Data Structures and Algorithms

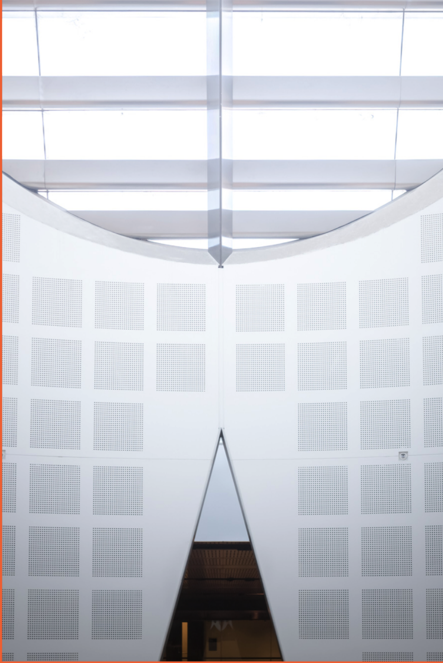
Algorithm analysis

[GT 1.1.5-1.1.6, 1.3]

Presented by

André van Renssen
School of Computer Science

THE UNIVERSITY OF SYDNEY



Three abstractions

Computational problem:

- defines a computational task
- specifies what the input is and what the output should be

Algorithm:

- a step-by-step recipe to go from input to output
  (**what** your solution does)
- different from implementation

Correctness and complexity analysis:

- a formal proof that the algorithm solves the problem
  (**why** is what your solution does correct)
- analytical bound on the resources it uses

Pseudocode

Control flow

- if … then … [else …]
- while … do …
- repeat … until …
- for … do …
- Indentation replaces braces

Method call

- method (arg [, arg…])

Return value

- return *expression*

Small example

Computational problem:
We are given an array A of integers and we need to return the maximum.

Small example

Computational problem:
We are given an array A of integers and we need to return the maximum.

Algorithm:
We go through all elements of the array in order and keep track of the largest element found so far (initially $-\infty$). / So for each position i, we check if the value stored at $A[i]$ is larger than our current maximum, and if so we update the maximum. / After scanning through the array, return the maximum we found.

Small example

Computational problem:
We are given an array A of integers and we need to return the maximum.

Optional pseudocode:

```
$max \leftarrow -\infty$
for $i \leftarrow 0$ to $n-1$ do
  if $A[i] > max$ then
    $max \leftarrow A[i]$
return $max$
```

Small example

Computational problem:
We are given an array A of integers and we need to return the maximum.

Correctness:
We maintain the following invariant: after the k-th iteration, max stores the maximum of the first k elements.

Small example

Computational problem:
We are given an array A of integers and we need to return the maximum.

Correctness:
We maintain the following invariant: after the k-th iteration, max stores the maximum of the first k elements.

Prove using induction: when $k = 0$, max is $-\infty$, which is the maximum of the first 0 elements.

Small example

Computational problem:
We are given an array A of integers and we need to return the maximum.

Correctness:
We maintain the following invariant: after the k-th iteration, max stores the maximum of the first k elements.

Prove using induction: when $k = 0$, max is $-\infty$, which is the maximum of the first 0 elements.
Assume the invariant holds for the first k iterations, we show that it holds after the $(k+1)$-th iteration. In that iteration we compare max to $A[k]$ and update max if $A[k]$ is larger. Hence, afterwards max is the maximum of the first $k+1$ elements.

Small example

Computational problem:
We are given an array A of integers and we need to return the maximum.

Correctness:
We maintain the following invariant: after the k-th iteration, max stores the maximum of the first k elements.

Prove using induction: when $k = 0$, max is $-\infty$, which is the maximum of the first 0 elements.
Assume the invariant holds for the first k iterations, we show that it holds after the $(k + 1)$-th iteration. In that iteration we compare max to $A[k]$ and update max if $A[k]$ is larger. Hence, afterwards max is the maximum of the first $k + 1$ elements.

