Lecture 2: Algorithm Analysis

Data Structures and Algorithms

- A famous quote: Program = Algorithm + Data Structure
- Algorithm
  - Outline, the essence of a computational procedure, step-by-step instructions
- Program – an implementation of an algorithm in some programming language
- Data structure
  - **Organization** of data needed to solve the problem

Algorithm Specification

- Criteria
  - input: zero or more quantities that are externally supplied
  - output: at least one quantity is produced
  - definiteness: clear and unambiguous
  - finiteness: terminate after a finite number of steps

- Representation
  - A natural language, like English or Chinese.
  - A graphic, like flowcharts.
  - A computer language, like C.

Algorithm Analysis

- Analysis:
  - How to predict an algorithm’s performance
  - How well an algorithm scales up
  - How to compare different algorithms for a problem

- Data Structures
  - How to efficiently store, access, manage data
  - Data structures effect algorithm’s performance

Example

- Two algorithms for computing the Factorial
- Which one is better?

```
int factorial (int n) {
  if (n <= 1)    return 1;
  else   return n * factorial(n-1);
}
```

```
int factorial (int n) {
  if (n<=1)    return 1;
  else {
    fact = 1;
    for (k=2; k<=n; k++)
        fact *= k;
    return fact;
  }
}
```

Measuring Algorithm Performance?

- What metric should be used to judge algorithms?
  - Length of the program (lines of code)
  - Ease of programming (bugs, maintenance)
  - Memory required
  - Running time

- Running time is the dominant standard
  - Quantifiable and easy to compare
  - Often the critical bottleneck

Running Time

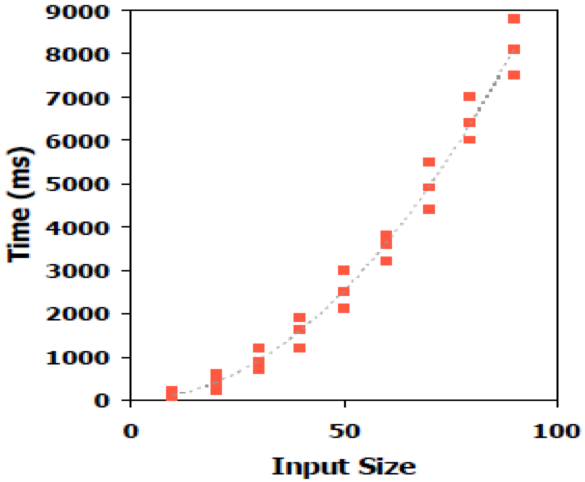
- The running time of an algorithm varies with the input and typically grows with the input size.

- Average case difficult to determine.

- In most of computer science we focus on the *worst* case running time.
  - Easier to analyze.
  - Crucial to many applications: what would happen if an autopilot algorithm ran drastically slower for some unforeseen, untested inputs?

Measuring running time?

- Experimentally
  - Write a program implementing the algorithm
  - Run the program with inputs of varying size
  - Measure the actual running times and plot the results



Why not?

- You have to implement the algorithm which isn't always doable!
- Your inputs may not entirely test the algorithm.
- The running time depends on the particular computer's hardware and software speed.

Theoretical Analysis

- Uses a high-level description of the algorithm instead of an implementation.
- Take into account all possible inputs.
- Evaluate speed of an algorithm independent of the hardware or software environment.
- By inspecting pseudocode, we can determine the number of statements executed by an algorithm as a function of the input size.

Elementary Operations

- Algorithmic "time" is measured in elementary operations:
  - Math (+, -, *, /, max, min, log, sin, cos, abs, ...)
  - Comparisons (==, >, <=, ...)
  - Function calls and value returns
  - Variable assignment
  - Variable increment or decrement
  - Array allocation
  - Creating a new object

- In practice, all of these operations take different amounts of time.
- For the purpose of algorithm analysis, we assume each of these operations takes the same time: "1 operation"

