COSC1285/2123: Algorithms & Analysis

Algorithmic Analysis

Jeffrey Chan

RMIT University
Email : jeffrey.chan@rmit.edu.au

Lecture 2

Outline

Overview

Overview

Levitin – The design and analysis of algorithms

This week we will be covering the material from Chapter 2.

Learning outcomes:

- Understand why it is important to be able to compare the complexiy of algorithms.
- Be able to:
  - measure complexity of algorithms and compare complexity classes.
  - analysis of non-recursive algorithms.
  - analysis of recursive algorithms.
- Be able to perform empirical analysis of algorithms.

Example: Which is faster?

Scenario: You work for a small medicine stocker. Your boss comes and asks your opinion on which of two approaches is faster for employees to search over an unordered set of medicines (strings).

Example: Which is faster?

Scenario: You work for a small medicine stocker. Your boss comes and asks your opinion on which of two approaches is faster for employees to search over an unordered set of medicines (strings).

Question: Which of the following two approaches will you tell your boss is faster?

Example: Which is faster?

Scenario: You work for a small medicine stocker. Your boss comes and asks your opinion on which of two approaches is faster for employees to search over an unordered set of medicines (strings).

Question: Which of the following two approaches will you tell your boss is faster?

- Solution A: Sequential search.

Example: Which is faster?

Scenario: You work for a small medicine stocker. Your boss comes and asks your opinion on which of two approaches is faster for employees to search over an unordered set of medicines (strings).

Question: Which of the following two approaches will you tell your boss is faster?

- Solution A: Sequential search.
- Solution B: Sort then search.

https://goo.gl/forms/34JJe2n9bAlX9eBP2

Which is faster?

- In this lecture, we look at the ways of estimating the running time of a program and how to compare the running times of two programs **without ever implementing them.**

Which is faster?

- In this lecture, we look at the ways of estimating the running time of a program and how to compare the running times of two programs **without ever implementing them.**
- It is vital to analyse the resource use of an algorithm, well before it is implemented and deployed.

Which is faster?

- In this lecture, we look at the ways of estimating the running time of a program and how to compare the running times of two programs **without ever implementing them.**
- It is vital to analyse the resource use of an algorithm, well before it is implemented and deployed.

Space is also important but we focus on time in this course.

Overview

Theoretical Analysis of Time Efficiency

How to estimate the running time of an algorithm?

Theoretical Analysis of Time Efficiency

How to estimate the running time of an algorithm?

Idea: An algorithm consists of some operations executed a number of times.

Hence, an estimate of the running time/time efficiency of an algorithm can be obtained from determining these operations, how long to execute them, and the number of times they are executed.

Theoretical Analysis of Time Efficiency

How to estimate the running time of an algorithm?

Idea: An algorithm consists of some operations executed a number of times.

Hence, an estimate of the running time/time efficiency of an algorithm can be obtained from determining these operations, how long to execute them, and the number of times they are executed.

These operations are called **basic operations** and the number of times is based on the **input size** of the problem.

Running example: a^n

This algorithm computes a^n:

```
// INPUT : a, n
// OUTPUT : s = a^n
set s = 1
for i = 1 to n do
    s = s * a
end for
return s
```

Basic Operation

What is a **basic operation**?

Basic Operation

What is a **basic operation**?

- Operation(s) that contribute most towards the total running time.

Basic Operation

What is a **basic operation**?

- Operation(s) that contribute most towards the total running time.
- Examples: compare, add, multiply, divide or assignment.

Basic Operation

What is a **basic operation**?

- Operation(s) that contribute most towards the total running time.
- Examples: compare, add, multiply, divide or assignment.
- Typically the operation most frequently executed, although dependent on the time of each operation.

Basic Operation

What is a **basic operation**?

- Operation(s) that contribute most towards the total running time.
- Examples: compare, add, multiply, divide or assignment.
- Typically the operation most frequently executed, although dependent on the time of each operation.

What is the basic operation of our example algorithm?

```
// INPUT : a, n
// OUTPUT : s = a^n
set s = 1
for i = 1 to n do
    s = s * a
end for
return s
```

Input Size

What is the **input size**?

- More of a characteristic of the problem, “Size of the problem”.

Input Size

What is the **input size**?

- More of a characteristic of the problem, “Size of the problem”.
- E.g., searching through an array of size n, the input size is n.

Input Size

What is the **input size**?

- More of a characteristic of the problem, “Size of the problem”.
- E.g., searching through an array of size n, the input size is n.
- We want to estimate the running time of an algorithm in terms of the problem, not in absolute terms.

Input Size

What is the **input size**?

- More of a characteristic of the problem, "Size of the problem".
- E.g., searching through an array of size n, the input size is n.
- We want to estimate the running time of an algorithm in terms of the problem, not in absolute terms.
- When analysing algorithms, we state the number of times that the basic operation is executed in terms of the input size.

Input Size

What is the **input size**?

- More of a characteristic of the problem, “Size of the problem”.
- E.g., searching through an array of size n, the input size is n.
- We want to estimate the running time of an algorithm in terms of the problem, not in absolute terms.
- When analysing algorithms, we state the number of times that the basic operation is executed in terms of the input size.

What is the input size of our example algorithm?

```
// INPUT : a, n
// OUTPUT : s = a^n
set s = 1
for i = 1 to n do
    s = s * a
end for
return s
```

More examples of basic operation and input size

Problem Type	Basic Operation	Input Size
Iterating through an array, of size n, to print its contents	?	n
Comparing between all pairs of points to find closest pair (n number of points).	Comparison	?

Theoretical Analysis of Time Efficiency

With the basic operation and input size, how do we estimate the running time of an algorithm?

Theoretical Analysis of Time Efficiency

With the basic operation and input size, how do we estimate the running time of an algorithm?

Running time approximately equal to time to execute a basic operation $\times$ number of basic operations

Theoretical Analysis of Time Efficiency

With the basic operation and input size, how do we estimate the running time of an algorithm?

Running time approximately equal to time to execute a basic operation $\times$ number of basic operations

$$t(n) \approx c_{op} \times C(n)$$

- $t(n)$ is the running time.
- n is the input size.
- c_{op} is the execution time for a basic operation.
- $C(n)$ is the number of times the basic operation is executed.

Estimating algorithm for a^n

What is the theoretical running time for our algorithm for computing a^n?

Recall: $t(n) \approx c_{op} \times C(n)$

// INPUT : a, n
// OUTPUT : $s = a^n$

1: set $s = 1$
2: **for** $i = 1$ to n **do**
3: $\quad s = s * a$
4: **end for**
5: **return** s

Estimating algorithm for a^n

What is the theoretical running time for our algorithm for computing a^n?

Recall: $t(n) \approx c_{op} \times C(n)$

```
// INPUT : a, n
// OUTPUT : s = a^n
set s = 1
for i = 1 to n do
    s = s * a
end for
return s
```

Answer: $C(n) = n$ (number of times basic op. executed)

Estimating algorithm for a^n

What is the theoretical running time for our algorithm for computing a^n?

Recall: $t(n) \approx c_{op} \times C(n)$

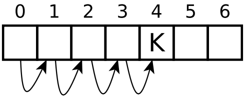
// INPUT : a, n
// OUTPUT : $s = a^n$

1: set $s = 1$
2: **for** $i = 1$ to n **do**
3: $\quad s = s * a$
4: **end for**
5: **return** s

Answer: $C(n) = n$ (number of times basic op. executed)

$t(n) \approx c_{op} \times C(n) = c_{op} \times n$

Example: Searching for a key in n items using Sequential Search



ALGORITHM **SequentialSearch** $(A[0 \ldots n-1], K)$

// INPUT : An array A of length n and a search key K.

// OUTPUT : The index of the first element of A which matches K or n (length of A) otherwise.

```
set i = 0
while i < n and A[i] ≠ K do
    set i = i + 1
end while
return i
```

Example: Searching for a key in n items using Sequential Search

ALGORITHM **SequentialSearch** $(A[0 \ldots n-1], K)$
// INPUT : An array A of length n and a search key K.
// OUTPUT : The index of the first element of A which matches K or n (length of A) otherwise.

```
set i = 0
while i < n and A[i] ≠ K do
    set i = i + 1
end while
return i
```

- Basic Operation?

Example: Searching for a key in n items using Sequential Search

ALGORITHM **SequentialSearch** $(A[0 \ldots n-1], K)$
// INPUT : An array A of length n and a search key K.
// OUTPUT : The index of the first element of A which matches K or n (length of A) otherwise.

