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
- different from implementation

Correctness and complexity analysis:

- a formal proof that the algorithm solves the problem
- analytical bound on the resources it uses

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

for every possible starting index i do
  for every possible ending index j not before i do
    compute $A[i] + A[i+1] + \cdots + A[j]$
    compare to current maximum

Naive algorithm

$result \leftarrow 0$
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 $s > result$ then
      $result \leftarrow s$
return $result$

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.

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

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

Definition
We say that $T(n) = \Omega(f(n))$ if
there exists $n_0, c > 0$ 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 $T(n) = \Omega(f(n))$

Examples of asymptotic growth

Polynomial

Logarithmic

Exponential

Examples of asymptotic growth

Polynomial

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

Logarithmic

Exponential

Examples of asymptotic growth

Polynomial

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

Logarithmic

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

Exponential

Examples of asymptotic growth

Polynomial

$O(n^c)$, consider 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

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 ...
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	$T(n) = \Theta(c^n)$

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

$result \leftarrow 0$
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 $s > result$ then
      $result \leftarrow s$
return $result$

Naive algorithm

$result \leftarrow 0$ $O(1)$

for $i \leftarrow 0$ to $n-1$ do

  for $j \leftarrow i$ to $n-1$ do

    {Compute $A[i] + A[i+1] + \cdots + A[j]$}

    $s \leftarrow 0$

    for $k \leftarrow i$ to j do

      $s \leftarrow s + A[k]$

    {Compare to current maximum}

    if $s > result$ then

      $result \leftarrow s$

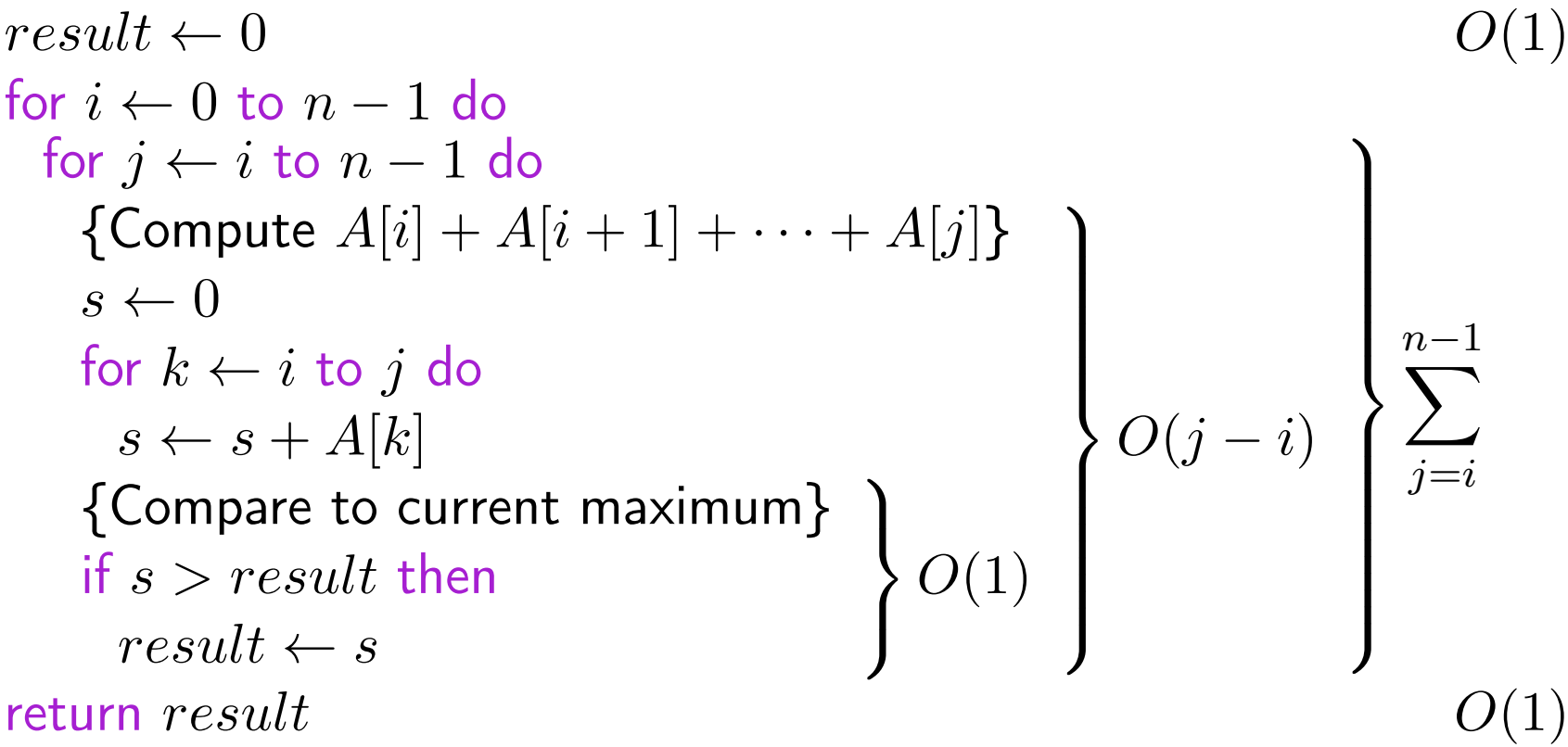
} $O(1)$

return $result$ $O(1)$

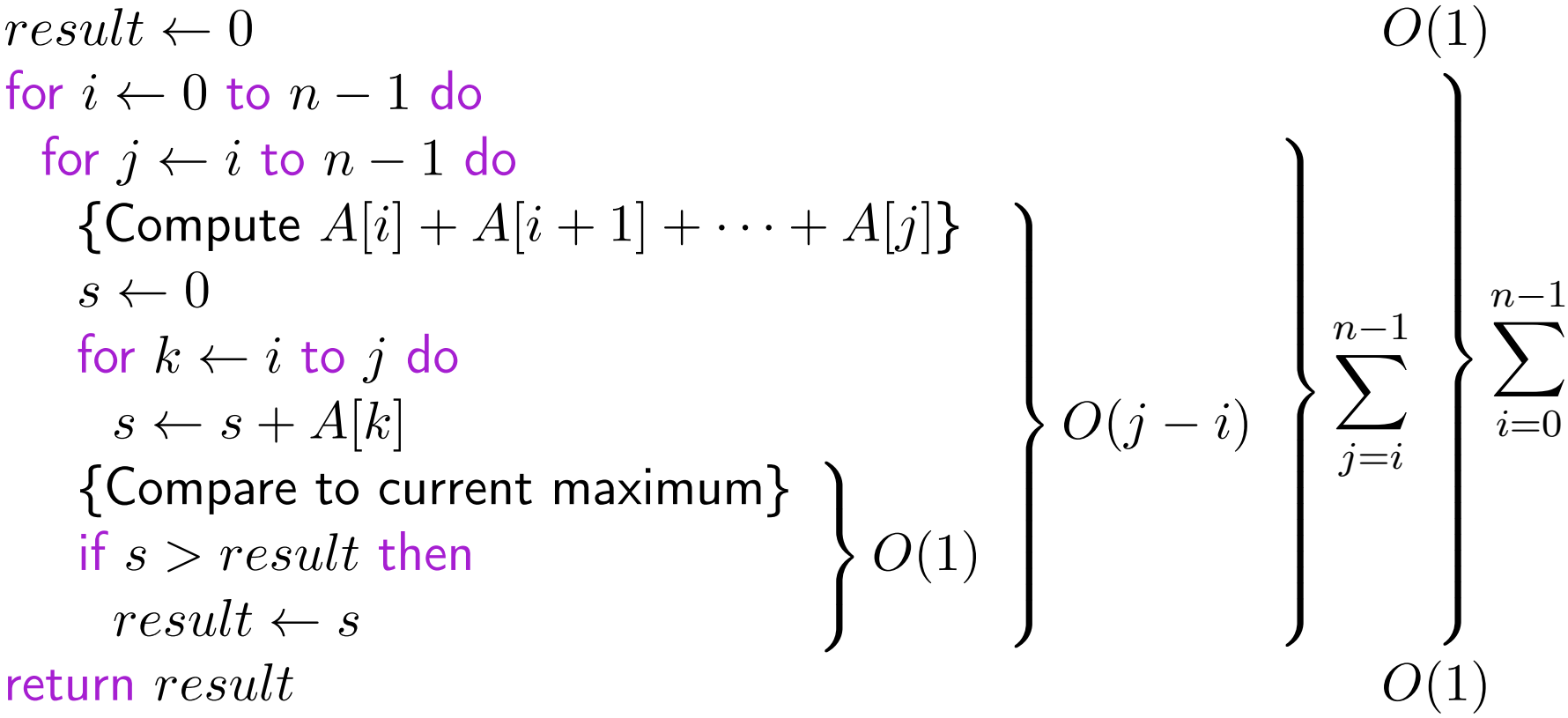
Naive algorithm

$$
\begin{array}{ll}
result \leftarrow 0 & O(1) \\
\textsf{for } i \leftarrow 0 \textsf{ to } n-1 \textsf{ do} & \\
\quad \textsf{for } j \leftarrow i \textsf{ to } n-1 \textsf{ do} & \\
\quad\quad \{\text{Compute } A[i] + A[i+1] + \cdots + A[j]\} & \\
\quad\quad s \leftarrow 0 & \\
\quad\quad \textsf{for } k \leftarrow i \textsf{ to } j \textsf{ do} & \\
\quad\quad\quad s \leftarrow s + A[k] & O(j-i) \\
\quad\quad \{\text{Compare to current maximum}\} & \\
\quad\quad \textsf{if } s > result \textsf{ then} & O(1) \\
\quad\quad\quad result \leftarrow s & \\
\textsf{return } result & O(1)
\end{array}
$$

Naive algorithm



Naive algorithm



Naive algorithm

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

Naive algorithm

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

See GT 1.2 for a refresher if needed.

Naive with preprocessing

```
result ← 0
for i ← 0 to n − 1 do
  for j ← i to n − 1 do
    {Compute A[i] + A[i + 1] + ··· + A[j]}
    s ← 0
    for k ← i to j do
      s ← s + A[k]
    {Compare to current maximum}
    if s > result then
      result ← s
return result
```

Why recompute s every time?

Naive with preprocessing