Best/Worst/Average Case

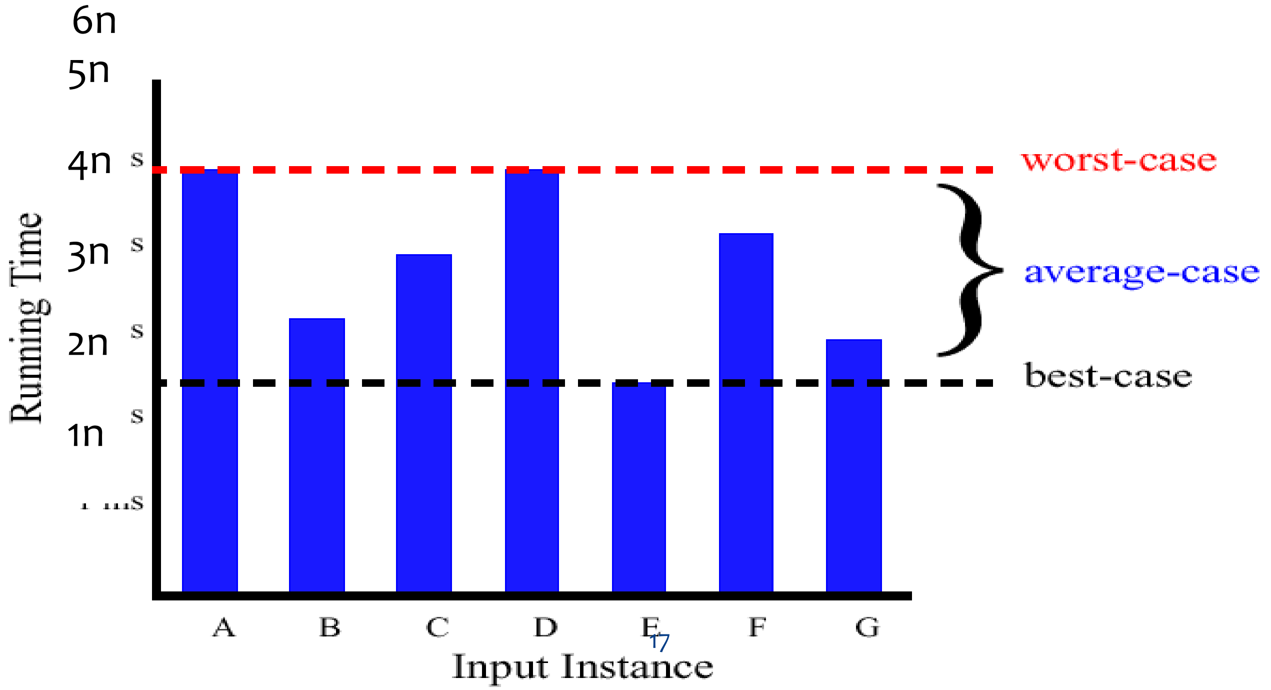
- **Best case**: elements already sorted ® t_j=*1*, running time = *f(n)*, i.e., *linear* time.

- **Worst case**: elements are sorted in inverse order
  ® t_j=*j*, running time = $f(n^2)$, i.e., *quadratic* time

