Algorithmic Analysis

Dr. Robert Amelard

(adapted from Dr. Igor Ivkovic)

ramelard@uwaterloo.ca

Objectives

- Introduction to Algorithmic Analysis
- Big-O Notation Formally Defined
- From Source Code to Big-O Notation
- Analysis of Recursive Algorithms

A Graphical Introduction /1

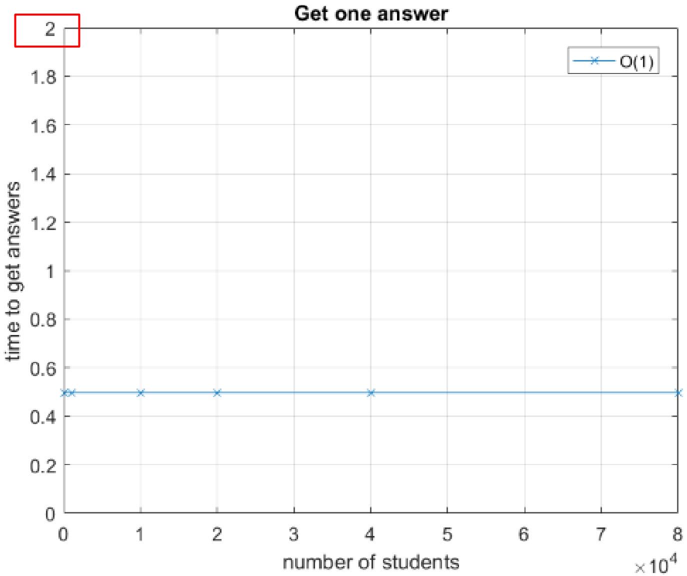
- It's all about **scale**
  - How fast/slow does your algorithm perform with more complex/larger cases?
- **Example**: asking the class a question

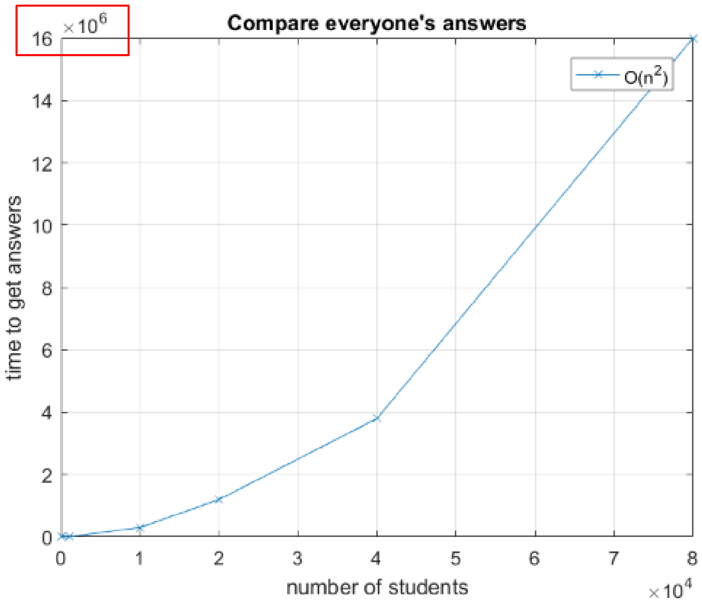
O(1)	**O(n)**	**O(n^2)**
Get 1 answer	Get everyone's answer	Compare everyone's answer to everyone else

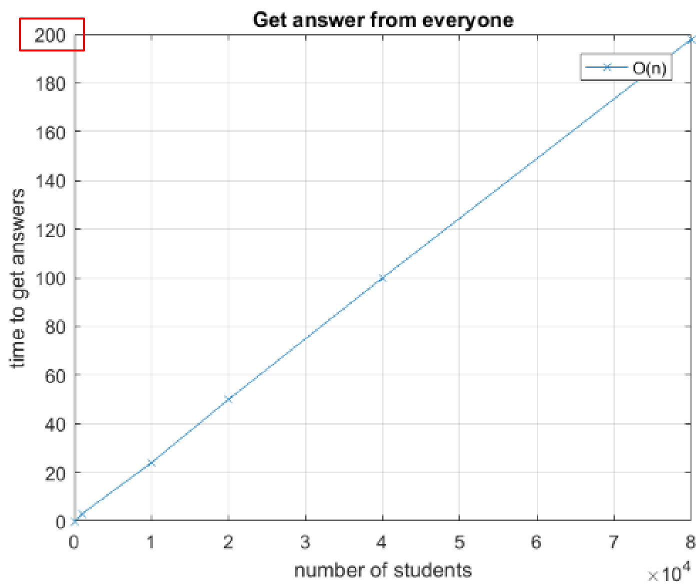
- Let's start with an interactive example with code

