Analysis of Algorithms

CSE 2320 – Algorithms and Data Structures
Vassilis Athitsos
Modified by Alexandra Stefan

University of Texas at Arlington
Book reading

• Chapter 2, especially 2.3-2.6
Analysis of Algorithms

• Given an algorithm, some key questions to ask are:
  – How efficient is this algorithm?
  – Can we predict its running time on specific inputs?
  – Should we use this algorithm or should we use an alternative?
  – Should we try to come up with a better algorithm?

• Chapter 2 establishes some guidelines for answering these questions.

• Using these guidelines, sometimes we can obtain easy answers.
  – At other times, getting the answers may be more difficult.
Empirical Analysis

• This is an alternative to the more mathematically oriented methods we will consider.

• Running two alternative algorithms on the same data and comparing the running times can be a useful tool.
  – 1 second vs. one minute is an easy-to-notice difference.

• However, sometimes empirical analysis is not a good option.
  – For example, if it would take days or weeks to run the programs.
Data for Empirical Analysis

• How do we choose the data that we use in the experiments?
Data for Empirical Analysis

- How do we choose the data that we use in the experiments?
  - Actual data.
    - Pros:
    - Cons:
  - Random data.
    - Pros:
    - Cons:
  - Perverse data.
    - Pros:
    - Cons:
Data for Empirical Analysis

• How do we choose the data that we use in the experiments?
  – Actual data.
    • Pros: give the most relevant and reliable estimates of performance.
    • Cons: may be hard to obtain.
  – Random data.
    • Pros: easy to obtain, make the estimate not data-specific.
    • Cons: may be too unrealistic.
  – Perverse data.
    • Pros: gives us worst case estimate, so we can obtain guarantees of performance.
    • Cons: the worst case estimate may be much worse than average performance.
Comparing Running Times

• When comparing running times of two implementations, we must make sure the comparison is fair.
  – We are often much more careful optimizing "our" algorithm compared to the "competitor" algorithm.

• Implementations using different programming languages may tell us more about the difference between the languages than the difference between implementations.

• An easier case is when both implementations use mostly the same codebase, and differ in a few lines.
  – Example: the different implementations of Union-Find in Chapter 1.
Avoid Insufficient Analysis

• Not performing analysis of algorithmic performance can be a problem.
  – Some programmers have no background in algorithms.
  – People with background in algorithmic analysis may be too lazy, or too pressured by deadlines, to use this background.

• Unnecessarily slow software is a common consequence when skipping analysis.
Avoid Excessive Analysis

• Worrying too much about algorithm performance can also be a problem.
  – Sometimes, slow is fast enough.
  – A user will not even notice an improvement from a millisecond to a microsecond.
  – The time spent optimizing the software should never exceed the total time saved by these optimizations.
    • E.g., do not spend 20 hours to reduce running time by 5 hours on a software that you will only run 3 times and then discard.

• Ask yourself: what are the most important bottlenecks in my code, that I need to focus on?
• Ask yourself: is this analysis worth it? What do I expect to gain?
Mathematical Analysis of Algorithms
Mathematical Analysis of Algorithms

- Some times it may be hard to mathematically predict how fast an algorithm will run.
- However, we will study a relatively small set of techniques that applies on a relatively broad range of algorithms.
- First technique: find **key operations** and **key quantities**.
  - Identify the important **operations in the program** that constitute the bottleneck in the computations.
    - This way, we can focus on estimating the number of times these operations are performed, vs. trying to estimate the number of CPU instructions and/or nanoseconds the program will take.
  - Identify a few key quantities that measure the **size of the data** that determine the running time.
Finding Key Operations

• We said it is a good idea to identify the important operations in the code, that constitute the bottleneck in the computations.
• How can we do that?
Finding Key Operations

• We said it is a good idea to identify the important *operations in the code*, that constitute the bottleneck in the computations.

• How can we do that?
  – One approach is to just think about it.
  – Another approach is to use software profilers, which show how much time is spent on each line of code.
Finding Key Operations

• What were the key operations for Union Find?
  – ???

• What were the key operations for Binary Search?
  – ???

• What were the key operations for Selection Sort?
  – ???
Finding Key Operations

• What were the key operations for Union Find?
  – Checking and changing ids in Find.
  – Checking and changing ids in Union.

• What were the key operations for Binary Search?
  – Comparisons between numbers.

• What were the key operations for Selection Sort?
  – Comparisons between numbers.

• In all three cases, the running time was proportional to the total number of those key operations.
Finding Key Quantities

• We said that it is a good idea to identify a few key quantities that measure the size of the data and that are the most important in determining the running time.

