Why Study Algorithms

CS 251 Design and Analysis of Algorithms

Anis ur Rahman
Department of Computing
NUST-SEECS
Islamabad

February 2, 2016

This course is about

1. Vocabulary for design and analysis of algorithms
   - Big-Oh notation
   - high level reasoning about algorithms
   - general analysis methods (Master Method)

This course is about

1. Vocabulary for design and analysis of algorithms
   - Big-Oh notation
   - high level reasoning about algorithms
   - general analysis methods (Master Method)
2. Divide and conquer algorithm design paradigm
   - applied to integer multiplication, sorting, matrix multiplication, closest pair, etc.

This course is about

1. Vocabulary for design and analysis of algorithms
   - Big-Oh notation
   - high level reasoning about algorithms
   - general analysis methods (Master Method)
2. Divide and conquer algorithm design paradigm
   - applied to integer multiplication, sorting, matrix multiplication, closest pair, etc.
3. Randomization in algorithm design
   - applied to QuickSort, primality testing, graph partitioning, hashing.

This course is about

1. Vocabulary for design and analysis of algorithms
   - Big-Oh notation
   - high level reasoning about algorithms
   - general analysis methods (Master Method)
2. Divide and conquer algorithm design paradigm
   - applied to integer multiplication, sorting, matrix multiplication, closest pair, etc.
3. Randomization in algorithm design
   - applied to QuickSort, primality testing, graph partitioning, hashing.
4. Primitives for reasoning about graphs
   - connectivity information, shortest paths, structure of information and social networks.

This course is about

1. Vocabulary for design and analysis of algorithms
   - Big-Oh notation
   - high level reasoning about algorithms
   - general analysis methods (Master Method)
2. Divide and conquer algorithm design paradigm
   - applied to integer multiplication, sorting, matrix multiplication, closest pair, etc.
3. Randomization in algorithm design
   - applied to QuickSort, primality testing, graph partitioning, hashing.
4. Primitives for reasoning about graphs
   - connectivity information, shortest paths, structure of information and social networks.
5. Use and implementation of data structures
   - Heaps, balanced binary search trees, hashing, and many more

Skills You'll Learn

1. Become a better programmer
2. Sharpen your mathematical and analytical skills
3. Start "thinking algorithmically"
4. Literacy with computer science's "greatest hits"
5. Ace your technical interviews

Who Are You?

1. Ideally, you know some programming.
2. Doesn't matter which language(s) you know.
   - But you should be capable of translating high-level algorithm descriptions into working programs in some programming language.
3. Some mathematical experience.
   - basic discrete math, proofs by induction, etc.
   - Check out "Mathematics for Computer Science", by Eric Lehman and Tom Leighton.

Logistics

Class meets.

- Mondays 11:00 – 12:00
- Tuesdays 11:00 – 12:00
- Wednesdays 11:00 – 12:00

Web Site.

- lms.nust.edu.pk

E-mail.

- anis.rahman@seecs.edu.pk

Source Material

No required textbook. A few of the many good ones:

1. Skiena, Algorithm Design Manual, 2008 (2nd edition).
2. Cormen/Leiserson/Rivest/Stein, Introduction to Algorithms, 2009 (3rd edition).
3. Kleinberg/Tardos, Algorithm Design, 2005.
4. Dasgupta/Papadimitriou/Vazirani, Algorithms, 2006.
5. Mehlhorn/Sanders, Data Structures and Algorithms: The Basic Toolbox, 2008.
6. Web is the best and greenest "textbook"

No specific development environment required.

- But you should be able to write and execute programs.

General information

Prerequisites.

- CS110, CS250

Assessment.

- 5-6 Quizzes (15%)
- 2-3 Assignments (10%)
- Term paper (5%)
- OHTs (30%)
- ESE (40%)

Algorithm: background

What is an Algorithm?
In mathematics, computer science, and related subjects, an algorithm is an effective method for solving a problem expressed as a finite sequence of instructions.

Algorithm: background

What is an Algorithm?
In mathematics, computer science, and related subjects, an algorithm is an effective method for solving a problem expressed as a finite sequence of instructions.

Abu Abdullah Muhammad bin Musa al-Khwarizmi

- a Muslim mathematician.
- well-known work: Kitab Al-jabr w'al-muqabala.
- invented algorithms to solve quadratic equations.
- word algorithm derived by his Latin name Algorithmi.

Algorithm: question

Which one is an algorithm?

- A recipe for making tomato soup?
- A procedure to sort 1000 numbers?
- A procedure to recognize a particular face in a crowd?
- A procedure to judge beauty?

Algorithm: designing algorithms

Four steps:

- What is a Problem?
- **Correctness.** is whether the proposed solution works.
- **Efficiency.** is the desired running time of the algorithm compared to some benchmark.
- How to express an Algorithm

Algorithm: solving a problem

Q. How to solve a new problem?

Algorithm: solving a problem

Q. How to solve a new problem?

- use basic building blocks
  - to achieve correctness then to worry about efficiency

"so simple only a genius could have thought of it"—Albert Einstein

Algorithm: what is good design?

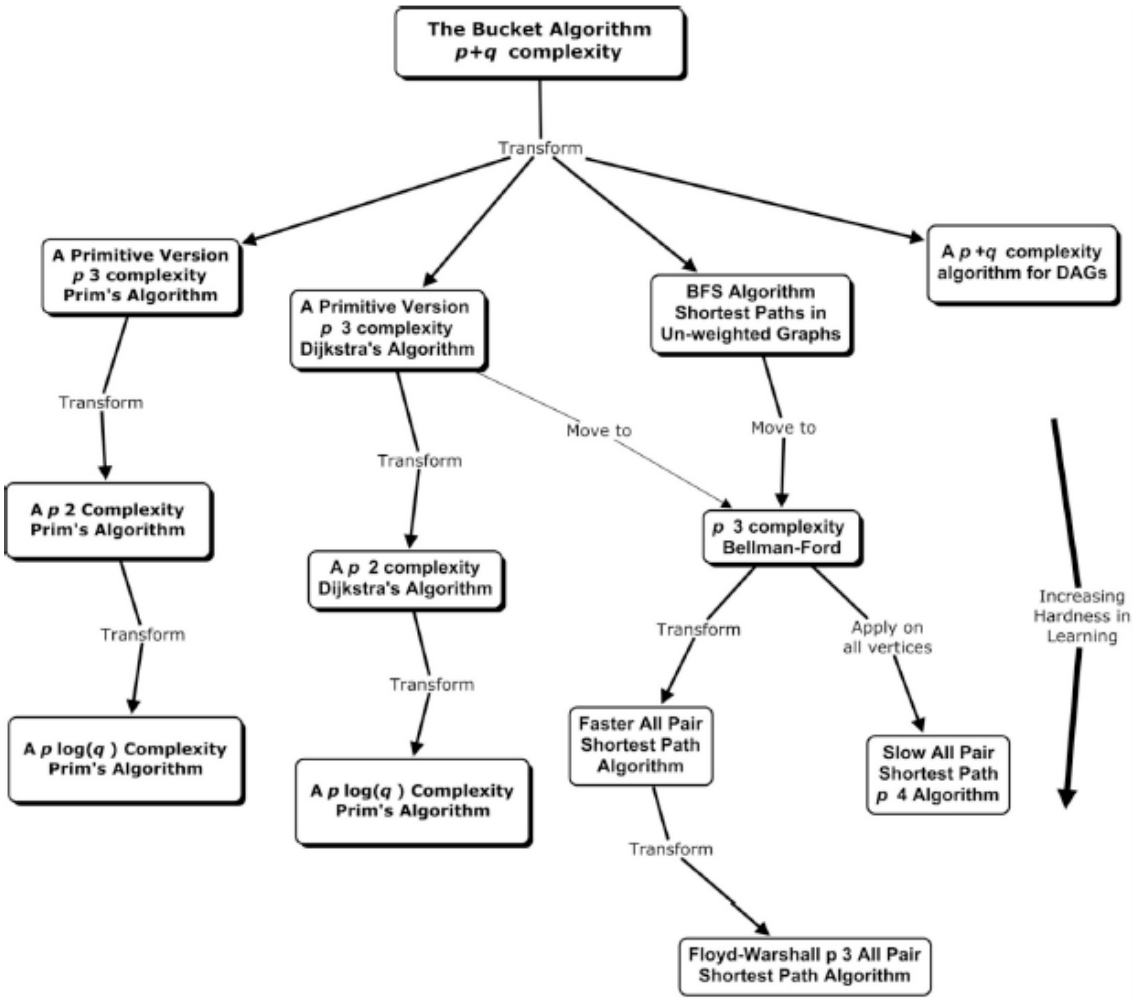
A good design corresponds to:

- selection of suitable building blocks.
- use an appropriate building block.
- find interconnections between these building blocks.

