APS1070, Winter 2022: Lecture 2

Sinisa Colic and Samin Aref

Based on course material by Mark C. Wilson

What is Lecture 2 about?

- An introduction to the analysis of algorithms and data structures.
- Asymptotic complexity analysis. Sorting algorithms.
- Hashing and the dictionary abstract data type.
- Prerequisite: Fundamental concepts from discrete mathematics (sets, functions, inequalities, limits) covered as "backgrounds" in this slide deck.
- This is a "theory" lecture but we also assess ability to implement these abstract structures and algorithms in projects.

Background: Mathematics review

Sets

- A set is an unordered collection of objects (called elements).

Sets

- A set is an unordered collection of objects (called elements).
- Important sets:

Sets

- A set is an unordered collection of objects (called elements).
- Important sets:
  - $\mathbb{N} = \{0, 1, 2, \dots\}$, the set of natural numbers.

Sets

- A set is an unordered collection of objects (called elements).
- Important sets:
  - $\mathbb{N} = \{0, 1, 2, \dots\}$, the set of natural numbers.
  - $\mathbb{R}$, the set of real numbers.

Sets

- A set is an unordered collection of objects (called elements).
- Important sets:
  - $\mathbb{N} = \{0, 1, 2, \dots\}$, the set of natural numbers.
  - $\mathbb{R}$, the set of real numbers.
  - $\varnothing = \{\}$, the empty set having no elements.

Sets

- A set is an unordered collection of objects (called elements).
- Important sets:
  - $\mathbb{N} = \{0, 1, 2, \dots\}$, the set of natural numbers.
  - $\mathbb{R}$, the set of real numbers.
  - $\varnothing = \{\}$, the empty set having no elements.
- Notation: $X = \{2, 3, 5, 7, 11\}$ or
  $X = \{x \in \mathbb{N} : x \text{ is prime and } x < 12\}$.

Sets

- A set is an unordered collection of objects (called elements).
- Important sets:
  - $\mathbb{N} = \{0, 1, 2, \dots\}$, the set of natural numbers.
  - $\mathbb{R}$, the set of real numbers.
  - $\varnothing = \{\}$, the empty set having no elements.
- Notation: $X = \{2, 3, 5, 7, 11\}$ or
  $X = \{x \in \mathbb{N} : x \text{ is prime and } x < 12\}$.
- Operations:

Sets

- A set is an unordered collection of objects (called elements).
- Important sets:
  - $\mathbb{N} = \{0, 1, 2, \dots\}$, the set of natural numbers.
  - $\mathbb{R}$, the set of real numbers.
  - $\varnothing = \{\}$, the empty set having no elements.
- Notation: $X = \{2, 3, 5, 7, 11\}$ or
  $X = \{x \in \mathbb{N} : x \text{ is prime and } x < 12\}$.
- Operations:
  - $A \cap B = \{x : x \in A \text{ and } x \in B\}$

Sets

- A set is an unordered collection of objects (called elements).
- Important sets:
  - $\mathbb{N} = \{0, 1, 2, \dots\}$, the set of natural numbers.
  - $\mathbb{R}$, the set of real numbers.
  - $\varnothing = \{\}$, the empty set having no elements.
- Notation: $X = \{2, 3, 5, 7, 11\}$ or
  $X = \{x \in \mathbb{N} : x \text{ is prime and } x < 12\}$.
- Operations:
  - $A \cap B = \{x : x \in A \text{ and } x \in B\}$
  - $A \cup B = \{x : x \in A \text{ or } x \in B\}$

Sets

- A set is an unordered collection of objects (called elements).
- Important sets:
  - $\mathbb{N} = \{0, 1, 2, \dots\}$, the set of natural numbers.
  - $\mathbb{R}$, the set of real numbers.
  - $\varnothing = \{\}$, the empty set having no elements.
- Notation: $X = \{2, 3, 5, 7, 11\}$ or $X = \{x \in \mathbb{N} : x \text{ is prime and } x < 12\}$.
- Operations:
  - $A \cap B = \{x : x \in A \text{ and } x \in B\}$
  - $A \cup B = \{x : x \in A \text{ or } x \in B\}$
  - $|A|$ is the number of elements of A.

Functions

- A function is a mapping f from a set X (the domain) to a set Y (the codomain) such that every $x \in X$ maps to a unique $y \in Y$.

Functions

- A function is a mapping f from a set X (the domain) to a set Y (the codomain) such that every $x \in X$ maps to a unique $y \in Y$.
- Important functions from $\mathbb{R}$ to $\mathbb{R}$:

Functions

- A function is a mapping f from a set X (the domain) to a set Y (the codomain) such that every $x \in X$ maps to a unique $y \in Y$.
- Important functions from $\mathbb{R}$ to $\mathbb{R}$:
  - Power functions $f(x) = x$, $f(x) = x^2$, $f(x) = x^3$, etc

Functions

- A function is a mapping f from a set X (the domain) to a set Y (the codomain) such that every $x \in X$ maps to a unique $y \in Y$.
- Important functions from $\mathbb{R}$ to $\mathbb{R}$:
  - Power functions $f(x) = x$, $f(x) = x^2$, $f(x) = x^3$, etc
  - Exponential functions $f(x) = 2^x$, $f(x) = (1.5)^x$, etc

Functions

- A function is a mapping f from a set X (the domain) to a set Y (the codomain) such that every $x \in X$ maps to a unique $y \in Y$.
- Important functions from $\mathbb{R}$ to $\mathbb{R}$:
  - Power functions $f(x) = x$, $f(x) = x^2$, $f(x) = x^3$, etc
  - Exponential functions $f(x) = 2^x$, $f(x) = (1.5)^x$, etc
  - Logarithm (inverse of exponential) has a different domain.

Functions

- A function is a mapping f from a set X (the domain) to a set Y (the codomain) such that every $x \in X$ maps to a unique $y \in Y$.
- Important functions from $\mathbb{R}$ to $\mathbb{R}$:
  - Power functions $f(x) = x$, $f(x) = x^2$, $f(x) = x^3$, etc
  - Exponential functions $f(x) = 2^x$, $f(x) = (1.5)^x$, etc
  - Logarithm (inverse of exponential) has a different domain.
- Ceiling rounds up to nearest integer, e.g. $\lceil 3.7 \rceil = 4$. Floor rounds down, e.g. $\lfloor 3.7 \rfloor = 3 = \lfloor 3 \rfloor$.

Basic properties of important functions

- For $a > 1$ the exponential $f(x) = a^x$ is increasing and positive, and satisfies $a^{x+y} = a^x a^y$ for all $x, y \in \mathbb{R}$.

Basic properties of important functions

- For $a > 1$ the exponential $f(x) = a^x$ is increasing and positive, and satisfies $a^{x+y} = a^x a^y$ for all $x, y \in \mathbb{R}$.
- For $a > 1$ the logarithm $\log_a$ is increasing and satisfies $\log_a(xy) = \log_a x + \log_a y$ for all $x, y > 0$.

Basic properties of important functions

- For $a > 1$ the exponential $f(x) = a^x$ is increasing and positive, and satisfies $a^{x+y} = a^x a^y$ for all $x, y \in \mathbb{R}$.
- For $a > 1$ the logarithm $\log_a$ is increasing and satisfies $\log_a(xy) = \log_a x + \log_a y$ for all $x, y > 0$.
- We write $\ln = \log_e$ and $\lg = \log_2$. Note that $\log_a x = \log_a b \log_b x$.

Basic properties of important functions

- For $a > 1$ the exponential $f(x) = a^x$ is increasing and positive, and satisfies $a^{x+y} = a^x a^y$ for all $x, y \in \mathbb{R}$.
- For $a > 1$ the logarithm $\log_a$ is increasing and satisfies $\log_a(xy) = \log_a x + \log_a y$ for all $x, y > 0$.
- We write $\ln = \log_e$ and $\lg = \log_2$. Note that $\log_a x = \log_a b \log_b x$.
- Derivatives: $\frac{d}{dx} e^x = e^x$, $\frac{d}{dx} \ln x = \frac{1}{x}$.

Sums

- A sequence is a function $f : \mathbb{N} \to \mathbb{R}$.

Sums

- A sequence is a function $f : \mathbb{N} \to \mathbb{R}$.
- Notation: $f_0, f_1, f_2, \cdots$ where $f_i = f(i)$.

Sums

- A sequence is a function $f : \mathbb{N} \to \mathbb{R}$.
- Notation: $f_0, f_1, f_2, \cdots$ where $f_i = f(i)$.
- Sum: $f_m + f_{m+1} + \cdots + f_n = \sum_{i=m}^{n} f_i$.

Sums

- A sequence is a function $f : \mathbb{N} \to \mathbb{R}$.
- Notation: $f_0, f_1, f_2, \cdots$ where $f_i = f(i)$.
- Sum: $f_m + f_{m+1} + \cdots + f_n = \sum_{i=m}^{n} f_i$.
- Important sums:

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

$$\sum_{i=m}^{n} a^i = \frac{a^{n+1} - a^m}{a-1}.$$

Part 1

Asymptotic Analysis of Algorithms

Warm call: Why should we analyse algorithms?

What is an algorithm?

- An algorithm is a sequence of clearly stated rules that specify a step-by-step method for solving a given problem.

What is an algorithm?

- An algorithm is a sequence of clearly stated rules that specify a step-by-step method for solving a given problem.
- The rules should be unambiguous and sufficiently detailed that they can be carried out without creativity.

