimes

Definition 20.3. An algorithm is **polynomial time** if its worst-case runtime is bounded above by some polynomial in its input size – $T(n) \leq O(n^c)$ for some constant c (which cannot depend on n).

So $O(n)$, $O(n^2)$, $O(n^{10})$ are all polynomial time; but $O(n^{\log n})$ is not polynomial time, and $O(2^n)$ is certainly not polynomial time.

This zooms out from details about running time and only makes big picture assessments. Thinking about whether algorithms run in polynomial time is useful.

Our algorithm for sandwich cutting runs in $O(L)$ time.

The input size, formally speaking, is the number of words in the word-RAM model occupied by the input. For sandwich cutting, we have L occupying one word, and v occupying $L + 1$ words. (Since we're always assuming that our integers fit into a single word.) So our input size is $\Theta(L)$. Since L^2 is polynomial in L, then this is polynomial time.

In fact, we have seen a lot of polynomial time algorithms – most (and maybe all) of the algorithms we've seen are polynomial time.

§20.3 Subset Sums

Problem 20.4. The input is a positive integer L, and some sequence $A = \{a_1, \ldots, a_n\}$ of positive integers. We want to figure out whether there's a subset of the integers A which sums to L.

The output is yes or no, returning whether there exists $A' \subseteq A$ such that

$$L = \sum_{a_i \in A} a_i.$$

We can think of this as whether we can cut the sandwich up into allowable lengths. (If you reintroduce values, then you get the knapsack problem.)

So this is a decision problem – the goal isn't to output an optimal path, but to decide.

Example 20.5

Suppose $A = \{2, 5, 7, 8, 9\}$.

Then for $L = 21$, we have $21 = 5 + 7 + 9$, so the answer is yes. Meanwhile, for $L = 25$, the answer is no.

It *seems* like we can use a dynamic programming approach similar to the sandwich cutting problem.

We want to decide whether to include 2 or not; to do this, we want to see whether the remaining has a subset summing to $L-2$. If it does, we should include 2; if it doesn't, then we shouldn't include 2.

(Remember that we don't need to actually keep track of the subset.)

S: We have a table $t(\ell, i)$ for $0 \leq \ell \leq L$ and $1 \leq i \leq n$, where $t(\ell, i)$ is `True` iff we can find a subset of $\{a_i, \ldots, a_n\}$ summing to ℓ.

R: For $t(\ell, i)$, we consider whether to include i or not: this gives us

$$t(\ell, i) = t(\ell, i+1) \texttt{ OR } t(\ell - a_i, i+1).$$

(We can ignore what happens when $\ell - a_i$ becomes negative.)

B: The base cases are $\ell = 0$, for which this is always `True`, and $i = n$, which is true iff $\ell = 0$ or $\ell = a_n$.

T: Fill in decreasing order of i.

O: The output is $t(L, 1)$.

T: Filling in a single entry takes $O(1)$ time. There are Ln entries, so this takes $O(Ln)$ time.

§20.4 Pseudopolynomial Time

> **Question 20.6.** Is this polynomial time?

The input size for subset sum is $n+1$. But L can be big – all that we know about it is that it fits in a single word (we know $L \leq 2^w$ where w is the word size). We don't have any constraint on w – we knwo w is at least big enough that we can store addresses to all the cells in our word-RAM memory. (Words have a certain size and can store integers, which are pointers to other places.) We know there's at least $n+1$ cells in the word-ram, so we know $w \geq \log(n+1)$, but we can't get an upper bound on w, so we can't get an upper bound on L.

So $O(Ln)$ *sounds* good; but L might be really big. In fact, it's not unreasonable to imagine that $w \approx n$, in which case L may be 2^n, and the algorithm would have *exponential* running time.

So for the first time, we've used a pretty powerful algorithmic technique (unlike the exponential algorithms we saw before, which were silly), and we're still getting an algorithm that might run in exponential time.

There is actually a name for algorithms "like this" – if we give the algorithm inputs whose numerical values are not huge, then the runtime really is reasonable.

> **Definition 20.7.** An algorithm has **pseudopolynomial time** if its runtime is bounded above by a polynomial in its input size, and the magnitude of the maximum number in its input.

So this means $T \leq O((nL)^c)$, where n is the input size and L is the maximum integer in the input.

Our subset sum algorithm is pseudopolynomial time. This achieves polynomial running times when $L \leq O(n^a)$.

Earlier, we talked about radix sort – radix sort was a linear time sorting algorithm *if* the integers it was given to sort were polynomially bounded in size.

> **Remark 20.8.** The reason sandwich cutting was polynomial time, and this wasn't: we're using basically the same algorithm, but in sandwich cutting we forced the input to be long enough – we *made* the input size equal to L (or rather, the sandwich length). So this let us compare the runtime to the input size. In subset sum we didn't do that.

§20.5 More About Runtime

A natural question we might ask ourselves, having seen this contrast, is – did we really need this? Maybe there's a better algorithm for subset sum.

> **Question 20.9.** Does subset sum have a polynomial time algorithm?

The answer is that nobody knows; this is an enduring mystery, and it extends to variants of almost every problem we've seen in this course (and many others).

In order to be able to talk about that question in the scope that it deserves, we can discuss how to think of all these problems in a unified way.