- **Average case**: t_j=*j/2*, running time = $f(n^2)$, i.e., *quadratic* time

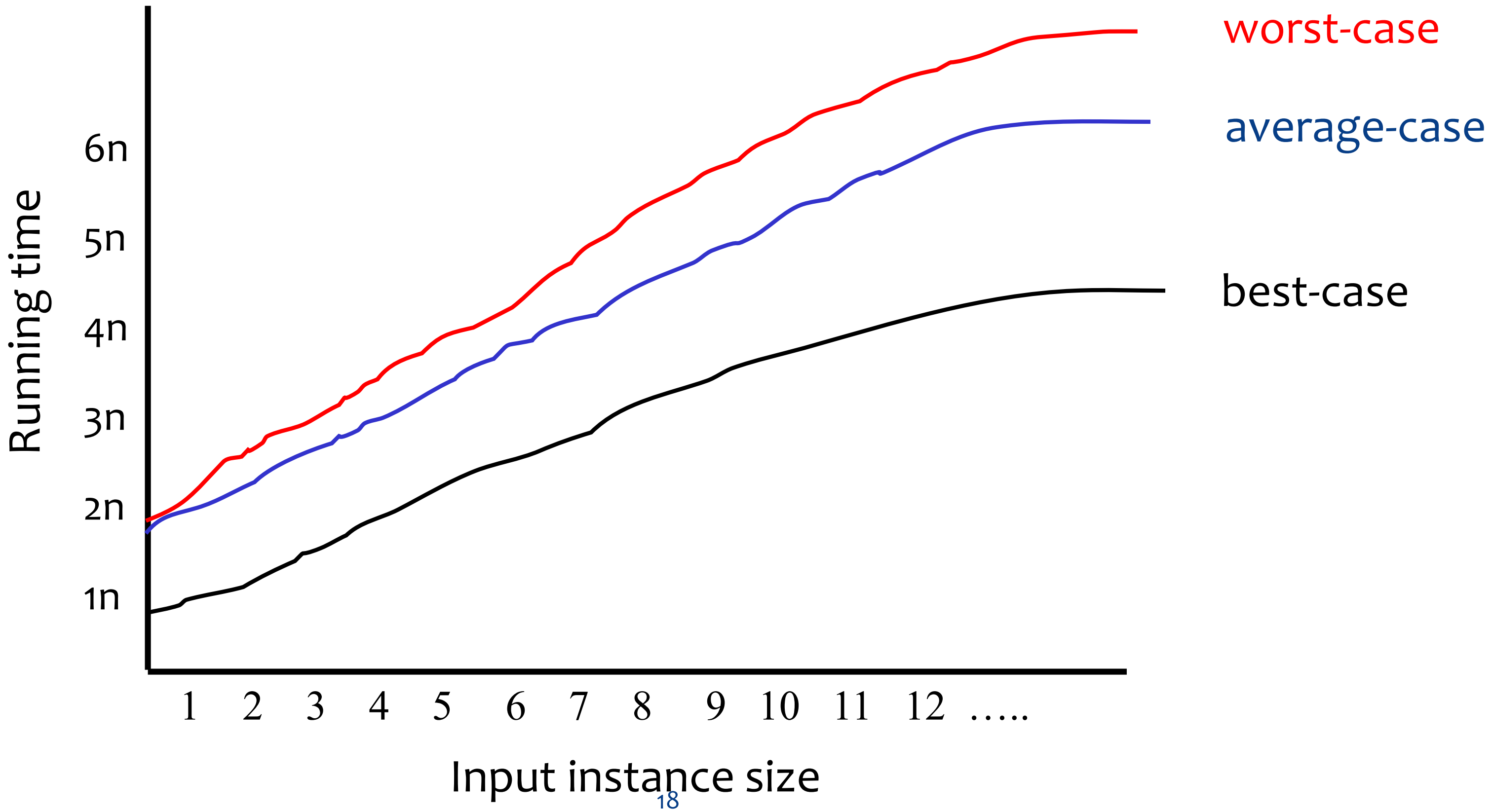
Best/Worst/Average Case

- For a specific size of input *n*, investigate running times for different input instances:



Best/Worst/Average Case

- For inputs of all sizes:



Best/Worst/Average Case

- **Worst case** is usually used:
  - It is an upper-bound and in certain application domains (e.g., air traffic control, surgery) knowing the **worst-case** time complexity is of crucial importance
  - For some algorithms **worst case** occurs fairly often
  - The **average case** is often as bad as the **worst case**
  - Finding the **average case** can be very difficult

Big O Notation

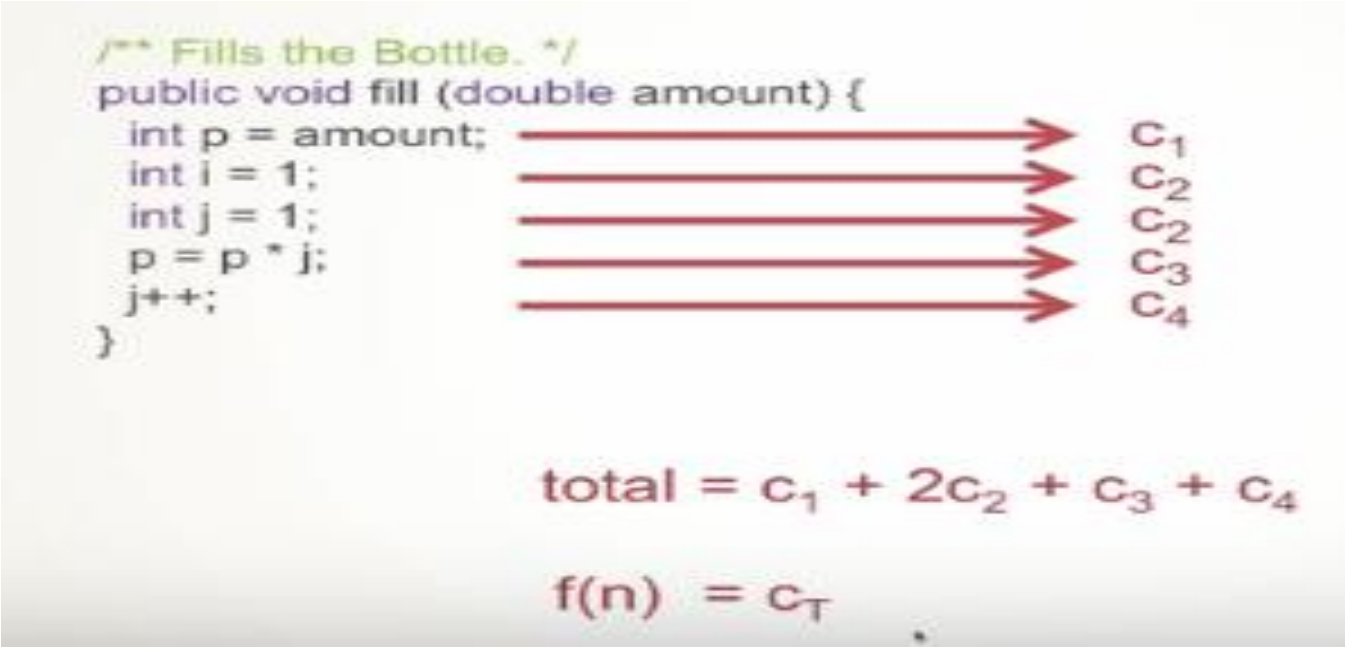
- Big O notation is used in Computer Science to describe the performance or complexity of an algorithm.