What is an algorithm?

- An algorithm is a sequence of clearly stated rules that specify a step-by-step method for solving a given problem.
- The rules should be unambiguous and sufficiently detailed that they can be carried out without creativity.
- Examples of algorithms: a (sufficiently detailed) cake recipe, primary school method for multiplication of decimal integers; quicksort.

What is an algorithm?

- An algorithm is a sequence of clearly stated rules that specify a step-by-step method for solving a given problem.
- The rules should be unambiguous and sufficiently detailed that they can be carried out without creativity.
- Examples of algorithms: a (sufficiently detailed) cake recipe, primary school method for multiplication of decimal integers; quicksort.
- Algorithms predate electronic computers by thousands of years (example: Euclid's greatest common divisor algorithm).

What is an algorithm?

- An algorithm is a sequence of clearly stated rules that specify a step-by-step method for solving a given problem.
- The rules should be unambiguous and sufficiently detailed that they can be carried out without creativity.
- Examples of algorithms: a (sufficiently detailed) cake recipe, primary school method for multiplication of decimal integers; quicksort.
- Algorithms predate electronic computers by thousands of years (example: Euclid's greatest common divisor algorithm).
- A program is a sequence of computer instructions implementing the algorithm.

Why analyse an algorithm?

- Experience shows that enormously more performance gains can be achieved by optimizing the algorithm than by optimizing other factors such as:

Why analyse an algorithm?

- Experience shows that enormously more performance gains can be achieved by optimizing the algorithm than by optimizing other factors such as:
  - processor

Why analyse an algorithm?

- Experience shows that enormously more performance gains can be achieved by optimizing the algorithm than by optimizing other factors such as:
  - processor
  - language

Why analyse an algorithm?

- Experience shows that enormously more performance gains can be achieved by optimizing the algorithm than by optimizing other factors such as:
  - processor
  - language
  - compiler

Why analyse an algorithm?

- Experience shows that enormously more performance gains can be achieved by optimizing the algorithm than by optimizing other factors such as:
  - processor
  - language
  - compiler
  - human programmer

Why analyse an algorithm?

- Experience shows that enormously more performance gains can be achieved by optimizing the algorithm than by optimizing other factors such as:
  - processor
  - language
  - compiler
  - human programmer
- The analysis process often results in us discovering simpler algorithms.

Why analyse an algorithm?

- Experience shows that enormously more performance gains can be achieved by optimizing the algorithm than by optimizing other factors such as:
  - processor
  - language
  - compiler
  - human programmer
- The analysis process often results in us discovering simpler algorithms.
- Many algorithms have parameters that must be set before implementation. Analysis allows us to set the optimal values.

Why analyse an algorithm?

- Experience shows that enormously more performance gains can be achieved by optimizing the algorithm than by optimizing other factors such as:
  - processor
  - language
  - compiler
  - human programmer
- The analysis process often results in us discovering simpler algorithms.
- Many algorithms have parameters that must be set before implementation. Analysis allows us to set the optimal values.
- Algorithms that have not been analysed for correctness often lead to major bugs in programs.

Fibonacci numbers

This sequence is recursively defined by

$$F(n) = \begin{cases} n & \text{if } n = 0 \text{ or } n = 1; \\ F(n-1) + F(n-2) & \text{if } n \geq 2. \end{cases}$$

This immediately suggests a recursive algorithm.

Algorithm 1 Slow method for computing Fibonacci numbers

```
1: function SLOWFIB(integer n)
2:     if n < 0 then return 0
3:     else if n = 0 then return 0
4:     else if n = 1 then return 1
5:     else return SLOWFIB(n − 1) + SLOWFIB(n − 2)
```

Improving over `slowfib`

- The algorithm `slowfib` is obviously correct, but does a lot of repeated computation. With a small (fixed) amount of extra space, we can do better, by working from the bottom up instead of from the top down.

Algorithm 2 Fast method for computing Fibonacci numbers

```
1: function FASTFIB(integer n)
2:     if n < 0 then return 0
3:     else if n = 0 then return 0
4:     else if n = 1 then return 1
5:     else
6:         a ← 1                          ▷ stores F(i) at bottom of loop
7:         b ← 0                          ▷ stores F(i − 1) at bottom of loop
8:         for i ← 2 to n do
9:             t ← a
10:            a ← a + b
11:            b ← t
12:    return a
```

Analysis of the fast algorithm

- Even a bad implementation in a slow interpreted language on an ancient machine of `fastfib` will beat the best implementation of `slowfib`, once n becomes big enough.

Colab time

- Implement `fastfib` and `slowfib` in Python. Which is faster for which values of n? What is the maximum value of n for which each gives a result in a reasonable time?

How to measure running time?

Basic performance measures

There are three main characteristics of an algorithm designed to solve a given problem.

- Domain of definition: the set of legal inputs.

Basic performance measures

There are three main characteristics of an algorithm designed to solve a given problem.

- Domain of definition: the set of legal inputs.
- Correctness: it gives correct output for each legal input. This depends on the problem we are trying to solve, and can be tricky to prove.

Basic performance measures

There are three main characteristics of an algorithm designed to solve a given problem.

- Domain of definition: the set of legal inputs.
- Correctness: it gives correct output for each legal input. This depends on the problem we are trying to solve, and can be tricky to prove.
- Resource use: usually computing time and memory space.

Basic performance measures

There are three main characteristics of an algorithm designed to solve a given problem.

- Domain of definition: the set of legal inputs.
- Correctness: it gives correct output for each legal input. This depends on the problem we are trying to solve, and can be tricky to prove.
- Resource use: usually computing time and memory space.
  - This depends on the input, and on the implementation (hardware, programmer skill, compiler, language, ...).

Basic performance measures

There are three main characteristics of an algorithm designed to solve a given problem.

- Domain of definition: the set of legal inputs.
- Correctness: it gives correct output for each legal input. This depends on the problem we are trying to solve, and can be tricky to prove.
- Resource use: usually computing time and memory space.
  - This depends on the input, and on the implementation (hardware, programmer skill, compiler, language, ...).
  - It usually grows as the input size grows.

Basic performance measures

There are three main characteristics of an algorithm designed to solve a given problem.

- Domain of definition: the set of legal inputs.
- Correctness: it gives correct output for each legal input. This depends on the problem we are trying to solve, and can be tricky to prove.
- Resource use: usually computing time and memory space.
  - This depends on the input, and on the implementation (hardware, programmer skill, compiler, language, ...).
  - It usually grows as the input size grows.
  - There is a tradeoff between resources (for example, time vs space).

Basic performance measures

There are three main characteristics of an algorithm designed to solve a given problem.

- Domain of definition: the set of legal inputs.
- Correctness: it gives correct output for each legal input. This depends on the problem we are trying to solve, and can be tricky to prove.
- Resource use: usually computing time and memory space.
  - This depends on the input, and on the implementation (hardware, programmer skill, compiler, language, ...).
  - It usually grows as the input size grows.
  - There is a tradeoff between resources (for example, time vs space).
  - Running time is usually more important than space use.

Basic performance measures

There are three main characteristics of an algorithm designed to solve a given problem.

- Domain of definition: the set of legal inputs.
- Correctness: it gives correct output for each legal input. This depends on the problem we are trying to solve, and can be tricky to prove.
- Resource use: usually computing time and memory space.
  - This depends on the input, and on the implementation (hardware, programmer skill, compiler, language, ...).
  - It usually grows as the input size grows.
  - There is a tradeoff between resources (for example, time vs space).
  - Running time is usually more important than space use.

Basic performance measures

There are three main characteristics of an algorithm designed to solve a given problem.

- Domain of definition: the set of legal inputs.
- Correctness: it gives correct output for each legal input. This depends on the problem we are trying to solve, and can be tricky to prove.
- Resource use: usually computing time and memory space.
  - This depends on the input, and on the implementation (hardware, programmer skill, compiler, language, ...).
  - It usually grows as the input size grows.
  - There is a tradeoff between resources (for example, time vs space).
  - Running time is usually more important than space use.

In this lecture we mainly consider how to estimate resource use of an algorithm, ignoring implementation as much as possible.

Warm call: How to compare algorithms?

- Given an algorithm $\mathcal{A}$, the actual running time on a given input ι depends on many implementation details. Can we compare algorithms when we do not know the exact input and details of implementation?

Warm call: How to compare algorithms?

- Given an algorithm $\mathcal{A}$, the actual running time on a given input ι depends on many implementation details. Can we compare algorithms when we do not know the exact input and details of implementation?
- The running time usually grows with the size of the input. Running time for very small inputs is not usually important; it is large inputs that cause problems if the algorithm is inefficient.

Running time: input

- We define a notion of input size on the data. This is a positive integer.

Running time: input

- We define a notion of input size on the data. This is a positive integer.
- Example: number of records in a database to be sorted.

Elementary operations

- We use the concept of elementary operation as our basic measuring unit of running time. This is any operation whose execution time does not depend on the size of the input.

Elementary operations

- We use the concept of elementary operation as our basic measuring unit of running time. This is any operation whose execution time does not depend on the size of the input.
- The running time $T(\iota)$ of algorithm $\mathcal{A}$ on input ι is the number of elementary operations used when ι is fed into $\mathcal{A}$.