1: set $i = 0$
2: **while** $i < n$ **and** $A[i] \neq K$ **do**
3:     set $i = i + 1$
4: **end while**
5: **return** i

- Basic Operation? comparison, addition, assignment (any of these are okay)

Example: Searching for a key in n items using Sequential Search

ALGORITHM **SequentialSearch** $(A[0 \ldots n-1], K)$
// INPUT : An array A of length n and a search key K.
// OUTPUT : The index of the first element of A which matches K or n (length of A) otherwise.

```
set i = 0
while i < n and A[i] ≠ K do
    set i = i + 1
end while
return i
```

- Basic Operation? comparison, addition, assignment (any of these are okay)
- Input size?

Example: Searching for a key in n items using Sequential Search

ALGORITHM **SequentialSearch** $(A[0 \ldots n-1], K)$
// INPUT : An array A of length n and a search key K.
// OUTPUT : The index of the first element of A which matches K or n (length of A) otherwise.

```
set i = 0
while i < n and A[i] ≠ K do
    set i = i + 1
end while
return i
```

- Basic Operation? comparison, addition, assignment (any of these are okay)
- Input size? n

Example: Searching for a key in n items using Sequential Search

What is the theoretical running time for Sequential Search?

Recall: $t(n) \approx c_{op} \times C(n)$

ALGORITHM **SequentialSearch** $(A[0 \ldots n-1], K)$
// INPUT : An array A of length n and a search key K.
// OUTPUT : The index of the first element of A which matches K or n (length of A) otherwise.

1: set $i = 0$
2: **while** $i < n$ **and** $A[i] \neq K$ **do**
3:     set $i = i + 1$
4: **end while**
5: **return** i

What is $C(n)$, the number of times the basic operation is executed?

Runtime Complexity

- **Worst Case** - Given an input of *n* items, what is the maximum running time for any possible input?

Runtime Complexity

- **Worst Case** - Given an input of n items, what is the maximum running time for any possible input?
- **Best Case** - Given an input of n items, what is the minimum running time for any possible input?

Runtime Complexity

- **Worst Case** - Given an input of *n* items, what is the maximum running time for any possible input?
- **Best Case** - Given an input of *n* items, what is the minimum running time for any possible input?
- **Average Case** - Given an input of *n* items, what is the average running time across all possible inputs?

Runtime Complexity

- **Worst Case** - Given an input of n items, what is the maximum running time for any possible input?
- **Best Case** - Given an input of n items, what is the minimum running time for any possible input?
- **Average Case** - Given an input of n items, what is the average running time across all possible inputs?

NOTE : **Average Case** is *not* the average of the worst and best case. Rather, it is the average performance across all possible inputs.

Sequential Search

ALGORITHM **SequentialSearch** $(A[0 \ldots n-1], K)$

```
set i = 0
while i < n and A[i] ≠ K do
    set i = i + 1
end while
return i
```

Best-case : The best case input is when the item being searched for is the first item in the list, so $C_b(n) = 1$.

Worst-case : The worst case input is when the item being searched for is not present in the list, so $C_w(n) = n$

Sequential Search

Average-case : What does average-case mean?

- Recall: average across all possible inputs – how to analyse this?
- Typically not straight forward.

Sequential Search

Average-case : What does average-case mean?

- Recall: average across all possible inputs – how to analyse this?
- Typically not straight forward.

Sequential Search Example:

- How often do we search for items in the array?
    - Where in the array do we find the item? 1st, 2nd, . . ., last position?
- How often do we search for items **not** in the array?

Sequential Search

Average-case Analysis : Skipping a few steps (notes of full derivation available on Canvas) and p is the probability of a successful search.

$$C_{avg}(n) = \frac{p(n+1)}{2} + n(1-p)$$

If $p = 1$, then $C_{avg}(n) = (n+1)/2$.

If $p = 0$, then $C_{avg}(n) = n$.

Summary for this part

- Input size, basic operation
- Time complexity estimate using input size and basic operation
- Best, worst, and average cases

Overview

Asymptotic Complexity

Problem:

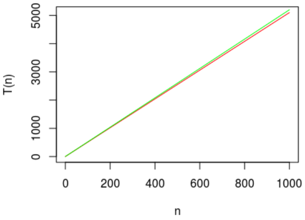
- We now have a way to analyse the running time (aka time complexity) of an algorithm, but every algorithms have their own time complexities. How to compare in a meaningful way?

Asymptotic Complexity

Consider the running times estimates of two algorithms:

$$\text{Algorithm 1: } T_1(n) = 5.1n$$
$$\text{Algorithm 2: } T_2(n) = 5.2n$$

Similar timing profiles as n grows?

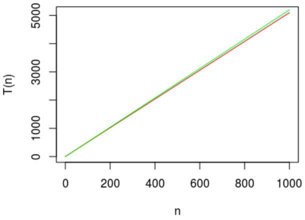


Asymptotic Complexity

Consider the running times estimates of two algorithms:

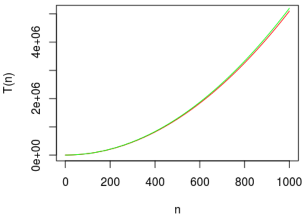
Algorithm 1: $T_1(n) = 5.1n$
Algorithm 2: $T_2(n) = 5.2n$

Similar timing profiles as n grows?



What about the following?:

Algorithm 3: $T_3(n) = 5.1n^2$
Algorithm 4: $T_4(n) = 5.2n^2$



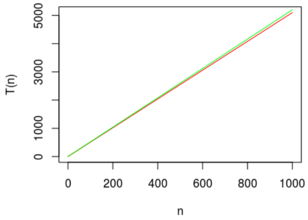
Asymptotic Complexity

Consider the running times estimates of two algorithms:

Algorithm 1: $T_1(n) = 5.1n$

Algorithm 2: $T_2(n) = 5.2n$

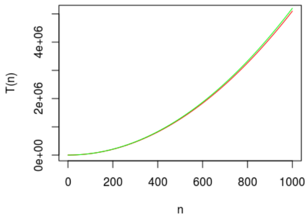
Similar timing profiles as n grows?



What about the following?:

Algorithm 3: $T_3(n) = 5.1n^2$

Algorithm 4: $T_4(n) = 5.2n^2$



Asymptotic Complexity

Problem:

- We now have a way to analyse the running time (aka time complexity) of an algorithm, but every algorithms have their own time complexities. How to compare in a meaningful way?

Solution:

- Group them into equivalence classes (for easier comparison and understanding), with respect to the input size.
- Focus of this part, asymptotic complexity and equivalence classes.

Asymptotic Complexity - bounds

Warning: Some (mathematical) definitions coming up!

Warning: Some (mathematical) definitions coming up!

Idea: Use bounds and asymptotic complexity (as n becomes large, what is the dominant term contributing to the running time ($t(n)$)?)

Asymptotic Complexity, Upper bounds

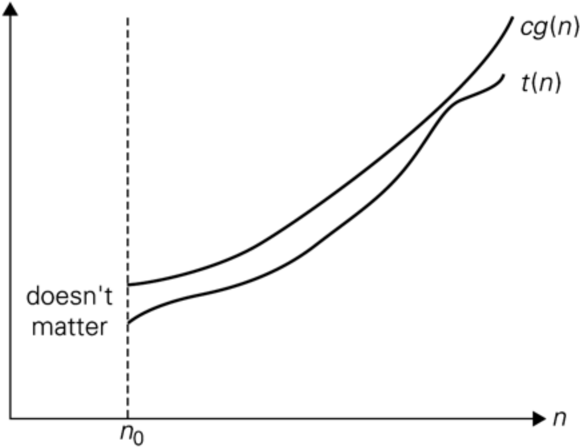
Definition: Given a function $t(n)$ (the running time of an algorithm):

- Let $c \times g(n)$ be a function that is an **upper** bound on $t(n)$ for some $c > 0$ and for *"large"* n.

