Lecture 09 Notes

Derek Harter

2015-03-31

1 First Session (11 - 11:40)

1.1 Introduction to Analysis of Algorithms

Analysis of algorithms is a branch of computer science that studies the performance of algorithms, especially their run time and space requirements. Most classes tend to emphasize the run time complexity, but space complexity can be just as important, e.g if your program can't fit into the size of primary memory for a supercomputing application, if often will go from being feasible to compute, to impossible.

- The practical goal of analysis of algorithms is to predict the performance of different algorithms, as a function of the input size, in order to guide design decisions.
- The goal is to make meaningful comparisons between algorithms, but some problems:
  - The relative performance might depend on characteristics of hardware. Alg 1 faster machine A, Alg 2 faster machine B. Specify a machine model.
  - Relative performance might depend on details of dataset. Thus usually look at best, worst and average case behavior.
  - Relative importance depends on the size of the input. A sorting algorithm will be slower for a small list than for a very large one.
- Usual solution for last: specify run time (or number of operations) as a function of problem size. Compare functions **asymptotically** as the problem size increases.

Suppose you have analyzed two algorithms and expressed their run times in terms of the size of the input: Algorithm A takes $100n + 1$ steps to solve a problem with size n; Algorithm B takes $n^2 + n + 1$ steps.

Which algorithm is "better" in terms of time complexity (number of steps)?

The following table shows run time of algorithms for different problem sizes:

Input size	Run time of Algorithm A	Run time of Algorithm B
10	1 001	111
100	10 001	10 101
1 000	100 001	1 001 001
10 000	1 000 001	$> 10^{10}$

At $n = 10$ Algorithm A looks pretty bad, it takes 10 times longer than B. But for $n = 100$ they are about the same, and for larger values of n, Algorithm A will be much better.

The fundamental reason is that for large values of n, any function that contains an n^2 term will grow faster than a function whose leading term is n. The **leading term** is the term with the highest exponent.

For Algorithm A the leading term has a large coefficient, 100, which is why B does better than A for small n. But regardless of constant coefficients like this, there will always be some point, some value of n where $an^2 > bn$.

The same argument applies to the non-leading terms, which is why we can ignore the $n + 1$ part of the n^2 algorithm B. Even if the run time of Algorithm A were $n + 1000000$, it would still do better than Algorithm B for sufficiently large n.

In general we expect an algorithm with a smaller leading term to be a better algorithm for large problems, but for smaller problems there may be a **crossover point** where another algorithm is better.

1.2 Order of Growth and the Big-Oh $O()$ Notation

An **order of growth** is a set of functions whose asymptotic growth behavior is considered equivalent. For example, $2n$, $100n$ and $n+1$ belong to the same order of growth, which is written $O(n)$ in **Big-Oh notation** and often called **linear** because every function in the set grows linearly with n.

All functions with the leading term n^2 belong to the $O(n^2)$; they are **quadratic**, which is a fancy word for functions with the leading term n^2.

The following table shows some of the orders of growth that appear most commonly in algorithm analysis, in increasing order of badness.

Order of Growth	Name
$O(1)$	Constant
$O(\log_b n)$	logarithmic (for any b)
$O(n)$	linear
$O(n \log_b n)$	"en log en"
$O(n^2)$	quadratic
$O(n^3)$	cubic
$O(c^n)$	exponential (for any c)

2 Second Session (11:45 - 12:30)

2.1 Searching Algorithms

- Why is Binary Search $O(\log_2 n)$?
- Why is Bubble Sort $O(n^2)$?

2.2 Sorting Arrays

- Merge Sort
  - A function to merge two sorted subarrays together.
  - Using merge() function, solve sort recursively, (array of size 1, sorted base case, array of size > 2 divide in 2, recursive call on subparts, merge the sorted subparts).

3 Third Session (12:40 - 1:40)

Algorithm	**Best case**	**Expected**	**Worst case**
Bubble sort	$O(n^2)$	$O(n^2)$	$O(n^2)$
Insertion sort	$O(n^2)$	$O(n^2)$	$O(n^2)$
Merge Sort	$O(n\ log\ n)$	$O(n\ log\ n)$	$O(n\ log\ n)$
Linear search	$O(1)$	$O(n)$	$O(n)
Binary search	$O(1)$	$O(log\ n)$	$O(log\ n)$