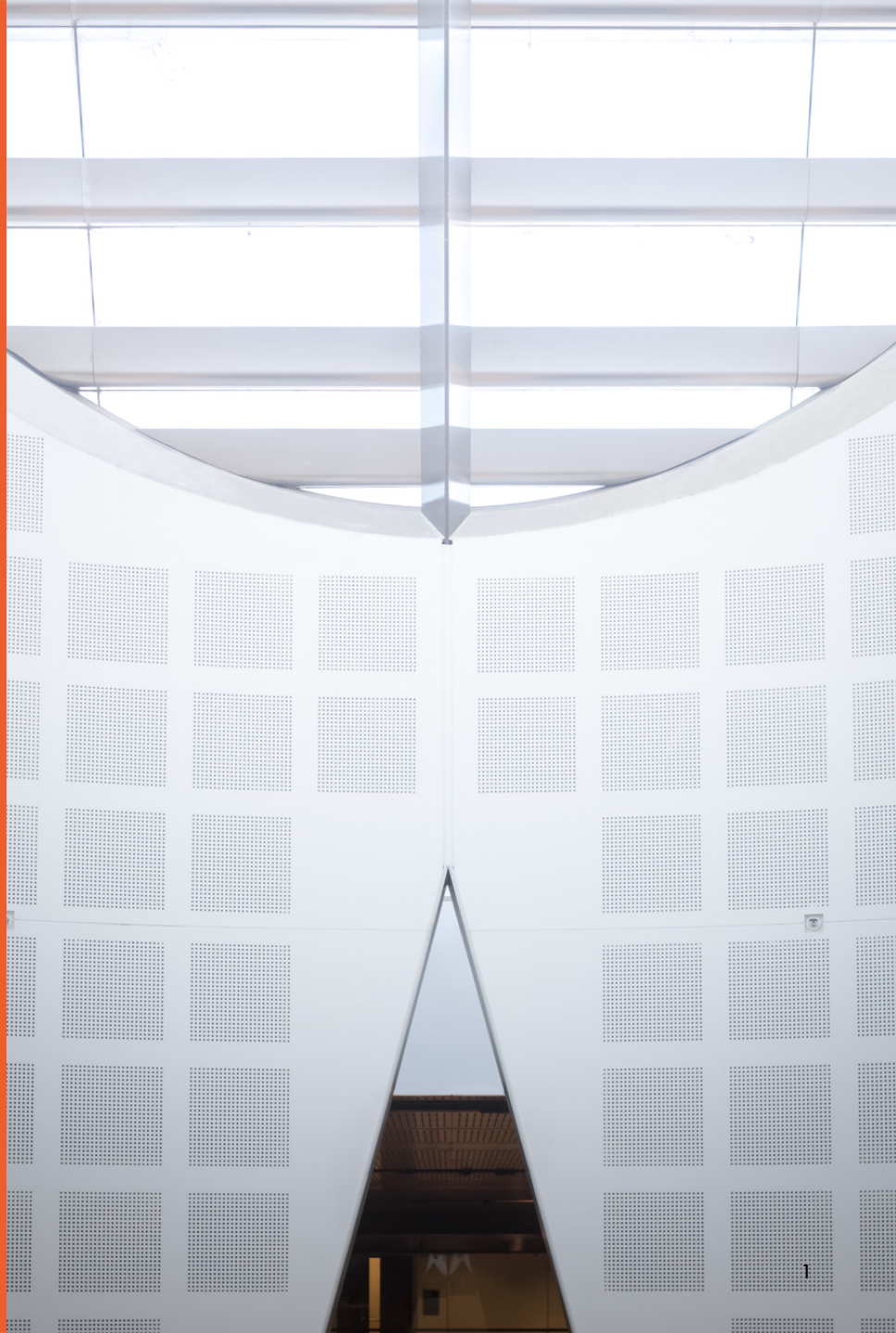
COMP3027: Algorithm Design

Lecture 1a: Admin

William Umboh

School of Computer Science

THE UNIVERSITY OF SYDNEY



Aims of this unit

This unit provides an introduction to the design and analysis of algorithms. We will learn about

- (i) how to reason about algorithms rigorously: Is it correct? Is it fast? Can we do better?
- (ii) how to develop algorithmic solutions to computational problems

Assumes:

- basic knowledge of data structures (stacks, queues, binary trees) and programming at level of COMP2123

- discrete math (graphs, big O notation, proof techniques) at level of MATH1004/MATH1064

Course Arrangements

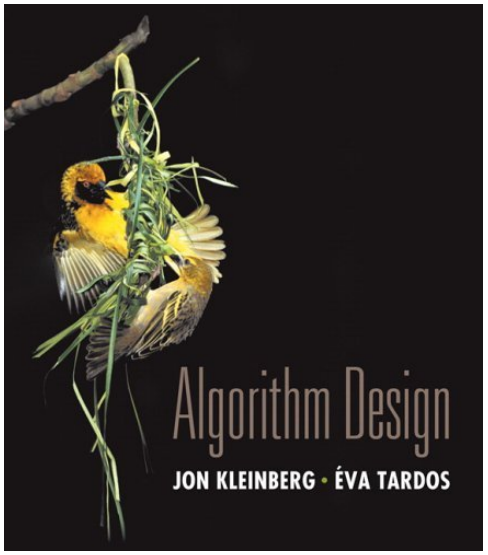
Course page: Canvas and Ed

Lecturer: William Umboh
Level 4, Room 410, School of Computer Science
william.umboh@sydney.edu.au
Ph. 0286277122