(Empty slide for notes on demo code)

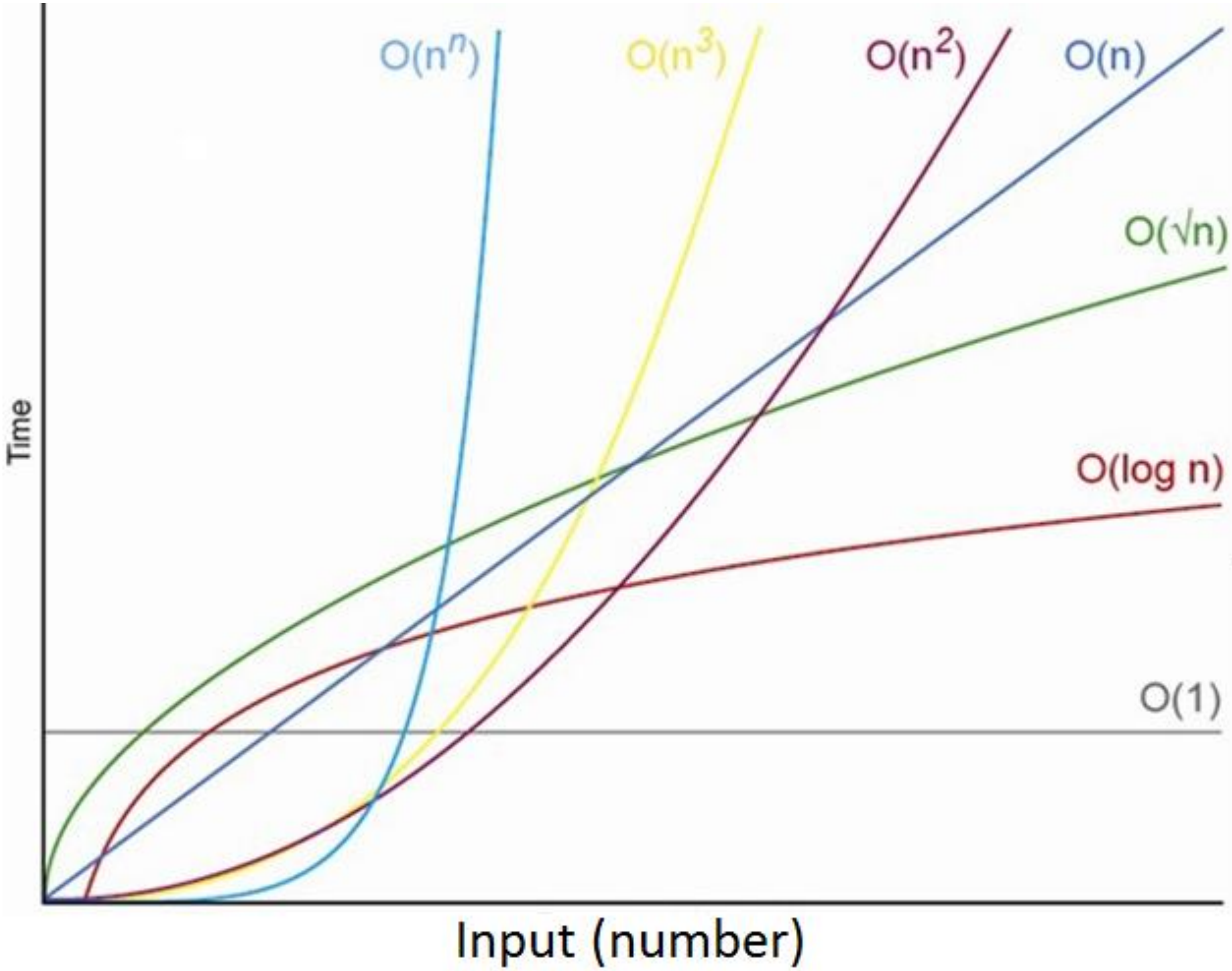
A Graphical Introduction /2







A Graphical Introduction /3



http://codedreaming.com

Introduction to Algorithmic Analysis /1

- **The process of measuring performance of a computer program can be challenging**
  - The performance may be affected by the algorithm, compiler, programming language, and machine architecture, all of which play a part in program execution
  - Instead of analyzing the entire execution process, we will focus on classifying only the algorithm and the number of operations that the algorithm performs
  - Moreover, we will attempt to calculate the efficiency of an algorithm before writing any code to save time and cost

Introduction to Algorithmic Analysis /2

- **Our first goal is a method by which we can formally compare two algorithms regardless of the input size**
  - To that end, we will express the number of steps an algorithm takes to run itself to completion (i.e., the number of steps taken) as a function f(n); in f(n), n is the size of the input

  - For each function/algorithm, we will aim to compute f(n) and then compare that measurements against equivalent measurements for other functions/algorithms

Introduction to Algorithmic Analysis /3

- **Searching algorithms:**
  - For the **`SequentialSearch`** function given below, the size of the input **`(n)`** is the size of the array **`(n = size)`**
  - In the best case scenario, the first element of the array is equal to K, and the function takes a handful of operations
  - In the worst case scenario, the array does not contain K, and the function takes at least **`n`** number of operations

```
int SequentialSearch(int A[], int size, int K) {
        for (int i = 0; i < size; i++) {
                if (A[i] == K)
                        return i;
        }
        return -1;
}
```

Introduction to Algorithmic Analysis /4

- **Another method that can be used to perform the search is the binary search that is shown below**
  - In the best case scenario, the middle element of the array is equal to K, and the function takes several operations
  - In the worst case scenario, the array does not contain K, but the function takes less than n number of operations

