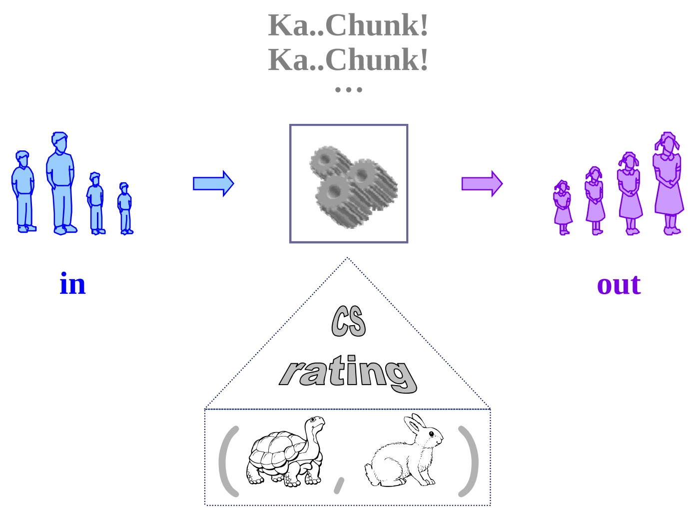
Analysis of Algorithms



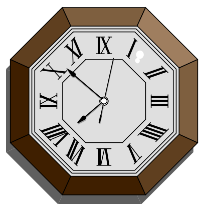
Which of the Many Ways (to a Common End)?

- Computer science → problem solving → algorithms
- Typically → many algorithms are available for solving a given problem
  - Which algorithm should we use?
- Among important criteria for choice of algorithm
  - *Time* efficiency
  - *Space* efficiency
  - Relative *complexity* ("*human role playing*" efficiency)
- Our discussion will focus on *time* efficiency
  - Running time analysis
    - *Best* case, *worst* case and *average* case
    - Order (of magnitude) analysis → in terms of *worst* case
  - (Similar analysis can be applied to *space* efficiency)

Running Time Analysis

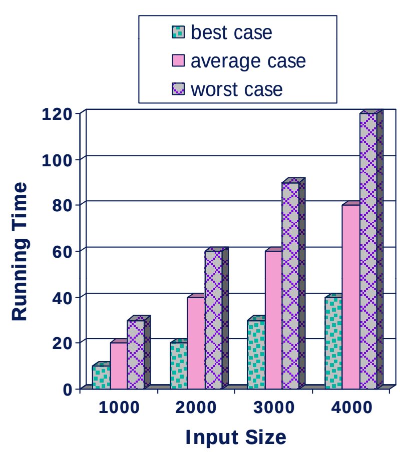
Running time analysis → characterizing algorithms' performance (running time requirements)

- Is an algorithm good (fast) enough to be practical?
- How much longer will algorithm take as input gets larger?
- Which of several different algorithms is best (fastest)?



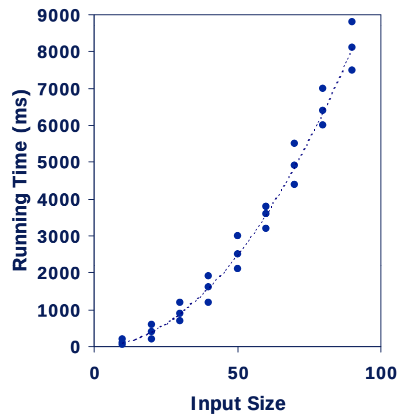
Running Time

- Running time of an algorithm varies with input and typically *grows with input size*
- Running time for *fixed input size* typically also varies depending on *composition (nature) of input*
  - *Best*, *worst* and *average* cases
  - E.g.: check if a value is in a set of values (stored in an unsorted array) using *sequential search*
  - Average case → not easy to determine
- *Worst case* running time usually used in running time analysis
  - Easier to determine
  - Typically leads to better algorithms



Determine Running Time Experimentally

- Write program implementing the algorithm
- Run program with inputs of varying size and composition
- Use a function (**`clock()`**, say) to get accurate measure of actual running time
- Plot results and perform statistical analysis to fit best curve (function) to experimental data



