Basics of Algorithms Analysis

CS 2c03

Ryszard Janicki

Department of Computing and Software, McMaster University, Hamilton, Ontario, Canada

Measuring the Running Time of a Program I

- The running time of a program depends on factors such as:
  1. the input to the program,
  2. the quality of code generated by the compiler used to create the object program,
  3. the nature and speed of the instructions on the machine used to execute the program, and
  4. the time complexity of the algorithm underlying the program.
- The fact that running time depends on the input tells us that the running time of a program should be defined as a function of the input.
- Often, the running time depends not on the exact input but only on the "size" of the input.

Measuring the Running Time of a Program II

- It is customary, then, to talk of $T(n)$, the **running time** of a program on inputs of size n. For example, some program may have a running time $T(n) = cn^2$, where c is a constant.
- The units of $T(n)$ will be left unspecified, but we can think of $T(n)$ as being the number of instructions executed on an idealized computer.
- For many programs, the running time is really a function of the particular input, and not just of the input size.
- In that case we define $T(n)$ to be **the worst case** running time, that is, the maximum, over all inputs of size n, of the running time on that input.
- We also consider $T_{avg}(n)$, **the average, over all inputs of size** n, of the running time on that input.
- While $T_{avg}(n)$ appears a fairer measure, *it is often fallacious to assume that all inputs are equally likely.*

Measuring the Running Time of a Program III

- In practice, the average running time is often much harder to determine than the worst-case running time, both because the analysis becomes mathematically intractable and because the notion of "average" input frequently has no obvious meaning.
- Thus, we shall use worst-case running time as the principal measure of time complexity, although we shall mention average-case complexity wherever we can do so meaningfully.

Cost of Basic Operations

Observation. Most primitive operations take constant time.

operation	example	nanoseconds †
variable declaration	`int a`	c_1
assignment statement	`a = b`	c_2
integer compare	`a < b`	c_3
array element access	`a[i]`	c_4
array length	`a.length`	c_5
1D array allocation	`new int[N]`	$c_6\ N$
2D array allocation	`new int[N][N]`	$c_7\ N^2$

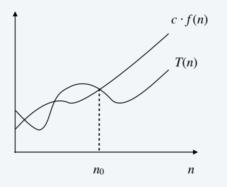
Caveat. Non-primitive operations often take more than constant time.

- We will assume that for basic operations $T(n) = c$.

Big-Oh notation

Definition (Upper bounds)

$T(n)$ is $O(f(n))$ if there exist constants $c > 0$ and $n_0 \geq 0$ such that $T(n) \leq c \cdot f(n)$ for all $n \geq n_0$.



Example

$T(n) = 32n^2 + 17n + 1$.

- $T(n)$ is $O(n^2)$. $\leftarrow$ choose $c = 50$, $n_0 = 1$
- $T(n)$ is also $O(n^3)$.
- $T(n)$ is neither $O(n)$ nor $O(n \log n)$.

Typical usage. Insertion makes $O(n^2)$ compares to sort n elements.

Notational abuses

- **Equals sign.** $O(f(n))$ is a set of functions, but computer scientists often write $T(n) = O(f(n))$ instead of $T(n) \in O(f(n))$.

Example

Consider $f(n) = 5n^3$ and $g(n) = 3n^2$.

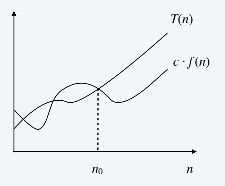
- We have $f(n) = O(n^3) = g(n)$.
- Thus, $f(n) = g(n)$.

- **Domain.** The domain of $f(n)$ is typically the natural numbers $\{0, 1, 2, \ldots\}$.
  - Sometimes we restrict to a subset of the natural numbers. Other times we extend to the reals.
- **Nonnegative functions.** When using big-Oh notation, we assume that the functions involved are (asymptotically) nonnegative.
- **Bottom line.** OK to abuse notation; not OK to misuse it.