Running time		**Input size**			
Function	*Notation*	10	100	1000	10^7
Constant	1	1	1	1	1
Logarithmic	$\log n$	1	2	3	7
Linear	n	1	10	100	10^6
"Linearithmic"	$n \log n$	1	20	300	7×10^6
Quadratic	n^2	1	100	10000	10^{12}
Cubic	n^3	1	1000	10^6	10^{18}
Exponential	2^n	1	10^{27}	10^{298}	$10^{3010296}$

Note: there are about 3×10^{18} nanoseconds in a century.

Warm call

- Algorithm A takes n^2 elementary operations to sort a file of n lines, while Algorithm B takes $50n \log n$. Which algorithm is better when $n = 10$?

Warm call

- Algorithm A takes n^2 elementary operations to sort a file of n lines, while Algorithm B takes $50n \log n$. Which algorithm is better when $n = 10$?
- when $n = 10^6$? How do we decide which algorithm to use?

How to measure running time?

Running time techniques: easy

- Running time of disjoint blocks adds.

Running time techniques: easy

- Running time of disjoint blocks adds.
- Running time of nested loops with non-interacting variables multiplies.

Running time techniques: easy

- Running time of disjoint blocks adds.
- Running time of nested loops with non-interacting variables multiplies.
- Example: single, double, triple loops with fixed number of elementary operations inside the inner loop yields linear, quadratic, cubic running time.

Algorithm 3 Swapping two elements in an array

Require: $0 \leq i \leq j \leq n-1$

function SWAP(array $a[0..n-1]$, integer i, integer j)

$t \leftarrow a[i]$

$a[i] \leftarrow a[j]$

$a[j] \leftarrow t$

return a

- Running time?

Algorithm 3 Swapping two elements in an array

Require: $0 \leq i \leq j \leq n-1$

function SWAP(array $a[0..n-1]$, integer i, integer j)

$t \leftarrow a[i]$

$a[i] \leftarrow a[j]$

$a[j] \leftarrow t$

return a

- Running time?
- This is a constant time algorithm.

Algorithm 5 Finding the maximum in an array

```
function FINDMAX(array a[0..n − 1])
    k ← 0                                  ▷ location of maximum so far
    for j ← 1 to n − 1 do
        if a[k] < a[j] then
            k = j
    return k
```

- Running time?

Algorithm 5 Finding the maximum in an array

```
function FINDMAX(array a[0..n − 1])
    k ← 0                                  ▷ location of maximum so far
    for j ← 1 to n − 1 do
        if a[k] < a[j] then
            k = j
    return k
```

- Running time?
- This is a linear time algorithm, since it makes one pass through the array and does a constant amount of work each time.

Snippet: other loop increments

Algorithm 7 Example: exponential change of variable in loop

$i \leftarrow 1$
while $i \leq n$ **do**
    $i \leftarrow 2 * i$
    print i

- Running time?

Snippet: other loop increments

Algorithm 7 Example: exponential change of variable in loop

$i \leftarrow 1$
while $i \leq n$ **do**
    $i \leftarrow 2 * i$
    print i

- Running time?
- This runs in logarithmic time because i doubles about $\lg n$ times until reaching n.

Example: nested loops

Algorithm 9 Snippet: Nested loops

```
for i ← 1 to n do
    for j ← i to n do
        print i + j
```

- Running time?

Example: nested loops

Algorithm 9 Snippet: Nested loops

```
for i ← 1 to n do
    for j ← i to n do
        print i + j
```

- Running time?
- The first iteration of the outer loop takes n elementary operations. The second iteration of the outer loop takes $n-1$ operations and so forth. Therefore, the algorithm takes $n + (n-1) + \cdots + 1 = n(n+1)/2$ elementary operations for input size n.

Warm call

- What do we know about the running time $T(n)$ of `slowfib` for term n of the Fibonacci sequence?

- `slowfib` makes $F(n)$ function calls each of which involves a constant number of elementary operations. It turns out that $F(n)$ grows exponentially in n, so this is an exponential time algorithm.

Asymptotic notation

Asymptotic comparison of functions

- In order to compare running times of algorithms we want a way of comparing the growth rates of functions.

Asymptotic comparison of functions

- In order to compare running times of algorithms we want a way of comparing the growth rates of functions.
- We want to see what happens for large values of n — small ones are not relevant.

Asymptotic comparison of functions

- In order to compare running times of algorithms we want a way of comparing the growth rates of functions.
- We want to see what happens for large values of n — small ones are not relevant.
- We are not usually interested in constant factors and only want to consider the dominant term.

Asymptotic comparison of functions

- In order to compare running times of algorithms we want a way of comparing the growth rates of functions.
- We want to see what happens for large values of n — small ones are not relevant.
- We are not usually interested in constant factors and only want to consider the dominant term.
- The standard mathematical approach is to use asymptotic notation O, Ω, Θ which we will now describe.

Big-O notation

- Suppose that f and g are functions from $\mathbb{N}$ to $\mathbb{R}$, which take on nonnegative values.

Big-O notation

- Suppose that f and g are functions from $\mathbb{N}$ to $\mathbb{R}$, which take on nonnegative values.
  - Say f is $O(g)$ ("f is Big-Oh of g") if there is some $C > 0$ and some $n_0 \in \mathbb{N}$ such that for all $n \geq n_0$, $f(n) \leq Cg(n)$. Informally, f grows at most as fast as g.

Big-O notation

- Suppose that f and g are functions from $\mathbb{N}$ to $\mathbb{R}$, which take on nonnegative values.
  - Say f is $O(g)$ ("f is Big-Oh of g") if there is some $C > 0$ and some $n_0 \in \mathbb{N}$ such that for all $n \geq n_0$, $f(n) \leq Cg(n)$. Informally, f grows at most as fast as g.
  - Say f is $\Omega(g)$ ("f is big-Omega of g") if g is $O(f)$. Informally, f grows at least as fast as g.

Big-O notation

- Suppose that f and g are functions from $\mathbb{N}$ to $\mathbb{R}$, which take on nonnegative values.
  - Say f is $O(g)$ ("f is Big-Oh of g") if there is some $C > 0$ and some $n_0 \in \mathbb{N}$ such that for all $n \geq n_0$, $f(n) \leq Cg(n)$.
    Informally, f grows at most as fast as g.
  - Say f is $\Omega(g)$ ("f is big-Omega of g") if g is $O(f)$.
    Informally, f grows at least as fast as g.
  - Say f is $\Theta(g)$ ("f is big-Theta of g") if f is $O(g)$ and g is $O(f)$.
    Informally, f grows at the same rate as g.

Big-O notation

- Suppose that f and g are functions from $\mathbb{N}$ to $\mathbb{R}$, which take on nonnegative values.
  - Say f is $O(g)$ ("f is Big-Oh of g") if there is some $C > 0$ and some $n_0 \in \mathbb{N}$ such that for all $n \geq n_0$, $f(n) \leq Cg(n)$.
    Informally, f grows at most as fast as g.
  - Say f is $\Omega(g)$ ("f is big-Omega of g") if g is $O(f)$.
    Informally, f grows at least as fast as g.
  - Say f is $\Theta(g)$ ("f is big-Theta of g") if f is $O(g)$ and g is $O(f)$.
    Informally, f grows at the same rate as g.
- Note that we could always reduce n_0 at the expense of a bigger C but it is often easier not to.

Asymptotic comparison — examples

- Every linear function $f(n) = an + b$, $a > 0$, is $O(n)$.

Asymptotic comparison — examples

- Every linear function $f(n) = an + b$, $a > 0$, is $O(n)$.
- Proof: $an + b \leq an + |b| \leq (a + |b|)n$ for $n \geq 1$.

What happens if there are many inputs of a given size?

- We usually don't want to have to consider the distribution of running time over all possible inputs of a given size. There may be (infinitely) many inputs of a given size, and running time may vary widely on these.

What happens if there are many inputs of a given size?

- We usually don't want to have to consider the distribution of running time over all possible inputs of a given size. There may be (infinitely) many inputs of a given size, and running time may vary widely on these.
- For example, for sorting the integers $1, \ldots, n$, there are $n!$ possible inputs, and this is large even for $n = 10$.

What happens if there are many inputs of a given size?

- We usually don't want to have to consider the distribution of running time over all possible inputs of a given size. There may be (infinitely) many inputs of a given size, and running time may vary widely on these.
- For example, for sorting the integers $1, \ldots, n$, there are $n!$ possible inputs, and this is large even for $n = 10$.
- We consider statistics of $T(\iota)$ such as worst-case $W(n)$ or average-case $A(n)$ running time for instances ι of size n.

What are the pros and cons of worst and average case analysis?

- Worst-case bounds are valid for all instances: this is important for mission-critical applications.

What are the pros and cons of worst and average case analysis?

- Worst-case bounds are valid for all instances: this is important for mission-critical applications.
- Worst-case bounds are often easier to derive mathematically.

What are the pros and cons of worst and average case analysis?

- Worst-case bounds are valid for all instances: this is important for mission-critical applications.
- Worst-case bounds are often easier to derive mathematically.
- Worst-case bounds often hugely exceed typical running time and have little predictive or comparative value.

What are the pros and cons of worst and average case analysis?

- Worst-case bounds are valid for all instances: this is important for mission-critical applications.
- Worst-case bounds are often easier to derive mathematically.
- Worst-case bounds often hugely exceed typical running time and have little predictive or comparative value.
- Average-case running time is often more realistic. Quicksort is a classic example.