Limitations of Experimental Approach

- Need to implement algorithm
  - May be difficult
- Results may not represent running time on other inputs
  - Inputs not included in experiment
- Same hardware and software environments must be used
  - In order to meaningfully compare different algorithms

Desirable, More General Analysis Approach

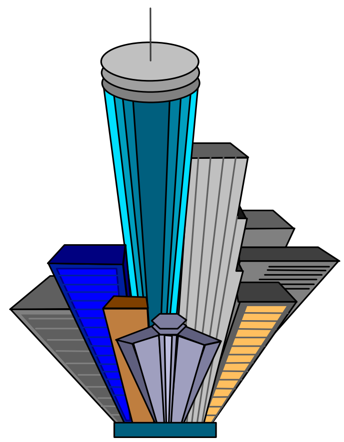
- Does not require that algorithm be implemented
  - Only need high-level (e.g., pseudocode) description of algorithm
- Takes into account all possible inputs
  - Not just selected inputs
- Allows evaluation of algorithm speed independent of hardware/software environment

"Toy" *E.g.* to Provide Gentle Introduction

- The stair-counting problem
- Adapted from Section 1.2 of textbook

The Stair-Counting Problem

Suppose that you and your friend are standing at the top of a high rise and you wonder how many steps you would have to take to get to the bottom.



Method 1

Walk down and keep a tally.

You grab a pen and a sheet of paper and head on your way. As you take each step, place a mark on the sheet of paper. When you reach the bottom, you run back up and tell your friend.

Method 2

Walk down, but let your friend tally.

Since your friend only has her most valuable pen, she is unwilling to part with it. So, to mark the steps,

- you take a step down placing your hat on the new step (to keep track of where you last stopped)
- run back to your friend to tell her to place a mark on the paper
- run back to the hat and take a new step down

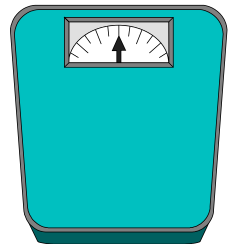
This continues until you reach the bottom step!

Method 3

Call up Bob.

You remember that Bob’s teacher made him run the steps just last week. So you play smart and call him up to find out his results.

Deciding What Operations Count



For each running time analysis, we must identify the *factors of significance* (in time) for weighing our comparisons:

- Which (primitive) operations are important (to be taken into consideration)

***Factors of Significance* for Stair-Counting Problem**

Operations that are important:

- Each time you walk up or down a step
- Each time a mark is placed on the paper

Adding Them Up

- Method 1: 2689 steps down + 2689 steps up + 2689 marks

  3 * 2689 = 8067

- Method 2: (1+2+3+…+2689) steps down

  (1+2+3+…+2689) steps up

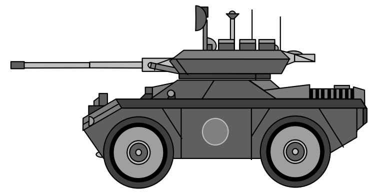
  2689 marks

  3,616,705 * 2 + 2689 = 7,236,099

- Method 3: 4 marks for the digits 2689 on the paper

In More General Terms

For stairway with n steps:



- Method 1: $3n$
- Method 2: $n^2 + 2n$
- Method 3: number of digits in n
  $= \text{floor}(\log_{10} n) + 1$

(refer to p. 17-19 of Textbook for more details of analysis)

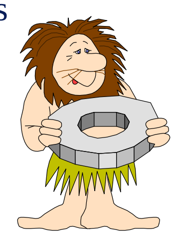
Order (of Magnitude) Analysis

- It's often *difficult and unnecessary* to calculate the *exact number* of operations:
  - Not every operation is executed every time
  - Different operations may require a different amount of effort or actual computing time
    - Addition is easier than multiplication
    - Comparison may be easier than assignment (storing)
- It's often enough to know just the "general behavior" in which the number of operations is affected by input size
  - Remains *constant*?
  - Varies in *logarithmic* fashion?
  - Varies in *linear* fashion?
  - Varies in *quadratic* fashion?
  - …