Notation:
$S_i = A[0] + A[1] + \cdots + A[i]$

Naive with preprocessing

Notation:
$S_i = A[0] + A[1] + \cdots + A[i]$

Observe:
$A[i] + A[i+1] + \cdots + A[j] = S_j - S_{i-1}$

Naive with preprocessing

Notation:
$S_i = A[0] + A[1] + \cdots + A[i]$

Observe:
$A[i] + A[i+1] + \cdots + A[j] = S_j - S_{i-1}$

So if we precompute $S_0, \ldots, S_{n-1}$, we can compute the sum of any subsequence in $O(1)$ time.

Naive with preprocessing

$result \leftarrow 0$
$S_{-1} \leftarrow 0$
for $i \leftarrow 0$ to $n-1$ do
  $S_i \leftarrow S_{i-1} + A[i]$
for $i \leftarrow 0$ to $n-1$ do
  for $j \leftarrow i$ to $n-1$ do
    {Compute $A[i] + A[i+1] + \cdots + A[j]$}
    $s \leftarrow S_j - S_{i-1}$
    {Compare to current maximum}
    if $s > result$ then
      $result \leftarrow s$
return $result$

Naive with preprocessing

$result \leftarrow 0$
$S_{-1} \leftarrow 0$
for $i \leftarrow 0$ to $n-1$ do
  $S_i \leftarrow S_{i-1} + A[i]$
for $i \leftarrow 0$ to $n-1$ do
  for $j \leftarrow i$ to $n-1$ do
    {Compute $A[i] + A[i+1] + \cdots + A[j]$}
    $s \leftarrow S_j - S_{i-1}$
    {Compare to current maximum}
    if $s > result$ then
      $result \leftarrow s$
return $result$

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

Reuse computation

Can we do even better?

Reuse computation

Can we do even better?

YES! Using dynamic programming.

Reuse computation

Can we do even better?

YES! Using dynamic programming.

Instead of precomputing the prefix sum S_i, compute the suffix sum M_j = the maximum subsequence that ends at index j:
$M_j = \max_{0 \le i \le j} \{A[i:j]\}$.

Reuse computation

Can we do even better?

YES! Using dynamic programming.