- It is the formal method of expressing the upper bound of an algorithm's running time. It's a measure of the longest amount of time it could possibly take for the algorithm to complete.

- Big O specifically describes the **worst-case** scenario, and can be used to describe the execution time required or the space used (e.g. in memory or on disk) by an algorithm.

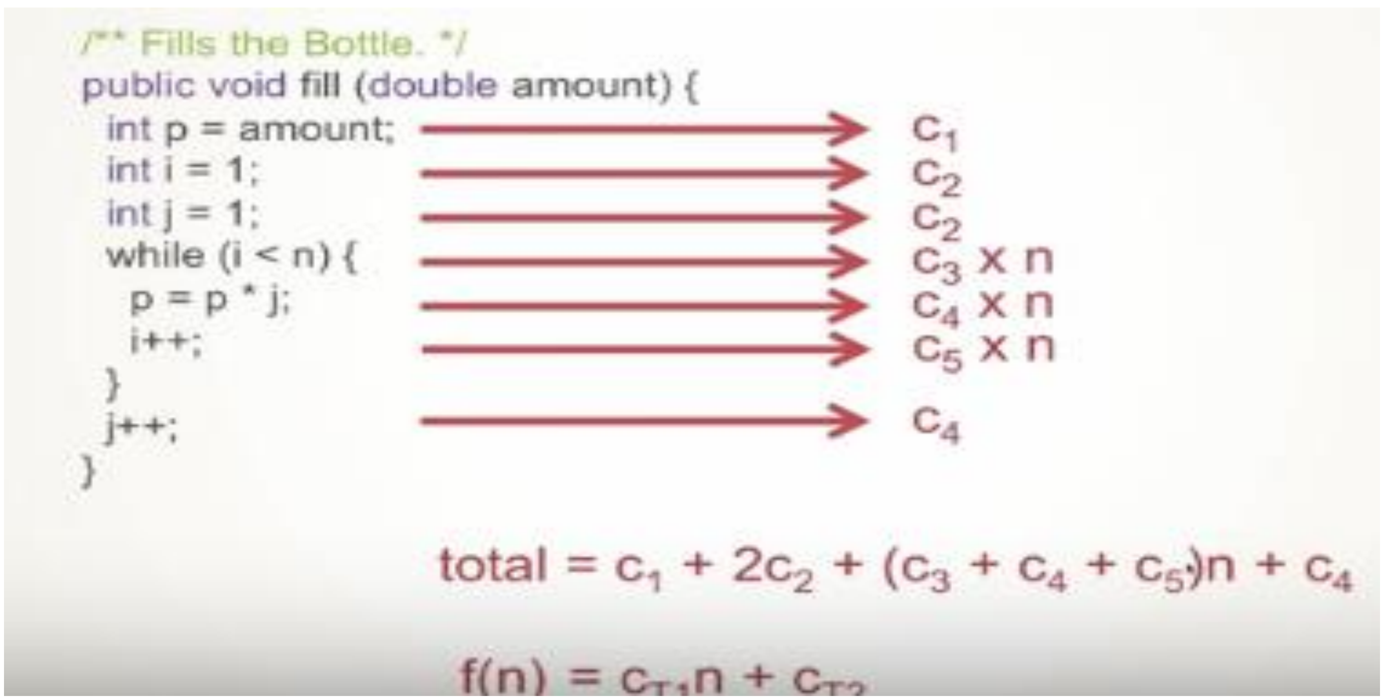
Constant Running Time

- **O(1)**
  - O(1) describes an algorithm that will always execute in the same time (or space) regardless of the size of the input data set.