The invariant implies that after n iterations, max contains the maximum of the first n elements, i.e., it's the maximum of A.

Example computational problem

Motivation:

- we have information about the daily fluctuation of a stock price
- we want to evaluate our best possible single-trade outcome

Input:

- an array with n integer values $A[0], A[1], ..., A[n-1]$

Task:

- find indices $0 \leq i \leq j < n$ maximizing

$$A[i] + A[i+1] + \cdots + A[j]$$

Naive algorithm

Naive algorithm

High level description:

- Iterate over every pair $0 \leq i \leq j < n$.
- For each compute $A[i] + A[i+1] + \cdots + A[j]$
- Return the pair with the maximum value

Naive algorithm

```
         value
$curr\_val$, $curr\_ans \leftarrow 0$, $(None, None)$
for $i \leftarrow 0$ to $n-1$ do
  for $j \leftarrow i$ to $n-1$ do
    // compute $A[i] + A[i+1] + \cdots + A[j]$
    $s \leftarrow 0$
    for $k \leftarrow i$ to $j$ do
      $s \leftarrow s + A[k]$
    // compare to current maximum
    if $s > curr\_val$ then
      $curr\_val$, $curr\_ans \leftarrow s$, $(i, j)$
return $curr\_ans$
```

Naive algorithm

```
$curr\_val$, $curr\_ans \leftarrow 0$, $(None, None)$
for $i \leftarrow 0$ to $n-1$ do
  for $j \leftarrow i$ to $n-1$ do
    // compute $A[i] + A[i+1] + \cdots + A[j]$
    $s \leftarrow 0$
    for $k \leftarrow i$ to $j$ do
      $s \leftarrow s + A[k]$
    // compare to current maximum
    if $s > curr\_val$ then
      $curr\_val$, $curr\_ans \leftarrow s$, $(i, j)$
return $curr\_ans$
```

Why recompute s every time?

Naive with preprocessing ☆

We can evaluate $A[i] + A[i+1] + \cdots + A[j]$ faster if we do some pre-processing.

- Pre-compute $B[i] = A[0] + A[1] + \cdots + A[i-1]$ using

$$B[i] = \begin{cases} 0 & \text{if } i = 0 \\ B[i-1] + A[i-1] & \text{if } i > 0 \end{cases}$$

- Iterate over every pair $0 \leq i \leq j < n$.
- For each compute $B[j+1] - B[i] = A[i] + A[i+1] + \cdots + A[j]$
- Return the pair with the maximum value

Naive with preprocessing

```
$curr\_val$, $curr\_ans \leftarrow 0$, $(None, None)$
$B \leftarrow$ new array of size $n + 1$
$B[0] \leftarrow 0$
for $i \leftarrow 1$ to $n + 1$ do
  $B[i] \leftarrow B[i-1] + A[i-1]$
for $i \leftarrow 0$ to $n - 1$ do
  for $j \leftarrow i$ to $n - 1$ do
    // compute $A[i] + A[i+1] + \cdots + A[j]$
    $s \leftarrow B[j+1] - B[i]$
    // compare to current maximum
    if $s > curr\_val$ then
      $curr\_val$, $curr\_ans \leftarrow s$, $(i, j)$
return $curr\_ans$
```

Efficiency

Definition (first attempt)
An algorithm is efficient if it runs quickly on real input instances

Not a good definition because it is not easy to evaluate:
- instances considered
- implementation details
- hardware it runs on

Our definition should implementation independent:
- count number of “steps”
- bound the algorithm’s worst-case performance

Efficiency

Definition (second attempt)

An algorithm is efficient if it achieves qualitatively better worst-case performance than a brute-force approach

Not a good definition because it is subjective:

- brute-force approach is ill-defined
- qualitatively better is ill-defined

Our definition should be objective:

- not tied to a strawman baseline
- independently agreed upon

Efficiency

Definition
An algorithm is efficient if it runs in polynomial time; that is, on an instance of size n, it performs no more than $p(n)$ steps for some polynomial $p(x) = a_d x^d + \cdots + a_1 x + a_0$.