One way to think of almost all our algorithms is that they're ways of efficiently searching through exponentially large search spaces.

For example, here the search space was the partitions of an integer L. An input of length n defines some search space of size around 2^n.

For example, in sorting, the search space is all possible permutations of a list. In Tarjan, the search space is all set partitions of the vertices of that graph into subsets. For shortest path algorithms, the search space is all possible simple paths in the graph. In all these cases, what our algorithms do is find an optimal object in an exponentially large space, and they do it without looking at all the objects in that space. (If we have an exponential search space and we run in polynomial time, we certainly didn't look at everything in the search space.)

That's the big idea behind this definition of polynomial time – it captures the idea that your algorithm is doing something really special, since it's finding its way through a huge search space without looking at everything in the search space.

We've seen techniques for doing this, like dynamic programming.

The much bigger question, hiding behind this one, is whether that's always possible – for *any* problem where length n inputs define a big search space, can you *always* navigate that search space efficiently? Or is subset sum an example with an exponentially large search space where we can't avoid looking through everything in it?

§21 Complexity

Last lecture, we started pulling the camera back – we've seen many algorithm design techniques, and now we'll see how they fit into the broader landscape of theoretical computer science. Last lecture, we saw an algorithm which used dynamic programming and took *pseudo-polynomial* time. We ended with a couple of questions:

- Does subset sum have a polynomial time algorithm?
- We spent a long time discussing graph algorithms – what if we asked for *longest* paths? Given two vertices u and v, we want to find the longest simple path between them. Does this have a polynomial time algorithm?

A lot of our algorithms in this course have in common an interesting structure – every input of size n defines a search space that's usually exponentially large (2^n or bigger). Somehow, our algorithms always navigate that search space, and our polynomial time algorithms clearly do this without looking at every object in the search space.

For example, with sorting algorithms, the search space is permutations; with shortest paths, it's all simple paths in the graph; with strongly connected components, it's all partitions of a set of vertices; and so on.

It turns out that we don't really know a way to talk about these very concrete problems, without talking about this big abstract question. But we don't have a mathematical way of talking about this yet. So we'd like to make it rigorous.

§21.1 Decision Problems

We need language that lets us talk generically about computational problems, rather than one problem in particular. To do that, we'll restrict to one type of problem:

> **Definition 21.1.** A computational problem is a **decision problem** if it has yes or no outputs.

We phrased subset sum as a decision problem. Most algorithms we talked about in this class have been for other problems – **search problems** (where the goal is to *find* an object, like a sorted permutation), **optimization problems** (where the goal is to find the value of the optimal object, like the length of a shortest path).

You can argue that search and optimization are the really useful things; but it's easier to talk mathematically about decision problems. And it actually turns out that we're not losing much by doign this – decision problems typically capture search and optimization problems. This is somewhat counterintuitive, but it's true: for example, consider shortest paths. If we have an algorithm for a decision variant, we can use it to construct an algorithm for the search or optimization version.

This idea of using an algorithm for one problem to construct an algorithm for another is **reduction** – which will be a key theme for today.

> **Definition 21.2.** A computational problem A **reduces** to B if the ability to solve B gives you the ability to solve A.

This is an informal definition; we'll see a formal one later.

We've seen several examples of reductions in this class: for example, all-pairs shortest paths could be reduced to single-source shortest paths using Johnson's algorithm.

> **Claim 21.3 —** Search and optimization problems reduce to decision problems.

Example 21.4

Consider shortest paths (as an optimization problem), and reduce to a decision problem.

We need a shortest-paths decision variant: the input is graph G, vertices u and v, and an integer k. As output, we should output yes if the shortest path distance is at most k, and otherwise output no.

If we have an efficient algorithm for the decision variant, then we have an efficient algorithm for the optimization problem, by binary search. We start by guessing some shortest path length, then if we get `yes` we cut the length in half and try again, or else we increase.

This example isn't specific to shortest paths – if we want to find the optimal value of a number, and we have a subroutine that tells us whether it's bigger or less than any k, then we can *always* do binary search.

So optimization reduces to decision via binary search. You can do something similar for search problems, but this requires more mathematical scaffolding.

§21.2 Complexity Classes

In this class, we've so far thought of efficiency and runtime as something associated with an algorithm. Now we're going to put problems first, rather than algorithms first.

Definition 21.5. P is the class of all decision problems with polynomial time algorithms.

So polynomial time was the property of an algorithm, and now P is the class of *problems* that have polynomial time algorithms. Here we're only discussing decision problems.

For instance, shortest paths is in P.

We can consider the universe of all decision problems. Inside it, we have a circle representing P.

P is called a **complexity class** – a complexity class is a subset of decision problems sharing a property.

You can define other complexity classes:

Definition 21.6. EXP is all decision problems with $2^{\text{poly}(n)}$ time algorithms.

EXP is bigger than P.

You can also define complexity classes based on other things than time. For example, you can define them based on how much *memory* is required:

Definition 21.7. PSPACE is all decision problems with algorithms using a polynomial amount of memory.

It turns out that PSPACE actually sits inside EXP.

One thing complexity theory is about is understnading the relationships between complexity classes. Today we'll only see a tiny sliver – about the exponentially large search space question.

