COP 4531
Complexity & Analysis of Data Structures & Algorithms

Growth of Functions
and Algorithmic Analysis

Thanks to several people who contributed to these slides including the text authors and Piyush Kumar
Algorithm Analysis
Time and Space Complexity

• Generally a function of the input size
  • E.g., sorting, multiplication
  - How we characterize input size depends:
    • Sorting: number of input items
    • Multiplication: total number of bits
    • Graph algorithms: number of nodes & edges
    • Etc
In this course

• We care most about asymptotic performance
  - How does the algorithm behave as the problem size gets very large?
    • Running time is our primary focus
    • Memory/storage requirements
    • Bandwidth/power requirements/logic gates/etc.
2.1 Computational Tractability

As soon as an Analytic Engine exists, it will necessarily guide the future course of the science. Whenever any result is sought by its aid, the question will arise - By what course of calculation can these results be arrived at by the machine in the shortest time? - Charles Babbage

Charles Babbage (1864)
Polynomial-Time

• Brute force. For many non-trivial problems, there is a natural brute force search algorithm that checks every possible solution.  
  - Typically takes $2^N$ time or worse for inputs of size $N$.  
  - Unacceptable in practice.
• Desirable scaling property. When the input size doubles, the algorithm should only slow down by some constant factor $C$.

There exists constants $c > 0$ and $d > 0$ such that on every input of size $N$, its running time is bounded by $c N^d$ steps.

• Def. An algorithm is **poly-time** if the above scaling property holds.
Worst VS Average Case Analysis

• **Worst case running time.** Obtain bound on largest possible running time of algorithm on input of a given size $N$.
  - Generally captures efficiency in practice.
  - Draconian view, but hard to find effective alternative.

• **Average case running time.** Obtain bound on running time of algorithm on random input as a function of input size $N$.
  - Hard (or impossible) to accurately model real instances by random distributions.
  - Algorithm tuned for a certain distribution may perform poorly on other inputs.
Worst-Case Polynomial-Time

- **Def.** An algorithm is **efficient** if its running time is polynomial.

- **Justification:** It really works in practice!
  - Although $6.02 \times 10^{23} \times N^{20}$ is technically poly-time, it would be useless in practice.
  - In practice, the poly-time algorithms that people develop almost always have low constants and low exponents.
  - Breaking through the exponential barrier of brute force typically exposes some crucial structure of the problem.

- **Exceptions.**
  - Some poly-time algorithms do have high constants and/or exponents, and are useless in practice.
  - Some exponential-time (or worse) algorithms are widely used because the worst-case instances seem to be rare.
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.

<table>
<thead>
<tr>
<th>$n$</th>
<th>$n \log_2 n$</th>
<th>$n^2$</th>
<th>$n^3$</th>
<th>$1.5^n$</th>
<th>$2^n$</th>
<th>$n!$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$n = 10$</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
</tr>
<tr>
<td>$n = 30$</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>18 min</td>
</tr>
<tr>
<td>$n = 50$</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>11 min</td>
<td>36 years</td>
</tr>
<tr>
<td>$n = 100$</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>1 sec</td>
<td>12,892 years</td>
<td>$10^{17}$ years</td>
</tr>
<tr>
<td>$n = 1,000$</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>1 sec</td>
<td>18 min</td>
<td>very long</td>
<td>very long</td>
</tr>
<tr>
<td>$n = 10,000$</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>2 min</td>
<td>12 days</td>
<td>very long</td>
<td>very long</td>
</tr>
<tr>
<td>$n = 100,000$</td>
<td>&lt; 1 sec</td>
<td>2 sec</td>
<td>3 hours</td>
<td>32 years</td>
<td>very long</td>
<td>very long</td>
</tr>
<tr>
<td>$n = 1,000,000$</td>
<td>1 sec</td>
<td>20 sec</td>
<td>12 days</td>
<td>31,710 years</td>
<td>very long</td>
<td>very long</td>
</tr>
</tbody>
</table>

