CS 320 Fall 2022
Runtime Analysis (Big Oh)

Sanjay Rajopadhye
Colorado State University
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*
Outline: four topics

- Algorithm time complexity
- Plotting data and the function clubs
- Digression: line of sight algorithm
- A survey of common running times
Algorithm Time Complexity
Algorithm Time Complexity

How do we measure the complexity (time, space requirements) of an algorithm?

As a function of its input size (an integer, n) denoting:

- Number of inputs (e.g., sorting)
- Number of bits to represent the input (e.g., primality)
- Sometimes multiple parameters, e.g., knapsack
  - Number of objects, n
  - Knapsack capacity, C

We want to determine the running time as a function of problem sizes, and analyze them asymptotically.
How to measure time?

- Seconds/nano-seconds?
  - No, too specific & machine dependent

- Number of instructions executed?
  - No, still too specific & machine dependent

- # of code fragments that take constant time?
  - Yes

- What kind of fragments/instructions?
  - Arithmetic operations, memory accesses, finite combinations of these
How to measure space?

- Bits?
  - Too detailed, but sometimes necessary (e.g., knapsack capacity)

- Integers?
  - Nicer, but dangerous – we can code a whole program in a single arbitrary sized integer, so we have to be careful about the size. Better to use machine words i.e, fixed size (e.g., 64, collections of bits)
Worst/average case time

- A bound on the maximum possible running time of the algorithm of inputs of size n
  - Usually captures the notion, but may be an overestimate

- Average case
  - More accurate but difficult – need to describe what is the range of inputs, and what is the distribution, statistical analysis. Let $I$ be the set of inputs, and $P_i$ and $C_i$ be the probability and computation time of input $i$

$$\sum_{i \in I} P_i C_i$$

- Often a constant factor of worst case time

Same considerations for space and other measures.
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>$n$</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>
<td>4 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>
<td>$10^{25}$ years</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>
<td>very long</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>
<td>very long</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>
<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>
<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>
<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>
<td>very long</td>
</tr>
</tbody>
</table>
For many problems, there is a natural, but likely naïve, brute force search algorithm that checks every possible solution.

- Enumerating such solutions is usually an exponential function of $n$ (recall counting from CS220).
- Hence naïve

**Definition:** an algorithm is said to be polynomial time if there exist positive constants $c$ and $d$, such that on any input of size $n$, the running time is bounded by $c n^d$

What about an algorithm whose running time is $c n \lg n$?
(Why) is the distinction important?

- One the one hand, a polynomial function like $6.03 \times 10^{23} n^{20}$ is polynomial, it is too large in practice (e.g., for $n=10$)

- On the other hand, some algorithm whose worst-case execution time is exponential behave much better in practice because the worst-case instances are (seem to be) rare
  - Simplex method for solving linear programming

So why?

- In practice, the polynomials have a low degree and coefficients

- The difference between the polynomial-exponential barrier reveals interesting and crucial structure of the problem
Asymptotic growth rates

- We are building mathematical functions that model the execution time (or other properties) of programs and algorithms.
- Need a mechanism to compare them.
  - How do we compare numbers? Using the relations: <, >, ≤, and ≥
  - Partial/total orders
- The Big-Oh, Big-Omega and Big-Theta notation (introduced in CS 220) is such an order relation.

Here, \( f \preceq g \) means that \( f \) grows slower than \( g \) (and also that \( g \) grows faster than \( f \)). We may also use \( g \succeq f \). So the following claims mean the same thing
  - \( f(n) \preceq g(n) \)
  - \( g \succeq f \) or \( g \succeq f \)
  - \( f = O(g) \)
  - \( g(n) = \Omega(f(n)) \)

- Often, one of the functions is our (complicated) model \( T(n) \) and the other is a simpler function (e.g., a monomial)
F(n) is $O(G(n))$

F(n) is $\Omega(H(n))$

if $G(n) = c \cdot H(n)$
then $F(n)$ is $\Theta(G(n))$

These measures were introduced in CS220
**Basic definitions**

A function $T(n)$ is $O(f(n))$ if there exist constants $c > 0$ and $n_0 > 0$ such that

for all $n \geq n_0 : \; T(n) \leq c f(n)$

- **Example:** $T(n) = 32n^2 + 16n + 32$.
  - $T(n)$ is $O(n^2)$
  - **ALSO TRUE:**
    - $T(n)$ is $O(n^3)$
    - $T(n)$ is $O(2^n)$