Algorithm: designing algorithms

- Start with simple building blocks
- Use tested tools and techniques
- Initial design should be as simple as possible
- Worry about correctness
- Then worry about efficiency

Algorithm: designing algorithms



Algorithm analysis: socratic questioning

Socratic questioning is systematic, disciplined, and deep, and usually focuses on fundamental concepts, principles, theories, issues, or problems.

- to pursue thought in many directions
- to open up issues and problems
- to uncover assumptions
- to analyze concepts
- to distinguish what we know from what we don't know
- to follow out logical implications of thought, or to control the discussion

Algorithm analysis: problem

Find the eldest person six billion people.

- a) We need to perform six billion operations?
- b) We can do it by doing less than six billion operations by using fast computers?
- c) We can do it by doing about three billion operations by using intelligent algorithms?

Algorithm: how to express one?

- Input
- Output
- Basic idea in simple language
- Complexity calculations

Tiny assignment for today

Q. A good example problem for showing algorithms with different orders of magnitude is the classic anagram detection problem for strings. One string is an anagram of another if the second is simply a rearrangement of the first. For example, 'heart' and 'earth' are anagrams; 'silent' and 'listen' are anagrams as well. For the sake of simplicity, assume that the two strings in question are of equal length and are made up of the set of 26 lowercase alphabetic characters. Our goal is to write a boolean function that will take two strings and return whether they are anagrams.

Algorithm: how to express one?

```
MATRIX-MULTIPLY(A,B)
   if A.columns ≠ B.rows
      error "incompatible dimensions"
   else
      let C be a new A.rows × B.columns matrix
      for i = 1 to A.rows
         for j = 1 to B.columns
            c_ij = 0
            for k = 1 to A.columns
               c_ij = c_ij + a_ik · b_kj
   return C
```

Algorithm: how to express one?

Precondition: An image X of size $W \times H$

1 **function** StaticPathway(X)
2 $X' \leftarrow$ RetinalFilter(X)
3 $M_{u,v} \leftarrow$ GaborFilterBank(X')
4 $M_{u,v} \leftarrow$ Interactions($M_{u,v}$)
5 $M_{u,v} \leftarrow$ Normalizations($M_{u,v}$)
6 $M^s \leftarrow$ Fusion($M_{u,v}$)
7 **return** M^s

Algorithm: problem

Input: An Array A of k numbers. $A[1...k]$. The first $k-1$ numbers are sorted in ascending order, the last number is out of place.

$$1, 3, 7, 9, 13, 17, 23, 4$$

Output: An array $A[1...k]$, which is sorted in ascending order.

$$1, 3, 4, 7, 9, 13, 17, 23$$

Q. Write the algorithm, verify its correctness.

Algorithm: moveMin

```
for i = k to 1
   if A[i] < A[i-1]
      SWAP(A[i],A[i-1])
```

Q. Will it swap till the first element or stop?

Q. Is this enough for moveMin?

Algorithm: moveMin

```
for i = k to 1
   if A[i] < A[i-1]
      SWAP(A[i],A[i-1])
```

Q. Will it swap till the first element or stop?

For correctness, it is important to check the entire array—to be sure.

Also, for simplicity of the algorithm.

Q. Is this enough for moveMin?

Algorithm: moveMin

```
for i = k to 1
   if A[i] < A[i-1]
      SWAP(A[i],A[i-1])
```

Q. Will it swap till the first element or stop?

For correctness, it is important to check the entire array—to be sure. Also, for simplicity of the algorithm.

Q. Is this enough for moveMin?

This is not a general algorithm to sort any unordered array. It is just a building block for the sorting algorithm.

Summary

- Algorithms are a finite set of operations to solve a problem.
- Initial design of an algorithm should be:
  - **Simple.** using known concepts and principles.
  - **Clear.** described in plain English, or simple notation.
  - **Correct.** solve the problem.

Algorithm analysis: background

Why is a algorithm analysis important?

- Computer scientists learn by experience.
- Learn techniques to solve for one problem, and use it for another.
- Algorithm design helps to solve more challenging problems.

Algorithm analysis: background

Why is a algorithm analysis important?

- Computer scientists learn by experience.
- Learn techniques to solve for one problem, and use it for another.
- Algorithm design helps to solve more challenging problems.

For example,

- Consider two solutions for one problem.
- Both give the correct result.
- One might be resource efficient compared to the other.

Algorithm analysis: background

Why is a algorithm analysis important?

- Computer scientists learn by experience.
- Learn techniques to solve for one problem, and use it for another.
- Algorithm design helps to solve more challenging problems.

For example,

- Consider two solutions for one problem.
- Both give the correct result.
- One might be resource efficient compared to the other.

In this lecture, we will learn analysis techniques to compare solely on their characteristics,

- not on the computer system used, or the implementation.

Algorithm analysis: background

Without algorithm analysis, there will always be some questions:

- Is the implementation correct?
- Are there any bugs?
- How much faster?

Algorithm analysis: background

"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 (1864)

Algorithm analysis: background

"It is convenient to have a measure of the amount of work involved in a computing process, even though it may be a very crude one. We may count up the number of times that various elementary operations are applied in the whole process..."

Alan Turing (1947)

Algorithm analysis: the challenge

Q. Will the algorithm fare well for large input?

- the program is very slow.
- the program is out of memory.

Algorithm analysis: the outcomes

Algorithm analysis is used to:

- Predict performance.
- Compare algorithms.
- Provide guarantees.

The main motivation is to avoid performance bugs.

Algorithm analysis: example

Discrete Fourier transform.

- Break down waveform of N samples into periodic components.
- **Applications.** spectral analysis, convolution, data compression, ...
- **Brute force method.** N^2 steps.
- **FFT algorithm.** *NlogN* steps, enables new technology.

Algorithm analysis: quantitative analysis

What to use quantitative, or qualitative analysis.
As an engineer, for the proposed algorithm:

- one must determine the actual costs (memory, time, monetary).
- comparison of qualities—faster, less memory—are unimportant.

Algorithm analysis: running time

Running time of the program is a function $T(N)$,

- represents the number of units of time taken by an algorithm on any input of size N, or
- the total number of statements executed by the program.

For example, a program may have a running time $T(N) = cN$, where c is some constant.

- the program is linearly proportional to the size of the input on which it is run.
- running time is linear time, or just linear.

Algorithm analysis: empirical analysis

Q. How to time a program?

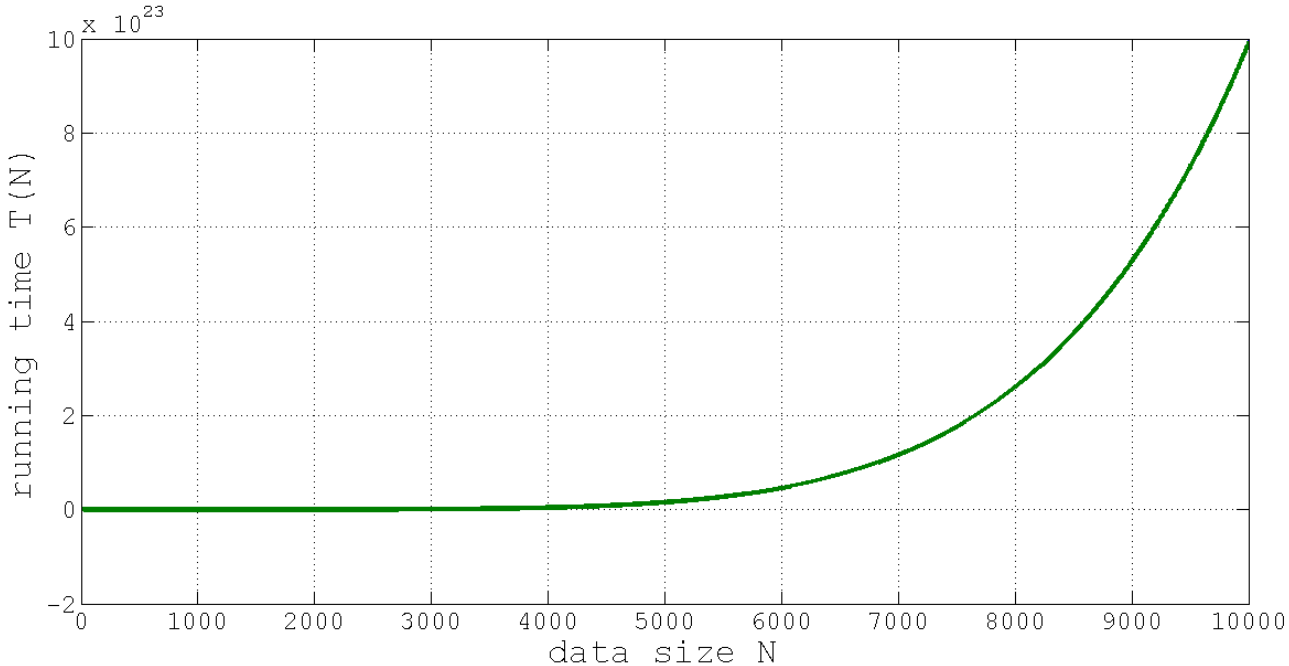
- **manual.** using a stopwatch
- **automatic.** using some timer function

Run the program for various input sizes and measure running time.

data size	time (ms)
250	5
500	8
1000	10
2000	15
4000	30
8000	50
16000	75

Algorithm analysis: data analysis

Plot running time $T(N)$ vs. input size N.



