Recit 1

April 24, 2024

→ $O(f(n))$ represents the set of functions with domain = $\mathbb{N}$ that satisfies the property:

"Non negative function $g(n)$ is a fn in O if and only if there exists a positive real # c and positive integer n_0 such that $g(n) \leq c \cdot f(n)$ for all $n \geq n_0$."

$f(n)$ is the one we assign as $n \log n$ ($O(n \log n)$), n^2 ($O(n^2)$) etc ⟹ this means $f(n)$ is an upper bound of $g(n)$ i.e. $O(f(n))$ are all fn's that is below $f(n)$...

Data Structure: Static Array

in an appropriate choice of data structure. For our example, we will use the most primitive data structure native to the Word-RAM: the **static array**. A static array is simply a contiguous sequence of words reserved in memory, supporting a static sequence interface:

- `StaticArray(n)`: allocate a new static array of size n initialized to 0 in $\Theta(n)$ time
- `StaticArray.get_at(i)`: return the word stored at array index i in $\Theta(1)$ time
- `StaticArray.set_at(i, x)`: write the word x to array index i in $\Theta(1)$ time

The `get_at(i)` and `set_at(i, x)` operations run in constant time because each item in the array has the same size: one machine word. To store larger objects at an array index, we can interpret the machine word at the index as a memory address to a larger piece of memory. A Python

↳ one real-life example closest to a static array is a Python tuple, although the Py tuple doesn't have a set_at(i,x).

↳ for context, a Python list is an implementation of a dynamic array.

Asymptotics Exercises

② Find a single, tight asymptotic bound for $\binom{n}{6006}$

↳ $\binom{n}{6006} = \frac{n!}{6006!\,(n-6006)!} = \frac{n\,(n-1)!}{c!\,(n-c)!} = n\,\frac{(n-1)!}{c!\,(n-c)!}$

↳ let $= c$

$= n(n-1) \cdot \frac{(n-2)!}{c!\,(n-c)!} = (n^2-n) \cdot \frac{(n-2)!}{c!\,(n-c)!}$

$= n(n-1)(n-2)\cdots(n-6005)\,/\,c!$

$\in O(n^{6006})$ ✓

* why is the sol'n using Θ instead of O? how is this a lower bound too?

(3) Do the same for $\log_{600_6}((\log(n^{\sqrt{n}}))^2)$

$= 2\log_{600_6}(\log(n^{\sqrt{n}}))$

$= 2\log_{600_6}(\sqrt{n} \cdot \log n)$

$= 2[\log_{600_6}(n^{1/2}) + \log_{600_6}(n)]$

$= \log_{600_6}(n) + 2\log_{600_6}(n)$

$= 3\log_{600_6}(n)$

$\in \Theta(\log_{600_6}(n))$

$\log_x(ab) = \log_x(a) + \log_x(b)$, $\log_a b = \frac{\log b}{\log a}$

$= \frac{2}{\log 600_6}(\log(\sqrt{n}\log n))$

$= c(\frac{1}{2}\log(n) + \log\log n)$

$\in \Theta(\log(n))$

... no more time for the rest. Just looked @ the answers..

(7) Remember: $\log(n!) \in \Theta(n\log n)$ using Sterling's approx!

Recitation 1

Algorithms

The study of algorithms searches for efficient procedures to solve problems. The goal of this class is to not only teach you how to solve problems, but to teach you to **communicate** to others that a solution to a problem is both **correct** and **efficient**.

- A **problem** is a binary relation connecting problem inputs to correct outputs.
- A (deterministic) **algorithm** is a procedure that maps inputs to single outputs.
- An algorithm **solves** a problem if for every problem input it returns a correct output.

While a problem input may have more than one correct output, an algorithm should only return one output for a given input (it is a function). As an example, consider the problem of finding another student in your recitation who shares the same birthday.

> **Problem:** Given the students in your recitation, return either the names of two students who share the same birthday and year, or state that no such pair exists.

This problem relates one input (your recitation) to one or more outputs comprising birthday-matching pairs of students or one negative result. A problem input is sometimes called an **instance** of the problem. One algorithm that solves this problem is the following.