Tutors:

Lindsey Deryckere (TA)	Ben Gane
Oliver Scarlet (TA)	Allen Lu
Ryder Chen	Ada Fang
Cameron Eggins	Holly Craig
James Wood	Tristan Heywood
Lilian Hunt	TJ Kojima
Chris Natoli (programming)	

Course Arrangements



Course book:

J. Kleinberg and E. Tardos
Algorithm Design

Additional Reference:

J. Erickson
Algorithms
Available free online

Outline:

13 lectures (Thu 10-12 & Thu 4-5 (Adv))
4 assignments
10 quizzes
Exam

Tutorials:

12 tutorials

Assessment

Assessment:

- Quizzes 15% (average of best 8 out of 10)
- Each assignment 6.25% (4 assignments – total 25%)
- Exam 60% (minimum 40% required to pass)

Submissions:

- Theory part – Gradescope
  - Entry code 3YXX5Y
  - Sign up using university email
- Implementation – Ed (Assignments 1 and 2 only)

Collaboration:

- General ideas – Yes!
- Formulation and writing – No!
  - Read Academic Dishonesty and Plagiarism.

Quizzes

10 assessed quizzes (and one self-review quiz)

Average of best 8 of the 10 quizzes will count

Worth 15% of final mark

Quizzes are due 23:59:00 AEDT on each Sunday
- Late submissions will **not** be accepted

You get a single attempt at each quiz only

You have 20 minutes from the time you open the quiz

Assignments

There will be **4** homework assignments

The objective of these is to teach problem solving skills and how to communicate your ideas clearly

Late submissions will be penalized by 20% of the total marks per day. Assignments > 2 days late get 0.

For example, say you get 90% on your assignment:

If submitted on time = 90%
Late but within 24 hours = 70%
Between 24 and 48 hours = 50%

Theory part needs to be typed (LaTeX > GDocs, Word), no handwritten submissions accepted

Some assignments will involve programming (only Python allowed)
The simplicity of Python lets one focus on core algorithmic ideas

Academic Integrity (University policy)

- "The University of Sydney is unequivocally opposed to, and intolerant of, plagiarism and academic dishonesty.
  - Academic dishonesty means seeking to obtain or obtaining academic advantage for oneself or for others (including in the assessment or publication of work) by dishonest or unfair means.
  - Plagiarism means presenting another person's work as one's own work by presenting, copying or reproducing it without appropriate acknowledgement of the source." [from site below]
- http://sydney.edu.au/elearning/student/EI/index.shtml
- Submitted work is compared against other work (from students, the internet etc)
  - Turnitin for textual tasks (through eLearning), other systems for code
- Penalties for academic dishonesty or plagiarism can be severe
- Complete self-education AHEM1001

Academic Integrity (University policy)

- The penalties are severe and include:

  1) a permanent record of academic dishonesty, plagiarism and misconduct in the University database and on your student file

  2) mark deduction, ranging from 0 for the assignment to Fail for the course

  3) expulsion from the University and cancelling of your student visa

Academic Integrity (University policy)

- Do not confuse legitimate co-operation and cheating! You can discuss the assignment with another student, this is legitimate collaboration, but you cannot complete the assignment together – everyone must write their own code or report

- When there is copying between students, note that both students are penalised – the student who copies and the student who makes his/her work available for copying

- You may not seek external help on assignments, e.g. posting the assignment online (such as Chegg, StackExchange, etc.), private tutors

Final exam

The final will be 2 hours long, consisting of 4 problems similar to those seen in the tutorials and assignments

The final will test your problem solving skills

There is a 40% exam barrier

The final exam represents 60% of your final mark

Our advice is that you work hard on the assignments throughout the semester. It's the best preparation for the final

Tutorials

After the main lecture, we will post a tutorial sheet for the week on Ed

To get the most out of the tutorial, try to solve as many problems as you can *before* the tutorial. Your tutor is there to help you out if you get stuck, not to lecture

We will post solutions to tutorials on Ed

If you are unable to attend a tutorial, you may ask your questions on Ed

Contacting us

Unless you have a personal issue, do not send us direct email

Instead, post your question on Ed so that others can benefit from the answers.

Feel free to answer another student's question. This will help you digest the material as well. The best way to learn is to teach others.

The staff will vet student answers.

Tips for success

This course emphasises creative problem-solving and being able to explain solution to others

Passively listening to lectures and tutorials, reading slides will not cut it

The only way to learn is by solving problems and explaining it

Participate actively in lectures, tutorials and Ed

Special Consideration (University policy)

- If your performance on assessments is affected by illness or misadventure
- Follow proper bureaucratic procedures
  - Have professional practitioner sign special USyd form
  - Submit application for special consideration online, upload scans
  - Note you have only a quite short deadline for applying
  - http://sydney.edu.au/current_students/special_consideration/
- Also, notify coordinator by email *as soon as anything begins to go wrong*
- There is a similar process if you need special arrangements eg for religious observance, military service, representative sports

Assistance

- There are a wide range of support services available for students
- Please make contact, and get help
- You are not required to tell anyone else about this
- If you are willing to inform the unit coordinator, they may be able to work with other support to reduce the impact on this unit
  - eg provide advice on which tasks are most significant

Do you have a disability?

