CS170: Discrete Methods in Computer Science
Spring 2025
Runtime and Order Notation

Instructor: Shaddin Dughmi[1]

[1] These slides adapt some content from similar slides by Aaron Cote.

Outline

1. Quantifying Runtimes
2. Order Notation (Big-O and friends)
3. Comparing Runtimes

Comparing Algorithms

- An algorithm takes an input and produces an output
  - E.g. Takes in an unsorted array, and sorts it
- There are often different algorithms for the same task
  - Bubble sort vs mergesort vs quicksort vs insertion sort ...
- How to compare them?
  - Runtime
  - Memory
  - Simplicity
  - Communication bandwidth
  - ...

Comparing Algorithms

- An algorithm takes an input and produces an output
  - E.g. Takes in an unsorted array, and sorts it
- There are often different algorithms for the same task
  - Bubble sort vs mergesort vs quicksort vs insertion sort . . .
- How to compare them?
  - Runtime
  - Memory
  - Simplicity
  - Communication bandwidth
  - . . .

Measuring Runtime

How should we measure runtime?

- Time on the clock: Depends on details of underlying architecture, number of processors, whether you upgrade your machine, etc.
- Number of basic operations: Number of basic instructions in the programming language or machine model

Measuring Runtime

How should we measure runtime?

- Time on the clock: Depends on details of underlying architecture, number of processors, whether you upgrade your machine, etc.
- Number of basic operations: Number of basic instructions in the programming language or machine model

We go with number of operations:

- Different instruction sets / programming languages tend to be effectively equivalent here (more on this later).

Runtime Functions

- We consider worst-case runtime over inputs of any given size
  - E.g. for arrays of size n, we judge our algorithm by the most time-consuming array to sort

Runtime Functions

- We consider worst-case runtime over inputs of any given size
  - E.g. for arrays of size n, we judge our algorithm by the most time-consuming array to sort

Runtime Function

Given an algorithm, its (worst-case) runtime function is $f : \mathbb{N} \to \mathbb{N}$ where $f(n)$ is the maximum, over all inputs of size n, of the number of operations of the algorithm on that input.

Usually, and for all our purposes in this class, f is non-decreasing.

Runtime Functions

- We consider worst-case runtime over inputs of any given size
  - E.g. for arrays of size n, we judge our algorithm by the most time-consuming array to sort

Runtime Function

Given an algorithm, its (worst-case) runtime function is $f : \mathbb{N} \to \mathbb{N}$ where $f(n)$ is the maximum, over all inputs of size n, of the number of operations of the algorithm on that input.

Usually, and for all our purposes in this class, f is non-decreasing.
But what is the "size" of an input?

- In the strictest sense, it is the number of bits used to write that input down
- Sometimes, we cut corners and quantify size differently
  - E.g. By the length of the array in sorting
- So long as you're clear about what your n "means", you can choose the measure of size that best suits your problem.

Worst-Case vs Average Case

In CS, it is most common to consider worst-case runtime, instead of "average case". Why?

- Gives iron-clad gaurantees that always hold regardless of real-world setting
- Tends to be predictive in practice
- No need to make assumptions on real-world inputs, which often are hard to formulate.
    - What is "average case" array, social network, image?
- Gives rise to elegant theory that has had practical impact

Worst-Case vs Average Case

In CS, it is most common to consider worst-case runtime, instead of "average case". Why?

- Gives iron-clad gaurantees that always hold regardless of real-world setting
- Tends to be predictive in practice
- No need to make assumptions on real-world inputs, which often are hard to formulate.
  - What is "average case" array, social network, image?
- Gives rise to elegant theory that has had practical impact

Nevertheless, sometimes average case, or something between average and worst case, makes sense. We won't get into that in this class.

Common Examples of Runtimes

- Constant: 3, 5, 134893430
- Linear: n, $2n + 1$, $100n + 3$, ...
- Quadratic: n^2, $3n^2 + 1000n - 1$, ...
- Polynomial: $2n^5 + n^3 - n + 2$, ...
- Logarithmic: $\log n$, $5 \log n \log \log n + 3$, ...
- Exponential: 2^n, $3 \cdot 5^n + n^2$, ...
- ...

Granularity of Runtimes

At what granularity do we want to quantify runtime?

- Capture aspects of runtime that persist as we tweak architecture, basic instructions, increase number of cores,
  - Ignore constant multiples. n^2 and $5n^2$ should be "effectively the same"
- Judge runtime "as input grows large"
  - Ignore vanishing terms. n^2 and $n^2 + n + 7$ should be "effectively the same",

Granularity of Runtimes

At what granularity do we want to quantify runtime?