Algorithm analysis: question

What are the different factor affecting the running-time of an algorithm?

Algorithm analysis: running time

Different factors influencing the running time of an algorithm are:

- processing power of computer
- algorithm used
- size of the input
- structure of the input
- amount of available memory
- implementation language

Algorithm analysis: running time

We consider three simplifications to analyze algorithms:

- without any implementation
- taking the worst possible inputs
- ignoring details

Algorithm analysis: running time

We consider three simplifications to analyze algorithms:

- without any implementation => RAM model
- taking the worst possible inputs => worst-case running time
- ignoring details => Big-O or Landau notation

Algorithm analysis: running time

We consider three simplifications to analyze algorithms:

- without any implementation => **RAM model**
- taking the worst possible inputs => worst-case running time
- ignoring details => Big-O or Landau notation

RAM model: overview

Random access machine.

- Considers a very simple computer.
- Motivation is to analyze an algorithm on a minimal computer model.

Q. What do you consider to be a minimal computer model for analysis?

RAM model: overview

A simple model to find how an algorithm will perform on a real computer.

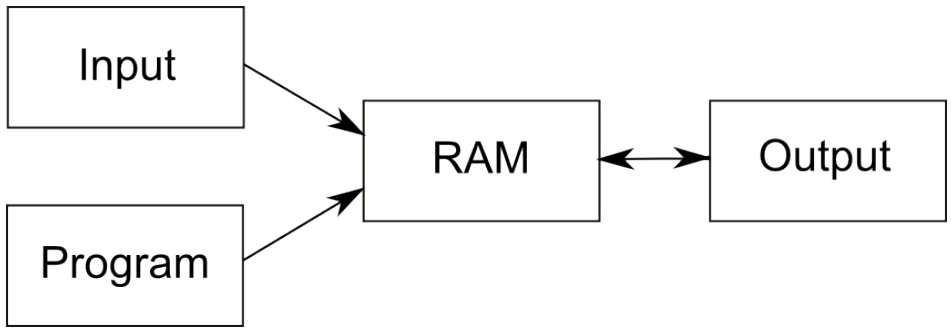
- Mouse
- Keyboard
- Display
- CD-ROM
- Graphics processor
- Working memory
- Processor
- Operating system
- File system
- Input/Output
- Programming capability
- CD-ROM

RAM model: overview

A simple model to find how an algorithm will perform on a real computer.

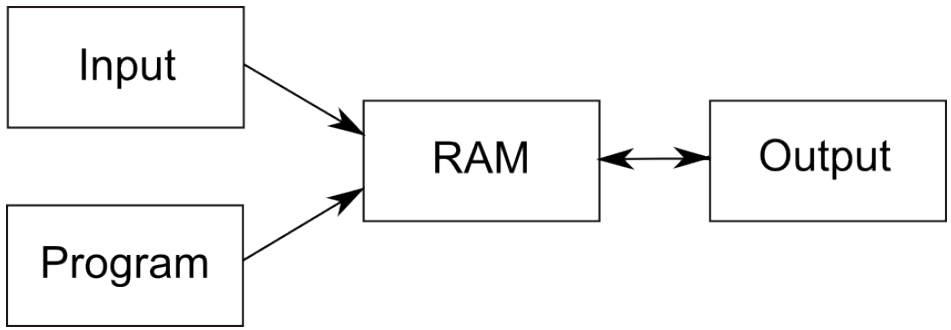
- Mouse
- Keyboard
- Display
- CD-ROM
- Graphics processor
- **Working memory**
- **Processor**
- Operating system
- File system
- Input/Output
- **Programming capability**
- CD-ROM

RAM model: question



- Simple operations (+,-,*,/) take 1 timestep
- Loops comprise a condition plus multiple operations in each iteration
- Read/write to memory is free

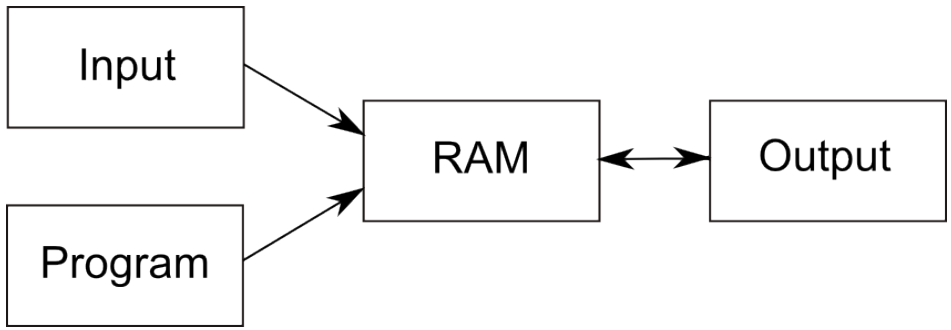
RAM model: question




```
1  a=1
2  b=2*a
```

```
1  s=0
2  while s<10:
3      s+=1
```

RAM model: question

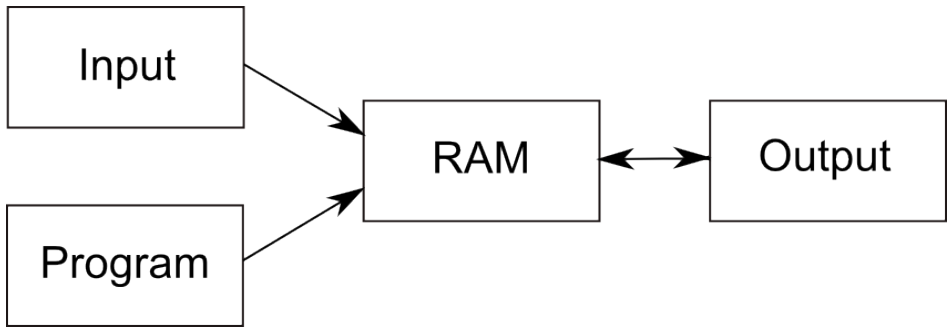



```
1  a=1
2  b=2*a
```

# of timesteps = 1

```
1  s=0
2  while s<10:
3      s+=1
```

RAM model: question




```
1  a=1
2  b=2*a
```

# of timesteps = 1

```
1  s=0
2  while s<10:
3      s+=1
```

# of timesteps = 21

RAM model: the problem

We observe some problem with RAM,

- memory access times depends whether from cache or disk.
- unlimited memory is unrealistic.
- $a + b \neq a \times b$.

However, its simplicity makes it an excellent tool to analyze algorithms.

Algorithm analysis: running time

We consider three simplifications to analyze algorithms:

- without any implementation => RAM model
- taking the worst possible inputs **=> worst-case running time**
- ignoring details => Big-O or Landau notation

Worst case running time: overview

- RAM simply counts operations for any input.
- Worst case running time combines notion of running time with structure of input.
  - determines good or bad algorithm based on all instances of inputs.