• What were the key quantities for Union-Find?
  – ???

• What were the key quantities for Binary Search?
  – ???

• What were the key quantities for Selection Sort?
  – ???
Finding Key Quantities

• We said that it is a good idea to identify a few key quantities that measure the size of the data and that are the most important in determining the running time.

• What were the key quantities for Union-Find?
  – Number of nodes, number of edges.

• What were the key quantities for Binary Search?
  – Size of the array.

• What were the key quantities for Selection Sort?
  – Size of the array.
Finding Key Quantities

• These key quantities are different for each set of data that the algorithm runs on.

• Focusing on these quantities greatly simplifies the analysis.
  – For example, there is a huge number of integer arrays of size 1,000,000, that could be passed as inputs to Binary Search or to Selection Sort.
  – However, to analyze the running time, we do not need to worry about the contents of these arrays (which are too diverse), but just about the size, which is expressed as a single number.
Describing Running Time

• Rule: most algorithms have a primary parameter $N$, that measures the size of the data and that affects the running time most significantly.
• Example: for binary search, $N$ is ???
• Example: for selection sort, $N$ is ???
• Example: for Union-Find, $N$ is ???
Describing Running Time

• Rule: most algorithms have a primary parameter $N$, that measures the size of the data and that affects the running time most significantly.
• Example: for binary search, $N$ is the size of the array.
• Example: for selection sort, $N$ is the size of the array.
• Example: for Union-Find, $N$ is ???
  – Union-Find is one of many exceptions.
  – Two key parameters, number of nodes, and number of edges, must be considered to determine the running time.
Describing Running Time

- Rule: most algorithms have a primary parameter $N$, that affects the running time most significantly.
- When we analyze an algorithm, our goal is to find a function $f(N)$, such that the running time of the algorithm is proportional to $f(N)$.
- Why proportional and not equal?
Describing Running Time

• Rule: most algorithms have a primary parameter $N$, that affects the running time most significantly.

• When we analyze an algorithm, our goal is to find a function $f(N)$, such that the running time of the algorithm is proportional to $f(N)$.

• Why proportional and not equal?

• Because the actual running time is not a defining characteristic of an algorithm.
  – Running time depends on programming language, actual implementation, compiler used, machine executing the code, ...
Describing Running Time

• Rule: most algorithms have a primary parameter $N$, that affects the running time most significantly.

• When we analyze an algorithm, our goal is to find a function $f(N)$, such that the running time of the algorithm is proportional to $f(N)$.

• We will now take a look at the most common functions that are used to describe running time.
The Constant Function: \( f(N) = 1 \)

- \( f(N) = 1 \). What does it mean to say that the running time of an algorithm is described by \( 1 \)?
The Constant Function: $f(N) = 1$

- $f(N) = 1$. What does it mean to say that the running time of an algorithm is described by 1?
- It means that the running time of the algorithm is proportional to 1, which means...
The Constant Function: \( f(N) = 1 \)

- \( f(N) = 1 \): What does it mean to say that the running time of an algorithm is described by \( 1 \)?
- It means that the running time of the algorithm is proportional to \( 1 \), which means...
  - that the running time is constant, or at least bounded by a constant.
- This happens when all instructions of the program are executed only once, or at least no more than a certain fixed number of times.
- If \( f(N) = 1 \), we say that the algorithm takes constant time. This is the best case we can ever hope for.
The Constant Function: \( f(N) = 1 \)

- What algorithm (or part of an algorithm) have we seen whose running time is constant?
The Constant Function: $f(N) = 1$

• What algorithm (or part of an algorithm) have we seen whose running time is constant?

• The **find** operation in the quick-find version of Union-Find.
Logarithmic Time: $f(N) = \lg N$

- $f(N) = \lg N$: the running time is proportional to the logarithm of $N$.
- How good or bad is that?
Logarithmic Time: \( f(N) = \lg N \)

- \( f(N) = \lg N \): the running time is proportional to the logarithm of \( N \).
- How good or bad is that?
  - \( \lg 1000 \sim \ldots \).
  - The logarithm of one million \( (10^6) \) is about \( \ldots \).
  - The logarithm of one billion \( (10^9) \) is about \( \ldots \).
  - The logarithm of one trillion \( (10^{12}) \) is about \( \ldots \).
Logarithmic Time: \( f(N) = \lg N \)