Remark 21.8. When we have our TCS hat on, we like to take P as our *definition* of efficient computation. Efficient computation is this squishy thing – what do we mean if an algorithm is efficient, or a problem is efficiently solvable?

There are some problems: what if there's a computational problem with a n^{8000} time algorithm? That's officially in P, but is that really efficient? Or what about a problem with a $n^{\log\log\log n}$ algorithm? This isn't in P – is that *less* efficient than the n^{8000} time one?

This is a natural bug, but it turns out that in real life, this doesn't really come up – when a problem has a polynomial-time algorithm, it almost always has one that runs reasonably (for example, n^2 or n^3).

And this class P is robust – running time is defined as the number of steps you need to solve your problem on the word-RAM. Why the word-RAM – when we code, we don't actually code on the word-RAM, and we don't care about whether what you can do in constant time on Python is the same as on word-RAM. So what if we redefined polynomial time to depend on Python rather than the word-RAM? The point is that P is robust to these changes – word-RAM can simulate Python in polynomial time, and Python can simulate word-RAM in polynomial time.

§21.3 Exponentially Large Search Spaces

Question 21.9. How are we going to capture the idea of a problem with an exponentially large search space?

One useful intuition is that if we have a problem of this nature, there's *always* a brute force algorithm, which looks at every object in the search space, and decides whether this is the special object or not.

So one characteristic of exponential-size search space problems is they have brute-force algorithms, where you enumerate over all $c \in$ Search Space and check whether c is the thing we want. For instance, in subset

sum, the search space is all subsets of a given set of integers. So we could enumerate over all subsets, and check whether they sum up to the target sum. For longest paths, we could enumerate over all possible paths.

Implicit in this intuition is that you must be able to *check* if a given object in the search space is the one you want. We need to be able to check efficiently, meaning in polynomial time.

So this intuition about what it means to have a brute force algorithm is what we'll use to make a mathematically rigorous definition.

Definition 21.10. NP is the class of decision problems A such that there exists a polynomial time algorithm V which takes input as some instance x of A and a certificate $c \in \{0,1\}^{|x|^t}$ for some constant t (a boolean string that's polynomial in the length of x), and outputs: if x is a `YES` instance of A, then there exists *some* certificate which you could hand V such that $V(x,c)$ is `YES`; if x is a `NO` instance, then for *every* possible certificate, $V(x,c)$ is `NO`.

Remark 21.11. NP stands for non-deterministic polynomial time. It does *not* stand for not polynomial. V in the definition stands for verifier.

Example 21.12

Subset sum is in NP.

Proof. We can construct a verifier. The intuition is that our verifier should be related to the exponential search space – it's supposed to be this thing that checks the things in the search space.

In this case, x is a set of integers $\{a_1, \ldots, a_n\}$ and a target sum. It also gets a certificate, which is a boolean string describing the subset – if the ith element is a 0, this means a_i is not in the subset, and if 1, it is. So we can interpret certificates c as a subset. Then we output `YES` if

$$\sum_{i \in c} a_i = L,$$

and `NO` otherwise.

So our verifier is just checking whether something is a good subset. Our verifier is checking, for each thing inside the search space, whether it's good or not.

If x is a yes instance, then when you feed in that subset as the certificate, the verifier will output yes. Meanwhile, if x is a no, then no matter what subset you give it, it'll check that subset, see that it doesn't add up, and will output no.

So this shows subset sum is in NP. □

In fact, longest paths is also in NP.

Now that we've seen an example, we want to check whether we've captured this intuition about the squishy class of exponential-sized search spaces with brute force algorithms.

Claim 21.13 — NP is a subset of EXP – every decision problem in NP has an exponential time algorithm.

We'll interpret this algorithm as the brute force algorithm.

In the formal algorithm, the space is the space of certificates. So we brute-force search over the certificates.

Proof. Take $A \in NP$. Now we can construct an exponential-time decision algorithm for A on input x: for all strings $c \in \{0,1\}^{n^t}$, we now perform brute force enumeration – we check whether $V(x,c)$ outputs yes. If we ever get a yes, then we output yes; otherwise we output no. Each call to the verifier is polynomial time. Meanwhile, the certificates represent the exponentially large search space: there's 2^{n^t} search spaces. So the runtime is at most 2^{n^t} times some polynomial, which is $O(2^{O(n^t)})$. Here $n = |x|$.

So this algorithm has exponential time, which means A belongs to EXP. □

So that's some evidence that our mathematically formal definition has done an acceptable job of capturing the idea of exponential search spaces.

Claim 21.14 — P is a subset of NP.

Proof. We want to check that if a problem has a polynomial-time decision, it also has a polynomial-time verifier. If $A \in P$, we can build a verifier as follows: $V(x,c)$ just ignores the certificate, runs the algorithm for A, and if x is a yes input for A it outputs yes; if x is a no input for A, then it outputs no. This satisfies the input-output property, and A was polynomial time, so V is as well. □

So then NP lives somewhere between P and EXP. Now we have a mathematical way to capture our big question about navigating exponential search spaces efficiently: is NP more like EXP, or more like P?

Question 21.15 (Big Question)**.** Is $P = NP$?

This is perhaps the biggest unsolved problem in theoretical computer science. It is an open problem – we're not going to answer this big question, because we don't know the answer.