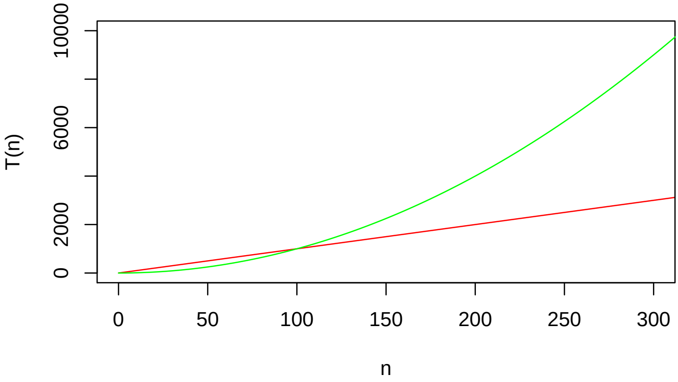
Asymptotic Complexity, Upper bounds

Definition: Given a function $t(n)$ (the running time of an algorithm):

- Let $c \times g(n)$ be a function that is an **upper** bound on $t(n)$ for some $c > 0$ and for *"large"* n.



Upper bounds Examples



Asymptotic Complexity, $\mathcal{O}(n)$

Rather than talking about individual upper bounds for each running time function, we seek to group them into *equivalence classes* that describe their **order of growth.**

Asymptotic Complexity, $\mathcal{O}(n)$

Rather than talking about individual upper bounds for each running time function, we seek to group them into *equivalence classes* that describe their **order of growth.**

Big-O notation, $\mathcal{O}(n)$: Given a function $t(n)$,

- Informally: $t(n) \in \mathcal{O}(g(n))$ means $g(n)$ is a function that, as n increases, provides an upper bound for $t(n)$.
- Formally: $t(n) \in \mathcal{O}(g(n))$, if $g(n)$ is a function and $c \times g(n)$ is an **upper** bound on $t(n)$ for some $c > 0$ and for *"large"* n.

Asymptotic Complexity, $\mathcal{O}(n)$

Rather than talking about individual upper bounds for each running time function, we seek to group them into *equivalence classes* that describe their **order of growth.**

Big-O notation, $\mathcal{O}(n)$: Given a function $t(n)$,

- Informally: $t(n) \in \mathcal{O}(g(n))$ means $g(n)$ is a function that, as n increases, provides an upper bound for $t(n)$.
- Formally: $t(n) \in \mathcal{O}(g(n))$, if $g(n)$ is a function and $c \times g(n)$ is an **upper** bound on $t(n)$ for some $c > 0$ and for *"large"* n.

E.g., if $t(n) = 5.1n$,

- $g(n) = n$

Asymptotic Complexity, $\mathcal{O}(n)$

Rather than talking about individual upper bounds for each running time function, we seek to group them into *equivalence classes* that describe their **order of growth.**

Big-O notation, $\mathcal{O}(n)$: Given a function $t(n)$,

- Informally: $t(n) \in \mathcal{O}(g(n))$ means $g(n)$ is a function that, as n increases, provides an upper bound for $t(n)$.
- Formally: $t(n) \in \mathcal{O}(g(n))$, if $g(n)$ is a function and $c \times g(n)$ is an **upper** bound on $t(n)$ for some $c > 0$ and for *"large"* n.

E.g., if $t(n) = 5.1n$,

- $g(n) = n$
- $g(n) = 0.001n - 6$

Asymptotic Complexity, $\mathcal{O}(n)$

Rather than talking about individual upper bounds for each running time function, we seek to group them into *equivalence classes* that describe their **order of growth.**

Big-O notation, $\mathcal{O}(n)$: Given a function $t(n)$,

- Informally: $t(n) \in \mathcal{O}(g(n))$ means $g(n)$ is a function that, as n increases, provides an upper bound for $t(n)$.
- Formally: $t(n) \in \mathcal{O}(g(n))$, if $g(n)$ is a function and $c \times g(n)$ is an **upper** bound on $t(n)$ for some $c > 0$ and for *"large"* n.

E.g., if $t(n) = 5.1n$,

- $g(n) = n$
- $g(n) = 0.001n - 6$
- $g(n) = n^2$

Asymptotic Complexity, $\mathcal{O}(n)$

Rather than talking about individual upper bounds for each running time function, we seek to group them into *equivalence classes* that describe their **order of growth.**

Big-O notation, $\mathcal{O}(n)$: Given a function $t(n)$,

- Informally: $t(n) \in \mathcal{O}(g(n))$ means $g(n)$ is a function that, as n increases, provides an upper bound for $t(n)$.
- Formally: $t(n) \in \mathcal{O}(g(n))$, if $g(n)$ is a function and $c \times g(n)$ is an **upper** bound on $t(n)$ for some $c > 0$ and for *"large"* n.

E.g., if $t(n) = 5.1n$,

- $g(n) = n$
- $g(n) = 0.001n - 6$
- $g(n) = n^2$

Asymptotic Complexity, $\mathcal{O}(n)$

Rather than talking about individual upper bounds for each running time function, we seek to group them into *equivalence classes* that describe their **order of growth.**

Big-O notation, $\mathcal{O}(n)$: Given a function $t(n)$,

- Informally: $t(n) \in \mathcal{O}(g(n))$ means $g(n)$ is a function that, as n increases, provides an upper bound for $t(n)$.
- Formally: $t(n) \in \mathcal{O}(g(n))$, if $g(n)$ is a function and $c \times g(n)$ is an **upper** bound on $t(n)$ for some $c > 0$ and for *"large"* n.

E.g., if $t(n) = 5.1n$,

- $g(n) = n$
- $g(n) = 0.001n - 6$
- $g(n) = n^2$

What $g(n)$ to use if $t(n) = 5.2n$?

Asymptotic Complexity, $\mathcal{O}(n)$

Rather than talking about individual upper bounds for each running time function, we seek to group them into *equivalence classes* that describe their **order of growth**.

Big-O notation, $\mathcal{O}(n)$: Given a function $t(n)$,

- Informally: $t(n) \in \mathcal{O}(g(n))$ means $g(n)$ is a function that, as n increases, provides an upper bound for $t(n)$.
- Formally: $t(n) \in \mathcal{O}(g(n))$, if $g(n)$ is a function and $c \times g(n)$ is an **upper** bound on $t(n)$ for some $c > 0$ and for *"large"* n.

E.g., if $t(n) = 5.1n$,

- $g(n) = n$
- $g(n) = 0.001n - 6$
- $g(n) = n^2$

What $g(n)$ to use if $t(n) = 5.2n$? Any of the above $g(n)$ functions are possible!

Common Equivalence Classes

Previous slide, we had multiple upper bounds. But we can in fact group them to a smaller number.

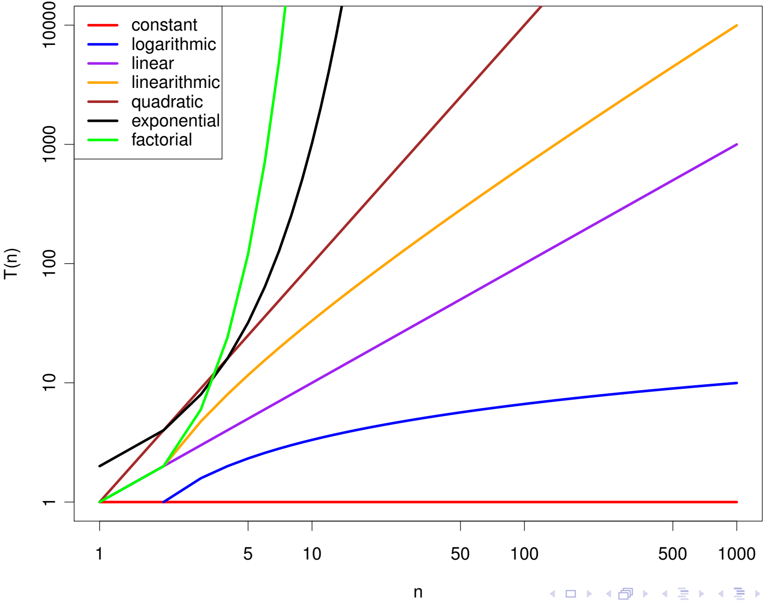
Common Equivalence Classes

Previous slide, we had multiple upper bounds. But we can in fact group them to a smaller number.

First we look at common equivalence classes:

- Constant - $\mathcal{O}(1)$: Access array element
- Logarithmic - $\mathcal{O}(\log n)$: Binary search
- Linear - $\mathcal{O}(n)$: Link list search
- Linearithmic (Supralinear) - $\mathcal{O}(n \log n)$: Merge Sorting
- Quadratic - $\mathcal{O}(n^2)$: Selection Sorting
- Exponential - $\mathcal{O}(2^n)$: Generating all subsets
- Factorial - $\mathcal{O}(n!)$: Generating all permutations

Common Complexity Bounds



Asymptotic Complexity

Recall: We want to find equivalence function class that upper bounds different $t(n)$.