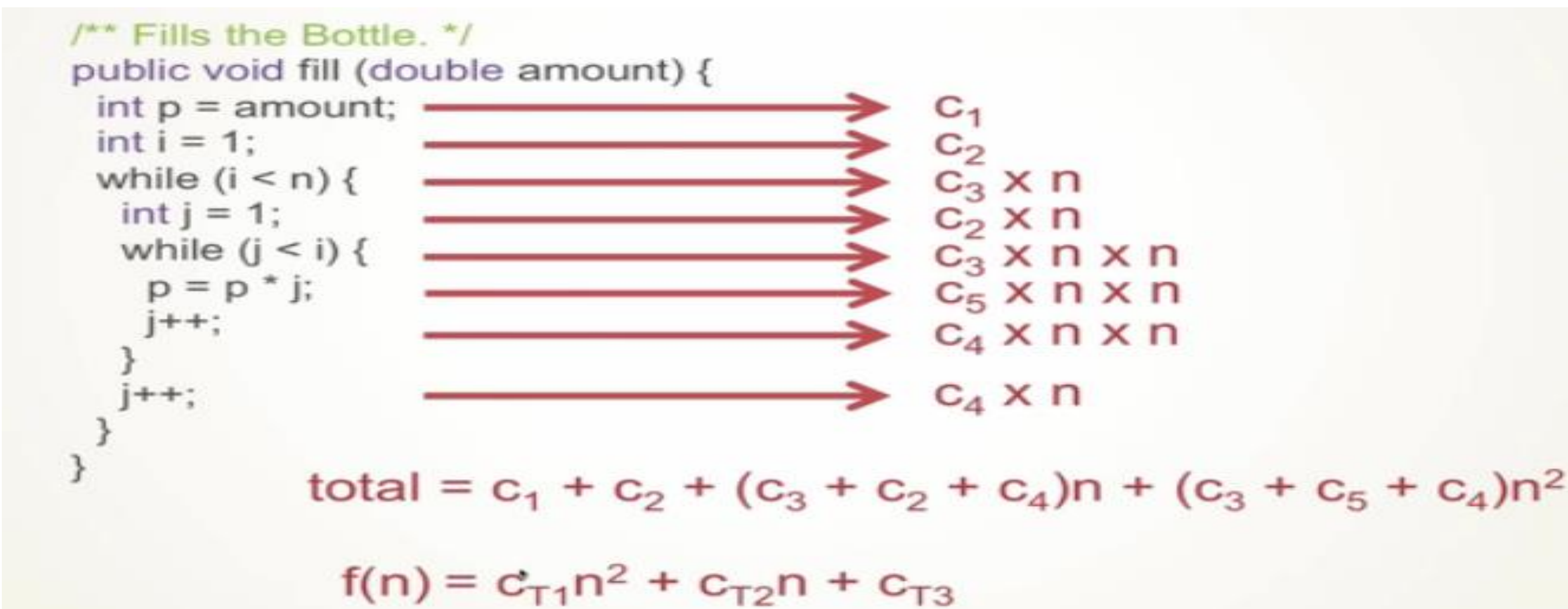
Linear Running Time

- **O(N)**
  - O(N) describes an algorithm whose performance will grow linearly and in direct proportion to the size of the input data set.



Quadratic Running Time

- **O(N²)**
  - $O(N^2)$ represents an algorithm whose performance is directly proportional to the square of the size of the input data set.
  - This is common with algorithms that involve nested iterations over the data set. Deeper nested iterations will result in $O(N^3)$, $O(N^4)$ etc.



Logarithms O(log N)

- **Logarithms O(log N)**
  - The iterative halving of data sets described in the binary search example produces a growth curve that peaks at the beginning and slowly flattens out as the size of the data sets increase.

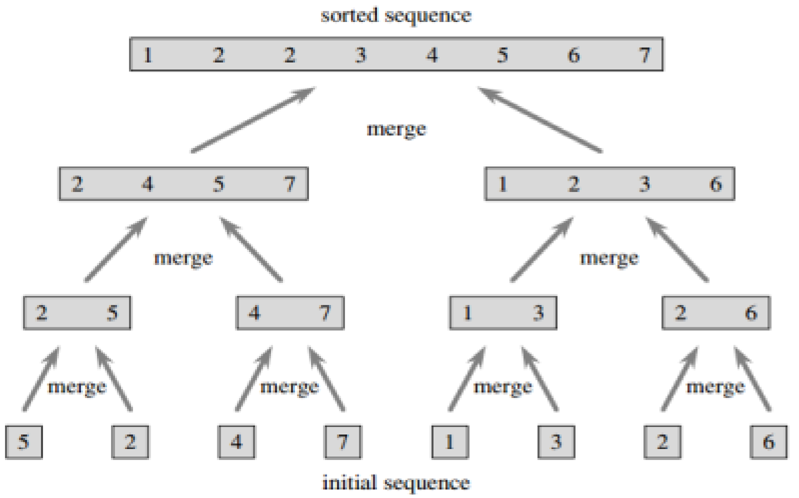
$O(2^N)$

- **$O(2^N)$**
  - $O(2^N)$ denotes an algorithm whose growth will double with each additional element in the input data set. The execution time of an $O(2^N)$ function will quickly become very large.

