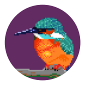
Algorithms

or: the Unofficial Guide to the Georgia Institute of Technology's **CS6515**: *Graduate Algorithms*



George Kudrayvtsev
george.k@gatech.edu

Last Updated: February 22, 2020

The only way to get through this course is by solving an uncountably-infinite number of practice problems while fueled by copious amounts of caffeine ☕.

WHY do we need to study algorithms, and why these specifically? The most important lesson that should come out of this course (that is unfortunately only mentioned in Chapter 8 of *Algorithms* and the 4th lecture of *Graduate Algorithms*) is that many problems can be reduced to an algorithm taught here; they are considered the **fundamental algorithms in computer science**, and if you know them inside-and-out, you can "transform" novel problems into a problem that can be solved by one of these algorithms.

For example, a complicated problem like "can you make change for v using some limited number of coins" can be *reduced* to a Knapsack problem, a quest to "find the length of the longest palindromic subsequence in a string" can be *reduced* to the finding the Longest Common Subsequence, and the arbitrage problem of finding a way to trade currencies on international exchanges to make a profit can be *reduced* to finding negative weight cycles via Shortest Paths.

Keep this in mind as you work through exercises.

List of Algorithms

PART I

Notes

BEFORE we begin to dive into all things algorithmic, some things about formatting are worth noting.

An item that is highlighted **like this** is a "term" that is cross-referenced wherever it's used. Cross-references to these vocabulary words are subtly highlighted like this and link back to their first defined usage; most mentions are available in the Index.

I also sometimes include margin notes like the one here (which just links back here) that reference content sources so you can easily explore the concepts further. Linky

Contents

Dynamic Programming

THE **dynamic programming** (commonly abbreviated as DP to make undergraduates giggle during lecture) problem-solving technique is a powerful approach to creating efficient algorithms in scenarios that have a lot of repetitive data. The key to leveraging dynamic programming involves approaching problems with a particular mindset (paraphrased from *Algorithms*, pp. 158):

> *From the original problem, identify a collection of subproblems which share two key properties: (a) the subproblems have a distinct ordering in how they should be performed, and (b) subproblems should be related such that solving "earlier" or "smaller" subproblems gives the solution to a larger one.*

Keep this mindset in mind as we go through some examples.

1.1 Fibbing Our Way Along. . .

A classic toy example that we'll start with to demonstrate the power of dynamic programming is a series of increasingly-efficient algorithms to compute the **Fibonacci sequence**:

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, \ldots$$

In general, $F_n = F_{n-1} + F_{n-2}$, with the exceptional base-case that $F_n = n$ for $n \leq 1$. The simplest, most naïve algorithm (see algorithm 1.1) for calculating the n^{th} Fibonacci number just recurses on each F_m as needed.

Notice that each branch of the recursion tree operates independently despite them doing almost identical work: we know that to calculate F_{n-1} we need F_{n-2}, but we are also calculating F_{n-2} separately for its own sake. . . That's a lot of extra work that increases *exponentially* with n!

Wouldn't it be better if we kept track of the Fibonacci numbers that we generated as we went along? Enter FIB2(n), which no longer recurses down from F_n but instead

Algorithm 1.1: Fib1 (n), a naïve, recursive Fibonacci algorithm.

```
Input: An integer $n \geq 0$.
Result: $F_n$
if $n \leq 1$ then
    return $n$
end
return Fib1 $(n-1)$ + Fib1 $(n-2)$
```

builds up to it, saving intermediate Fibonnaci numbers in an array:

Algorithm 1.2: Fib2 (n), an improved, iterative Fibonacci algorithm.

```
Input: An integer $n \geq 0$.
Result: $F_n$
if $n \leq 1$ then
    return $n$
end
$\mathcal{F} := \{0, 1\}$
foreach $i \in [2, n]$ do
    $\mathcal{F}[i] = \mathcal{F}[i-1] + \mathcal{F}[i-2]$
end
return $\mathcal{F}[n]$
```

The essence of dynamic programming lies in identifying the potential to implement two main principles:

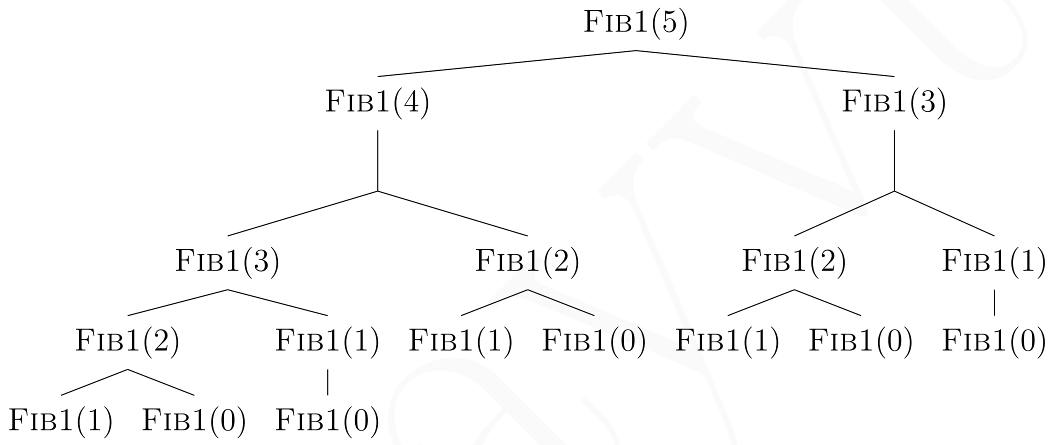
- *avoid repetitive work* and instead store it after computing it once. Identifying the overlapping subproblems (like the fact that F_{n-1} and F_{n-2} share large amounts of work) is a key part in developing a dynamic programming approach. This means you should not shy away from high memory usage when implementing a dynamic programming algorithm—the speed savings from caching repeated intermediate results outweigh the "cost" of memory.
- *avoid recursion* and instead use an iterative approach. This point is actually **not** universal when it comes to dynamic programming and pertains specifically to our course. We could have likewise made algorithm 1.2 pass around an array parameter to a recursive version; this would be an example of a **memoization** technique.

Memoization and recursion often go hand-in-hand when it comes to dynamic programming solutions, but this class shies away from them. Some of the walk-throughs may not, though, since it's (in my opinion) an arbitrary restriction that may make problems harder than they need to be.

1.1.1 Recursive Relations

Let's look at algorithm 1.1 through a different lens and actually try to map out the recursion tree as it develops.

Suppose we want to calculate F_5... our algorithm would then try to calculate F_4 and F_3 separately, which will try to calculate F_3 and F_2, and so on...



Notice the absurd duplication of work that we avoided with Fib2()... Is there a way we can represent the amount of work done when calling Fib2(n) in a compact way?

Suppose $T(n)$ represents the running time of Fib1(n). Then, our running time is similar to the algorithm itself. Since the base cases run in constant time and each recursive case takes $T(n-1)$ and $T(n-2)$, respectively, we have:

$$T(n) \leq O(1) + T(n-1) + T(n-2)$$

So $T(n) \geq F_n$; that is, our *running time* takes at least as much time as the Fibonacci number itself! So F_{50} will take 12,586,269,025 steps (that's 12.5 *billion*) to calculate with our naïve formula...

The Golden Ratio

Interestingly enough, the Fibonacci numbers grow exponentially in φ, the **golden ratio**, which is an interesting mathematical phenomenon that occurs often in nature.

The golden ratio is an irrational number:

$$\varphi = \frac{1+\sqrt{5}}{2} \approx 1.6180\ldots$$

and the Fibonacci numbers increase exponentially by approximately $\frac{\varphi^n}{\sqrt{5}}$.

1.2 Longest Increasing Subsequence

A common, more-practical example to demonstrate the power of dynamic programming is finding the longest increasing subsequence in a series.

A **series** is just a list of numbers:

$$5, 7, 4, -3, 9, 1, 10, 4, 5, 8, 9, 3$$

A **subsequence** is a set of numbers from a series that is still in order (but is not necessarily consecutive):

$$4, 1, 4, 9$$

Thus, what we're looking for in a longest-increasing-subsequence (or LIS) algorithm is the longest set of numbers that are in order relative to the original series that increases from smallest to largest. For our example, that would be the subsequence:

$$-3, 1, 4, 5, 8, 9$$

1.2.1 Finding the Length

To start off a little simpler, we'll just be trying to find the length. Let's start by doing the first thing necessary for all dynamic programming problems: *identify shared work.*

Suppose our sequence was one element shorter:

$$5, 7, 4, -3, 9, 1, 10, 4, 5, 8, 9$$

Intuitively, wouldn't everything we need to do stay almost the same? The only thing that should matter is checking whether or not the new digit 3 can affect our longest sequence in some way. And wouldn't 3 only come into play for subsequences that are currently smaller than 3?

That's a good insight: at each step, we need to compare the new digit to the largest element of all previous subsequences. And since we don't need the subsequence itself, we only need to keep track of its length and its maximum value. Note that we track *all* previous subsequences, because the "best" one at a given point in time will not necessarily be the best one at *every* point in time as the series grows.

What is a potential LIS' maximum value? Well, it's the increasing subsequence that ends in that value! So given a list of numbers: $\mathcal{S} = \{x_1, x_2, \ldots, x_n\}$, we want a subproblem $L(i)$ that is the longest increasing subsequence of $x_{1..i}$ that, critically, includes x_i itself:

$$L(i) = 1 + \max_{1 \le j < i} \{L(j) \mid x_j < x_i\}$$

Algorithm 1.3: LIS1 $(\mathcal{S})$, an approach to finding the longest increasing subsequence in a series using dynamic programming.

Input: $\mathcal{S} = \{x_1, \ldots, x_n\}$, a series of numbers.
Result: L, the length of the LIS of $\mathcal{S}$.

```
/* Initialize the LIS length to a baseline. */
$\mathcal{L} := 0$
foreach $x_i \in \mathcal{S}$ do
    $m := 1$
    foreach $j \in [1, i)$ do                /* find the best LIS to append to */
        if $x_i > x_j \wedge \mathcal{L}[j] \geq m$ then
            $m = \mathcal{L}[j]$
        end
    end
    $\mathcal{L}[i] = m + 1$
end
return $\max(\mathcal{L})$
```

The running time of the algorithm is $O(n^2)$ due to the double `for`-loop over (up to) n elements each.

1.3 Longest Common Subsequence

Let's move on to another dynamic programming example. Given two equal-length strings,

$$X = \{x_1, x_2, \ldots, x_n\}$$
$$Y = \{y_1, y_2, \ldots, y_n\}$$

we want to find the length ℓ of their longest common subsequence (commonly abbreviated LCS). The list of characters that appear in the same relative order (possibly with gaps) is a common subsequence. For example, given:

$$X = BCDBCDA$$
$$Y = ABECBAB$$

the LCS is $BCBA$. How can we find this algorithmically?

1.3.1 Step 1: Identify Subproblems

Dynamic programming solutions are supposed to build upon themselves. Thus, we should naturally expect our subproblems to just be increasingly-longer prefixes of the input strings. For example, suppose we're three characters in and are analyzing the 4th character:

$$X = BCD \leftarrow \boldsymbol{B} = x_{i=4}$$
$$Y = ABE \leftarrow \boldsymbol{C} = y_{i=4}$$

Notice that $\ell = 1$ before and $\ell = 2$ after, since we go from B to BC as our LCS. In general, though, there are two cases to consider, right? Either $x_i = y_i$, in which case we know *for sure* that ℓ increases since we can just append both characters to the LCS, or $x_i \neq y_i$, in which case we need to think a little more.

It's possible that either the new x_i or the new y_i makes a new LCS viable. We can't integrate them into the LCS at the same time, so let's suppose we only have a new y_i:

$$X = BCD$$
$$Y = ABE \leftarrow \boldsymbol{C}$$

Wait, what if we built up X character-by-character to identify where C fits into LCSs? That is, we first compare C to $X = B$, then to $X = BC$, then to $X = BCD$, tracking the length at each point in time? We'd need a list to track the best ℓ:

	B	C	D
C	0	1	1

What about if there was already a C? Imagine we instead had $X' = BCC$. By our logic, that'd result in the length table

	B	C	C
C	0	1	2

In proper dynamic programming fashion, though, we should assume that our previous subproblems gave us accurate results. In other words, we'd know that $\ell_1 = 1$ because of the earlier $y_2 = B$, so we can assume that our table will automatically build up:

	B	C	D
C	1	2	2

How would we know that? One idea would be to compare the new character y_i to all of the previous characters in X. Then, if $x_j = y_i$, we know that the LCS will increase, so we'd increment ℓ_i:

$$\ell_i = 1 + \max_{1 \le j < i} \{\ell_j\}$$

For our example, when we're given $y_4 = C$, we see that $x_2 = C, \ell_2 = 1$, so we set $\ell_4 = 2$.

1.3.2 Step 2: Find the Recurrence

Under this mindset, what's our recurrence? We need to track ℓs for every character in our string, and each new character might increment any of the $j < i$ subsequences before it. One might envision a matrix relating the characters of one string to the other, filled out row-by-row with the LCS length at each index:

	B	C	D	B
A	0	0	0	0
B	1	1	1	1
E	1	1	1	1
C	1	2	2	2

With the introduction of each i^{th} character, we need to compare to the previous characters in the other string. We can't evaluate both x_i and y_i simultaneously (unless they're equal), and so we say $L(i, j)$ contains the LCS length for an i-length X and a j-length Y.