- \( f(N) = \lg N \): the running time is proportional to the logarithm of \( N \).
- How good or bad is that?
  - \( \lg 1000 \approx 10 \).
  - The logarithm of one million \( (10^6) \) is about ...
  - The logarithm of one billion \( (10^9) \) is about ...
  - The logarithm of one trillion \( (10^{12}) \) is about ...
Logarithmic Time: \( f(N) = \log N \)

- \( f(N) = \log N \): the running time is proportional to the logarithm of \( N \).
- How good or bad is that?
  - \( \log 1000 \approx 10 \).
  - The logarithm of one million (\( 10^6 \)) is about 20.
  - The logarithm of one billion (\( 10^9 \)) is about 30.
  - The logarithm of one trillion (\( 10^{12} \)) is about 40.
- Function \( \log N \) grows very slowly:
- This means that the running time when \( N = \) one trillion is only four times the running time when \( N = 1000 \). This is really good scaling behavior.
Logarithmic Time: \( f(N) = \lg N \)

- If \( f(N) = \lg N \), we say that the algorithm takes logarithmic time.
- What algorithm (or part of an algorithm) have we seen whose running time is proportional to \( \lg N \)?
Logarithmic Time: $f(N) = \log N$

- If $f(N) = \log N$, we say that the algorithm takes logarithmic time.
- What algorithm (or part of an algorithm) have we seen whose running time is proportional to $\log N$?
  - Binary Search.
  - The **Find** function on the weighted-cost quick-union version of Union-Find.
Logarithmic Time: $f(N) = \log N$

• Logarithmic time commonly occurs when solving a big problem is solved in a sequence of steps, where:
  – Each step reduces the size of the problem by some constant factor.
  – Each step requires no more than a constant number of operations.

• Binary search is an example:
  – Each step reduces the size of the problem by a factor of 2.
  – Each step requires only one comparison, and a few variable updates.
Linear Time: $f(N) = N$

- $f(N) = N$: the running time is proportional to $N$.
- This happens when we need to do some fixed amount of processing on each input element.
- What algorithms (or parts of algorithms) are examples?
Linear Time: $f(N) = N$

- $f(N) = N$: the running time is proportional to $N$.
- This happens when we need to do some fixed amount of processing on each input element.
- What algorithms (or parts of algorithms) are examples?
  - The **Union** function in the quick-find version of Union-Find.
  - Sequential search for finding the min or max value in an array.
  - Sequential search for determining whether a value appears somewhere in an array.
    - Is this ever useful? Can't we always just do binary search?
Linear Time: $f(N) = N$

- $f(N) = N$: the running time is proportional to $N$.
- This happens when we need to do some fixed amount of processing on each input element.
- What algorithms (or parts of algorithms) are examples?
  - The **Union** function in the quick-find version of Union-Find.
  - Sequential search for finding the min or max value in an array.
  - Sequential search for determining whether a value appears somewhere in an array.
    - Is this ever useful? Can't we always just do binary search?
    - If the array is not already sorted, binary search does not work.
N \log N Time

• \( f(N) = N \log N \): the running time is proportional to \( N \log N \).

• This running time is commonly encountered, especially in algorithms working as follows:
  – Break problem into smaller subproblems.
  – Solve subproblems independently.
  – Combine the solutions of the subproblems.

• Many sorting algorithms have this complexity.

• Comparing linear to \( N \log N \) time.
  – \( N = 1 \) million, \( N \log N \) is about ???
  – \( N = 1 \) billion, \( N \log N \) is about ???
  – \( N = 1 \) trillion, \( N \log N \) is about ???
N \lg N Time

• Comparing linear to N \lg N time.
  – N = 1 million, N log N is about 20 million.
  – N = 1 billion, N log N is about 30 billion.
  – N = 1 trillion, N log N is about 40 trillion.
• N \lg N is worse than linear time, but not by much.
Quadratic Time

• $f(N) = N^2$: the running time is proportional to the square of $N$.

• In this case, we say that the running time is **quadratic** to $N$.

• Any example where we have seen quadratic time?
Quadratic Time

• \( f(N) = N^2 \): the running time is proportional to the square of \( N \).

• In this case, we say that the running time is \textit{quadratic} to \( N \).

• Any example where we have seen quadratic time?
  – Selection Sort.
Quadratic Time

• Comparing linear, $N \lg N$, and quadratic time.

<table>
<thead>
<tr>
<th>$N$</th>
<th>$N \log N$</th>
<th>$N^2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$10^6$ (1 million)</td>
<td>about 20 million</td>
<td>$10^{12}$ (one trillion)</td>
</tr>
<tr>
<td>$10^9$ (1 billion)</td>
<td>about 30 billion</td>
<td>$10^{18}$ (one quintillion)</td>
</tr>
<tr>
<td>$10^{12}$ (1 trillion)</td>
<td>about 40 trillion</td>
<td>$10^{24}$ (one septillion)</td>
</tr>
</tbody>
</table>

• Quadratic time algorithms become impractical (too slow) much faster than linear and $N \lg N$ time algorithms.