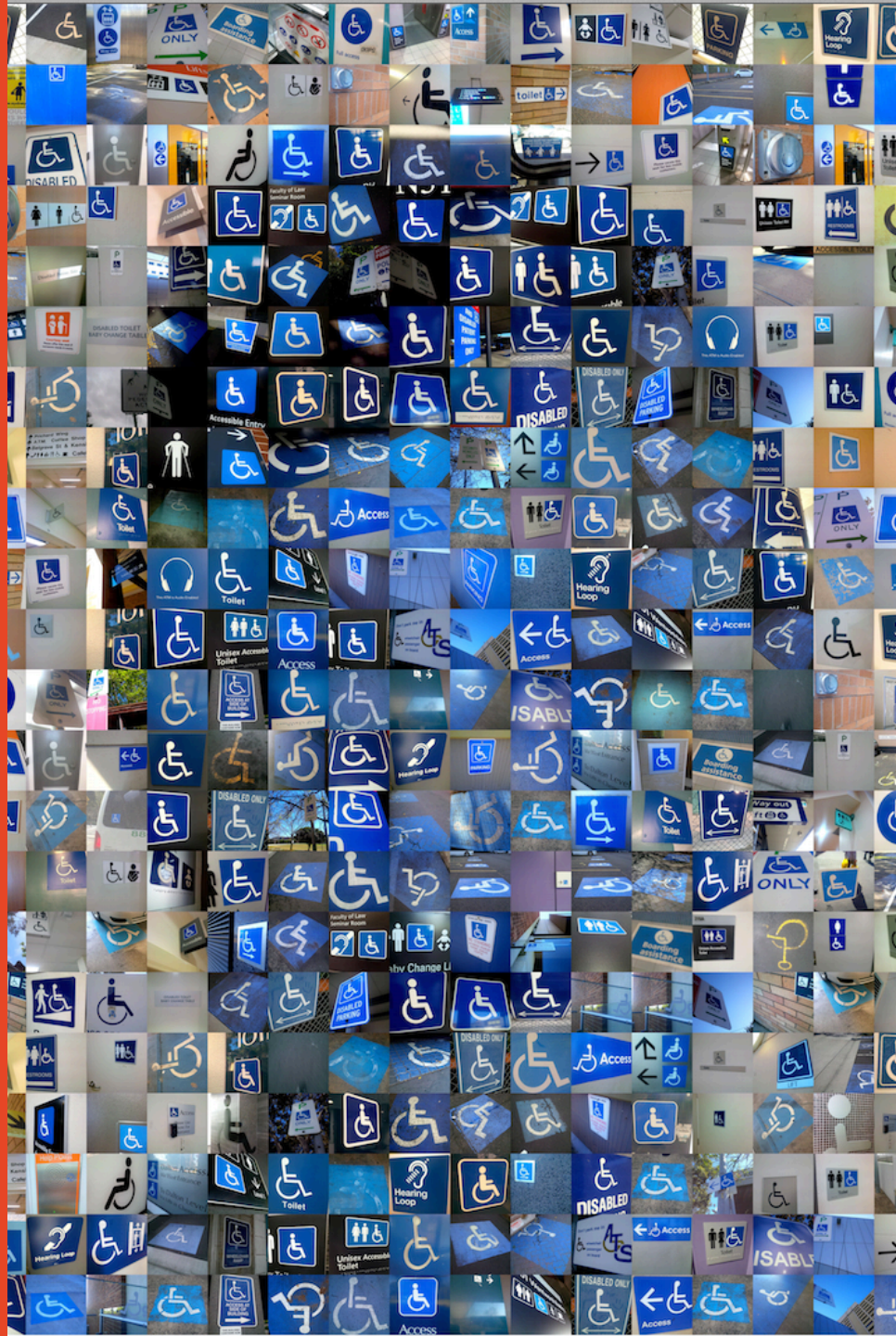
You may not think of yourself as having a ‘disability’ but the definition under the **Disability Discrimination Act (1992)** is broad and includes temporary or chronic medical conditions, physical or sensory disabilities, psychological conditions and learning disabilities.

The types of disabilities we see include:
Anxiety // Arthritis // Asthma // Autism // ADHD
Bipolar disorder // Broken bones // Cancer
Cerebral palsy // Chronic fatigue syndrome
Crohn’s disease // Cystic fibrosis // Depression
Diabetes // Dyslexia // Epilepsy // Hearing impairment // Learning disability // Mobility impairment // Multiple sclerosis // Post-traumatic stress // Schizophrenia // Vision impairment
and much more.

Students needing assistance must register with Disability Services. It is advisable to do this as early as possible. Please contact us or review our website to find out more.