Note: table from More Programming Pearls – so it is out of date. Now a reasonable bound is still $10^6$ Mips or 1 step takes a pico-second rather than a microsecond. So divide by a million. Note however that “very long” can be easily labeled instead as $10^{19}$ years.
2.2 Order of Growth

- A way to describe behavior of functions in the limit. We are interested in asymptotic efficiency.
- Describe growth of functions.
- Focus on what’s important by abstracting away low-order terms and constant factors.
- How we indicate running times of algorithms.
- A way to compare “sizes” of functions:
  \( O \approx \leq \)
  \( \Omega \approx \geq \)
  \( \Theta \approx = \)
  \( o \approx < \)
  \( \omega \approx > \)
**O-notation**

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

$g(n)$ is an asymptotic upper bound for $f(n)$. If $f(n) \in O(g(n))$, we write $f(n) = O(g(n))$. 
Exercise

• Prove that $f(n) = 2n^2$ is in $O(n^3)$
Exercise

• Prove that $n^2 + n$ is in $O(n^2)$
Exercise

• Prove that $n/1000$ is in $O(n^3)$
Log base doesn’t matter!

- Prove that $\log_a(n)$ is in $O(\log_b(n))$
- Proof:
  - Recall change of base formula:
    \[
    \log_a(n) = \frac{\log_b(n)}{\log_b(a)}
    \]
  - Set $c = 1/\log_b(a)$ and $n_0 = 1$
\[ \Omega(g(n)) = \{ f(n) : \text{there exist positive constants } c \text{ and } n_0 \text{ such that } 0 \leq cg(n) \leq f(n) \text{ for all } n \geq n_0 \}. \]

g(n) \text{ is an asymptotic lower bound for } f(n). \text{ If } f(n) \in \Omega(g(n)), \text{ we write } f(n) = \Omega(g(n)).
Exercise

• Prove that $n^2/100$ is in $\Omega(n)$
Exercise

- Prove that $n^{2.0001}$ is in $\Omega(n^2)$
Exercise

• Prove that $n^2 - 200n$ is in $\Omega(n)$
\[ \Theta(g(n)) = \{ f(n) \colon \text{there exist positive constants } c_1, c_2 \text{ and } n_0 \text{ such that } 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \text{ for all } n \geq n_0 \}. \]

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

\[ g(n) \text{ is an asymptotic tight bound for } f(n). \text{ If } f(n) \in \Theta(g(n)), \text{ we write } f(n) = \Theta(g(n)). \]
Exercise

• Prove that $n^2/2 - 2n$ is in $\Theta(n^2)$. 
o-notation

\[ o(g(n)) = \{ f(n) : \text{for all constants } c > 0, \text{ there exists a constant } n_0 \text{ such that } 0 \leq f(n) < cg(n) \text{ for all } n \geq n_0 \}. \]

How to prove that \( f(n) \) is \( o(g(n)) \)?
- Directly from definition
- Or, prove that \( \lim_{n \to \infty} \frac{f(n)}{g(n)} = 0 \)

You may use L'Hôpital's rule to simplify the equation.
Exercise

• Prove that $n^2/\lg n$ is in $o(n^2)$. 
Exercise

• Prove that $n^2/1000$ is not in $o(n^2)$. 
ω-notation

ω(g(n)) = \{ f(n): \text{for all constants } c > 0, \text{ there exists a constant } n_0 \text{ such that } 0 \leq cg(n) < f(n) \text{ for all } n \geq n_0 \}. 

