Asymptotic analysis

A problem to solve

- Given a student ID, find the student's name
  - What operations can we perform?

retrieve a student record. ?
retrieve the next record ?
compare student ID
retrieve arbitrary record ?

  - How are the students organized (if at all)?

e.g list
• ordered vs unordered

Efficiency

- Complexity theory studies algorithm *efficiency*
  - Particularly, how well an algorithm *scales* as problem size increases

- For two algorithms that solve the same problem, we want to compare on some measure of efficiency, e.g.
  - Time (how long it takes to run) Time complexity
  - Space (how much memory is used while running)
  - Other attributes?
    - Expensive operations, e.g. I/O
    - Code elegance, tricks/shortcuts
    - Energy/power
    - Ease of programming, legal issues, etc.

Analysing runtime

```
...
old2 = 1;
old1 = 1;
for (i = 3; i < n; i++) {
  result = old2 + old1;
  old1 = old2;
  old2 = result;
}
```

- How long does this take?

It depends!

- What is n?
- What hardware?
- What programming language?
- What compiler?

Want a description that does not depend on so many factors

Analysing number of operations

- Focusing on only one complexity measure – number of operations performed by the algorithm on an input of given size, e.g.
  - # instructions executed
  - # comparisons

- Some operations are more costly than others, but as a rough indicator, counting operations is good enough

Analysing runtime

```
...
old2 = 1;
old1 = 1;
for (i = 3; i < n; i++) {
  result = old2 + old1;
  old1 = old2;
  old2 = result;
}
```

- How many operations does this take?

It depends!

- What is n?

- Running time is a function of n such as $T(n)$
- Runtime analysis in this way no longer depends on hardware or subjective conditions

Input size

integer

- What is meant by the input size n? Some application-specific examples:
  - Dictionary: # of words
  - Restaurant: # of customers, # of menu choices, # of employees etc.
  - Airline: # of flights, # of customers, # of luggage etc.

- Find a way to express the number of operations performed as a function of the input size n

Back to comparing algorithms

and scalability

- Suppose we have two different algorithms
  - up to $n = 200$, algorithm A is faster
  - beyond $n = 200$, algorithm B is faster
  - which one is really faster?
- Computer science emphasises studying big versions of problems
  - i.e. when the input size scales up to a very large number
- But we still want to have a simple, *approximate* way to make comparisons between the behaviours of different algorithms' rates of growth
  - use a simple, well-understood function as a reference