THE UNIVERSITY OF SYDNEY

Disability Services Office
sydney.edu.au/disability
02-8627-8422



Other support

- Learning support
  - http://sydney.edu.au/study/academic-support/learning-support.html
- International students
  - http://sydney.edu.au/study/academic-support/support-for-international-students.html
- Aboriginal and Torres Strait Islanders
  - http://sydney.edu.au/study/academic-support/aboriginal-and-torres-strait-islander-support.html
- Student organization (can represent you in academic appeals etc)
  - http://srcusyd.net.au/ or http://www.supra.net.au/
- Please make contact, and get help
- You are not required to tell anyone else about this
- If you are willing to inform the unit coordinator, they may be able to work with other support to reduce the impact on this unit
  - eg provide advice on which tasks are most significant

WHS INDUCTION

School of Information Technologies

THE UNIVERSITY OF SYDNEY

General Housekeeping – Use of Labs

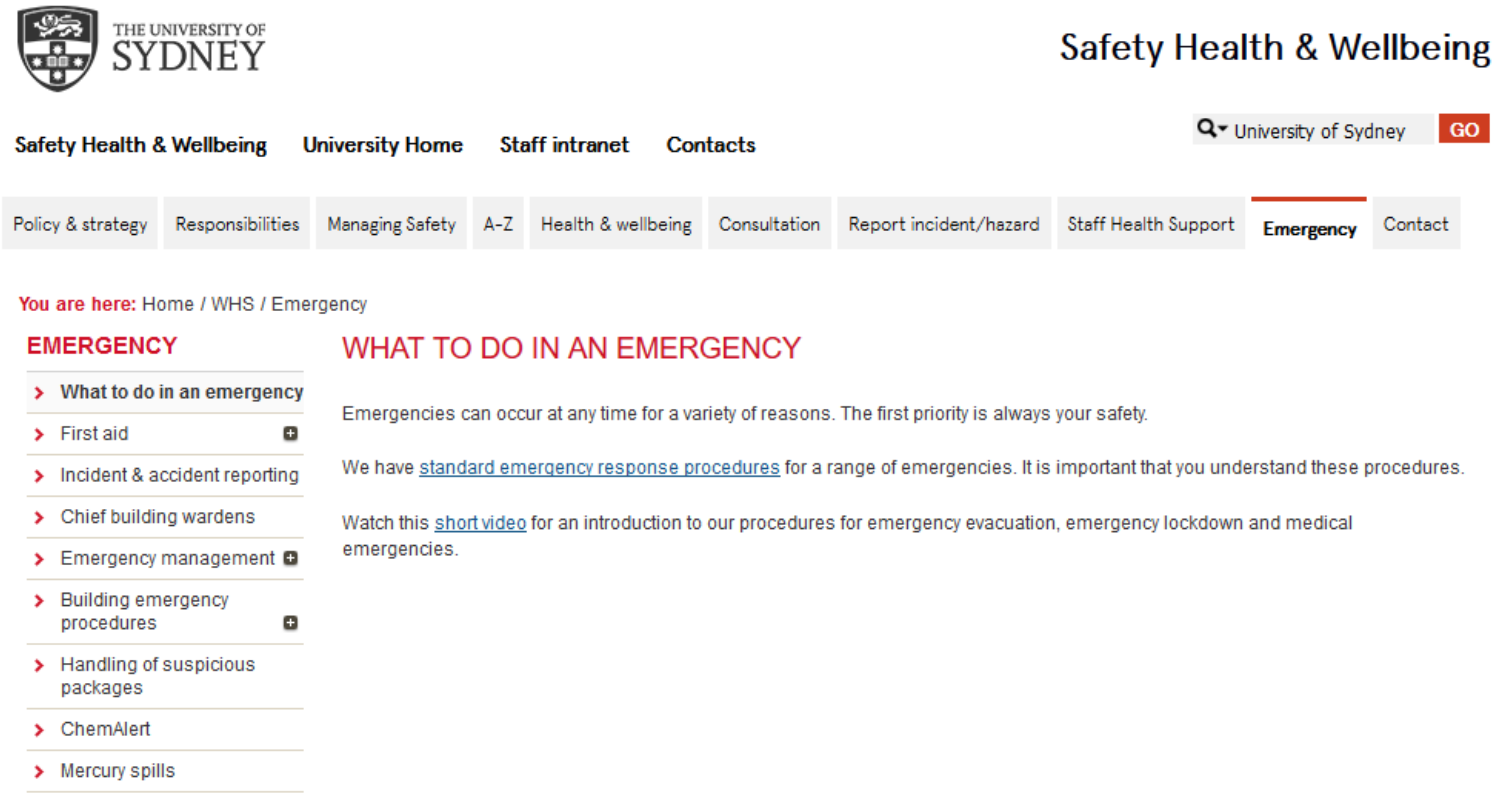
- Keep work area clean and orderly
- Remove trip hazards around desk area
- No food and drink near machines
- No smoking permitted within University buildings
- Do not unplug or move equipment without permission