This gives us some information about the expected behavior of the algorithm and is useful for making predictions and comparing different algorithms.

Asymptotic growth analysis

Let $T(n)$ be the worst-case number of steps of our algorithm on an instance of size n.

Problem: figuring out $T(n)$ *exactly* might be really hard! Also, the fine-grained details are not necessarily that relevant.

Example: $T(n) = 4n^2 + 4n + 5$, or $T(n) = 5n^2 - 2n + 100$. Which one is best? Do the constants matter (recall, one "step" might take a slightly different time based on implementation or architecture details)?

Asymptotic growth analysis

Insight: in both examples, the worst-case number of steps $T(n)$ grew *quadratically* with n.
If n is multiplied by 2, then we expect $T(n)$ to be multiplied by 4.

More generally: if $T(n)$ is a polynomial of degree d, then doubling the size of the input should roughly increase the running time by a factor of 2^d.

Asymptotic growth analysis gives us a tool for focusing on the terms that make up $T(n)$, which **dominate** the running time.

Asymptotic growth analysis

Recap: We want to analyse $T(n)$, the worst-case number of steps of our algorithm on an instance of size n.

But figuring out $T(n)$ exactly might be very hard (impossible), and also is often "too much information".

We instead do asymptotic growth analysis (Big-Oh notation), which provides a coarser but sufficient way to summarise how $T(n)$ behaves when n increases.

Asymptotic growth analysis

Definition
We say that $T(n) = O(f(n))$ if
there exist $n_0, c > 0$ such that $T(n) \leq cf(n)$ for all $n > n_0$.

Definition
We say that $T(n) = \Omega(f(n))$ if
there exist $n_0, c > 0$ such that $T(n) \geq cf(n)$ for all $n > n_0$.

Definition
We say that $T(n) = \Theta(f(n))$ if
$T(n) = O(f(n))$ and $T(n) = \Omega(f(n))$.

Asymptotic growth analysis

Definition

We say that $T(n) = O(f(n))$ if
there exist $n_0, c > 0$ such that $T(n) \leq cf(n)$ for all $n > n_0$.

$T(n) = 32n^2 + 17n + 32$
$T(n)$ is $O(n^2)$ and $O(n^3)$, but not $O(n)$.

Definition

We say that $T(n) = \Omega(f(n))$ if
there exist $n_0, c > 0$ such that $T(n) \geq cf(n)$ for all $n > n_0$.

Definition

We say that $T(n) = \Theta(f(n))$ if
$T(n) = O(f(n))$ and $T(n) = \Omega(f(n))$.

Asymptotic growth analysis

Definition

We say that $T(n) = O(f(n))$ if
there exist $n_0, c > 0$ such that $T(n) \leq cf(n)$ for all $n > n_0$.

$T(n) = 32n^2 + 17n + 32$
$T(n)$ is $O(n^2)$ and $O(n^3)$, but not $O(n)$.

Definition

We say that $T(n) = \Omega(f(n))$ if
there exist $n_0, c > 0$ such that $T(n) \geq cf(n)$ for all $n > n_0$.

$T(n) = 32n^2 + 17n + 32$
$T(n)$ is $\Omega(n^2)$ and $\Omega(n)$, but not $\Omega(n^3)$.

Definition

We say that $T(n) = \Theta(f(n))$ if
$T(n) = O(f(n))$ and $T(n) = \Omega(f(n))$.

Asymptotic growth analysis

Definition

We say that $T(n) = O(f(n))$ if
there exist $n_0, c > 0$ such that $T(n) \leq cf(n)$ for all $n > n_0$.

$T(n) = 32n^2 + 17n + 32$
$T(n)$ is $O(n^2)$ and $O(n^3)$, but not $O(n)$.

