Algorithms

or: The Unofficial Notes on the Georgia Institute of Technology's **CS6515**: *Graduate Algorithms*

George Kudrayvtsev
george.k@gatech.edu

Last Updated: January 15, 2020

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.

Linky 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.

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 ≤ 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 ≤ 1 then
    return n
end
F := {0, 1}
foreach i ∈ [2, n] do
    F[i] = F[i − 1] + F[i − 2]
end
return F[n]
```

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

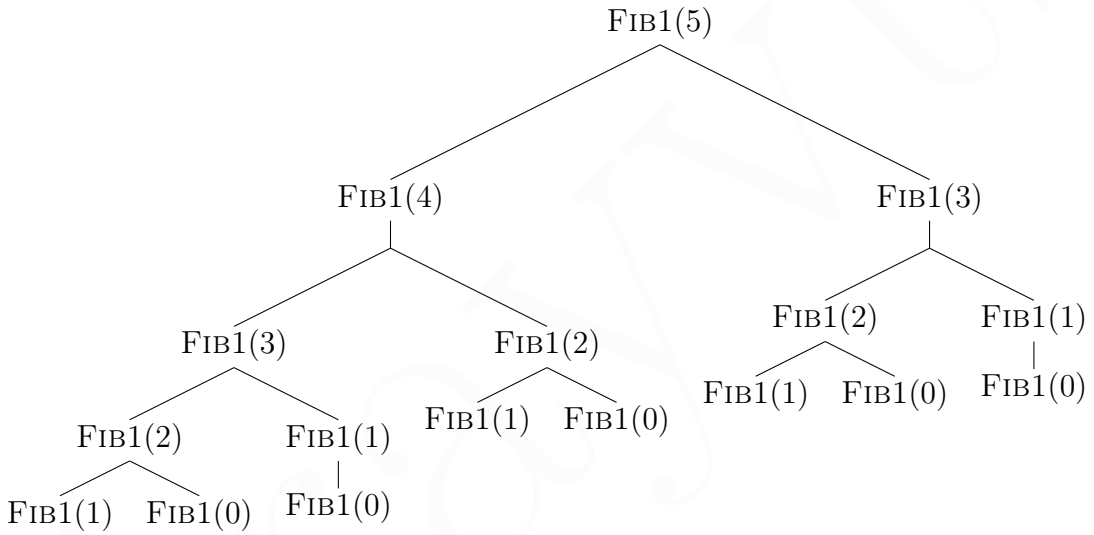
- *avoid repetitive work* and instead store it after computing it once. Identifying the overlapping subproblems (like the fact that F_{n-1} and F_{n-2} share large amounts of work) is a key part in developing a dynamic programming approach. This means you should not shy away from high memory usage when 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

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

Input: $\mathcal{S}$, a series of numbers.
Result: n, the length of the LIS of $\mathcal{S}$.

