Algorithms - Day 3

Instructor: Pat Devlin — prd41@math.rutgers.edu

Summer, 2016

Running times

When we want to analyze an algorithm, we need to do two things.

(1) Decide how we should measure the "size" of the problem.

(2) Decide how we should measure the "cost" of running the algorithm.

We may have very different notions of "size" and "cost" depending on our problem. These are called *models of computation.* For example

- Search problem

  Size:

  Cost:

- Guessing game ("twenty questions")

  Size:

  Cost:

- Looking for the poisoned bottles ("group testing") [homework 1, problem 4]

  Size:

  Cost:

- Dropping an egg from a building [homework 1, problem 5]

  Size:

  Cost:

- Finding which object is 'best' [homework 1, problem 6]

  Size:

  Cost:

- Non-adaptive search [homework 1, problem 7]

  Size:

  Cost:

Because each problem has its own model of computation, it's as if each problem has its own set of 'rules' and the 'cost' of algorithms solving different problems cannot be meaningfully compared. From a practical point of view, when we want to analyze an algorithm, what we *really* want to do is understand how *long* it will take to run. This is called the ***running time*** of an algorithm. The running time would be a way to apply the same notion of 'cost' to *any algorithm*.

However, there are a few *serious* issues between us and defining a meaningful notion of "running time" that makes sense for any algorithm.

Question 1 What are some possible issues or challenges that might make it difficult to rigorously define the "running time" of an algorithm? (Note: we would want this definition to apply in the same way to any algorithm whatsoever, and we don't want our definition to depend on the limitations or structure of our current technology.)

Alan Turing (1912–1954) was an English mathematician who first formalized the concept of an algorithm. He introduced the idea of a *Turing machine*, which is an abstract mathematical model for a computer. These machines are capable of performing any algorithm, and they allow us to meaningfully compare algorithms that solve different problems.

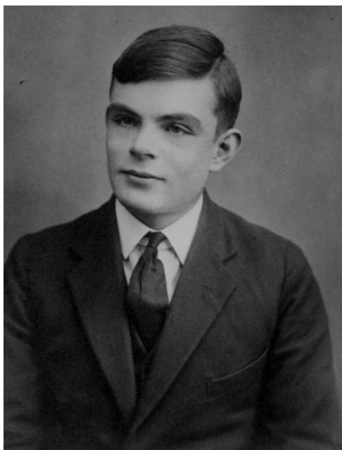

Figure 1: Alan Turing when he was sixteen

Some Alan Turing facts:

- Turing machines provided the groundwork for how mathematicians rigorously understand algorithms
  - provided a *universal* model of computation by which *any* algorithms can be compared
  - still used today as the main tool in computer science
- the father of computer science
- "Without him we would have lost the war." – Cpt. J. Roberts
- convicted criminal
- strange death at age 41

Unfortunately, Turing machines are difficult to understand [it's like understanding the inside of your phone], but they still enable us talk about 'running time' in a meaningful way.

- For us, the *running time* of an algorithm is how many 'steps' it takes to complete.
- We always use worst-case scenario, and we always use big-O or theta notation.

Question 2 Why do we only care about the *asymptotic growth* of our running times (i.e., the information conveyed in theta notation)?

Question 3 You have two algorithms to solve the same problem, which have running times $T_1(n) = \Theta(n^3)$ and $T_2 = \Theta(2^n)$. When you run them for $n = 10$, they both finish in one minute. About how long would each algorithm take to run when $n = 20$? What's the largest n each algorithm could finish if you allow them to run for an entire year?

As a first silly example, consider the following algorithm, which generalizes a well-known carol.

```
Sing_Song(n):
begin
  for i from 1 to n do
     sing "On day i my true love gave to me:"
     sing something about gift[i], then gift[i − 1], on so on until gift[1]
  od
end
```

Question 4 How many steps are in this algorithm? What's its running time? (Use theta notation)

Running time is important because when we write a computer program, we can't make the computer run fundamentally faster; all we can do is ask it to do fewer things.

One of the most important problems in computer science is the sorting problem. One reason this is important is the following.

Question 5 If we are looking for an item in a *sorted* list of length n, we can use `Binary_Search`. What is the asymptotic running time for this? Now suppose we are looking an item in an *unsorted* list. How long would that take?

Example 6 (The sorting problem) We are given an unsorted list of length n, and we need to sort it. We can only gain information by comparing two elements to see which is bigger.

Proposition 7 *Any algorithm solving the sorting problem requires* <u>*at least*</u> $\Theta(n \lg(n))$ *comparisons.*

Proof:

Here's one sorting algorithm.

```
Insertion_Sort:
begin
   Split the elements into two piles: a 'sorted pile' and an 'unsorted pile'
   At the beginning, every element starts in the 'unsorted pile' and the 'sorted pile' is empty
   Pick up any element you want from the 'unsorted pile', and move it into the 'sorted pile'
                                                        //At this point, the thing is now set up.
   while the 'unsorted pile' still has stuff in it do
         Grab something from the 'unsorted pile'
         Figure out where it should go in the 'sorted pile' and put it there
   od
   return the 'sorted pile'
end
```

Question 8 Does this algorithm work? What's its running time? Can you think of how to do better?

Example 9 Better insertion sort.

Example 10 Merge sort.

Example 11 Quick sort.