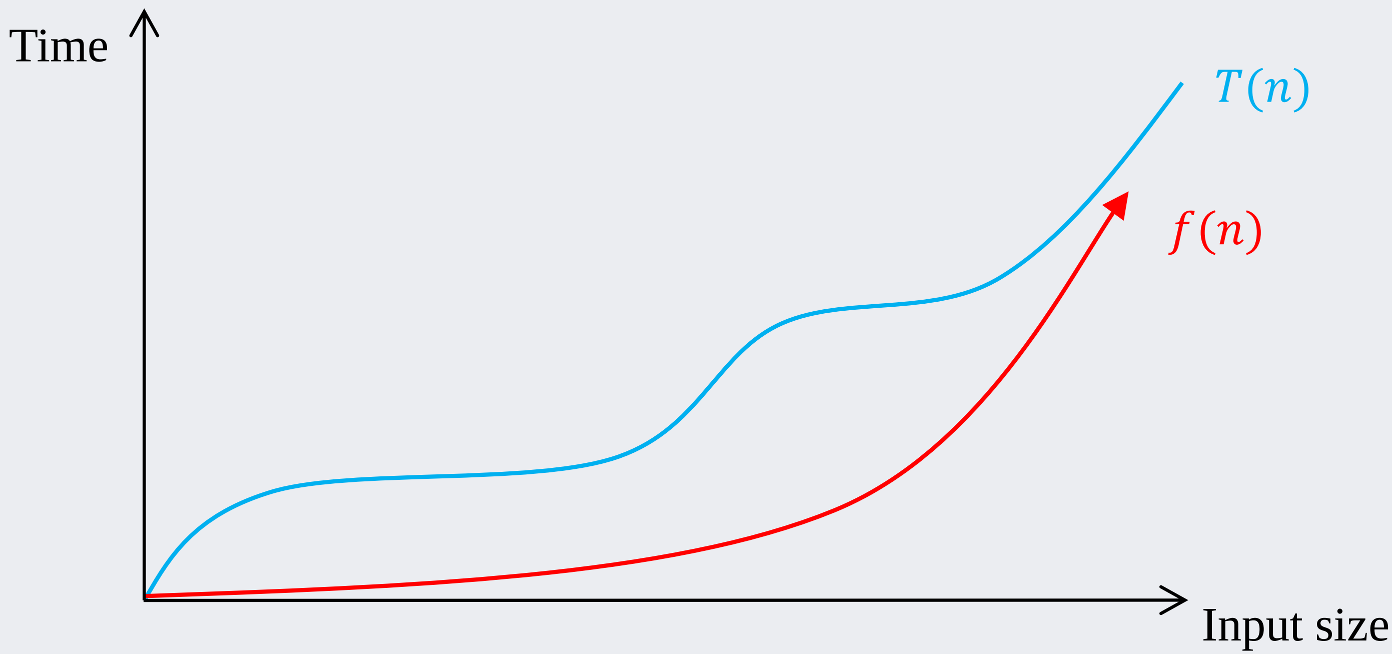
Order notation

our algorithm reference function T(n) is in big-O of f(n)

- Let $T(n)$ and $f(n)$ be functions mapping $\mathbb{Z}^+ \to \mathbb{R}^+$
- $T(n) \in O\big(f(n)\big)$ if there are constants c and n_0 such that $T(n) \leq c \cdot f(n)$ for all $n \geq n_0$

$c \in \mathbb{R}^+$ $n_0 \in \mathbb{N}$

We want to compare the "overall" runtime (or memory usage, etc.) of our function against a familiar, simple function



O-notation, visually

our function reference function scaling factor > 0 input size threshold

- $T(n) \in O(f(n))$ if there are constants c and n_0 such that $T(n) \leq c \cdot f(n)$ for all $n \geq n_0$

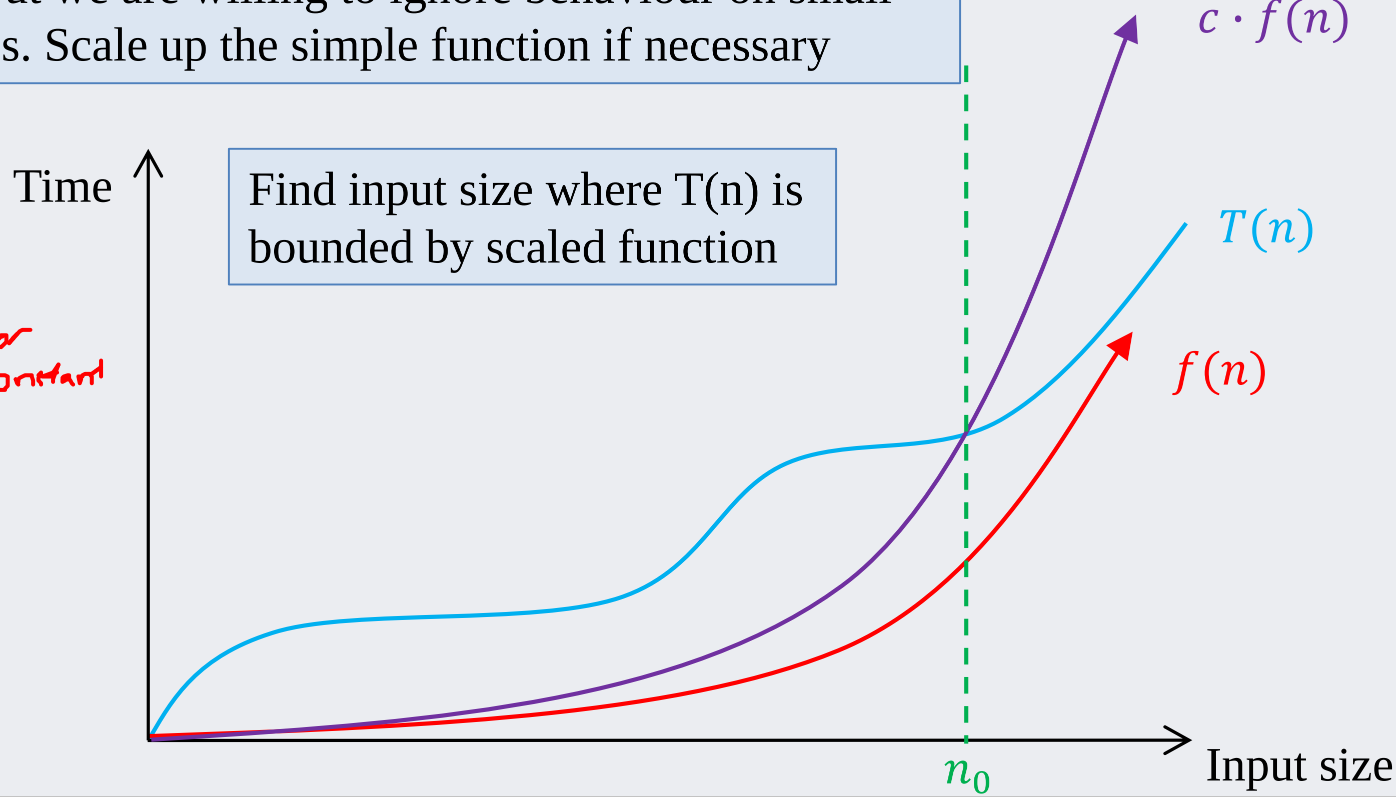
"our function $\leq$ scaled reference function"