- Capture aspects of runtime that persist as we tweak architecture, basic instructions, increase number of cores,
  - Ignore constant multiples. n^2 and $5n^2$ should be "effectively the same"
- Judge runtime "as input grows large"
  - Ignore vanishing terms. n^2 and $n^2 + n + 7$ should be "effectively the same",

This is what Order Notation does (Next)

Outline

Big-O

Big-O

For two functions $f, g : \mathbb{N} \to \mathbb{N}$, we say $f(n) = O(g(n))$ if there are constants n_0 and c such that $f(n) \leq cg(n)$ for all $n \geq n_0$.

In other words, $f(n)$ eventually less than $g(n)$, if you don't care about constants. We refer to this as asymptotic order of growth.

Big-O

Big-O

For two functions $f, g : \mathbb{N} \to \mathbb{N}$, we say $f(n) = O(g(n))$ if there are constants n_0 and c such that $f(n) \leq cg(n)$ for all $n \geq n_0$.

In other words, $f(n)$ eventually less than $g(n)$, if you don't care about constants. We refer to this as asymptotic order of growth.

Another Definition of big-O

$f(n) = O(g(n))$ if $\lim_{n \to \infty} \frac{f(n)}{g(n)} < \infty$.

Equivalent when the limit exists, which is the case most of the time.

Big-O

Big-O

For two functions $f, g : \mathbb{N} \to \mathbb{N}$, we say $f(n) = O(g(n))$ if there are constants n_0 and c such that $f(n) \leq cg(n)$ for all $n \geq n_0$.

In other words, $f(n)$ eventually less than $g(n)$, if you don't care about constants. We refer to this as asymptotic order of growth.

Another Definition of big-O

$f(n) = O(g(n))$ if $\lim_{n\to\infty} \frac{f(n)}{g(n)} < \infty$.

Equivalent when the limit exists, which is the case most of the time.

This is abuse of the $=$ symbol

If $f = O(g)$, we can't say $O(g) = f$. Really, it should be $f \in O(g)$, where $O(g)$ is the class of functions that asymptotically grow no faster than g, but this abuse of notation is with us for historical reasons.

Examples

- $10n^3 = O(n^3)$
- $10000n = O(n^2)$
- $\log n = O(n)$
- $10000n^{100} = O(2^n)$
- $\ldots$

Friends of Big-O

- $f(n) = \Omega(g(n))$: $\exists c, n_0 \; \forall n \geq n_0 \;\; f(n) \geq cg(n)$
  - Equivalent to $g(n) = O(f(n))$.
- $f(n) = \Theta(g(n))$: Both $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$
  - f and g are within a constant of each other for large enough n.
- $f(n) = o(g(n))$: $\forall c > 0 \; \exists n_0 \; \forall n \geq n_0 \;\; f(n) < cg(n)$
  - When limit ratio exists: Same as $\lim_{n \to \infty} \frac{f(n)}{g(n)} = 0$. Also same as $f(n) = O(g(n)$ but not $f(n) = \Omega(g(n))$.
- $f(n) = \omega(g(n))$: $\forall c > 0 \; \exists n_0 \; \forall n \geq n_0 \;\; f(n) > cg(n)$
  - Equivalent to $g(n) = o(f(n))$.
  - When limit ratio exists: Same as $\lim_{n \to \infty} \frac{f(n)}{g(n)} = \infty$. Also same as $f(n) = \Omega(g(n)$ but not $f(n) = O(g(n))$.

Friends of Big-O

- $f(n) = \Omega(g(n))$: $\exists c, n_0 \; \forall n \geq n_0 \;\; f(n) \geq cg(n)$
  - Equivalent to $g(n) = O(f(n))$.
- $f(n) = \Theta(g(n))$: Both $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$
  - f and g are within a constant of each other for large enough n.
- $f(n) = o(g(n))$: $\forall c > 0 \; \exists n_0 \; \forall n \geq n_0 \;\; f(n) < cg(n)$
  - When limit ratio exists: Same as $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$. Also same as $f(n) = O(g(n)$ but not $f(n) = \Omega(g(n))$.
- $f(n) = \omega(g(n))$: $\forall c > 0 \; \exists n_0 \; \forall n \geq n_0 \;\; f(n) > cg(n)$
  - Equivalent to $g(n) = o(f(n))$.
  - When limit ratio exists: Same as $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$. Also same as $f(n) = \Omega(g(n)$ but not $f(n) = O(g(n))$.

Think of $O, \Omega, \Theta, o, \omega$ as $\leq, \geq, =, <, >$ respectively for comparing order of growth.

Outline

Exercise

Compare $\log n$ and $\sqrt{n}$. (Hint: Use L'Hopital's rule)

Common Rules of Thumb

- Constants are best
- Then logs and polylogs
- Then polynomials
- Then exponentials

These are the most common, but there is other stuff between them, and also beyond exponentials.

Exercise

Order the following runtimes

- n^n
- $\log^2 n$
- $n^{1.01}$
- 1.01^n
- $2^{\sqrt{\log n}}$
- $n \log^{1000} n$