Then, the latest LCS length is based on whether or not the last characters match:

$$L(i,j) = \begin{cases} 1 + L(i-1, j-1) & \text{if } x_i = y_j \\ \max\left(L(i-1,j), L(i,j-1)\right) & \text{otherwise} \end{cases}$$

In other words, if the last characters match, we increment the LCS that would've resulted *without* those letters. If they don't, we just consider the best LCS if we hadn't used x_i xor y_j.

Our base case is trivially: $L(0, \cdot) = L(\cdot, 0) = 0$. That is, when finding the LCS between strings where one of them is the empty string, the length is obviously zero.

1.4 Knapsack

A popular class of problems that can be tackled with dynamic programming are known as **knapsack** problems. In general, they follow a formulation in which you must select the optimal objects from a list that "fit" under some criteria and also maximize a value.

More formally, a knapsack problem is structured as follows:

Input: A set of n objects with values and weights:

$$\begin{aligned} V &= \{v_1, v_2, \ldots, v_n\} \\ W &= \{w_1, w_2, \ldots, w_n\} \end{aligned}$$

as well as a total capacity B.

Goal: To find the subset $S \in V$ that both:

(a) maximizes the total value: $\max\limits_{S \in B} \sum\limits_{i \in S} v_i$

(b) while fitting in the knapsack: $\sum\limits_{i \in S} w_i \leq B$

There are two primary variants of the knapsack problem: in the first, there is only one copy of each object, whereas in the other case objects can be repeatedly added to the knapsack without limit.

1.4.1 Greedy Algorithm

A natural approach to solving a problem like this might be to greedily grab the highest-valued object every time. Unfortunately, this does not always maximize the total value, and even simple examples can demonstrate this.

Suppose we're given the following values:

$$\begin{aligned} V &= \{v_1 = 15, v_2 = 10, v_3 = 8, v_4 = 1\} \\ W &= \{w_1 = 15, w_2 = 12, w_3 = 10, w_4 = 5\} \\ B &= 22 \end{aligned}$$

A greedy algorithm would sort the objects by their value-per-weight: $r_i \frac{v_i}{w_i}$. In this case (the objects are already ordered this way), it would take v_1, then v_4 (since after taking on w_1, w_4 is the only one that can still fit). However, that only adds up to 16, whereas the optimal solution picks up v_2 and v_3, adding up to 18 and coming in exactly at the weight limit.

1.4.2 Optimal Algorithm

As before, we first define a subproblem in words, then try to express it as a recurrence relation.

Attempt #1

The easiest baseline to start with is by defining the subproblem as operating on a smaller prefix of the input. Let's define $K(i)$ as being the maximum achievable using a subset of objects up to i. Now, let's try to express this as a recurrence.

What does our 1D table look like when working through the example above? Given

values:	15	10	8	1
weights:	15	12	10	5

We can work through and get $K: \begin{array}{|c|c|c|c|} 15 & 15 & 15 & 15 \end{array}$. On the first step, obviously 15 is better than nothing. On the second step, we can't take both, so keeping 15 is preferred. On the third step, we can grab both v_3 and v_2, but notice that this requires bigger knowledge than what was available in $K(1)$ or $K(2)$. We needed to take a *sub*optimal solution in $K(2)$ that gives us enough space to properly consider object 3…

This tells us our subproblem definition was insufficient, and that we need to consider suboptimal solutions as we build up.

Solution

Let's try a subproblem that keeps the $i-1^{\text{th}}$ problem with a smaller capacity $b \leq B - w_i$. Now our subproblem tracks a 2D table $K(i, b)$ that represents the maximum value achievable using a subset of objects (up to i, as before), AND keeps their total weight $\leq b$. The solution is then at $K(n, B)$.

Our recurrence relation then faces two scenarios: either we include w_i in our knapsack if there's space for it AND it's better than excluding it, or we don't. Thus,

$$K(i,b) = \begin{cases} \max \begin{cases} v_i + K(i-1, b-w_i) \\ K(i-1, b) \end{cases} & \text{if } w_i \leq b \\ K(i-1, b) & \text{otherwise} \end{cases}$$

with the trivial base cases $K(0, b) = K(i, 0) = 0$, which is when we have no objects and no knapsack, respectively. This algorithm is formalized in algorithm 1.4; its running time is very transparently $O(nB)$.

1.5 Knapsack With Repetition

In this variant, we have an unlimited supply of objects: we can use an ojbect as many times as we'd like.

Algorithm 1.4: Knapsack($\cdot$), the standard knapsack algorithm with no object repetition allowed, finding an optimal subset of values to fit in a capacity-limited container.

```
Input: List of object weights: $W = \{w_1, \dots, w_n\}$
Input: List of object values: $V = \{v_1, \dots, v_n\}$
Input: A capacity, $B$.
Result: The collection of objects resulting in the maximum total value without
        exceeding the knapsack capacity.
// For the base cases, only the first row and column are required to
   be zero, but we zero-init the whole thing for brevity.
$\mathbf{K}_{n+1\times B+1} \gets \mathbf{0}$
foreach $i \in [1..n]$ do
    foreach $b \in [1..B]$ do
        $x = \mathbf{K}[i-1, b]$
        if $w_i \leq b$ then
            $\mathbf{K}[i, b] = \max\left[x, v_i + \mathbf{K}[i-1, b-w_i]\right]$
        else
            $\mathbf{K}[i, b] = x$
        end
    end
end
return $K(n, B)$
```

1.5.1 Simple Extension

Let's try to solve this with the same formulation as before: let's define $K(i, b)$ as being the maximum value achievable using a **multiset** (that is, a set where duplicates are allowed) of objects $\{1, \dots, i\}$ with the weight $\leq b$. Our resulting recurrence relation is almost identical, but we don't remove the object from our pool of viabilities when considering smaller bags:

$$K(i, b) = \max \begin{cases} K(i-1, b) \\ v_i + K(i, b-w_i) \end{cases}$$

1.5.2 Optimal Solution

Do we even need i—the variable we use to track the objects available to us—anymore? Intuition says "no," and we might actually be able to formulate an even better solution because of it.

Since we're never considering the object set, we only really need to define our subproblem as an adjustment of the available capacity. Let's define $K(b)$ as the maximum value achievable within the total weight $\leq b$.

Now we need to consider all the *possibilities* of a last object, rather than restrict ourselves to the i^{th} one, and take the best one. The recurrence can then be expressed as:

$$K(b) = \max_i \{v_i + K(b - w_i) \mid 1 \leq i \leq n, w_i \leq b\}$$

This is formalized in algorithm 1.5.

ALGORITHM 1.5: KNAPSACKREPEATING($\cdot$), the generalized knapsack algorithm in which unlimited object repetition is allowed, finding an optimal multiset of values to fit in a capacity-limited container.

Input: List of object weights: $W = \{w_1, \ldots, w_n\}$
Input: List of object values: $V = \{v_1, \ldots, v_n\}$
Input: A capacity, B.
Result: The collection of objects (possibly with duplicates) resulting in the maximum total value without exceeding the knapsack capacity.

```
$K \leftarrow \mathbf{0}$                // defines our 1D, $B$-element array, $K$.
foreach $b \in [1..B]$ do
    $K(b) = 0$
    foreach $w_i \in W \mid w_i \leq b$ do
        $v \leftarrow v_i + K(b - w_i)$
        if $v > K(b)$ then
            $K(b) = v$
        end
    end
end
return $K(B)$
```

1.6 Matrix Multiplication

Multiplying any two matrices, $\mathbf{A}_{n\times m}$ and $\mathbf{B}_{m\times k}$, takes $O(nmk)$ time. Anyone who has worked with matrices knows that the order in which you multiply a chain of matrices can heavily impact performance. For example, given a 2×5 matrix $\mathbf{A}$, a 5×10 matrix

B, and a 10×2 matrix **C** as such:

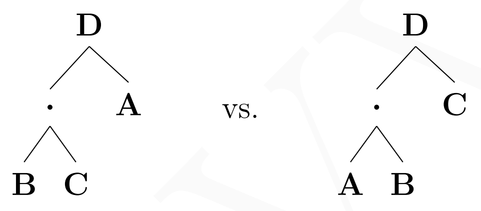
$$\mathbf{D}_{2\times 2} = \underbrace{(\mathbf{A}_{2\times 5} \cdot \mathbf{B}_{5\times 10})}_{2\times 10 \text{ result, 100 ops}} \cdot \mathbf{C}_{10\times 2}$$

takes 140 total ops, a bit less than the alternative order which takes 150 ops:

$$\mathbf{D}_{2\times 2} = \mathbf{A}_{2\times 5} \cdot \underbrace{(\mathbf{B}_{5\times 10} \cdot \mathbf{C}_{10\times 2})}_{5\times 2 \text{ result, 100 ops}}$$

The set of ordering possibilities grows exponentially as the number of matrices increases. Can we compute the best order efficiently?

To start off, envision a particular parenthesization as a binary tree, where the root is the final product and the branches are particular matrix pairings:

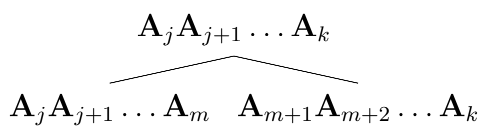


Then, if we associate a cost with each node (the product of the matrix dimensions), a cost for each possible tree bubbles up. Naturally, we want the tree with the lowest cost.[1] The dynamic programming solution comes from a key insight: **for a root node to be optimal, all of its subtrees must be optimal**.

1.6.1 Subproblem Formulation

Let's formally define our problem. If we have n matrices, $\mathbf{A}_1, \mathbf{A}_2, \dots, \mathbf{A}_n$, we can define their sizes as being $m_{i-1} \times m_i$, since the inner dimensions of a matrix product must match. That is, the size of matrix $\mathbf{A}_2$ would be $m_1 \times m_2$. The cost of a particular product, $\mathbf{A}_i\mathbf{A}_j$, is then $c_{ij} = m_{i-1}m_im_j$.

Now we can generalize the above binary tree's structure: the subtree of the product on the left will be $\mathbf{A}_1\mathbf{A}_2 \dots \mathbf{A}_i$ while the subtree on the right will be $\mathbf{A}_{i+1}\mathbf{A}_{i+2} \dots \mathbf{A}_n$. Thus, an arbitrary subtree is a product defined by some range $[j, k]$: $\mathbf{A}_j\mathbf{A}_{j+1} \dots \mathbf{A}_k$ which can be thought of as a "substring" of $\{1, 2, \dots, n\}$.


$\mathbf{A}_j\mathbf{A}_{j+1} \dots \mathbf{A}_k$
$\mathbf{A}_j\mathbf{A}_{j+1} \dots \mathbf{A}_m$
$\mathbf{A}_{m+1}\mathbf{A}_{m+2} \dots \mathbf{A}_k$

[1] It should be fairly obvious how we compute cost. Multiplying two matrices, $\mathbf{W}_{a\times b}$ and $\mathbf{Y}_{b\times c}$, takes $O(abc)$ time. A single entry z_{ij} in the result is the dot product between the i^{th} row of $\mathbf{W}$ and the j^{th} column of $\mathbf{Y}$: $z_{ij} = \sum_{k=0}^{b} (w_{ik} \cdot y_{kj})$. This takes b operations and occurs ac times.

This indicates a subproblem with two indices: let $M(i, j)$ be the minimum cost for computing $\mathbf{A}_i \dots \mathbf{A}_j$. Then our solution will be at $M(1, n)$. Obviously it's a little strange if $j < i$, so we'll force $j \geq i$ and focus on building up the "upper triangle" of the 2D array.

1.6.2 Recurrence Relation

Let's formulate the base case for our $M(i, j)$. Unlike with the subproblems we've seen before, $M(1, j)$ doesn't mean anything special. However, $M(i, i)$ does: there are no ops required to find $\mathbf{A}_i$, so $M(i, i) = 0$.

To compute $M(i, j)$, we need to consider the best point at which $\mathbf{A}_i \dots \mathbf{A}_j$ can be split, so choose the k that results in the best split:

$$M(i,j) = \min_{i \leq k < j} [\underbrace{M(i,k)}_{\substack{\text{left}\\\text{subtree}}} + \underbrace{M(k+1,j)}_{\substack{\text{right}\\\text{subtree}}} + \underbrace{m_{i-1}m_k m_j}_{\substack{\text{cost of new}\\\text{product}}}]$$

Notice that each new cell depends on cells both below it and to the left of it, so the array must be built from the diagonal outward for each off-diagonal (hence the weird indexing in algorithm 1.6).

Computing the best value for a single cell takes $O(n)$ time in the worst case and the table has $\frac{n^2}{2}$ cells to fill, so the overall complexity is $O(n^3)$ which, though large, is far better than exponential time.

Algorithm 1.6: An $O(n^3)$ time algorithm for computing the cost of the best chain matrix multiplication order for a set of matrices.

Input: A set of sizes corresponding to matrix dimensions, $[m_1, m_2, \dots, m_n]$.
Result: The lowest-cost matrix multiplication possible.

```
$\mathbf{M}_{n \times n} \leftarrow \emptyset$
$\mathbf{M}[i, i] = 0 \qquad \forall i \in [1, n]$
foreach $d \in [1, n-1]$ do
    foreach $i \in [1, n-d]$ do
        $j \leftarrow i + d$
        $\mathbf{M}[i, j] = \min_{i \leq k < j} [\mathbf{M}[i, k] + \mathbf{M}[k+1, j] + m_{i-1} m_k m_j]$
    end
end
return M
```

Divide & Conquer

It is the rule in war, if ten times the enemy's strength, surround them; if five times, attack them; if double, be able to divide them; if equal, engage them; if fewer, defend against them; if weaker, be able to avoid them.