for all input sizes larger than the threshold

We want comparison to be valid for all sufficiently large inputs, but we are willing to ignore behaviour on small examples. Scale up the simple function if necessary

$\lim_{n \to \infty} \frac{f(n)}{T(n)} = 0$ or constant

Find input size where T(n) is bounded by scaled function



Why do we bother?

- Suppose a computer executes 10^{12} operations per second

$\boldsymbol{n}$ / $\cancel{T} f(\boldsymbol{n})$	10	100	1000	10,000	10^{12}
n	10^{-11} s	10^{-10} s	10^{-9} s	10^{-8} s	1 s
$n \log n$	10^{-11} s	10^{-9} s	10^{-8} s	10^{-7} s	40 s
n^2	10^{-10} s	10^{-8} s	10^{-6} s	10^{-4} s	10^{12} s
n^3	10^{-9} s	10^{-6} s	10^{-3} s	1 s	10^{24} s
2^n	10^{-9} s	10^{18} s	10^{289} s		

- For reference:
  - 10^4 s = 2.8 hours, 10^{18} s ≈ 30 billion years

Order notation

big · O

- $T(n) \in O\big(f(n)\big)$ if there are constants c and n_0 such that $T(n) \leq c \cdot f(n)$ for all $n \geq n_0$ f(n) is upper bound on T(n)
  - $T(n)$ is bounded from above by $c \cdot f(n)$
  - i.e. the growth of $T(n)$ is no faster than $f(n)$

big · omega

- $T(n) \in \Omega\big(f(n)\big)$ if $f(n) \in O\big(T(n)\big)$ f(n) is a lower bound
  - $T(n)$ is bounded from below by $d \cdot f(n)$ Ω ←
  - i.e. $T(n)$ grows no slower than $f(n)$