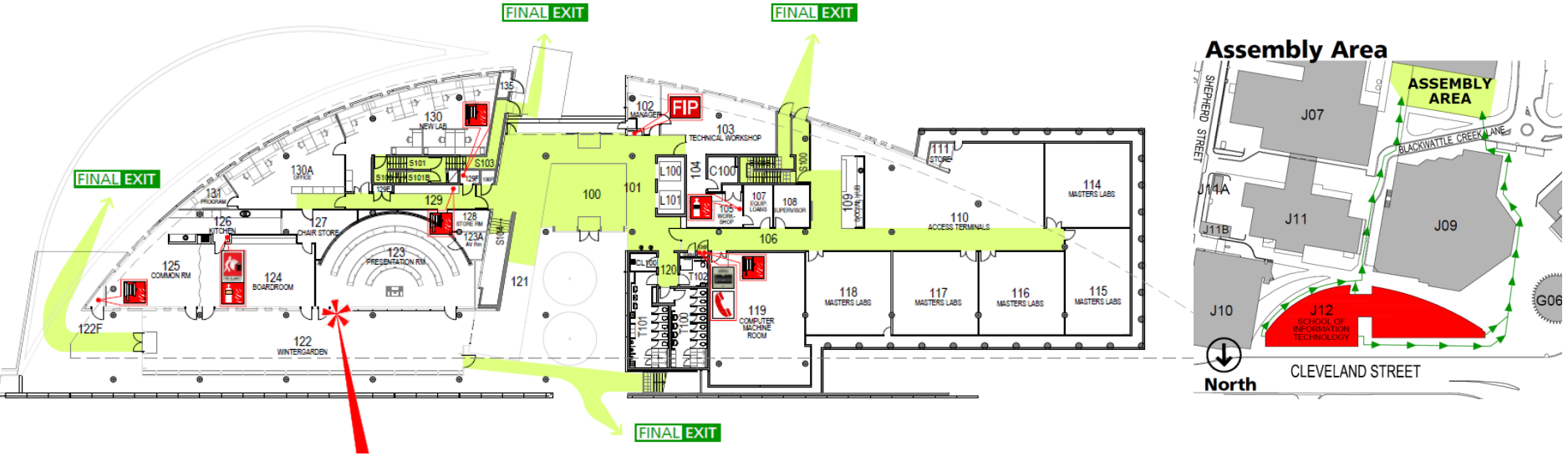
EMERGENCIES – Be prepared

www.sydney.edu.au/whs/emergency



EMERGENCIES

WHERE IS YOUR CLOSEST SAFE EXIT ?



EMERGENCIES

Evacuation Procedures

ALARMS

BEEP... BEEP... Prepare to evacuate

1. Check for any signs of immediate danger.
2. Shut Down equipment / processes.
3. Collect any nearby personal items.

WHOOP... WHOOP... Evacuate the building

1. Follow the EXIT exit signs.
2. Escort visitors & those who require assistance.
3. DO NOT use lifts.
4. Proceed to the assembly area.

EMERGENCY RESPONSE

1. Warn anyone in immediate danger.
2. Fight the fire or contain the emergency, if safe & trained to do so.

If necessary...

3. Close the door, if safe to do so.
4. Activate the "Break Glass" Alarm  or 
5. Evacuate via your closest safe exit. EXIT 
6. Report the emergency to 0-000 & 9351-3333

MEDICAL EMERGENCY

- If a person is seriously ill/injured:
  1. **call an ambulance 0-000**
  2. **notify the closest Nominated First Aid Officer**

     If unconscious– send for Automated External Defibrillator (AED)

     AED **locations**.

     NEAREST to CS Building (J12)
     - Electrical Engineering Building, L2 (ground) near lifts
     - Seymour Centre, left of box office
     - Carried by all Security Patrol vehicles
  3. **call Security -** 9351-3333
  4. **Facilitate the arrival of Ambulance Staff (via Security)**

Nearest Medical Facility
University Health Service in Level 3, Wentworth Building

First Aid kit – SIT Building (J12)
kitchen area adjacent to Lab 110

School of Computer Science Safety Contacts

CHIEF WARDEN
Greg Ryan
Level 1W 103
9351 4360
0411 406 322

FIRST AID OFFICERS

Julia Ashworth
Level 2E Reception
9351 3423

Will Calleja
Level 1W 103
9036 9706
0422 001 964

Katie Yang
Level 2E 237
9351 4918

Orally REPORT all INCIDENTS & HAZARDS to your SUPERVISOR

OR

Undergraduates: to Katie Yang 9351 4918

Coursework Postgraduates: to Cecille Faraizi 9351 6060

CS School Manager: Priyanka Magotra 8627 4295

Emergency procedures

- In the unlikely event of an emergency we may need to evacuate the building
- If we need to evacuate, we will ask you to take your belongings and follow the green exit signs and proceed to the assembly area.
- In some circumstances, we might be asked to remain inside the building for our own safety. We call this a lockdown or shelter-in-place.
- Further information is available at www.sydney.edu.au/emergency

Coronavirus (COVID-19)

- **All staff and students who have cold or flu symptoms should isolate themselves from others**

- If you have a non-infectious condition such as asthma or hayfever please let your teacher and classmates know

- If you are otherwise unwell with cold or flu symptoms please excuse yourself from this class and **we will support you to continue the work remotely**

- Make sure you read the information on **special consideration in the unit outline.**