— Sun Tzu, *The Art of War*

DIVIDING a problem into 2 or more parts and solving them individually to conquer a problem is an extremely common and effective approach. If you were tasked with finding a name in a thick address book, what would you do? You'd probably divide it in half, check to see if the letter you're looking for comes earlier or later, then check midway through that half, disregarding the other half for the remainder of your search. This is exactly the motivation behind **binary search**: given a sorted list of numbers, you can halve your search space every time you compare to a value, taking $O(\log n)$ time rather than the naïve $O(n)$ approach of checking every number.

2.1 An Exercise in D&C: Multiplication

Consider the traditional approach to multiplication with decimal numbers that we learned in grade school: for every digit, we multiply it with every other digit.

$$
\begin{array}{ccccccc}
 & & & 1 & 2 & 3 & 4 \\
\times & & & & 1 & 2 & 3 \\
\cline{3-7}
 & & & 3 & 7 & 0 & 2 \\
+ & & 2 & 4 & 6 & 8 & \\
+ & 1 & 2 & 3 & 4 & & \\
\hline
 & 1 & 5 & 1 & 7 & 8 & 2
\end{array}
$$

The process overall in $O(n^2)$. Can we do better? Let's think about a simpler divide-and-conquer approach that leverages the fact that we can avoid certain multiplications.

Quick Maffs: Off-By-One

This minor digression gives us insight on how we can do better than expected when multiplying two complex numbers together. The underlying assumption is that addition is cheaper than multiplication.[a]

We're given two complex numbers: $a+bi$ and $c+di$. Their product, via the FOIL approach we all learned in middle school, can be found as follows:

$$\begin{aligned}
&= (a+bi)(c+di) \\
&= ac + adi + bci + bdi^2 \\
&= (ac - bd) + (ad + bc)i
\end{aligned}$$

There are four independent multiplications here. The clever trick realized by Gauss is that the latter sum can be computed without computing the individual terms. The insight comes from the fact that $ad + bc$ looks like it comes from a FOIL just like the one we did: a, b on one side, d, c on the other:

$$(a+b)(d+c) = \underbrace{ad + bc}_{\text{what we want}} + \underbrace{ac + bd}_{\text{what we have}}$$

Notice that these "extra" terms are already things we've computed! Thus,

$$ad + bc = (a+b)(d+c) - ac - bd$$

Thus, we only need three multiplications: ac, bd, and $(a+b)(d+c)$. We do need more additions, but we entered this derivation operating under the assumption that this is okay. A similar insight will give us the ability to multiply n-bit integers faster, as well.

[a] This is undeniably true, even on modern processors. On an Intel processor, 32-bit multiplication takes xxx CPU cycles while adding only takes xxx.

We're working with the n-bit integers x and y, and our goal is to compute $z = xy$.[1] Divide-and-conquer often involves splitting the problem into two halves; can we do that here? We can't split xy, but we can split x and y themselves into two $n/2$-bit halves:

$$x = \boxed{\;x_l\;}\boxed{\;x_r\;}$$

$$y = \boxed{\;y_l\;}\boxed{\;y_r\;}$$

The original x is a composition of its parts: $x = 2^{n/2}x_l + x_r$, and likewise for y. Thus,

[1] We're operating under the assumption that we can't perform these multiplications on a computer using the hardware operators alone. Let's suppose we're working with massive n-bit integers, where n is something like 1024 or more (like the ones common in cryptography).

$$\begin{aligned} xy &= \left(2^{n/2}x_l + x_r\right)\left(2^{n/2}y_l + y_r\right) \\ &= 2^n x_l y_l + 2^{n/2}(x_l y_r + x_r y_l) + x_r y_r \end{aligned}$$

This demonstrates a recursive way to calculate xy: it's composed of 4 $n/2$-bit multiplications. Of course, we're still doing this in $O(n^2)$ time;[2] there's no improvement yet. Can we now apply a similar trick as the one described in the earlier aside, though, and instead solve *three* subproblems?

Beautifully, notice that $x_l y_r + x_r y_l$ can be composed from the other multplications:

$$(x_l + x_r)(y_l + y_r) = \underbrace{x_l y_r + x_r y_l}_{\text{what we want}} + \underbrace{x_l y_l + x_r y_r}_{\text{what we have}}$$

If we let $A = x_l y_l$, $B = x_r y_r$, and $C = (x_l + x_r)(y_l + y_r)$, then we only need to combine three subproblems:

$$xy = 2^n A + 2^{n/2}(C - A - B) + B$$

which is $O\left(n^{\sqrt{3} \approx 1.59}\right)$, better than before! Turns out, there is an *even better* D&C approach that'll let us find the solution in $O(n \log n)$ time, but this will require a long foray into polynomials, complex numbers, and roots of unity that we won't get into right now.

2.2 Another Exercise in D&C: Median-Finding

Suppose you have an unsorted list of numbers, $A = [a_1, a_2, \ldots, a_n]$. Can you find its median value without sorting it? Turns out, you can. Let's concretely define the median value as being the $\left\lceil \frac{n}{2} \right\rceil^{\text{th}}$-smallest number. It'll actually be easier to solve a "harder" problem: finding the k^{th} smallest value in A.[3]

Naïvely, we could sort A then return the k^{th} element; this would take $O(n \log n)$ time, but let's do better. Let's choose a **pivot**, p, and divide A into three sub-arrays: those less than p, those greater than p, and those equal to p:

$$A_{<p}, A_{=p}, A_{>p}$$

Now we know which sublist the element we're looking for is in: if $k < |A_{<p}|$, we know it's in there. Similarly, if $|A_{<p}| < k < |A_{<p}| + |A_{=p}|$, we know it IS p. Finally, if $k > |A_{<p}| + |A_{=p}|$, then it's in $A_{>p}$.

[2] You can refer to Solving Recurrence Relations or apply the Master theorem directly: $T(n) = 4T(\frac{n}{2}) + O(n) = O(n^2)$.

[3] Critically, it's not the k^{th}-smallest *unique* element, so ties are treated independently. That is, the 2nd smallest element of $[1, 2, 1]$ would be 1.

Let's look at a concrete example, we (randomly) choose $p = 11$ for:

$$A = [5, 2, 20, 17, 11, 13, 8, 9, 11]$$

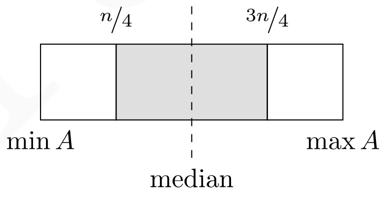
Then,

$$A_{<p} = [5, 2, 8, 9]$$
$$A_{=p} = [11, 11]$$
$$A_{>p} = [20, 17, 13]$$

If we're looking for the $(k \leq 4)^{\text{th}}$-smallest element, we know it's in the first sublist and can recurse on that. If we're looking for the $(k > 6)^{\text{th}}$ smallest element, then we want the $(k - 6)^{\text{th}}$-smallest element in the last sublist (to offset the 6 elements in the two other lists we're now ignoring).

Now the question is reduced to finding a good pivot point, p. If we always chose the median, this would divide the array into two equal subarrays and an array of the median values, so $|A_{<p}| \approx |A_{>p}|$. That means we'd always *at least* halve our search space every time, but we also need to do $O(n)$ work to do the array-splitting; this gives us an $O(n)$ total running time, exactly what we're looking for. However, this is a bit of a chicken-and-egg problem: our original problem is searching for the median, and now we're saying if we knew the median we could efficiently find the median? Great.

What if, instead, we could find a pivot that is *close* to the median (say, within ±25% of it). Namely, if we look at a sorted version of A, that our pivot would lie within the shaded region:



In the worst case, then the size of our subproblem will be $3n/4$ of what it used to be, so: $T(n) = T\left(\frac{3n}{4}\right) + O(n)$, which, magnificently, still solves to $O(n)$. In fact, *any* valid subproblem that's smaller than the original will solve to $O(n)$. We'll keep this notion of a *good* pivot as being within $1/4^{\text{th}}$ of the median, but remember that if we recurse on a subproblem that is at least one element smaller than the previous problem, we'll achieve linear time.

If we chose a pivot randomly, our probability of choosing a good pivot is 50%. The ordering within the list doesn't matter: exactly half of the elements make good pivots. We can check whether or not a chosen pivot is "good" in $O(n)$ time since that's what splitting into subarrays means, and if it's not a good p we can just choose another one. Like with flipping a coin, the **expectation** of p being good is 2 "flips"[4] (that is, it'll take two tries on average to find a good pivot). Thus, we can find a good pivot in $O(n)$ time on average (two tries of n-time each), and thus we can find the n^{th}-smallest element in $O(n)$ on average since we have a good pivot. Finding the median is then just specifying that we want to find the $\frac{n}{2}^{\text{th}}$-smallest element. $\square$

2.3 Solving Recurrence Relations

Recall the typical form of a recurrence relation for divide-and-conquer problems:

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

Here, n is the size of the input problem, a is the number of subproblems in the recursion, b is the factor by which the subproblem size is reduced in each recursive call, and $f(n)$ is the complexity of any additional non-recursive processing.

Lesson 10: DC3

Consider the (recursive) merge sort algorithm. In it, we split our n-length array into two halves and recurse into each, then perform a linear merge. Its recurrence relation is thus:

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

It's possible to go from any recursive relationship to its big-O complexity with ease. For example, we can use the merge sort recurrence to derive the big-O of the algorithm: $O(n \log n)$. Let's see how to do this in the general case.

2.3.1 Example 1: Integer Multiplication

The brute force integer multiplication algorithm's recurrence relation looks like this:

$$T(n) = 4T\left(\frac{n}{2}\right) + O(n)$$

We can similarly derive that its time complexity is $O(n^2)$, but ***how?*** Let's work through it. To start off, let's drop the $O(n)$ term. From the definition of big-O, we know there is some constant c that makes the following equivalent:

$$\begin{aligned}
T(n) &= 4T\left(\frac{n}{2}\right) + O(n) \\
&\le 4T\left(\frac{n}{2}\right) + cn, \qquad \text{where } T(1) \le c
\end{aligned}$$

[4] The expectation can be defined and solved recursively. If we say flipping "heads" makes a good pivot, we obviously need at least one flip. Then, the chance of needing to flip again is 50%, and we'll again need to see the average flip-count to get heads, so $E_h = 1 + \frac{1}{2}E_h \implies E_h = 2$.

Now let's substitute in our recurrence relation twice: once for $T(n/2)$, then again for the resulting $T(n/4)$.

$$\begin{aligned}
T(n) &\leq cn + 4T\left(\frac{n}{2}\right) \\
&\leq cn + 4\left[c\left(\frac{n}{2}\right) + 4T\left(\frac{n}{4}\right)\right] && \text{plug in } T\left(\tfrac{n}{2}\right) \\
&= cn\left(1 + \frac{4}{2}\right) + 4^2T\left(\frac{n}{2^2}\right) && \text{rearrange and group} \\
&\leq cn\left(1 + \frac{4}{2}\right) + 4^2\left[4T\left(\frac{n}{2^3}\right) + c\left(\frac{n}{2^2}\right)\right] && \text{plug in } T\left(\tfrac{n}{2^2}\right) \\
&\leq cn\left(1 + \left(\frac{4}{2}\right) + \left(\frac{4}{2}\right)^2\right) + 4^3T\left(\frac{n}{2^3}\right) && \text{rearrange and group, again}
\end{aligned}$$

Notice that we're starting to see a geometric series form in the cn term, and that its terms come from our original recurrence relation, $4T\left(\frac{n}{2}\right)$:

$$1 + \frac{4}{2} + \left(\frac{4}{2}\right)^2 + \ldots + \left(\frac{4}{2}\right)^n +$$

The pattern is clear after two substitions; the general form for i subtitutions is:

$$T(n) \leq cn\left(1 + \left(\frac{4}{2}\right) + \left(\frac{4}{2}\right)^2 + \left(\frac{4}{2}\right)^3 + \ldots \left(\frac{4}{2}\right)^{i-1}\right) + 4^iT\left(\frac{n}{2^i}\right)$$

But when do we stop? Well our base case is formed at $T(1)$, so we stop when $\frac{n}{2^i} = 1$. Thus, we stop at $i = \log_2 n$, giving us this expansion at the final iteration:

$$T(n) \leq cn\left(1 + \left(\frac{4}{2}\right) + \left(\frac{4}{2}\right)^2 + \ldots + \left(\frac{4}{2}\right)^{\log_2 n-1}\right) + 4^{\log_2 n} \cdot T(1)$$

We can simplify this expression. From the beginning, we defined $c \geq T(1)$, so we can substitute that in accordingly and preserve the inequality:

$$T(n) \leq \underbrace{cn}_{O(n)} \underbrace{\left(1 + \left(\frac{4}{2}\right) + \left(\frac{4}{2}\right)^2 + \ldots + \left(\frac{4}{2}\right)^{\log n-1}\right)}_{\text{increasing geometric series}} + \underbrace{4^{\log n} \cdot c}_{O(n^2)}$$

Thankfully, we established in Geometric Growth that an increasing geometric series with ratio r and k steps has a complexity of $\Theta(r^k)$, meaning our series above has complexity:

$$
\begin{aligned}
O\left(1+\left(\frac{4}{2}\right)+\left(\frac{4}{2}\right)^2+\ldots+\left(\frac{4}{2}\right)^{\log n-1}\right) &= O\left(\left(\frac{4}{2}\right)^{\log n}\right) \\
&= O\left(\frac{4^{\log n}}{2^{\log n}}\right) \\
&= O\left(\frac{2^{\log_2 n}\cdot 2^{\log_2 n}}{n}\right) && \text{recall that } x^i \cdot y^i = (xy)^i \\
&= O\left(\frac{n\cdot n}{n}\right) && \text{remember, } b^{\log_b n} = n \\
&= O(n)
\end{aligned}
$$

Thus, we ultimately have quadratic complexity, as expected:

$$T(n) = O(n)\cdot O(n) + O(n^2) = \boxed{O(n^2)} \quad \square$$

QUICK MAFFS: Generalizing Complexity

One could argue that we got a little "lucky" earlier with how conveniently we could convert $4^{\log n} = n^2$, since 4 is a power of two. Can we do this generically with any base? How would we solve $3^{\log n}$, for example?