> **Algorithm:** Maintain an initially empty record of student names and birthdays. Go around the room and ask each student their name and birthday. After interviewing each student, check to see whether their birthday already exists in the record. If yes, return the names of the two students found. Otherwise, add their name and birthday to the record. If after interviewing all students no satisfying pair is found, return that no matching pair exists.

Of course, our algorithm solves a much more general problem than the one proposed above. The same algorithm can search for a birthday-matching pair in **any** set of students, not just the students in your recitation. In this class, we try to solve problems which generalize to inputs that may be arbitrarily large. The birthday matching algorithm can be applied to a recitation of any size. But how can we determine whether the algorithm is correct and efficient?

Correctness

Any computer program you write will have finite size, while an input it acts on may be arbitrarily large. Thus every algorithm we discuss in this class will need to repeat commands in the algorithm via loops or recursion, and we will be able to prove correctness of the algorithm via **induction**. Let's prove that the birthday algorithm is correct.

> *Proof.* Induct on the first k students interviewed. Base case: for $k = 0$, there is no matching pair, and the algorithm returns that there is no matching pair. Alternatively, assume for induction that the algorithm returns correctly for the first k students. If the first k students contain a matching pair, than so does the first $k + 1$ students and the algorithm already returned a matching pair. Otherwise the first k students do not contain a matching pair, so if the $k+1$ students contain a match, the match includes student $k+1$, and the algorithm checks whether the student $k + 1$ has the same birthday as someone already processed. □

Efficiency

What makes a computer program efficient? One program is said to be more **efficient** than another if it can solve the same problem input using fewer resources. We expect that a larger input might take more time to solve than another input having smaller size. In addition, the resources used by a program, e.g. storage space or running time, will depend on both the algorithm used and the machine on which the algorithm is implemented. We expect that an algorithm implemented on a fast machine will run faster than the same algorithm on a slower machine, even for the same input. We would like to be able to compare algorithms, without having to worry about how fast our machine is. So in this class, we compare algorithms based on their **asymptotic performance** relative to problem input size, in order to ignore constant factor differences in hardware performance.

Asymptotic Notation

We can use **asymptotic notation** to ignore constants that do not change with the size of the problem input. $O(f(n))$ represents the set of functions with domain over the natural numbers satisfying the following property.

> **O Notation:** Non-negative function $g(n)$ is in $O(f(n))$ if and only if there exists a positive real number c and positive integer n_0 such that $g(n) \leq c \cdot f(n)$ for all $n \geq n_0$.

This definition upper bounds the **asymptotic growth** of a function for sufficiently large n, i.e., the bound on growth is true even if we were to scale or shift our function by a constant amount. By convention, it is more common for people to say that a function $g(n)$ **is** $O(f(n))$ or **equal to** $O(f(n))$, but what they really mean is set containment, i.e., $g(n) \in O(f(n))$. So since our problem's input size is cn for some constant c, we can forget about c and say the input size is $O(n)$ (**order** n). A similar notation can be used for lower bounds.

> **Ω Notation:** Non-negative function $g(n)$ is in $\Omega(f(n))$ if and only if there exists a positive real number c and positive integer n_0 such that $c \cdot f(n) \leq g(n)$ for all $n \geq n_0$.

When one function both asymptotically upper bounds **and** asymptotically lower bounds another function, we use Θ notation. When $g(n) = \Theta(f(n))$, we say that $f(n)$ represents a **tight bound** on $g(n)$.

> **Θ Notation:** Non-negative $g(n)$ is in $\Theta(f(n))$ if and only if $g(n) \in O(f(n)) \cap \Omega(f(n))$.

We often use shorthand to characterize the asymptotic growth (i.e., **asymptotic complexity**) of common functions, such as those shown in the table below[1]. Here we assume $c \in \Theta(1)$.