- Many possible upper bounds for one function! We always look for the best (lowest) upper bound, but it is not always easy to establish
Properties of $\prec, \succ$ and $0$

- **Transitivity**
  - $f \preceq g$ and $g \preceq h$ implies $f \preceq h$

- **Additivity** (Additive slowdown)
  - $f \preceq h$ and $g \preceq h$ implies $f + g \preceq h$

- **Multiplication by a constant**
  - $f \preceq g$ implies $c \times f \preceq g$ (and of course $f \preceq c \times g$ holds by definition)
Although Big-Oh and Big-Omega are equivalent, a special need arises when our model $T(n)$ is quantified over all algorithms to solve the given problem.

- Example: consider the claim that any comparison based algorithm must make at least $c \times n \log n$ comparisons, for some constant, $c$. We say that comparison based sorting is lower bounded by $n \lg n$, i.e., that $T(n)$ is $\Omega(n \lg n)$ and we often reserve the $\Omega$ notation for this.

- Problems have lower bounds
  - A common lower bound is the size of the input itself (any algorithm to solve the problem must read all the inputs).
  - Sometimes we can prove better/tighter lower bounds (e.g., sorting above and searching is structured data (CS 420).
If $T(n)$ is $\Omega(f(n))$ and $T(n)$ is also $O(f(n))$ we have a tight bound, and we write that $T(n)$ is $\Theta(f(n))$.

It means that we have closed the problem, since the algorithm that we have attains the lower bound on the problem.
Closed and Open Problems

Sorting is a closed problem

- It has a lower bound of $n \log n$. We say that sorting is $\Omega(n \log n)$
- There are many sorting algorithms whose execution time is $O(n \log n)$ (see how we use big-Oh to talk about an algorithm)

Matrix multiplication is an open problem

- It is $\Omega(n^2)$.
  - The standard algorithm is $O(n^3)$
  - Another well known algorithm is $O(n^{2.376})$ and further improvements reduce the polynomial degree even further

See how the polynomial degree does not have to be integer
Plotting functions cleanly
In empirical CS (HPC, performance optimization, parallel programming) we plot functions describing the run time (or the memory use) of a program:

- This can be as a function of the input size (or other parameters like # of processors)

The functions are usually positive and monotonically increasing

We are interested in the asymptotic behavior, i.e.,

$$\lim_{n \to \infty} f(n)$$

How should we graph/plot them (e.g., lab report)?
The plot shows the ideal (expected) vs empirical (observed) values. Which one is ideal, and which is “just a bit off?”

- Series 1 (blue)
- Series 2 (orange)
Ungraded Quiz (survey)

- Same question, data is plotted differently.
  - Series 1 (blue)
  - Series 2 (orange)
Three functions: f, g and h

<table>
<thead>
<tr>
<th>n</th>
<th>f(n)</th>
<th>g(n)</th>
<th>h(n)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>9</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>12</td>
<td>18</td>
<td>6</td>
</tr>
<tr>
<td>3</td>
<td>36</td>
<td>35</td>
<td>24</td>
</tr>
<tr>
<td>4</td>
<td>80</td>
<td>68</td>
<td>68</td>
</tr>
<tr>
<td>5</td>
<td>150</td>
<td>131</td>
<td>162</td>
</tr>
</tbody>
</table>

- What class of functions are f, g, and h?
  - Polynomial? What degree?
  - Exponential? What base?
  - Impossible/hard to tell
The human visual system is very good at identifying linear (straight line) plots.

Everything else is approximate.

Asymptotically increasing functions just “swoosh up,” i.e., \( \lim_{n \to \infty} f(n) = \infty \)

Not enough range of data in second set of examples here just 1 ... 5)
Much better idea now about which function may be polynomial vs exponential? But still

- all is not clear (order, base ...)
- \(h(n)\) may spike up later...

