Basics of Algorithm Analysis

- We measure running time as a function of \( n \), the size of the input (in bytes assuming a reasonable encoding).
- We work in the RAM model of computation. All “reasonable” operations take “1” unit of time. (e.g. +, *, -, /, array access, pointer following, writing a value, one byte of I/O...)

What is the running time of an algorithm

- Best case (seldom used)
- Average case (used if we understand the average)
- Worst case (used most often)
Example

1 input: \( A[n] \)
2 for \( i = 1 \) to \( n \)
3 \hspace{1em} if \( (A[i] == 7) \)
4 \hspace{1em} for \( j = 1 \) to \( n \)
5 \hspace{2em} for \( k = 1 \) to \( n \)
6 \hspace{3em} Print “hello”

- What is the worst case running time?
- What is the best case running time?
- What is the average case running time?
Example

1 input: $A[n]$
2 for $i = 1$ to $n$
3     if ($A[i] == 7$)
4         for $j = 1$ to $n$
5             for $k = 1$ to $n$
6                 Print “hello”

- What is the worst case running time? $O(n^3)$
- What is the best case running time? $O(n)$
- What is the average case running time? What is an average array?
How do we measure the running time?

We measure as a function of $n$, and ignore low order terms.

- $5n^3 + n - 6$ becomes $n^3$
- $8n \log n - 60n$ becomes $n \log n$
- $2^n + 3n^4$ becomes $2^n$
Asymptotic notation

big-O

\[ O(g(n)) = \{ f(n) : \text{there exist positive constants } c \text{ and } n_0 \text{ such that } 0 \leq f(n) \leq cg(n) \text{ for all } n \geq n_0 \}. \]

Alternatively, we say

\[ f(n) = O(g(n)) \text{ if there exist positive constants } c \text{ and } n_0 \text{ such that } 0 \leq f(n) \leq cg(n) \text{ for all } n \geq n_0 \]

Informally, \( f(n) = O(g(n)) \) means that \( f(n) \) is asymptotically less than or equal to \( g(n) \).

big-Ω

\[ \Omega(g(n)) = \{ f(n) : \text{there exist positive constants } c \text{ and } n_0 \text{ such that } 0 \leq cg(n) \leq f(n) \text{ for all } n \geq n_0 \}. \]

Alternatively, we say

\[ f(n) = \Omega(g(n)) \text{ if there exist positive constants } c \text{ and } n_0 \text{ such that } 0 \leq cg(n) \leq f(n) \text{ for all } n \geq n_0 \]

Informally, \( f(n) = \Omega(g(n)) \) means that \( f(n) \) is asymptotically greater than or equal to \( g(n) \).
big-$\Theta$

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

Informally, \( f(n) = \Theta(g(n)) \) means that \( f(n) \) is asymptotically equal to \( g(n) \).

**INFORMAL summary**

- \( f(n) = O(g(n)) \) roughly means \( f(n) \leq g(n) \)
- \( f(n) = \Omega(g(n)) \) roughly means \( f(n) \geq g(n) \)
- \( f(n) = \Theta(g(n)) \) roughly means \( f(n) = g(n) \)
- \( f(n) = o(g(n)) \) roughly means \( f(n) < g(n) \)
- \( f(n) = w(g(n)) \) roughly means \( f(n) > g(n) \)
Big-O proofs

(turn on light)

- $3n = O(n^2)$
- $2n + 7 = O(n)$
- $n^{\log n} = O(2^n)$
Use of big-O

\[ 2n + 7 = O(n) \]

\[ 2n + 7 = O(n^3) \]

\[ 2n + 7 = O(n^{4.5} \log n) \]

\[ 2n + 7 = O(2^n) \]

Which of these do we care about?
Use of big-O

\[ 2n + 7 = O(n) \]

\[ 2n + 7 = O(n^3) \]

\[ 2n + 7 = O(n^{4.5} \log n) \]

\[ 2n + 7 = O(2^n) \]

Which of these do we care about?