What are the pros and cons of worst and average case analysis?

- Worst-case bounds are valid for all instances: this is important for mission-critical applications.
- Worst-case bounds are often easier to derive mathematically.
- Worst-case bounds often hugely exceed typical running time and have little predictive or comparative value.
- Average-case running time is often more realistic. Quicksort is a classic example.
- Average-case analysis requires a good understanding of the probability distribution of the inputs.

What are the pros and cons of worst and average case analysis?

- Worst-case bounds are valid for all instances: this is important for mission-critical applications.
- Worst-case bounds are often easier to derive mathematically.
- Worst-case bounds often hugely exceed typical running time and have little predictive or comparative value.
- Average-case running time is often more realistic. Quicksort is a classic example.
- Average-case analysis requires a good understanding of the probability distribution of the inputs.
- Conclusion: a good worst-case bound is always useful, but it is just a first step and we should aim to refine the analysis for important algorithms. Average-case analysis is often more practically useful, provided the algorithm will be run on "random" data.

Why can constants often be ignored?

- A linear time algorithm when implemented will take at most $An + B$ seconds to run on an instance of size n, for some implementation-specific constants A, B.

Why can constants often be ignored?

- A linear time algorithm when implemented will take at most $An + B$ seconds to run on an instance of size n, for some implementation-specific constants A, B.
- For large n, this is well approximated by An. Small n are not usually of interest anyway, since almost any algorithm is good enough for tiny instances.

Why can constants often be ignored?

- A linear time algorithm when implemented will take at most $An + B$ seconds to run on an instance of size n, for some implementation-specific constants A, B.
- For large n, this is well approximated by An. Small n are not usually of interest anyway, since almost any algorithm is good enough for tiny instances.
- No matter what A is, we can easily work out how the running time scales with increasing problem size (linearly!).

Why can constants often be ignored?

- A linear time algorithm when implemented will take at most $An + B$ seconds to run on an instance of size n, for some implementation-specific constants A, B.
- For large n, this is well approximated by An. Small n are not usually of interest anyway, since almost any algorithm is good enough for tiny instances.
- No matter what A is, we can easily work out how the running time scales with increasing problem size (linearly!).
- The difference between a linear and a quadratic time algorithm is usually huge, no matter what the constants are. For large enough n, a linear time algorithm will always beat a quadratic time one.

Why can constants often be ignored?

- A linear time algorithm when implemented will take at most $An + B$ seconds to run on an instance of size n, for some implementation-specific constants A, B.
- For large n, this is well approximated by An. Small n are not usually of interest anyway, since almost any algorithm is good enough for tiny instances.
- No matter what A is, we can easily work out how the running time scales with increasing problem size (linearly!).
- The difference between a linear and a quadratic time algorithm is usually huge, no matter what the constants are. For large enough n, a linear time algorithm will always beat a quadratic time one.
- Conclusion: in practice we often need to make only crude distinctions. We only need to know whether the running time scales like $n, n^2, n^3, n \log n, 2^n, \ldots$. If we need finer distinctions, we can do more analysis.

Can we always ignore constants?

- When we want to choose between two good algorithms for the same problem ("is my linear-time algorithm faster than your linear-time algorithm?"), we may need to know constants. These must be determined empirically.

Can we always ignore constants?

- When we want to choose between two good algorithms for the same problem ("is my linear-time algorithm faster than your linear-time algorithm?"), we may need to know constants. These must be determined empirically.
- For important algorithms that will be used many times, it is worth being more precise about the constants. Even small savings will be worth the trouble.

Can we always ignore constants?

- When we want to choose between two good algorithms for the same problem ("is my linear-time algorithm faster than your linear-time algorithm?"), we may need to know constants. These must be determined empirically.
- For important algorithms that will be used many times, it is worth being more precise about the constants. Even small savings will be worth the trouble.
- An algorithm with running time $10^{-10}n^2$ is probably better in practice than one with running time $10^{10}n$, since the latter will eventually beat the former, but only on instances of size at least 10^{20}, which is rarely met in practice.

Can we always ignore constants?

- When we want to choose between two good algorithms for the same problem ("is my linear-time algorithm faster than your linear-time algorithm?"), we may need to know constants. These must be determined empirically.
- For important algorithms that will be used many times, it is worth being more precise about the constants. Even small savings will be worth the trouble.
- An algorithm with running time $10^{-10}n^2$ is probably better in practice than one with running time $10^{10}n$, since the latter will eventually beat the former, but only on instances of size at least 10^{20}, which is rarely met in practice.
- Conclusion: we should have at least a rough feel for the constants of competing algorithms. However, in practice the constants are usually of moderate size.

Summary

- Our goal is to find an asymptotic approximation for the (worst or average case) running time of a given algorithm. Ideally we can find a simple function f and prove that the running time is $\Theta(f(n))$.

Summary

- Our goal is to find an asymptotic approximation for the (worst or average case) running time of a given algorithm. Ideally we can find a simple function f and prove that the running time is $\Theta(f(n))$.
- The main $f(n)$ occurring in applications are $\log n, n, n \log n, n^2, n^3, 2^n$, and each grows considerably faster than the previous one. The gap between n and $n \log n$ is the smallest.

Part 2

The problem of sorting

Sorting

- Given n keys $a_1, \ldots, a_n$ from a totally ordered set, put them in increasing order. The keys may be just part of a larger data record.

Sorting

- Given n keys $a_1, \ldots, a_n$ from a totally ordered set, put them in increasing order. The keys may be just part of a larger data record.
- Common examples: integers with $\leq$, words with alphabetical (lexicographic) order. More complicated examples can often be reduced to these.

Sorting

- Given n keys $a_1, \ldots, a_n$ from a totally ordered set, put them in increasing order. The keys may be just part of a larger data record.
- Common examples: integers with $\leq$, words with alphabetical (lexicographic) order. More complicated examples can often be reduced to these.
- Sorting data makes many other problems easier, e.g. selection, finding duplicates. Sorting is ubiquitous in computing.

Analysis of sorting algorithms

- We use comparisons and swaps as our elementary operations.

Analysis of sorting algorithms

- We use comparisons and swaps as our elementary operations.
- We can swap the position of elements (at positions i and j say). Each swap requires 3 data moves (updates of variables).

Mergesort and quicksort

- Each algorithm splits the input list into two sublists, recursively sorts each sublist, and then combines the sorted sublists to sort the original list.

Mergesort and quicksort

- Each algorithm splits the input list into two sublists, recursively sorts each sublist, and then combines the sorted sublists to sort the original list.
- Mergesort (J. von Neumann, 1945): splitting is very easy, most of the work is in combining. We divide the list into left and right half, and merge the recursively sorted lists with a procedure `merge`.

Mergesort and quicksort

- Each algorithm splits the input list into two sublists, recursively sorts each sublist, and then combines the sorted sublists to sort the original list.
- Mergesort (J. von Neumann, 1945): splitting is very easy, most of the work is in combining. We divide the list into left and right half, and merge the recursively sorted lists with a procedure `merge`.
- Quicksort (C. A. R. Hoare, 1962): most of the work is in the splitting, combining is very easy. We choose a pivot element, and use a procedure `partition` to alter the list so that the pivot is in its correct position. Then the left and right sublists on each side of the pivot are sorted recursively.

Algorithm 11 Mergesort

function MERGESORT(list $a[0..n-1]$)
  if $n = 1$ **then**
    return a
  else
    $m \leftarrow \lfloor (n-1)/2 \rfloor$ ▷ median index of list
    $l \leftarrow \text{MERGESORT}(a, 0, m)$ ▷ sort left half
    $r \leftarrow \text{MERGESORT}(a, m+1, n-1)$ ▷ sort right half
  return $\text{MERGE}(l, r)$ ▷ merge both halves

Linear-time merging

- If L_1 and L_2 are sorted lists, they can be merged into one sorted list in linear time.

Linear-time merging

- If L_1 and L_2 are sorted lists, they can be merged into one sorted list in linear time.
  - Start pointers at the beginning of each list.

Linear-time merging

- If L_1 and L_2 are sorted lists, they can be merged into one sorted list in linear time.
  - Start pointers at the beginning of each list.
  - Compare the elements being pointed to and choose the lesser one to start the sorted list. Increment that pointer.

Linear-time merging

- If L_1 and L_2 are sorted lists, they can be merged into one sorted list in linear time.
  - Start pointers at the beginning of each list.
  - Compare the elements being pointed to and choose the lesser one to start the sorted list. Increment that pointer.
  - Iterate until one pointer reaches the end of its list, then copy remainder of other list to the end of the sorted list.

Linear-time merging

- If L_1 and L_2 are sorted lists, they can be merged into one sorted list in linear time.
  - Start pointers at the beginning of each list.
  - Compare the elements being pointed to and choose the lesser one to start the sorted list. Increment that pointer.
  - Iterate until one pointer reaches the end of its list, then copy remainder of other list to the end of the sorted list.
- In any case the number of comparisons is in $\Theta(n)$.

Mergesort analysis

- The number of comparisons C_n satisfies (approximately)

$$C_n = \begin{cases} C_{\lceil n/2 \rceil} + C_{\lfloor n/2 \rfloor} + n & \text{if } n > 1; \\ 0 & \text{if } n = 1. \end{cases}$$

Mergesort analysis

- The number of comparisons C_n satisfies (approximately)

$$C_n = \begin{cases} C_{\lceil n/2 \rceil} + C_{\lfloor n/2 \rfloor} + n & \text{if } n > 1; \\ 0 & \text{if } n = 1. \end{cases}$$

- Thus if n is a power of 2 we have $C_n = 2C(n/2) + n$. We need to solve this recurrence!