• Of course, what we consider "impractical" depends on the application.
  – Some applications are more tolerant of longer running times.
Cubic Time

- $f(N) = N^3$: the running time is proportional to the cube of $N$.
- In this case, we say that the running time is cubic to $N$. 
Cubic Time

• Example Code with cubic running time: 3 nested loops (one inside the other)

• Example of a problem whose solution has cubic running time: the assignment problem.
  – We have two sets $A$ and $B$. Each set contains $N$ items.
  – We have a cost function $C(a, b)$, assigning a cost to matching an item $a$ of $A$ with an item $b$ of $B$.
  – Find the optimal one-to-one correspondence (i.e., a way to match each element of $A$ with one element of $B$ and vice versa), so that the sum of the costs is minimized.
Cubic Time

• Wikipedia example of the assignment problem:
  – We have three workers, Jim, Steve, and Alan.
  – We have three jobs that need to be done.
  – There is a different cost associated with each worker doing each job.

<table>
<thead>
<tr>
<th></th>
<th>Clean bathroom</th>
<th>Sweep floors</th>
<th>Wash windows</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jim</td>
<td>$1</td>
<td>$3</td>
<td>$3</td>
</tr>
<tr>
<td>Steve</td>
<td>$3</td>
<td>$2</td>
<td>$3</td>
</tr>
<tr>
<td>Alan</td>
<td>$3</td>
<td>$4</td>
<td>$2</td>
</tr>
</tbody>
</table>

– What is the optimal job assignment?

• Cubic running time means that it is too slow to solve this problem for, let's say, $N = 1$ million.
Exponential Time

• \( f(N) = 2^N \): this is what we call \textit{exponential running time}.
• Such algorithms are usually too slow unless \( N \) is small.
• Even for \( N = 100 \), \( 2^N \) is too large and the algorithm will not terminate in our lifetime, or in the lifetime of the Universe.
• Exponential time arises when we try all possible combinations of solutions.
  – Example: travelling salesman problem: find an itinerary that goes through each of \( N \) cities, visits no city twice, and minimizes the total cost of the tickets.
• Quantum computers (if they ever arrive) may solve \textit{some} of these problems with manageable running time.
Some Useful Constants and Functions

<table>
<thead>
<tr>
<th>symbol</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>e</td>
<td>2.71828…</td>
</tr>
<tr>
<td>γ (gamma)</td>
<td>0.57721…</td>
</tr>
<tr>
<td>φ (phi)</td>
<td>(1 + √5) / 2 = 1.61803…</td>
</tr>
</tbody>
</table>

function name approximation

<table>
<thead>
<tr>
<th>function</th>
<th>name</th>
<th>approximation</th>
</tr>
</thead>
<tbody>
<tr>
<td>[x]</td>
<td>floor function</td>
<td>x</td>
</tr>
<tr>
<td>[x]</td>
<td>ceiling function</td>
<td>x</td>
</tr>
<tr>
<td>F_N</td>
<td>Fibonacci numbers</td>
<td>φ^N / √5</td>
</tr>
<tr>
<td>H_N</td>
<td>harmonic numbers</td>
<td>ln(N) + γ</td>
</tr>
<tr>
<td>N!</td>
<td>factorial function</td>
<td>(N / e)^N</td>
</tr>
<tr>
<td>lg(N!)</td>
<td></td>
<td>N · lg(N) - 1.44N</td>
</tr>
</tbody>
</table>

These tables are for reference. We may use such symbols and functions as we discuss specific algorithms.
Motivation for Big-Oh Notation

• Given an algorithm, we want to find a function that describes the running time of the algorithm.
• Key question: how much data can this algorithm handle in a reasonable time?
• There are some details that we would actually **NOT** want this function to include, because they can make a function unnecessarily complicated.
  – Constants.
  – Behavior fluctuations on small data.
• The Big-Oh notation, which we will see in a few slides, achieves that, and greatly simplifies algorithmic analysis.
Why Constants Are Not Important

• Does it matter if the running time is $f(N)$ or $5*f(N)$?
Why Constants Are Not Important