- Given a function \( f(n) \), we want to know the “smallest” \( g(n) \) such that \( f(n) = O(g(n)) \) and \( g(n) \) is “simple”
Simple Functions

• Given a function $f(n)$, we want to know the “smallest” $g(n)$ such that $f(n) = O(g(n))$ and $g(n)$ is “simple”

• Typical simple functions include (but are not limited to)
  
  - $1$
  - $\log \log n$
  - $\log n$
  - $\log^2 n$
  - $n$
  - $n \log n$
  - $n^2$
  - $n^3$
  - $2^n$
  - $n!$

• We use these to classify algorithms into classes

See chart for justification
Polynomial Time

An algorithm runs in polynomial time if, on an input of size $n$, its running time is $O(n^k)$ for some constant $k$.

$2^n$ is NOT polynomial. Let’s try to prove that it is polynomial and see what goes wrong.
Proving Omega and Theta

\[ f(n) = \Omega(g(n)) \text{ if there exist positive constants } c \text{ and } n_0 \text{ such that } \\
0 \leq cg(n) \leq f(n) \text{ for all } n \geq n_0 \} . \]

\[ f(n) = \Theta(g(n)) \text{ if and only if } f(n) = O(g(n)) \text{ and } f(n) = \Omega(g(n)). \]
3 useful formulas

Arithmetic series

\[ \sum_{i=1}^{n} i = \frac{n(n + 1)}{2} \]

Geometric series

\[ \sum_{i=0}^{\infty} a^i = \frac{1}{1 - a} \quad \text{for} \quad 0 < a < 1 \]

Harmonic series

\[ \sum_{i=1}^{n} \frac{1}{i} = \ln n + O(1) = \Theta(\ln n) \]
Arithmetic Series in PseudoCode

1 for i = 1 to n
2 for j = 1 to n
3 Jump up and down

compared to

1 for i = 1 to n
2 for j = 1 to i
3 Jump up and down
Geometric Series

1 for $i = 1$ to $\log n$
2 for $j = 1$ to $2^i$
3 Jump up and down

or

1 $\text{JUMP}(n)$
2 if $n = 1$
3 Jump up and down once
4 else
5 Jump up and down $n$ times
6 $\text{JUMP}([n/2])$
A few facts about logs

- $\log_b a = \frac{\log_c a}{\log_c b}$ for any $c > 1$
- therefore $\ln n = O(\log n)$
- in general, the base of the logarithm in a big-O statement is not important

$$n + \frac{n}{2} + \frac{n}{3} + \frac{n}{4} + \frac{n}{5} + \ldots + 1 = n \left(1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \frac{1}{5} + \ldots + \frac{1}{n}\right) = O(n \log n)$$
Algorithmic Correctness

- Very important, but we won’t typically prove correctness from first principles.
- We will use loop invariants
- We will use other problem specific methods
Divide and Conquer

- Divide a problem into pieces
- **Recursively** solve the pieces
- Combine the solutions to the subproblems

**Strassen**

- divide into $7 \frac{n}{2} \times \frac{n}{2}$ size problems
- solved recursive problems
- used 18 additions to combine the pieces
MergeSort

1 \text{MergeSort}(A, p, r)
2 \text{if } p < r
3 \quad q = \lfloor (p + r)/2 \rfloor
4 \quad \text{MergeSort}(A, p, q)
5 \quad \text{MergeSort}(A, q + 1, r)
6 \quad \text{Merge}(A, p, q, r)

Let $T(n)$ be the running time of MergeSort on $n$ items. Merge takes $O(n)$ time.

$$T(n) = \begin{cases} 
\Theta(1) & \text{if } n = 1, \\
2T(n/2) + \Theta(n) & \text{if } n > 1.
\end{cases}$$
3 Recurrence Trees

1. $T(n) = 2T(n/2) + n$
2. $T(n) = 2T(n/2) + 1$
3. $T(n) = 2T(n/2) + n^2$
Master Theorem

Master Theorem for Recurrences  Let $a \geq 1$ and $b > 1$ be constants, let $f(n)$ be a function, and let $T(n)$ be defined on the non-negative integers by the recurrence

$$T(n) = aT(n/b) + f(n),$$

where we interpret $n/b$ to mean either $\lfloor n/b \rfloor$ or $\lceil n/b \rceil$. Then $T(n)$ can be bounded asymptotically as follows.

1. If $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$.
2. If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a \lg n})$.
3. If $f(n) = \Omega(n^{\log_b a + \epsilon})$ for some constant $\epsilon > 0$, and if $af(n/b) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large $n$, then $T(n) = \Theta(f(n))$. 