Recitation 1

Algorithms

The study of algorithms searches for efficient procedures to solve problems.

- A **problem** is a binary relation connecting problem inputs to correct outputs.
- An **algorithm** is a procedure that maps inputs to single outputs.
- An algorithm **solves** a problem if it returns a correct output given any problem input.

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 two students in your recitation who share 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:** Go around the room and ask each student their birthday. While doing so, maintain a record of birthdays seen so far. If ever the same birthday appears twice in the record, return the names of the students associated with the match; otherwise, 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. When studying algorithms, we try to solve problems having a large (often infinite) number of possible inputs, possibly with different sizes. We expect that a larger input might take more time to solve than another input having smaller size. In order to say precisely whether an algorithm is **efficient**, we would like to compare the amount of resources used by the algorithm, relative to the size of the input being solved.

The **size of an input** is how much space would be needed to store the input on a computer. For the generalized birthday problem, an input consists of the name and birthday of every student in the set. If there are n students and the amount of space needed to store the ith student's name and birthday is s_i, then the size of the input is simply the sum $\sum_{i=1}^{n} s_i$. We will often make reasonable assumptions to simplify our model of the problem. For example, we might assume a maximum length to any student's name (say 20 characters) and a maximum number of possible birthdays (say 366×2018), so that the space needed to store any student's information is no more than some fixed constant c. Then the size of any input set containing n students is **upper bounded** by (i.e. is no greater than) cn.

Asymptotic Notation

Instead of saying that the generalized birthday problem's input size is upper bounded by some constant times the number of input students, we can use **asymptotic notation** to ignore constants that do not change as the size of the problem grows. $O(f(n))$ 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. Here we assume $c \in \Theta(1)$.

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

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 $\Theta(n^2)$, containing each element of the $n \times n$ matrix.

Efficiency

What makes a computer program efficient? One program is said to be more **efficient** than another if it can solve the same problem using fewer resources. The resources used by a program, e.g. storage space or running time, depend on both an algorithm 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. One benefit of comparing algorithms based on their asymptotic running times is that we can ignore minor differences in hardware performance.

Let's analyze the running time of our birthday problem algorithm. Assume that speaking to each student and getting their information takes constant $O(1)$ time. To record the information, we first check whether the birthday already exists in our record in time T_c. If it does not exist, we add the student to the record in time T_r; otherwise, we return a matching pair in time T_m. Then the time to interview one student is at most $O(1) + T_c + T_r + T_m$. In the worst case, we have to interview every student, so we must do this n times.

How long do each of these operations take? It depends on how the record is stored, i.e. the **data structure** used. A data structure is a way of storing data that makes certain operations on the data efficient. Many problems can be solved trivially by storing data in an appropriate choice of data structure. For our example, we could use an array of length n to store our record. The array supports random access to read or write from any bucket of the array in $O(1)$ time.

Assign each student a number from 0 to $n-1$ based on the order in which they are interviewed. We will store student i's information in array bucket i. If k is the number of students interviewed so far, we can check to see if a birthday already exists in the array, simply by looping through the first k array buckets and checking each entry. Then $T_c = O(k)$ because we must check all k buckets in the worst case. If the birthday does not exist, writing student k's information to bucket k takes $T_r = O(1)$ time; and if it does, then returning the pair of student names also takes $T_m = O(1)$ time. Thus, using an array to store the record in our algorithm results in an overall quadratic running time: $O(\sum_{k=1}^{n} k + 1) = O(n^2)$. This is a polynomial time algorithm, though more efficient algorithms exist. We will be able to solve this problem asymptotically faster by using other data structure discussed later in this class.

```python
def birthday_pair(students):
    '''
    Find a pair of students having the same birthday.
    Input:  array of student objects, each with name and birthday
    Output: tuple of student names or None
    '''
    n = len(students)                # O(1)
    record = [None] * n              # O(n) Preallocate record
    for s in students:               # O(n) Loop through students
        for i in range(k):           # O(k) Loop through current record
            if record[i].birthday == s.birthday:  # O(1)
                return (record[i].name, s.name)   # O(1)
        record[k] = s                             # O(1)
    return None                      # O(1)
```

Model of Computation

In order to precisely calculate the resources used by an algorithm, we need a model for 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 our birthday example above, we made an assumption that the length of a student's name is at most a constant number of characters so that we could read and write names in constant time. However, if we have n students and want to allow each student to have a different name, we would need to store each name using at least $\log n$ bits, or else there would be more students than possible names. If we want to allow representations of names or numbers containing a logarithmic number of bits, does that mean it will take $\Omega(\log n)$ time to read a name or number that is $\Omega(\log n)$ bits long?

In actuality, modern computers perform computation on **words**, not individual bits. A word is a fixed-length sequence of w bits: words on a 32-bit machine are 32-bits long, and 64-bits long on a 64-bit machine. The **Word-RAM** model of computation models a computer's memory as an array of words, with each word accessible in constant time given a **memory address**, the word's index (i.e. **pointer**) into the array. The Word-RAM also has a number of w-bit length word registers upon which it can perform simple operations in constant time: read, write, add, compare, etc.

