Lecture 5: Order of Growth Classifications

BT 3051 – Data Structures and Algorithms for Biology

Karthik Raman
Department of Biotechnology
Bhupat and Jyoti Mehta School of Biosciences
Indian Institute of Technology Madras

Introduction

What is a fast algorithm?

Consider the following problems:

1. **Genome Assembly Problem.** Find the shortest common super-string of a set of sequences (*reads*): given strings $\{s_1, s_2, \ldots, s_n\}$; find the super-string T such that every s_i is a sub-string of T
2. **Alignment Problems.** How do you align two protein sequences? structures? graphs?
3. **Parameter Estimation Problem.** Given a set of data $\mathcal{D}$, find the set of model parameters Θ that minimises model error $\mathcal{E}$, with respect to $\mathcal{D}$. Assume that $\theta_i \in \{10^{-4}, 10^3\}$.
4. **8 Queens Problem.** How do you place 8 queens on a chessboard, so that no two queens threaten one another?

What is a fast algorithm?

Consider the following problems:

1. **Genome Assembly Problem.** Find the shortest common super-string of a set of sequences (*reads*): given strings $\{s_1, s_2, \ldots, s_n\}$; find the super-string T such that every s_i is a sub-string of T
2. **Alignment Problems.** How do you align two protein sequences? structures? graphs?
3. **Parameter Estimation Problem.** Given a set of data $\mathcal{D}$, find the set of model parameters Θ that minimises model error $\mathcal{E}$, with respect to $\mathcal{D}$. Assume that $\theta_i \in \{10^{-4}, 10^3\}$.
4. **8 Queens Problem.** How do you place 8 queens on a chessboard, so that no two queens threaten one another?

What is a fast algorithm?

Consider the following problems:

1. **Genome Assembly Problem.** Find the shortest common super-string of a set of sequences (*reads*): given strings $\{s_1, s_2, \ldots, s_n\}$; find the super-string T such that every s_i is a sub-string of T
2. **Alignment Problems.** How do you align two protein sequences? structures? graphs?
3. **Parameter Estimation Problem.** Given a set of data $\mathcal{D}$, find the set of model parameters Θ that minimises model error $\mathcal{E}$, with respect to $\mathcal{D}$. Assume that $\theta_i \in \{10^{-4}, 10^3\}$.
4. **8 Queens Problem.** How do you place 8 queens on a chessboard, so that no two queens threaten one another?

What is a fast algorithm?

Consider the following problems:

1. **Genome Assembly Problem.** Find the shortest common super-string of a set of sequences (*reads*): given strings $\{s_1, s_2, \ldots, s_n\}$; find the super-string T such that every s_i is a sub-string of T
2. **Alignment Problems.** How do you align two protein sequences? structures? graphs?
3. **Parameter Estimation Problem.** Given a set of data $\mathcal{D}$, find the set of model parameters Θ that minimises model error $\mathcal{E}$, with respect to $\mathcal{D}$. Assume that $\theta_i \in \{10^{-4}, 10^3\}$.
4. **8 Queens Problem.** How do you place 8 queens on a chessboard, so that no two queens threaten one another?

Brute Force

- Involves checking every possible solution to a problem
- Typically takes exponential time ($\sim 2^n$ or even $\sim n!$)
- Often useless in practice, esp. for large problems

Aside: See http://en.wikipedia.org/wiki/Four_color_theorem

Polynomial algorithms

- Polynomial algorithms scale better with input size
- Desirable: with doubling of input size, algorithm slows down by some constant factor c
- Polynomial algorithms are usually referred to as 'efficient'
- Usually work well; have low constants and exponents
- Importantly, breaking down the exponential barrier exposes interesting aspects of problem structure

Except…

- If the polynomial algorithm has pathologically high constants!
- …or exponents!
- e.g. $10n^{100}$ vs $n^{1+\lg n}$
- Some exponential-time algorithms are used widely in practice because the worst-case instances seem to be rare