```
int BinarySearch(int A[], int L, int R, int K) {
        // A must be already sorted for this to work
        int mid = (L + R) / 2;
        if (R < L)
                return -1;
        else if (A[mid] == K)
                return mid;
        else if (K > A[mid])
                return BinarySearch(A, mid + 1, R, K);
        else
                return BinarySearch(A, L, mid - 1, K);
}
```

Introduction to Algorithmic Analysis /5

- **So which of the two search functions is faster when given a sorted array as input?**
  - In the best scenarios, both of them take a handful of operations, so both of them perform roughly the same
  - However, what if we are not dealing with the best case?

- **What we are interested in is the increase in the number of operations as the input size grows**
  - For small inputs, both search algorithms may finish in negligible amount of time
  - As the input size grows, so does the time it takes for each algorithm to complete
  - The growth rate of an algorithm will determine which algorithm performs faster for large input sizes

Introduction to Algorithmic Analysis /6

- **To compare the growth of two algorithms, we will consider the worst-case scenario for each algorithm**
  - For the two search algorithms, `SequentialSearch` will take roughly `n` operations to complete if `K` is not found
  - At the same time, `BinarySearch` will take less than n operations, and more specifically around `log(n)` operations, to complete if `K` is not found

- **We will express the performance of an algorithm by assigning it to its own class/category**
  - We will use something called Big-O notation to express performance information about an algorithm
  - Using the Big-O notation, `SequentialSearch` can be classified as `O(n)` and `BinarySearch` as `O(log(n))`

Introduction to Algorithmic Analysis /7

- **There are several common classes of algorithms based on the Big-O notation:**
  - `O(1)` – constant time algorithms
  - `O(log(n))` – logarithmic time algorithms
  - `O(n)` – linear time algorithms
  - `O(n x log(n))`
  - $O(n^2)$ – quadratic time algorithms
  - $O(n^3)$ – cubic time algorithms
  - $O(2^n)$ – exponential time algorithms
  - $O(10^n)$
  - `O(n!)` – factorial time algorithms