Well, we can turn the 3 into something base-2 compatible by the definition of a logarithm:

$$
\begin{aligned}
3^{\log n} &= \left(2^{\log 3}\right)^{\log n} \\
&= 2^{\log 3\cdot\log n} && \text{power rule: } {x^a}^b = x^{ab} \\
&= 2^{\log\left(n^{\log 3}\right)} && \text{log exponents: } c\cdot\log n = \log(n^c) \\
&= n^{\log 3} && \text{d-d-d-drop the base!}
\end{aligned}
$$

This renders a generic formulation that can give us big-O complexity for any exponential:

$$b^{\log_a n} = n^{\log_a b} \tag{2.1}$$

2.3.2 Example 2: Better Integer Multiplication

Armed with our substitution and pattern-identification techniques as well as some mathematical trickery up our sleeve, solving this recurrence relation should be much faster:

$$T(n) = 3T\left(\frac{n}{2}\right) + O(n)$$

As before, we substitute and generalize:

$$\begin{aligned} T(n) &\leq cn + 3T\left(\frac{n}{2}\right) \\ &\leq cn\left(1 + \left(\frac{3}{2}\right) + \left(\frac{3}{2}\right)^2 + \ldots + \left(\frac{3}{2}\right)^{i-1}\right) + 3^i T\left(\frac{n}{2^i}\right) \end{aligned}$$

Our last term is $i = \log_2 n$, as before, so we can again group each term by its complexity:

$$\begin{aligned} T(n) &\leq \underbrace{cn}_{O(n)} \underbrace{\left(1 + \left(\frac{3}{2}\right) + \left(\frac{3}{2}\right)^2 + \ldots + \left(\frac{3}{2}\right)^{\log n - 1}\right)}_{\text{again, increasing: } O\left(\left(\frac{3}{2}\right)^{\log n} = \frac{3^{\log n}}{2^{\log n}} \approx n^{0.585}\right)} + \underbrace{3^{\log n} T(1)}_{O\left(n^{\log 3 \approx 1.585}\right)} \\ &\approx O(n) \cdot O(n^{0.585}) + O(n^{1.585}) \\ &\approx \boxed{O(n^{1.585})} \end{aligned}$$

2.3.3 General Form

If we know certain things about the structure of the recurrence relation, we can actually arrive at the complexity by following a series of rules; no substitutions or derivations necessary!

Given the general form of a recurrence relation, where $a > 0, b > 1$:

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

we have three general cases depending on the outcome of the geometric series after expansion:

$$T(n) = cn \underbrace{\left(1 + \left(\frac{a}{b}\right) + \left(\frac{a}{b}\right)^2 + \ldots + \left(\frac{a}{b}\right)^{\log_b n - 1}\right)} + a^{\log_b n} \cdot T(1)$$

- If $a > b$, the series is dominated by the last term, so the overall complexity is $O(n^{\log_b a})$.
- If $a = b$, the series is just the sum of $\log_b n - 1$ ones, which collapses to $O(\log_b n)$, making the overall complexity $O(n \log_b n)$.
- If $a < b$, the series collapses to a constant, so the overall complexity is simply $O(n)$.

Of course, not every algorithm will have $O(n)$ additional non-recursive work. The general form uses $O\left(n^d\right)$; this is the **Master theorem** for recurrence relations:

Property 2.1. *For the general case of of the recurrence relation,*

$$T(n) = aT\left(\frac{n}{b}\right) + O\left(n^d\right)$$

our case depends on the relationship between d and $\log_b a$.

$$T(n) = \begin{cases} O\left(n^d\right) & \text{if } d > \log_b a \\ O\left(n^d \log_2 n\right) & \text{if } d = \log_b a \\ O\left(n^{\log_b a}\right) & \text{if } d < \log_b a \end{cases}$$

Memorize these rules into the deepest recesses of your mind (at least until May).

2.4 Fast Fourier Transform

I don't know enough about the **fast Fourier transform** to take good notes; this is just a quick-reference guide.

The **n^{th} roots of unity** are defined as:

$$\omega_n^0, \omega_n^1, \ldots, \omega_n^{n-1}$$

where

$$\omega_n = e^{\frac{2\pi i}{n}}$$

so, for example, the 4th of the 8th roots of unity is $\omega_8^4 = \left(e^{\frac{2\pi i}{8}}\right)^4 = e^{\pi i}$. Notice that $\omega_n^n = \omega_n^0 = 1$.

The FFT takes in a list of k constants and specifies the n^{th} roots of unity to use. It outputs a list of evaluations of the polynomial defined by the constants. Namely, if $\mathcal{K} = \{c_0, c_1, c_2, \ldots, c_{k-1}\}$ defines:

$$A(x) = c_0 + c_1 x + c_2 x^2 + \ldots + c_{k-1} x^{k-1}$$

Then the FFT of $\mathcal{K}$ with the n^{th} roots of unity will return n evaluations of $A(x)$:

$$(z_1, z_2, \ldots, z_n) = \text{FFT}(\mathcal{K}, \omega_n)$$

The **inverse fast Fourier transform** just uses a scaled version of the FFT with the inverse roots of unity:

$$\text{IFFT}(\mathcal{K}, \omega_n) = \frac{1}{2n}\text{FFT}(\mathcal{K}, \omega_n^{-1})$$

where ω_n^{-1} is the value that makes $\omega_n^{-1} \cdot \omega_n = 1$. This is defined as ω_n^{n-1}:

$$\begin{aligned}
\omega_n^{-1} = \omega_n^{n-1} &= e^{\frac{2\pi i(n-1)}{n}} \\
&= e^{\frac{2\pi i n}{n}} \cdot e^{\frac{-2\pi i}{n}} && \text{distribute and split} \\
&= e^{2\pi i} \cdot e^{-\frac{2\pi i}{n}} && \text{cancel } n\text{s} \\
&= e^{-\frac{2\pi i}{n}} && \text{Euler's identity: } e^{\pi i} = 1
\end{aligned}$$

Since $\omega_n \cdot \omega_n^{n-1} = e^{\frac{2\pi i}{n}} \cdot e^{-\frac{2\pi i}{n}} = e^0 = 1$.

Graphs

I fucking love graphs.

Definitions

There are many terms when discussing graphs and they all have very specific meanings. This "glossary" serves as a convenient quick-reference.

The typical way a graph is defined is $G = (V, E)$. Sometimes, the syntax $\overrightarrow{G}$ is used to indicate a **directed graph**, where edges can only be traversed in a single direction.

- The syntax $(a, b) \in E$ indicates an edge from $a \to b$ within G.
- The syntax $w(a, b)$ indicates the **weight** (or *distance* or *length*) of the **edge** from $a \to b$.
- A **walk** through a graph can visit a vertex any number of times.
- A **path** through a graph can only visit a vertex once.
- A **cycle** is a path through a graph that starts and ends at the same node.
- A graph is **fully-connected** if each vertex is connected to every other vertex. In this case, there are $|E| = |V|^2$ edges.
- The **degree** of a vertex is the number of edges it has. For directed graphs, there is also $d_{\text{in}}(v)$ to indicate incoming edges (u, v) and $d_{\text{out}}(v)$ to indicate outgoing edges (v, u).

When discussing the big-O complexity of a graph algorithm, typically n refers to $|V|$, the number of vertices, and m refers to $|E|$, the number of edges. In a fully-connected graph, then, a $O(nm)$ algorithm can actually be said to take $O(n^3)$ time, for example.

3.1 Common Algorithms

The EXPLORE function described in algorithm 3.1 is important; it creates the building block for almost any traversal algorithm. It traverses the graph by digging as deep as possible of a path down a single vertex, then backtracking and traversing neighbors until all nodes are searched.

The optional PREVISIT and POSTVISIT functions described in algorithm 3.1 allow nodes to be processed as they're traversed; the latter of these will be used heavily soon.

ALGORITHM 3.1: EXPLORE(G, v), a function for visiting vertices in a graph.

Input: $G = (V, E)$, the graph itself.
Input: $v \in V$, a vertex from which to start exploring.
Input: Optionally, the functions PREVISIT(z) and POSTVISIT(z), which can do additional processing on the first and last time a vertex z is visited.

Result: The global *visited* array (of length $|V|$) is marked with visited[u] $= true$ if the vertex u can be visited from v.
Result: The global *prev* array (also of length $|V|$) is marked with prev[z] $= w$, where w is the first vertex to explore z.

```
visited[$v$] = $true$
PREVISIT($v$)                                   // optional
foreach edge $(v, u) \in E$ do
    if $\neg visited[u]$ then
        EXPLORE($G, u$)
        prev[$u$] = $v$
    end
end
POSTVISIT($v$)                                  // optional
```

3.1.1 Depth-First Search

Performing **depth-first search** on an undirected graph (described in algorithm 3.2) is just a simple way of leveraging EXPLORE() (see algorithm 3.1) on all nodes.

3.1.2 Breadth-First Search

While depth-first search recursively explores nodes until it reaches leaves, **breadth-first search** explores the graph layer by layer. Its output is different than that of

Algorithm 3.2: DFS(G), depth-first search labeling of connected components.

```
Input: $G = (V, E)$, an undirected graph to traverse.
$cc := 0$
foreach $v \in V$ do
    visited[$v$] = $false$
    prev[$v$] = $\emptyset$
end
foreach $v \in V$ do
    if $\neg visited[v]$ then
        $cc \mathrel{+}= 1$
        EXPLORE($G, v$)
    end
end
```

DFS, which tells us about the connectivity of a graph. Breadth-first search (starting from a vertex s) fills out a $dist[\cdot]$ array, which is the minimum number of edges from $s \to v$, for all $v \in V$ (or ∞ if there's no path). It gives us *shortest* paths. Its running time is likewise $O(n + m)$.

3.2 Shortest Paths

First off, its worth reviewing **Dijkstra's algorithm** (see *Algorithms*, pp. 109–112) so that we can use it as a black box for building new graph algorithms. In summary, it performs BFS on graphs with **positively**-weighted edges, outputting the length of the shortest path from $s \to v$ using a `min-heap` data structure. Its running time is $O((n + m) \log n)$.

3.2.1 From One Vertex: Bellman-Ford

In the subproblem definition, we condition on the number of edges used to build the shortest path. Define $D(i, z)$ as being the shortest path between a starting node s and a final node z using exactly i edges.

$$D(i, z) = \min_{y:(y,z)\in E} \{D(i - 1, y) + w(y, z)\}$$

But what if we can get from $s \to z$ without needing exactly i edges? We'd need to find $\min_{j \le i} D(j, z)$, right? Instead, we can incorporate it into the recurrence directly:

$$D(i, z) = \min \begin{cases} D(i-1, z) \\ \min\limits_{y:(y,z) \in E} \{D(i-1, y) + w(y, z)\} & \text{iterate over every edge that goes into } z \end{cases}$$

where we start with the base cases $D(0, s) = 0$, and $D(0, z) = \infty$ for all $z \neq s$, and the solutions for *all* of the vertices is stored in $D(n-1, \cdot)$.

This is the **Bellman-Ford algorithm**. Its complexity is $O(nm)$, where n is the number of vertices and m is the number of edges (that is, $n = |V|, m = |E|$). Note that for a fully-connected graph, there are n^2 edges, so this would take $O(n^3)$ time.

Negative Cycles If the graph has a negative weight cycle, then we will notice that the shortest path changes after $n-1$ iterations (when otherwise it wouldn't). So if we keep doing Bellman-Ford and the n^{th} iteration results in a different row than the $(n-1)^{\text{th}}$ iteration, there is a negative weight cycle. Namely, check if $\exists z \in V : D(n, z) < D(n-1, z)$; if there is, the cycle involves z and we can backtrack through D to identify the cycle.

3.2.2 From All Vertices: Floyd-Warshall

What if we don't just want the shortest path from $s \to z$ for all $z \in V$, but for all *pairs* of vertices (that is, from *any* start to any end). The naïve approach would be to compute Bellman-Ford for all vertices, taking $O(mn^2)$ time. However, we can use the **Floyd-Warshall** algorithm to compute it in $O(n^3)$ time, instead (better for dense graphs, since $m \le n^2$).

This should intuitively be possible, since the shortest path from $s \to z$ probably overlaps a lot with the shortest path from $s' \to z$ if s and s' are neighbors.

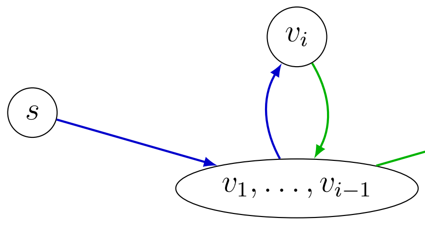
First, lets assign a number to each vertex, so $V = \{v_1, v_2, \ldots, v_n\}$. Now our subproblem is the "prefix" of the vertices, so we solve the all-pairs problem by only using some subset of vertices: $\{v_1, v_2, \ldots, v_i\}$ (and allowing $\emptyset$ as our base case). This is done for all possible start and end vertices, so $s, t \in \{v_1, v_2, \ldots, v_n\}$. Then,

Let $D(i, s, t)$ be the shortest path from $s \to t$ using a subset of the first i vertices. To start off, if we allow no intermediate vertices, only vertices that share an edge have a valid base path length; namely, the base cases are:

$$D(0, s, t) = \begin{cases} \infty & \text{if } \exists (s, t) \in E \\ w(s, t) & \text{otherwise} \end{cases}$$

Now lets look at the recurrence. There's a shortest path $\mathcal{P}$ from $s \to t$, right? Well, what if v_i is not on that path? Then we can exclude it, defering to the $(i-1)^{\text{th}}$ prefix.