Polynomial algorithms

- Polynomial algorithms scale better with input size
- Desirable: with doubling of input size, algorithm slows down by some constant factor c
- Polynomial algorithms are usually referred to as 'efficient'
- Usually work well; have low constants and exponents
- Importantly, breaking down the exponential barrier exposes interesting aspects of problem structure

Except…

- If the polynomial algorithm has pathologically high constants!
- …or exponents!
- e.g. $10n^{100}$ vs $n^{1.1 \lg n}$
- Some exponential-time algorithms are used widely in practice because the worst-case instances seem to be rare

Polynomial algorithms

- Polynomial algorithms scale better with input size
- Desirable: with doubling of input size, algorithm slows down by some constant factor c
- Polynomial algorithms are usually referred to as 'efficient'
- Usually work well; have low constants and exponents
- Importantly, breaking down the exponential barrier exposes interesting aspects of problem structure

Except…

- If the polynomial algorithm has pathologically high constants!
- …or exponents!
- e.g. $10n^{100}$ vs $n^{1+\lg n}$
- Some exponential-time algorithms are used widely in practice because the worst-case instances seem to be rare

Polynomial algorithms

- Polynomial algorithms scale better with input size
- Desirable: with doubling of input size, algorithm slows down by some constant factor c
- Polynomial algorithms are usually referred to as 'efficient'
- Usually work well; have low constants and exponents
- Importantly, breaking down the exponential barrier exposes interesting aspects of problem structure

Except...

- If the polynomial algorithm has pathologically high constants!
- ...or exponents!
- e.g. $10n^{100}$ vs $n^{1+\lg n}$
- Some exponential-time algorithms are used widely in practice because the worst-case instances seem to be rare

Polynomial algorithms

- Polynomial algorithms scale better with input size
- Desirable: with doubling of input size, algorithm slows down by some constant factor c
- Polynomial algorithms are usually referred to as 'efficient'
- Usually work well; have low constants and exponents
- Importantly, breaking down the exponential barrier exposes interesting aspects of problem structure

Except...

- If the polynomial algorithm has pathologically high constants!
- ...or exponents!
- e.g. $10n^{100}$ vs $n^{1+\lg n}$
- Some exponential-time algorithms are used widely in practice because the worst-case instances seem to be rare

Polynomial algorithms

- Polynomial algorithms scale better with input size
- Desirable: with doubling of input size, algorithm slows down by some constant factor c
- Polynomial algorithms are usually referred to as 'efficient'
- Usually work well; have low constants and exponents
- Importantly, breaking down the exponential barrier exposes interesting aspects of problem structure

Except...

- If the polynomial algorithm has pathologically high constants!
- ...or exponents!
- e.g. $10n^{100}$ vs $n^{1+\lg n}$
- Some exponential-time algorithms are used widely in practice because the worst-case instances seem to be rare

Polynomial algorithms

- Polynomial algorithms scale better with input size
- Desirable: with doubling of input size, algorithm slows down by some constant factor c
- Polynomial algorithms are usually referred to as 'efficient'
- Usually work well; have low constants and exponents
- Importantly, breaking down the exponential barrier exposes interesting aspects of problem structure

Except...

- If the polynomial algorithm has pathologically high constants!
- ...or exponents!
- e.g. $10n^{100}$ vs $n^{1+lg\ n}$
- Some exponential-time algorithms are used widely in practice because the worst-case instances seem to be rare

Polynomial algorithms

- Polynomial algorithms scale better with input size
- Desirable: with doubling of input size, algorithm slows down by some constant factor c
- Polynomial algorithms are usually referred to as 'efficient'
- Usually work well; have low constants and exponents
- Importantly, breaking down the exponential barrier exposes interesting aspects of problem structure

Except...

- If the polynomial algorithm has pathologically high constants!
- ...or exponents!
- e.g. $10n^{100}$ vs $n^{1+lg\ n}$
- Some exponential-time algorithms are used widely in practice because the worst-case instances seem to be rare

Polynomial algorithms

- Polynomial algorithms scale better with input size
- Desirable: with doubling of input size, algorithm slows down by some constant factor c
- Polynomial algorithms are usually referred to as 'efficient'
- Usually work well; have low constants and exponents
- Importantly, breaking down the exponential barrier exposes interesting aspects of problem structure

Except...

