Algorithms and Programs

- This is a lecture on algorithms, not on programming.
- An algorithm is an abstract concept to solve a given problem. In contrast, a program is a concrete implementation of an algorithm.
- In order to implement an algorithm as a program we have to cover every detail, be it trivial or not. On the other hand, to specify an algorithm it is often sufficient to describe just the interesting aspects.

Example: An algorithm to solve birthday matching

- Maintain a record of names and birthdays (initially empty)
- Interview each student in some order
  - If birthday exists in record, return found pair!
  - Else add name and birthday to record
- Return None if last student interviewed without success

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.

Algorithms and Programs

- This is a lecture on algorithms, not on programming.
- An algorithm is an abstract concept to solve a given problem. In contrast, a program is a concrete implementation of an algorithm.
- In order to implement an algorithm as a program we have to cover every detail, be it trivial or not. On the other hand, to specify an algorithm it is often sufficient to describe just the interesting aspects.

Example: An algorithm to solve birthday matching

- Maintain a record of names and birthdays (initially empty)
- Interview each student in some order
  - If birthday exists in record, return found pair!
  - Else add name and birthday to record
- Return None if last student interviewed without success

Problem: Given the students in your class, 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 class) to one or more outputs comprising birthday-matching pairs of students or one negative result. 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.

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 class of any size. But how can we determine whether the algorithm is correct and efficient?

Desirable Properties of Algorithms

Before we start with our discussion of algorithms we should think about our goals when designing algorithms.

- Algorithms have to be correct.
- Algorithms should be efficient with respect to both computing time, memory, and, last not least, energy consumption.
- Algorithms should be simple.

The first goal in this list is so self-evident that it is often overlooked. The importance of the last goal might not be as obvious as the other goals. However, the reason for the last goal is economical: If it takes very long to code an algorithm, the cost of the implementation might well be unaffordable. Furthermore, even if the time budget to implement an algorithm is next to unlimited, there is another reasons to strife for simple algorithms: If the conceptual complexity of an algorithm is too high, maintenance might become a nightmare and it might be impossible to guarantee the correctness of the implementation. Therefore, the third goal is strongly related to the first goal.

Correctness and Induction

- Programs/algorithms have fixed size, so how to prove correct?
- For small inputs, can use case analysis
- For arbitrarily large inputs, algorithm must be recursive or loop in some way
- Must use induction (why recursion is such a key concept in computer science)
- Example: Proof of correctness of birthday matching algorithm
  - Induct on k: the number of students in record
  - **Hypothesis**: if first k contain match, returns match before interviewing student $k+1$
  - **Base case**: $k = 0$, first k contains no match
  - Assume for induction hypothesis holds for $k = k'$, and consider $k = k' + 1$
  - If first k' contains a match, already returned a match by induction
  - Else first k' do not have match, so if first $k' + 1$ has match, match contains $k' + 1$
  - Then algorithm checks directly whether birthday of student $k' + 1$ exists in first k'

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.

An Approach to Measure Complexity

Sometimes it is necessary to have a precise understanding of the complexity of an algorithm. In order to obtain this understanding we could proceed as follows:

1. We implement the algorithm in assembly language.
2. We count how many additions, multiplications, assignments, etc. are needed for an input of a given size. Additionally, we have to count all storage accesses.
3. We look up the amount of time that is needed for the different operations in the processor handbook.
4. Using the information discovered in the previous two steps we predict the running time of our algorithm for given input.

This approach is problematic for a number of reasons.

a. It is very complicated and therefore far too time-consuming.
b. The execution time of the basic operations is highly dependent on the memory hierarchy of the computer system: For many modern computer architectures, adding two numbers that happen to be stored in registers is much faster than adding two numbers that reside in main memory. Unless we peek into the machine code generated by our compiler, it is very difficult to predict whether a variable will be stored in memory or in a register. Even if a variable is stored in main memory, we still might get lucky if the variable is also stored in a cache.
c. If we would later code the algorithm in a different programming language or if we would port the program to a computer with a different processor we would have to redo most of the computation.

This final reason shows that the approach sketched above is not well suited to measure the complexity of an algorithm: After all, the notion of an algorithm is more abstract than the notion of a program and we really need a notion measuring the complexity of an algorithm that is more abstract than the notion of the running time of a program.

An Analogy

Imagine the following scenario: You've got a file on a hard drive and you need to send it to your friend who lives across the country. You need to get the file to your friend as fast as possible. How should you send it?