• Does it matter if the running time is $f(N)$ or $5*f(N)$?
• For the purposes of algorithmic analysis, it typically does NOT matter.
• Constant factors are NOT an inherent property of the algorithm. They depend on parameters that are independent of the algorithm, such as:
  – Choice of programming language.
  – Quality of the code.
  – Choice of compiler.
  – Machine capabilities (CPU speed, memory size, ...)
Why Asymptotic Behavior Matters

• Asymptotic behavior: The behavior of a function as the input approaches infinity.
Why Asymptotic Behavior Matters

• Which of these functions works best asymptotically?
Why Asymptotic Behavior Matters

• Which of these functions works best asymptotically?
  – $g(N)$ seems to grow VERY slowly after a while.
Big-Oh Notation

• A function $g(N)$ is said to be $O(f(N))$ if there exist constants $c_0$ and $N_0$ such that:

$$g(N) < c_0 f(N) \text{ for all } N > N_0.$$ 

• Remember and understand the definition above.

• Typically, $g(N)$ is the running time of an algorithm, in your favorite units, implementation, and machine. This can be a rather complicated function.

• In algorithmic analysis, we try to find a $f(N)$ that is **simple**, and such that $g(N) = O(f(N))$. 

Why Use Big-Oh Notation?

• A function $g(N)$ is said to be $O(f(N))$ if there exist constants $c_0$ and $N_0$ such that:

$$g(N) < c_0 f(N) \quad \text{for all } N > N_0.$$  

• The Big-Oh notation greatly simplifies the analysis task, by:

  1. Ignoring constant factors. How is this achieved?
     • By the $c_0$ in the definition. We are free to choose ANY constant $c_0$ we want, to make the formula work.
     • Thus, Big-Oh notation is independent of programming language, compiler, machine performance, and so on...
Why Use Big-Oh Notation?

• A function $g(N)$ is said to be $O(f(N))$ if there exist constants $c_0$ and $N_0$ such that:

$$g(N) < c_0 f(N) \text{ for all } N > N_0.$$ 

• The Big-Oh notation greatly simplifies the analysis task, by:

  2. Ignoring behavior for small inputs. How is this achieved?
    • By the $N_0$ in the implementation. If a finite number of values are not compatible with the formula, just ignore them.
    • Thus, big-Oh notation focuses on asymptotic behavior.
Why Use Big-Oh Notation?

• A function $g(N)$ is said to be $O(f(N))$ if there exist constants $c_0$ and $N_0$ such that:

$$g(N) < c_0 f(N) \text{ for all } N > N_0.$$ 

• The Big-Oh notation greatly simplifies the analysis task, by:

3. Allowing us to describe complex running time behaviors of complex algorithms with simple functions, such as $N$, $\lg N$, $N^2$, $2^N$, and so on.

• Such simple functions are sufficient for answering many important questions, once you get used to Big-Oh notation.
Inferences from Big-Oh Notation

• Binary search takes logarithmic time.
• This means that, if \( g(N) \) is the running time, there exist constants \( c_0 \) and \( N_0 \) such that:

\[
g(N) < c_0 \lg(N) \quad \text{for all } N > N_0.
\]

• Can this function handle trillions of data in reasonable time?
  – NOTE: the question is about \textbf{time}, not about \textbf{memory}. 

Inferences from Big-Oh Notation

• Binary search takes logarithmic time.
• This means that, if \( g(N) \) is the running time, there exist constants \( c_0 \) and \( N_0 \) such that:
  \[
g(N) < c_0 \lg(N) \quad \text{for all } N > N_0.
\]
• Can this function handle trillions of data in reasonable time?
  – NOTE: the question is about time, not about memory.
• The answer is an easy YES!
  – We don't even know what \( c_0 \) and \( N_0 \) are, and we don't care.
  – The key thing is that the running time is \( O(\lg(N)) \).
Inferences from Big-Oh Notation

• Selection Sort takes quadratic time.
• This means that, if $g(N)$ is the running time, there exist constants $c_0$ and $N_0$ such that:

$$g(N) < c_0 N^2 \quad \text{for all } N > N_0.$$  

• Can this function handle one billion data in reasonable time?
Inferences from Big-Oh Notation

• Selection Sort takes quadratic time.
• This means that, if \( g(N) \) is the running time, there exist constants \( c_0 \) and \( N_0 \) such that:

\[
g(N) < c_0 N^2 \quad \text{for all } N > N_0.
\]

• Can this function handle one billion data in reasonable time?
• The answer is an easy NO!
  – Again, we don't know what \( c_0 \) and \( N_0 \) are, and we don't care.
  – The key thing is that the running time is quadratic.
Is Big-Oh Notation Always Enough?