Worst case running time: question

Consider an example,

```
1 count = 0
2 for char in str:
3   if char == 'a':
4     count += 1
```

In the example,

- N: length of string
- A: number of 'a's in str

Running time = $\quad \times N + \quad \times A +$

Worst case running time: question

Consider an example,

```
1 count = 0
2 for char in str:
3   if char == 'a':
4     count += 1
```

In the example,

- N: length of string
- A: number of 'a's in str

Running time $= 2 \times N + 1 \times A + 0$ or 1

Worst case running time: the cases

Consider a sample problem, e.g. sorting (arranging items in order)

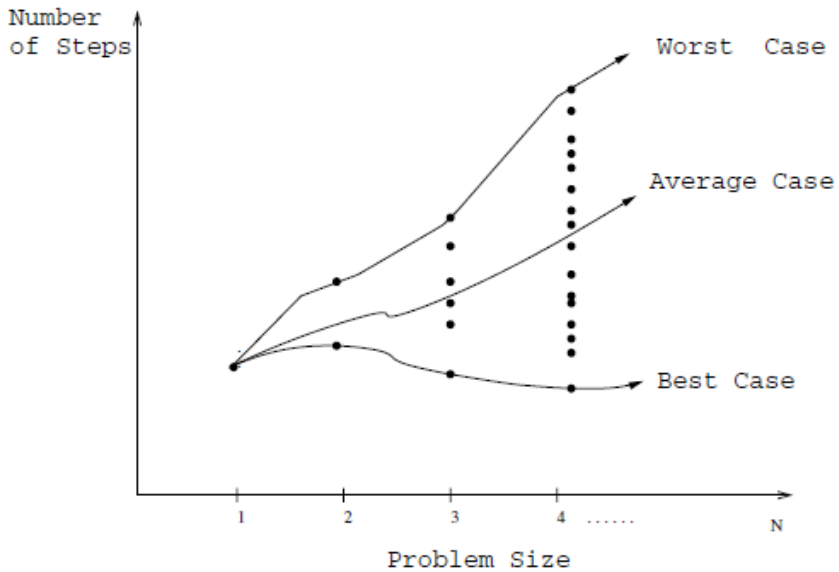


Figure 2.1: Best, worst, and average-case complexity

Worst case running time: the cases

The plot illustrates three functions:

- **Worst-case complexity:**
  - upper bound on cost.
  - determined by most difficult input.
  - provides a guarantee for all inputs.
- **Best-case complexity:**
  - lower bound on cost.
  - determined by easiest input.
  - provides a goal for all inputs.
- **Average-case complexity:**
  - expected cost for random input.
  - requires a model for random input.
  - provides a way to predict performance.

Worst case running time: the challenge

Actual data might not match input running time model?

- Need to understand input to effectively process it.
- Two directions to follow:
  - **Approach 1.** design for the worst case.
  - **Approach 2.** randomize, depend on probabilistic guarantee.

Worst case running time: the outcomes

Goals.

- establish difficulty of a problem.
- develop optimal algorithms.

Approach.

- suppress details in analysis; analyze to within a constant factor.
- eliminate variability in input model by focusing on the worst case.

Optimal algorithm.

- performance guarantee (to within a constant factor) for any input.
- no other algorithm can provide a better performance guarantee.

Worst case running time: question

Q. Which one do you consider is used? And why?

Worst case running time: question

Q. Which one do you consider is used? And why?

- Best case is improbable.
- Average case is difficult to determine.
- Worst-case only guarantees running times.

Worst case running time: question

Consider an example,

```
1 count = 0
2 for char in str:
3   if char == 'a':
4     count += 1
```

In the example consider,

- N: length of string
- A: number of 'a's in str

Question?

$$Worst - case\ running\ time = \quad \times N + \quad \times A +$$

$$Best - case\ running\ time = \quad \times N + \quad \times A +$$

$$Average - case\ running\ time =$$

Worst case running time: question

Consider an example,

```
1 count = 0
2 for char in str:
3   if char == 'a':
4     count += 1
```

In the example consider,

- N: length of string
- A: number of 'a's in str

Question?

$$Worst - case\ running\ time = 3 \times N + 0 \times A + 1$$

$$Best - case\ running\ time = 2 \times N + 0 \times A + 1$$

$$Average - case\ running\ time = ?$$

Worst case running time: the problem

Sample code counts 'ab' substring in string *str*.

```
1 count = 0
2 for i in range(N):
3   if str[i] == 'a':
4     if str[i+1] == 'b':
5       count += 1
```

As a function grows larger,

- counting exact running times becomes tedious.
- determining complexities becomes difficult.

To further simplify we use Big-O notation.

Algorithm analysis: running time (optional)

According to Donald E. Knuth in the book,"The Art of Computer Programming",

- Total running time = sum of cost × frequency for all operations.

The mathematical model for analysis is defined as,

- analyze the program to determine set of operations.
- determine cost and frequency,
  - **frequency.** depends on algorithm, input data.
  - **cost.** depends on machine, compiler.

Algorithm analysis: the power law

The power law. $a \cdot N^b$

- N is the size of the data.
- b is the slope, determines the order of growth.
- a is a constant.

Algorithm analysis: the power law

System independent factors, or frequencies.

- Algorithm.
- Input data.

System dependent factors, or costs.

- Hardware: CPU, memory, cache, ...
- Software: compiler, interpreter, garbage collector, ...
- System: operating system, network, other apps, ...

Algorithm analysis: the power law

System independent factors, or frequencies.

- Algorithm.
- Input data.

} determines exponent b in power law.

System dependent factors, or costs.

- Hardware: CPU, memory, cache, ...
- Software: compiler, interpreter, garbage collector, ...
- System: operating system, network, other apps, ...

Algorithm analysis: the power law

System independent factors, or frequencies.

- Algorithm.
- Input data.

System dependent factors, or costs.

- Hardware: CPU, memory, cache, ...
- Software: compiler, interpreter, garbage collector, ...
- System: operating system, network, other apps, ...

} determines constant a in power law.

Algorithm analysis: mathematical models

In principle, there exist accurate mathematical models for algorithm analysis.
In practice, such models,

- use complicate formulas.
- require advanced mathematics.
- defined and used only for experts.

Algorithm analysis: mathematical models

An example model could be,

$$T(N) = c_1 A + c_2 B + c_3 C + c_4 D + c_5 E$$

$$\begin{cases} A = \text{array access} \\ B = \text{integer add} \\ C = \text{integer compare} \\ D = \text{increment} \\ E = \text{variable assignment} \end{cases}$$

where,

- costs $[c_1, c_2, c_3, c_4, c_5]$ (depend on machine, compiler).
- frequencies $[A, B, C, D, E]$ (depend on algorithm, input).

Algorithm analysis: mathematical models

An example model could be,

$$T(N) = c_1 A + c_2 B + c_3 C + c_4 D + c_5 E$$

$$\begin{cases} A = \text{array access} \\ B = \text{integer add} \\ C = \text{integer compare} \\ D = \text{increment} \\ E = \text{variable assignment} \end{cases}$$

where,

- costs $[c_1, c_2, c_3, c_4, c_5]$ (depend on machine, compiler).
- frequencies $[A, B, C, D, E]$ (depend on algorithm, input).

- Much easier and cheaper to determine approximate running times,
  - can run huge number of experiments
- However, it is difficult to get precise measurements.
- In this course, we will use approximate models, i.e. for matrix multiplication $T(N) \sim cN^3$.

Algorithm analysis: running time

We consider three simplifications to analyze algorithms:

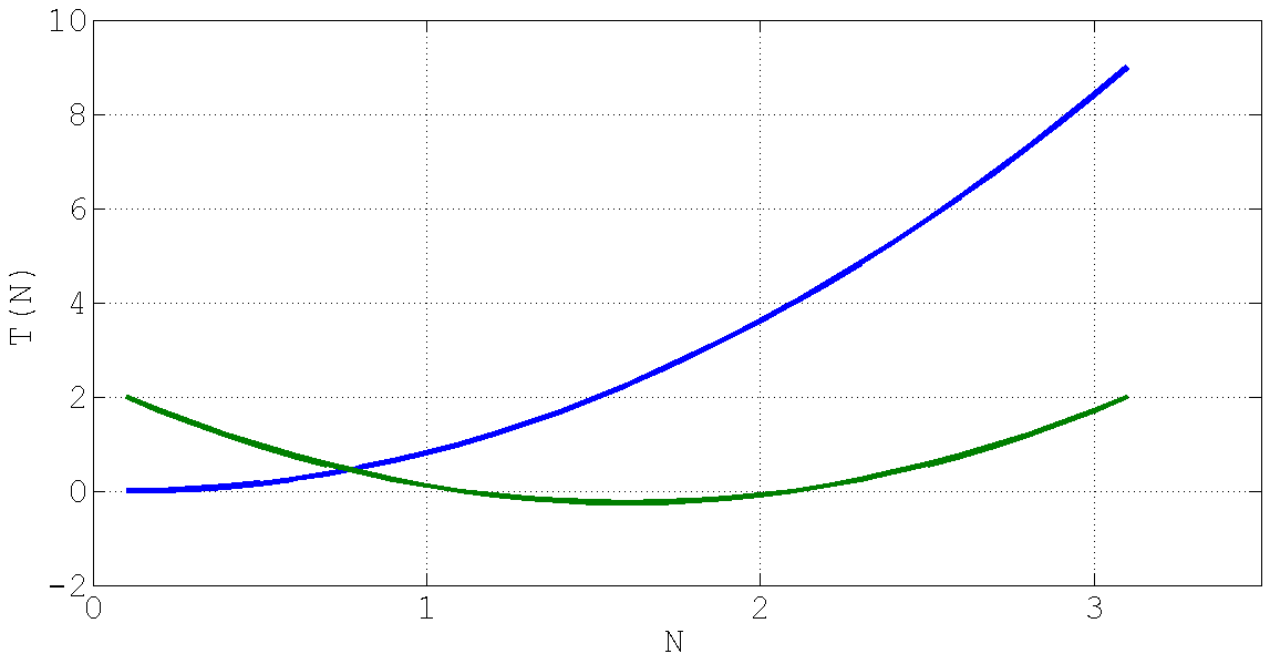
- without any implementation => RAM model
- taking the worst possible inputs => worst-case running time
- ignoring details => Big-O or Landau notation