And if it is? Then our path looks something like this:



(note that $\{v_1, \ldots, v_{i-1}\}$ might be empty, so in fact $s \to v_i \to t$, but that doesn't matter)

Then, our recurrence when v_i is on the path is just the shortest path to v_i and the shortest path *from* v_i! So, in total,

$$D(i, s, t) = \min \begin{cases} D(i-1, s, t) & \textit{(not on the path)} \\ D(i-1, s, i) + D(i-1, i, t) & \textit{(on the path)} \end{cases}$$

Notice that this implicitly determines whether or not $\{v_1, \ldots v_i\}$ is sufficient to form the path $s \to t$ because of the base case: we have ∞ wherever $s \nrightarrow t$, only building up the table for reachable places. Thus, $D(n, \cdot, \cdot)$ is a 2D matrix for all pairs of (start, end) points.

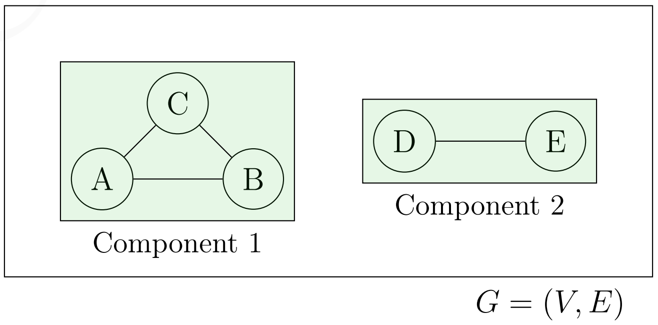
Negative Cycles If the graph has a negative weight cycle, a diagonal entry in the final matrix will be negative, so check if $\exists v \in V : D(n, v, v) < 0$.

3.3 Connected Components

Given a graph, there's a basic question we need to be able to answer:

What parts of the graph are reachable from a given vertex?

A **connected component** in a graph is a fancy way of referring to a set of vertices that are reachable from each other:



DFS lets us find the connected components easily in $O(n + m)$ time since whenever it encounters an unvisited node (post-exploration) it must be a separate component.

3.3.1 Undirected Graphs

How do we find a path between two vertices (s, e) using DFS on an undirected graph? Well, since $\mathit{prev}[e]$ tracks the first vertex to explore e, we can traverse this until we reach our original vertex s, so the path becomes: $\mathcal{P} = \{e, \mathit{prev}[e], \mathit{prev}[\mathit{prev}[e]], \ldots, s\}$. ~~Notice that the exploration order of Explore() actually makes this the shortest path, since $\mathit{prev}[v]$ is only modified the *first* time that v is visited (and the earliest visit obviously comes from the quickest exploration).~~

(I don't think this last statement is actually true, so proceed with caution)

3.3.2 Directed Graphs

Directed graphs (also called **digraphs**) can be traversed as-is using algorithm 3.2, but finding paths is a little more challenging from an intuitive standpoint. However, its implementation is just as simple as before.

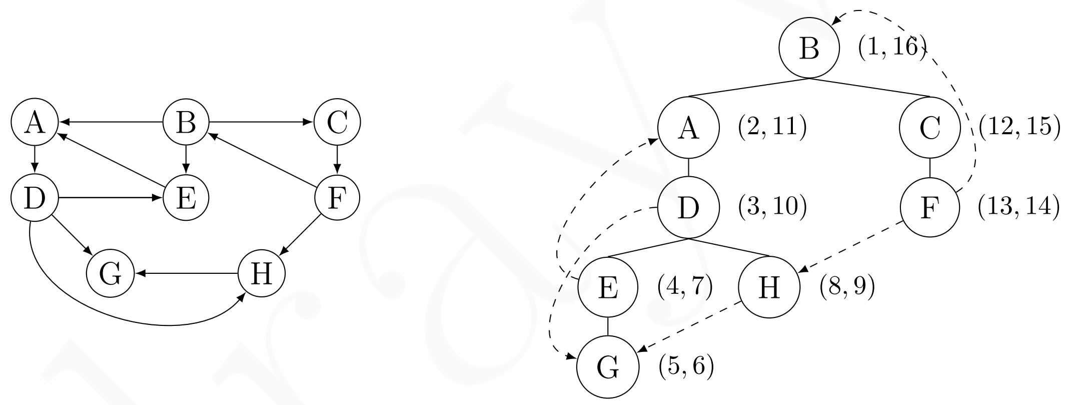


Figure 3.1: A directed graph (left) and its DFS tree with pre- and post-visit clock values at each vertex (right). The dashed edges in the tree represent edges that were ignored because the vertices were already marked "visited."

We'll create a so-called "clock" that tracks the first and last "times" that a node was visited, putting these in global $\mathit{pre}[\cdot]$ and $\mathit{post}[\cdot]$ arrays, respectively. Specifically, we'll define Previsit(v) to store $\mathit{pre}[v] = \text{clock++}$ and Postvisit(v) to do likewise but store in $\mathit{post}[v]$.[1] An example of the values in these variables at each vertex is shown in Figure 3.1 after a DFS traversal. Note that a DFS traversal tree can start at any node; the tree in Figure 3.1 is just one of the possible traversals of its graph (namely, when starting at vertex B).

Types of Edges Given a directed edge $z \to w$, it's classified as a **tree edge** if it's one of the primary, first-visited edges. Examples from Figure 3.1 include all of the

[1] Here, the post-increment syntax is an allusion to C, so the clock is incremented *after* assigning its previous value to the *post* array.

black edges, like (B, A). The other, "redundant" edges are broken down into three categories depending on their relationship in the graph; it's called a...

- **back edge** if it's an edge between vertices going *up* the traversal tree to an **ancestor**. Examples from Figure 3.1 include the dashed edges like (F, B) and (E, A).
- **forward edge** if it's an edge going *down* the tree to a **descendant**. Examples from Figure 3.1 include the dashed edges like (D, G) and (F, H).
- **cross edge** if it's an edge going *across* the tree to a **sibling** (a vertex at the same depth). Unfortunately there are no examples in Figure 3.1, but it could be (E, H) if such an edge existed.

In all but the forward-edge case, $post[z] > post[w]$; in the latter case, though, it's the inverse relationship: $post[z] < post[w]$.

The various categorizations of (possibly-hidden) edges in the DFS traversal tree can give important insights about the graph itself. For example,

Property 3.1. *If a graph contains a cycle, the DFS tree will have a back edge.*

3.3.3 Acyclic Digraphs

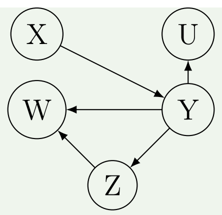
A **directed acyclic graph** (or DAG) contains no cycles (and thus no back edges, as we just saw). Given a DAG, we want to *topologically sort* it: order the vertices so that the higher nodes of the DFS tree have less children than than lower nodes (that is, the degree of a vertex is inversely-proportional to its depth in the DFS tree).

We can achieve this by running DFS and ordering the vertices by their post-order clock values; this takes linear time.[2]

Example 3.1: **Topological Ordering**

Given the following graph, arrange the vertices by (one of their possible) topological ordering:

[2] It takes linear time since we can avoid a $O(n \log n)$ sort: given a fixed number of contiguous, unique values (the post-orders go from $1 \to 2n$), we can just put the vertex into its slot in a preallocated $2n$-length array. DFS takes linear time, too, so it's $O(n + m)$ overall.



ANSWER: $\{X, Y, Z, U, W\}$, though that's only one of the three possible answers. In all cases, X and Y must go first.

After we've transformed a graph based on its topological ordering, some vertices have important properties:

- a **source** vertex has no *incoming* edges ($d_{\text{in}}(v) = 0$)
- a **sink** vertex has no *outgoing* edges ($d_{\text{in}}(v) = 0$)

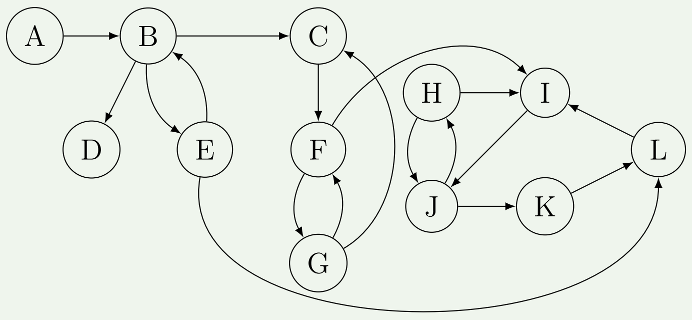
A DAG will always have at least one of each. By this rationale, we can topologically sort a DAG by repeatedly removing sink vertices until the graph is empty. The question now becomes: how do we find sink vertices?

First, we'll answer a related question: what's the analog of a connected component in generic digraphs?

3.4 Strongly-Connected Components

Vertices v and w are **strongly connected** if there is a path from $v \to w$ and vice-versa, from $w \to v$. Then, a **strongly-connected component** in a digraph is a maximal set of strongly-connected vertices.

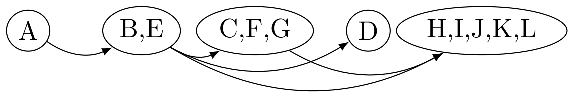
Identify the strongly-connected components in this directed graph:



ANSWER: There are three components, and they are highlighted below.

The highlighted components are rendered in Figure 3.2.

Consider the meta-graph of the strongly-connected components presented in the example above: what if we collapsed each component into a single vertex? The labeled result (see Figure 3.2) is actually a DAG in itself!



This will always be the case, giving us a nice property:[3]

Property 3.2. *Every directed graph is a directed acyclic graph of its strongly-connected components.*

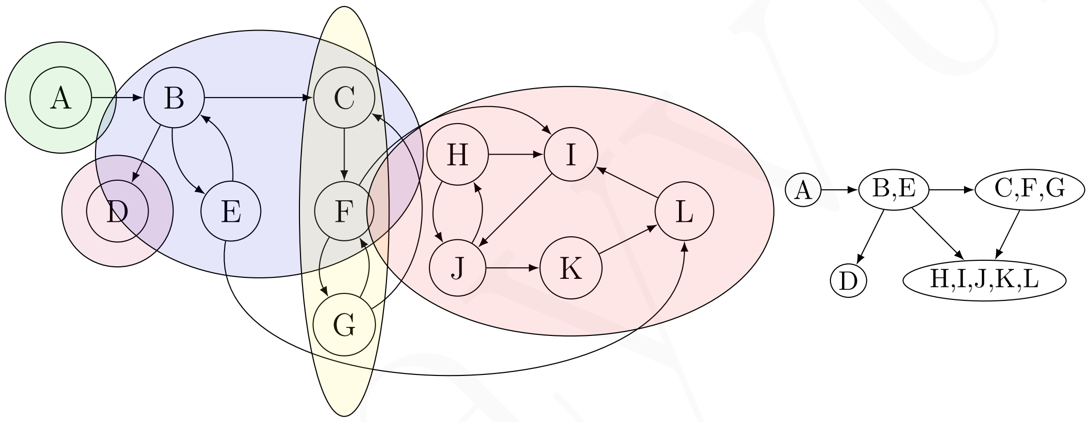


Figure 3.2: A directed graph broken up into its five strongly-connected components, and the resulting meta-graph from collapsing these components into a single "meta-vertex" (again, ignore the implication that vertex F is part of the blue component—this is just a visual artifact).

Let's discuss the algorithm that finds the strongly-connected components; in fact, the *order* in which it finds these components will be its topologically-sorted order (remember, that's in order of decreasing post-visit numbers). We'll actually achieve this in linear time with two passes of DFS.

3.4.1 Finding SCCs

Remember our basic idea for topologically sorting a graph: find a sink, rip it out, and repeat until the graph is empty. The idea for finding SCCs is similar: we're going to find a sink *SCC* (that is, a strongly-connected component that is a sink in its metagraph, so it only has incoming edges) and output it, repeating until the graph is empty.

[3] The proof of this is fairly straightforward: if there was a cycle in the metagraph, then there is a way to get from some vertex S to another vertex S' and back again (by definition). However, S and S' can't connected to each other, because if they were, they'd be a single strongly-connected component. Thus we have a contradiction, and post-SCC metagraph must be a DAG. □

That begs the question: how do we find a sink SCC?[4] More specifically, as explained in footnote 4, how do we find any vertex within a sink component?

A key property holds that will let us find such a vertex:

Property 3.3. *In a general directed graph, the vertex v with the* ***highest*** *post-visit order number will always lie in a* ***source*** *strongly-connected component.*

Given this property, how do we find a vertex w in a *sink* SCC? Well, it'd be great if we could invert our terms: we can find a source SCC and want a sink SCC, but what if we could find sink SCCs and wanted to find a source SCC? Well, what if we just... reversed the graph? So now sink vertices become sources and vice-versa, and we can use the property above to find a source-in-reversed-but-sink-in-original vertex!