Increasing order of growth

Introduction to Algorithmic Analysis /8

- **Using the Big-O classification, we can compare and group algorithms based on their performance**
  - Out of the two searching algorithms, the binary search, which is an O(log(n)) algorithm, performs faster than the linear search, which is an O(n) algorithm
  - For sorting, O(n x log(n)) algorithms are the best performing when it comes to comparison-based sorting
  - Exponential algorithms, such as $O(2^n)$, appear in certain classes of problems (e.g., graph problems)
  - However, exponential algorithms are impractical for very large inputs since they could take a very long time to run
    - There are reasonable algorithms that could run “from now until the end of time” and still not complete their function ☺

Big-O Notation Formally Defined /1

- **Big-O notation describes the worst case runtime of a given algorithm**
  - More specifically, Big-O describes the absolute worst case in terms of the number of operations that could occur when running an algorithm against input of size n

- **Formally, for a function f(n) that represents the number of operations for an algorithm:**
  - A function f(n) is classified as O(g(n)) if there exist two positive constants K and n_0 such that
    $|f(n)| \leq K|g(n)| \text{ for all } n \geq n_0$
  - Visually, there exists a positive constant K for which $K|g(n)|$ lies above $f(n)$ for all $n \geq n_0$

Big-O Notation Formally Defined /2

- **Visual demonstration of Big-O notation:**
  - Let f(x) = x, g(x) = x^2, and h(x) = log(x)
  - Then, f(x) = O(g(x)) since g(x) lies above f(x)
  - Similarly, h(x) = O(g(x)) and h(x) = O(f(x))

Source: https://www.desmos.com/calculator

Big-O Notation Formally Defined /3

- **Simplification rules regarding Big-O notation:**
  - All logarithmic functions regardless of their base belong to the same O(log(n)) logarithmic class of algorithms
  - All polynomial functions (e.g., $ax^2 + bx + c$) where k is the largest degree belong to the same $O(n^k)$ class
  - Constant terms and multipliers can be ignored when simplifying the expressions (e.g., $O(2 \times n^2) = O(n^2)$)
    - Constant terms are important if we are trying to compare the exact number of operations between two algorithms
  - Exponential functions belong to different classes depending on their base (e.g., $O(2^n)$ & $O(3^n)$ are distinct)

Big-O Notation Formally Defined /4

- **Example1:**
  - Let $f(n) = 100n^2$ and $g(n) = n^2$. Show that $f(n) = O(g(n))$
  - Steps:
    - Select $n_o = 1$, so then for $n \geq n_o$

$$100n^2 \leq K * n^2$$

$$100 \leq K$$

    - Hence, for $K \geq 100$ and $n \geq 1$, $f(n) = O(g(n))$

Big-O Notation Formally Defined /5

- **Example2:**
  - Let $f(n) = n^2$ and $g(n) = n^3$. Show that $f(n) = O(g(n))$
  - Steps:
    - Select $n_o = 1$, so then for $n \geq n_o$