Recurrence Relation

- When an algorithm contains a recursive call to itself, its running time can often be described by a recurrence equation or recurrence, which describes the overall running time on a problem of size n in terms of the running time on smaller inputs.
- Example: (Merge Sort)

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \,, \\ 2T(n/2) + \Theta(n) & \text{if } n > 1 \,. \end{cases}$$

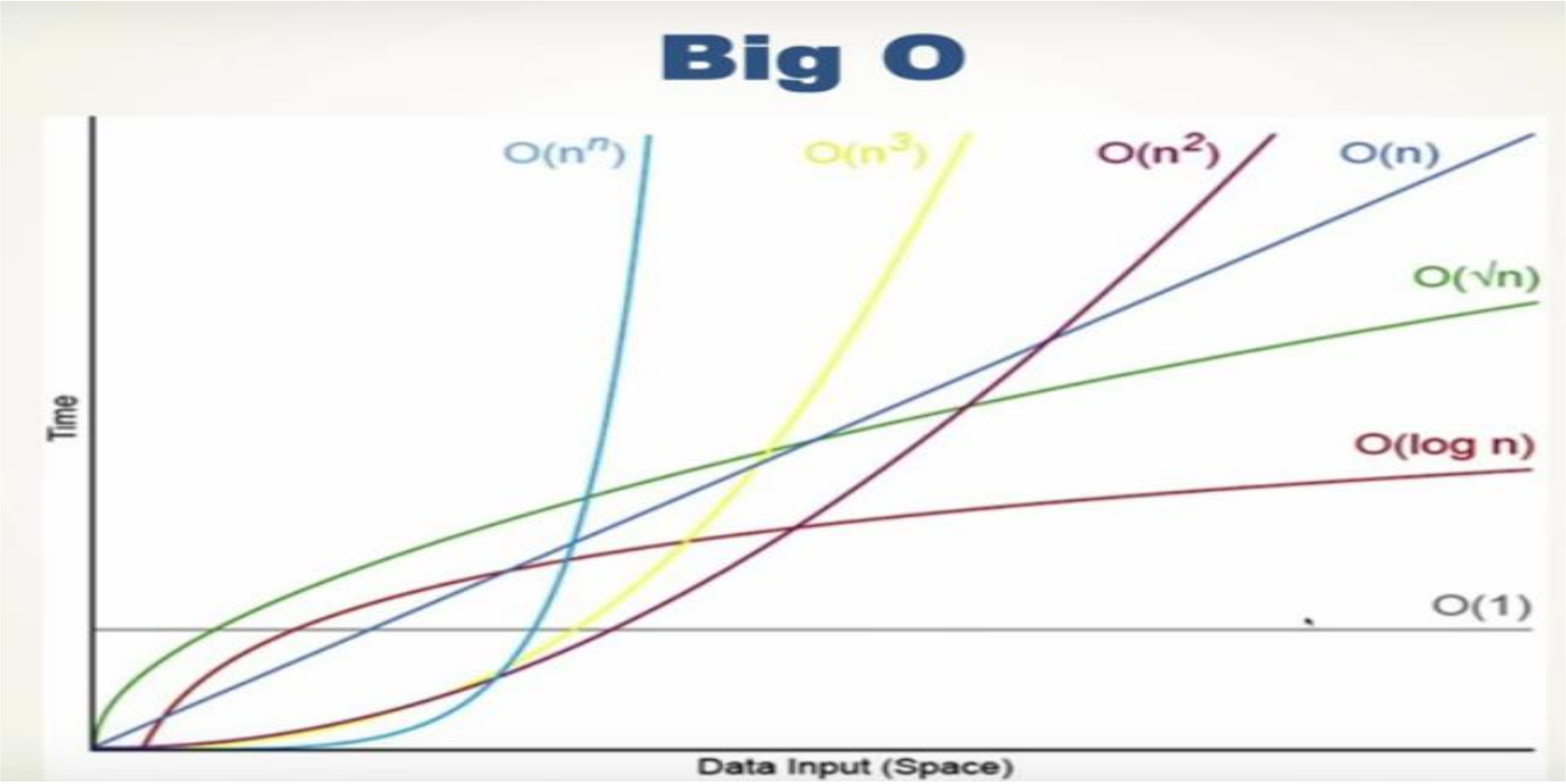


The operation of merge sort on the array $A = \langle 5, 2, 4, 7, 1, 3, 2, 6 \rangle$. The lengths of the sorted sequences being merged increase as the algorithm progresses from bottom to top.