But nonetheless, we would like some way to use this mathematical framework that we've developed, to give some kind of answer to the more concrete questions we've started with – does subset sum or longest paths have a polynomial-time algorithm? We can't quite answer this big question, but maybe we can still get some ideas about the smaller, more concrete questions.

§21.4 Concrete Questions

It is strongly believed (by most members of the computer science community) that $P \neq NP$. We'll return to why at the end of the lecture. We can't mathematically establish one way or another, but it's strongly believed.

If you believe $P \neq NP$, then these questions about subset sum and longest paths become a question fo which belong to P, and which are probably outside. We want a way to tell whether something's probably outside P or not.

So we'll return to the power of reductions.

Definition 21.16. A problem A **reduces** in polynomial time to a problem B if there exists an algorithm, in polynomial time, such that for all instances of A, if $A(x)$ is `yes`, then we can feed x into the algorithm, feed the *output* of that algorithm into B, and get the right answer; while if $A(x)$ is no, then we can do the same and get no.

So what this means is if we get a polynomial algorithm for B, then we get one for A.

Another way to interpret A reducing to B is that B is at least as hard as A – if A is hard, B can only be harder. This is notated as $A \leq_P B$.

Question 21.17. How can we figure out whether a problem is probably outside or inside?

Definition 21.18. A problem B is NP-hard if $A \leq_P B$ for all $A \in NP$ – meaning *every* problem in NP reduces to B.

This means B is at least as hard as every problem in NP. This sounds crazy – can there be an algorithm that solves *every* decision problem?

The answer is yes – NP-hard problems exist! NP seems really hard, so maybe that algorithm has to be really slow though ($2^{2^{2^n}}$ or similar). We want to avoid that.

Definition 21.19. B is NP-complete if it's NP-hard and in NP.

Then it needs a polynomial-time verifier and exponential-time algorithm.

The amazing thing is that NP-complete problems exist – there are problems in NP which are as hard as the rest of NP – if you could solve that one problem, then you could solve every NP problem. Not only do NP-complete problems exist, but we've seen one:

Fact 21.20 — Subset sum and longest path are NP-complete!

In fact, so is a host of other problems – traveling salesman, boolean satisfiability, graph coloring. Tons of problems turn out to be NP-complete. We don't have space im this course to discuss how to prove that – take 6.046. But the fact that thousands of people have spent millions of hours over the last decades, trying to design algorithms for these particular problems, is evidence that no one has a polynomial time algorithm – if subset sum had one then all of NP would, but despite decades of effort we haven't found any.

So we take a problem being NP-complete as strong evidence it lies on the outside.

The punchline for today is that reductions are a powerful way to understand relationships, and it's possible to rigorously mathematically capture the class of problems with exponentially sized search spaces; and there's strong evidence that some such problems lack polynomial time algorithms.

§21.5 The Punch Line

We've been pulling the lens back, and instead of individual problems, we've been developing language to talk about broad classes of problems all at once. We talked about the class P – problems with polynomial-time algorithms to solve – and NP – problems which have polynomial-time *verification*. This is a way of intuitively capturing the idea of a problem requiring us to search through an exponentially large search space.

We've discussed the question $P = NP$ – we haven't solved this mathematically, but it's strongly believed $P \neq NP$, and we can use this to try to figure out which elements of NP are probably not in P. So we developed the idea of NP-completeness to describe this: a problem A is NP-hard if every problem in NP can be reduced to A, and it's NP-complete if it's NP-hard and in NP.

If $P \neq NP$, then NP-complete problems are not in P – if some NP-complete problem had a polynomial time algorithm, then via reduction, every problem in NP would have one. So NP-completeness is a tool to give evidence that a problem is sitting outside of P.

To show that a problem A is NP-complete, you show that $A \in NP$, and find some NP-hard B and construct a reduction $B \leq_P A$. Then since B is NP-hard, every problem in NP reduces to B, and therefore to A.

We won't discuss in this course how to construct such reductions.

Remark 21.21. This obviously doesn't work if you don't know any NP-hard problems. To get the first NP-hard problem, you have to do something else. What you can do is construct a computational problem that has to do with verifiers – if your goal is to find something that everything in NP reduces to, and all you know is that they have verifiers, your problem should be about verification. So you can build such a problem about verification, and then you can reduce other stuff to that problem. In 6.046 or 6.045, you will see this.

You can view a lot of other things through NP-hardness – protein folding (understanding what shape a protein folds into) is NP-hard, which leads to a mystery about how come nature can fold proteins.

§22 Computability

Let's pull the lens out even further. Up until last lecture, every problem we've talked about had *some* polynomial time algorithm, and we saw last lecture that there's some which don't. But still, every problem we've talked about in this course has *some* algorithm.

What it means for a (decision) problem to have an algorithm, intuitively means that there's a Python problem which will correctly get the yes or no answer. Here we don't care about the running time.

Question 22.1. Do all problems have algorithms?

That's a vague philosophical question, so we'll try to make this a bit more precise. Take 6.045 to see the full details.

First, we need to decide what a problem is.

A **decision problem** has a yes/no answer for every input.

Definition 22.2. An **input** is a binary string – inputs come in $\{0,1\}^* = \{\varnothing, 0, 1, 00, 01, \ldots\}$.

We've previously thought of an input as a series of words on the word-RAM; this is the same thing since we can chop up the binary string into words.