$$n^2 \leq K * n^3$$

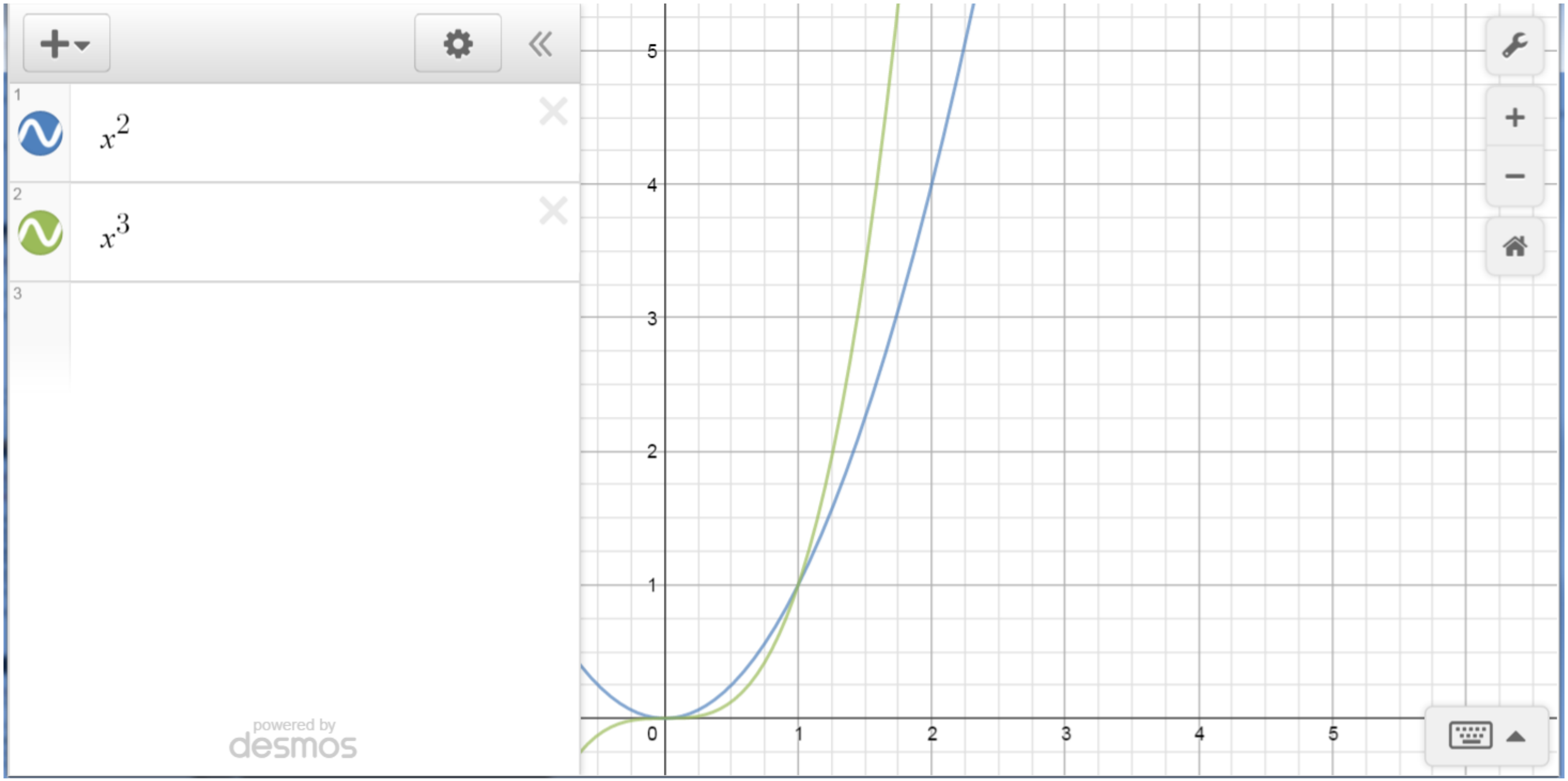
$$1 \leq Kn$$

$$\frac{1}{K} \leq n$$

    - From there: $1 / n \leq K$, and since $n \geq 1$ then $K \geq 1$
    - Hence, for $K \geq 1$ and $n \geq 1$, $f(n) = O(g(n))$

Big-O Notation Formally Defined /6

- **Example2 Visualized:**
  - For $f(x) = x^2$ and $g(x) = x^3$, it follows that $f(x) = O(g(x))$ for $x \geq 1$



Source: https://www.desmos.com/calculator

Big-O Notation Formally Defined /7

- **Another approach to comparing function growth:**
  - Compute $\lim_{n \to \infty} \frac{f(n)}{g(n)}$
  - If the value is **0**, then **f(n)** grows slower than **g(n)**; that is, **f(n) = O(g(n))** (e.g., n / n^2)
  - If the value is a constant **c**, then **f(n)** grows as fast as **g(n)**; that is, **f(n) = O(g(n))** (e.g., $2n / n$)
  - If the value is infinity ∞, then **f(n)** grows faster than **g(n)**; that is, **g(n) = O(f(n))** (e.g., n^2 / n)