Algorithm analysis: quadratic growth

Consider the two functions,

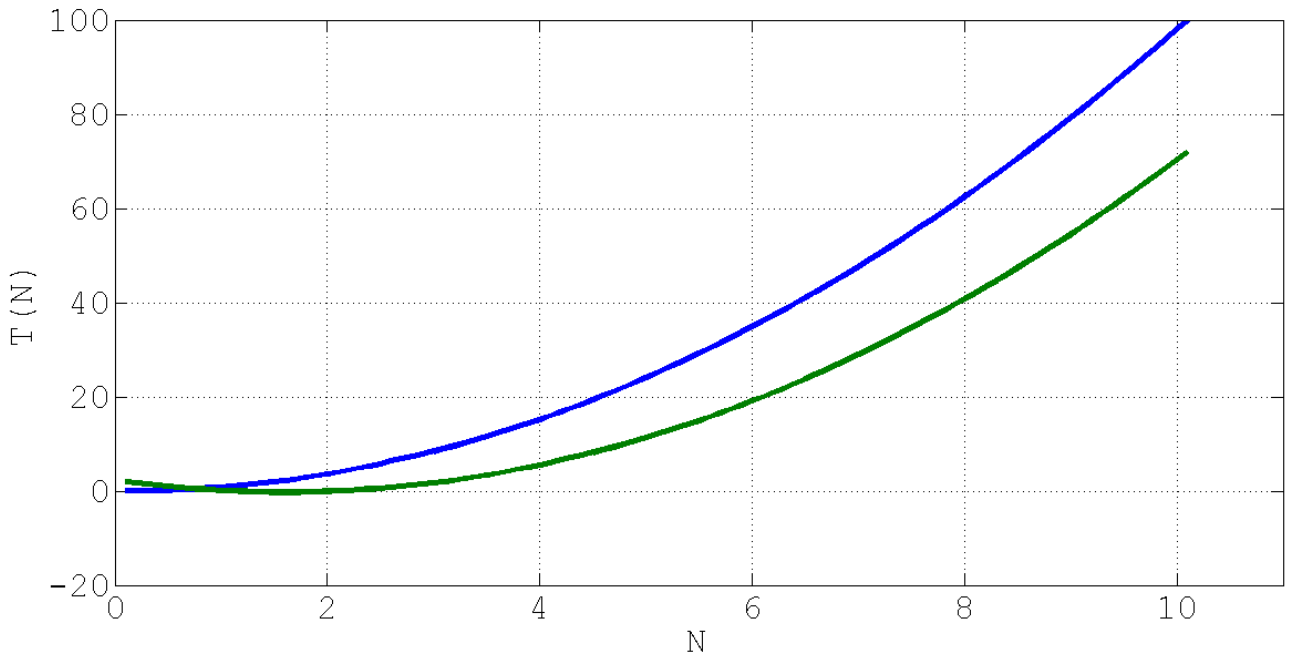
$$f(n) = n^2$$
$$g(n) = n^2 - 3n + 2$$

Around n = 3, they look very different.



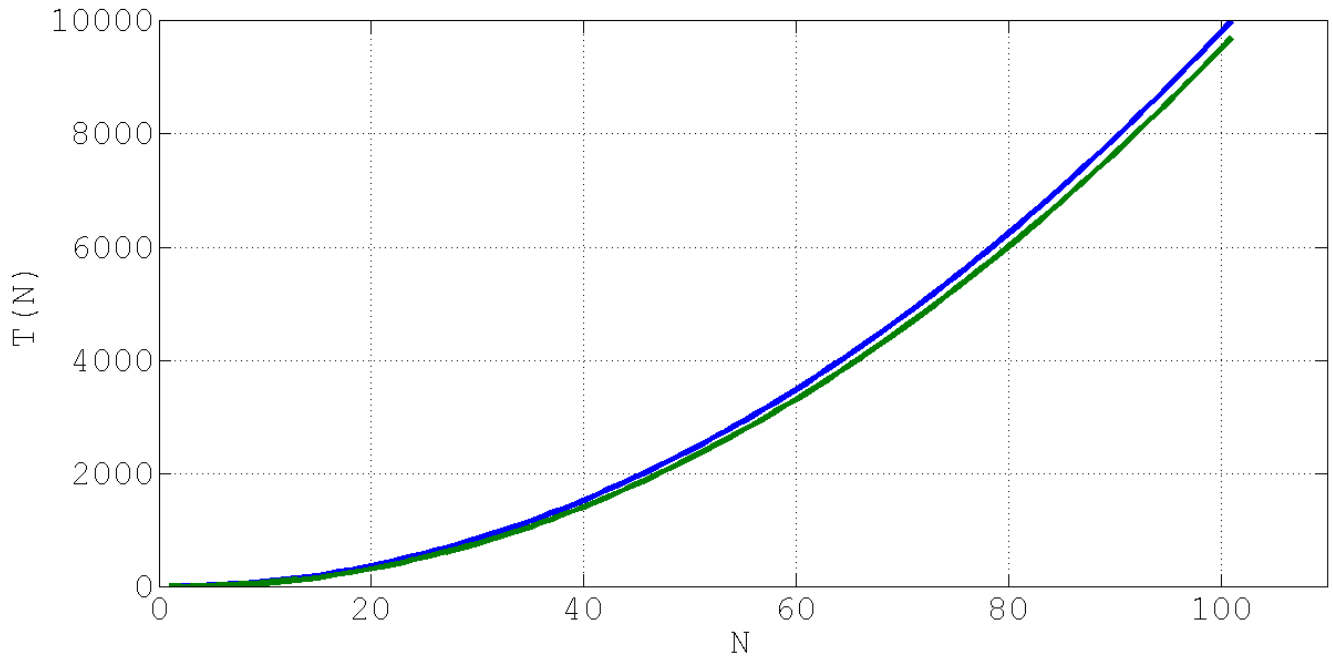
Algorithm analysis: quadratic growth

Around n = 10, they look similar.



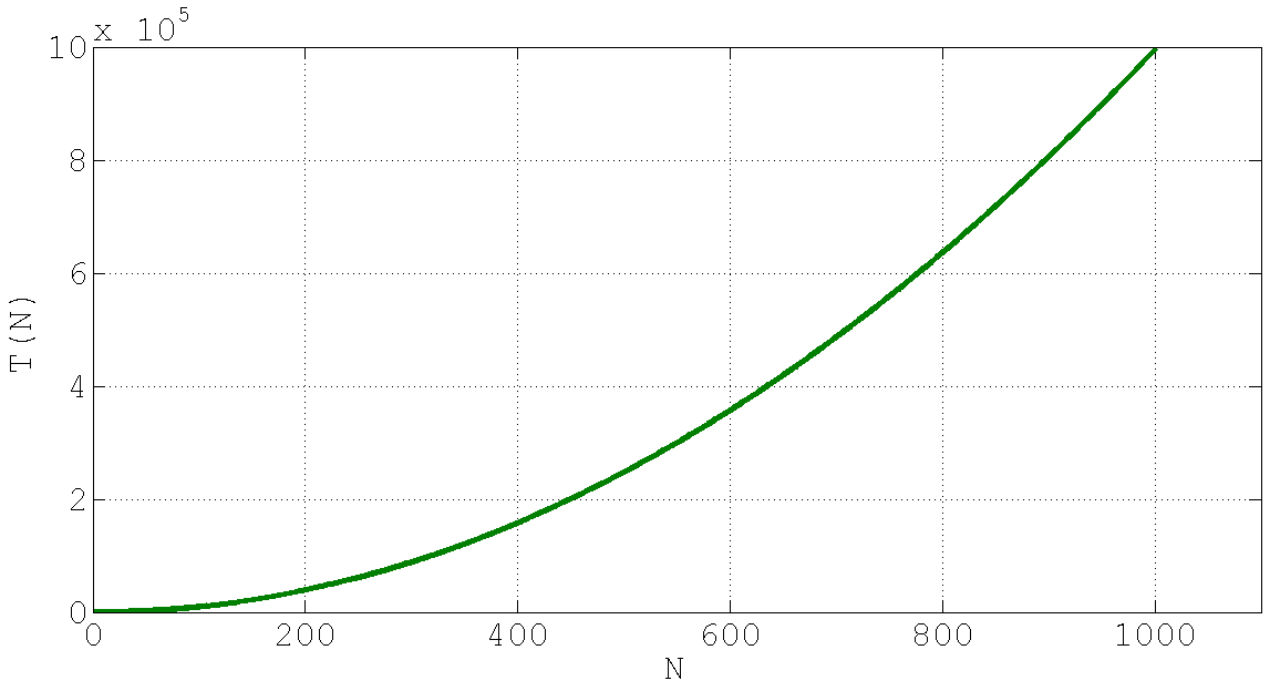
Algorithm analysis: quadratic growth

Around n = 100, they are almost same.