Asymptotic Complexity

Recall: We want to find equivalence function class that upper bounds different $t(n)$.

But we might be given an upper bound $g(n)$ that isn't quite in the form of the equivalence classes. How to get the equivalence classes?

Asymptotic Complexity

Recall: We want to find equivalence function class that upper bounds different $t(n)$.

But we might be given an upper bound $g(n)$ that isn't quite in the form of the equivalence classes. How to get the equivalence classes?

Adhere to the following guidelines:

- Do not include constants (e.g., omit '-6' in $0.01n - 6$ to become $0.01n$).
- Omit all the lower order terms (e.g., $n^3 + n^2 + n$, we simplify to n^3).
- Omit the coefficient of the highest-order term (e.g., $0.01n$ becomes n).

Recall: We want to find equivalence function class that upper bounds different $t(n)$.

But we might be given an upper bound $g(n)$ that isn't quite in the form of the equivalence classes. How to get the equivalence classes?

Adhere to the following guidelines:

- Do not include constants (e.g., omit '-6' in $0.01n - 6$ to become $0.01n$).
- Omit all the lower order terms (e.g., $n^3 + n^2 + n$, we simplify to n^3).
- Omit the coefficient of the highest-order term (e.g., $0.01n$ becomes n).

Returning to previous example, we can write the two upper bounds $\mathcal{O}(0.01n - 6)$ and $\mathcal{O}(n)$ as $\mathcal{O}(n)$.

Example

Let

$$g(n) = 2n^4 + 43n^3 - n + 50$$

be an **upper** bound for the running time $t(n)$ of an algorithm.

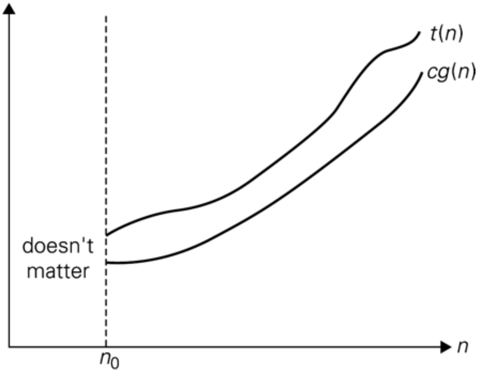
Using the previous guidelines, what equivalence **class** should you use to describe the **order of growth** of the algorithm?

`https://goo.gl/forms/nybF74i02wBFUXWP2`

Asymptotic Complexity, Lower Bound ($\Omega(n)$)

Lower Bound Definition: Given a function $t(n)$,

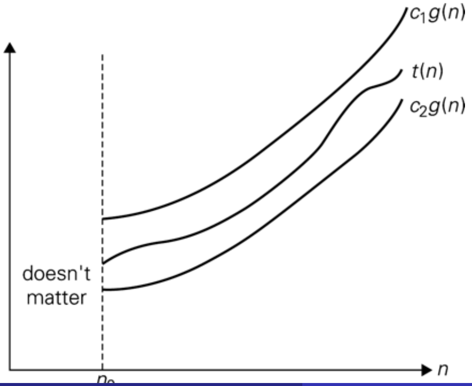
- Informally: $t(n) \in \Omega(g(n))$ means $g(n)$ is a function that, as n increases, is a lower bound of $t(n)$
- Formally: $t(n) \in \Omega(g(n))$, if $g(n)$ is a function and $c \times g(n)$ is a **lower** bound on $t(n)$ for some $c > 0$ and for *"large"* n.



Asymptotic Complexity, Exact Bounds ($\Theta(n)$)

Exact Bound Definition: Given a function $t(n)$,

- Informally: $t(n) \in \Theta(g(n))$ means $g(n)$ is a function that, as n increases, is both a upper and lower bound of $t(n)$
- Formally: $t(n) \in \Theta(g(n))$, if $g(n)$ is a function and $c_1 \times g(n)$ is an **upper** bound on $t(n)$ and $c_2 \times g(n)$ is an **lower** bound on $t(n)$, for some $c_1 > 0$ and $c_2 > 0$ and for *"large" n*



Examples

$t(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n^3)$	$\Omega(n)$	$\Omega(n^2)$	$\Omega(n^3)$	Θ
$\log_2 n$	T	T	T	F	F	F	$\Theta(\log_2 n)$
$10n + 5$	T	T	T	T	F	F	$\Theta(n)$
$n(n-1)/2$	F	T	T	T	T	F	$\Theta(n^2)$
$(n+1)^3$	F	F	T	T	T	T	$\Theta(n^3)$
2^n	F	F	F	T	T	T	$\Theta(2^n)$

For example, $10n + 5$ is in $\mathcal{O}(n)$.

Terminology Clarification

- $\mathcal{O}(n)$ is not the same thing as "Worst Case Efficiency".
- $\Omega(n)$ is not the same thing as "Best Case Efficiency".
- $\Theta(n)$ is not the same thing as "Average Case Efficiency".

It is perfectly reasonable to want the $\Omega(n)$ *and* $\mathcal{O}(n)$ worst case efficiency bounds for a class of algorithms.

Terminology Clarification

- $\mathcal{O}(n)$ is not the same thing as "Worst Case Efficiency".
- $\Omega(n)$ is not the same thing as "Best Case Efficiency".
- $\Theta(n)$ is not the same thing as "Average Case Efficiency".

It is perfectly reasonable to want the $\Omega(n)$ *and* $\mathcal{O}(n)$ worst case efficiency bounds for a class of algorithms.

Why all the bounds?

- Generally $\mathcal{O}(n)$ is most commonly used , but exact bounds tell us the bounds are tight and the algorithm doesn't have anything outside what we expect.
- Lower bounds are useful to describe the (theoretical) limits of whole classes of algorithms, and also sometimes useful to state how fast can the best case reach.

Overview

Time Efficiency of Algorithms

Typically, we are given the pseudo code of an algorithm, not a nice $t(n)$ function.

Question

So how do we determine **bounds** on the **order of growth** of an algorithm?

- Next two parts, we focus on answering this question.

Time Efficiency of Algorithms

Typically, we are given the pseudo code of an algorithm, not a nice $t(n)$ function.

Question

So how do we determine **bounds** on the **order of growth** of an algorithm?

- Next two parts, we focus on answering this question.
- We first study how to do this for non-recursive algorithms, then recursive ones.

Time Efficiency of Algorithms

Typically, we are given the pseudo code of an algorithm, not a nice $t(n)$ function.

Question

So how do we determine **bounds** on the **order of growth** of an algorithm?

- Next two parts, we focus on answering this question.
- We first study how to do this for non-recursive algorithms, then recursive ones.
- Keep in mind: Generally, we first need to estimate the running time $t(n)$, then determine a bound and order of growth.

Time Efficiency of Non-Recursive Algorithms

Question

So how do we determine **bounds** on the **order of growth** of an algorithm?

Recall $t(n) = c_{op}C(n)$.

Time Efficiency of Non-Recursive Algorithms

Question

So how do we determine **bounds** on the **order of growth** of an algorithm?

Recall $t(n) = c_{op}C(n)$.

1. Determine what is used to measure the input size.
2. Identify the algorithm's basic operation

Time Efficiency of Non-Recursive Algorithms

Question

So how do we determine **bounds** on the **order of growth** of an algorithm?

Recall $t(n) = c_{op}C(n)$.

1. Determine what is used to measure the input size.
2. Identify the algorithm's basic operation
3. Determine the number of basic operation executions in terms of the input size? (i.e., How to determine $C(n)$?)
   - Setup a summation for $C(n)$ reflecting the algorithm's loop structure.
   - Simplify the summation by using standard formulas in Appendix A of textbook.

Time Efficiency of Non-Recursive Algorithms

Question

So how do we determine **bounds** on the **order of growth** of an algorithm?

Recall $t(n) = c_{op}C(n)$.

1. Determine what is used to measure the input size.
2. Identify the algorithm's basic operation
3. Determine the number of basic operation executions in terms of the input size? (i.e., How to determine $C(n)$?)
   - Setup a summation for $C(n)$ reflecting the algorithm's loop structure.
   - Simplify the summation by using standard formulas in Appendix A of textbook.
4. Determine a equivalence class $g(n)$ that bounds $t(n)$. Recall we generally want the tightest bound possible.

Non Recusive Example: a^n

// INPUT : a, n
// OUTPUT : $s = a^n$

set $s = 1$
for $i = 1$ to n **do**
    $s = s * a$
end for
return

Non Recusive Example: a^n

// INPUT : a, n
// OUTPUT : $s = a^n$

set $s = 1$
for $i = 1$ to n **do**
    $s = s * a$
end for
return

Input size: n
Basic operation: multiplication
$C(n)$**:** ?

