Fundamental Algorithms

Dan Barrish-Flood
What is an Algorithm?

• A problem is a precise specification of input and desired output

• An algorithm is a precise specification of a method for solving a problem.
What We Want in an Algorithm

• Correctness
  – Always halts
  – Always gives the right answer

• Efficiency
  – Time
  – Space

• Simplicity
A Typical Problem

Sorting

Input:
A sequence $b_1, b_2, \ldots b_n$

Output:
A permutation of that sequence $b_1, b_2, \ldots b_n$ such that

$$b_1 \leq b_2 \leq b_3 \leq \ldots \leq b_n$$
A Typical Algorithm

Insertion-Sort($A$)

1 for $j \leftarrow 2$ to $\text{length}[A]$
2 do $\text{key} \leftarrow A[j]$
3 $\triangleright$ Insert $A[j]$ into the sorted sequence $A[1..j-1]$
4 $i \leftarrow j - 1$
5 while $i > 0$ and $A[i] > \text{key}$
6 do $A[i+1] \leftarrow A[i]$
7 $i \leftarrow i - 1$
8 $A[i+1] \leftarrow \text{key}$
Running Time

Insertion - Sort($A$)

1 \textbf{for} j ← 2 \textbf{to} length[$A$]

2 \textbf{do} key ← $A[j]$

3 \quad \text{Insert } A[j] \text{ into the sorted sequence } A[1..j - 1]

4 \quad i ← j - 1

5 \quad \textbf{while} i > 0 \text{ and } A[i] > key

6 \quad \textbf{do} $A[i + 1] ← A[i]$

7 \quad i ← i - 1

8 \quad A[i + 1] ← key

$t_j$ is the number of times the “while” loop test in line 5 is executed for that value of $j$. Total running time is

$$c_1 (n) + c_2(n-1) + \ldots + c_8(n-1)$$
A Better Way to Estimate Running Time

- Trying to determine the exact running time is tedious, complicated and platform-specific.
- Instead, we’ll bound the worst-case running time to within a constant factor for large inputs.
- Why worst-case?
  - It’s easier.
  - It’s a guarantee.
  - It’s well-defined.
  - Average case is often no better.
Insertion Sort, Redux

• Outer loop happens about $n$ times.
• Inner loop is done $j$ times for each $j$, worst case.
• $j$ is always $< n$
• So running time is upper-bounded by about $n \times n = n^2$

• “Insertion sort runs in $\theta(n^2)$ time, worst-case.”
• We’ll make all the “about”’s and “rough”’s precise.

• These so-called asymptotic running times dominate constant factors for large enough inputs.
Functions

• constants
  \(0, \, 1\)

• polynomials
  \(n, \, 2n^3 - 5\)

• exponentials
  \(2^n, \, 3^{n^2}\)

• logarithms and polylogarithms
  \(\log_2 n, \, \log^2 n\)

• factorial
  \(n! = n \times (n - 1) \times \ldots \times 1\)
Polynomials

\[ \sum_{i=0}^{d} a_i n^i \]

E.g. \(2n^3 - 5 = -5n^0 + 0n^1 + 0n^2 + 2n^3\)

(More generally, Sum of terms \(an^c\) where \(c\) is some real number. E.g.

\(n^{1/2} = \sqrt{n}\), \(n^{\sqrt{3}}\)
f(n) is *monotonically increasing* iff $m \leq n$ implies $f(m) \leq f(n)$.

I.e. It never “goes down” — always increases or stays the same.

For *strictly increasing*, replace $\leq$ in the above with $<$.  

I.e. The function is always going up.
Are constant functions monotonically increasing?

Yes! (But not \textit{strictly} increasing.)

If \( c > 0 \), then \( n^c \) is strictly increasing.

So even \( n^{.001} \) (\( = 1000\sqrt{n} \)) increases forever to \( \infty \).
Exponentials

\[ f(n) = a^n \]

Facts:

\[ a^0 = 1 \]

\[ a^1 = a \]

\[ a^{-1} = \frac{1}{a} \quad a^{\frac{1}{n}} = n\sqrt{a} \]

\[ a^{m+n} = a^m a^n \]

\[ 2^{n-1} = 2^n 2^{-1} = \frac{1}{2} 2^n \]

\[ (a^m)^n = a^{mn} = (a^n)^m \]

\[ 2^{\frac{n}{2}} = 2^{(\frac{1}{2})n} = \sqrt{2}^n \]

If \( a > 1 \), \( a^n \) is strictly increasing to \( \infty \)
Logarithms

$log_b n$ is the number $x$ such that $b^x = n$.