For a digraph $G = (V, E)$, we look at $G^R = (V, E^R)$, the reverse graph of G, where $E^R = \{(w, v) : (v, w) \in E\}$, the reverse of every edge in E. All of our previous notions hold: the source SCCs in G become the sink SCCs in G^R and vice-versa, so the topologically-sorted DAG is likewise reversed. From a high level, the algorithm looks as follows:

1. First, create the reverse graph G^R.
2. Perform DFS on it to create a traversal tree.
3. Sort the vertices decreasing by their post-order numbers, identifying the vertex v to start DFS from in the original graph.
4. Run the standard connected-component algorithm using DFS (from algorithm 3.2, increasing the component whenever exploration of a vertex ends) on G starting at v to label each SCC.
5. When DFS terminates for a component, set v to the vertex to the one with the next-highest post-order number and go to Step 4.
6. The output is the metagraph DAG in reverse topological order.

3.5 Satisfiability

This is an application of our algorithms for determining (strongly-)connected components: solving **SAT** or **sat**isfiability problems.

[4] Though the same rationale of "find one and remove it" applies to source SCCs, sinks are easier to work with. If v is in a sink SCC, then EXPLORE (v) visits all of the vertices in the SCC and nothing else; thus, all visited vertices lie in the sink. The same does not apply for source vertices because exploration does not terminate quickly.

3.5.1 Notation

First, we need to define what a Boolean formula is; we're given:

n **variables**:	$x_1, x_2, \dots, x_n$
$2n$ **literals**:	$x_1, \overline{x_1}, x_2, \overline{x_2}, \dots, x_n, \overline{x_n}$

combined using the logical operators AND ($\wedge$) and OR ($\vee$) into **conjunctive normal form** (CNF).

A **clause** is the OR of several literals: for example, $x_3 \vee \overline{x_5} \vee \overline{x_1}$; to "satisfy" a clause, at least one of its members must be true. Finally, a **formula** in CNF is the AND of m clauses; for example:

$$f = \underbrace{(x_2)}_{\text{clause}} \wedge \underbrace{(\overline{x_3} \vee x_4)}_{\text{clause}} \wedge \underbrace{(x_3 \vee \overline{x_5} \vee \overline{x_1} \vee x_2)}_{\dots} \wedge (\overline{x_2} \vee \overline{x_1})$$

To satisfy f, at least one **literal** in each **clause** must be true. For example, $x_1 = F$, $x_2 = T$, and $x_3 = F$ satisfies the above formula. Any formula can be converted to CNF, but its size may blow up quickly.

3.5.2 An SAT Problem

Given an input formula f in CNF with n variables and m clauses, we want to find the assignment (T or F to each variable) that **satisfies** f, if one exists (or "no" otherwise).

EXAMPLE 3.2: SAT

Given the following formula,

$$f = (\overline{x_1} \vee \overline{x_2} \vee x_3) \wedge (x_2 \vee x_3) \wedge (\overline{x_3} \vee \overline{x_1}) \wedge$$

assign a value to each x_i to satisfy f.

ANSWER: $x_1 = F, x_2 = T, x_3 = F$ is a viable solution.

Generally-speaking, a k-SAT problem limits the clauses to having $\leq k$ elements, so the above example is 3-SAT. We'll see later than k-SAT is NP-complete for all $k \geq 3$, but there's a polynomial time algorithm for 2-SAT.

3.5.3 Solving 2-SAT Problems

Since we are limiting our clauses to have up to two elements, we can break any formula down into two categories: either all of the clauses have two elements, or at least one is a **unit clause**.

Simplifying

A unit clause is a clause with a single literal, such as (x_4). If a formula f has unit clauses, we know more about the satisfiability conditions and can employ a basic strategy:

1. Find a unit clause, say the literal a_i.
2. Satisfy it, setting $a_i = T$.
3. Remove clauses containing a_i (since they're now satisfied) and drop the literal $\overline{a_i}$ from any clauses that contain it.
4. Let f' be the resulting formula, then go to Step 1.

Obviously, f is satisfiable iff f' is satisfiable. We repeatedly remove unit clauses until either (a) the whole function is satisfied or (b) only two-literal clauses exist. We can now proceed as if we have two-literal clauses.

Graphing 2-SATs

Let's convert our n-variable, m-clause formula (where every clause has two literals) to a directed graph:

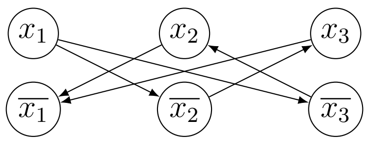
- $2n$ vertices corresponding to $x_1, \overline{x_1}, \ldots, x_n, \overline{x_n}$.
- $2m$ edges corresponding to two "implications" per clause.

In general, given we have the clause $(\alpha \vee \beta)$, the implication graph has edges $\overline{\alpha} \to \beta$ and $\overline{\beta} \to \alpha$.

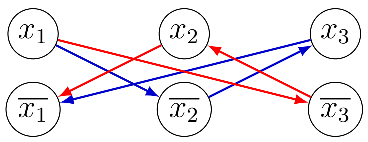
For example, given

$$f = (\overline{x_1} \vee \overline{x_2}) \wedge (x_2 \vee x_3) \wedge (\overline{x_3} \vee \overline{x_1})$$

the resulting graph would be:



Now, we follow the path for a particular literal. For example, here are the implication paths starting from $x_1 = T$:



Clearly, there's a path $x_1 \rightsquigarrow \overline{x_1}$, and obviously that implication is nonsense—$x_1 \implies \overline{x_1}$ is a contradiction. However, if $x_1 = F$, we have no leads (that is, $\overline{x_1}$ is a sink vertex), so it *might* be okay?

Obviously, if $x_1 = F$ also led to an implication path resulting in a contradiction, we could conclude that f is not satisfiable. Purely from the perspective of graph theory, having a path from x_1 to $\overline{x_1}$ and vice-versa implies that they're part of a connected component.

> **Property 3.4.** *If a literal a_i and its inversion $\overline{a_i}$ are part of the same strongly-connected component, then f is not satisfiable.*

With this property, we can prove when f is satisfiable: if, for every variable x, x_i and $\overline{x_i}$ are in *different* strongly-connected components, then f is satisfiable. Here's an algorithm that adds a little more structure to this basic intuition:

1. Find the sink SCC, S.
2. Set $S = T$; that is, satisfy all of the literals in S.
3. Since these are all tail-ends of their implications, the head will implicitly be satisfied. Thus, we can rip out S from the graph and repeat.
4. Then, $\overline{S}$ will be a *source* SCC, so that setting $\overline{S} = F$ has no effect (setting the head of an implication to false means we don't need to follow any of its outgoing edges).

This relies on an important property which deserves a proof.

> **Property 3.5.** *If $\forall i$, the literals x_i and $\overline{x_i}$ are in different strongly-connected components, then:*
>
> $$S \text{ is a sink SCC} \iff \overline{S} \text{ is a source SCC}$$

Given this property, we can formalize the 2-SAT algorithm a little further:

1. First, assume that the Boolean function in CNF-form, f, only has two-literal clauses (since we can force that to be the case by simplifying unit clauses).
2. Construct the graph G for f.
3. Find a sink SCC, S, using DFS and set $S = T$ and $\overline{S} = F$.
4. Remove both S and $\overline{S}$.
5. Repeat until the graph is empty.

The only complex task here is finding SCCs which we know takes linear time, so the overall running time is $O(n + m)$.

3.6 Minimum Spanning Trees

The input to the **minimum spanning tree** problem (or MST) is an undirected graph $G = (V, E)$ with weights $w(e)$ for $e \in E$. The goal is to find the minimal-size, minimum weight, connected subgraph that "spans" the graph. Naturally, the overall weight of a tree is the sum of the weights of edges.

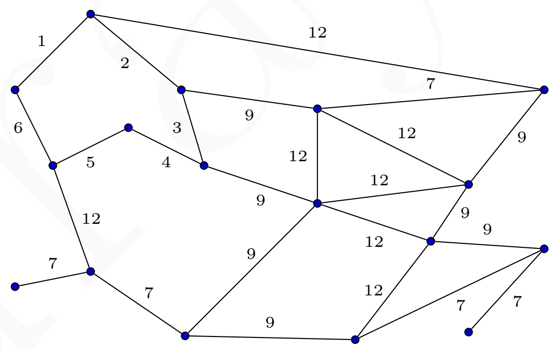
Note some basic properties about trees:

- A tree on n vertices has exactly $n - 1$ edges.
- Exactly one path exists between every pair of vertices.
- *Any* connected $G = (V, E)$ with $|E| = |V| - 1$ is a tree.

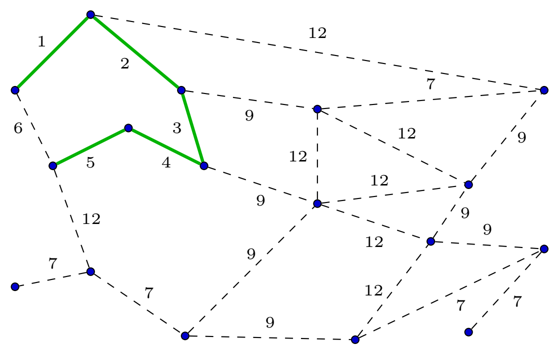
  This last point bears repeating, as it will be important in determining the minimum cut of a graph, later. The only thing necessary to prove that a graph is a tree is to show that its edge count is its vertex count minus one.

3.6.1 Greedy Approach: Kruskal's Algorithm

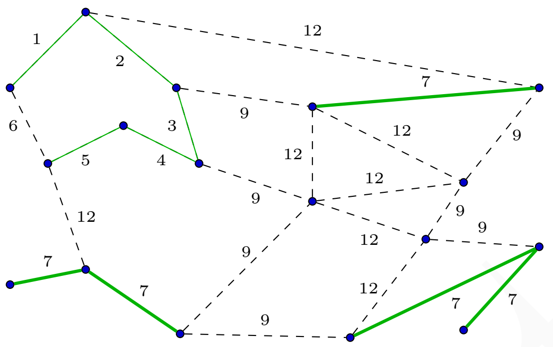
We'll first attempt to build the minimum spanning tree by consistently building up from vertices with the lowest cost without creating any cycles. Our running example graph will be:



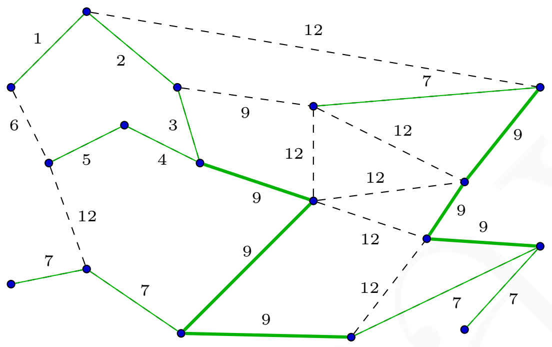
The first 5 edges are extremely easy to determine, but the 6-cost edge would cause a cycle:

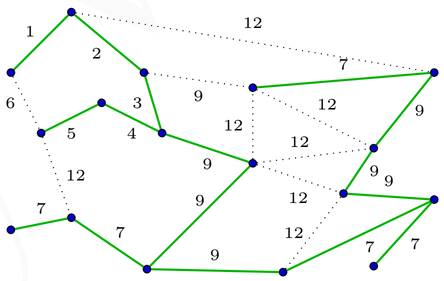


All of the 7-cost edges can be added in without trouble—notice that them not being connected is not relevant:



Finally, all but the last 9-cost edge can be added in to create a final MST for the graph. Notice that the vertices touched by the unused 9-cost edge are connected by a longer path.





This is **Kruskal's algorithm**, and its formalized in algorithm 3.3, below. Its **running time** is $O(m \log n)$ overall, since sorting takes $O(m \log n)$ time and checking for cycles takes $O(\log n)$ time using the **union-find** data structure and operates on m items.[5] Its correctness can be proven by induction on X, the MST for a subgraph of G, as we add in new edges.

3.6.2 Graph Cuts

The proof of correctness of Kruskal's algorithm (omitted from the above) is deeply related to **graph cuts**, which are our next topic of discussion. A cut in a graph is a set of edges that partitions the graph into two subgraphs. Formally, for an undirected graph $G = (V, E)$ and the partition $V = S \cup \overline{S}$, the cut is:

$$\begin{aligned} \text{cut}\left(S, \overline{S}\right) &= \{(v, w) \in E : v \in S, w \in \overline{S}\} \\ &= \text{edges crossing } S \leftrightarrow \overline{S} \end{aligned}$$

An cut on our running example graph is demonstrated in Figure 3.3. We'll look at determining both the minimum cut—the fewest number of edges necessary to divide a

[5] Reference *Algorithms*, pp. XX for details on the union-find data structure.

Algorithm 3.3: Kruskal(·), a greedy algorithm for finding the minimum spanning tree of a graph.

Input: An undirected graph, $G = (V, E)$ and its edge weights, $w(e)$.
Result: The minimum spanning tree of G.

```
Sort $E$ by ascending weight (via MergeSort($E$), etc.)
Let $X := \emptyset$
foreach $e = (v, w) \in E$ do                                    // in sorted order
    if $X \cup e$ does not create a cycle then
        $X = X \cup e$
    end
end
return $X$
```

graph into two components—and the maximum cut—the cut of the largest size—later when discussing optimization problems.

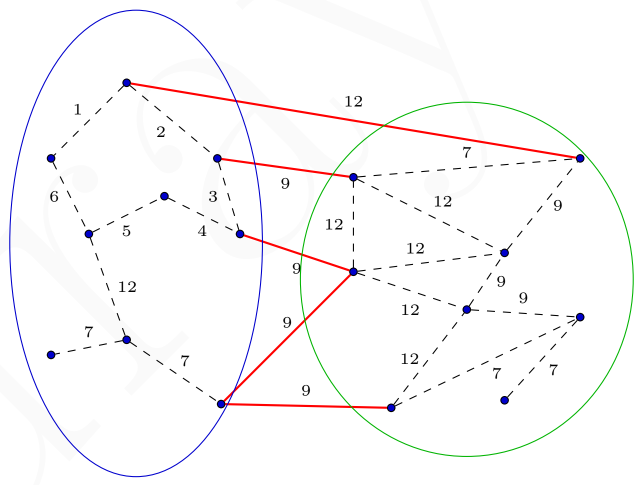


Figure 3.3: The edges in red cut the graph into the blue and green subgraphs.

