These notes are serve as a concise presentation of topics taught in CS170, Efficient Algorithms and Intractable Problems, at UC Berkeley. Everything in these notes are in scope for exams, excluding sections specifically designated as [EXTRA] and anything in the footnotes.

In these notes, we will assume knowledge of programming, data structures and mathematical proofs.

We will use Pythonic psuedocode. Assume lists are implemented as Python lists, so accessing an element at a given index takes constant time.[1]

0 Motivation

Algorithms[2] form the foundations of modern computing technology. Wireless networks (WiFi), for example, were only made possible by an algorithm for efficient Fourier transforms, and GPS navigation applications by Dijkstra's algorithm. In this course, we will explore the (fast) Fourier transform and Dijkstra's, along with a multitude of other algorithms that form the theoretical foundations of computer science. During this process, we hope to impart insights on how to devise algorithms, prove their correctness, and analyze their time and space complexity - skills that are essential for any computer scientist or software engineer. Welcome to CS170!

1 Algorithms

An algorithm is a finite sequence of well-defined instructions used to perform a computation. There may exist many algorithms to solve a given problem, although for practical purposes, we focus on the most efficient ones.

When discussing an algorithm, we generally break the process down into three parts: a description of the algorithm, a proof of correctness, and runtime/memory analysis. We may rigorously express an algorithm by its pseudocode; or, more often, we opt for a high-level description to succinctly highlight the main ideas and omit implementation details. Both of these approaches are equally acceptable for presenting an algorithm.

Example: Finding the maximum element in a list of integers (assume the list is finite and nonempty, so the maximum is well-defined).

High-Level Description:

[1] This is because a Python list is implemented as a (dynamic) array (e.g Java ArrayList), and an array is stored as a contiguous block of memory. So, accessing at an index is some arithmetic, then following a pointer to a memory address.

[2] The term 'algorithm' is named after the 9th century mathematician and polymath al-Khwarizmi, known for creating fundamental arithmetic algorithms and inventing algebra, among other things.

Assign the first number in the list as the max element. For each remaining number in the list: if this number is larger than max, replace max with this number. When there are no numbers left in the list to iterate over, return max.

Pseudocode:

Algorithm 1 Maximum Element of Nonempty List

```
1: procedure GET_MAX(lst)
2:     max_element ← lst[0]
3:     for num in lst do
4:         if num > max_element then
5:             max_element ← num
6:     return max_element
```

Proof of Correctness:

Observe that the value of max_element is always assigned to be equal to some number in the provided list. This implies that max_element cannot be strictly greater than every number in the list, and is less than or equal to the true maximum element of the list. Simultaneously, the max variable must be greater than or equal to every value in the list due to the for loop. These two inequalities imply that the returned value is exactly equal to the maximum value of the list.

Runtime:

The initialization of the max is $O(1)$, and the loop does $O(1)$ for each element in the list, which has n elements. Thus, the overall runtime is $O(n)$.

2 Asymptotics

The widely used method to measure the performance of an algorithm by its running time (also known as "time complexity") is by counting the number of basic computer steps as a function of the size of the input(s), usually denoted by n. As most algorithms finish quickly on modern day computers when n is small, we are most interested in the asymptotic behavior, i.e when n is large. To achieve this, we use *big O notation*.

Intuitively, big O notation[3] allows us to compare two functions asymptotically; for two functions f and g, $f = O(g)$ is an analog of $f \leq g$.

Formally, let $f(n)$ and $g(n)$ be functions from the positive integers to the positive reals representing an algorithm's running time with an input of size n. We say $f(n) =$

[3]We are using Donald Knuth's definition of Big-O Notation. There is a slightly different definition used in analytic number theory.

$O(g(n))$ if there exists a constant $c > 0$ such that $f(n) \leq c \cdot g(n)$.[4]

Example: An algorithm that takes $f(n) = 5n^3 + 4n + 3$ steps for an input of size n is $O(n^3)$, as we may set $c = 12$. Similarly, the same algorithm is also $O(n^{100})$ and $O(2^n)$, however these are less informative. To avoid these absurdities, we generally consider the tightest big O bound we can.

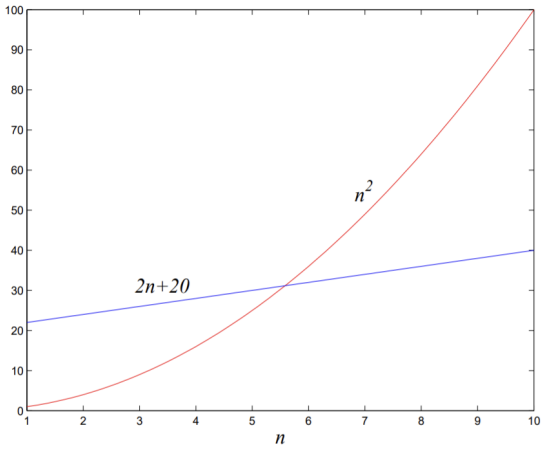
As stated before, $f(n) = O(g(n))$ is an analog of $f \leq g$. Similarly,

$f(n) = \Omega(g(n)) \iff g(n) = O(f(n))$ is an analog of $f \geq g$.