big · Theta

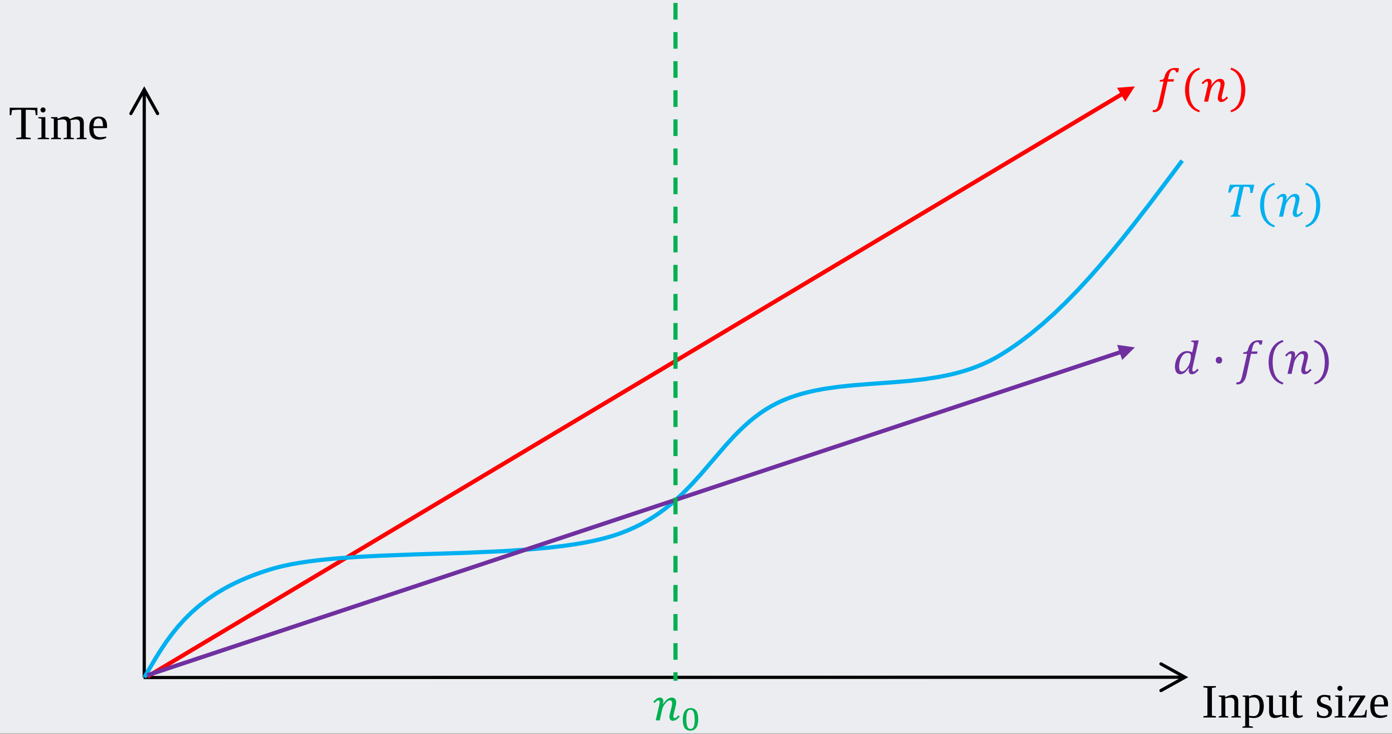
- $T(n) \in \Theta\big(f(n)\big)$ if $T(n) \in O\big(f(n)\big)$ and $T(n) \in \Omega\big(f(n)\big)$
  - $T(n)$ is bounded from above and below by $f(n)$ f(n) is a tight bound
  - i.e. $T(n)$ grows at the same rate as $f(n)$ Θ ←

Ω-notation, visually

there exist   for all

- $T(n) \in \Omega\big(f(n)\big)$ if $\exists\ d, n_0$ such that $T(n) \geq d \cdot f(n)\ \forall\ n \geq n_0$

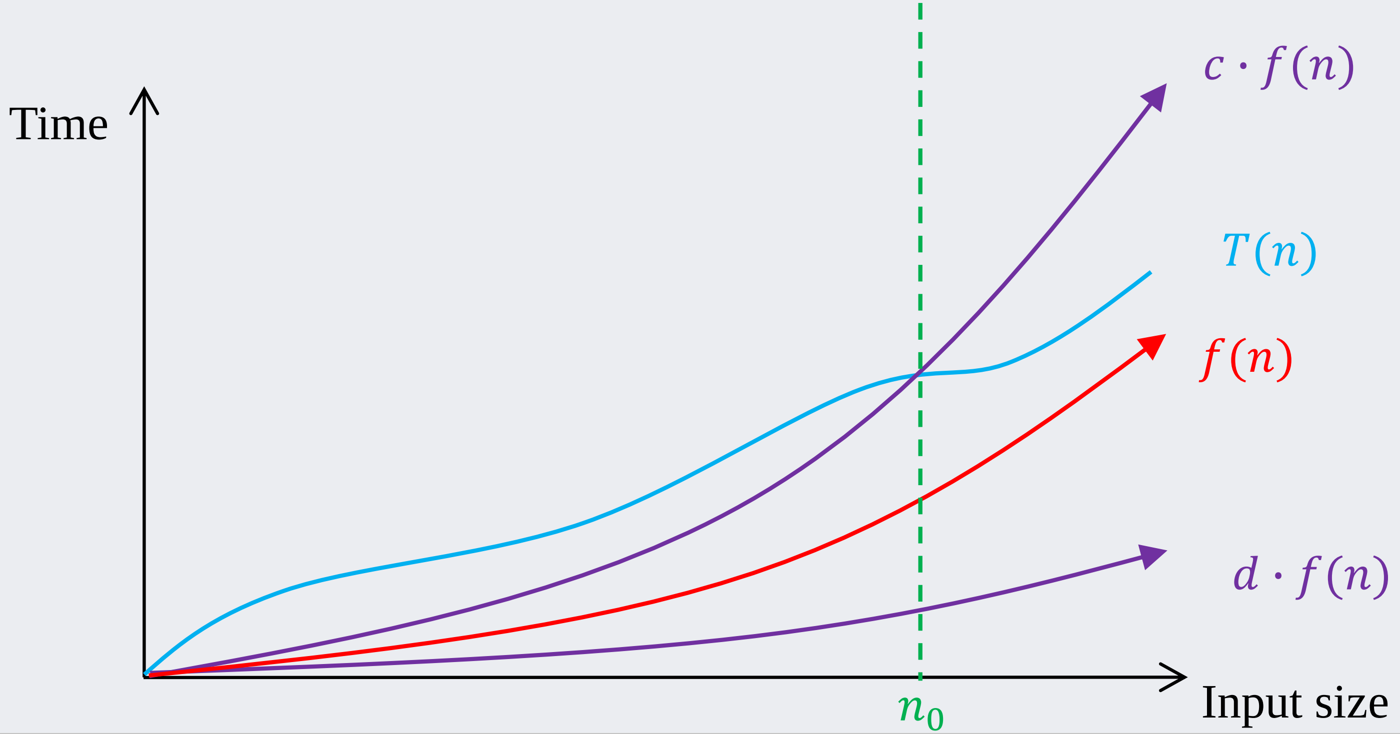
Given the same sort of adjustments,
$T(n)$ should be ***greater*** than $d \cdot f(n)$