Solving mergesort recurrence

- Consider the mergesort recurrence $C(n) = 2C(n/2) + n$ (makes sense for n being a power of 2, $n = 2^k$).

Solving mergesort recurrence

- Consider the mergesort recurrence $C(n) = 2C(n/2) + n$ (makes sense for n being a power of 2, $n = 2^k$).
- We can apply the recurrence on itself to find a general pattern.

Solving mergesort recurrence

- Consider the mergesort recurrence $C(n) = 2C(n/2) + n$ (makes sense for n being a power of 2, $n = 2^k$).
- We can apply the recurrence on itself to find a general pattern.
- Applying 2 steps of recurrence $C(n/2) = 2C(n/4) + n/2$, $C(n) =$

Solving mergesort recurrence

- Consider the mergesort recurrence $C(n) = 2C(n/2) + n$ (makes sense for n being a power of 2, $n = 2^k$).
- We can apply the recurrence on itself to find a general pattern.
- Applying 2 steps of recurrence $C(n/2) = 2C(n/4) + n/2$, $C(n) =$
- Applying 3 steps of recurrence $C(n/4) = 2C(n/8) + n/4$, $C(n) =$

Solving mergesort recurrence

- Consider the mergesort recurrence $C(n) = 2C(n/2) + n$ (makes sense for n being a power of 2, $n = 2^k$).
- We can apply the recurrence on itself to find a general pattern.
- Applying 2 steps of recurrence $C(n/2) = 2C(n/4) + n/2$, $C(n) =$
- Applying 3 steps of recurrence $C(n/4) = 2C(n/8) + n/4$, $C(n) =$
- If we continue this for k steps, we would get $C(n) = 2^k C(n/2^k) + nk$

Solving mergesort recurrence

- Consider the mergesort recurrence $C(n) = 2C(n/2) + n$ (makes sense for n being a power of 2, $n = 2^k$).
- We can apply the recurrence on itself to find a general pattern.
- Applying 2 steps of recurrence $C(n/2) = 2C(n/4) + n/2$, $C(n) =$
- Applying 3 steps of recurrence $C(n/4) = 2C(n/8) + n/4$, $C(n) =$
- If we continue this for k steps, we would get $C(n) = 2^k C(n/2^k) + nk$
- Replacing the terms with k based on terms based on n, we get $C(n) = nC(1) + n \log n = n \log n$

Running time of mergesort

- We proved that the running time of mergesort for input size n is $O(n \log n)$.

Quicksort

Algorithm 12 Quicksort - basic

Require: $0 \leq i \leq j \leq n-1$

function QUICKSORT(list $a[0..n-1]$, integer i, integer j)

if $i < j$ **then**

$q \leftarrow \text{PARTITION}(a, i, j)$ ▷ put pivot in correct position

$\text{QUICKSORT}(a, i, q-1)$ ▷ sort left half

$\text{QUICKSORT}(a, q+1, j)$ ▷ sort right half

Linear time partitioning

- Given a list L and an element p of L called the pivot, we can partition so that all elements to the left of L are $\leq p$ and all to the right are $\geq p$. One method always using the first element as the pivot (Hoare, 1960) — there are others:

Linear time partitioning

- Given a list L and an element p of L called the pivot, we can partition so that all elements to the left of L are $\leq p$ and all to the right are $\geq p$. One method always using the first element as the pivot (Hoare, 1960) — there are others:
  - Start with pointers at opposite ends of the list. Stop when pointers cross.

Linear time partitioning

- Given a list L and an element p of L called the pivot, we can partition so that all elements to the left of L are $\leq p$ and all to the right are $\geq p$. One method always using the first element as the pivot (Hoare, 1960) — there are others:
  - Start with pointers at opposite ends of the list. Stop when pointers cross.
  - At each step, increment the left pointer until we reach an element $\geq p$, and decrement the right one until we reach an element $\leq p$.

Linear time partitioning

- Given a list L and an element p of L called the pivot, we can partition so that all elements to the left of L are $\leq p$ and all to the right are $\geq p$. One method always using the first element as the pivot (Hoare, 1960) — there are others:
  - Start with pointers at opposite ends of the list. Stop when pointers cross.
  - At each step, increment the left pointer until we reach an element $\geq p$, and decrement the right one until we reach an element $\leq p$.
  - Swap these elements and continue. When pointers cross, swap right pointee with pivot.

Algorithm 13 Partition - Hoare's method

Require: $0 \leq i \leq j \leq n-1$

function PARTITION(list $a[0..n-1]$, integer i, integer j)

```
p ← a[i]                                   ▷ pivot element
l ← i                                      ▷ left pointer
r ← j + 1                                  ▷ right pointer
while True do
    repeat
        l ← l + 1                          ▷ find big element
    until a[l] ≥ p
    repeat
        r ← r − 1                          ▷ find small element
    until a[r] ≤ p
    if l < r then
        swap(a, l, r)                      ▷ swap big and small elements
    else
        swap(a, i, r)                      ▷ put pivot in correct place
        return r
```

Quicksort analysis

- The number of comparisons satisfies a recurrence like $C_n = C_{p-1} + C_{n-p} + n$.

Quicksort analysis

- The number of comparisons satisfies a recurrence like $C_n = C_{p-1} + C_{n-p} + n$.
- If we are unlucky (pivot being at one of the two ends), this degenerates to $C_n = C_{n-1} + n$ and after finding the position of pivot, we still have a list of $n - 1$ elements.

Quicksort analysis

- The number of comparisons satisfies a recurrence like $C_n = C_{p-1} + C_{n-p} + n$.
- If we are unlucky (pivot being at one of the two ends), this degenerates to $C_n = C_{n-1} + n$ and after finding the position of pivot, we still have a list of $n - 1$ elements.
- In the worst case, we continue being unlucky in every recursive step and get quadratic running time (adding $n - 1$,$n - 2, \ldots$ all the way down).

Quicksort analysis

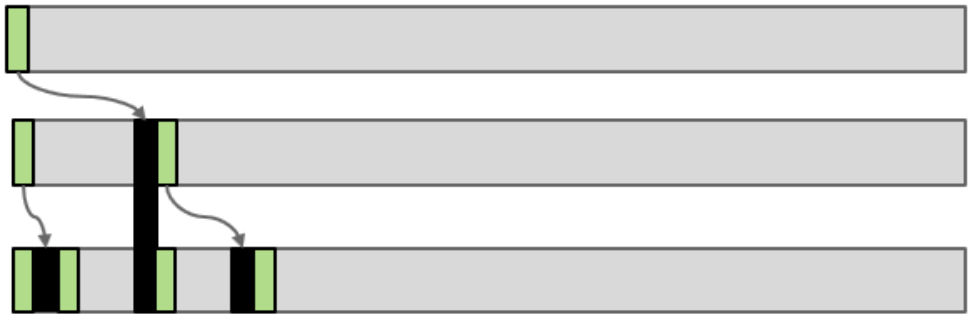
- The number of comparisons satisfies a recurrence like $C_n = C_{p-1} + C_{n-p} + n$.
- If we are unlucky (pivot being at one of the two ends), this degenerates to $C_n = C_{n-1} + n$ and after finding the position of pivot, we still have a list of $n - 1$ elements.
- In the worst case, we continue being unlucky in every recursive step and get quadratic running time (adding $n - 1$,$n - 2, \ldots$ all the way down).
- Therefore, the worst-case running time of quicksort is $O(n^2)$.

Quicksort analysis

- The number of comparisons satisfies a recurrence like $C_n = C_{p-1} + C_{n-p} + n$.
- If we are unlucky (pivot being at one of the two ends), this degenerates to $C_n = C_{n-1} + n$ and after finding the position of pivot, we still have a list of $n - 1$ elements.
- In the worst case, we continue being unlucky in every recursive step and get quadratic running time (adding $n - 1$,$n - 2, \ldots$ all the way down).
- Therefore, the worst-case running time of quicksort is $O(n^2)$.
- However, we can prove that if elements are distinct and all input permutations are equally likely, then quicksort has average running time in $O(n \log n)$.

Why quicksort average running time is $O(n \log n)$?