Shorthand	Constant	Logarithmic	Linear	Quadratic	Polynomial	Exponential[1]
$\Theta(f(n))$	$\Theta(1)$	$\Theta(\log n)$	$\Theta(n)$	$\Theta(n^2)$	$\Theta(n^c)$	$2^{\Theta(n^c)}$

Linear time is often necessary to solve problems where the entire input must be read in order to solve the problem. However, if the input is already accessible in memory, many problems can be solved in sub-linear time. For example, the problem of finding a value in a sorted array (that has already been loaded into memory) can be solved in logarithmic time via binary search. We focus on polynomial time algorithms in this class, typically for small values of c. There's a big difference between logarithmic, linear, and exponential. If $n = 1000$, $\log n \approx 10^1, n \approx 10^3$, while $2^n \approx 10^{300}$. For comparison, the number of atoms in the universe is estimated around 10^{80}. It is common to use the variable 'n' to represent a parameter that is linear in the problem input size, though this is not always the case. For example, when talking about graph algorithms later in the term, a problem input will be a graph parameterized by vertex set V and edge set E, so a natural input size will be $\Theta(|V| + |E|)$. Alternatively, when talking about matrix algorithms, it is common to let n be the width of a square matrix, where a problem input will have size $\Theta(n^2)$, specifying each element of the $n \times n$ matrix.

[1]Note that exponential $2^{\Theta(n^c)}$ is a convenient abuse of notation meaning $\{2^p \mid p \in \Theta(n^c)\}$.

Model of Computation

In order to precisely calculate the resources used by an algorithm, we need to model how long a computer takes to perform basic operations. Specifying such a set of operations provides a **model of computation** upon which we can base our analysis. In this class, we will use the w-bit **Word-RAM** model of computation, which models a computer as a random access array of machine words called **memory**, together with a **processor** that can perform operations on the memory. A **machine word** is a sequence of w bits representing an integer from the set $\{0, \ldots, 2^w - 1\}$. A Word-RAM processor can perform basic binary operations on two machine words in constant time, including addition, subtraction, multiplication, integer division, modulo, bitwise operations, and binary comparisons. In addition, given a word a, the processor can read or write the word in memory located at address a in constant time. If a machine word contains only w bits, the processor will only be able to read and write from at most 2^w addresses in memory[2]. So when solving a problem on an input stored in n machine words, we will always assume our Word-RAM has a word size of at least $w > \log_2 n$ bits, or else the machine would not be able to access all of the input in memory. To put this limitation in perspective, a Word-RAM model of a byte-addressable 64-bit machine allows inputs up to $\sim 10^{10}$ GB in size.

Data Structure

The running time of our birthday matching algorithm depends on how we store the record of names and birthdays. A **data structure** is a way to store a non-constant amount of data, supporting a set of operations to interact with that data. The set of operations supported by a data structure is called an **interface**. Many data structures might support the same interface, but could provide different performance for each operation. Many problems can be solved trivially by storing data in an appropriate choice of data structure. For our example, we will use the most primitive data structure native to the Word-RAM: the **static array**. A static array is simply a contiguous sequence of words reserved in memory, supporting a static sequence interface:

- `StaticArray(n)`: allocate a new static array of size n initialized to 0 in $\Theta(n)$ time
- `StaticArray.get_at(i)`: return the word stored at array index i in $\Theta(1)$ time
- `StaticArray.set_at(i, x)`: write the word x to array index i in $\Theta(1)$ time

The `get_at(i)` and `set_at(i, x)` operations run in constant time because each item in the array has the same size: one machine word. To store larger objects at an array index, we can interpret the machine word at the index as a memory address to a larger piece of memory. A Python `tuple` is like a static array without `set_at(i, x)`. A Python `list` implements a **dynamic array** (see L02).

[2]For example, on a typical 32-bit machine, each byte (8-bits) is addressable (for historical reasons), so the size of the machine's random-access memory (RAM) is limited to (8-bits)$\times(2^{32}) \approx 4$ GB.

```
1  class StaticArray:
2      def __init__(self, n):
3          self.data = [None] * n
4      def get_at(self, i):
5          if not (0 <= i < len(self.data)): raise IndexError
6          return self.data[i]
7      def set_at(self, i, x):
8          if not (0 <= i < len(self.data)): raise IndexError
9          self.data[i] = x
10
11 def birthday_match(students):
12     '''
13     Find a pair of students with the same birthday
14     Input:  tuple of student (name, bday) tuples
15     Output: tuple of student names or None
16     '''
17     n = len(students)                              # O(1)
18     record = StaticArray(n)                        # O(n)
19     for k in range(n):                             # n
20         (name1, bday1) = students[k]               # O(1)
21         for i in range(k):                         # k     Check if in record
22             (name2, bday2) = record.get_at(i)      # O(1)
23             if bday1 == bday2:                     # O(1)
24                 return (name1, name2)              # O(1)
25         record.set_at(k, (name1, bday1))           # O(1)
26     return None                                    # O(1)
```