Θ-notation, visually

- $T(n) \in \Theta(f(n))$ if $\exists\ c, d, n_0$ such that $d \cdot f(n) \leq T(n) \leq c \cdot f(n)\ \forall\ n \geq n_0$

Given the same sort of adjustments, $T(n)$ should be ***between*** the adjusted versions of $f(n)$



Asymptotic Analysis Hacks

Running time approximation

- Eliminate low order terms
  - $4n + 5 \implies 4n$
  - $0.5n \log n - 2n + 7 \implies 0.5n \log n$
  - $2^n + n^3 + 3n \implies 2^n$

- Eliminate constant coefficients
  - $4n \implies n$
  - $0.5n \log n \implies n \log n$
  - $n \log(n^2) = 2n \log n \implies n \log n$

constant

$\log 2n = \cancel{\log 2} + \log n$

Examples

- $10{,}000n^2 + 25n \in \Theta(n^2)$
- $10^{-10}n^2 \in \Theta(n^2)$
- $n\log n \in O(n^2)$ ✓ $n \log n \notin \Omega(n^2)$
- $n\log n \in \Omega(n)$ ✓ $n \log n \notin O(n)$
- $n^3 + 4 \in O(n^4)$, but not $\Theta(n^4)$
- $n^3 + 4 \in \Omega(n^2)$, but not $\Theta(n^2)$

Common growth rate functions

$\log_a b = \frac{\log_c b}{\log_c a} = \boxed{\frac{1}{\log_c a}} \cdot \log_c b$

a is a constant　　c is any constant　　constant

- Typical growth rates in order
  - Constant: $O(1)$
  - Logarithmic: $O(\log n)$　　$\left(\log_k n, \log(n^2) \in O(\log n)\right)$
  - Poly-log: $O\left((\log n)^k\right)$
  - Sublinear $O(n^c)$ $0<c<1$
  - Linear: $O(n)$
  - Log-linear: $O(n \log n)$ $\in \Omega(\quad)$ $\in O(\quad)$
  - Superlinear: $O(n^{1+c})$　　(c is a constant, $0 < c < 1$)
  - Quadratic: $O(n^2)$
  - Cubic: $O(n^3)$
  - Polynomial $O(n^k)$　　(k is a constant)　　"tractable"
  - Exponential $O(c^n)$　　(c is a constant > 0)　"intractable"

- Factorial $O(n!)$

~~exp~~ vs factorial

$\underbrace{2 \cdot 2 \cdot 2 \cdot 2 \cdot}_{n}$　　$\underbrace{1 \cdot 2 \cdot 3 \cdot 4 \cdot \ldots \cdot n}_{n}$

Dominance

- We can look at the dominant term to guess at a big-O growth rate. e.g.
  - $T(n) = 2n^2 + 600n + 60000$
    - Up to $n = 100$, the constant term dominates
    - Between $n = 100$ and $n = 300$, the linear term dominates
    - Beyond $n = 300$, the quadratic term dominates, $T(n) \in O(n^2)$

- Which will be faster in the long run? n^3 vs $n^3 \log n$?
  - split up and use dominance relationships

$\underbrace{n^3}\cdot 1$ vs $\underbrace{n^3}\cdot \log n$

- n^3 vs $n^{3.01}/\log n$?