Suppose pivot always ends up at least 10% from either edge



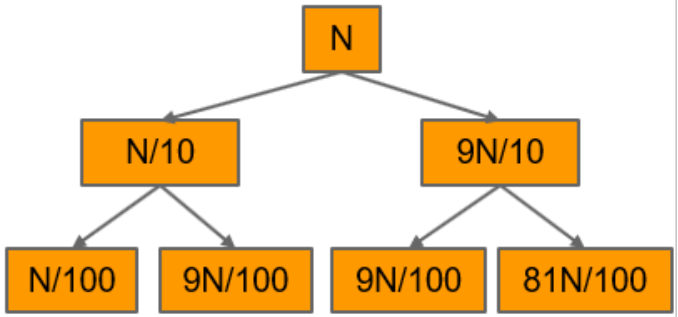
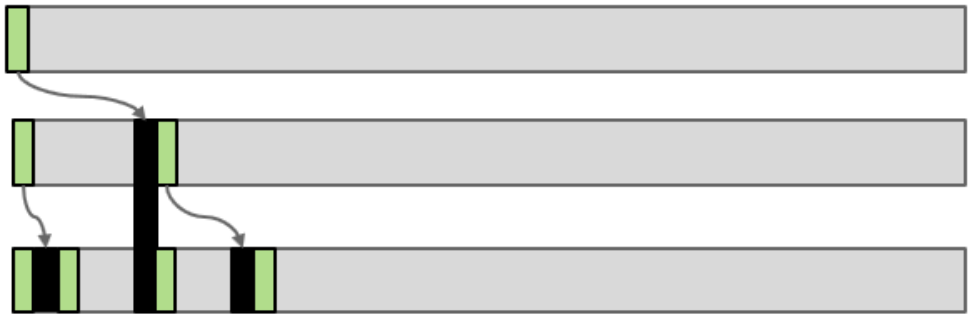


Image from Josh Hug from (`www.datastructur.es`)

- Work at each level: $O(n)$

Why quicksort average running time is $O(n \log n)$?

Suppose pivot always ends up at least 10% from either edge



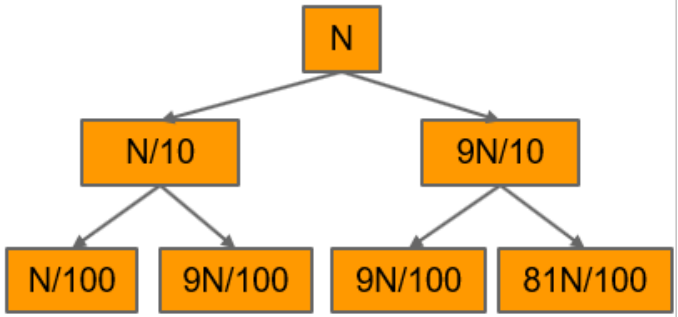


Image from Josh Hug from (`www.datastructur.es`)

- Work at each level: $O(n)$
- Total work: $O(nh)$ and the height h scales with $\log n$ (base $10/9$).

Why quicksort average running time is $O(n \log n)$?

Suppose pivot always ends up at least 10% from either edge

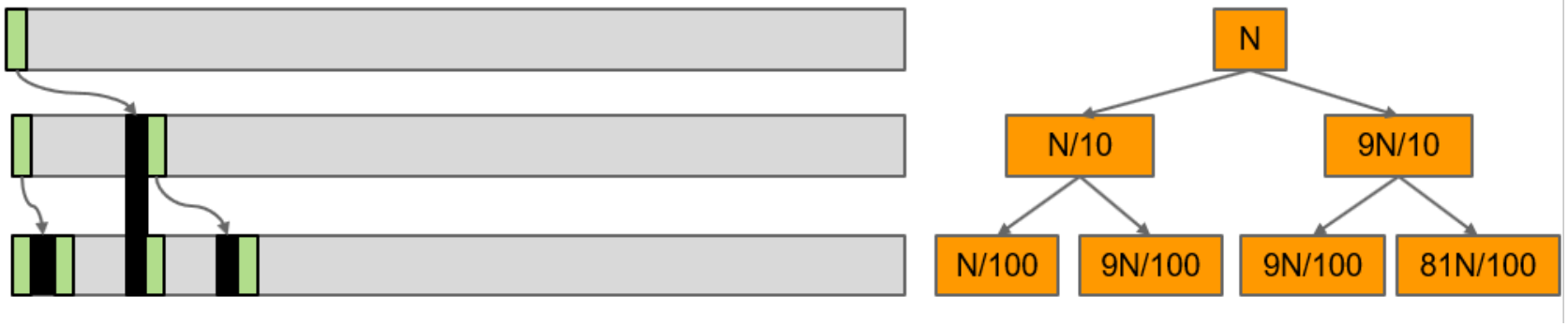


Image from Josh Hug from (`www.datastructur.es`)

- Work at each level: $O(n)$
- Total work: $O(nh)$ and the height h scales with $\log n$ (base $10/9$).
- Same argument works for pivot being always 1% from either edge or 0.1% and so on.

Why quicksort average running time is $O(n \log n)$?

Suppose pivot always ends up at least 10% from either edge

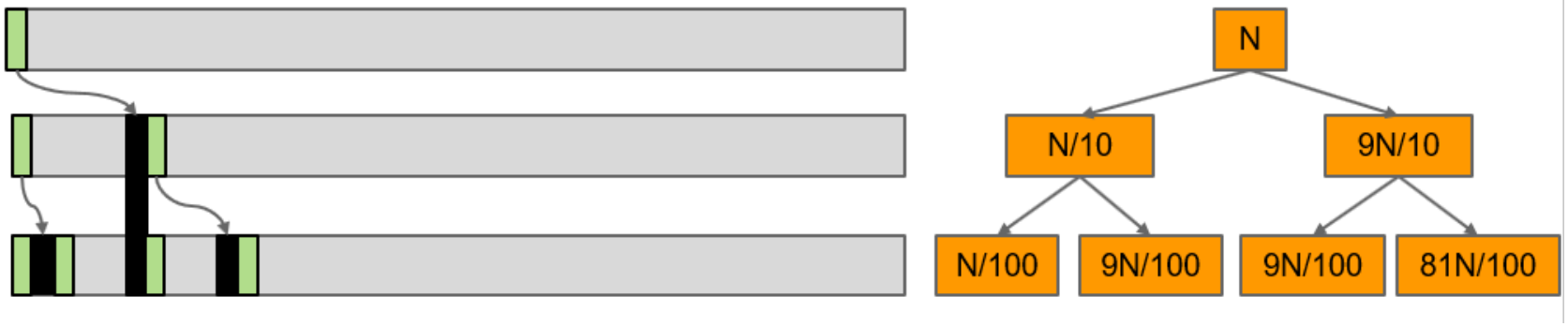


Image from Josh Hug from (`www.datastructur.es`)

- Work at each level: $O(n)$
- Total work: $O(nh)$ and the height h scales with $\log n$ (base $10/9$).
- Same argument works for pivot being always 1% from either edge or 0.1% and so on.
- Quicksort average running time is $O(n \log n)$

Comparison of divide and conquer sorting algorithms

Table: Characteristics of sorting methods

Method	Worst	Average	Best
Mergesort	$n \log n$	$n \log n$	$n \log n$
Quicksort	n^2	$n \log n$	$n \log n$

- Running times give asymptotic order only.
- Despite these theoretical results, quicksort is more efficient in practice.
- The Achilles' heel of quicksort is its worst-case.

No better comparison-based algorithm (from a worst-case perspective)

Table: Characteristics of sorting methods

Method	Worst	Average	Best
Mergesort	$n \log n$	$n \log n$	$n \log n$
Quicksort	n^2	$n \log n$	$n \log n$

- Lower bound analysis for the worst-case of any comparison-based sorting algorithm shows that there is a sequence of input which requires the algorithm to do $O(n \log n)$ elementary operations.
- Therefore, it is impossible to develop a comparison-based algorithm whose worst-case running time is $O(n)$.
- More on sorting at `https://youtu.be/kPRA0W1kECg?t=37`

Part 3

Dictionary Abstract Data Type (ADT)

Dictionary ADT

- An abstract data type that supports operations to insert, find, and delete an element with given search key.
- Used for databases. Other names are table ADT and associative array.
- There are many ways in which this could be implemented:
  - unsorted list;
  - sorted list;
  - binary search tree;
  - hash table.

Unsorted list

- Inserting an element is constant time.

Unsorted list

- Inserting an element is constant time.
- The only way to find an element is to check each element.

Unsorted list

- Inserting an element is constant time.
- The only way to find an element is to check each element.
- This takes time in $O(n)$ for any reasonable implementation.

Unsorted list

- Inserting an element is constant time.
- The only way to find an element is to check each element.
- This takes time in $O(n)$ for any reasonable implementation.
- Also, deletion is $O(n)$ because first we should find it and then other elements must be pushed to the left (array).

Sorted list

- In a sorted list, inserting an element is harder because you have to find the right place and therefore it is $O(n)$.

Sorted list

- In a sorted list, inserting an element is harder because you have to find the right place and therefore it is $O(n)$.
- In a sorted list, finding an element is easier because the order allows us to perform a binary search which is $O(\log n)$

Sorted list

- In a sorted list, inserting an element is harder because you have to find the right place and therefore it is $O(n)$.
- In a sorted list, finding an element is easier because the order allows us to perform a binary search which is $O(\log n)$
- Again, deletion is $O(n)$ because other elements must be pushed to the left (array).

