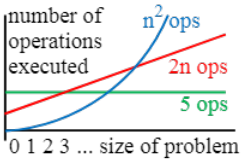
Introduction to algorithmic complexity

Introduction

We will define a notion of "runtime complexity", which will allow us to compare the speed and space requirements of different algorithms. For example, we will want to answer questions like these: Which of two sorting algorithms runs the fastest on large arrays? Which uses more space? How can we tell that a particular sorting algorithm will be infeasible (taking far too much time), without actually running it, when attempting to sort arrays of size 10,000?

In analyzing different algorithms for speed and space, we should not have to depend on the computer on which it is running, the operating system being used, and so on. Our analysis of the algorithms will be independent of such considerations.

Also, we are interested in the speed and space requirements when the data gets *large*. You can use almost *any* algorithm to sort an array of size 10 or 50 —and you won't notice the difference. But when sorting an array of size 1,000 or 10,000 or 100,000, you will see *huge* differences in the time the sorting algorithms take. Thus, we will talk about *asymptotic complexity* —the speed and space requirements as the size of the data gets large, even approaches infinity.



We will also be interested in *classes* of algorithms, depending on their time complexity. Take a look at the graph to the right. As `n` gets large, an algorithm that takes time proportional to n^2 is much much slower than one that takes time proportional to `n`. And, for large `n`, the difference between `n` and `2n` is inconsequential compared to the difference between n^2 and `n`, so we don't want to differentiate between `n` operations and `2n` operations. Thus, we will introduce notation like this: O(`1`) is the set of constant functions `f(n)`, like `f(n) = 2` and `f(n) = 5`. O(`n`) is the set of functions that are *linear* in `n`, or proportional to `n`, like `f(n) = 2n` and `f(n) = 500 + n`. O(n^2) is the set of functions that are *quadratic* in `n`, like `f(n)` = n^2 and `f(n) = n(n–1)/2`.

Finally, we are most interested in:

1. The *worst-case* time of an algorithm —the longest time it takes, depending on its input.
2. The *expected* or average time of an algorithm —the average time over all possible inputs of a certain size.

One also hears talk of the *best-case* time, but it's of little importance compared to the other two.

You first step in the study of complexity will be to learn what a "basic step" is and to get some practice in counting the basic steps in execution of an algorithm. One often has difficulty with this at the beginning, so we provide a number of examples and also give you some exercises to do yourself.

Introduction to algorithmic complexity