$f(n) = \Theta(g(n)) \iff (f(n) = O(g(n)) \wedge g(n) = O(f(n)))$ is an analog of $f = g$.

The default in this class is to use $O()$ even if $\Theta()$ is eligible.

Figure 1: n^2 grows faster than $2n + 20$. Source: DPV, pg. 16



Usefully, big O may also be defined in terms of limits. This is generally the preferred method for determining asymptotic relationship between two functions. The definitions are as follows:

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} < \infty \iff f(n) = O(g(n))$$
$$\lim_{n\to\infty} \frac{f(n)}{g(n)} > 0 \iff f(n) = \Omega(g(n))$$

This is actually a simplification - these definitions only hold when the limit is defined.[5] Trying the limit test on our example above, $\lim_{n\to\infty} \frac{f(n)}{n^3} = 5$ after applying L'Hopital's rule. Thus, $f = O(n^3)$ and $f = \Omega(n^3)$, so $f = \Theta(n^3)$.

[4]This is technically an abuse of notation, as the equal sign here is not symmetric: $O(n) = O(n^2)$ is true, but $O(n^2) = O(n)$ is certainly false. It'd be more fitting to say $n \in O(n^2)$, but the equals sign is customary.

[5]Mathematicians bypass the edge case of an undefined limit using supremum or infimum.

Below are a few general rules for big O notation. Try to justify each statement yourself (Exercise 0.1):

- Exterior multiplicative constants can be omitted, e.g. $14n^2 = O(n^2)$, $3^{n+1} = O(3^n)$.
- Lower order terms can be eliminated, e.g. $n^2 + n \ln n + 10 = O(n^2)$
- Exponentials $>$ polynomials $>$ logarithms $>$ constants, e.g. $3 = O(log(n))$, $\ln n = O(n)$, $n^2 = O(2^n)$.
- Exponentials of higher base dominate those of lower base: a^n dominates b^n if $a > b$, e.g. $2^n = O(3^n)$.
- Polynomials of higher degree dominate those of lower degree: n^a dominates n^b if $a > b$, e.g. $n = O(n^2)$
- Logarithm bases do not matter: $\log_a n = \Theta(\log_b n)$ for any a, b.

Big O notation is also used for measuring the memory usage of an algorithm (also known as "space complexity"). Space complexity generally refers to the additional amount of memory used in an algorithm, i.e excluding the size of the inputs. Usually, this is straightforward to calculate and does not require the analysis above.

It is important to remember that big O is just a model for simplification of analysis. It disregards many important factors of real-life computing speed such as the time it takes reading from disk or the effects of large constant factors. Ultimately, however, it is a very useful framework that we will use for the rest of the class to analyze efficiency.

3 Fibonacci

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

The above sequence is known as the Fibonacci numbers. The Fibonacci numbers are defined recursively:

$$\begin{aligned} F_0 &= 0 \\ F_1 &= 1 \\ F_n &= F_{n-1} + F_{n-2}. \end{aligned}$$

We would like to devise an algorithm to calculate the nth Fibonacci number. The first algorithm we may come up with, in pseudocode:

Pseudocode:

Algorithm 2 Naive Fibonacci Algorithm

```
1: procedure FIB(n)
2:     if n = 0 then return 0
3:     if n = 1 then return 1
4:     return FIB(n − 1) + FIB(n − 2)
```

This algorithm is obviously correct, as we have directly transcribed the definition. However, how efficient is it?

Let's do an inductive analysis: let $T(n)$ be the number of computer steps used to calculate F_n. Assuming arithmetic operations are constant, the function's runtime is a constant plus the amount of time of the recursive calls: $T(n) = T(n-1) + T(n-2) + O(1)$. We can observe $T(n-1) \geq T(n-2)$, so $T(n) \geq T(n-2) + T(n-2) = 2T(n-2)$. Furthermore, using the same reasoning, $T(n-2) \geq 2T(n-4)$, so $T(n) \geq 2 * 2T(n-4)$, and so on.

Thus, $T(n) >= 2^{N/2}$ follows with an inductive argument with $T(0) = 1, T(1) = 2$ as the base case. So, this algorithm is on the order of exponential time.

What can we do to improve this algorithm? We may observe that we recalculate FIB(x) for most values of x many times in our recursion (e.g FIB(3) is calculated from scratch for every $x > 3$). To remove this repetition, we can instead store the values of FIB(x) in an array, so we can access them in constant time instead of recalculating them.