Instead of precomputing the prefix sum S_i, compute the suffix sum
$M_j =$ the maximum subsequence that ends at index j:
$M_j = \max_{0 \le i \le j} \{A[i:j]\}$.

The empty subsequence should also be allowed:
$M_j = \max\{0, \max_{0 \le i \le j} \{A[i:j]\}\}$.

Reuse computation

Example:
A: 3, 5, -10, 5, 2, -4, 9

Reuse computation

Example:
A: 3, 5, -10, 5, 2, -4, 9

$M_0 = 3$
$M_1 = 8$

Reuse computation

Example:
A: 3, 5, -10, 5, 2, -4, 9

$M_0 = 3$
$M_1 = 8$
$M_2 = 0$

Reuse computation

Example:
A: 3, 5, -10, 5, 2, -4, 9

$$M_0 = 3$$
$$M_1 = 8$$
$$M_2 = 0$$
$$M_3 = 5$$
$$M_4 = 7$$

Reuse computation

Example:
A: 3, 5, -10, 5, 2, -4, 9

$$M_0 = 3$$
$$M_1 = 8$$
$$M_2 = 0$$
$$M_3 = 5$$
$$M_4 = 7$$
$$M_5 = 3$$

Reuse computation

Example:
A: 3, 5, -10, 5, 2, -4, 9

$$M_0 = 3$$
$$M_1 = 8$$
$$M_2 = 0$$
$$M_3 = 5$$
$$M_4 = 7$$
$$M_5 = 3$$
$$M_6 = 12$$

Reuse computation

How to compute M_j efficiently?

Reuse computation

How to compute M_j efficiently?

M_0:
If $A[0] \geq 0$, the maximum subsequence that ends at $A[0]$ is $A[0]$.
Otherwise, it is empty.

Reuse computation

How to compute M_j efficiently?

M_0:
If $A[0] \geq 0$, the maximum subsequence that ends at $A[0]$ is $A[0]$.
Otherwise, it is empty.

$$M_0 = \max\{0, A[0]\}$$

Reuse computation

How to compute M_j efficiently?

M_j:
If $M_{j-1} + A[j] \geq 0$, we gain something by using M_{j-1} before $A[j]$.
So $M[j]$ is $M[j-1]$ with A[j] appended to it.
Otherwise, it is better to start a new sequence after $A[j]$. Thus, M[j] is empty.

Reuse computation

How to compute M_j efficiently?

M_j:
If $M_{j-1} + A[j] \geq 0$, we gain something by using M_{j-1} before $A[j]$.
So $M[j]$ is $M[j-1]$ with A[j] appended to it.
Otherwise, it is better to start a new sequence after $A[j]$. Thus, M[j] is empty.

$$M_j = \max\{0, M_{j-1} + A[j]\}$$

Reuse computation

$$M_0 = \max\{0, A[0]\}$$
$$M_j = \max\{0, M_{j-1} + A[j]\}$$

Reuse computation

$$M_0 = \max\{0, A[0]\}$$
$$M_j = \max\{0, M_{j-1} + A[j]\}$$

$M_0 \leftarrow \max\{0, A[0]\}$
for $i \leftarrow 1$ to $n-1$ do
  $M_i \leftarrow \max\{0, M_{i-1} + A[i]\}$
$result \leftarrow M_0$
for $i \leftarrow 0$ to $n-1$ do
  $result \leftarrow \max\{result, M_i\}$
return $result$

Reuse computation

$$M_0 = \max\{0, A[0]\}$$
$$M_j = \max\{0, M_{j-1} + A[j]\}$$

$M_0 \leftarrow \max\{0, A[0]\}$
for $i \leftarrow 1$ to $n-1$ do
  $M_i \leftarrow \max\{0, M_{i-1} + A[i]\}$
$result \leftarrow M_0$
for $i \leftarrow 0$ to $n-1$ do
  $result \leftarrow \max\{result, M_i\}$
return $result$

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

Recap

Asymptotic time complexity gives us some information about the expected 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 analyze 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.
- Analyze 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

We will be using in this class follows closely 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.

This week

Tutorial sheet 1:

- posted last Monday
- make sure you work out a few problems before the tutorial

Assignment 1:

- posted on Fri 28 Feb
- due on Thu 12 Mar