Big-Omega notation

Definition (Lower bounds)

$T(n)$ is $\Omega(f(n))$ if there exist constants $c > 0$ and $n_0 \geq 0$ such that $T(n) \geq c \cdot f(n)$ for all $n \geq n_0$.



Example

$T(n) = 32n^2 + 17n + 1$.

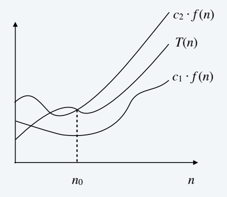
- $T(n)$ is both $\Omega(n^2)$ and $\Omega(n)$. $\leftarrow$ choose $c = 32$, $n_0 = 1$
- $T(n)$ is also $O(n^3)$.
- $T(n)$ is neither $\Omega(n^3)$ nor $\Omega(n^3 \log n)$.

Typical usage. Any compare-based sorting algorithm requires $\Omega(n \log n)$ compares in the worst case. We will discuss details later.

Big-Theta notation

Definition (Tight bounds)

$T(n)$ is $\Theta(f(n))$ if there exist constants $c_1 > 0$, $c_2 > 0$ and $n_0 \geq 0$ such that $c_1 \cdot f(c) \leq T(n) \leq c_2 \cdot f(n)$ for all $n \geq n_0$.



Example

$T(n) = 32n^2 + 17n + 1$.

- $T(n)$ is $\Theta(n^2)$. $\leftarrow$ choose $c_1 = 32, c_2 = 50, n_0 = 1$
- $T(n)$ is neither $\Theta(n)$ nor $\Theta(n^3)$.

Typical usage. Mergesort makes $\Omega(n \ \log \ n)$ compares to sort n elements. We will discuss details later.

Useful facts

Proposition

If $\lim_{n\to\infty} \frac{f(n)}{g(n)} = c > 0$*, then* $f(n)$ *is* $\Theta(g(n))$*.*

Proof.

By definition of the limit, there exists n_0 such such that for all $n \geq n_0$

$$\frac{1}{2}c < \frac{f(n)}{g(n)} < 2c$$

- Thus, $f(n) \leq 2cg(n)$ for all $n \geq n_0$, which implies $f(n)$ is $O(g(n))$.
- $f(n) \geq \frac{1}{2}cg(n)$ for all $n \geq n_0$, which implies $f(n)$ is $\Omega(g(n))$. $\square$

Proposition

If $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$*, then* $f(n)$ *is* $O(g(n))$*.*

Tilde Notation (Textbook)

In the textbook, if $f(n)$ is $\Theta(g(n))$ we will write

$$f(n) \sim g(n).$$

Formally:

Definition

$f(x) \sim g(x) \iff \lim_{n\to\infty} \frac{f(n)}{g(n)} = 1.$

Asymptotic bounds for some common functions

- **Polynomials.** Let $T(n) = a_0 + a_1 n + \ldots + a_d n^d$ with $a_d > 0$. Then, $T(n)$ is $\Theta(n^d)$.

  Proof. $\lim_{n\to\infty} \frac{a_0 + a_1 n + \ldots + a_d n^d}{n^d} = a_d > 0$.
- **Logarithms.** $Theta(\log_a n)$ is $\Theta(\log_b n)$ for any constants $a, b > 0$.

  Proof. Since $\log_a n = \frac{\log_n n}{\log_b a}$.
- **Exponentials and polynomials.** For every $r > 1$ and every $d > 0$, n^d is $O(r^n)$.

  Proof. Since $\lim_{n\to\infty} \frac{n^d}{r^d} = 0$.

Big-Oh notation with multiple variables

Definition (Upper bounds)

$T(m,n)$ is $O(f(m,n))$ if there exist constants $c > 0$, $m^0 \geq 0$, and $n_0 \geq 0$ such that $T(m,n) \leq c \cdot f(m,n)$ for all $n \geq n_0$ and $m \geq m_0$.