Algorithm analysis: quadratic growth

Around n = 1000, the difference is indistinguishable.



Algorithm analysis: quadratic growth

The absolute difference is large, such that,

$$f(1000) = 1000000$$
$$g(1000) = 997002$$

but the relative difference is very small,

$$\left|\frac{f(1000) - g(1000)}{f(1000)}\right| = -0.002998 < 0.3\%$$

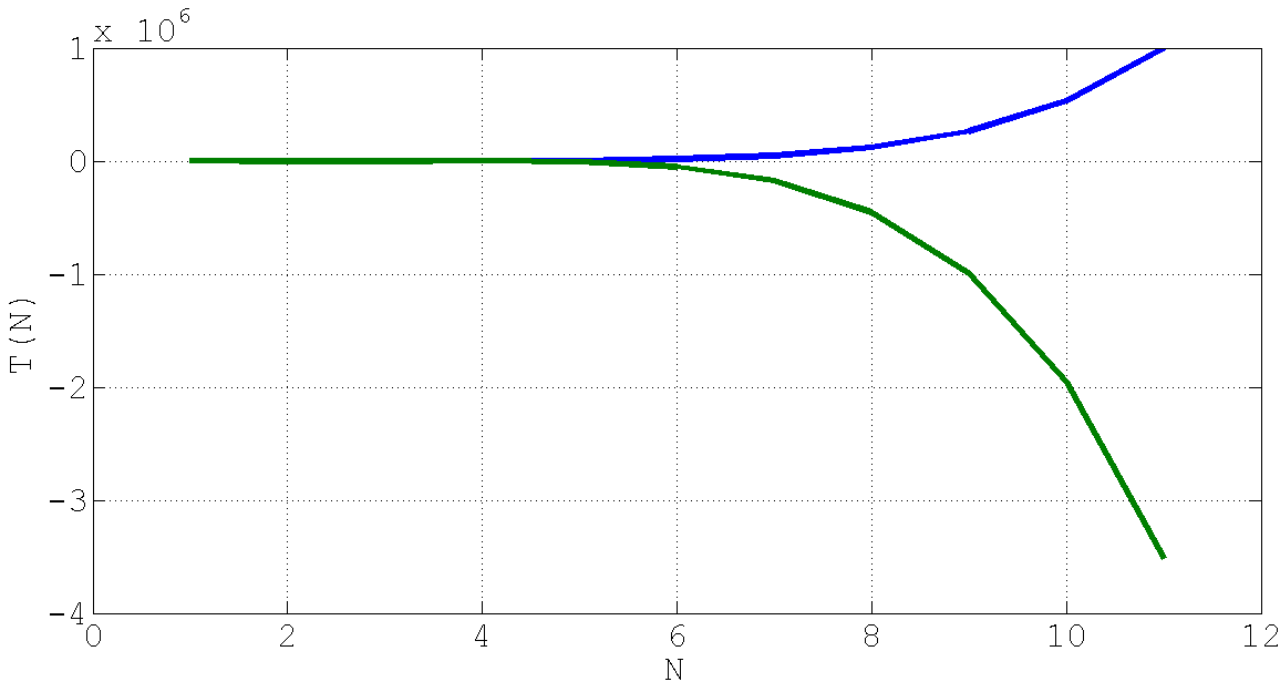
and this difference goes to zero as $n \to \infty$.

Algorithm analysis: polynomial growth

To demonstrate with another example,

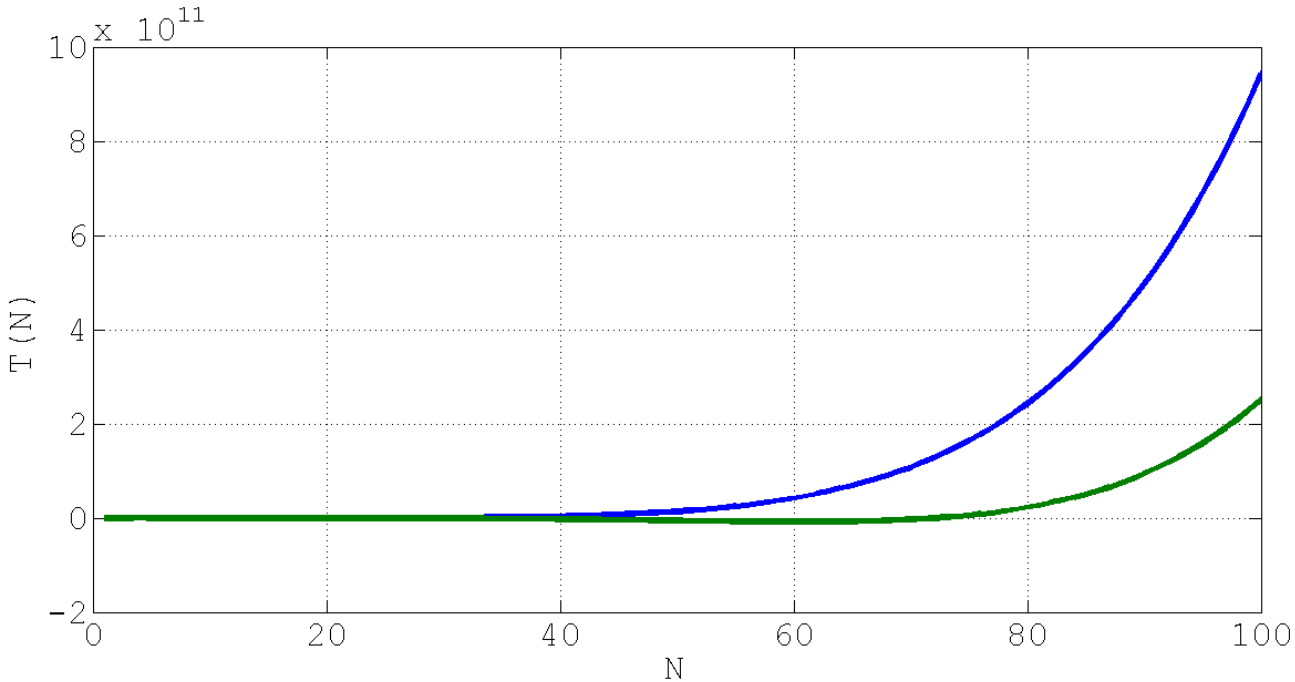
$$f(n) = n^6$$
$$g(n) = n^6 - 76n^5 + 343n^4 - 345n^3 + 45n^2 - 344n$$

Around n = 10, they are very different



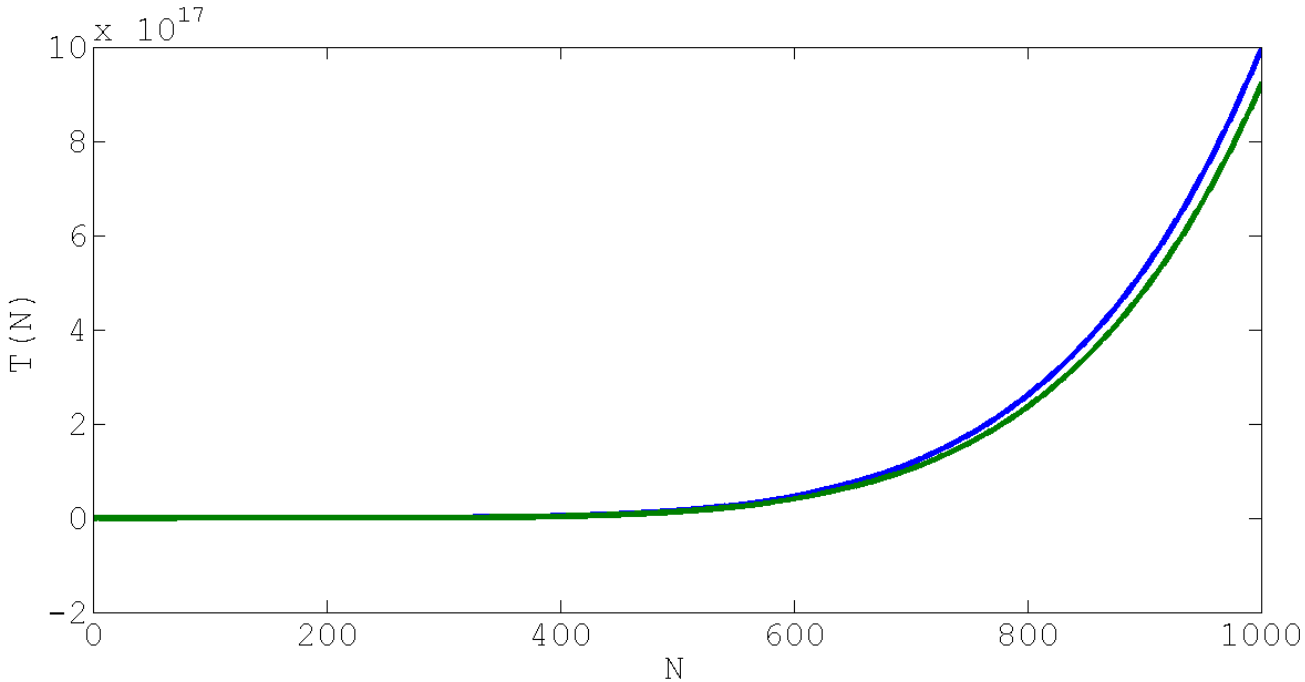
Algorithm analysis: polynomial growth

Around n = 100, they are not similar.