Such general behavior can be expressed using ***Big O notation***

Big-O Notation, Summarily

- Big-O notation is a way for expressing the ***asymptotic upper-bound order*** of an algorithm
  - *Order* → how operations count (thus time) grows with input size ⇒ constant, linear, quadratic, etc.
  - *Upper-bound* → cannot be any worse than
  - *Asymptotic* → taking into consideration an input size that can be as large as it needs be



- Essentially, Big-O indicates what the "*dominant term*" is in the general expression for operations count when input size (n) becomes *sufficiently large*
  - *Sufficiently large* → as large as it needs to be for some term to emerge as the dominant term

Big-O Notation, Mathematically

- Consider an algorithm which executes $T(n)$ operations when subjected to an input of size n
- We say that the algorithm is $O(\ F(n)\ )$ if

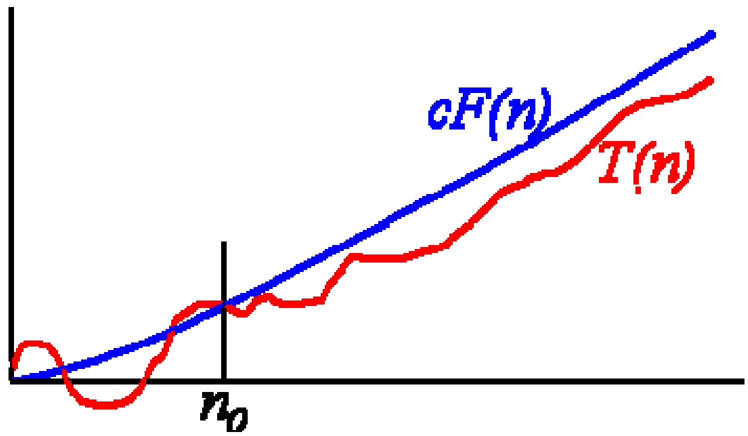
  $T(n) \leq cF(n)$ whenever n is sufficiently large
  - c is some constant
  - n sufficiently large → $n \geq$ some fixed value (*threshold*)
- That is, $T(n)$ *is* $O(\ F(n)\ )$, often written $T(n) = O(\ F(n)\ )$

  if there are positive numbers c and n_0 such that

  $T(n) \leq cF(n)$ for every $n \geq n_0$

Big-O Notation, Conceptually

T(n) is O(F(n))



Big-O Rules

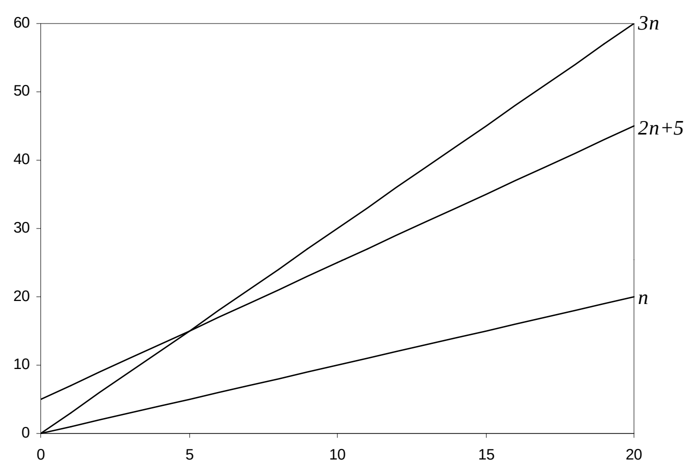
- If $T(n)$ is polynomial of degree d, then $T(n)$ is $O(n^d)$, i.e.,
  - Drop lower-order terms
  - Drop constant factors
- Use *smallest* possible class of functions
  - (characterize as "tightly" as possible)
  - Say “$2n$ is $O(n)$” instead of “$2n$ is $O(n^2)$”, “$2n$ is $O(n^3)$”, etc.
- Use simplest expression of the class
  - Say “$3n + 5$ is $O(n)$” instead of “$3n + 5$ is $O(3n)$”