Some notation:

- $\lg n = \log_2 n$
- $\ln n = \log_e n$
- $\log^k n = (\log n)^k$
- $\log \log n = \log (\log n)$

Recall:

- $\log_b b = 1$
- $\log_b 1 = 0$

For $0 < x < 1$, $\log_b x < 0$. 
Log Facts

1. \( n = b^{\log_b n} \)
   (definition)

2. \( \log_b(xy) = \log_b x + \log_b y \)

3. \( \log_b a^n = n \log_b a \)

4. \( \log_b x = \frac{\log_c x}{\log_c b} \)

5. \( \log_b(1/a) = -\log_b a \)

6. \( \log_b a = 1/(\log_a b) \)

7. \( a^{\log_b c} = c^{\log_b a} \)

- if \( b > 1 \), then \( \log_b n \) is strictly increasing.
More About Logs

A polylogarithm is just

$$\log^k n$$

For $k > 0$ and $n > 0$, $\log^k n$ is strictly increasing to $\infty$. 
Factorial

\[ n! = n \times (n-1) \times (n-2) \times \ldots \times 1 \]

\[ = \prod_{i=1}^{n} i \]

0! = 1  (by definition)

\( n! \) is strictly increasing to \( \infty \)

Stirling's approximation:

\[ n! = \sqrt{2\pi n} \left(\frac{n}{e}\right)^n (1 + \Theta(\frac{1}{n})) \]

Even without Stirling, we can see:

For \( n > 2, \quad 2^n < n! < n^n \)
More About Monotonically Increasing Functions

If \( f(n) \) and \( g(n) \) are monotonically increasing, then so are

\[
\begin{align*}
    f(n) + g(n) \\
    f(g(n)) \\
    f(n) \times g(n)
\end{align*}
\]

and if \( f(n) \) and \( g(n) \) are also \( > 0 \), then

\[
\begin{align*}
    f(n) \times g(n)
\end{align*}
\]

is monotonically increasing, too.

Examples: \( n \log n \)

\[
\begin{align*}
    \log \log n \\
    2^n + n^3
\end{align*}
\]
Bounding Functions

• We are more interested in finding upper and lower bounds on functions that describe the running times of algorithms, rather than finding the exact function.

• Also, we are only interested in considering large inputs: everything is fast for small inputs.

• So we want asymptotic bounds.

• Finally, we are willing to neglect constant factors when bounding. They are tedious to compute, and are often hardware-dependent anyway.
Example

$3n + 8$ is certainly not upper-bounded by $n$...

but when $n \geq 2$, $3n + 8$ is upper-bounded by $10n$. 
Notation for Function Bounds

(The most important definitions in the course. We'll use them every day.)

Upper Bound: $O$ ("big - oh")
We say $f(n) = O(g(n))$ iff there exist two positive constants $c$ and $n_0$ such that

$$f(n) \leq cg(n) \text{ for all } n \geq n_0$$

E.g.

$$3n + 8 = O(n)$$

[Proof: choose $c = 10$ and $n_0 = 2$; there are many other choices.]
Big-Oh Facts

- The “=” is misleading. E.g
  \[ O(g(n)) = f(n) \]
  is meaningless.
- It denotes an upper bound, not necessarily a tight bound. Thus,
  \[ n = O(n) \]
  \[ n = O(n^2) \]
  \[ n = O(2^n) \] etc.
- Transitivity:
  If \( f(n) = O(g(n)) \) and \( g(n) = O(h(n)) \) then \( f(n) = O(h(n)) \)
Examples

We’ve seen that $3n + 8 = O(n)$.
Is $10n = O(3n + 8)$?
Yes: choose $c = 10$, $n_0 = 0$:

$$10n \leq 30n + 80$$
More Examples

\[2n^2 + 5n - 8 = O(n^2)?\]

Yes:

\[2n^2 + 5n - 8 \leq cn^2\]

\[\div n^2: \quad 2 + \frac{5}{n} - \frac{8}{n^2} \leq c\]

Choose \(c = 3\); subtracting 2:

\[\frac{5}{n} - \frac{8}{n^2} \leq 1\]

Choose \(n_o = 5\)
More Examples

Is $2^n = O(3^n)$?

Yes: in fact, $2^n \leq 3^n \quad (c = 1, \; n_0 = 0)$

Is $3^n = O(2^n)$?

$3^n \leq c2^n$

$\div 2^n: \quad 3^n / 2^n \leq c$

$(3 / 2)^n \leq c$

But since $3/2 > 1$, $(3/2)^n$ is strictly increasing to $\infty$ - it cannot be upper-bounded by any constant.
Lower Bound Notation

\[ f(n) = \Omega(g(n)) \] iff there are positive constants \( c \) and \( n_0 \) such that \( f(n) \geq cg(n) \) for all \( n \geq n_0 \).