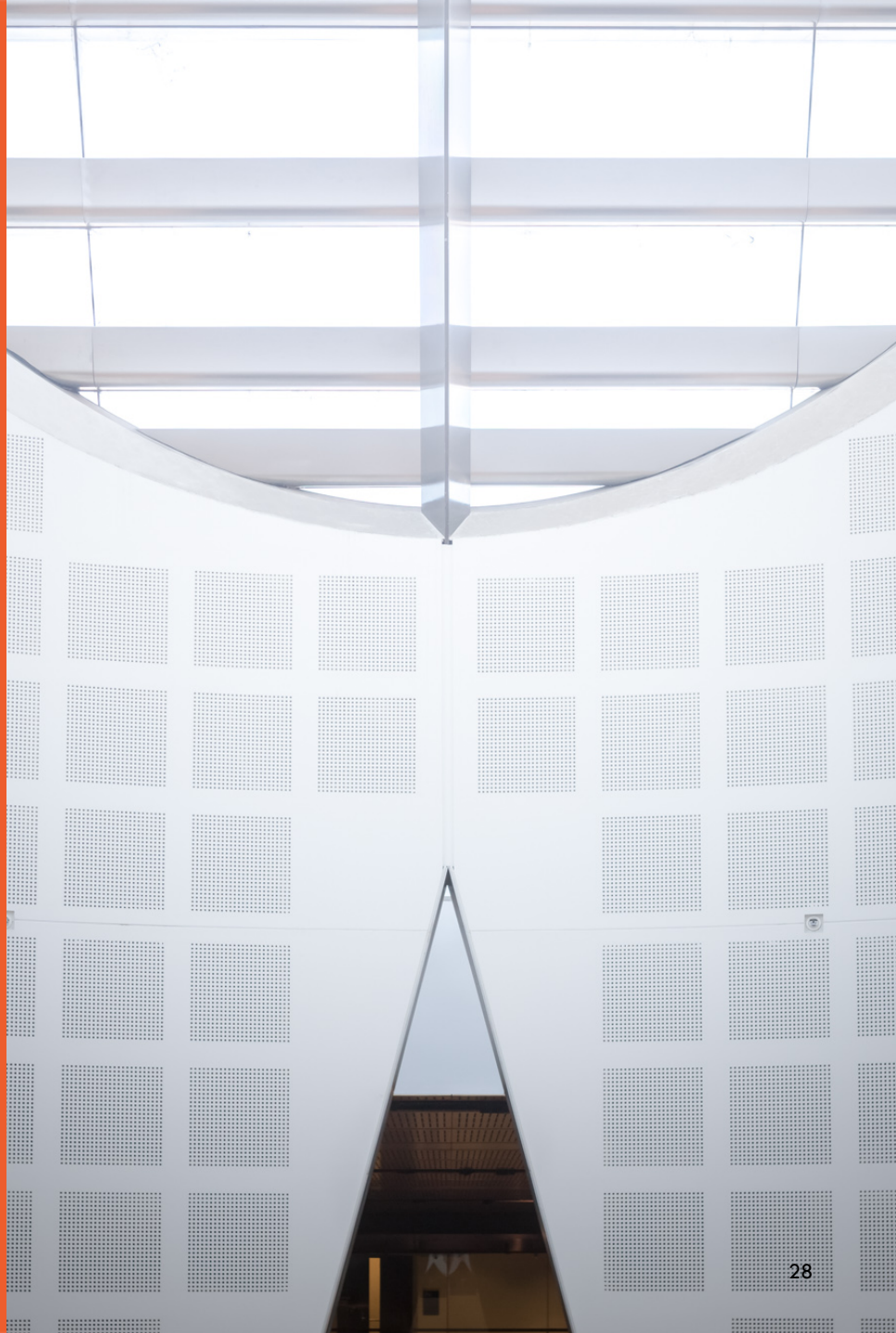
COMP3027: Algorithm Design

Lecture 1b: Introduction

William Umboh

School of Computer Science

THE UNIVERSITY OF SYDNEY



Algorithms are everywhere

- Operating Systems
- Compilers
- Networking
- Computational Biology
- Cryptography

Algorithms are useful

As inputs get bigger, having good algorithms is crucial

THE UNIVERSITY OF SYDNEY

What's in an algorithm?

- Algorithms can have huge impact
- For example:

  A report to the White House from 2010 includes the following.

  - Professor Martin Grotschel
    - A benchmark production planning model solved using linear programming would have taken 82 years to solve in 1988, using the computers and the linear programming algorithms of the day.
    - Fifteen years later, in 2003, this same model could be solved in roughly 1 minute, an improvement by a factor of roughly 43 million!

    [Extreme case, but even the average factor is very high.]

THE UNIVERSITY OF SYDNEY

What's in an algorithm?

- In 2003 there were examples of problems that we can solve 43 million times faster than in 1988
  - This is because of better hardware and better algorithms

What's in an algorithm?

- In 1988
  - Intel 386 and 386SX
    - About 275,000 transistors
    - clock speeds of 16MHz, 20MHz, 25MHz, and 33MHz
  - MSDOS 4.0 and windows 2.0
  - VGA
- In 2003
  - Pentium M
    - About 140 million transistors
    - Up to 2.2 GHz
  - AMD Athlon 64
  - Windows XP

What's in an algorithm?

- In a report to the White House from 2010 includes the following.
  - Professor Martin Grotschel:
    - A benchmark production planning model solved using linear programming would have taken 82 years to solve in 1988, using the computers and the linear programming algorithms of the day.
    - Fifteen years later, in 2003, this same model could be solved in roughly 1 minute, an improvement by a factor of roughly 43 millions

Observation:

- Hardware: 1,000 times improvement
- Algorithms: 43,000 times improvement

Algorithms are fun!