Big-O Notation, Examples

- $T(n)$ is $O(F(n))$
  if there are positive numbers c and n_0 such that
  $T(n) \le cF(n)$ for every $n \ge n_0$
- Examples:
  - $2n + 5$ is $O(n)$ (pick $c = 3$, $n_0 = 5$)
  - $3n^2 + 9n$ is $O(n^2)$ (pick $c = 6$, $n_0 = 3$)
  - $3n^3 + 5$ is $O(n^3)$ (pick $c = 4$, $n_0 = 2$)
  - $n - 100$ is $O(n)$ (pick $c = 1$, $n_0 = 1$)
  - 15 is $O(1)$ (pick $c = 15$, $n_0 = 1$)
  - $\frac{1}{2} \log_{10} n$ is $O(\log n)$ (pick $c = 1$, $n_0 = 1$)

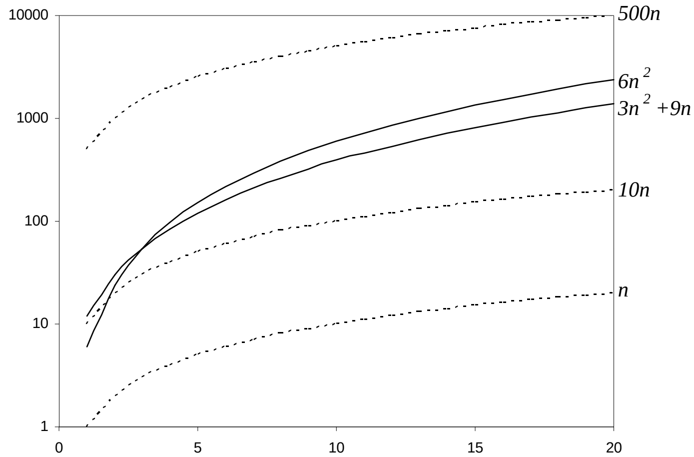
Big-O Notation, Examples

$2n + 5$ is $O(n)$ (pick $c = 3$, $n_0 = 5$)



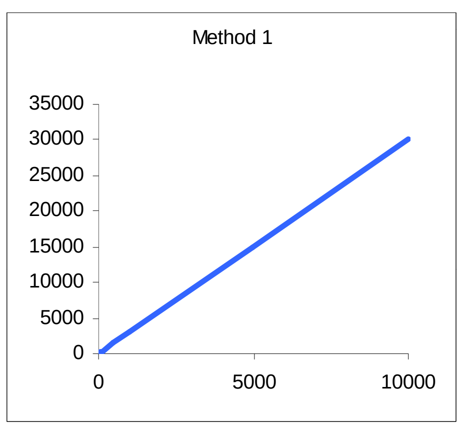
Big-O Notation, Examples

$3n^2 + 9n$ *is* $O(n^2)$ (pick $c = 6, n_0 = 3$)



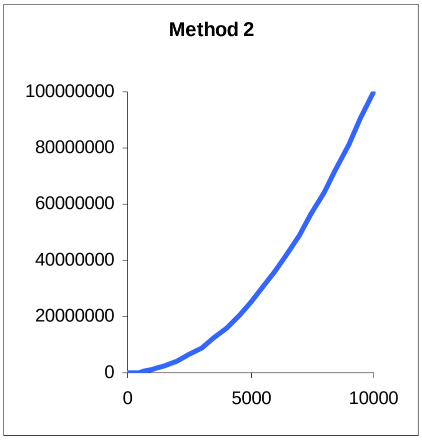
Method 1 Up Close

- 3*n* operations
- O(*n*)
- ***Linear*** Time



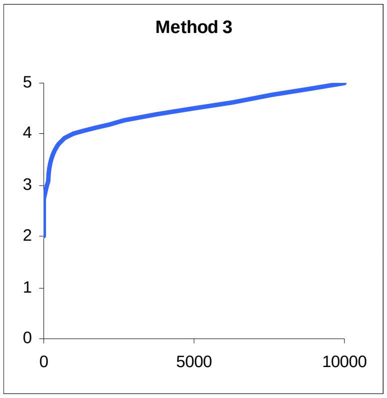
Method 2 Up Close

- $n^2 + 2n$ operations
- $O(n^2)$
- ***Quadratic*** Time



Method 3 Up Close

- floor($\log_{10} n$) + 1
- $O(\log n)$
- ***Logarithmic*** Time



Constant Time

Had we identified the *factors of significance* to include only "walking up or down steps" (i.e., "placing a mark on paper" deemed insignificant), how would the analysis change?