- If the polynomial algorithm has pathologically high constants!
- ...or exponents!
- e.g. $10n^{100}$ vs $n^{1+lg\ n}$
- Some exponential-time algorithms are used widely in practice because the worst-case instances seem to be rare

Polynomial algorithms

- Polynomial algorithms scale better with input size
- Desirable: with doubling of input size, algorithm slows down by some constant factor c
- Polynomial algorithms are usually referred to as 'efficient'
- Usually work well; have low constants and exponents
- Importantly, breaking down the exponential barrier exposes interesting aspects of problem structure

Except...

- If the polynomial algorithm has pathologically high constants!
- ...or exponents!
- e.g. $10n^{100}$ vs $n^{1+lg\ n}$
- Some exponential-time algorithms are used widely in practice because the worst-case instances seem to be rare

Worst-case analysis

- Running time **guarantees** for *any input* of size n!
- Captures efficiency, in practice

Other analyses

- Probabilistic (expected running time of a randomised algo)
- Amortized complexity (worst-case running time for any sequence of n operations)
- Average-case (expected running time for a random input of size n)
- ...

Worst-case analysis

- Running time **guarantees** for *any input* of size n!
- Captures efficiency, in practice

Other analyses

- Probabilistic (expected running time of a randomised algo)
- Amortized complexity (worst-case running time for any sequence of n operations)
- Average-case (expected running time for a random input of size n)
- ...

Worst-case analysis

- Running time **guarantees** for *any input* of size n!
- Captures efficiency, in practice

Other analyses

- Probabilistic (expected running time of a randomised algo)
- Amortized complexity (worst-case running time for any sequence of n operations)
- Average-case (expected running time for a random input of size n)
- ...

Worst-case analysis

- Running time **guarantees** for *any input* of size n!
- Captures efficiency, in practice

Other analyses

- Probabilistic (expected running time of a randomised algo)
- Amortized complexity (worst-case running time for any sequence of n operations)
- Average-case (expected running time for a random input of size n)
- ...

Worst-case analysis

- Running time **guarantees** for *any input* of size n!
- Captures efficiency, in practice

Other analyses

- Probabilistic (expected running time of a randomised algo)
- Amortized complexity (worst-case running time for any sequence of n operations)
- Average-case (expected running time for a random input of size n)
- ...

Worst-case analysis

- Running time **guarantees** for *any input* of size n!
- Captures efficiency, in practice

Other analyses

- Probabilistic (expected running time of a randomised algo)
- Amortized complexity (worst-case running time for any sequence of n operations)
- Average-case (expected running time for a random input of size n)
- ...

Asymptotic Notations

Big-O Notation

Definition

Let $f(n)$ and $g(n)$ be functions mapping positive integers to positive real numbers. We say that $f(n)$ is $O(g(n))$ if there is a real constant $c > 0$ and an integer constant $n_0 \geq 1$ such that $f(n) \leq cg(n)$, for $n \geq n_0$.

Big-O Notation

Definition

Let $f(n)$ and $g(n)$ be functions mapping positive integers to positive real numbers. We say that $f(n)$ is $O(g(n))$ if there is a real constant $c > 0$ and an integer constant $n_0 \geq 1$ such that $f(n) \leq cg(n)$, for $n \geq n_0$.

This definition is often referred to as the "big-O" notation, for it is sometimes pronounced as "$f(n)$ is big-O of $g(n)$."

Big-O Notation

Definition

Let $f(n)$ and $g(n)$ be functions mapping positive integers to positive real numbers. We say that $f(n)$ is $O(g(n))$ if there is a real constant $c > 0$ and an integer constant $n_0 \geq 1$ such that $f(n) \leq cg(n)$, for $n \geq n_0$.