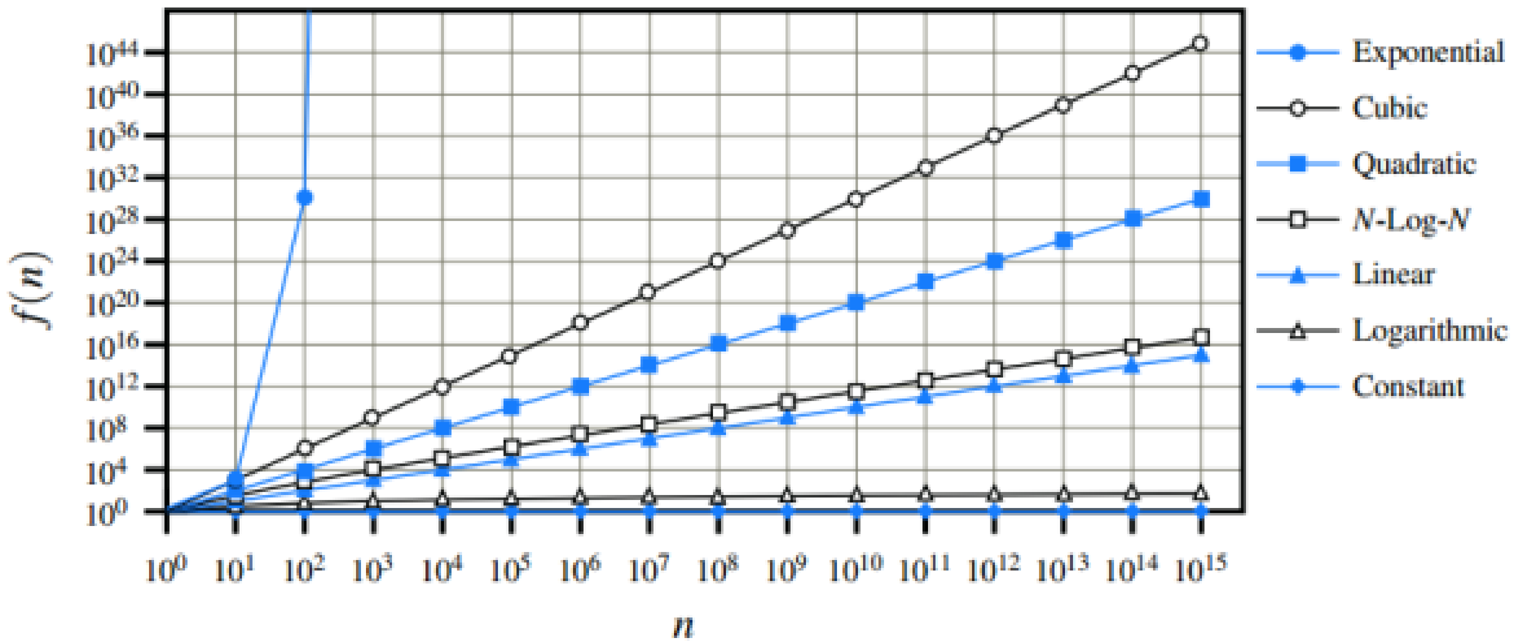
Common functions for analysis of algorithms

#	Name	Function and Description	Example
1	Constant	$f(n) = c$, for some fixed constant c	Array Indexing
2	Logarithm	$f(n) = \log_b n$, for some constant b>1	Binary Search
3	linear	$f(n) = n$	Linked List indexing
4	N-Log-N	$f(n) = n\log n$	Merge Sort
5	Quadratic	$f(n) = n^2$	Insertion Sort
6	Cubic	$f(n) = n^3$	
7	Polynomials	$f(n) = a_0 + a_1 n + a_2 n^2 + a_3 n^3 + \cdots + a_d n^d$	
8	Exponential	$f(n) = b^n$ where b is a positive constant, called the base, and the argument n is the exponent	TSP (Travelling Sales Person)