Definition 22.3. A **decision problem** is a map $A : \{0,1\}^* \to \{Y, N\}$.

Definition 22.4. An algorithm is a Python program which takes as input a binary string and outputs yes or no – essentially, a finite piece of Python code.

(There is a more precise definition, but we won't go into that.)

Now we've made our question more rigorous.

Remark 22.5. This has a pretty illustrious history. This course has been about the mathematics to describe computation. In the 20th and 21st century, there was a lot of mathematics developed to describe computation.

But in the 1800s, the thing driving the development of mathematics was physics. In the 1860s, one of the hottest topics was Fouriers series, since you use them to solve the heat equation. These are infinite sums of sines and cosines.

It turns out to understnad properties of Fourier series, you had to understand questions about the real numbers that no one knew how to answer – even questions like how many real numbers are there. Those questions, motivated by physics, led to the formation of a mathematical program led by Hilbert, to see whether there could be a finite set of axioms on which all of mathematics could be based. Then we could start deriving theorems from our axioms until we answered the questions.

This question fell out of that program – you can imagine that at least for any problem which is mathematically described (including all of the problems we've seen in the class), if there's a set of axioms and a procedure for deriving answers from the axioms, then you would have a program to solve everything; and our answer would be yes.

So this question was of intense interest in the early 1900s, even before computers. But the mathematics developed to answer it actually led to the beginnings of computer science.

So there's a virtuous cycle where questions about physics led to questions about the deep foundations of math, which led to the foundations for computer science.

The answer is no. Not all problems have algorithms. So Hilbert was wrong, and this shook the world.

Proof. This proof uses a technique called **diagonalization**.

Suppose otherwise, so every problem has an algorithm. Because algorithms are finite pieces of Python code, we can put them in alphabetical order. That puts the decision problems in a list (by the alphabetical order of their algorithm).

So we have a table of decision problems A_1, A_2, A_3, ...; and every decision problem appears in this list.

Now we're going to do a thing that comes sort of out of left field. Write these problems in the rows of our table.

Meanwhile, write the list of all possible inputs as columns, in the order $\varnothing$, 0, 1, 00, ...

Then fill in the entries with the answer to the corresponding decision problem on the corresponding string. So we have a table of yes and no.

We're going to get a contradiction by building a decision problem that isn't in the table – we just showed that all decision problems are in the table, so this would be a contradiction.

Look at the diagonal. Define A to be the problem where $A(x)$ is the *opposite* of whatever appears in x's entry on the diagonal. So if x is the ith string, then $A(x) = -A_i(x)$. According to the definition of decision problems, this is a valid decision problem.

But it's not in the list – because if it were the ith element of the list for some i, then it disagrees with that element on the ith input. Essentially, what we did is we ensured that our decision problem disagrees with every decision problem *somewhere*.

So A is not in the list! Contradiction. □

Remark 22.6. An algorithm is a finite piece of Python code. You can list these first by length, and then by alphabetical order. We don't actually need an algorithm that can do this – we just need to know that the ordering *exists*.

This is somewhat unsatisfying. It does prove, in some strange abstract sense, that there is some decision problem without an algorithm. But it's very abstractly described. And it's unclear whether it gets to the heart of what Hilbert asked – whether this funny weird thing we described even counts as a mathematical thing.

But we can actually describe a very concrete decision problem which does not have an algorithm.

Definition 22.7. `HALTING` is a decision problem. As input, it takes a program (a Python program) and a binary string. As output, we output yes if $P(x)$ halts – meaning it eventually produces an output and stops running – and no otherwise (if P enters an infinite loop, starts enumerating the digits of π, or anything similar).

So we actually have a mathematical problem. And this problem is actually useful – if you're writing a compiler, it would be useful to know whether a program is going to run infinitely or not.

Theorem 22.8

`HALTING` has no algorithm.

THis theorem was proved in 1936 – you'll notice that this is *still* before there were any computers. The first computers were built in the early 1940s with vacuum tubes. So the foundation of CS came before computers. And there's evidence that the teams who built the computers were aware of this theorem, and understood the idea inside it – the idea is of building a computer which runs the code given to it as input. Before the idea was developed, computing machines existed – each machine (like a calculator) solved one computational problem. Implicit in this idea is the idea that you can build a machine (an interpreter, or operating system) that runs an arbitrary program. And you need that in order to build a computer!

SO even though the answer is a negative, the ideas in the proof were sinilar to the ideas needed to build a computer for the first time.

The theorem was also proved by Alan Turing. Turing proved it in his 20s, this is just one of his contributions to humanity. (After 1936 he worked for the British codebreakers and they broke the Enigma code; it's widely believed that this gave them intelligence that they allowed the end of WW2 long before it would've otherwise, so his work saved millions of lives. That work remains largely unknown since it was classified. So Turing was a war hero who saved millions of lives. He contributed a lot to the foundations of CS. He was also gay at a time when that was not acceptable and was punishable by the government; he was severely punished by the government, and it's widely believed that this led to his death by suicide in his 40s. Who knows what contribution of science we missed because of that kind of oppression?)

Proof. We're going to do something a lot like that diagonal argument. Suppose otherwise, so `HALTING` has an algorithm.