Efficiency of various implementations of Dictionary ADT

Table: Average case running time (asymptotic order)

Data structure	Insert	Delete	Find
Unsorted list (array)	1	n	n
Sorted list (array)	n	n	$\log n$

Efficiency of various implementations of Dictionary ADT

Table: Average case running time (asymptotic order)

Data structure	Insert	Delete	Find
Unsorted list (array)	1	n	n
Sorted list (array)	n	n	$\log n$
Binary search tree	$\log n$	$\log n$	$\log n$

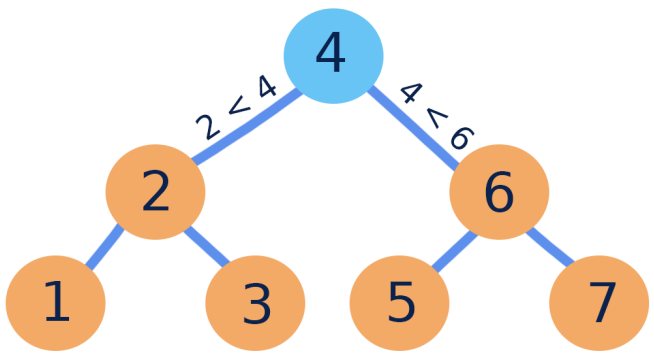


Image from Tamara Nelson-Fromm, UIUC
https://courses.engr.illinois.edu/cs225/sp2019/notes/bst/

Efficiency of various implementations of Dictionary ADT

Table: Average case running time (asymptotic order)

Data structure	Insert	Delete	Find
Unsorted list (array)	1	n	n
Sorted list (array)	n	n	$\log n$
Binary search tree	$\log n$	$\log n$	$\log n$
???	1	1	1

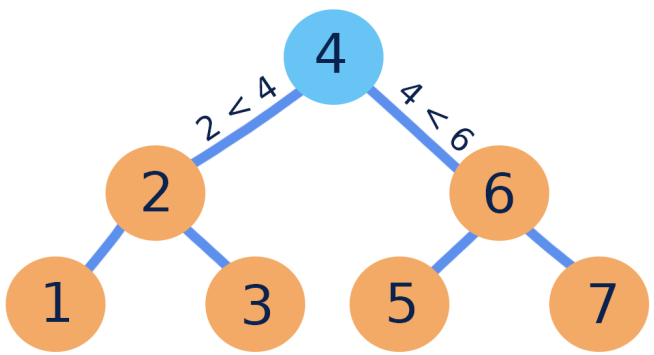


Image from Tamara Nelson-Fromm, UIUC
https://courses.engr.illinois.edu/cs225/sp2019/notes/bst/

Hashing

Hashing

- A hash function is a function h that outputs an integer value for each key. A hash table is an array implementation of the table ADT, where each key is mapped via a hash function to an array index.

Hashing

- A hash function is a function h that outputs an integer value for each key. A hash table is an array implementation of the table ADT, where each key is mapped via a hash function to an array index.
- The number of possible keys is usually enormously more than the actual number of keys. Thus allocating an array with enough size to fit all possible keys would be very inefficient. So hash functions are not 1-to-1; that is, two keys may be mapped to the same index (a collision).

Hash functions

- There are several desirable properties of a hash function:

Hash functions

- There are several desirable properties of a hash function:
  - it should be computable quickly (constant time).

Hash functions

- There are several desirable properties of a hash function:
  - it should be computable quickly (constant time).
  - if keys are drawn uniformly at random, then the hashed values should be uniformly distributed.

Hash functions

- There are several desirable properties of a hash function:
  - it should be computable quickly (constant time).
  - if keys are drawn uniformly at random, then the hashed values should be uniformly distributed.
  - keys that are "close" should have their hash values "spread out".

Hash functions

- There are several desirable properties of a hash function:
  - it should be computable quickly (constant time).
  - if keys are drawn uniformly at random, then the hashed values should be uniformly distributed.
  - keys that are "close" should have their hash values "spread out".
- A hash function should be deterministic, but appear "random" - in other words it should pass some statistical tests (similar to pseudorandom number generators).

Collision resolution policies

- We need a collision resolution policy to prescribe what to do when collisions occur.

Collision resolution policies

- We need a collision resolution policy to prescribe what to do when collisions occur.
- We discuss two policies for resolving collisions (open addressing and chaining).

Policy 1: Collision resolution via open addressing

- Open addressing uses no extra space - every element is stored in the hash table.

Policy 1: Collision resolution via open addressing

- Open addressing uses no extra space - every element is stored in the hash table.
- If a key k hashes to a value $h(k)$ that is already occupied, we probe (look for an empty space).

Policy 1: Collision resolution via open addressing

- Open addressing uses no extra space - every element is stored in the hash table.
- If a key k hashes to a value $h(k)$ that is already occupied, we probe (look for an empty space).
  - The most common probing method is linear probing, which moves left one index at a time, wrapping around if necessary, until it finds an empty address.

Collision resolution via open addressing (from our textbook DGW2016)

Table 3.3: Open addressing with linear probing (OALP).

Data [key,value]	**Hash**: key/10	**Table address**	**Comments**
[20,A]	2	2	
[15,B]	1	1	
[45,C]	4	4	
[87,D]	8	8	
[39,E]	3	3	
[31,F]	3	0	try 3, 2, 1, 0
[24,G]	2	9	try 2, 1, 0, 9

Policy 1: Collision resolution via open addressing

- Open addressing uses no extra space - every element is stored in the hash table. If it gets overfull, we can reallocate space and rehash.
- If a key k hashes to a value $h(k)$ that is already occupied, we probe (look for an empty space).
  - The most common probing method is linear probing, which moves left one index at a time, wrapping around if necessary, until it finds an empty address.
  - Another method is double hashing. Use a second hash $\Delta(k) = t$ to find a place t and if it is occupied again move to the left by a fixed step size t, wrapping around if necessary, until we find an empty address.

Collision resolution via double hashing (from our textbook DGW2016)

if the hash function is given by $\Delta(k) = (h(k) + k) \bmod 10$.

Table 3.4: Open addressing with double hashing (OADH).

Data [key,value]	**Hash**: key/10	**Table address**	**Comments**
[20,A]	2	2	
[15,B]	1	1	
[45,C]	4	4	
[87,D]	8	8	
[39,E]	3	3	
[31,F]	3	9	using $\Delta(31) = 4$
[24,G]	2	6	using $\Delta(24) = 6$

Policy 2: Collision resolution via chaining

- Chaining uses an "overflow" list for each element in the hash table.

Policy 2: Collision resolution via chaining

- Chaining uses an "overflow" list for each element in the hash table.
- Elements that hash to the same slot are placed in a list.

Policy 2: Collision resolution via chaining

- Chaining uses an "overflow" list for each element in the hash table.
- Elements that hash to the same slot are placed in a list.
- A drawback is the additional space overhead. Also, the distribution of sizes of lists turns out to be very uneven.

Warm call

- When hashing n keys into a table with m slots, how often do you think collisions occur when n is much smaller than m ?

Analysis of hashing

- We count the cost (running time) by number of key comparisons.

Analysis of hashing

- We count the cost (running time) by number of key comparisons.
- We often use the simple uniform hashing model. That is, each of the n keys is equally likely to hash into any of the m slots. So we can consider a "balls in bins" model.

Analysis of hashing

- We count the cost (running time) by number of key comparisons.
- We often use the simple uniform hashing model. That is, each of the n keys is equally likely to hash into any of the m slots. So we can consider a "balls in bins" model.
- If n is much smaller than m, collisions will be few and most slots will be empty. If n is much larger than m, collisions will be many and no slots will be empty. The most interesting behaviour is when m and n are of comparable size.

Analysis of hashing

- We count the cost (running time) by number of key comparisons.
- We often use the simple uniform hashing model. That is, each of the n keys is equally likely to hash into any of the m slots. So we can consider a "balls in bins" model.
- If n is much smaller than m, collisions will be few and most slots will be empty. If n is much larger than m, collisions will be many and no slots will be empty. The most interesting behaviour is when m and n are of comparable size.
- Define the load factor to be $\lambda := n/m$.

How often do collisions occur?

- The probability of no collisions when n balls are thrown into m bins uniformly at random is $Q(m, n)$.

How often do collisions occur?

- The probability of no collisions when n balls are thrown into m bins uniformly at random is $Q(m, n)$.
- Note that when the load factor is very small $\lambda \to 0$, collisions are unlikely (for example $Q(m, 0) = 1$ and $Q(m, 1) = 1$).

How often do collisions occur?

- The probability of no collisions when n balls are thrown into m bins uniformly at random is $Q(m, n)$.
- Note that when the load factor is very small $\lambda \to 0$, collisions are unlikely (for example $Q(m, 0) = 1$ and $Q(m, 1) = 1$).
- At the other extreme case of $\lambda > 1$, collisions are absolutely certain (i.e. $Q(m, n) = 0$) according to the *pigeonhole principle*.

How often do collisions occur?

- The probability of no collisions when n balls are thrown into m bins uniformly at random is $Q(m, n)$.
- Note that when the load factor is very small $\lambda \to 0$, collisions are unlikely (for example $Q(m, 0) = 1$ and $Q(m, 1) = 1$).
- At the other extreme case of $\lambda > 1$, collisions are absolutely certain (i.e. $Q(m, n) = 0$) according to the *pigeonhole principle*.
- Birthday Paradox: If there are 23 or more people in a room, the chance is greater than 50% that two or more of them have the same birthday.

How often do collisions occur?

- The probability of no collisions when n balls are thrown into m bins uniformly at random is $Q(m, n)$.
- Note that when the load factor is very small $\lambda \to 0$, collisions are unlikely (for example $Q(m, 0) = 1$ and $Q(m, 1) = 1$).
- At the other extreme case of $\lambda > 1$, collisions are absolutely certain (i.e. $Q(m, n) = 0$) according to the *pigeonhole principle*.
- Birthday Paradox: If there are 23 or more people in a room, the chance is greater than 50% that two or more of them have the same birthday.
- When a table with 365 slots is only $23/365 = 6.3\%$ full, the next attempt to insert an element is slightly more likely to result in a collision than not.