Example

$T(m,n) = 32mn^2 + 17mn + 32n^3$.

- $T(m,n)$ is both $O(mn^2 + n^3)$ and $O(mn^3)$.
- $T(m,n)$ is neither $O(n^3)$ nor $O(mn^2)$.

Typical usage. Breadth-first search takes $O(m+n)$ time to find the shortest path from s to t in a digraph. We will discuss details later.

Why it matters

Table 2.1 The running times (rounded up) of different algorithms on inputs of increasing size, for a processor performing a million high-level instructions per second. In cases where the running time exceeds 10^{25} years, we simply record the algorithm as taking a very long time.

	n	$n \log_2 n$	n^2	n^3	1.5^n	2^n	$n!$
$n = 10$	$<$ 1 sec	$<$ 1 sec	$<$ 1 sec	$<$ 1 sec	$<$ 1 sec	$<$ 1 sec	4 sec
$n = 30$	$<$ 1 sec	$<$ 1 sec	$<$ 1 sec	$<$ 1 sec	$<$ 1 sec	18 min	10^{25} years
$n = 50$	$<$ 1 sec	$<$ 1 sec	$<$ 1 sec	$<$ 1 sec	11 min	36 years	very long
$n = 100$	$<$ 1 sec	$<$ 1 sec	$<$ 1 sec	1 sec	12,892 years	10^{17} years	very long
$n = 1{,}000$	$<$ 1 sec	$<$ 1 sec	1 sec	18 min	very long	very long	very long
$n = 10{,}000$	$<$ 1 sec	$<$ 1 sec	2 min	12 days	very long	very long	very long
$n = 100{,}000$	$<$ 1 sec	2 sec	3 hours	32 years	very long	very long	very long
$n = 1{,}000{,}000$	1 sec	20 sec	12 days	31,710 years	very long	very long	very long

Typical O(...): (notation $log\ n = log_2 n$)

- $O(log\ n)$ $O(log(log(n)) \ldots$
- $O(n)$
- $O(n\ log\ n)$
- $O(n^2)$ $O(n^k)$
- $O(2^n)$ $O(k^n)$

Classification:

$$\left.\begin{array}{l} O(log\ n) \\ O(n) \\ O(n\ log\ n) \end{array}\right\}\text{desired}$$

$O(n^k)$: may be acceptable for small k

$O(2^n)$: UNACCEPTABLE

Fact

For every $k \geq 0$ and every $\alpha > 1$, there exists n_0 such that for every $n > n_0$:

$$n^k < \alpha^n$$

Another classification:

$O(n^k)$: polynomial, i.e. **might be OK**

$O(\alpha^n)$: non-polynomial, i.e. **usually BAD**

Helpful Results

Lemma

1. $O(f(n)) + O(g(n)) = O(f(n) + g(n)) = O(\max(f(n), g(n))$
2. $O(f(n))O(g(n)) = O(f(n)g(n))$

- for each polynomial $f(n) = a_n x^n + a_{n-1}x^{n-1} + \ldots + a_1 x + a_0$, where $a_n \neq 0$, we have:

$$O(f(n)) = O(x^n)$$

- $O(2^n + n^{10000000000}) = O(2^n)$
- $O(n^{10000000000}) = O(2^n)$
- $O(2^n) \neq O(n^k)$ for any k
- Since $log_b n = \frac{1}{log\ b} log\ n$, for any b we have

$$O(log_b n) = O(log\ n)$$

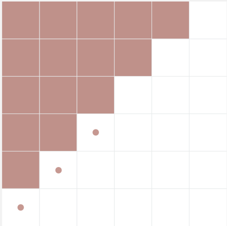
Some Examples I

Q. How many instructions as a function of input size N?

```
int count = 0;
for (int i = 0; i < N; i++)
   for (int j = i+1; j < N; j++)
      if (a[i] + a[j] == 0)
         count++;
```

$$0+1+2+\ldots+(N-1) = \frac{1}{2}N(N-1) = \binom{N}{2}$$

Pf. [n even]



$$0+1+2+\ldots+(N-1) = \underbrace{\frac{1}{2}N^2}_{\text{half of square}} - \underbrace{\frac{1}{2}N}_{\text{half of diagonal}}$$

- $T(n) = \Theta(N^2) = \sim N^2$ (loose loops counting).