In order to perform operations requiring an address (like a read or copy), registers need to be able to contain a memory address within a word. If a word is w bits long, the memory address space will be limited to 2^w values, bounding the possible size of your computer's accessible memory[1]. Thus, when solving a problem on an input with size n, the Word-RAM model assumes word size w is at least $\Omega(\log n)$ bits, or else you 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 storable in up to $\sim 10^{10}$ GB.

So in a Word-RAM model, we assume that access (read, write, copy, etc.) and bitwise (add, subtract, shift, mod, compare, etc.) operations can be performed on numbers stored using a constant number of words in constant time, a reasonable approximation of existing CPU architectures. So, if student names are at most $O(\log n)$ characters, each name can fit into a constant number of words and can be operated on in constant time. Most of the time, we will ignore this subtlety and simply assume that numbers or identifying strings appearing in a problem input fit inside a constant number of words, but how we store numbers will be relevant later in the term, when we talk about linear sorting and hashing.

[1]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) will be limited to (8-bits)$\times(2^{32}) \approx 4$ GB.

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 $2^{n+1} = \Theta(2^n)$, but that $2^{2^{n+1}} = \Omega(2^{2^n})$.

   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 $(6n)! = \Omega(n!)$, but that $\log((6n)!) = \Theta(\log(n!))$.

   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.

Peak Finding (Local Search)

Let's use Python to find a **peak** in a 1D array of n integers. By peak, we mean a (weak) **local** maximum, i.e., an element that is at least as large as its neighbors in the array. There's an obvious linear time, brute force algorithm: scan through each element, and check whether it is a peak.

```python
def peak_find_brute(A):
    '''Find index of a peak from an input list of integers.'''
    n = len(A)                                      # O(1)
    for i in range(n):                              # O(n) Loop
        if ((i == 0     or A[i] >= A[i - 1]) and   # O(1) Check left
            (i == n - 1 or A[i] >= A[i + 1])):     # O(1) Check right
            return i                                # O(1) Peak found!
    return None                                     # O(1) No peak found
```

By contrast, a (weak) **global** maximum is an element that is at least as large as **every** element of the array, so a global maximum is also a peak. A global maximum always exists, so the last line of the code will never be reached (assuming the array is nonempty). A global maximum requires $\Omega(n)$ time to find; if it didn't, at least one array index cannot have been observed by the algorithm, and an adversary could place the largest integer there. By contrast, we will be able to find a local peak in sub-linear time.

We will use a greedy approach analogous to binary search: repeatedly reduce the peak search range by a constant factor. Since we know the array contains a peak, if we divide the array in half, a peak must exist in at least one of the halves (possibly both). If we can quickly identify one half that contains a peak, we can continue the search in that half recursively. If we can guarantee that the search space contains a peak each time we reduce it, when we finally reduce the search space to a single element of the array, that element must be a peak. This type of argument is called an **inductive argument**. The guarantee is called an **invariant**, which we must prove holds throughout the computation. The invariant we would like to maintain is that each sub-array in our search contains a peak.

How can we quickly identify a half of the array that contains a peak? Consider the sub-array extending from index i to index j. If $A[i]$ is at least as large as its left neighbor ($A[i-1] \leq A[i]$) and $A[j]$ is at least as large as its right neighbor ($A[j] \geq A[j+1]$), then a global maximum of that sub-array must also be a peak.[2] Therefore, we can quickly guarantee the existence of a peak by looking only at the boundary of a range, without looking at all of its elements. This property suggests a stronger invariant than peak containment: if the elements stored in each endpoint of the search range are at least as large as their outer neighbors, then the range contains a peak. To maintain the invariant, we simply look at the middle two elements, and recursively search on the side containing the larger of the two.

[2]Proving this might be a nice way to brush up on your proof writing from 6.042. Consider the cases where the max lies on the boundary or not.

```
def peak_find(A, i = 0, j = None):
    '''
    Find index of a peak from range (i, j) of an input list of integers.
    Assumes that the range specified by (i, j) contains a peak.
    '''
    if j is None:                   # O(1)
        j = len(A)                  # O(1) Default input
    if i + 1 == j:                  # O(1)
        return i                    # O(1) Base case, one element range
    c = (i + j) // 2                # O(1) Compute center
    if A[c - 1] > A[c]:             # O(1)
        return peak_find(A, i, c)   # O(?) Left half has peak
    else:                           # O(1)
        return peak_find(A, c, j)   # O(?) Right half has peak
```

What is the running time of this recursive algorithm? We can compute it by evaluating a recurrence relation, where $T(k)$ represents the running time of the function on an input (i.e., a remaining search range) of size k. This function does a constant amount of work, then makes at most one recursive call on a problem of roughly half the size. A recurrence relation representing our algorithm is $T(k) = T(k/2) + O(1)$, where solving the base case takes constant time $T(1) = O(1)$. One way to solve a recurrence is to guess a solution, and substitute into the recurrence to show that the solution is correct. Guessing $T(n) = O(\log n)$ shows that this algorithm runs in logarithmic time relative to the input. We will see other methods of solving recurrences next week.