Pseudocode:

Algorithm 3 Fibonacci Linear-time Algorithm

```
1: procedure FIB(n)
2:     if n = 0 then return 0
3:     lst ← [0..n − 1]
4:     lst[0] ← 0
5:     lst[1] ← 1
6:     for i in [2..n] do
7:         lst[i] ← lst[i − 1] + lst[i − 2]
8:     return lst[n − 1]
```

Now, each FIB(x) is only calculated once! Since the calculation of FIB(x) for any x is $O(1)$ and there are n values calculated up to FIB(n), the total running time of the algorithm is $O(n)$. At the expense of $O(n)$ memory, we have now reduced our algorithm to linear time![6]

[6] This is actually an example of dynamic programming, which will be covered in detail later in this course.

4 Bit Complexity and Addition

How many digits are in a non-negative[7] number N in base b? With k digits, we can express numbers up to $b^k - 1$; for example in base 10, the largest number that can be represented with $k = 3$ digits is 999. Thus solving for k, we find a number N has $\lceil \log_b N + 1 \rceil$, or $O(\log N)$ digits. In this course, unless we specify otherwise we will be using base 2, i.e binary, so we may use 'digits' and 'bits' interchangeably.

Let's revisit the elementary algorithm for how to add two numbers. Recall we are interested in measuring an algorithm's runtime with respect to the size of the inputs. For the addition problem, the size of the inputs are the number of digits in the two numbers. Hence, we are interested in the *bit complexity* of the algorithm. Suppose we have two numbers x and y that each are n bits long. Using the elementary algorithm, each individual bit summation is calculated using some fixed amount of time (say, using a truth table), and there are at most $n + 1$ digits in the result. Thus, the entire algorithm runs in $O(n)$.

So, it's true that adding larger numbers takes longer than adding smaller numbers. In the rest of the course, however, we shall treat addition and all other basic arithmetic operations as a constant time operation unless otherwise specified.[8]

5 Subroutines

When devising new algorithms, we often take an existing algorithm and use it as a subroutine. For example, in multiplication, which we'll cover in the next note, we will use addition as a subroutine. It is useful to think of the subroutine algorithm as a 'black-box', meaning we only care about the inputs and outputs, and ignore all the internal details. In fact, all recursive algorithms take advantage of subroutines, as they use a smaller version of themselves (i.e subproblems) as subroutines! Any algorithm taught in lecture may be used as a subroutine without proof if unmodified in any way. If modified, it must be proven from scratch, as even a small modification can affect an algorithm's correctness.

The process of using an algorithm without modification as a subroutine is formally called a *reduction*, which will be a focus of the second half of this course.

[7] Note that we are only concerned about non-negative integers, though there are several schemes to handle negative numbers by using an extra bit such as Two's Complement, which is covered in CS61C.

[8] Why can we make this assumption? One reason is that in many languages such as Java, ints have a fixed length representation of 32 bits. This means that all basic arithmetic operations involving ints are constant, as the length of all ints are fixed. The downside is that there are limits to the size of numbers that can be represented with ints, but this is generally not a problem since we rarely have to deal with numbers larger than 32 bits. If we do need to work with arbitrarily large numbers then we must use some other representation and arithmetic can no longer be assumed to be constant time.

6 Exercises

1. Justify each of the statements in the Big O section of this note.
2. Find a tight bound for the runtime of Selection sort. Selection sort iteratively finds the minimum element of the active list, moves it to the front, then removes it from the active candidates.
3. Find a tight bound for the summation $\sum_{i=0}^{n} 2^i$.
4. Improve the linear time fibonacci algorithm above to $O(1)$ space instead of $O(n)$.
5. Can the bit complexity of addition be improved? Justify why or why not.

7 Solutions

1. Apply the limit test for each statement. The solutions for each individual property are omitted for brevity.
2. On the first iteration, Selection sort examines n elements, then $n-1$, then $n-2$, and so on down to the last element. The runtime is $n+n-1+n-2+\cdots+1 = n(n+1)/2 = O(n^2)$.
3. By the formula for a finite geometric series, the summation is equal to $2^{n+1}-1 = O(2^n)$.
4. Notice that to calculate $\text{FIB}(n)$, we only need the results of $\text{FIB}(n-1)$ and $\text{FIB}(n-2)$. Thus, it is inefficient to store an entire array of $\text{FIB}(x)$ for all x when we only need the last two entries. We can make do with just two variables that we iteratively update.
5. No. As a lower bound, to even read the inputs or write the answer requires $O(n)$ time, where n is the number of bits in the larger input.

Feedback form: https://forms.gle/cM1io6eXyH65ytMP8

Contributors:

- Kevin Zhu
- Axel Li