Now we're going to build another program P on input x. First the program parses x and checks whether it's a valid Python program. The mental switcheroo here is that program inputs themselves can be interpreted as programs – this makes sense to us now (with characters and ASCII), but is a mental switcheroo.

So you parse x to get a Python program P_x (if you don't get one, just output no). Then we assumed we had an algorithm for `HALTING`, so run the algorithm for `HALTING` on (P_x, x). So this returns yes or no in finite time.

Now we can switch the answer. So if `HALTING` gave us yes, then just enter an infinite loop. Meanwhile, if the answer is no, then halt and output yes. (Or no. It doesn't matter.)

So the behavior of P is the opposite of what P_x is on x – if P_x halted (on x), then P will not halt; while if P_x did not halt, then P will. So this is the same switch thing from before, but done algorithmically.

Now let's get a contradiction. Let y be the code for P, meaning the binary string which parses to the program P. Now consider what happens when we run $P(y)$.

This is very self-referential. We want to see whether $P(y)$ halts or not.

Suppose $P(y)$ halts. Now we can trace through what happens. P parses the input y and gets back P; so then it runs halting on (P, y). We assumed that it halts, so halting will return yes, and we'll enter an infinite loop. That means $P(y)$ does not halt. That's a contradiction.

So $P(y)$ does not halt. Again, when we parse y we get the program P. Then we run halting on (P, y). We just said it doesn't halt, so halting will give us a no answer. But then it halts.

So neither of these can be the case, which means we have a contradiction.

So there cannot have been an algorithm for halting, adn halting is undecidable! □

Remark 22.9. Why did people let this bother them, when these problems are weirdly self-referential? But it's not just the halting problem that's undecidable; it turns out that basically any interesting property of algorithms is impossible. There are problems you'd actulaly want to know answers to, which are undecidable.

Example 22.10

`TOTALITY` is a problem. Its input is a program P, and its output is yes if P halts on all inputs, and no otherwise.

Imagine again that you're a compiler writer. Youd like to tell them if they made some error that would let the code loop forever. To do that, you want to solve totality. But it turns out totality is undecidable.

To prove this, we can use reductions. We'll reduce HALTING to TOTALITY. (Now we don't need the reduction to run in poly time – we just need to know that the ability to solve TOTALITY gives us the ability to solve HALTING.)

Proof. We'll construct a reduction HALTING to TOTALITY. Suppose we're given (P, x) as input to HALTING. Now we'll construct a new program Q on input y. What Q does is that it ignores y, and runs $P(x)$. Clearly you can implement this using an algorithm – if you give me P and x, I can spit out the code for Q.

Now give this code for Q to totality. Then Q is a yes instance for totality iff (P, x) is a yes instance for halting. So we've reduced halting to totality, which means totality is also undecidable. □

Example 22.11

`PROGRAM EQUIVALENCE` (PE) takes two programs P and Q as input. It outputs whehter P and Q have the sane functional behavior (even though their code may be different) – yes iff $P(x) = Q(x)$ for all x, and no otherwise. (This also means if P doesn't halt, then Q shouldn't halt, and vice versa.)

This is the easiest to see why you would want it. When you write a compiler, very often you take a piece of code, and you want to optimize it. Compilers can sometimes propose possible optimizations. You want to check that the optimization preserves functionality, and then this is exactly what you want to decide.

Proof. Now we have another problem, so we'll actually reduce from totality – we'll show totality reduces to program equivalence.

Assume we have something that solves program equivalence, we'll use it to solve totality.

Suppose our input to totality is some program Q. Now we're going to take Q and spit out a new program Q', which does the following: on input x, Q' first runs Q, and if Q halts, then output yes. (We ignore what Q output.) Otherwise Q' doesn't halt either.

Now create R on input x, which ignores x and just outputs yes. So R is just trivial.

If Q is total, then Q' always outputs yes. So Q' is also the constant function.

Meanwhile if Q is not total, there's some x for which Q' doesn't halt, so it's not the constant function.

So Q' is functionally equivalent to R iff Q is total. This proves that totality reduces to program equivalence. □

§22.1 Final Remarks

Like NP-hardness, it seems at first discouraging. But it turns out there's often ways around uncomputability. Once you know that you shouldnn't try to solve the problem for every input, you can still try to solve it for classes of input. There do exist optimizing compilers – these cleverly work around issues like totality and programming equivalence. So these concepts can inform you as you try to solve an actual design challenge – it's not that all hope is lost.

§23 Parallel and Asynchronous Computation

These days, when you have a CPU, it's probably a multicore processer – there are multiple computing cores on the same chip. Each has their own cache, and they're connected via a bus to a shared memory.

The first commercial multicore chip was the SUN "Niagara 1" which has 8 cores and memory attached. This chip was a sensation when it came out.

In an ideal scaling world, if you have a multicore with two cores, you write your code nicely parallelized, and you get 1.8 speedup. Then when you have four, you break it up and get 3.6 speedup, and similarly 7 for 8.

But unfortunately, it's not that simple. What actually happens is that we have to coordinate these processes. This is a big problem, which is why writing this kind of code is complicated. Paralleization and synchronization require great care.

In sequential computation, you have threads working on a shared memory, with shared objects. This class we designed all kinds of data structures to do algorithms; now imagine we're doing the same thing, but multiple threads are accessing our stuff at the same time.