<table>
<thead>
<tr>
<th>(n)</th>
<th>(f(n))</th>
<th>(g(n))</th>
<th>(h(n))</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>9</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>12</td>
<td>18</td>
<td>6</td>
</tr>
<tr>
<td>3</td>
<td>36</td>
<td>35</td>
<td>24</td>
</tr>
<tr>
<td>4</td>
<td>80</td>
<td>68</td>
<td>68</td>
</tr>
<tr>
<td>5</td>
<td>150</td>
<td>131</td>
<td>162</td>
</tr>
<tr>
<td>7</td>
<td>400</td>
<td>520</td>
<td>624</td>
</tr>
<tr>
<td>10</td>
<td>1100</td>
<td>4106</td>
<td>2510</td>
</tr>
<tr>
<td>12</td>
<td>1872</td>
<td>16396</td>
<td>5196</td>
</tr>
</tbody>
</table>
Straight lines

We get the most information from **straight lines**!

- We can easily recognize a straight line
  \[ y = ax + b \]
- The **slope** \((a)\) and **y intercept** \((b)\) tells us all.

- **How to “massage the data” into straight lines.**
- Change the scale to logarithmic: it turns a **multiplicative** factor into a **shift** (y axis crossing \(b\)) , and an **exponential** into a **multiplicative factor** (slope \(a\))
Four exponential functions

\[ y = 2^n \quad \log_{10}(y) = n \log_{10}2 \quad \text{linear in } n \]

\[ y = 3^n \quad \log_{10}(y) = n \log_{10}3 \]
the slope is the (log of the) base of the exponent

\[ y = 6 \times 3^n \quad \log_{10}(y) = n \log_{10}3 + \log_{10}6 \]
6 shifts up (in log scale)

\[ y = 3^n / 5 \quad \log_{10}(y) = n \log_{10}3 - \log_{10}5 \]
5 shifts down
Use a semi-log plot

<table>
<thead>
<tr>
<th>n</th>
<th>$2^n$</th>
<th>$3^n$</th>
<th>$20 \times 3^n$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>20</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>60</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>9</td>
<td>180</td>
</tr>
<tr>
<td>3</td>
<td>8</td>
<td>27</td>
<td>540</td>
</tr>
<tr>
<td>4</td>
<td>16</td>
<td>81</td>
<td>1620</td>
</tr>
<tr>
<td>5</td>
<td>32</td>
<td>243</td>
<td>4860</td>
</tr>
<tr>
<td>7</td>
<td>128</td>
<td>2087</td>
<td>41740</td>
</tr>
<tr>
<td>10</td>
<td>1024</td>
<td>56349</td>
<td>1126980</td>
</tr>
</tbody>
</table>

semi-log plot:
- y–axis on log scale
- x-axis linear

angle: base
shift: multiplicative factor
What about polynomials?

What is the logarithm of a polynomial (actually a monomial)?

- \( y = 5n^3 \)
- \( \log_{10}(y) = \log_{10}5 + \log_{10}n^3 = \log_{10}5 + 3\log_{10}n \)

Definitely not a straight line.

- But what about this?
- So we use a log-log scale/plot
Polynomial on log-log plot

<table>
<thead>
<tr>
<th>n</th>
<th>n^2</th>
<th>n^3</th>
<th>20*n^3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>20</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>8</td>
<td>160</td>
</tr>
<tr>
<td>4</td>
<td>16</td>
<td>64</td>
<td>1280</td>
</tr>
<tr>
<td>8</td>
<td>64</td>
<td>512</td>
<td>10240</td>
</tr>
<tr>
<td>16</td>
<td>256</td>
<td>4096</td>
<td>81820</td>
</tr>
<tr>
<td>32</td>
<td>1024</td>
<td>32768</td>
<td>655360</td>
</tr>
</tbody>
</table>

slope: degree
shift: multiplicative factor
Handling multiple terms

- Functions like $f(n) = 3^n + 4^n$ and polynomials that have more than one term. We don’t have a simple algebraic rule to compute logarithms of the sum of multiple terms.
  - Now, $f(n) = 3^n + 4^n = 4^n \left(1 + \left(\frac{3}{4}\right)^n\right)$
  - and since $\left(\frac{3}{4}\right) < 1$, so $\lim_{{n \to \infty}} \left(1 + \left(\frac{3}{4}\right)^n\right) = 1$
  - so, as $n \to \infty$, we have $\log f(n) \to \log 4^n \times 1 = \log 4 \times n$ i.e., only the dominant term matters