Algorithm analysis: polynomial growth

Around n = 1000, they seem similar.



Algorithm analysis: polynomial growth

Both function pairs of polynomials are similar, they each had the same leading term:

- n^2 in the first case
- n^6 in the second case

In each case, both functions exhibit the same rate of growth,

- one is always proportionally larger than the other.

Big-O notation

Consider two algorithms to solve some problem,

Algorithm 1	**Algorithm 2**
$3n^2 + 2n + 33$	$2^n - 5n + 5$

Here, n is the problem size.

Question

Which one will you prefer?

Big-O notation

Consider two algorithms to solve some problem,

Algorithm 1	**Algorithm 2**
$3n^2 + 2n + 33$	$2^n - 5n + 5$

Here, n is the problem size.

Question

Which one will you prefer?

- When answering, you ignore all unnecessary details.
- Only $n^2 < 2^n$ considered.

Big-O notation

Consider two algorithms to solve some problem,

Algorithm 1	**Algorithm 2**
$3n^2 + 2n + 33$	$2^n - 5n + 5$
f(n)	g(n)

Here, n is the problem size.

- $\exists\ n_0$ and c, where $c \cdot g(n_0) \geq f(n_0)$
- $\forall\ n' > n_0$, where $c \cdot g(n) \geq f(n)$

Big-O notation

Consider two algorithms to solve some problem,

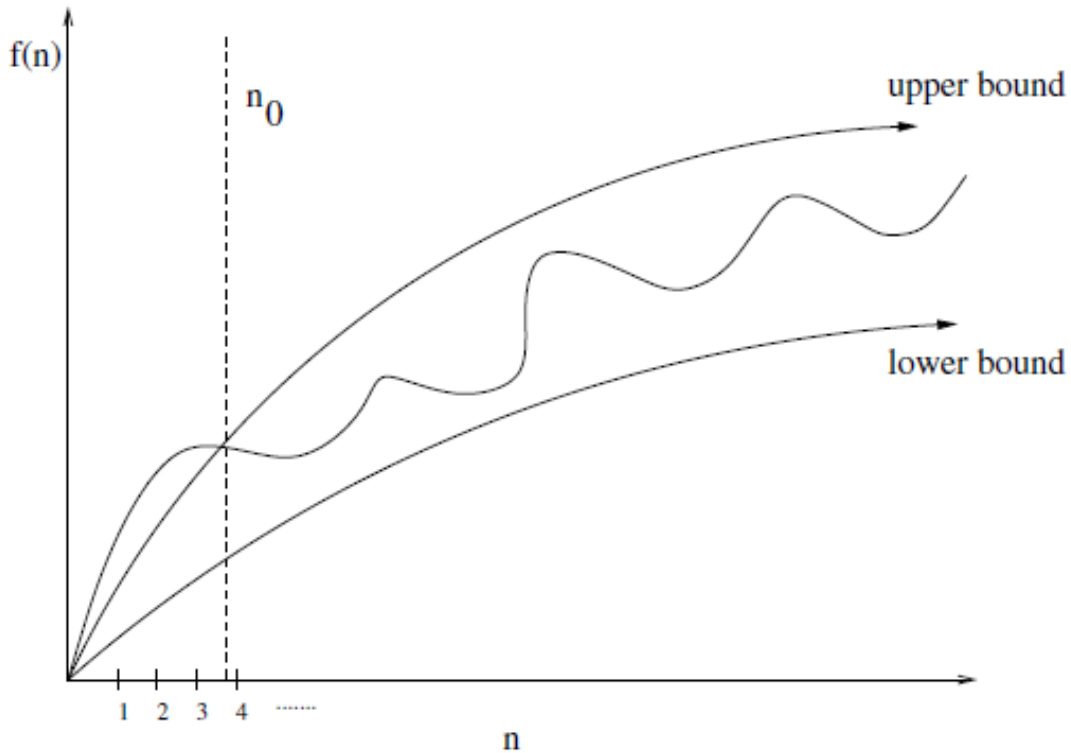
Algorithm 1	**Algorithm 2**
$3n^2 + 2n + 33$	$2^n - 5n + 5$
f(n)	g(n)

Here, n is the problem size.

- $\exists\ n_0$ and c, where $c \cdot g(n_0) \geq f(n_0)$
- $\forall\ n' > n_0$, where $c \cdot g(n) \geq f(n)$

$f(n) \in O(g(n))$, or $f(n) = O(g(n))$
It means the function $f(n)$ has growth rate no greater than $g(n)$.

Big-O notation



Upper and lower bounds valid for $n > n_0$ smooth out the behavior of complex functions.

Big-O notation

Formally,

- $f(n) = O(g(n))$ means $c \cdot g(n)$ is an upper bound on $f(n)$. Thus, there exists some constant c such that $f(n)$ is always $\leq c \cdot g(n)$, for large enough n (i.e. , $n \geq n_0$ for some constant n_0).
- $f(n) = \Omega(g(n))$ means $c \cdot g(n)$ is a lower bound on $f(n)$. Thus there exists some constant c such that $f(n)$ is always $\geq c \cdot g(n)$, for all $n \geq n_0$.
- $f(n) = \Theta(g(n))$ means $c_1 \cdot g(n)$ is an upper bound on $f(n)$ and $c_2 \cdot g(n)$ is a lower bound on $f(n)$, for all $n \geq n_0$. Thus there exist constants c_1 and c_2 such that $f(n) \leq c_1 \cdot g(n)$ and $f(n) \geq c_2 \cdot g(n)$. This means that $g(n)$ provides a nice, tight bound on $f(n)$.