- Most people's first thought would be email, FTP, or some other means of electronic transfer. That thought is reasonable, but only half correct. If it's a small file, you're certainly right, it would take 5-10 hours to get to an airport, hop on a flight, and then deliver it to your friend.
- But what if the file were really, really large? Is it possible that it's faster to physically deliver it via plane? Yes, actually it is, A one-terabyte file could take more than a day to transfer electronically. It would be much faster to just fly it across the country. If your file is that urgent (and cost isn't an issue), you might just want to do that.
- What if there were no flights, and instead you had to drive across the country? Even then, for a really huge file, it would be faster to drive.

This is what the concept of asymptotic runtime, or big 0 ($\mathcal{O}$) time, means. We could describe the data transfer "algorithm" runtime as:

- Electronic Transfer: $\mathcal{O}(s)$, where s is the size of the file. This means that the time to transfer the file increases linearly with the size of the file. (Yes, this is a bit of a simplification, but that's okay for these purposes.)
- Airplane Transfer: $\mathcal{O}(1)$ with respect to the size of the file. As the size of the file increases, it won't take any longer to get the file to your friend. The time is constant. No matter how big the constant is and how slow the linear increase is, linear will at some point surpass constant.

Asymptotic Notation

- $\mathcal{O}$ Notation characterizes an upper bound on the asymptotic behavior of a function.
  Consider, for example, the function $7n^3 + 100n^2 - 20n + 6$. Its highest-order term is $7n^3$, and so we say that this function's rate of growth is n^3. Because this function grows no faster than n^3, we can write that it is $\mathcal{O}(n^3)$
- Time estimate below based on one operation per cycle on a 1 GHz single-core machine
- Particles in universe estimated $< 10^{100}$

Input	Constant	Logarithmic	Linear	Log-linear	Quadratic	Polynomial	Exponential
n	$\mathcal{O}(1)$	$\mathcal{O}(logn)$	$\mathcal{O}(n)$	$\mathcal{O}(nlogn)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n^c)$	$2^{\mathcal{O}(n^c)}$
1000	1	≈ 10	1000	$\approx 10,000$	1,000,000	1000^c	$2^{1000} \approx 10^{301}$
Time	1 ns	10 ns	1 μs	10 μs	1ms	10^{3c-9}s	10^{281} millenia

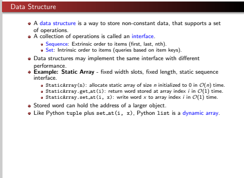
Table: Asymptotic Time Table

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 ma- chine 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.

Model of Computation

- Specification for what operations on the machine can be performed in $\mathcal{O}(1)$ time
- Model in this class is called the Word-RAM
- **Machine word**: block of w bits (w is word size of a w-bit Word-RAM)
- **Memory**: Addressable sequence of machine words
- **Processor** supports many **constant time** operations on a $\mathcal{O}(1)$ number of words (**integers**):
  - **integer** arithmetic: ($+$, $-$, $*$, //, %)
  - **logical** operators: (&&, ||, !, ==, >, <, <=, >=)
  - **bitwise** arithmetic: (&, |, >, <<)
  - Given a word a, can **read** word at address a, **write** word to address a
- Memory address must be able to access every place in memory
  - Requirement: $w \geq$ # bits to represent largest memory address, i.e., $log_2 n$
  - 32-bit words $\rightarrow$ max $\approx$ 4 GB memory,
  - 64-bit words $\rightarrow$ max $\approx$ 16 exabytes of memory.

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. 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 $\approx 10^{10}$ GB in size.



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 static array of size n initialized to 0 in $\mathcal{O}(n)$ time.
- `StaticArray.get_at(i)`: return word stored at array index i in $\mathcal{O}(1)$ time.
- `StaticArray.set_at(i, x)`: write word x to array index i in $\mathcal{O}(1)$ time.

The `get_at(i)` and `set_at(i)` 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)`. A Python `list` implements a dynamic array.

```python
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)
```

- Two loops: outer $k \in \{0, \ldots, n-1\}$, inner is $i \in \{, \ldots, k\}$
- Running time is $\theta(n) + \sum_{k=0}^{n-1}(\theta(1) + k * \theta(1)) = \theta(n^2)$

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 time3. We step through the algorithm line by line. All the lines take constant time except for lines 18, 29, and 21. Line 18 takes (n) time to initialize the static array record; line 19 loops at most n times; and line 21 loops through the k items existing in the record. Thus the running time for this algorithm is at most:

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

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.