How to prove that f(n) is w(g(n))? 
- Directly from definition 
- Or, prove that \[ \lim_{n \to \infty} \frac{f(n)}{g(n)} = \infty \]

You may use L'Hôpital's rule to simplify the equation.
Exercise

• Prove that $n^{2.001}$ is in $\omega(n^2)$. 
Exercise

• Prove that $n^2 \log n$ is in $\omega(n^2)$
Exercise

• Prove that \( n^2 \) is not in \( \omega(n^2) \)
Properties: Exercise 1

- Prove that if \( f = O(g) \) and \( g = O(h) \) then \( f = O(h) \).
- Prove that if $f = \Omega(h)$ and $g = \Omega(h)$ then $f + g = \Omega(h)$.
Properties

• Transitivity.
  - If \( f = O(g) \) and \( g = O(h) \) then \( f = O(h) \).
  - If \( f = \Omega(g) \) and \( g = \Omega(h) \) then \( f = \Omega(h) \).
  - If \( f = \Theta(g) \) and \( g = \Theta(h) \) then \( f = \Theta(h) \).

• Additivity.
  - If \( f = O(h) \) and \( g = O(h) \) then \( f + g = O(h) \).
  - If \( f = \Omega(h) \) and \( g = \Omega(h) \) then \( f + g = \Omega(h) \).
  - If \( f = \Theta(h) \) and \( g = O(h) \) then \( f + g = \Theta(h) \).
Asymptotic Bounds for Some Common Functions

- Polynomials. \(a_0 + a_1 n + \ldots + a_d n^d\) is \(\Theta(n^d)\) if \(a_d > 0\).

- Polynomial time. Running time is \(O(n^d)\) for some constant \(d\) independent of the input size \(n\).

- Logarithms. \(\Theta(\log_a n) = \Theta(\log_b n)\) for any constants \(a, b > 0\).

  can avoid specifying the base

- Logarithms. For every \(x > 0\), \(\log n = O(n^x)\).

  \(\log\) grows slower than every polynomial

- Exponentials. For every \(r > 1\) and every \(d > 0\), \(n^d = O(r^n)\).

  every exponential grows faster than every polynomial
2.3 Survey of some common running times

- $O(n)$
- $O(n \log n)$
- $O(n^2)$
- $O(n^3)$
- $O(n^k)$
- exponential time +
Code analysis: Exercise 1

• Analyze the running time of the following code segment using $\Theta$ notation.

```javascript
let k = 0
for (i=1; i<n; i++)
    k++
```

• SOLUTION: $\Theta(n)$
Code analysis: Exercise 2

• Analyze the running time of the following code segment using $\Theta$ notation.

Let $k=0$

for (i=1; i<n; i++)
    for (j=1; j<i; j++)
        k++

• SOLUTION: $\Theta(n^2)$
Code analysis: Exercise 3

- Analyze the running time of the following code segment using $\Theta$ notation.

```c
for (i=1; i<n; i++)
    for (j=1; j<n^2+1; j++)
        return 0;
```

- SOLUTION: $\Theta(1)$
Code analysis: Exercise 4

- Analyze the running time of the following code segment using \( \Theta \) notation.

\[
\text{for } (i=1; \ i<n; \ i++) \\
\quad \text{for } (j=1; \ j<\log n; \ j++) \\
\quad \quad A[j] = A[j]-1
\]

- SOLUTION: \( \Theta(n \ lg \ n) \)
Code analysis: Exercise 5

• Analyze the running time of the following code segment using $\Theta$ notation.

$x = n$
while (x>1)
    $x = x/2$

• SOLUTION: $\Theta(\lg n)$
Code analysis: Exercise 6

• Analyze the running time of the following code segment using $\Theta$ notation.

\[
x = 0 \\
\text{while (x<n)} \\
\quad x = x+3
\]

• SOLUTION: $\Theta(n)$
Code analysis: Exercise 7

m ← a_1
for i = 2 to n {
    if (a_i > m)
        m ← a_i
}

• What does this code do?
• What is its asymptotic running time?
Linear Time: $O(n)$

- Computes maximum of $n$ numbers $a_1, \ldots, a_n$.
- Linear time. Running time is at most a constant factor times the size of the input.

```plaintext
m ← a_1
for i = 2 to n {
    if (a_i > m)
        m ← a_i
}
```
Algorithm design exercise 1

• How would you combine two sorted lists $A = a_1, a_2, \ldots, a_n$ with $B = b_1, b_2, \ldots, b_n$ into a sorted whole?