Cuts come with an important property that becomes critical in proving correctness of MST algorithms. In one line, its purpose is to show that **any minimum edge across a cut will be part of an MST.**[6]

[6] Though its important to understand the proof of correctness of Property 3.6, it's left out here because the lectures cover the material to a sufficient degree of formality.

Property 3.6. *For an undirected graph, $G = (V, E)$, given:*

- *a subset of edges that is also a subset of a minimum spanning tree T of G (even if T is unknown); that is, for:*

  $$X \subset E \text{ and } X \subset T$$

- *and a subset of vertices $S \subset V$ where no edge of X is in*

  $$\text{cut}\left(S, \overline{S}\right)$$

any of the minimum-weight edges, e^*, in the above cut will be in a new minimum spanning tree, T'. That is:

$$X \cup e^* \subset T'$$

Note that T' after adding the e^* described in Property 3.6 and the original (potentially-unknown) T may be different MSTs, but the purpose is to find *any* MST rather than a specific MST. Critically, the weight of T' is *at most* the weight of T, so we're always improving.

3.7 Flow

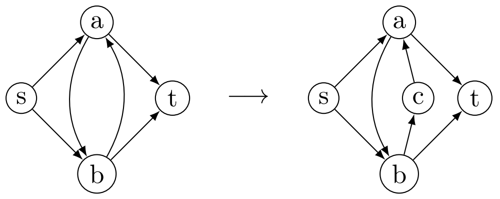
The idea of **flow** is exactly what it sounds like: given some resource and a network describing an infrastructure for that resource, a flow is a way to get said resource from one place to another. To model this more concretely as a graph problem, we have source and destination vertices and edges that correspond to "capacity" between points; a flow is a path through the graph.

Given such a network, it's not unreasonable to try to determine the **maximum flow** possible: what's the maximum amount of resource sendable from $s \to t$ without exceeding any capacities?

Formally, the **input** is a *flow network*: a directed graph $G = (V, E)$, a designated $s, t \in V$, and a capacity for each $e \in E : c_e > 0$. The **goal** is to *find a flow*—that is, a set of "usages" on each edge—that does not exceed any edge's capacity and maximizes the incoming flow to t.

Notice from the example below that cycles in a flow network are not problematic: there's simply no reason to utilize an *entire* cycle when finding a flow.

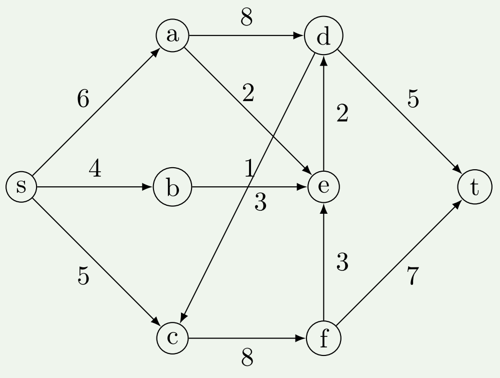
There's a small simplification we can make for certain types of networks. Consider the following simple flow network (left), which contains the **anti-parallel** edges $a \longleftrightarrow b$, and its alternative (right) which does not:



We'll see why the latter version is preferable soon, but just note that the maximum flow on both will be equivalent if $c_{(b,c)} = c_{(c,a)} = c_{(a,b)}$ (that is, the capacities of the broken-up edges match the capacity of the single, anti-parallel edge).

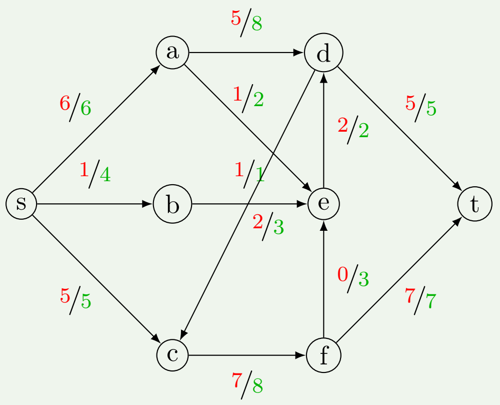
EXAMPLE 3.3: Max-Flow

Given the following graph:



determine a flow of maximum size that satisfies the edge constraints.

The solution is as follows:



This is obviously the maximum flow because the total capacity incoming to t is 12, so we've reached the hard cap. Namely,

$$\text{size}(f) = \underbrace{6+1+5}_{\text{outgoing}} = \underbrace{5+7}_{\text{incoming}} = \boxed{12}$$

3.7.1 Ford-Fulkerson Algorithm

The running time of the **Ford-Fulkerson algorithm** is **pseudo-polynomial** in $O(C\,|E|)$, where C is the theoretical maximum output flow to t (the sum of its incoming vertices' capacities).

This analysis relies on a (huge) assumption that all capacities are integers; then, the algorithm guarantees that the flow increases by ≥ 1 unit per round. Since there are $\leq C$ rounds to the algorithm, and a single round takes $O(n+m)$ time (dominated by path-finding), it requires $O(mC)$ time in total (if we assume $|E| \geq |V|$ to simplify).

ALGORITHM 3.4: The Ford-Fuklerson algorithm for computing max-flow.

Input: A flow network: a digraph $G = (V, E)$, a capacity for each $e \in E$, c_e, and start/end vertices $s, t \in V$.
Result: The maximal flow graph, where f_e indicates the maximum flow along the edge $e \in E$.

```
$\forall e \in E : f_e = 0$
while a path $s \rightsquigarrow t$ exists do
    Build the residual network, $G^f$ for the current flow, $f$.
    Find any path $P = s \rightsquigarrow t$ in $G^f$.
    if no such path then
        return $f$
    end
    Given $p$, let $c(p)$ be the minimum capacity along $p$ in $G^f$: $c(p) = \min_{c_e} e \in P$.
    Augment $f$ by $c(p)$ units along $P$.
end
```

Its correctness follows from the max-flow = min-cut theorem, shown later.

3.7.2 Edmonds-Karp Algorithm

This algorithm fundamental varies in the way it augments f. Rather than using *any* path (found via depth-first search, for example), it uses the *shortest* path found by breadth-first search. When this is the case, it can be proven that it takes at-most mn rounds to find the maximum flow; thus, the overall running time of the **Edmonds-Karp algorithm** is $O(nm^2)$.

FUN FACT: Record-Breaker

James Orlin achieved a max-flow algorithm with a $O(mn)$ run time in 2013.

3.7.3 Minimum Cut

3.7.4 Application: Image Segmentation

PART II

Additional Assignments

THIS part of the guide is dedicated to walking through selected problems from the homework assignments for U.C. Berkeley's CS170 (*Efficient Algorithms and Intractable Problems*) which uses the same textbook. I specifically reference the assignments from spring of 2016 because that's the last time Dr. Umesh Vazirani (an author of the textbook, *Algorithms*) appears to have taught the class. Though the course follows a different path through the curriculum, I believe the rigor of the assignments will help greatly in overall understanding of the topic.

I'll try to replicate the questions that I walk through here; typically, the problems themselves are heavily inspired by the exercises in *Algorithms* itself.

Contents

Homework #0

This diagnostic assignment comes from the course rather than CS170's curriculum. It should be relatively easy to follow by anyone who has taken an algorithms course before. If it's not fairly straightforward, I highly recommend putting in extra prerequisite effort now rather than later to get caught up on the "basics."

4.1 Problem 1: From *Algorithms*, Ch. 0

Recall the definition of big-O:

> *Given two functions, f and g, we can say that $f = O(g)$ if:*
>
> $\exists c \in \mathbb{N}$ such that $f \leq c \cdot g$
>
> *That is, there exists some constant c that makes g always bigger than f. In essence, $f(n) = O(g(n))$ is analogous to saying $f \leq g$ with "big enough" values for n.*

Note that if $g = O(f)$, we can also say $f = \Omega(g)$, where Ω is somewhat analogous to $\geq$. And if both are true (that is, if $f = O(g)$ *and* $f = \Omega(g)$) then $f = \Theta(g)$. With that in mind...

(a) Given

$$f(n) = 100n + \log n$$
$$g(n) = n + \log^2 n$$

we should be able to see, relatively intuitively, that $\boxed{f = \Omega(g)}$, (B). This is the case because $100n = \Theta(n)$ since the constant can be dropped, but $\log^2 n = \Omega(\log n)$, much like $n^2 \geq n$.

(b) We're given

$$f(n) = n \log n$$
$$g(n) = 10n \log 10n$$

Since constants are basically irrelevant in big-O notation, it should be self-evident that f and g are equal with large-enough n, thus $\boxed{f = \Theta(g)}$, (C).

(c) Given

$$f(n) = \sqrt{n}$$
$$g(n) = \log^3 n$$

we shouldn't let the exponents confuse us. We are still operating under the basic big-O principle that *logarithms grow slower than polynomials*. The plot clearly reaffirms this intuition; the answer should be $\boxed{f = \Omega(g)}$, (B).

(d) Given

$$f(n) = \sqrt{n}$$
$$g(n) = 5^{\log n}$$

there should be no question that $\boxed{f = O(g)}$, since an exponential will grow much, *much* faster than a linear polynomial, and $\sqrt{n} \le n$ so it continues to hold there, too.

4.2 Problem 2: Big-Ordering

(a) At first glance, we should be able to identify at least three groups: linear, logarithmic, and the rest. Note that I will assume all of the logarithms are in base-2.

In the logarithmic group (that is, $f = \Theta(\log n)$) we have:

- $\log n$
- $\log n^2$ (remember the power rule for logarithms: $\log n^k = k \cdot \log n$)

Then, in the linear group $f = \Theta(n)$:

- n
- $2n + 5$

Finally, we have the remaining groups which all have one member:

- In the group $f = \Theta(n \log n)$ we have $n \log n + 2019$.
- In the groups $f = \Theta(\sqrt{n})$, $f = \Theta(n^{2.5})$, $f = \Theta(2^n)$, and $f = \Theta(n \log^2 n)$ is the exact member itself.

The order of the groups should be relatively obvious: logarithms, polynomials, and finally exponentials. Specifically, $\log n, n \log n, n \log^2 n, \sqrt{n}, n, n^{2.5}, 2^n$.

(b) We're asked to prove that

$$S(n) = \sum_{i=0}^{n} a^i = 1 + a + a^2 + \ldots + a^n = \frac{a^{n+1} - 1}{a - 1}$$

This result was also used in (5.1) and elaborated-on in this aside as part of Geometric Growth in a supplemental homework. The proof comes from polynomial long division. The deductions for the big-O of $S(n)$ under the different a ratios are also included in that problem.

Homework #1

CS 170: Homework #1

DON'T forget that in computer science, our logarithms are always in base-2! That is, $\log n = \log_2 n$. Also recall the definition of big-O, which we briefly reviewed for section 4.1

5.1 Compare Growth Rates

I'll only do a handful of these since they should generally be pretty straightforward and mostly rely on recalling esoteric algebraic manipulation rules.

(a) Given

$$\begin{aligned} f(n) &= n^{1/2} \\ g(n) &= n^{2/3} \end{aligned}$$

we can determine that $\boxed{f = O(g)}$ because:

$$\begin{aligned} f &\overset{?}{\leq} c \cdot g \\ \frac{f}{g} &\overset{?}{\leq} c \\ \frac{n^{1/2}}{n^{2/3}} = n^{-1/6} &\leq 1 \qquad \forall n \in \mathbb{N} \end{aligned}$$

That is, there are no natural numbers that would make this fraction $\frac{1}{n^{1/6}}$ larger than $c = 1$. Note that we don't actually need this level of rigor; we can also just say: $f = O(g)$ because it has a smaller exponent.

(b) Given

$$\begin{aligned} f(n) &= n \log n \\ g(n) &= (\log n)^{\log n} \end{aligned}$$

Considering the fact that $g(n)$ is an exponential while $f(n)$ is a polynomial, we should intuitively expect $\boxed{f = O(g)}$. By expanding the logarithm with a quick undo-redo (that is, by adding in the base-2 no-op), we can make this even more evident:

$$\begin{aligned}
g(n) &= (\log n)^{\log n} \\
&= 2^{\log\left((\log n)^{\log n}\right)} && \text{recall that applying the base to a logarithm is a no-op, so: } b^{\log_b x} = x \\
&= 2^{\log n \cdot \log\log n} && \text{we can now apply the power rule: } \log x^k = k \log x \\
&= 2^{\log n^{\log\log n}} && \text{then the power rule of exponents: } x^{ab} = (x^a)^b \\
&= n^{\log\log n} && \text{and finally undo our first step}
\end{aligned}$$

Very clearly, n^m will grow *much* faster than some nm.

5.2 Geometric Growth

We need to prove the big-Os of geometric series under all conditions:

$$\sum_{i=0}^{k} c^i = \begin{cases} \Theta(c^k) & \text{if } c > 1 \\ \Theta(k) & \text{if } c = 1 \\ \Theta(1) & \text{if } c < 1 \end{cases}$$

We'll break this down piece-by-piece. For shorthand, let's say $f(k) = \sum_{i=0}^{k} c_i$.