Non Recusive Example: a^n

// INPUT : a, n
// OUTPUT : $s = a^n$

1: set $s = 1$
2: **for** $i = 1$ to n **do**
3: $s = s * a$
4: **end for**
5: **return**

Input size: n
Basic operation: multiplication
$C(n)$**:** ? $\underbrace{1 + 1 + 1 + \ldots + 1}_{n}$

Non Recusive Example: a^n

```
// INPUT : a, n
// OUTPUT : s = a^n
set s = 1
for i = 1 to n do
    s = s * a
end for
return
```

Input size: n
Basic operation: multiplication
$C(n)$**:** ? $\underbrace{1 + 1 + 1 + \ldots + 1}_{n} = \sum_{i=1}^{n} 1$

Given two (square) matrices A and B, both of dimensions n by n, the following algorithm computes $C = A + B$.

```
for (int i = 0; i <= n-1; i++) {
  for (int j = 0; j <= n-1; j++) {
    C[i,j] = A[i,j] + B[i,j];
  }
}
```

Non-Recursive Examples (continued): Adding two matrices

Given two (square) matrices A and B, both of dimensions n by n, the following algorithm computes $C = A + B$.

```
for (int i = 0; i <= n-1; i++) {
  for (int j = 0; j <= n-1; j++) {
    C[i,j] = A[i,j] + B[i,j];
  }
}
```

Input size: ?

Given two (square) matrices A and B, both of dimensions n by n, the following algorithm computes $C = A + B$.

```
for (int i = 0; i <= n-1; i++) {
   for (int j = 0; j <= n-1; j++) {
      C[i,j] = A[i,j] + B[i,j];
   }
}
```

Input size: ? n
Basic operation: ?

Given two (square) matrices A and B, both of dimensions n by n, the following algorithm computes $C = A + B$.

```
for (int i = 0; i <= n-1; i++) {
  for (int j = 0; j <= n-1; j++) {
    C[i,j] = A[i,j] + B[i,j];
  }
}
```

Input size: ? n
Basic operation: ? addition
$C(n)$**:** ?

Given two (square) matrices A and B, both of dimensions n by n, the following algorithm computes $C = A + B$.

```
for (int i = 0; i <= n-1; i++) {
   for (int j = 0; j <= n-1; j++) {
      C[i,j] = A[i,j] + B[i,j];
   }
}
```

Input size: ? n

Basic operation: ? addition

$C(n)$**:** ?

$$\sum_{i=0}^{n-1} \sum_{j=0}^{n-1} 1$$

Time Efficiency of Non-Recursive Algorithms

Question

So how do we determine **bounds** on the **order of growth** of an algorithm?

Recall $t(n) = c_{op}C(n)$.

1. Determine what is used to measure the input size.
2. Identify the algorithm's basic operation.
3. Determine the number of basic operation executions in terms of the input size? (i.e., How to determine $C(n)$?)
   - Setup a summation for $C(n)$ reflecting the algorithm's loop structure.
   - **Simplify the summation by using standard formulas in Appendix A of textbook.**
4. Determine a function family $g(n)$ that bounds $t(n)$. Recall we generally want the tightest bound possible.

Useful series (from Appendix A)

- R1 (Sum): $\sum_{i=l}^{n} 1 = n - l + 1$.
- R2 (Geometric): $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$.
- R3 (Distributive): $\sum_{i=1}^{n} c * a_i = c \sum_{i=1}^{n} a_i$.
- R4 (Associative): $\sum_{i=1}^{n} (a_i \pm b_i) = \sum_{i=1}^{n} a_i \pm \sum_{i=1}^{n} b_i$.

Non Recusive Example a^n: Simplification

// INPUT : a, n
// OUTPUT : $s = a^n$

```
set s = 1
for i = 1 : n do
    s = s * a
end for
return
```

$C(n) = \sum_{i=1}^{n} 1$

Non Recusive Example a^n: Simplification

$$C_(n) = \sum_{i=1}^{n} 1 \tag{1}$$

(2)

(3)

Non Recusive Example a^n: Simplification

$$C_(n) = \sum_{i=1}^{n} 1 \tag{1}$$

$$= n - 1 + 1 \text{ (Use R1)} \tag{2}$$

$$\tag{3}$$

Non Recusive Example a^n: Simplification

$$C_(n) = \sum_{i=1}^{n} 1 \tag{1}$$

$$= n - 1 + 1 \text{ (Use R1)} \tag{2}$$

$$= n \tag{3}$$

Non-Recursive Example Adding two matrices: Simplication

Given two (square) matrices A and B, both of dimensions n by n, the following algorithm computes $C = A + B$. **Example:**

```
for (int i = 0; i <= n-1; i++) {
  for (int j = 0; j <= n-1; j++) {
    C[i,j] = A[i,j] + B[i,j];
  }
}
```

$C(n) = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} 1$

Non-Recursive Example Adding two matrices: Simplication

$$C_(n) = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} 1 \tag{1}$$

(2)

(3)

(4)

(5)

(6)

Non-Recursive Example Adding two matrices: Simplication

$$C_(n) = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} 1 \tag{1}$$

$$= \sum_{i=0}^{n-1} (n - 1 + 1) \text{ (Use R1 on inner summation)} \tag{2}$$

(3)

(4)

(5)

(6)

Non-Recursive Example Adding two matrices: Simplication

$$
\begin{align}
C_(n) &= \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} 1 \tag{1}\\
&= \sum_{i=0}^{n-1} (n - 1 + 1) \text{ (Use R1 on inner summation)} \tag{2}\\
&= \sum_{i=0}^{n-1} (n) \text{ (Simplify)} \tag{3}\\
& \tag{4}\\
& \tag{5}\\
& \tag{6}
\end{align}
$$

Non-Recursive Example Adding two matrices: Simplication

$$
\begin{aligned}
C_(n) &= \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} 1 && (1) \\
&= \sum_{i=0}^{n-1} (n - 1 + 1) \text{ (Use R1 on inner summation)} && (2) \\
&= \sum_{i=0}^{n-1} (n) \text{ (Simplify)} && (3) \\
&= n \sum_{i=0}^{n-1} 1 \text{ (Use R3 to take } n \text{ out of summation)} && (4) \\
& && (5) \\
& && (6)
\end{aligned}
$$

Non-Recursive Example Adding two matrices: Simplication

$$
\begin{align}
C_(n) &= \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} 1 \tag{1}\\
&= \sum_{i=0}^{n-1} (n - 1 + 1) \text{ (Use R1 on inner summation)} \tag{2}\\
&= \sum_{i=0}^{n-1} (n) \text{ (Simplify)} \tag{3}\\
&= n \sum_{i=0}^{n-1} 1 \text{ (Use R3 to take } n \text{ out of summation)} \tag{4}\\
&= n * (n - 1 + 1) \text{ (Use R1 on remaining summation)} \tag{5}\\
& \tag{6}
\end{align}
$$

Non-Recursive Example Adding two matrices: Simplication

$$C_(n) = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} 1 \tag{1}$$

$$= \sum_{i=0}^{n-1} (n - 1 + 1) \text{ (Use R1 on inner summation)} \tag{2}$$

$$= \sum_{i=0}^{n-1} (n) \text{ (Simplify)} \tag{3}$$

$$= n \sum_{i=0}^{n-1} 1 \text{ (Use R3 to take } n \text{ out of summation)} \tag{4}$$

$$= n * (n - 1 + 1) \text{ (Use R1 on remaining summation)} \tag{5}$$

$$= n * n \text{ (Simplify)} \tag{6}$$

Overview

Recursion

- Recursion is fundamental tool in computer science.
- A recursive program (or function) is one that calls itself.
- It must have a termination condition defined.

Recursion

- Recursion is fundamental tool in computer science.
- A recursive program (or function) is one that calls itself.
- It must have a termination condition defined.
- Many interesting algorithms are simply expressed with a recursive approach.

Recursive Algorithm Example: Factorial

Factorial: $\mathcal{F}(n) = n * (n-1) * (n-2) * (n-3) * \ldots * 3 * 2 * 1$

ALGORITHM $\mathcal{F}(n)$

if $n = 1$ **then**
    return 1
else
    return $\mathcal{F}(n-1) * n$
end if

Time Efficiency of Recursive Algorithms

1. Determine what is used to measure the input size.
2. Identify the algorithm's basic operation

Time Efficiency of Recursive Algorithms

1. Determine what is used to measure the input size.
2. Identify the algorithm's basic operation
3. Setup a recurrence relation for $C(n)$, including termination condition(s).