(Same as definition of \( O \), with \( \leq \) replaced by \( \geq \).)

E.g.

\[ 3^n = \Omega(2^n) \]
\[ n^2 = \Omega(n) \]

In fact:

If \( f(n) = O(g(n)) \), then \( g(n) = \Omega(f(n)) \), and vice versa.
“Tight” Bound Notation
(Both upper and lower)

\[ f(n) = \Theta(g(n)) \iff \text{there are positive constants } c_1, c_2, \text{ and } n_0 \text{ such that } c_1 g(n) \leq f(n) \leq c_2 g(n) \text{ for all } n \geq n_0. \]
More Facts

\( \Theta \) is transitive, just like \( O \) and \( \Omega \).

\[
f(n) = \Theta(g(n)) \iff f(n) = O(g(n)) \\
\text{and } f(n) = \Omega(g(n))
\]

\[
f(n) = \Theta(g(n)) \iff f(n) = O(g(n)) \\
\text{and } g(n) = O(f(n))
\]

\[
f(n) = \Theta(g(n)) \iff g(n) = \Theta(f(n))
\]
Examples

We've seen that \(3n + 8 = O(n)\), and \(n = O(3n + 8)\) so
\[
3n + 8 = \Theta(n).
\]

\[
2n^2 + 5n - 8 = \Theta(n^2).\] This is because :

We saw earlier that \(2n^2 + 5n - 8 = O(n^2)\)

Also, \(2n^2 + 5n - 8 = \Omega(n^2) [c = 1, n_0 = 2]\).

\(n^2 \neq \Theta(n^3)\). This is because :
\[
n^2 = O(n^3), \text{ but not vice versa.}
\]

\[
\log_{b_1} n = \Theta(\log_{b_2} n)
\]

Since log functions differ by a constant factor.

Therefore, we can write \(\Theta(lg n)\) for all logarithmic growth.
Another Upper-Bound Notation

We say $f(n) = o(g(n))$ ["little oh"] iff

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = 0.$$ 

E.g. $2n = o(n^2)$

$2n^2$ is not $o(n^2)$.

Think of $f(n) = o(g(n))$ as

"$f(n)$ is (asymptotically) negligible with respect to $g(n)$."
Functions and Bounds

Constant functions:

All constant functions are $\Theta(1)$. (also $\Theta(2), \Theta(3)$, etc., but we use 1 by convention.)

If $f(n)$ is a constant function and $g(n)$ is strictly increasing to $\infty$, then $f(n) = o(g(n))$.

(Proof: $g(n) \to \infty$, so $\lim_{n \to \infty} \frac{f(n)}{g(n)} = 0$.)
Logarithms

\[ \text{lg} n = o(n) \]

In fact,

\[ (*) \quad \log^b n = o(n^a) \text{ for any constants } a, b \]

where \( a > 0 \).

So - perhaps surprisingly -

\[ \text{lg}^{100} n = o(n^{.01}) \]

in spite of the fact that for \( n = 4 \),

\[ \text{lg}^{100} n = 2^{100} \text{ (huge!)} \text{ while } n^{.01} = 1 + \varepsilon \]

Substituting \( \text{lg} n \) for \( n \) in (*) , we see \( (\text{lg} \text{lg} n)^b = o(\text{lg}^a n) \)

and so on.
Polynomials

Any polynomial of degree \( d \) whose leading coefficient is positive is \( \Theta(n^d) \).

**Proof:**

Homework for next class.

Substituting \( 2^n \) for \( n \) in \( \lg^b n = o(n^a) \), \( a > 0 \), we have:

\[
(lg(2^n))^b = o((2^n)^a)
\]

\[
n^b = o((2^a)^n) \quad \text{(recall \( lg n \) means log base 2)}
\]

Renaming \( 2^a \), (where \( a > 0 \)) to \( a \), (where \( a > 1 \)),

\[
n^b = o(a^n), \ a > 1 \quad \text{i.e. polynomials} = o \text{ (exponentials)}
\]

Thus, \( n^{100} = o(1.001^n) \)
Comparing Functions

$n$ vs. $\sqrt{n}$

$2^n$ vs. $2^{n/2}$

$n \lg n$ vs. $n^2$

$\lg n$ vs. $\ln n^2$

$n / \lg n$ vs. $\lg n$

$n$ vs. $\lg^2 n$

$n \lg n$ vs. $n$

$n^{\lg n}$ vs. $\lg(n^n)$

$\sqrt{\lg n}$ vs. $\lg \sqrt{n}$