Big-O notation

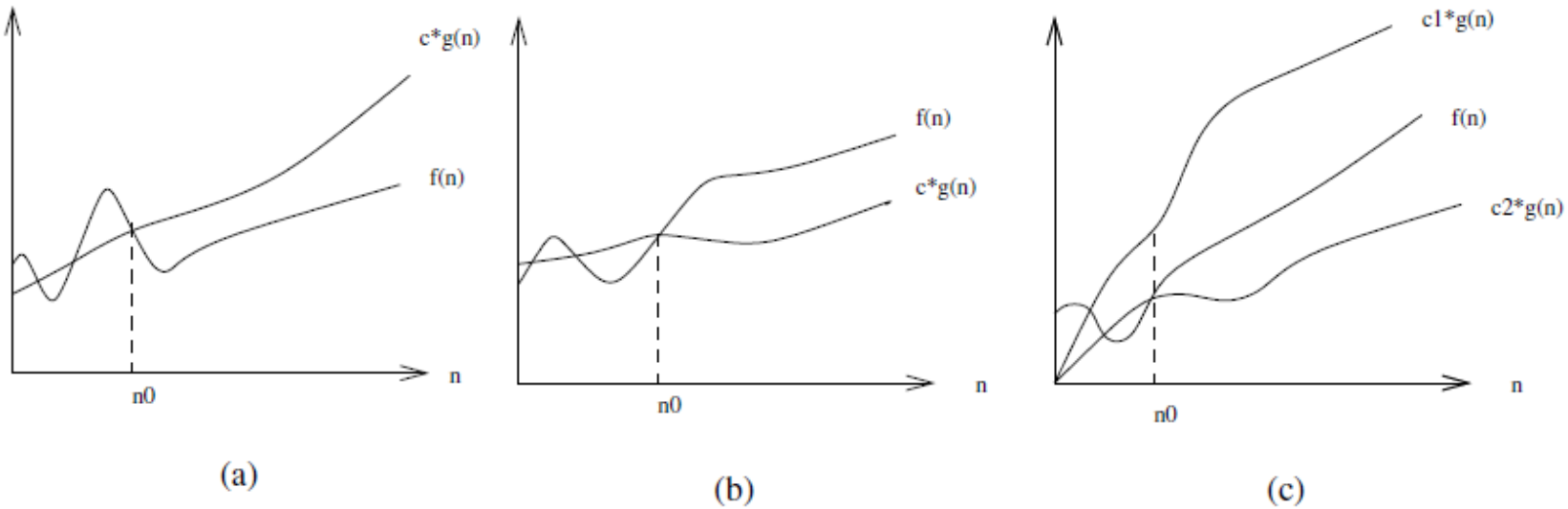


Figure 2.3: Illustrating the big (a) O, (b) Ω, and (c) Θ notations

Big-O notation

$f(n) \in g(n)$
$g(n)$ grows at least as fast as $f(n)$

Big-O notation

$f(n) \in g(n)$
$g(n)$ grows at least as fast as $f(n)$

$5n + 2 \in O(n)$

Big-O notation

$f(n) \in g(n)$
$g(n)$ grows at least as fast as $f(n)$

$5n + 2 \in O(n)$
$12n^2 - 10 \in O(n^2)$

Big-O notation

$f(n) \in g(n)$
$g(n)$ grows at least as fast as $f(n)$

$5n + 2 \in O(n)$
$12n^2 - 10 \in O(n^2)$
$3^n - 3n^8 + 23 \in O(3^n)$

Big-O notation

$f(n) \in g(n)$
$g(n)$ grows at least as fast as $f(n)$

$5n + 2 \in O(n)$
$12n^2 - 10 \in O(n^2)$
$3^n - 3n^8 + 23 \in O(3^n)$
$3^n \cdot n^3 - 3n^8 + 23 \in O(3^n \cdot n^3)$

Big-O notation

$f(n) \in g(n)$
$g(n)$ grows at least as fast as $f(n)$

$5n + 2 \in O(n)$ => $5n + 2 \in O(n^2)$
$12n^2 - 10 \in O(n^2)$ => $12n^2 - 10 \in O(n^3)$
$3^n - 3n^8 + 23 \in O(3^n)$
$3^n \cdot n^3 - 3n^8 + 23 \in O(3^n \cdot n^3)$

However, it is preferable to keep the bound as tight as possible.

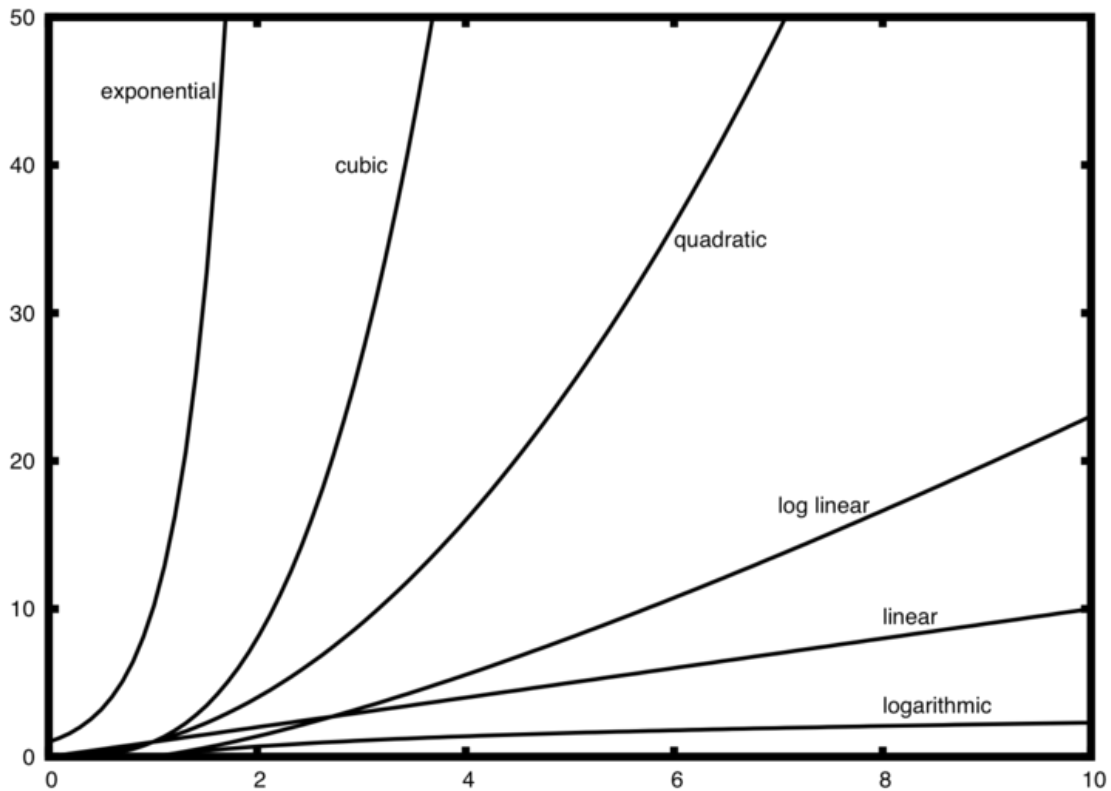
Big-O notation

The most common Big-O functions are given names:

- $O(1)$ => constant
- $O(ln(n))$ => logarithmic
- $O(n)$ => linear
- $O(nln(n))$ => log linear
- $O(n^2)$ => quadratic
- $O(n^3)$ => cubic
- $2^n, e^n, 4^n$ => exponential

Big-O notation

Plot for common Big-O functions:



Big-O notation

Sample code counts 'ab' substring in string *str*.

```
1 count = 0
2 for char in str:
3   if char == 'a':
4     count += 1
```

Using Big-O simplification,

- We require no counting of operations.
- It can be simply written as, $O(n)$, a linear algorithm.

Big-O notation: problem

Given the following code fragment, what is its Big-O running time?

```
1 test = 0
2 for i in range(n):
3     for j in range(n):
4         test = test + i * j
```

a) $O(n)$
b) $O(n^2)$
c) $O(logn)$
d) $O(n^3)$

Big-O notation: problem

Given the following code fragment what is its Big-O running time?

```
1  test = 0
2  for i in range(n):
3      test = test + 1
4
5  for j in range(n):
6      test = test - 1
```

a) $O(n)$
b) $O(n^2)$
c) $O(logn)$
d) $O(n^3)$

Big-O notation: problem

Given the following code fragment what is its Big-O running time?

```
1  i = n
2  while i > 0:
3      k = 2 + 2
4      i = i / 2
5  }
```

a) $O(n)$
b) $O(n^2)$
c) $O(logn)$
d) $O(n^3)$

Big-O notation: problem

Running time of a sorting algorithm.

- Computer A:
  - Speed is one billion instructions per second.
  - Uses Insertion Sort and requires $2n^2$ instructions to sort n numbers.
- Computer B:
  - Speed is 10 million instructions per second.
  - Uses Merge Sort and requires $50nlgn$ instructions.

Big-O notation: problem

Computer A takes.

$$\frac{2 \cdot (10^7)^2 instructions}{10^{10} instructions/second} = 20,000 secs (> 5.5 hrs)$$

Computer B takes.

$$\frac{50 \cdot (10^7) lg 10^7 instructions}{10^7 instructions/second} = 1,163 secs (< 20 mins)$$

Computer B runs 17 times faster than computer A,

- using an algorithm that grows slowly.
- even if uses a slow processor.

Summary

We will use Landau symbols to describe the complexity of algorithms.

- e.g., adding a list of n numbers in linear time $\Theta(n)$ algorithm.

An algorithm is said to have polynomial time complexity if its runtime may be described by $O(n^d)$ for some fixed $d \geq 0$.

- We will consider such algorithms to be efficient.

Problems that have no known polynomial-time algorithms are said to be intractable.

- e.g. Traveling salesman problem. find the shortest path that visits n cities.
- Best run time: $O(n^2 2^n)$

In general, you don't want to implement exponential-time or exponential-memory algorithms.

Guiding Principles #1

Worst–case analysis. running time bound holds for every input of length n.

- particularly appropriate for "general-purpose" routines

As opposed to

- "average case" analysis
- benchmarks

BONUS. worst case usually easier to analyze.

Guiding Principles #2

Won't pay much attention to constant factors, lowerForder terms
Justifications.

- way easier
- constants depend on architecture / compiler / programmer anyways
- lose very little predictive power (as we'll see)

Guiding Principles #3

Asymptotic Analysis. focus on running time for large input sizes n

- e.g. $6nlog_2 + 6n$ "better than" $\frac{1}{2}n^2$

Justification. only big problems are interesting!

What Is a "Fast" Algorithm?

Adopt these three biases as guiding principles

- fast algorithm worst-case running time grows slowly with input size

Goal. want as close to linear (O(n)) as possible