• NO! Big-Oh notation does not always tell us which of two algorithms is preferable.
Is Big-Oh Notation Always Enough?

• NO! Big-Oh notation does not always tell us which of two algorithms is preferable.
  – Example 1: if we know that the algorithm will only be applied to relatively small N, we may prefer a running time of \( N^2 \) nanoseconds over \( \lg(N) \) centuries.
  – Example 2: even constant factors can be important. For many applications, we strongly prefer a running time of \( 3N \) over \( 1500N \).
Is Big-Oh Notation Always Enough?

• NO! Big-Oh notation does not always tell us which of two algorithms is preferable.
  – Example 1: if we know that the algorithm will only be applied to relatively small $N$, we may prefer a running time of $N^2$ nanoseconds over $\lg(N)$ centuries.
  – Example 2: even constant factors can be important. For many applications, we strongly prefer a running time of $3N$ over $1500N$.

• Big-Oh notation is not meant to tells us everything about running time.

• But, Big-Oh notation tells us a lot, and is often much easier to compute than actual running times.
Simplifying Big-Oh Notation

• Suppose that we are given this running time:
  \[ g(N) = 35N^2 + 41N + \log(N) + 1532. \]

• How can we express \( g(N) \) in Big-Oh notation?
Simplifying Big-Oh Notation

• Suppose that we are given this running time: 
  \( g(N) = 35N^2 + 41N + \log(N) + 1532. \)

• How can we express \( g(N) \) in Big-Oh notation?

• Typically we say that \( g(N) = O(N^2). \)

• The following are also correct, but unnecessarily complicated, and thus less useful, and rarely used.
  – \( g(N) = O(N^2) + O(N). \)
  – \( g(N) = O(N^2) + O(N) + O(\log(N)) + O(1). \)
  – \( g(N) = O(35N^2 + 41N + \log(N) + 1532). \)
Simplifying Big-Oh Notation

• Suppose that we are given this running time:
  \[ g(N) = 35N^2 + 41N + \log(N) + 1532. \]

• We say that \( g(N) = O(N^2). \)

• Why is this mathematically correct?
  – Why can we ignore the non-quadratic terms?

• Ans 1: Using the Big-Oh definition: we can find an \( N_0 \) such that, for all \( N > N_0 \):
  \[ g(N) < 36N^2. \]
  – If you don't believe this, do the calculations for practice.
Simplifying Big-Oh Notation

• Suppose that we are given this running time:
  \( g(N) = 35N^2 + 41N + \lg(N) + 1532. \)
• We say that \( g(N) = O(N^2) \).
• Why is this mathematically correct?
  – Why can we ignore the non-quadratic terms?
• Ans 2 (another way to show correctness): as \( N \) goes to infinity, what is the limit of \( g(N) / N^2 \) ?
Simplifying Big-Oh Notation

• Suppose that we are given this running time: 
  \[ g(N) = 35N^2 + 41N + \log(N) + 1532. \]

• We say that \( g(N) = O(N^2). \)

• Why is this mathematically correct?
  – Why can we ignore the non-quadratic terms?

• Ans 2 (another way to show correctness): as \( N \) goes to infinity, what is the limit of \( g(N) / N^2 \)?
  – 35.
  – This shows that the non-quadratic terms become negligible as \( N \) gets larger.
Trick Question

• Let $g(N) = N \lg N$.
• Is it true that $g(N) = O(N^{100})$?
Trick Question

• Let $g(N) = N \ lg \ N$.
• Is it true that $g(N) = O(N^{100})$?
• Yes. Let's look again at the definition of Big-Oh:
• A function $g(N)$ is said to be $O(f(N))$ if there exist constants $c_0$ and $N_0$ such that:

\[ g(N) < c_0 f(N) \quad \text{for all } N > N_0. \]

• Note the "<" sign to the right of $g(N)$.
• Thus, if $g(N) = O(f(N))$ and $f(N) < h(N)$, it follows that $g(N) = O(h(N))$. 
Omega (Ω) and Theta (Θ) Notations

• If $f(N) = O(g(N))$, then we also say that $g(N) = \Omega(f(N))$.

• If $g(N) = O(f(N))$ and $g(N) = \Omega(f(N))$, then we say that $g(N) = \Theta(f(N))$.

• The Theta notation is clearly stricter than the Big-Oh notation:
  - We can say that $N^2 = O(N^{100})$.
  - We cannot say that $N^2 = \Theta(N^{100})$.  

Using Limits

- if \( \lim_{N \to \infty} \frac{g(N)}{f(N)} \) is a constant, then \( g(N) = ???(f(N)) \).
  - "Constant" includes zero, but does NOT include infinity.