Some Examples II

Q. Approximately how many array accesses as a function of input size N?

```
int count = 0;
for (int i = 0; i < N; i++)
   for (int j = i+1; j < N; j++)
      for (int k = j+1; k < N; k++)
         if (a[i] + a[j] + a[k] == 0)
            count++;
```

"inner loop"

$$\binom{N}{3} = \frac{N(N-1)(N-2)}{3!}$$

- $T(n) = \Theta(N^3) = \sim N^3$ (loose loops counting).

- **Goal**. Given a sorted array and a key, find index of the key in the array?
- **Binary search**. Compare key against middle entry.
  - Too small, go left.
  - Too big, go right.
  - Equal, found.

See Binary Seach Demo.

Binary Search: Time Complexity

Proposition

Binary search uses at most $1 + \log N$ key compares to search in a sorted array of size N, i.e. it has time complexity $T(N) = O(\log N)$.

Proof: *Sketch.*

Binary search recurrence: $T(N) \leq T(N/2) + 1$ for $N > 1$, with $T(1) = 1$.
Assume N is a power of 2.

$$
\begin{aligned}
T(N) &\leq T(N/2) + 1 && \text{[given]} \\
&\leq T(N/4) + \underbrace{1 + 1}_{2=\log 4} && \text{[apply recurrence to first term]} \\
&\leq T(N/8) + \underbrace{1 + 1 + 1}_{3=\log 8} && \text{[apply recurrence to second term]} \\
&\vdots \\
&\leq T(N/N) + \underbrace{1 + 1 + \ldots + 1}_{\log N} && \text{[stop applying, } T(1) = 1\text{]} \\
&= 1 + \log N = O(\log N).
\end{aligned}
$$

$\square$

Binary Search: Java

```
public static int binarySearch(int[] a, int key)
{
   int lo = 0, hi = a.length-1;
   while (lo <= hi)
   {
       int mid = lo + (hi - lo) / 2;
       if      (key < a[mid]) hi = mid - 1;
       else if (key > a[mid]) lo = mid + 1;
       else return mid;
  }
  return -1;
}
```

one "3-way compare"

Best case. Lower bound on cost.

- Determined by "easiest" input.
- Provides a goal for all inputs.

Worst case. Upper bound on cost.

- Determined by "most difficult" input.
- Provides a guarantee for all inputs.

Average case. Expected cost for random input.

- Need a model for "random" input.
- Provides a way to predict performance.

Summary of Complexity Measurements

notation	provides	example	shorthand for	used to
Tilde	leading term	$\sim 10\,N^2$	$10\,N^2$ $10\,N^2 + 22\,N \log N$ $10\,N^2 + 2\,N + 37$	provide approximate model
Big Theta	asymptotic order of growth	$\Theta(N^2)$	$\frac{1}{2}\,N^2$ $10\,N^2$ $5\,N^2 + 22\,N \log N + 3N$	classify algorithms
Big Oh	$\Theta(N^2)$ and smaller	$O(N^2)$	$10\,N^2$ $100\,N$ $22\,N \log N + 3\,N$	develop upper bounds
Big Omega	$\Theta(N^2)$ and larger	$\Omega(N^2)$	$\frac{1}{2}\,N^2$ N^5 $N^3 + 22\,N \log N + 3\,N$	develop lower bounds

- The most popular and the most useful is **Big-Oh**.