Big-O Notation Formally Defined /8

- **Example using limits:**
  - Let f(n) = n^2 and g(n) = 2^n. Show that f(n) = O(g(n))
  - Steps: $$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{n^2}{2^n}$$
    $$= \lim_{n\to\infty} \frac{2n}{\ln(2)\, 2^n} \text{ (using L'Hospital's Rule)}$$
    $$= \lim_{n\to\infty} \frac{2}{\ln(2)^2\, 2^n} \text{ (using L'Hospital's Rule)}$$
    $$= \frac{2}{\ln(2)^2} \lim_{n\to\infty} \frac{1}{2^n} = 0$$
  - Hence, f(n) = O(g(n))

From Source Code to Big-O Notation /1

- **How can we compute the Big-O measurement directly from source code?**
  - Express each loop as a summation/sigma, $\sum$
  - For segments of code that are repeated on each iteration, express each segment as a constant (e.g., a)
  - For nested loops, use nested summations (e.g., $\sum \sum$)
  - Finally, use summation formulas for simplification

```
for i = 0 to n – 2 do {
        for j = i + 1 to n – 1 do {
                for k = i to n do {
                        // constant steps
                }
        }
}
```

$$f(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} \sum_{k=i}^{n} a$$

From Source Code to Big-O Notation /2

$$\boldsymbol{f(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} \sum_{k=i}^{n} a =}$$

$$\boldsymbol{a \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} (n - i + 1) = a \sum_{i=0}^{n-2} (n - i - 1)(n - i + 1) =}$$

$$\boldsymbol{a((n + 1)(n - 1) + n(n - 2) + \cdots + 3 * 1) =}$$

$$\boldsymbol{a \sum_{j=1}^{n-1} (j + 2)j = a \sum_{j=1}^{n-1} j^2 + a \sum_{j=1}^{n-1} 2j =}$$

$$\boldsymbol{a \frac{(n - 1)n(2n - 1)}{6} + 2a \frac{(n - 1)n}{2} =}$$

$$\boldsymbol{a \frac{n(n - 1)(2n + 5)}{6} = a\left(\frac{1}{3} n^3 + \frac{1}{2} n^2 - \frac{5}{6} n\right) = O(n^3)}$$

Analysis of Recursive Algorithms /1

- **How can we compute the Big-O measurement for recursive algorithms?**
  - Define the base cases and the recursive case
  - Use backwards substitution to go from the recursive case down to the base cases
  - Use summation and other formulas for simplification

```
int BinarySearch(int A[], int L, int R, int K) {
        // A must be already sorted for this to work
        int mid = (L + R) / 2;
        if (R < L)
                return -1;
        else if (A[mid] == K)
                return mid;
        else if (K > A[mid])
                return BinarySearch(A, mid + 1, R, K);
        else
                return BinarySearch(A, L, mid - 1, K);
}
```

```
Let n = R – L + 1, then
  T(1) = a
  T(n) = b + T(n/2)
```

Analysis of Recursive Algorithms /2

```
Let n = R – L + 1, then
  T(1) = a
  T(n) = b + T(n/2)
```

$$T(n) = b + T(\frac{n}{2})$$
$$= b + b + T(\frac{n}{4})$$
$$= b + b + b + T(\frac{n}{8})$$
$$= \ldots$$
$$= ib + T(\frac{n}{2^i})$$

Backwards substitution

When $(n / 2^i) = 1$, let $i = c$

It follows that $(n / 2^c) = 1$, and $n = 2^c$, so $c = \log_2(n)$

$$T(n) = cb + T(\frac{n}{2^c})$$
$$= cb + T(1)$$
$$= blog_2(n) + a$$
$$= O(log(n))$$

Lecture Notes Summary

- **What do you need to know?**
  - Measuring algorithm performance
  - Using Big-O to classify algorithms
  - Defining Big-O notation formally
  - Mapping source code to Big-O
  - Analyzing recursive algorithms

Food for Thought

- **Read:**
  - Chapter 5 (Algorithmic Analysis) from the course handbook

- **Additional Readings:**
  - Section 1.2 (Running Time Analysis) and Appendix B (More Big-O Notation) from “Data Structures and Other Objects Using C++” by Main and Savitch