§23.1 Problems

The first problem is that this is asynchronous – we can't really predict how long an operation will take. This is because of cache misses, page faults, and scheduling quantum being used up; then a thread can disappear and appear, so we can't rely on all of them taking steps together. Scheduling quantum is that when you have a process on your machine (say, you're running two windows), the operating system allocates some time for each to run, and when it's over, takes the process away.

So we can't really predict when things can happen.

§23.2 Parallel Primality Testing

We want to print the primes from 1 to 10^{10}. We're given a ten-core multicore processor, with one thread per core, and we want ten-fold speedup.

One way to do this is by **load balancing** – split the range 1 to 10^{10} into ten intervals of length 10^9, and each thread executes its range.

```
    void primePrint {
        int i = ThreadID.get();
        for (j = i*10^9 + 1, j < (i + 1)*10^9 + 1, j++) {
            check if prime and print;
        }
    }
```

But higher ranges have fewer primes, and larger numbers are harder to test. So it's hard to guess how long it will take; the work is unevenly split.

SO what we can do about htat is **dynamic load balancing** – we balance the load based on what the processors are achieving, rather than dividing ahead of time.

The idea is to have a **shared counter**. Each thread will go to this shared counter, get a number, check if it's prime, and then go and get another number until they're done.

```
    Counter counter = new Counter(1);

    void primePrint {
        long j = 0;
        while j < 10^10:
            j = counter.getAndIncrement();
            if (isPrime(j)):
                priint(j);
    }
```

WHat `getAndIncrement` does is what it sounds like – you get a number, and then move up the counter so that the next guy gets the next number.

To do this, we need this shared counter object. Now the work is balanced perfectly.

The shared counter is an integer sitting in memory. We have our separate cores with their own caches, and their own local variables which are inside the processor and cache. We then have a bus (a communication path between the processors and memory). The processors send information and get information back from the memory.

So the shared counter is inside the shared memory that everyone can reach – if it were in one of the caches, then the other guys couldn't read it.

Then we go and do this, and we return each enw value accessing this shared memory.

```
    public class Counter {
        private long value;
        public long getAndIncrement() {
            return value += 1;
        }
    }
```

This would work for a single thread, but it doesn't work in concurrent threads.

The reason is that when we do this thing where we increment and return the value, there's two operations: first we read and do `temp = value` and then we do `value = temp + 1` and `return temp`. Reading and writing are two separate things.

But suppose we start out with the value being 1. Then at time 0, both threads read the counter and get a vlaue of 1. Because hte system is asynchronous, one of the guys could be really slow, and taking a long time before they do the write. In the meantime, the faster guy does a bunch of stuff. So then when this guy finally writes, he's overwriting with a previous thing.

So you have this shared thing and when you concurrently mix reads and writes, you get a mess.

> **Question 23.1.** Is this problem inherent?

This is essentially what happens when you and another person are walking towards each other, and both of you move, and then you bump into each other – when you see them, you move, but you don't knwo whether they're moving as well or not.

This is inherent; there's a proof that you cannot solve this problem in asynchronous problems.

If you could *glue* reads and writes together (meaning you look and move at the same time), then we wouldn't have a problem. So if we glue them together, then this phenomenon doesn't happen.

So what we want to do is make the two steps `temp = value` and `value = temp + 1` atomic. That's what we do on modern multiprocessors.

One way is that most modern processors provide operations that do this – they give you an atomic read and write. This is a hardware solution.

But you can also solve in software, by doing]vocabMutual exclusion:

§23.3 Mutual Exclusion

```
    public long getAndIncrement() {
2       lock();
            temp = value;
4           value = temp + 1;
        unlock();
6       return temp;
    }
```

We want certaion conditions on lock that guarantee that this isn't being performed by two things at the same time.

> **Question 23.2.** Imagine that we have two people Alice and Bob, and they have a pond surrounded by trees. Bob has a pet, and Bob has a pet. The pets don't like each other, so they can't let the pets be in the pond at the same time. But we want them to be able to get into the pond.

In order to formalize this, there are two formal properties we want: the **safety properties** are that nothing bad ever happens, and the **liveness properties** are that something good eventually happens.

A safety property is something like you never go directly from green to red (you always have yellow in the middle), and a liveness property means that eventually the light turns green (so that cars can get through). In sequential computation we typically just care about safety.

So we can formalize our pet problem: the safety property is **mutual exclusion**, that both pets are never in the pond simultaneously. The liveness property is **no deadlock** – if one pet wants to get in, then it will; and if both pets want to get in, then one of them will. So essentially, all the time pets can be going in and out of the pond.

So to solve the mutual exclusion problem, we need to provide both mutula exclusion and no deadlock.

This is necessary because: in real life, the simple protocol is to just look at the poind, see if the other pet is there, and let your pet out. But you can't, because there are trees. Threads can't see what other threads are doing, so communication is required.

The next thing you can try to do is use your cell phone – Bob calls Alice (or vice versa) when they let their pet out. But if Bob takes a shower or Alice recharges the battery or Bob is out shopping for pet food, then there's no way to let the pet out.

So this kind of solution doesn't really work. The interpretation is that you can't solve mutual exclusion by sending messages (this is a real thing) – the recipient might not be listening, or even there. THe communication must be persistent (like writing), rather than transient (like speaking – because if you talk and I'm not listening, I won't hear you).