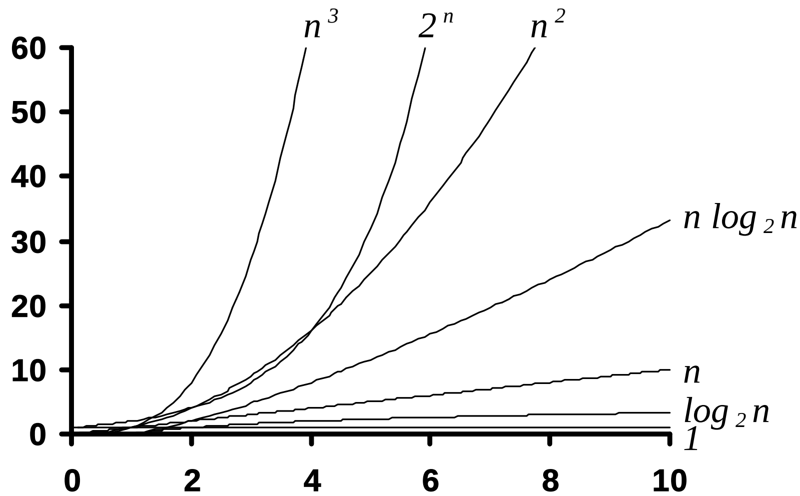
- Method 1 and 2 would remain the same
- Method 3 would have a *constant time*
  - No matter how many steps, it will take you (essentially) the same time since you don't have to run any steps

7 Common Functions in Algorithm Analysis

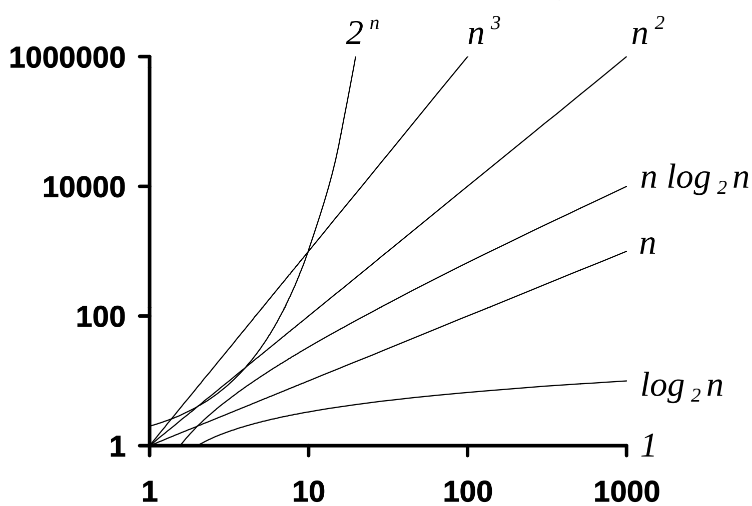
In order of increasing growth rates

1) 1 (constant)
2) $\log n$ (logarithmic)
3) n (linear)
4) $n \log n$ (n-log-n)
5) n^2 (quadratic)
6) n^3 (cubic)
7) a^n (exponential)

Graph of Common Computing Time Functions



Graph of Common Computing Time Functions



Analysis Applied to Program Code

Analysis technique seen in the stairs counting problem can be applied to program code, such as a C++ function

- Section 1.2 of textbook (pp. 22-23) gives an example
- More examples in the next set of lecture notes
  - ***010OrderAnalysisOfAlgorithms02***

Textbook Readings

- Chapter 1
  - Section 1.2
- Appendix B