• What is the running time of your solution?
Linear time: $O(n)$

- Claim. Merging two lists of size $n$ takes $O(n)$ time.
- Pf. After each comparison, the length of output list increases by 1.

```
i = 1, j = 1
while (both lists are nonempty) {
    if ($a_i \leq b_j$) append $a_i$ to output list and increment $i$
    else ($a_i \leq b_j$) append $b_j$ to output list and increment $j$
}
append remainder of nonempty list to output list
```
Algorithm design exercise 2

also referred to as linearithmic time

• How would you sort n elements?
• What is the runtime of your solution?

• Mergesort and heapsort are sorting algorithms that perform $O(n \log n)$ comparisons (more on this in the next section).
Algorithm design exercise 3

• Largest empty interval. Given $n$ time-stamps $x_1, \ldots, x_n$ on which copies of a file arrive at a server, what is largest interval of time when no copies of the file arrive?

• What is the runtime of your solution?
Solution

- Solution:
  - Sort the time stamps $O(n \lg n)$
  - perform a linear scan, computing the difference between consecutive time stamps. $(n)$
  - So the running time is $O(n \lg n+n)$ which is $O(n \lg n)$. 
Algorithm design exercise 4

- Given a list of \( n \) points in the plane \((x_1, y_1), \ldots, (x_n, y_n)\), find the pair that is closest.

- Analyse the runtime of your solution.
Quadratic Time: $O(n^2)$

- $O(n^2)$ solution. Try all pairs of points.

\[
\begin{align*}
\text{min} &\leftarrow (x_1 - x_2)^2 + (y_1 - y_2)^2 \\
\text{for } i = 1 \text{ to } n \{ \text{for } j = i+1 \text{ to } n \{ \\
&\quad d \leftarrow (x_i - x_j)^2 + (y_i - y_j)^2 \\
&\quad \text{if } (d < \text{min}) \\
&\quad\quad \text{min} \leftarrow d \\
&\quad \}\} \\
&\}\}
\]

--- don't need to take square roots
Algorithm design exercise 5

• Independent set of size k. Given a graph, are there k nodes such that no two are joined by an edge?
• Analyze the runtime of your solution.
Polynomial Time: $O(n^k)$ Time

- $O(n^k)$ solution. Enumerate all subsets of $k$ nodes.

```java
foreach subset $S$ of $k$ nodes {
    check whether $S$ is an independent set
    if ($S$ is an independent set)
        report $S$ is an independent set
}
```

- Check whether $S$ is an independent set = $O(k^2)$.
- Number of $k$ element subsets < $n^k / k!$
- $O(k^2 \cdot n^k / k!) = O(n^k)$.
Running time exercise 6

• Independent set. Given a graph, what is maximum size of an independent set?

• Analyze the runtime of your solution.
Exponential Time

- Enumerate all subsets. $O(n^2 2^n)$ solution.

```plaintext
S* ← φ
foreach subset S of nodes {
    check whether S in an independent set
    if (S is largest independent set seen so far)
        update S* ← S
}
```
Summary

- $\Theta(1)$: Constant Time, can’t beat it.
- $\Theta(\log n)$: Typically the speed of most efficient data structures (Binary tree search?)
- $\Theta(n)$: Needed by an algorithm to look at all its input.
Summary

- $\Theta(n^x), \ x > 1$ : Polynomial running time. Acceptable when exponent ($x$) / input data size is not too large.
- $\Theta(y^n), \ y > 1$ : Used when input is very small or worst case does not happen.
- $\Theta(n!) \text{ or } \Theta(n^n)$ : Useful for really small inputs most of the time. ($n < 20$)