Definition

We say that $T(n) = \Omega(f(n))$ if
there exist $n_0, c > 0$ such that $T(n) \geq cf(n)$ for all $n > n_0$.

$T(n) = 32n^2 + 17n + 32$
$T(n)$ is $\Omega(n^2)$ and $\Omega(n)$, but not $\Omega(n^3)$.

Definition

We say that $T(n) = \Theta(f(n))$ if
$T(n) = O(f(n))$ and $T(n) = \Omega(f(n))$.

$T(n) = 32n^2 + 17n + 32$
$T(n)$ is $\Theta(n^2)$, but not $\Theta(n)$ or $\Theta(n^3)$.

Asymptotic growth analysis

tl;dr: think of those as

- $T(n) = O(f(n))$: $T(n)$ is "smaller" than $f(n)$ (up to a constant factor)
- $T(n) = \Omega(f(n))$: $T(n)$ is "bigger" than $f(n)$ (up to a constant factor)
- $T(n) = \Theta(f(n))$: $T(n)$ is "equal" to $f(n)$ (up to a constant factor)

Important: Asymptotic growth analysis $(O(\cdot), \Omega(\cdot), \Theta(\cdot))$ is just a mathematical tool to compare functions when the input n grows. We are using this tool to analyse the worst-case behaviour of our algorithms. But asymptotic growth analysis and worst-case analysis are not the same thing (they just go hand in hand)!

Examples of asymptotic growth

Polynomial

Logarithmic

Exponential

Examples of asymptotic growth

Polynomial

$O(n^d)$, considered efficient since most algorithms have small c

Logarithmic

Exponential

Examples of asymptotic growth

Polynomial

$O(n^d)$, considered efficient since most algorithms have small c

Logarithmic

$O(\log n)$, typical for search algorithms like Binary Search

Exponential

Examples of asymptotic growth

Polynomial

$O(n^d)$, considered efficient since most algorithms have small c

Logarithmic

$O(\log n)$, typical for search algorithms like Binary Search

Exponential

$O(2^n)$, typical for brute force algorithms exploring all possible combinations of elements

Comparison of running times

each operation takes 1 nanosecond

size	n	$n \log n$	n^2	n^3	2^n	$n!$
10	$<$ 1s	$<$ 1s	$<$ 1s	<1s	<1s	3s
50	$<$ 1s	$<$ 1s	$<$ 1s	<1s	17m	-
100	$<$ 1s	$<$ 1s	$<$ 1s	1s	35y	-
1,000	$<$ 1s	$<$ 1s	1s	15m	-	-
10,000	$<$ 1s	$<$ 1s	2s	11d	-	-
100,000	$<$ 1s	1s	2h	31y	-	-
1,000,000	1s	10s	4d	-	-	-

Properties of asymptotic growth

Transitivity:

- If $f = O(g)$ and $g = O(h)$ then $f = O(h)$
- If $f = \Omega(g)$ and $g = \Omega(h)$ then $f = \Omega(h)$
- If $f = \Theta(g)$ and $g = \Theta(h)$ then $f = \Theta(h)$

Sums of functions:

- If $f = O(g)$ and $g = O(h)$ then $f + g = O(h)$
- If $f = \Omega(h)$ then $f + g = \Omega(h)$

Asymptotic analysis is a powerful tool that allows us to ignore unimportant details and focus on what's important.

Survey of common running times

Let $T(n)$ be the running time of our algorithm.

We say that $T(n)$ is ...	if ...
constant	$T(n) = \Theta(1)$
logarithmic	$T(n) = \Theta(\log n)$
linear	$T(n) = \Theta(n)$
quasi-linear	$T(n) = \Theta(n \log n)$
quadratic	$T(n) = \Theta(n^2)$
cubic	$T(n) = \Theta(n^3)$
exponential	$T(n) = \Theta(c^n)$

What operations take $O(1)$ time?