- Designing algorithms requires both creativity and logical thinking
- Many interesting surprises
- Many interesting research questions

Three abstractions

Problem statement:

- 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 (e.g., time and space) it uses

Three abstractions

Problem statement:

- 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 (e.g., time and space) it uses

Three abstractions

Problem statement:

- 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 (e.g., time and space) it uses

Three abstractions

Problem statement:

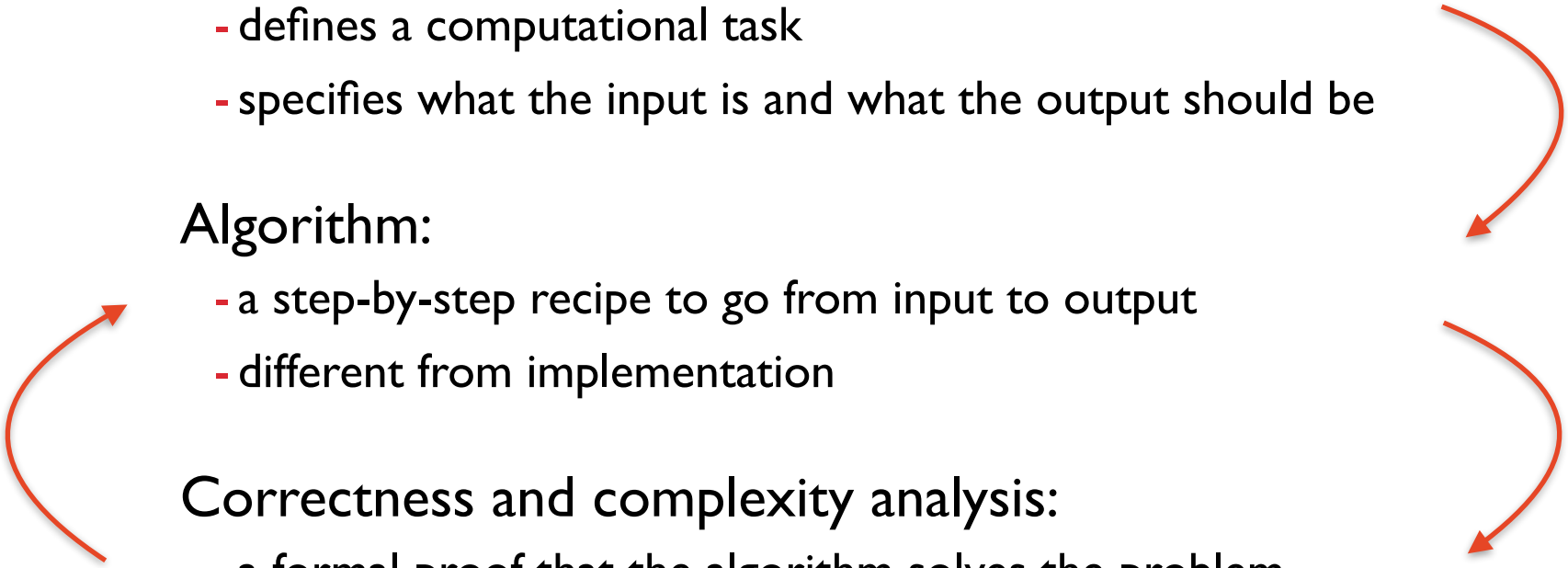
- defines a computational task
- specifies what the input is and what the output should be

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 (e.g., time and space) it uses



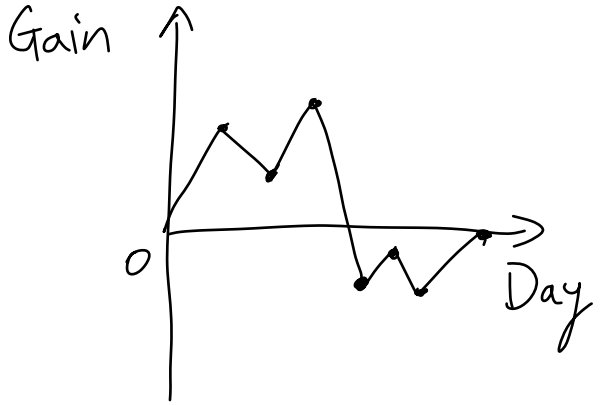
A computational problem

A computational problem

Motivation

- We are a cryptocurrency trading firm and have just developed a fancy quantum neural network algorithm to predict future price fluctuations of Bitcoin
- Given these predictions, we want to find the best investment time window

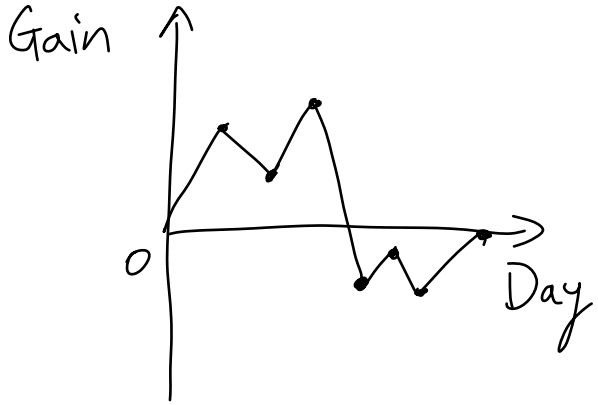
A computational problem



Motivation

- We are a cryptocurrency trading firm and have just developed a fancy quantum neural network algorithm to predict future price fluctuations of Bitcoin
- Given these predictions, we want to find the best investment time window