- For a polynomial like $f(n) = 4 \times n^3 + 3 \times n^2$ we do the same thing $f(n) = n^3 \left(4 + \frac{3}{4n}\right)$ and
  as $n \to \infty$, the term in parentheses approaches 4,
  so $\log f(n) \to \log 4 \times n^3 = \log 4 + 3 \log n$

- Message: when plotting your data, look for the trend among the points with larger input values
The semi-log plot does not give a straight line, so \( f \) is not exponential.
YES! The log-log plot is asymptotically a straight line, so $f$ is polynomial, but what is its leading term?

<table>
<thead>
<tr>
<th>$n$</th>
<th>$f(n)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>12</td>
</tr>
<tr>
<td>3</td>
<td>36</td>
</tr>
<tr>
<td>4</td>
<td>80</td>
</tr>
<tr>
<td>5</td>
<td>150</td>
</tr>
<tr>
<td>7</td>
<td>400</td>
</tr>
<tr>
<td>10</td>
<td>1100</td>
</tr>
<tr>
<td>12</td>
<td>1872</td>
</tr>
</tbody>
</table>
Continue empirically

<table>
<thead>
<tr>
<th>n</th>
<th>f(n)</th>
<th>n²</th>
<th>n³</th>
<th>n⁴</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>12</td>
<td>4</td>
<td>8</td>
<td>16</td>
</tr>
<tr>
<td>3</td>
<td>36</td>
<td>9</td>
<td>27</td>
<td>81</td>
</tr>
<tr>
<td>4</td>
<td>80</td>
<td>16</td>
<td>64</td>
<td>256</td>
</tr>
<tr>
<td>5</td>
<td>150</td>
<td>25</td>
<td>125</td>
<td>625</td>
</tr>
<tr>
<td>7</td>
<td>400</td>
<td>49</td>
<td>343</td>
<td>2401</td>
</tr>
<tr>
<td>10</td>
<td>1100</td>
<td>100</td>
<td>1000</td>
<td>10000</td>
</tr>
<tr>
<td>12</td>
<td>1872</td>
<td>144</td>
<td>1728</td>
<td>20736</td>
</tr>
</tbody>
</table>

Compare with n, n², n³, n⁴

It is degree 3, no multiplicative factor
All polynomial functions are members:

- \( f(n) \) is in the club iff \( f = \theta(n^k) \) for some constant, \( k \).

**Membership test:** to enter the club you scan your id

- checker is just a log-log plotter you’re in if it’s a straight line with slope between 0° to 90°

**Slowest** growing polynomial (**fastest** algorithms) are polynomials \( f(n) = n^\epsilon \) where, \( \epsilon \) is an arbitrarily small constant.

**Fastest** growing polynomial (**slowest** algorithms) \( f(n) = n^\Gamma \) where, \( \Gamma \) is an arbitrarily large constant.
The exponential club

- All exponential functions are members
- Membership test: to enter the club you scan your id
  - checker is just a semi-log plotter you’re in if it’s a straight line with slope between 0° to 90°
- Slowest growing exponential (fastest algorithms) are exponential $f(n) = \varepsilon^n$ where, $\varepsilon$ is an arbitrarily small constant.
- Fastest growing exponential (slowest algorithms) $f(n) = \Gamma^n$ where, $\Gamma$ is an arbitrarily large constant.
The basic mathematical definition of $\ll, \gg, \Theta$ and $\Omega$ still hold: for large enough $n$ one function exceeds the other.

The plotting trick is simply to compress the $x$ or $y$ axis plotting, and it doesn’t change asymptotic behavior.

What if we compress the $x$ axis and not the $y$ axis: a so-called log-semi plot (but this naming convention is soon going to prove inadequate).

- These are the poly-log functions: polynomials of $\log n$.
- The worst poly-log algorithm is faster the fastest polynomial algorithm $\log^\Gamma n \ll n^\varepsilon$.

Super-exponential functions: straight line when we plot $\log \log f(n)$ vs $n$. 
Each club conducts their internal tournaments, and ranks their members.