This definition is often referred to as the "big-O" notation, for it is sometimes pronounced as "$f(n)$ is big-O of $g(n)$."

Actually, $O(g(n))$ is a set of functions, but notation is usually (mis-)written as $f(n) = O(g(n))$, rather than $f(n) \in O(g(n))$.

Asymptotic Notations

$f(n) = \Theta(g(n))$ — Big Theta

- $f(n)$ and $g(n)$ have the *same* order of magnitude
- Tight bound: classify algorithms
- e.g. $n^2, 100n^2, n^2 + 10 lg\, n \in \Theta(n^2)$

$f(n) = O(g(n))$ — Big O

- order of magnitude of $f(n)$ is *less than or equal to* $g(n)$
- Upper bound, e.g. $\Theta(n^2)$ and smaller
- e.g. $100n^2, 100n, n lg\, n + 10n \in O(n^2)$

$f(n) = \Omega(g(n))$ — Big Omega

- order of magnitude of $f(n)$ is *greater or equal to* $g(n)$
- Lower bound, e.g. $\Theta(n^2)$ and larger
- e.g. $n^5, 100n^2, n^3 + n^2 lg\, n \in \Omega(n^2)$

Asymptotic Notations

$f(n) = \Theta(g(n))$ — Big Theta

- $f(n)$ and $g(n)$ have the *same* order of magnitude
- Tight bound: classify algorithms
- e.g. $n^2, 100n^2, n^2 + 10lg\, n \in \Theta(n^2)$

$f(n) = O(g(n))$ — Big O

- order of magnitude of $f(n)$ is *less than or equal to* $g(n)$
- Upper bound, e.g. $\Theta(n^2)$ and smaller
- e.g. $100n^2, 100n, nlg\, n + 10n \in O(n^2)$

$f(n) = \Omega(g(n))$ — Big Omega

- order of magnitude of $f(n)$ is *greater or equal to* $g(n)$
- Lower bound, e.g. $\Theta(n^2)$ and larger
- e.g. $n^5, 100n^2, n^3 + n^2 lg\, n \in \Omega(n^2)$

Asymptotic Notations

$f(n) = \Theta(g(n))$ — Big Theta

- $f(n)$ and $g(n)$ have the *same* order of magnitude
- Tight bound: classify algorithms
- e.g. $n^2, 100n^2, n^2 + 10 lg\, n \in \Theta(n^2)$

$f(n) = O(g(n))$ — Big O

- order of magnitude of $f(n)$ is *less than or equal to* $g(n)$
- Upper bound, e.g. $\Theta(n^2)$ and smaller
- e.g. $100n^2, 100n, nlg\, n + 10n \in O(n^2)$

$f(n) = \Omega(g(n))$ — Big Omega

- order of magnitude of $f(n)$ is *greater or equal to* $g(n)$
- Lower bound, e.g. $\Theta(n^2)$ and larger
- e.g. $n^5, 100n^2, n^3 + n^2 lg\, n \in \Omega(n^2)$

Asymptotic Notations

Remember, $O(g(n))$ is a set of functions, but notation is usually written as $T(n) = O(g(n))$, rather than $T(n) \in O(g(n))$

Leading Term Approximation

- Gives an approximate idea of performance
- $10n^2$, $10n^2 + 100n \lg n$, $10n^2 + 10n + 1000 \rightarrow \sim 10n^2$

Asymptotic Notations

Remember, $O(g(n))$ is a set of functions, but notation is usually written as $T(n) = O(g(n))$, rather than $T(n) \in O(g(n))$

We will (try to) stick to ~ (tilde) notation in this course

...

Leading Term Approximation

- Gives an approximate idea of performance
- $10n^2$, $10n^2 + 100n\ lg\ n$, $10n^2 + 10n + 1000 \rightarrow \sim 10n^2$

Asymptotic Notations

Remember, $O(g(n))$ is a set of functions, but notation is usually written as $T(n) = O(g(n))$, rather than $T(n) \in O(g(n))$

We will (try to) stick to ~ (tilde) notation in this course

...