Time Efficiency of Recursive Algorithms

1. Determine what is used to measure the input size.
2. Identify the algorithm's basic operation
3. Setup a recurrence relation for $C(n)$, including termination condition(s).
4. Simplify the recurrence relation using methods in Appendix B of textbook. (**Backward substitution**)
5. Determine a equivalence class $g(n)$ that bounds $t(n) = c_{op}C(n)$. Recall we generally want the tightest bound possible.

Recurrence Relations

A **recurrence relation** represents a sequence of terms, that results from a recursion on one or more of the previous terms.

The recurrence relation include the termination condition.

Recurrence Relations

A **recurrence relation** represents a sequence of terms, that results from a recursion on one or more of the previous terms.

The recurrence relation include the termination condition.

For example, the sequence for a factorial of n, $F(n) = n * (n-1) * (n-2) * (n-3) * \ldots * 3 * 2 * 1$, can be represented as the recurrence relation:

Recurrence Relations

A **recurrence relation** represents a sequence of terms, that results from a recursion on one or more of the previous terms.

The recurrence relation include the termination condition.

For example, the sequence for a factorial of n, $F(n) = n * (n-1) * (n-2) * (n-3) * \ldots * 3 * 2 * 1$, can be represented as the recurrence relation:

$$F(n) = F(n-1) * n, \; F(1) = 1$$

Time Efficiency of Recursive Algorithms

1. Determine what is used to measure the input size.
2. Identify the algorithm's basic operation.
3. Setup a recurrence relation for $C(n)$, including termination condition(s).
4. Simplify the recurrence relation using methods in Appendix B of textbook. (Backward substitution)
5. Determine a function family $g(n)$ that bounds $t(n) = c_{op}C(n)$. Recall we generally want the tightest bound possible.

Recursive Algorithm Example: Factorial

```
ALGORITHM F(n)
if n = 1 then
    return 1
else
    return F(n − 1) ∗ n
end if
```

Recursive Algorithm Example: Factorial

ALGORITHM $\mathcal{F}(n)$

```
if n = 1 then
    return 1
else
    return F(n − 1) ∗ n
end if
```

Basic operation: ?

Recursive Algorithm Example: Factorial

ALGORITHM $\mathcal{F}(n)$

```
if n = 1 then
    return 1
else
    return F(n − 1) ∗ n
end if
```

Basic operation: ? multiplication **Input size:** ?

Recursive Algorithm Example: Factorial

ALGORITHM $\mathcal{F}(n)$

```
if n = 1 then
    return 1
else
    return F(n − 1) ∗ n
end if
```

Basic operation: ? multiplication **Input size:** ? n,

Recursive Algorithm Example: Factorial

ALGORITHM $\mathcal{F}(n)$

if $n = 1$ **then**
  return 1
else
  return $\mathcal{F}(n-1) * n$
end if

Basic operation: ? multiplication **Input size:** ? n,

Recurrence relation and conditions: The number of multiplications is $C(n) = C(n-1) + 1$ for $n > 1$, and $C(1) = 0$.

Recursive Algorithm Example: Factorial

ALGORITHM $\mathcal{F}(n)$

```
if n = 1 then
    return 1
else
    return F(n − 1) ∗ n
end if
```

Basic operation: ? multiplication **Input size:** ? n,

Recurrence relation and conditions: The number of multiplications is $C(n) = C(n-1) + 1$ for $n > 1$, and $C(1) = 0$.

- "$+1$" is the number of multiplication operations at each recursive step.

Recursive Algorithm Example: Factorial

ALGORITHM $\mathcal{F}(n)$

```
if n = 1 then
    return 1
else
    return F(n − 1) ∗ n
end if
```

Basic operation: ? multiplication **Input size:** ? n,

Recurrence relation and conditions: The number of multiplications is $C(n) = C(n-1) + 1$ for $n > 1$, and $C(1) = 0$.

- "$+1$" is the number of multiplication operations at each recursive step.
- When $n = 1$, we have our termination/base case, where we stop the recursion. When we reach this base case, the number of multiplcations is 0, hence $C(1) = 0$.

Time Efficiency of Recursive Algorithms

1. Determine what is used to measure the input size.
2. Identify the algorithm's basic operation.
3. Setup a recurrence relation for $C(n)$, including termination condition(s).
4. Simplify the recurrence relation using methods in Appendix B of textbook. (**Backward substitution**)
5. Determine a function family $g(n)$ that bounds $t(n) = c_{op}C(n)$. Recall we generally want the tightest bound possible.

Backward Substitution: Factorial Example

Recurrence: $C(n) = C(n-1) + 1$ for $n > 1$, and $C(1) = 0$

Aim of simplification and backward substitution: Convert $C(n) = C(n-1) + 1$ to $C(n) =$ function(n), e.g., $C(n) = n + 1$.

Backward Substitution: Factorial Example

Recurrence: $C(n) = C(n-1) + 1$ for $n > 1$, and $C(1) = 0$

Aim of simplification and backward substitution: Convert $C(n) = C(n-1) + 1$ to $C(n) = \text{function}(n)$, e.g., $C(n) = n + 1$.

1. Start with the recurrence relation: $C(n) = C(n-1) + \ldots$

Backward Substitution: Factorial Example

Recurrence: $C(n) = C(n-1) + 1$ for $n > 1$, and $C(1) = 0$

Aim of simplification and backward substitution: Convert $C(n) = C(n-1) + 1$ to $C(n) = \text{function}(n)$, e.g., $C(n) = n + 1$.

1. Start with the recurrence relation: $C(n) = C(n-1) + \ldots$
2. Substitute $C(n-1)$ with its RHS ($C(n-1) = C(n-2) + \ldots$) in original equation $C(n) = C(n-2) + \ldots$

Backward Substitution: Factorial Example

Recurrence: $C(n) = C(n-1) + 1$ for $n > 1$, and $C(1) = 0$

Aim of simplification and backward substitution: Convert $C(n) = C(n-1) + 1$ to $C(n) = \text{function}(n)$, e.g., $C(n) = n + 1$.

1. Start with the recurrence relation: $C(n) = C(n-1) + \ldots$
2. Substitute $C(n-1)$ with its RHS ($C(n-1) = C(n-2) + \ldots$) in original equation $C(n) = C(n-2) + \ldots$
3. Substitute $C(n-2)$ with its RHS ($C(n-2) = C(n-3) + \ldots$) in original equation $C(n) = C(n-3) + \ldots$

Backward Substitution: Factorial Example

Recurrence: $C(n) = C(n-1) + 1$ for $n > 1$, and $C(1) = 0$

Aim of simplification and backward substitution: Convert $C(n) = C(n-1) + 1$ to $C(n) = \text{function}(n)$, e.g., $C(n) = n + 1$.

1. Start with the recurrence relation: $C(n) = C(n-1) + \ldots$
2. Substitute $C(n-1)$ with its RHS ($C(n-1) = C(n-2) + \ldots$) in original equation $C(n) = C(n-2) + \ldots$
3. Substitute $C(n-2)$ with its RHS ($C(n-2) = C(n-3) + \ldots$) in original equation $C(n) = C(n-3) + \ldots$
4. Spot the pattern of $C(n)$ and introduce a variable to express this pattern.

Backward Substitution: Factorial Example

Recurrence: $C(n) = C(n-1) + 1$ for $n > 1$, and $C(1) = 0$

Aim of simplification and backward substitution: Convert $C(n) = C(n-1) + 1$ to $C(n) =$ function(n), e.g., $C(n) = n + 1$.

1. Start with the recurrence relation: $C(n) = C(n-1) + \ldots$
2. Substitute $C(n-1)$ with its RHS ($C(n-1) = C(n-2) + \ldots$) in original equation $C(n) = C(n-2) + \ldots$
3. Substitute $C(n-2)$ with its RHS ($C(n-2) = C(n-3) + \ldots$) in original equation $C(n) = C(n-3) + \ldots$
4. Spot the pattern of $C(n)$ and introduce a variable to express this pattern.
5. Determine when the value of this variable that relates $C(n) = C(1) + \ldots$

Backward Substitution: Factorial Example

Recurrence: $C(n) = C(n-1) + 1$ for $n > 1$, and $C(1) = 0$

Aim of simplification and backward substitution: Convert $C(n) = C(n-1) + 1$ to $C(n) = \text{function}(n)$, e.g., $C(n) = n + 1$.