Algorithm designers try to invent new algorithms for open problems
  - When they give a new algorithm when the previous best was in the same club, they reduce the slope by a constant, and it’s a big accomplishment, e.g., going from $O(n^3)$ to $O(n^{2.7})$
    - even if that improvement comes at a “cost” of a factor that is equal the slowest member of a faster club
  - A new algorithm that’s in a faster club a major breakthrough.

Breakthroughs between the exponential and polynomial clubs are increasingly unlikely
First, we define what it means for a function to be strictly linear and asymptotically linear.

- A function $y = g(x)$ is said to be **strictly linear** if there are constants $m$ and $b$ such that $y = mx + b$

- A function $y = g(x)$ is said to be **asymptotically linear** if there exists a constant $m'$ such that $\lim_{x \to \infty} \frac{y}{x} = m'$ (we sometimes drop the adjective asymptotically)
What happens when you add two functions

- \( g(x) = f_1(x) + f_2(x) \)

If they are members of the same (e.g., polynomial) club, e.g., \( f_1(x) = x\sqrt{x}, f_2(x) = x^2 \)

- \( g(x) \) is a member of the club but is asymptotically (not strictly) linear on a log-log plot.
- We saw this with multiple terms (slide 31)

If they are members of different clubs, say, \( f_1(x) = 2^x \), and \( f_2(x) = x^2 \)

- \( g(x) \) is a member of the exponential club, and asymptotically linear on a semi-log plot
Additive Slowdown

Same Clubs

Different Clubs

$n$, $n^2$, $n^2+n$

$n^2$, $2^n$, $n^2+2^n$
What happens when you multiply two functions

\[ g(x) = f_1(x) f_2(x) \]

If they are members of the same (e.g., polynomial) club, e.g., \( f_1(x) = x \sqrt{x} \), \( f_2(x) = x^2 \)

- \( g(x) \) is a member of the club and is strictly linear on a log-log plot.
- But asymptotically linear if any of the functions has additive lower degree terms
- Slope of \( g(x) \) is the sum of the slopes of \( f_1(x) \) and \( f_2(x) \)

If they are members of different clubs, say, \( f_1(x) = 2^x \), and \( f_2(x) = x^2 \)

- \( g(x) \) is a not a member of the exponential club, and asymptotically linear on a semi-log plot

This is the effect of compression: some non-members are asymptotically linear
Multiplicative Slowdown

**Same Clubs**

- $n^2$
- $n^3$
- $n^2 \cdot n^3$

**Different Clubs**

- $n^{1+\varepsilon}$
- $\log n$
- $n$
- $n \log n$
Let $y = f(x)$ be an arbitrary (asymptotically monotonically increasing) function that represents the execution time of a program on an input of size $x$.

We can introduce four scaling variables to massage the input or output data.

- $y' = \log y$
- $y'' = \log \log y$
- $x' = \log x$
- $x'' = \log \log x$
Substituting the scaling variables yields the following nine massaging functions for the normal, log, and log log cases of each variable. This allows us to massage the input and output data for the different frames of reference in the graphs.

- $y = h_0(x)$ is linear for linear functions
- $y' = h_1(x)$ is linear for exponential functions
- $y'' = h_2(x)$ is linear for doubly exponential
- $y = h_3(x')$ is linear for logarithmic functions
- $y = h_4(x'')$
- $y' = h_5(x')$ is linear for polynomial functions
- $y' = h_6(x'')$ is linear for poly logarithmic functions
- $y'' = h_7(x')$
- $y'' = h_8(x'')$
Consider three clubs, $C_a$, $C_b$ and $C_c$, with membership defining functions, $h_a$, $h_b$, and $h_c$, where $C_a$ grows faster than $C_b$ and $C_c$ grows slower than $C_b$. Membership defining functions are the massaging functions corresponding to these clubs.

Consider a function $f(x) \in C_b$.

- $f(x)$ **swoops up** if it grows strictly faster than a membership defining function, i.e., even faster than a straight line with slope approaching $90^\circ$.
- $f(x)$ **swoops right** if it grows strictly slower than a membership defining function, i.e., even slower than a straight line with slope approaching $0^\circ$. 
Linear Club - $y = h_0(x)$
Polynomial Club – $y' = h_5(x')$
Exponential Club - $y' = h_1(x)$
Logarithmic Club – $y = h_3(x')$