Case 1: $c < 1$. Let's start with the easy case. Our claim $f = \Theta(1)$ means:

$$\begin{aligned} &\exists m \text{ such that } f(k) \leq m \cdot 1 \qquad \textbf{and} \\ &\exists n \text{ such that } f(k) \geq n \cdot 1 \end{aligned}$$

Remember from calculus that an infinite geometric series for these conditions converges readily:

$$\sum_{i=0}^{\infty} c^i = \frac{1}{1-c}$$

This is a hard-cap on our series (since $k \leq \infty$), so we can confidently say that $f = O(1)$ for $m = \frac{1}{1-c}$. We also have to show the inverse, too, which is trivial: just let $n = 1$. Since the sum can only get bigger with k, the smallest is $f(k = 0) = 1$. □

Case 2: $c = 1$. This case is actually even easier… Notice that the series' sum is always just k:

$$f(k) = \sum_{i=0}^{k} c^i = \underbrace{1 + 1 + \ldots + 1}_{k+1 \text{ times}} = k + 1$$

meaning our claim of $f = \Theta(k)$ is self-evident. □

Case 3: $c > 1$**.** The final case results in an increasing series; the formula for the sum for k terms is given by:

$$f(k) = \sum_{i=0}^{k} c^i = \frac{1 - c^{k+1}}{1 - c} \tag{5.1}$$

Our claim is that this is capped by $f(k) = \Theta(c^k)$. We know that obviously the following holds true:[1]

$$c^{k+1} > c^{k+1} - 1 > c^k$$

Notice that this implies $c^{k+1} - 1 = O(c^{k+1})$, and $c^{k+1} - 1 = \Omega(c^k)$. That's really close to what we want to show... What if we divided the whole thing by $c - 1$?

$$\frac{c^{k+1}}{c-1} > \frac{c^{k+1} - 1}{c-1} > \frac{c^k}{c-1}$$

Now notice that the middle term is $f(k)$! Thus,

$$\frac{c}{c-1} c^k > f(k) > \frac{1}{c-1} c^k$$

Meaning it *must* be the case that $f = \Theta(c^k)$ because we just showed that it had c^k as both its upper and lower bounds (with different constants $m = \frac{c}{c-1}, n = \frac{1}{c-1}$). □

Quick Maffs: Deriving the geometric summation.

We want to prove (5.1):

$$f(k) = \sum_{i=0}^{k} c^i = \frac{1 = -c^{k+1}}{1 - c}$$

The proof is a simple adaptation of this page for our specific case of $i = 0$ and $a_0 = 1$. First, note the complete expansion of our summation:

$$\begin{aligned} \sum_{i=0}^{k} c^i &= 1 + c + c^2 + \ldots + c^k \\ &= c^k + c^{k-1} + \ldots + c^2 + c + 1 \end{aligned}$$

or, more conventionally for polynomials...

[1] This is true for $k > 0$, but if $k = 0$, then we can show $f(0) = \Theta(c^0) = O(1)$ just like in *Case 2*.

A basic property of polynomials is that dividing $x^n - 1$ by $x - 1$ gives:

$$\frac{x^n - 1}{x - 1} = x^{n-1} + x^{n-2} + \ldots + x^2 + x + 1$$

By reverse-engineering this, we can see that our summation can be the result of a similar division:

$$\frac{c^{k+1} - 1}{c - 1} = c^k + c^{k-1} + \ldots + c^2 + c + 1 \qquad \square$$

5.3 Recurrence Relations

Refer to Solving Recurrence Relations for an overview.

(a) We can solving some of these using the Master theorem.

(i) $T(n) = 3T\left(\frac{n}{4}\right) + 4n$

We have $d = 1, \log_b a = \log_4 3 \approx 0.79$, so the first branch applies: $\boxed{O(n)}$.

(ii) $45T\left(\frac{n}{3}\right) + 0.1n^3$

We have $d = 3, \log_3 45 \approx 3.46$, so the third branch applies: $\boxed{O\left(n^{3.46}\right)}$.

(iii) $T(n) = T(n-1) + c^n$

For this one, we will need to derive the relation ourselves. Let's start with some substitutions to see if we can identify a pattern:

$$\begin{aligned} T(n) &= T(n-1) + c^n \\ &= \underbrace{T(n-2) + c^{n-1}}_{T(n-1)} + c^n \\ &= \underbrace{T(n-3) + c^{n-2}}_{T(n-2)} + c^{n-1} + c^n \\ &= T(n-i) + \sum_{j=0}^{i-1} c^{n-j} \end{aligned}$$

Now it's probably safe to assume that $T(1) = O(1)$, so we can stop at $i = n - 1$, meaning we again have a geometric series:

$$T(n) = O(1) + c^n + c^{n-1} + \ldots + 1$$

whose complexity we know is $\boxed{O(c^n)}$.[2]

[2] We're being cheeky here: the biggest case of $O(c^n)$ covers all cases of c for a geometric series. It's critical to remember the imprecision of $O(\cdot)$ relative to $\Theta(\cdot)$!

Homework #2

CS 170: Homework #2

6.1 Peak Element

Prof. B. Inary just moved to Berkeley and would like to buy a house with a view on Euclid Ave. Needless to say, there are n houses on Euclid Ave: they are arranged in a single row, and have distinct heights. A house "has a view" if it is taller than its neighbors. For example, if the houses heights were $[2, 7, 1, 8]$, then 7 and 8 would "have a view."

Devise an efficient algorithm to help Prof. Inary find a house with a view on Euclid Ave. (If there are multiple such houses, you may return any of them.)

I mean... this can't possibly be that easy, right? It's almost trivial to imagine a simple, linear-time algorithm that just compares each building to its neighbors. Well duh: it's not supposed to be that easy. We can also use divide-and-conquer technique to achieve $O(\log n)$ performance.

Algorithm 6.1: A logarithmic-time algorithm for finding a house with a view.

```
Input: $\mathcal{B} = \{b_1, b_2, \dots, b_n\}$, the list of house heights.
$h = \left\lfloor \frac{n}{2} \right\rfloor$
if $b_h > b_{h-1}$ and $b_h > b_{h+1}$ then
    return true
else if $b_h < b_{h-1}$ then
    return HasView($\{b_{1..h-1}\}$)
end
return HasView($\{b_{h+1..n}\}$)
```

The key insight actually comes from the linear approach: if a chosen house does *not* have a view and its left-neighbor is taller, then somewhere in the left half there must be a house with a view. Namely, either that very house, or the tallest house in that half. The same logic can be applied for the right-neighbor being taller.

6.2 Majority Elements

An array $A[1...n]$ is said to have a **majority element** if more than half of its entries are the same. Given an array, the task is to design an efficient algorithm to tell whether the array has a majority element, and, if so, to find that element. The elements of the array are not necessarily from some ordered domain like the integers, and so there can be no comparisons of the form "is $A[i] > A[j]$?". (Think of the array elements as `GIF` files, say.) However, you can answer questions of the form: "is $A[i] = A[j]$?" in constant time.

This can be done in $O(n \log n)$ time with one approach, then $O(n)$ with another; both involve D&C. Both solutions below start with the hint provided in the instructions.

For convenience, let's say m is the number of elements needed for a majority. That is, $m = \left\lfloor \frac{n}{2} \right\rfloor + 1$. Also, lets let $\emptyset$ be a sentinel value indicating "no majority element."

6.2.1 A $O(n \log n)$ Solution

The assignment provides a useful hint:

> *Split the array A into two arrays A_1 and A_2 of half the size. Does knowing the majority elements of A_1 and A_2 help you figure out the majority element of A? If so, you can use a divide-and-conquer approach.*

By definition, for some a_i to be the majority element, $\textsc{Count}(A_1, a_i) + \textsc{Count}(A_2, a_i) \geq m$. Then, **it must *also* be the majority element in at least one of the subarrays**; if it's not, there's no way the sum of occurrences is $\geq m$. Thus, the majority element of A must be *one* of the majority elements of A_1, A_2 (if it exists). If the respective majorities are different, it's whichever one has more "support" in the respective array.

For the sake of convenience, let's assume $|A|$ is a power of two. We'll define a helper D&C algorithm below that returns the element that occurs most-frequently in A and the number of times it occurs, x. Then, determining majority is checking if $x \geq m$.

Base Case If the input array has two elements, it has a majority element if they're both the same. Thus, if $A = [a_1, a_2]$, we'd return $(a_1, 2)$ if $a_1 = a_2$ and $(\emptyset, 0)$ otherwise.

Divide We recurse on the two halves separately, splitting on the middle element, ending up with the two majorities and their counts (let $h = n/2$):

$$\begin{aligned} A_1 &= [a_1, \dots, a_{h-1}] & A_2 &= [a_h, \dots, a_n] \\ (j_1, c_1) &= \textsc{Major}(A_1) & (j_2, c_2) &= \textsc{Major}(A_2) \end{aligned}$$

Conquer We walk A_2 searching for j_1 and then walk A_1 searching for j_2 to determine each element's "support." The bubbled-up result is the one with more:

$$\begin{aligned} s_1 &= c_1 + |\{a_k \in A_2 \mid a_k = j_1\}| \\ s_2 &= c_2 + |\{a_k \in A_1 \mid a_k = j_2\}| \\ \text{return } &\begin{cases} (j_1, s_1) & \text{if } s_1 > s_2 \\ (j_2, s_2) & \text{otherwise} \end{cases} \end{aligned}$$

Obviously if $j_1 = j_2$, we just return $c_1 + c_2$ instead. This also covers the case of both returning $\emptyset$; if only one does, we walk for whichever one isn't.

Complexity

The running time is the same as `mergesort`: halving on each recursion with $O(n)$ extra work comes out to $O(n \log n)$ total work.

Proof of Correctness

The first claim to prove is that the majority element of A must also be the majority of A_1 and/or A_2. Suppose it's not. Let $c_1 = \textsc{Count}(A_1)$ and $c_2 = \textsc{Count}(A_2)$, then by our supposition, $c_1 < \frac{m}{2}$ and $c_2 < \frac{m}{2}$ by definition (since A_1 is half as large as A, you need half as many elements to be the majority: $m/2$). Obviously, then, $c_1 + c_2 < \frac{m}{2} + \frac{m}{2}$, so $c_1 + c_2 < m$, meaning it can't be the majority of A. This is a contradiction. $\square$

The second claim to prove is that returning the element with the most "support" will give us the majority element if there is one. Well at each level of recursion, we are returning the element that occurs most-frequently in $A_1 \parallel A_2$ (which is A). Obviously the majority element, if it exists, occurs the most-frequently in A, so it will be the final result. $\square$

6.2.2 A $O(n)$ Solution

This one also comes with a hint:

> *Pair up the elements of A arbitrarily to get about $^n/_2$ pairs.*
>
> - *Look at each pair: if the two elements are different, discard both of them; if they are the same, keep just one of them*
> - *If $|A|$ is odd, there will be one unpaired element. What should you do with this element?*
>
> *Show that after this procedure there are at most $^n/_2$ elements left, and that they have a majority element if A does.*

The first claim is relatively easy to demonstrate. In the best possible case (that is, the case with the maximum remaining elements) all numbers with a match are paired up. Since every other number gets dropped, there are obviously half as many left. □

The second claim is tougher to show. If A has a majority element a^*, there are two scenarios:

- If $|A|$ is odd, it's possible that every instance of a^* gets paired with something that's not a^* but then the remaining element MUST be a^* (otherwise there wouldn't be m of them).
- Otherwise (that is, both if $|A|$ is *not* odd AND if $|A|$ is odd and the "worst case" pairing does NOT occur), at least one instance of (a^*, a^*) occurs, so it remain in the array of "kept" elements.

  Why? Well if $|A|$ is even, then $m > \frac{n}{2}$. If every a^* is mapped to some other value, there must be $> 2 \cdot \frac{n}{2}$ of "other" values, which is a contradiction.

  Furthermore, if the non-worst-case pairing occurs in the odd-$|A|$ case, by definition there is such a pair. □

Thus, the resulting array of "kept" elements, A', must have the majority element in it; the odd-case straggler should always be included into it. So we know for sure that $a^* \in A'$, but is a^* still its majority element...? Find out on the next episode of *I Have No Idea What I'm Doing*, Season 2.

Homework #7

CS 170: Homework #7

Because the course follows *Algorithms* in a different order, relevant problems from CS170 are scattered throughout the homeworks. Thus, these are collected here in the order in which they're found.

7.1 The Quest Begins

You're on a quest to visit the magical city of Pimentel, and the only way to get there is through a narrow path across a mountain. A dragon lives on the mountaintop, and eats all those who dare intrude upon his mountain – unless they pay a sufficiently high bribe. You'll need to earn as many gold coins as you can in the n days before you have to face the dragon. There are two nearby villages where you can earn some money doing odd jobs. The villages publish lists $A[1..n]$ and $B[1..n]$, where $N[i]$ is the non-negative integer number of gold coins available to be earned by working at village N on day i. (You're confident that you're a hard enough worker that you can always earn all $N[i]$ coins available.) The travel time between the two villages is a day, so that if you're working at A on day d and decide to switch to B, you can start working there on day $d + 2$. You can start at either village on day 1. Design an efficient algorithm to find the highest number of gold coins you could earn in the next n days.

On any given day, you want to being doing the job that pays the most while also being able to do the next-best job efficiently. Let's define a subproblem $G(i)$ as being the highest amount of gold we can earn on day i. Then, our recurrence considers the best job in the same village while also considering the next-available job in the next village. Let's define auxiliary variables N, M to represent the village we're currently at and the other one. Then,

$$G(i) = G(i-1) + \max[N[i+1], M[i+2]]$$

Our base case is the best first job: $G(1) = \max[A[1], B[1]]$.

Is this valid? It's essentially a greedy approach, so definitely not.

Index of Terms