- if \( \lim_{N \to \infty} \frac{f(N)}{g(N)} = \infty \) then \( g(N) = ???(f(N)) \).

- if \( \lim_{N \to \infty} \frac{f(N)}{g(N)} \) is a constant, then \( g(N) = ???(f(N)) \).
  - Again, "constant" includes zero, but not infinity.

- if \( \lim_{N \to \infty} \frac{f(N)}{g(N)} \) is a non-zero constant, then \( g(N) = ???(f(N)) \).
  - In this definition, both zero and infinity are excluded.
Using Limits

• if \( \lim_{N \to \infty} \frac{g(N)}{f(N)} \) is a constant, then \( g(N) = O(f(N)) \).
  
  – "Constant" includes zero, but does NOT include infinity.

• if \( \lim_{N \to \infty} \frac{f(N)}{g(N)} = \infty \) then \( g(N) = O(f(N)) \).

• if \( \lim_{N \to \infty} \frac{f(N)}{g(N)} \) is a constant, then \( g(N) = \Omega(f(N)) \).
  
  – Again, "constant" includes zero, but not infinity.

• if \( \lim_{N \to \infty} \frac{f(N)}{g(N)} \) is a non-zero constant, then \( g(N) = \Theta(f(N)) \).
  
  – In this definition, both zero and infinity are excluded.
Using Limits - Comments

• The previous formulas relating limits to big-Oh notation show once again that big-Oh notation ignores:
  – constants
  – behavior for small values of N.

• How do we see that?
  – In the previous formulas, it is sufficient that the limit is equal to a constant. **The value of the constant does not matter.**
  – In the previous formulas, only **the limit at infinity** matters. This means that we can ignore behavior up to any finite value, if we need to.
Basic Recurrences

• How do we compute the running time of an algorithm in Big-Oh notation?
• Sometimes it is easy, sometimes it is hard.
• We will learn a few simple tricks that work in many cases that we will encounter this semester.
Recurrences

1. Reduce the problem size by 1 in **linear** time
   - E.g. Check all items, eliminate 1
2. *Halve* problem in **constant** time
3. *Halve* problem in **linear** time
4. Break the problem into 2 *halves* in **linear** time
5. Break the problem into 2 *halves* in **constant** time
Case 1: Check All Items, Eliminate One

• In this case, the algorithm proceeds in a sequence of similar steps, where:
  – each step loops through all items in the input, and eliminates one item.

• Any examples of such an algorithm?
Case 1: Check All Items, Eliminate One

• In this case, the algorithm proceeds in a sequence of similar steps, where:
  – each step loops through all items in the input, and eliminates one item.

• Any examples of such an algorithm?
  – Selection Sort.
Case 1: Check All Items, Eliminate One

- Let $g(N)$ be an approximate estimate of the running time, measured in time units of our convenience.
  - In this case, we choose as time unit the time that it takes to examine one item.
  - Obviously, this is a simplification, since there are other things that such an algorithm will do, in addition to just examining one item.
  - That is one of the plusses of using Big-Oh notation. We can ignore parts of the algorithm that take a relatively small time to run, and focus on the part that dominates running time.
Case 1: Check All Items, Eliminate One

• Let $g(N)$ be the running time.
• Then, $g(N) = ???$
Case 1: Check All Items, Eliminate One

• Let \( g(N) \) be the running time.

• Then, \( g(N) = g(N-1) + N \). Why?
  – Because we need to examine all items (\( N \) units of time), and then we need to run the algorithm on \( N-1 \) items.

• \( g(N) = N + g(N-1) \)
  = \( N + (N-1) + g(N-2) \)
  = \( N + (N-1) + (N-2) + g(N-3) \)
  \[ \ldots \]
  = \( N + (N-1) + \ldots + 3 + 2 + 1 \)
  = \( N(N + 1) / 2 \)
  = \( O(N^2) \)

• Conclusion: The algorithm takes quadratic time.
Case 2: Halve the Problem in Constant Time

• In this case, each step of the algorithm consists of:
  – performing a constant number of operations, and then reducing the size of the input by half.

• Any example of such an algorithm?
Case 2: Halve the Problem in Constant Time

• In this case, each step of the algorithm consists of:
  – performing a constant number of operations, and then reducing the size of the input by half.

• Any example of such an algorithm?
  – Binary Search.

• What is a convenient unit of time to use here?
Case 2: Halve the Problem in Constant Time

• In this case, each step of the algorithm consists of:
  – performing a constant number of operations, and then reducing the size of the input by half.