Analysis of balls in bins

- The probability of no collisions when n balls are thrown into m boxes uniformly at random is $Q(m, n)$.

Analysis of balls in bins

- The probability of no collisions when n balls are thrown into m boxes uniformly at random is $Q(m, n)$.
- The probability of no collisions is

$$Q(m,n) = \frac{m}{m}\frac{m-1}{m}\ldots\frac{m-n+1}{m} = \frac{m!}{(m-n)!m^n}.$$

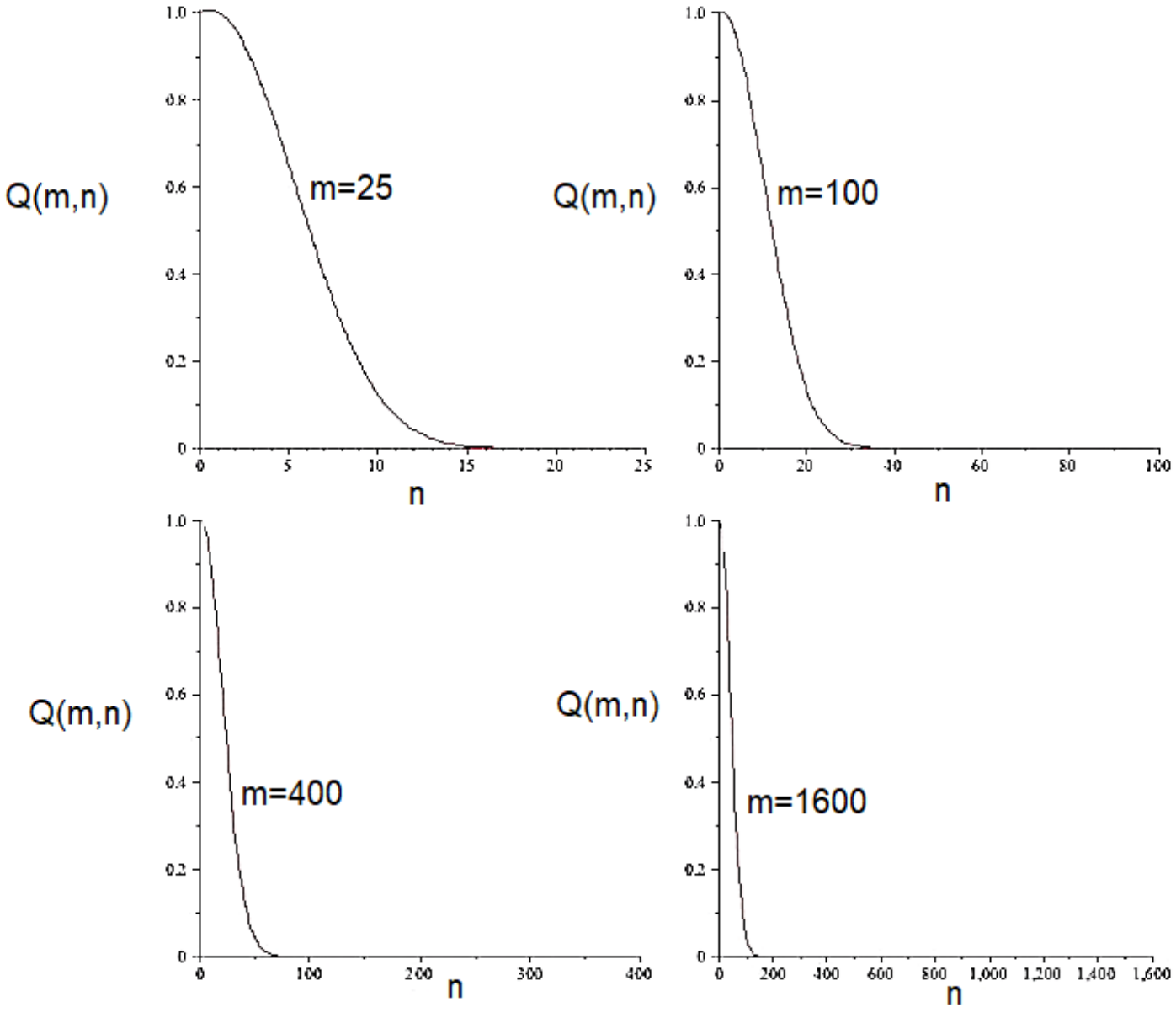
Analysis of balls in bins

- The probability of no collisions when n balls are thrown into m boxes uniformly at random is $Q(m, n)$.
- The probability of no collisions is

$$Q(m,n) = \frac{m}{m}\frac{m-1}{m}\ldots\frac{m-n+1}{m} = \frac{m!}{(m-n)!m^n}.$$

- $Q(m, n) = 0$ unless $0 \leq n \leq m$.

Plots of $Q(m,n)$ against n, $m = 25, 100, 400, 1600$



Plots of $Q(m, n)$ against n, $m = 25, 100, 400, 1600$

- Actually, the point where collision become almost certain scales with $\sqrt{m}$. This means that for example, in a table with 1600 slots, we almost certainly get a collision after filling 128 elements (table being 92% empty).

Running time for chaining (under simple uniform hashing)

- The expected length of a chain is $\lambda = n/m$ under the "balls in bins" model because each ball goes to a given slot with the probability $1/m$ and there are n balls.

Running time for chaining (under simple uniform hashing)

- The expected length of a chain is $\lambda = n/m$ under the "balls in bins" model because each ball goes to a given slot with the probability $1/m$ and there are n balls.
- The average cost for unsuccessful search is the average list length, namely λ because we exhaust the list.

Running time for chaining (under simple uniform hashing)

- The expected length of a chain is $\lambda = n/m$ under the "balls in bins" model because each ball goes to a given slot with the probability $1/m$ and there are n balls.
- The average cost for unsuccessful search is the average list length, namely λ because we exhaust the list.
- The average cost for successful search is roughly $\lambda/2$ because on average we will find it half-way down the list.

Running time for chaining (under simple uniform hashing)

- The expected length of a chain is $\lambda = n/m$ under the "balls in bins" model because each ball goes to a given slot with the probability $1/m$ and there are n balls.
- The average cost for unsuccessful search is the average list length, namely λ because we exhaust the list.
- The average cost for successful search is roughly $\lambda/2$ because on average we will find it half-way down the list.
- Thus provided the load factor is kept bounded, find, delete, and insert run in constant time, $O(\lambda)$ on average.

Efficiency of various implementations of Dictionary ADT

Table: Average case running time (asymptotic order)

Data structure	Insert	Delete	Find
Unsorted list (array)	1	n	n
Sorted list (array)	n	n	$\log n$
Binary search tree	$\log n$	$\log n$	$\log n$
Hash table (chaining)	λ	λ	λ

Efficiency of various implementations of Dictionary ADT

Table: Worst case running time (asymptotic order)

Data structure	Insert	Delete	Find
Unsorted list (array)	1	n	n
Sorted list (array)	n	n	$\log n$
Binary search tree	n	n	n
Hash table (chaining)	n	n	n

Questions

- What hashing methods are used by major programming languages?

Hashing in practice (Dec 2018)

- Java Collections Framework uses chaining to implement `HashMap`, resizing when $\lambda > 0.75$, and table size a power of 2.

Hashing in practice (Dec 2018)

- Java Collections Framework uses chaining to implement `HashMap`, resizing when $\lambda > 0.75$, and table size a power of 2.
- C++ uses chaining to implement `unordered_map`, resizing when $\lambda > 1$, and prime table size.

Hashing in practice (Dec 2018)

- Java Collections Framework uses chaining to implement `HashMap`, resizing when $\lambda > 0.75$, and table size a power of 2.
- C++ uses chaining to implement `unordered_map`, resizing when $\lambda > 1$, and prime table size.
- C# uses chaining, resizing when $\lambda > 1$, and prime table size.

Hashing in practice (Dec 2018)

- Java Collections Framework uses chaining to implement `HashMap`, resizing when $\lambda > 0.75$, and table size a power of 2.
- C++ uses chaining to implement `unordered_map`, resizing when $\lambda > 1$, and prime table size.
- C# uses chaining, resizing when $\lambda > 1$, and prime table size.
- Python uses open addressing, resizing when $\lambda > 0.66$, and table size a power of 2.

Some hash functions

- **Division method**: $h(k) = k \mod m$
  - Especially bad if m has some common factors with k.
- **Multiplication method**: $h(k) = ak \mod 2^w >> (w - r)$
  - Does not work well for some situations.
- **Universal hashing**: $h_{a,b}(k) = [(ak + b) \mod p] \mod m$
  - Requires selecting p a prime, $p > |U|$ and selecting a, b, $0 \leq a, b \leq p - 1$.
  - For worst-case keys, probability of collision is bounded by $2/m$.

Next time

- Reading assignment 2 (due in 5 days as per course schedule)
  - Pages 257-264 (page numbers from the pdf file) Section 8.1 in Chapter 8 of "Mathematics for Machine Learning" by Marc P. Deisenroth et al., 2020
    `https://mml-book.github.io/book/mml-book.pdf`
  - Complete a quiz on Quercus and submit it by the deadline
- Week 2 Lab on Thursday and Friday: Tutorial 1 - Basic Data Science
- Week 3 Lecture (next Tuesday and Wednesday) – Foundations of Learning
- Project 1 Due - Feb. 4 at 23:00