Running Time Analysis

Now let's analyze the running time of our birthday matching algorithm on a recitation containing n students. We will assume that each name and birthday fits into a constant number of machine words so that a single student's information can be collected and manipulated in constant time[3]. We step through the algorithm line by line. All the lines take constant time except for lines 8, 9, and 11. Line 8 takes $\Theta(n)$ time to initialize the static array record; line 9 loops at most n times; and line 11 loops through the k items existing in the record. Thus the running time for this algorithm is at most:

$$O(n) + \sum_{k=0}^{n-1}(O(1) + k \cdot O(1)) = O(n^2)$$

This is quadratic in n, which is polynomial! Is this efficient? No! We can do better by using a different data structure for our record. We will spend the first half of this class studying elementary data structures, where each data structure will be tailored to support a different set of operations efficiently.

[3]This is a reasonable restriction, which allows names and birthdays to contain $O(w)$ characters from a constant sized alphabet. Since $w > \log_2 n$, this restriction still allows each student's information to be distinct.

Asymptotics Exercises

1. Have students generate 10 functions and order them based on asymptotic growth.

2. Find a simple, tight asymptotic bound for $\binom{n}{6006}$.

   Solution: Definition yields $n(n-1)\dots(n-6005)$ in the numerator (a degree 6006 polynomial) and 6006! in the denominator (constant with respect to n). So $\binom{n}{6006} = \Theta(n^{6006})$.

3. Find a simple, tight asymptotic bound for $\log_{6006}\left(\left(\log\left(n^{\sqrt{n}}\right)\right)^2\right)$.

   Solution: Recall exponent and logarithm rules: $\log ab = \log a + \log b$, $\log\left(a^b\right) = b\log a$, and $\log_a b = \log b / \log a$.

   $$\begin{aligned}\log_{6006}\left(\left(\log\left(n^{\sqrt{n}}\right)\right)^2\right) &= \frac{2}{\log 6006}\log\left(\sqrt{n}\log n\right)\\ &= \Theta(\log n^{1/2} + \log\log n) = \Theta(\log n)\end{aligned}$$

4. Show that $2^{n+1} \in \Theta(2^n)$, but that $2^{2^{n+1}} \notin O(2^{2^n})$.

   Solution: In the first case, $2^{n+1} = 2\cdot 2^n$, which is a constant factor larger than 2^n. In the second case, $2^{2^{n+1}} = \left(2^{2^n}\right)^2$, which is definitely more than a constant factor larger than 2^{2^n}.

5. Show that $(\log n)^a = O(n^b)$ for all positive constants a and b.

   Solution: It's enough to show $n^b/(\log n)^a$ limits to ∞ as $n \to \infty$, and this is equivalent to arguing that the **log** of this expression approaches ∞:

   $$\lim_{n\to\infty}\log\left(\frac{n^b}{(\log n)^a}\right) = \lim_{n\to\infty}(b\log n - a\log\log n) = \lim_{x\to\infty}(bx - a\log x) = \infty,$$

   as desired.

   Note: for the same reasons, $n^a = O(c^n)$ for any $c > 1$.

6. Show that $(\log n)^{\log n} = \Omega(n)$.

   Solution: Note that $m^m = \Omega(2^m)$, so setting $n = 2^m$ completes the proof.

7. Show that $(6n)! \notin \Theta(n!)$, but that $\log((6n)!) \in \Theta(\log(n!))$.

   Solution: We invoke Sterling's approximation,

   $$n! = \sqrt{2\pi n}\left(\frac{n}{e}\right)^n\left(1 + \Theta\left(\frac{1}{n}\right)\right).$$

   Substituting in $6n$ gives an expression that is at least 6^{6n} larger than the original. But taking the logarithm of Sterling's gives $\log(n!) = \Theta(n\log n)$, and substituting in $6n$ yields only constant additional factors.