One way is the **can protocol**. You put cans on the windowsill, and every time you want to pass a bit, you pull the can. So there are cans on Alice's windowsill, where the strings lead to Bob's house; Bob can pull strings to knock over the cans. But eventually you'll run out of cans – someone has to reset them.

THe interpretation is that you can't solve mutual exclusion with interrupts.

But one thing you can do is by the **flag protocol**. Alice and Bob each have a flag. What they'll do is Alice raises her flag and lets her pet out, and then lets the flag down when she's done. Similarly for Bob.

So Alice raises her flag, waits until Bob's flag is down, unleashes her pet, and then lowers the flag when the pet returns. Bob does the same – he raises his flag, waits until Alice's flag is down, unleashes the pet, and lowers the flag.

The problem is what if they both raise their flags. Then they might never get in – we're providing mutual exclusion, but there's a problem in terms of no deadlock if they both raise their flags and no one can get in.

So then we want to make the protocol asymmetric. We keep Alice's protocol the same. But for Bob, he raises the flag. Then while Alice's flag is up, he lowers the flag, waits for Alice's flag to go down, and then raises the flag. So Bob defers to Alice – if he sees Alice's flag up, he lowers his flag so that she can let her pet in. Then he releases his pet.

We guarantee mutual exclusion by the flag principle – if both flags are raised and both of them look, then one of them is the last person to see the flags up, and will not let their pet in.

We can turn this into a proof: assume both pets are in the pond, and we'll derive a contradiction by reasoning backwards. Consider teh last time Alice and Bob each looked before letting the pets in. WLOG Alice was the last to look.

Alice looked before letting her pet into the pond. Alice raised her flag before her last look. We know Bob let his pet into the pond; that means that the last time he looked, he did not see Alice's flag, so Bob's last look was before the last time Alice raised her flag. But then Bob's last raised flag was before he actually looked, which was before this entire thing. So then Alice's last look must have seen Bob's raised flag, and Alice wouldn't have let her pet into the pond.

So that proves mutual exclusion. But we also need to prove no deadlock.

If only one pet wants in, it gets in. Deadlock requires that both are continually trying to get in. But if Bob sees Alice's flag, then he lowers his flag and Alice will get in. So there's no deadlock. The point is that the protocol is asymmetric.

So we've just proven that with simple flags, you can solve mutual exclusion. All you need is one bit raised in memory.

But of course the protocol is unfair – Bob's pet might never get in. It also uses waiting – if the pet goes out of the pond but you're still waiting for the flag to be lowered, and Bob gets eaten by his pet, then Alice's pet might never get in.

The moral is that Mutual Exclusion cannot be solved with transient communication or interrupts, but it can be solved by one-bit shared variables that can be read or written.

§23.4 Parallel Computing and Amdahl's Law

Speedup is the 1-thread execution time divided by the n-thread execution time.

Theorem 23.3 (Amdahl's Law)

$$\text{Speedup} = \frac{1}{1 - p + \frac{p}{n}}.$$

Here we imagine that the total execution time is 1 (in the sequential execution time, with one thread). We let p be the **parallel fraction** of that time – you take that unit of time in the sequential execution. You can take some part p of it and speed that part up. Meanwhile, the $1 - p$ part is the **sequential fraction**, the part that you cannot parallelize and have to do sequentially.

For example, in the shared counter example, you can do all the primality tests in the $\frac{p}{n}$ part. But the two operations of the shared counter are sequential, so in the $1 - p$ thing.

The parallel fraction gets speeded up by a factor of n, where n is the number of processors. So the new time ends up being $1 - p + \frac{p}{n}$, since the sequential part remains whatever it was, and the parallel part gets cut down into smaller pieces.

Example 23.4

Suppose we have 10 processors, with 60% of the process concurrent and 40% sequential.

Then the speedup we get is

$$\frac{1}{0.4 + 0.6/10} = 2.17.$$

This is much less than 10. If we now have 80%, then we get 3.57. If weh ave 90%, then we end up with 5.26. This is still much less than 10. This is not good news.

If we do 99% concurrent and 1% sequential, *then* we get 9 times. That's what's hard – to get speedups, we have to minimize the pieces that remain sequential.

That's what happens here – when we add more and more cores, there's more coordination and unparallelizable stuff that's problematic. (If you took the whole code and put it in the lock, that would be fully sequential.)

So what you have to do is reduce the sequential part of the code – essentially, you have to reduce parts where threads are talking to each other. Those are the shared data structures.

Say we had concurrent DFS, so we have all kinds of data structures managing the graph. If these pieces are 25% of the code (and are shared), and the rest of what we did is all parallelizable, then on the 25% where they talk to each other, it matters what the granularity of mutual exclusion is.

If we just had one lock over everything, what we'd get is all those processors are waiting for the current one to finish and you only get 2.9 speedup with 4 processors, and 4 times speedup with ∞ processors.

So really this is why fine-grained parallelism matters – you need to figure out how to do the parts with communication effectively.

To summarize, parallelism and concurrency are complex topics. As you go on and implement algorithms, you'll get exposed to the world of making algorithms fast on these machines. Multicores are everywhere; any algorithm, if you actually implemented, you'd probably have to do it concurrently.