$\underline{n^3}\cdot 1$ vs $\underline{n^3}\cdot \underline{n^{0.01}} / \log n$

What does order notation tell us?

- Note that the definitions of O and Ω use inequality, thus the statements for a function $f(n) = 3n \log_2 n$:
  - $f(n) \in O(n \log n)$ and
  - $f(n) \in O(2^n)$
  - are both true

- However, one is more meaningful than the other
  - "Our function $f(n)$ has growth behaviour no worse than this other pretty well-behaved function", vs
  - "Our function $f(n)$ has growth behaviour no worse than one of the worst functions known"

- We aim to obtain the "tightest" upper or lower bounding function that still satisfies the O/Ω relation

Asymptotic analysis proofs

- Use the definitions of O and/or Ω to determine either a witness pair (c, n_0) satisfying the definition, or show that no such witness pair is possible

- Example: Prove that for $f(n) = 2\log_6 n$ and $g(n) = 3n$, $f(n) \in O\big(g(n)\big)$

There are constants $c > 0$ and $n_0 > 0$ such that $2\log_6 n \leq c \cdot 3n$ for all $n \geq n_0$

Choose $c = 1$, $n_0 = 6$, it can be seen that LHS $\leq$ RHS and remains so as n increases.

O notation proofs

Prove $f(n) = 2 \cdot \log_6 n$ $g(n) = 3 \cdot n$, $f(n) \in O(g(n))$

$$2 \cdot \log_6 n \le c \cdot 3 \cdot n \text{ , for all } n \ge n_0$$

$$2 \cdot \log_6 n \le 2 \cdot \log_6 n$$

$$f(n) = 2 \cdot \log_6 n \le 2 \cdot n \qquad n \ge 6$$

$$\le c \cdot \underbrace{3 \cdot n}_{g(n)} \qquad n \ge 6, \; c = \frac{2}{3}$$

$$f(n) \in O(g(n)) \qquad c = \frac{2}{3}, \; n_0 = 6$$

O notation proofs

$f(n) = 2n^3 + 4n + 6$ $\qquad g(n) = 3n^4$

prove $f(n) \in O(g(n))$

$$
\begin{aligned}
f(n) = 2n^3 + 4n + 6 &\leq 2n^3 + 4n^3 + 6n^3 \quad n \geq 1 \\
&\leq 2n^4 + 4n^4 + 6n^4 \quad n \geq 1 \\
&\leq 12n^4 \\
&\leq 4 \cdot \underbrace{3 \cdot n^4}_{g(n)}
\end{aligned}
$$

$c = 4 \quad , \quad n_0 = 1$

prove $3n \notin O(2^{\log_6 n})$

wrong approach: $3n \nleq c \cdot 2^{\log_6 n}$ for all $n \geq n_0$

Let $c = 1$ and $n_0 = 6$

clearly LHS $\geq$ RHS

we have shown that one specific (c, n_0) pair doesn't work.

we need to show that no pair can possibly work

Asymptotic analysis proofs

- Example: Prove that for $f(n) = 2\log_6 n$ and $g(n) = 3n$, $g(n) \notin O\big(f(n)\big)$

Assume for the purpose of a contradiction, that $g(n) \in O\big(f(n)\big)$

Then, there are constants $c > 0$ and $n_0 > 0$ such that $3\text{n} \le c \cdot 2\log_6 n$ for all $n \ge n_0$

Solving the inequality for c, we obtain $c \ge \frac{3n}{2\log_6 n}$

by L'Hôpital's rule

However, as n increases, the value of $\frac{3n}{2\log_6 n}$ increases, and there is no such constant c which can remain at least as large this increasing value – contradicting our initial assumption

Therefore $g(n) \notin O\big(f(n)\big)$

Input size

- We have described the number of operations as a function of a given input size n
  - But, how are the n items organised?
  - e.g., to find my favourite riding boots in my closet

Types of analysis

- Bound flavour
  - Upper bound (O)
  - Lower bound (Ω), useful for *problems*
  - Asymptotically tight (Θ)
- Analysis case
  - Best case (lucky)
  - Worst case (adversary)
  - Average case
  - "common" case
- Analysis quality
  - Loose bound (any true analysis)
  - Tight bound (no better "meaningful" bound that is asymptotically different)

Rare, mostly useless

Useful, pessimistic

Useful, tricky to determine

Useful, poorly defined

Analysing code

```
int find(int key, int arr[], int n) {
  int i;
  for (i = 0; i < n; i++) {
    if (arr[i] == key)
      return i;
  }
  return -1;
}
```

- Step 1: What is the input size n?
- Step 2: What kind of analysis should we perform?
  - Worst case? Best case? Average case?
- Step 3: How much does each line cost?
  - (are lines even the correct unit?)