Constant time:
Running time does not depend on the size of the input.

- Assignments ($a \leftarrow 42$)
- Comparisons ($=$, $<$, $>$)
- Boolean operations (and, or, not)
- Basic mathematical operations (+, -, *, /)
- Constant sized combinations of the above ($a \leftarrow (2 * b + c)/4$)

Recall stock trading problem

Motivation:

- we have information about the daily fluctuation of a stock price
- we want to evaluate our best possible single-trade outcome

Input:

- an array with n integer values $A[0], A[1], ..., A[n-1]$

Task:

- find indices $0 \leq i \leq j < n$ maximizing

$$A[i] + A[i+1] + \cdots + A[j]$$

Naive algorithm

```
$curr\_val$, $curr\_ans \leftarrow 0$, $(None, None)$
for $i \leftarrow 0$ to $n-1$ do
  for $j \leftarrow i$ to $n-1$ do
    // compute $A[i] + A[i+1] + \cdots + A[j]$
    $s \leftarrow 0$
    for $k \leftarrow i$ to $j$ do
      $s \leftarrow s + A[k]$
    // compare to current maximum
    if $s > curr\_val$ then
      $curr\_val$, $curr\_ans \leftarrow s$, $(i, j)$
return $curr\_ans$
```

Naive algorithm

```
$curr\_val$, $curr\_ans \leftarrow 0$, $(None, None)$                          $O(1)$
for $i \leftarrow 0$ to $n-1$ do
  for $j \leftarrow i$ to $n-1$ do
    // compute $A[i] + A[i+1] + \cdots + A[j]$
    $s \leftarrow 0$                                    $\}\, O(1)$
    for $k \leftarrow i$ to $j$ do
      $s \leftarrow s + A[k]$                           $\}\, O(1)$
    // compare to current maximum
    if $s > curr\_val$ then                             $\Big\}\, O(1)$
      $curr\_val$, $curr\_ans \leftarrow s$, $(i, j)$
return $curr\_ans$                                                             $O(1)$
```

Naive algorithm

```
$curr\_val$, $curr\_ans \leftarrow 0$, $(None, None)$                                   $O(1)$
for $i \leftarrow 0$ to $n-1$ do
  for $j \leftarrow i$ to $n-1$ do
    // compute $A[i] + A[i+1] + \cdots + A[j]$
    $s \leftarrow 0$                                     $\}\ O(1)$
    for $k \leftarrow i$ to $j$ do
      $s \leftarrow s + A[k]$                           $\}\ O(1)$      $\}\ O(j-i)$
    // compare to current maximum
    if $s > curr\_val$ then                                $\}\ O(1)$
      $curr\_val$, $curr\_ans \leftarrow s$, $(i, j)$
return $curr\_ans$                                                                     $O(1)$
```

Naive algorithm

```
$curr\_val$, $curr\_ans \leftarrow 0$, $(None, None)$                          $O(1)$
for $i \leftarrow 0$ to $n-1$ do
  for $j \leftarrow i$ to $n-1$ do
    // compute $A[i] + A[i+1] + \cdots + A[j]$
    $s \leftarrow 0$                                   } $O(1)$
    for $k \leftarrow i$ to $j$ do
      $s \leftarrow s + A[k]$                          } $O(1)$        } $O(j-i)$   } $\sum_{j=i}^{n-1}$
    // compare to current maximum
    if $s > curr\_val$ then
      $curr\_val$, $curr\_ans \leftarrow s$, $(i, j)$     } $O(1)$
return $curr\_ans$                                                              $O(1)$
```