• Any example of such an algorithm?
  – Binary Search.

• What is a convenient unit of time to use here?
  – The time it takes to do the constant number of operations to halve the input.
Case 2: Halve the Problem in Constant Time

• In this case, each step of the algorithm consists of:
  – performing a constant number of operations, and then reducing the size of the input by half.
• $g(2^n) = \text{??}$
Case 2: Halve the Problem in Constant Time

• In this case, each step of the algorithm consists of:
  – performing a constant number of operations, and then reducing the size of the input by half.

• $g(2^n) = 1 + g(2^{n-1})$
  $= 2 + g(2^{n-2})$
  $= 3 + g(2^{n-3})$
  \[\vdots\]
  $= n + g(2^0)$
  $= n + 1$.

• $O(n)$ time for $N = 2^n$.

• Substituting $n$ with $\lg N$: $O(\lg N)$ time.
Case 3: Halve the Problem in Linear Time

- In this case, each step of the algorithm consists of:
  - Performing a linear (i.e., $O(N)$) number of operations, and then reducing the size of the input by half.
- $g(N) = ???$
Case 3: Halve the Problem in Linear Time

• In this case, each step of the algorithm consists of:
  – Performing a linear (i.e., $O(N)$) number of operations, and then reducing the size of the input by half.

• $g(N) = g(N/2) + N$
  = $g(N/4) + N/2 + N$
  = $g(N/8) + N/4 + N/2 + N$
  ...
  = $1 + 2 + 4 + \ldots + N/4 + N/2 + N$
  = ??? (do you recognize this series?)
Case 3: Halve the Problem in Linear Time

• In this case, each step of the algorithm consists of:
  – Performing a linear (i.e., $O(N)$) number of operations, and then reducing the size of the input by half.

• $g(N) = g(N/2) + N$
  $= g(N/4) + N/2 + N$
  $= g(N/8) + N/4 + N/2 + N$
  $\ldots$
  $= 1 + 2 + 4 + \ldots + N/4 + N/2 + N$
  $= ???$

• $1 + 2 + 4 + \ldots + 2^x = ???$

• What is the general formula for the above series?
Case 3: Halve the Problem in Linear Time

• In this case, each step of the algorithm consists of:
  – Performing a linear (i.e., $O(N)$) number of operations, and then reducing the size of the input by half.

• $g(N) = g(N/2) + N$
  $= g(N/4) + N/2 + N$
  $= g(N/8) + N/4 + N/2 + N$
  $\ldots$
  $= 1 + 2 + 4 + \ldots + N/4 + N/2 + N$
  $= \text{about} \ 2N$

• $O(N)$ time.
Case 4: Break Problem Into Two Halves in Linear Time

• In this case, each step of the algorithm consists of:
  – Doing $O(N)$ operations to split the problem into two halves.
  – Calling the algorithm recursively on each half.
  – Doing $O(N)$ operations to combine the two answers.

• $g(N) = ???$
Case 4: Break Problem Into Two Halves in Linear Time

• In this case, each step of the algorithm consists of:
  – Doing $O(N)$ operations to split the problem into two halves.
  – Calling the algorithm recursively on each half.
  – Doing $O(N)$ operations to combine the two answers.

• $g(N) = 2g(N/2) + N$
  $= 4g(N/4) + N + N$
  $= 8g(N/8) + N + N + N$
  $\ldots$
  $= N \log N$
Case 4: Break Problem Into Two Halves in Linear Time

• In this case, each step of the algorithm consists of:
  – Doing $O(N)$ operations to split the problem into two halves.
  – Calling the algorithm recursively on each half.
  – Doing $O(N)$ operations to combine the two answers.

• Note: we have not seen any examples of this case yet, but we will see several such examples when we study sorting algorithms.
Case 5: Break Problem Into Two Halves in Constant Time

• In this case, each step of the algorithm consists of:
  – Doing $O(1)$ operations to split the problem into two halves.
  – Calling the algorithm recursively on each half.
  – Doing $O(1)$ operations to combine the two answers.

• $g(N) = ???$
Case 5: Break Problem Into Two Halves in Constant Time

• In this case, each step of the algorithm consists of:
  – Doing $O(1)$ operations to split the problem into two halves.
  – Calling the algorithm recursively on each half.
  – Doing $O(1)$ operations to combine the two answers.

• $g(N) = 2g(N/2) + 1$
  $= 4g(N/4) + 2 + 1$
  $= 8g(N/8) + 4 + 2 + 1$
  
  ... 
  $= \text{about } 2N$