Analysing code

```
int find(int key, int arr[], int n) {
  int i;
  for (i = 0; i < n; i++) {
    if (arr[i] == key)
      return i;
  }
  return -1;
}
```

- Step 4: What is $T(n)$ in its raw form?
- Step 5: Simplify $T(n)$ and convert to order notation
  - Also, which notation? O, Θ, Ω
- Step 6: Prove the asymptotic bound by finding constants c and n_0 satisfying the required inequality(ies)

Example 1

```
for i = 1 to n do
  for j = 1 to n do
    sum = sum + 1
```

$$\left.\begin{array}{l}1\\ \left.\begin{array}{l}1\\1\end{array}\right\} n \text{ times}\end{array}\right\} n \text{ times}$$

- A straightforward example in pseudocode
- Each loop runs n times, and a constant amount of work is done inside each loop
  - might be different absolute amounts of work, but still constant

$$T(n)=\sum_{i=1}^{n}\overbrace{\left(1+\underbrace{\sum_{j=1}^{n}2}_{\text{cost of inner loop}}\right)}^{\text{single outer loop execution}}=\sum_{i=1}^{n}(1+2n)=n+2n^{2}=O\left(n^{2}\right)$$

Example 1

Simpler version

```
for i = 1 to n do
  for j = 1 to n do
    sum = sum + 1
```

1
1 } n times
1 } n times

- Count the number of times `sum = sum + 1` is executed

cost of sum = sum+1

$$T(n) = \sum_{i=1}^{n} \sum_{j=1}^{n} 1 = \sum_{i=1}^{n} n = n^2 = O(n^2)$$

same bound as before

Example 2

```
i = 1
while i < n do
  for j = i to n do
    sum = sum + 1
  i++
```

n^2

$\frac{1}{2} \cdot n^2$

- Time complexity:
  - a) $\Theta(n)$
  - b) $\Theta(n \log n)$
  - c) $\Theta(n^2)$
  - d) $\Theta(n^2 \log n)$
  - e) None of these

Example 2

Pure math approach

```
i = 1
while i < n do
  for j = i to n do
    sum = sum + 1
  i++
```

"1" operation

`i` varies from `1` to `n-1`

`j` varies from `i` to `n`

"1" operation

"1" operation

$i=6 \quad n=10$

$6 \quad 7 \quad 8 \quad 9 \quad 10$

$n-i+1$

$1+(1+2+3\cdots n-1)$

$$T(n) = 1 + \sum_{i=1}^{n-1}\left(1 + \sum_{j=i}^{n} 1\right)$$

$$= 1 + \sum_{i=1}^{n-1}\left(1 + \underbrace{n - i + 1}\right) = 1 + \sum_{i=1}^{n-1}(n - i + 2)$$

Example 2

Pure math approach

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

$\sum (a+b) = \sum a + \sum b$

$\sum_{i=1}^{n-1} i = \frac{n(n-1)}{2}$

$$T(n) = 1 + \sum_{i=1}^{n-1} (n - i + 2) = 1 + \sum_{i=1}^{n-1} (n+2) - \sum_{i=1}^{n-1} i$$

$$= 1 + (n-1)(n+2) - \sum_{i=1}^{n-1} i = 1 + n^2 + n - 2 - \frac{n(n-1)}{2}$$

$$= \frac{n^2}{2} + \frac{3n}{2} - 1$$

$$T(n) \in \Theta(n^2)$$

$\sum_{i=1}^{n} i = \underbrace{1 + 2 + 3 + \ldots + (n-1)}_{\sum_{i=1}^{n-1}} + n = \frac{n(n+1)}{2}$

$\sum_{i=1}^{n-1} + n = \frac{n(n+1)}{2}$

$\sum_{i=1}^{n-1} = \frac{n(n+1)}{2} - n$

$\frac{n^2 + n}{2} - \frac{2n}{2} = \frac{n^2 - n}{2} = \frac{n(n-1)}{2}$

Example 2

Simplified math approach

```
i = 1
while i < n do
  for j = i to n do
    sum = sum + 1
  i++
```