Comparison



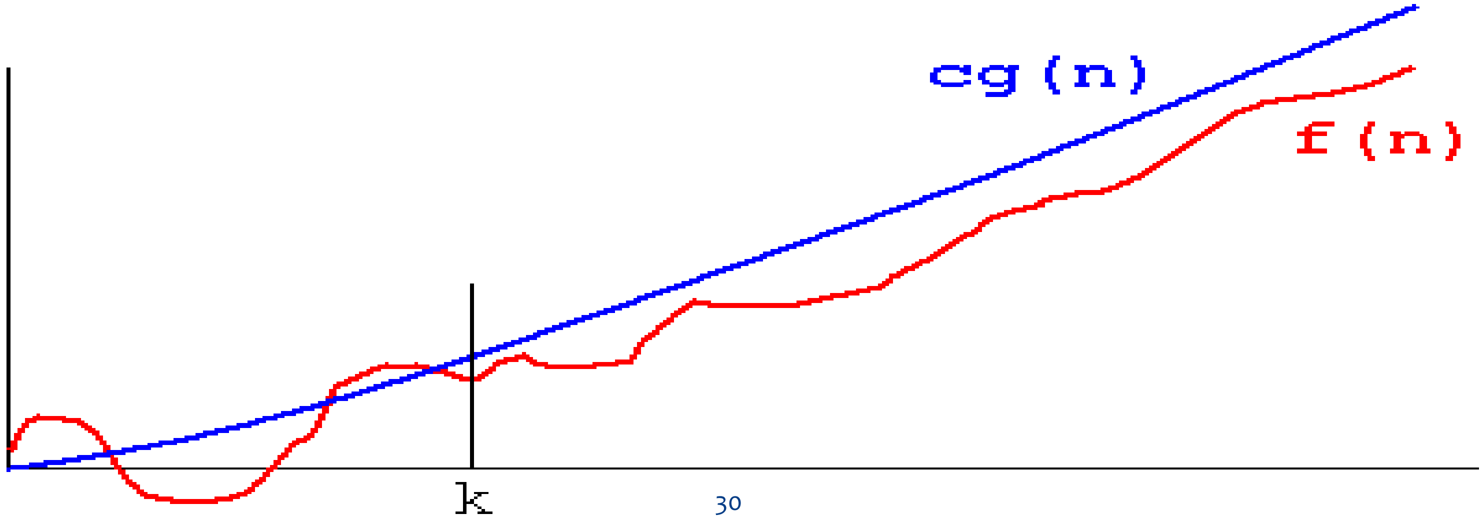
Comparison



Lg N < N < N log N < N^2 < N^3 < b^N

Comparison

- Let f(n) and g(n) be two functions on positive integers. We say f(n) is O(g(n)) if there exist two positive constants c and k such that **f(n) <= cg(n)** for all n >= k.



Proving Big-Oh: Example

Example 1: Prove that running time $T(n) = n^3 + 20n + 1$ is $O(n^3)$

Proof: by the Big-Oh definition, $T(n)$ is $O(n^3)$ if $T(n) \le c \cdot n^3$ for some $n \ge n_0$. Let us check this condition: if $n^3 + 20n + 1 \le c \cdot n^3$ then $1 + \frac{20}{n^2} + \frac{1}{n^3} \le c$. Therefore, the Big-Oh condition holds for $n \ge n_0 = 1$ and $c \ge 22$ $(= 1 + 20 + 1)$. Larger values of n_0 result in smaller factors c (e.g., for $n_0 = 10$ $c \ge 1.201$ and so on) but in any case the above statement is valid.

Proving Big-Oh: Example

- f(n) = 10n + 5 and g(n) = n
  f(n) is O(g(n))
- To show f(n) is O(g(n)) we must show constants c and k such that f(n) <= cg(n) for all n >=k
- or: 10n+5 <= cn for all n >= k
- Try c = 15. Then we need to show: 10n + 5 <= 15n.
- Solving for n we get: 5 <= 5n or 1 <= n.
- So f(n) = 10+5 <= 15g(n) for all n >= 1. (c = 15, k = 1).
- Therefore we have shown f(n) is O(g(n)).

Proving Big-Oh: Example

Show that $f(n) = n^2 + 2n + 1$ is $O(n^2)$.

Choose $k = 1$.

Assuming $n > 1$, then

$$\frac{f(n)}{g(n)} = \frac{n^2 + 2n + 1}{n^2} < \frac{n^2 + 2n^2 + n^2}{n^2} = 4$$

Choose $C = 4$. Note that $2n < 2n^2$ and $1 < n^2$.

Thus, $n^2 + 2n + 1$ is $O(n^2)$ because
$n^2 + 2n + 1 \leq 4n^2$ whenever $n > 1$.

Proving Big-Oh: Example

Show that $f(n) = 3n + 7$ is $O(n)$.

Choose $k = 1$.

Assuming $n > 1$, then

$$\frac{f(n)}{g(n)} = \frac{3n+7}{n} < \frac{3n+7n}{n} = \frac{10n}{n} = 10$$

Choose $C = 10$. Note that $7 < 7n$.

Thus, $3n + 7$ is $O(n)$ because $3n + 7 \leq 10n$ whenever $n > 1$.

Proving Big-Oh: Example

Show that $f(n) = (n+1)^3$ is $O(n^3)$.

Choose $k = 1$.

Assuming $n > 1$, then

$$\frac{f(n)}{g(n)} = \frac{(n+1)^3}{n^3} < \frac{(n+n)^3}{n^3} = \frac{8n^3}{n^3} = 8$$

Choose $C = 8$. Note that $n+1 < n+n$ and $(n+n)^3 = (2n)^3 = 8n^3$. Thus, $(n+1)^3$ is $O(n^3)$ because $(n+1)^3 \leq 8n^3$ whenever $n > 1$.

Proving Not Big-Oh: Example

Show that $f(n) = n^2 - 2n + 1$ is not $O(n)$.

Assume $n > 1$, then

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

$n > C + 2$ implies $n - 2 > C$ and $f(n) > Cn$.

So choosing $n > 1$, $n > k$, and $n > C + 2$ implies $n > k \wedge f(n) > Cn$.

- "Decrease" numerator to "simplify" fraction.