Leading Term Approximation

- Gives an approximate idea of performance
- $10n^2$, $10n^2 + 100n \lg n$, $10n^2 + 10n + 1000 \rightarrow \sim 10n^2$

Examples

From Steven Skiena

Big O

- $3n^2 - 100n + 6 = O(n^2)$
- $3n^2 - 100n + 6 = O(n^3)$
- $3n^2 - 100n + 6 \neq O(n)$

Big Omega: Ω

- $3n^2 - 100n + 6 = \Omega(n^2)$
- $3n^2 - 100n + 6 \neq \Omega(n^3)$
- $3n^2 - 100n + 6 = \Omega(n)$

Big Theta: Θ

- $3n^2 - 100n + 6 = \Theta(n^2)$
- $3n^2 - 100n + 6 \neq \Theta(n^3)$
- $3n^2 - 100n + 6 \neq \Theta(n)$

Theory of Algorithms

- Important field of computer science
- Concerns methods to
  - construct algorithms and
  - analyse algorithms mathematically
  - for correctness
  - and efficiency (e.g., running time and space used)
- Typically establish the difficulty of a problem, and develop optimal algorithms
- Focus on worst cases and provide performance guarantees

Theory of Algorithms

- Important field of computer science
- Concerns methods to
  - construct algorithms and
  - analyse algorithms mathematically,
  - for correctness
  - and efficiency (e.g., running time and space used)
- Typically establish the difficulty of a problem, and develop optimal algorithms
- Focus on worst cases and provide performance guarantees

Theory of Algorithms

- Important field of computer science
- Concerns methods to
  - construct algorithms and
  - analyse algorithms mathematically,
  - for correctness
  - and efficiency (e.g., running time and space used)
- Typically establish the difficulty of a problem, and develop optimal algorithms
- Focus on worst cases and provide performance guarantees

Theory of Algorithms

- Important field of computer science
- Concerns methods to
  - construct algorithms and
  - analyse algorithms mathematically,
  - for correctness
  - and efficiency (e.g., running time and space used)
- Typically establish the difficulty of a problem, and develop optimal algorithms
- Focus on worst cases and provide performance guarantees

Theory of Algorithms

- Important field of computer science
- Concerns methods to
  - construct algorithms and
  - analyse algorithms mathematically,
  - for correctness
  - and efficiency (e.g., running time and space used)
- Typically establish the difficulty of a problem, and develop optimal algorithms
- Focus on worst cases and provide performance guarantees

Theory of Algorithms

- Important field of computer science
- Concerns methods to
  - construct algorithms and
  - analyse algorithms mathematically,
  - for correctness
  - and efficiency (e.g., running time and space used)
- Typically establish the difficulty of a problem, and develop optimal algorithms
- Focus on worst cases and provide performance guarantees

Theory of Algorithms

- Important field of computer science
- Concerns methods to
  - construct algorithms and
  - analyse algorithms mathematically,
  - for correctness
  - and efficiency (e.g., running time and space used)
- Typically establish the difficulty of a problem, and develop optimal algorithms
- Focus on worst cases and provide performance guarantees

Theory of Algorithms

- Important field of computer science
- Concerns methods to
  - construct algorithms and
  - analyse algorithms mathematically,
  - for correctness
  - and efficiency (e.g., running time and space used)
- Typically establish the difficulty of a problem, and develop optimal algorithms
- Focus on worst cases and provide performance guarantees

Memory

- One must also be aware of memory usage of any program
- 32-bit machines used 4 byte pointers
- 64-bit machines use 8 byte pointers
- `double` takes up 8 bytes
- `bool` takes up only one byte
- Objects and other *complex* items have overheads
- 2D arrays (matrices) take up little over $8N^2$ bytes

Self-assessment Exercise

- Choose a simple problem that has more than one algorithm
- Discuss how algorithmic complexity varies from a naïve to a more sophisticated implementation
- Outcome
  - Understand algorithmic complexity better
  - Inspiration!
  - Technical writing practice!