Count only executions of this line

$$T(n) = \sum_{i=1}^{n-1} \boxed{\sum_{j=i}^{n} 1}$$

$$\sum_{i=1}^{n-1}(n+1) - \sum_{i=1}^{n-1} i$$

$$= \sum_{i=1}^{n-1}(n-i+1) = (n-1)(n+1) - \sum_{i=1}^{n-1} i$$

$$= n^2 - 1 - \frac{n(n-1)}{2} = \frac{n^2}{2} + \frac{n}{2} - 1$$

$$T(n) \in \Theta(n^2)$$

Example 3

```
for (i = 1; i <= n; i++)
  for (j = 1; j <= n; j = j*2)
    sum = sum + 1
```

O(n)

O(log n)

$$T(n)=\sum_{i=1}^{n}\sum_{j=1}^{?}1$$

$$j = 1,2,4,\ldots,x$$

$$= 2^0, 2^1, 2^2, \ldots, 2^k$$

$k = \lfloor \log_2 n \rfloor$ ($\log_2 n$)

$$T(n) = \sum_{i=1}^{n}\sum_{k=0}^{\lfloor \log_2 n \rfloor} 1 \leq \sum_{i=1}^{n} \log_2 n = (n \cancel{+1}) \log_2 n$$

$$T(n) \in O(n \log n)$$

base does not matter (asymptotically) $\log_a b = \frac{\log_c b}{\log_c a}$

A visual aid for loop executions

min value, max value.

- Determine the range of your loop variable
- Determine how many elements within that range will be "hit"

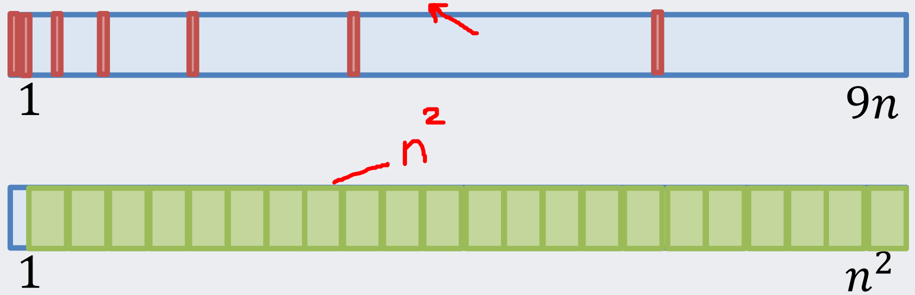
$\log(a \cdot b) = \log a + \log b$

- Complexities of nested loops are (usually) multiplied
- Complexities of separate loops are (usually) added

```
int i, j;
for (i = 1; i < 9*n; i = i*2) {
  for (j = n*n; j > 0; j--) {
    ...
  }
}
```

$\log_2(9n) =$

n^2



overall: $O(n^2 \log n)$

- Take extra care when an inner loop condition depends on the outer loop variable!

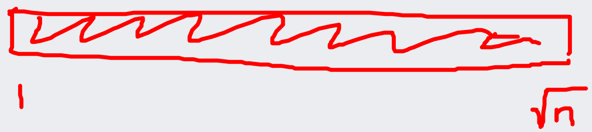
Readings for this lesson

- Thareja
  - Chapter 2

$\texttt{for(int j = 1; } \underbrace{j * j < n} \texttt{; j++)}$

$\sqrt{j^2} < \sqrt{n}$

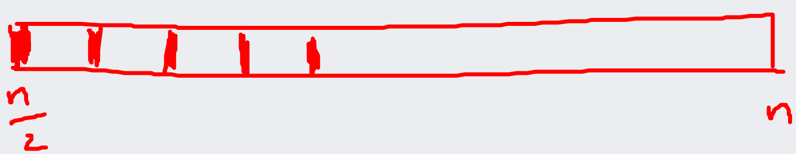
$j < \sqrt{n}$



$O(\sqrt{n})$

$i *= c$
$i /= c$ $\Big\}$ $\log_c(\text{range})$

$\texttt{for(int i = } n/2 \texttt{; i <= n; i += 5)}$



$\frac{1}{5} \cdot \frac{n}{2} = \frac{n}{10}$ $O(n)$

$i += c$
$i -= c$ $\Big\}$ $\frac{1}{c}(\text{range})$