1. Start with the recurrence relation: $C(n) = C(n-1) + \ldots$
2. Substitute $C(n-1)$ with its RHS ($C(n-1) = C(n-2) + \ldots$) in original equation $C(n) = C(n-2) + \ldots$
3. Substitute $C(n-2)$ with its RHS ($C(n-2) = C(n-3) + \ldots$) in original equation $C(n) = C(n-3) + \ldots$
4. Spot the pattern of $C(n)$ and introduce a variable to express this pattern.
5. Determine when the value of this variable that relates $C(n) = C(1) + \ldots$
6. Substitue the value of $C(1)$ and get $C(n)$ in terms of some expression of n.

Backward Substitution: Factorial Example

Recurrence: $C(n) = C(n-1) + 1$ for $n > 1$, and $C(1) = 0$

Backward Substitution: Factorial Example

Recurrence: $C(n) = C(n-1) + 1$ for $n > 1$, and $C(1) = 0$

1. $C(n) = C(n-1) + 1$

Backward Substitution: Factorial Example

Recurrence: $C(n) = C(n-1) + 1$ for $n > 1$, and $C(1) = 0$

1. $C(n) = C(n-1) + 1$
2. Substitute $C(n-1) = C(n-2) + 1$ into original equation

Backward Substitution: Factorial Example

Recurrence: $C(n) = C(n-1) + 1$ for $n > 1$, and $C(1) = 0$

1. $C(n) = C(n-1) + 1$
2. Substitute $C(n-1) = C(n-2) + 1$ into original equation
3. $C(n) = [C(n-2) + 1] + 1 = C(n-2) + 2$

Backward Substitution: Factorial Example

Recurrence: $C(n) = C(n-1) + 1$ for $n > 1$, and $C(1) = 0$

1. $C(n) = C(n-1) + 1$
2. Substitute $C(n-1) = C(n-2) + 1$ into original equation
3. $C(n) = [C(n-2) + 1] + 1 = C(n-2) + 2$
4. Substitute $C(n-2) = C(n-3) + 1$ into original equation

Backward Substitution: Factorial Example

Recurrence: $C(n) = C(n-1) + 1$ for $n > 1$, and $C(1) = 0$

1. $C(n) = C(n-1) + 1$
2. Substitute $C(n-1) = C(n-2) + 1$ into original equation
3. $C(n) = [C(n-2) + 1] + 1 = C(n-2) + 2$
4. Substitute $C(n-2) = C(n-3) + 1$ into original equation
5. $C(n) = [C(n-3) + 1] + 2 = C(n-3) + 3$

Backward Substitution: Factorial Example

Recurrence: $C(n) = C(n-1) + 1$ for $n > 1$, and $C(1) = 0$

1. $C(n) = C(n-1) + 1$
2. Substitute $C(n-1) = C(n-2) + 1$ into original equation
3. $C(n) = [C(n-2) + 1] + 1 = C(n-2) + 2$
4. Substitute $C(n-2) = C(n-3) + 1$ into original equation
5. $C(n) = [C(n-3) + 1] + 2 = C(n-3) + 3$
6. We see the pattern $C(n) = C(n-i) + i$ emerge, where $1 \leq i \leq n$.

Backward Substitution: Factorial Example

Recurrence: $C(n) = C(n-1) + 1$ for $n > 1$, and $C(1) = 0$

1. $C(n) = C(n-1) + 1$
2. Substitute $C(n-1) = C(n-2) + 1$ into original equation
3. $C(n) = [C(n-2) + 1] + 1 = C(n-2) + 2$
4. Substitute $C(n-2) = C(n-3) + 1$ into original equation
5. $C(n) = [C(n-3) + 1] + 2 = C(n-3) + 3$
6. We see the pattern $C(n) = C(n-i) + i$ emerge, where $1 \leq i \leq n$.
7. Now, we know $C(1)$ and want to determine when $C(n-i) = C(1)$, or when $n - i = 1$. This value is $i = n - 1$.

Backward Substitution: Factorial Example

Recurrence: $C(n) = C(n-1) + 1$ for $n > 1$, and $C(1) = 0$

1. $C(n) = C(n-1) + 1$
2. Substitute $C(n-1) = C(n-2) + 1$ into original equation
3. $C(n) = [C(n-2) + 1] + 1 = C(n-2) + 2$
4. Substitute $C(n-2) = C(n-3) + 1$ into original equation
5. $C(n) = [C(n-3) + 1] + 2 = C(n-3) + 3$
6. We see the pattern $C(n) = C(n-i) + i$ emerge, where $1 \leq i \leq n$.
7. Now, we know $C(1)$ and want to determine when $C(n-i) = C(1)$, or when $n - i = 1$. This value is $i = n - 1$.
   $C(n) = C(n-(n-1)) + n - 1 = C(1) + n - 1 = 0 + n - 1 = n - 1$.

Backward Substitution: Factorial Example

Recurrence: $C(n) = C(n-1) + 1$ for $n > 1$, and $C(1) = 0$

1. $C(n) = C(n-1) + 1$
2. Substitute $C(n-1) = C(n-2) + 1$ into original equation
3. $C(n) = [C(n-2) + 1] + 1 = C(n-2) + 2$
4. Substitute $C(n-2) = C(n-3) + 1$ into original equation
5. $C(n) = [C(n-3) + 1] + 2 = C(n-3) + 3$
6. We see the pattern $C(n) = C(n-i) + i$ emerge, where $1 \leq i \leq n$.
7. Now, we know $C(1)$ and want to determine when $C(n-i) = C(1)$, or when $n - i = 1$. This value is $i = n - 1$.
   $C(n) = C(n-(n-1)) + n - 1 = C(1) + n - 1 = 0 + n - 1 = n - 1.$

Hence $t(n) = c_{op} \cdot (n-1) \in \mathcal{O}(n)$.

Summary of these parts

- (non-recursive algorithm) Discussed now to write down expression for C(n) (number of basic operations executed in algorithm).
- Discussed how to simply this for non-recursive algorithm.
- (recursive algorithm) Discussed the idea of recurrence and how to write C(n).
- Discussed how to simply via backward substitution.

Next week: Empirical evaluation & Rule of thumb estimation of complexity!

Overview

Theory vs Practice

Formal Analysis (theoretical):

- **Pro:** No interference from implementation / hardware details.
- **Con:** Hides constant factors. Requires a different mindset.

Empirical Analysis (measure):

- **Pro:** Discovers the true impact of constant factors.
- **Con:** May be running on the “wrong” inputs.

“*In theory, theory and practice are the same. In practice, they are not.*”
– Albert Einstein

Empirical Analysis

- The complexity of an algorithm gives an estimate of the running time (we discussed this in detail).
- Measuring the *actual* time an implementation takes is also important, especially when comparing two algorithms with the same time complexity.
- When in doubt, **measure!**

Empirical Analysis of Algorithm Time Efficiency

When designing experiments:

1. Understand the experiment's purpose.

Empirical Analysis of Algorithm Time Efficiency

When designing experiments:

1. Understand the experiment's purpose.
2. Decide on the efficiency metric M to be measured and the measurement unit (an operation's count vs. a time unit).

Empirical Analysis of Algorithm Time Efficiency

When designing experiments:

1. Understand the experiment's purpose.
2. Decide on the efficiency metric M to be measured and the measurement unit (an operation's count vs. a time unit).
3. Decide on characteristics of the input sample (its range, size, and so on).

Empirical Analysis of Algorithm Time Efficiency

When designing experiments:

1. Understand the experiment's purpose.
2. Decide on the efficiency metric M to be measured and the measurement unit (an operation's count vs. a time unit).
3. Decide on characteristics of the input sample (its range, size, and so on).
4. Prepare a program implementing the algorithm (or algorithms) for the experimentation.

Empirical Analysis of Algorithm Time Efficiency

When designing experiments:

1. Understand the experiment's purpose.
2. Decide on the efficiency metric M to be measured and the measurement unit (an operation's count vs. a time unit).
3. Decide on characteristics of the input sample (its range, size, and so on).
4. Prepare a program implementing the algorithm (or algorithms) for the experimentation.
5. Generate a sample of inputs.