```
/* Initialize the lengths and maximums to a baseline.  */
```
$\mathcal{L} := \{1 \mid \forall i \in \mathcal{S}\}$
$\mathcal{M} := \{-\infty \mid \forall i \in \mathcal{S}\}$

foreach $x_i \in \mathcal{S}$ **do**
  for $j \in [0, i)$ **do**
    if $\mathcal{M}[j] < x_i$ **then**
      $\mathcal{L}[j] \mathrel{+}= 1$
      $\mathcal{M}[j] = x_i$
    end
  end
end
return $\max \mathcal{L}$

necessarily be the best one at *every* point in time as the series grows. This simple insight is formalized in algorithm 1.3.

The running time of the algorithm is $O(n^2)$ due to the double `for`-loop over (up to) n elements each; there are likely plenty of potential improvements to be made. Note the following additional insight that could boost performance: if we kept $\mathcal{M}$ sorted, we could identify which lengths to increase more efficiently via an $O(\log n)$ binary search. Though the sorting itself takes time... regardless, the point is that though there may be areas for improvement, our approach solves the task at hand.

1.2.2 Finding the Sequence

Let's kick it up a notch and find the sequence itself. Theoretically, we should be able to extend the previous algorithm 1.3 easily and store the entire sequence in $\mathcal{M}$ rather than just its last (and largest) element. In other words, $\mathcal{M}[i]$ is now another list rather than a single number.

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 4^{th} 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. we set increment

ℓ_i to $1 + \max_j\{\ell_j \mid \forall j < i\}$. 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.

If $L(i, i)$ is the LCS length, and its based on the previous "row" in the matrix:

$$L(i,i) = \begin{cases} 1 + \max_j\{L(i,j) \mid \forall j < i\} & \text{if } x_i = y_i \\ 1 + \max_j\{L(i,j) \mid \forall j < i\} & \text{if } x_i \neq y_i \end{cases}$$

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:

$$V = \{v_1, v_2, \ldots, v_n\}$$
$$W = \{w_1, w_2, \ldots, w_n\}$$

as well as a total capacity B.

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

(a) maximizes the total value: $\max_{S \in B} \sum_{i \in S} v_i$

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

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:

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

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 $K:\ 15\ \ 15\ \ 15\ \ 15$. 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

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: $W = \{w_1, \ldots, w_n\}$
Input: List of object values: $V = \{v_1, \ldots, v_n\}$
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.
```

$\mathbf{K}_{n+1 \times B+1} \leftarrow \mathbf{0}$
foreach $i \in [1..n]$ **do**
  foreach $b \in [1..B]$ **do**
    $x = \mathbf{K}[i-1, b]$
    if $w_i \leq b$ **then**
      $\mathbf{K}[i, b] = \max[x, v_i + \mathbf{K}[i-1, b-w_i]]$
    else
      $\mathbf{K}[i, b] = x$
    end
  end
end
return $K(n, B)$

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*

This section was haphazardly written out of order and is very incomplete. The intent was to arrive at the derivation of the Master theorem in order to solve recurrence relations easily for Homework #1... it got a little out of hand.

2.1 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.

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:

Lesson 10: DC3

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

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.

2.1.1 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:

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

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

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

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:

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

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:

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

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:

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

2.1.2 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:

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

2.1.3 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 $O\big(n^d\big)$; this is the **Master theorem** for recurrence relations:

For the general case of of the recurrence relation,

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

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

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

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

PART II

Homework Walkthroughs

This part of the notes is dedicated to walking through the homework assignments for this course. Everyone taking the course should have the PDFs readily available to them, so this will only feature the solutions (with allusions to the instruction).

Contents

Homework #0

This diagnostic assignment 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."

3.1 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 $\sqrt{n} \leq n$ so it continues to hold there, too.

3.2 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 (4.1) 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.

PART III

Additional Assignments

This part of the notes is dedicated to walking through 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.

PDFs of the homework assignments are available on my website, where I also may sometimes walk through certain problems in greater depth. The problems themselves are heavily inspired by the problems from *Algorithms* itself.

Contents

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 3.1

4.1 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

$$f(n) = n^{1/2}$$
$$g(n) = n^{2/3}$$

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

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

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

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

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.

4.2 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:

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

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:

$$f(k) = \sum_{i=0}^{k} c^i = \frac{1 - c^{k+1}}{1 - c} \tag{4.1}$$

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 (4.1):

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

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:

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

4.3 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 $\Theta(\cdot)$!

(b)

4.4 Integer Multiplcation

(a) For our baseline of truth, $11 \times 13 = 143$. To adopt the Al Khwarizmi technique to use a factor of 3, we need to... uhhhh...

Here are some examples of the technique:

11	13
3	52
1	156
	143

18	12
6	36
2	108
1	324
	216

15	13
5	39
1	117
	195

21	42
7	126
2	378
1	1134
	882

46	23
15	69
5	207
1	621
	1058

(b) The altered algorithm

4.5 More Integer Multiplcation

Index of Terms