Naive algorithm

```
$curr\_val$, $curr\_ans \leftarrow 0$, $(None, None)$                        $O(1)$
for $i \leftarrow 0$ to $n-1$ do
  for $j \leftarrow i$ to $n-1$ do
    // compute $A[i] + A[i+1] + \cdots + A[j]$
    $s \leftarrow 0$                          } $O(1)$
    for $k \leftarrow i$ to $j$ do                                              } $\sum_{i=0}^{n-1}$
      $s \leftarrow s + A[k]$                 } $O(1)$   } $O(j-i)$   } $\sum_{j=i}^{n-1}$
    // compare to current maximum
    if $s > curr\_val$ then
      $curr\_val$, $curr\_ans \leftarrow s$, $(i, j)$   } $O(1)$
return $curr\_ans$                                                         $O(1)$
```

Naive algorithm

Let $T(n)$ be the running time of the Naive Algorithm on an instance of size n.

$$T(n) = O(1) \overset{\star}{+} \sum_{i=0}^{n-1} \sum_{j=i}^{n-1} O(j-i)$$

Naive algorithm

Let $T(n)$ be the running time of the Naive Algorithm on an instance of size n.

$$\begin{aligned} T(n) &= O(1) + \sum_{i=0}^{n-1}\sum_{j=i}^{n-1} O(j-i) \\ &= O(1) + \sum_{i=0}^{n-1}\sum_{j=0}^{n-1} O(n) \\ &= O(1) + \sum_{i=0}^{n-1} O(n^2) \\ &= O(1) + O(n^3) \\ &= O(n^3) \end{aligned}$$

See GT 1.2 for a refresher if needed.

Naive with preprocessing

```
$curr\_val$, $curr\_ans \leftarrow 0$, $(None, None)$
$B \leftarrow$ new array of size $n+1$
$B[0] \leftarrow 0$
for $i \leftarrow 1$ to $n+1$ do
  $B[i] \leftarrow B[i-1] + A[i-1]$
for $i \leftarrow 0$ to $n-1$ do
  for $j \leftarrow i$ to $n-1$ do
    // compute $A[i] + A[i+1] + \cdots + A[j]$
    $s \leftarrow B[j+1] - B[i]$
    // compare to current maximum
    if $s > curr\_val$ then
      $curr\_val$, $curr\_ans \leftarrow s$, $(i, j)$
return $curr\_ans$
```

Naive with preprocessing

```
$curr\_val$, $curr\_ans \leftarrow 0$, $(None, None)$
$B \leftarrow$ new array of size $n+1$
$B[0] \leftarrow 0$
for $i \leftarrow 1$ to $n+1$ do
  $B[i] \leftarrow B[i-1] + A[i-1]$
for $i \leftarrow 0$ to $n-1$ do
  for $j \leftarrow i$ to $n-1$ do
    // compute $A[i] + A[i+1] + \cdots + A[j]$
    $s \leftarrow B[j+1] - B[i]$
    // compare to current maximum
    if $s > curr\_val$ then
      $curr\_val$, $curr\_ans \leftarrow s$, $(i, j)$
return $curr\_ans$
```

Improvement: $O(n^3) \rightarrow O(n^2)$

Recap

Asymptotic growth analysis gives us some information about the worst-case behavior of the algorithm. It is useful for making predictions and comparing different algorithms.

Why do we make a distinction between problem, algorithm, implementation and analysis?

- somebody can design a better algorithm for a given problem
- somebody can come up with better implementation
- somebody can come up with better analysis

A note on style

For your assessments, you will have to design and analyse an algorithm for a given problem. This always consists of three steps:

- Describe your algorithm: A high level description in **English**, optionally followed by pseudocode. Never submit code!
- Prove its correctness: A formal proof that the algorithm does what it's supposed to do.
- Analyse its time complexity: A formal proof that the algorithm runs in the time you claim it does.

Try to model your own solution after the solution published for the tutorial sheets. You are encouraged to use LaTeX.

A note on pseudocode style

What we will be using in this class closely follows the Python syntax:

- Arrays: we use zero-based indexing
- Slices: `[i:j:k]` is equivalent to Python's `range(i, j, k)`.
- References: Every non-basic data type is passed by reference

But we will deviate when writing things in plain English leads to easier to understand code.