Empirical Analysis of Algorithm Time Efficiency

When designing experiments:

1. Understand the experiment's purpose.
2. Decide on the efficiency metric M to be measured and the measurement unit (an operation's count vs. a time unit).
3. Decide on characteristics of the input sample (its range, size, and so on).
4. Prepare a program implementing the algorithm (or algorithms) for the experimentation.
5. Generate a sample of inputs.
6. Run the algorithm (or algorithms) on the sample's inputs and record the data observed.

Empirical Analysis of Algorithm Time Efficiency

When designing experiments:

1. Understand the experiment's purpose.
2. Decide on the efficiency metric M to be measured and the measurement unit (an operation's count vs. a time unit).
3. Decide on characteristics of the input sample (its range, size, and so on).
4. Prepare a program implementing the algorithm (or algorithms) for the experimentation.
5. Generate a sample of inputs.
6. Run the algorithm (or algorithms) on the sample's inputs and record the data observed.
7. Analyse the data obtained.

Benchmarking Algorithms: Significance of input

The input and algorithm can be significant determinant of performance.

Benchmarking Algorithms: Significance of input

The input and algorithm can be significant determinant of performance.

Thereotical: Merge-sort and Quick-sort have $\mathcal{O}(n\log(n))$ complexity, while Selection-sort is $\mathcal{O}(n^2)$.

Benchmarking Algorithms: Significance of input

The input and algorithm can be significant determinant of performance.

Thereotical: Merge-sort and Quick-sort have $\mathcal{O}(n\log(n))$ complexity, while Selection-sort is $\mathcal{O}(n^2)$.

Empirical: Running Times (in seconds) for different sorting algorithms on a randomised list:

List size	500	2,500	10,000
Merge-Sort	0.8	8.1	39.8
Quick-Sort	0.3	1.3	5.3
Selection-Sort	1.5	35.0	534.7

Benchmarking Algorithms: Significance of input

The input and algorithm can be significant determinant of performance.

Thereotical: Merge-sort and Quick-sort have $\mathcal{O}(n\log(n))$ complexity, while Selection-sort is $\mathcal{O}(n^2)$.

Empirical: Running Times (in seconds) for different sorting algorithms on a randomised list:

List size	500	2,500	10,000
Merge-Sort	0.8	8.1	39.8
Quick-Sort	0.3	1.3	5.3
Selection-Sort	1.5	35.0	534.7

For ordered lists for $N = 2,500$:

	Random	In Order	In Reverse Order
Merge-Sort	8.1	7.8	7.9
Quick-Sort	1.3	35.1	37.1
Selection-Sort	35.0	34.4	35.3

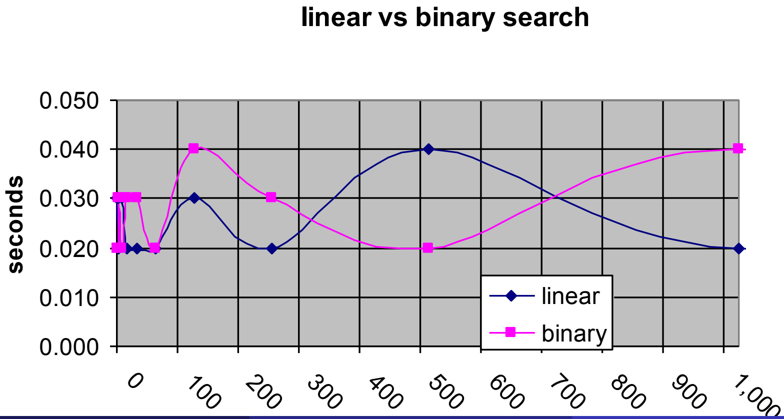
Comparing Search Algorithms: Significance of input size

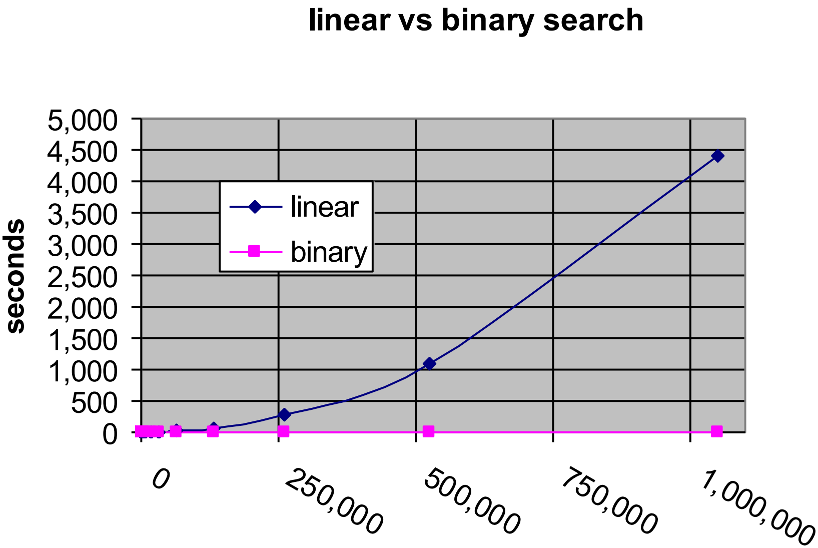
The input size can be significant determinant of performance.

Comparing Search Algorithms: Significance of input size

The input size can be significant determinant of performance.

Thereotical: Linear search have $\mathcal{O}(n)$ complexity while binary search have $\mathcal{O}(\log(n))$ complexity on a sorted list.




linear vs binary search
seconds
5,000
4,500
4,000
3,500
3,000
2,500
2,000
1,500
1,000
500
0
linear
binary
0
250,000
500,000
750,000
1,000,000

Overview

Approximate Estimate of Complexity

- As an initial analysis, sometimes we can use initiation to make an educated guess at the complexity, then use theoretical and/or empirical analysis to comfirm.

Approximate Estimate of Complexity

- As an initial analysis, sometimes we can use initiation to make an educated guess at the complexity, then use theoretical and/or empirical analysis to comfirm.
- We discuss a few such guidelines here, but will discuss more as we study further algorithms and data structures.

Approximate Estimate of Complexity

- As an initial analysis, sometimes we can use initiation to make an educated guess at the complexity, then use theoretical and/or empirical analysis to comfirm.
- We discuss a few such guidelines here, but will discuss more as we study further algorithms and data structures.
- Requires experience.

Constant time $O(1)$

- Typically algorithms or data structures whose operations does not depend on input size.

Constant time $O(1)$

- Typically algorithms or data structures whose operations does not depend on input size.
- E.g., Access to an element in an array.

Linear time $O(n)$

- Typically algorithms or data structures whose operations that needs to evaluate most or all of the elements of an problem (input size).

Linear time $O(n)$

- Typically algorithms or data structures whose operations that needs to evaluate most or all of the elements of an problem (input size).
- E.g., Scan through an array to search for an element.

Linear time $O(n)$

- Typically algorithms or data structures whose operations that needs to evaluate most or all of the elements of an problem (input size).
- E.g., Scan through an array to search for an element.
- E.g., Copy elements from a linked list to another.

Linear time $O(n)$

- Typically algorithms or data structures whose operations that needs to evaluate most or all of the elements of an problem (input size).
- E.g., Scan through an array to search for an element.
- E.g., Copy elements from a linked list to another.
- In terms of pseudo code, it usually involves a single for loop.

Quadratic time $O(n^2)$

- Typically algorithms or data structures whose operations that needs to process/evaluate pairs of elements of an problem.

Quadratic time $O(n^2)$

- Typically algorithms or data structures whose operations that needs to process/evaluate pairs of elements of an problem.
- E.g., Compare each element in an array to all its others to find duplicates (naive + unordered).

Quadratic time $O(n^2)$

- Typically algorithms or data structures whose operations that needs to process/evaluate pairs of elements of an problem.
- E.g., Compare each element in an array to all its others to find duplicates (naive + unordered).
- E.g., Compute all pairwise distance among a set of points.

Quadratic time $O(n^2)$

- Typically algorithms or data structures whose operations that needs to process/evaluate pairs of elements of an problem.
- E.g., Compare each element in an array to all its others to find duplicates (naive + unordered).
- E.g., Compute all pairwise distance among a set of points.
- In terms of pseudo code, it usually involves two nested for loops.

Overview

Summary

- We have covered the core theoretical framework for algorithmic analysis which will be used for the rest of the semester.
  - Problem input size, basic operation, time complexity, asymptotic complexity, worst/best/average cases.
  - Analysis of Non-recursive, resurive algorithms.
  - Empirical analysis.
  - Approximate analysis.

Next week, we look at first of the algorithmic paradigms in this course, brute force algorithms.