A computational problem



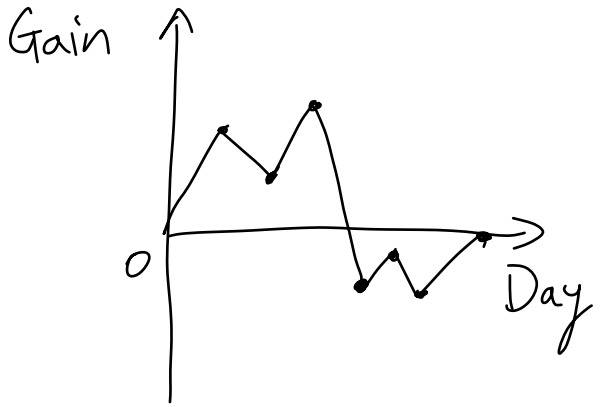
Motivation

- We are a cryptocurrency trading firm and have just developed a fancy quantum neural network algorithm to predict future price fluctuations of Bitcoin
- Given these predictions, we want to find the best investment time window

Input:

- An array with n integer values A[0], A[1],... , A[n-1] (can be +ve or -ve)

A computational problem



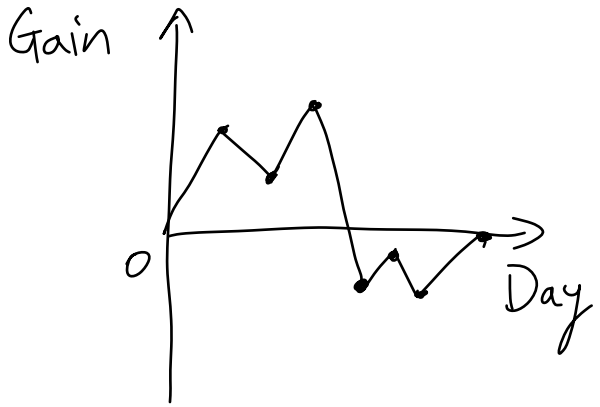
Motivation

- We are a cryptocurrency trading firm and have just developed a fancy quantum neural network algorithm to predict future price fluctuations of Bitcoin
- Given these predictions, we want to find the best investment time window

Input:

- An array with n integer values A[0], A[1],... , A[n-1] (can be +ve or -ve)

A computational problem



Motivation

- We are a cryptocurrency trading firm and have just developed a fancy quantum neural network algorithm to predict future price fluctuations of Bitcoin
- Given these predictions, we want to find the best investment time window

Input:

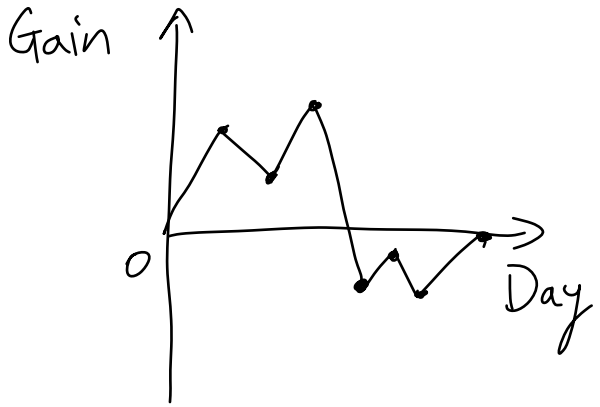
- An array with n integer values A[0], A[1], ... , A[n-1] (can be +ve or -ve)

Task:

- Find indices $0 \le i \le j < n$ maximizing

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

A computational problem

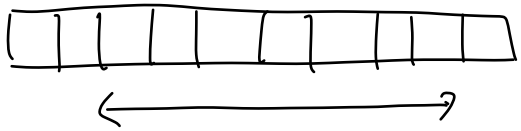


Motivation

- We are a cryptocurrency trading firm and have just developed a fancy quantum neural network algorithm to predict future price fluctuations of Bitcoin
- Given these predictions, we want to find the best investment time window

Input:

- An array with n integer values A[0], A[1], ... , A[n-1] (can be +ve or -ve)



Task:

- Find indices $0 \le i \le j < n$ maximizing

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

Naive algorithm

```
def naive(A):

  def evaluate(a,b)
     return A[a] + ... + A[b]

  n = size of A
  answer = (0,0)
  for i = 0 to n-1
    for j = i to n-1
      if evaluate(i,j) > evaluate(answer[0],answer[1])
         answer = (i,j)
 return answer
```

Naive algorithm

```
def naive(A):

  def evaluate(a,b)
     return A[a] + ... + A[b]

  n = size of A
  answer = (0,0)
  for i = 0 to n-1
    for j = i to n-1
      if evaluate(i,j) > evaluate(answer[0],answer[1])
         answer = (i,j)
 return answer
```

Questions:

- how efficient is this algorithm?
- is this the best algorithm for this task?

Efficiency

Efficiency

Def. 1: An algorithm is *efficient* if it runs quickly on real input instances

Efficiency

Def. 1: An algorithm is *efficient* if it runs quickly on real input instances

Not a good definition because it depends on

- how big our instances are
- how restricted/general our instance are
- implementation details
- hardware it runs on

Efficiency

Def. 1: An algorithm is *efficient* if it runs quickly on real input instances

Not a good definition because it depends on

- how big our instances are
- how restricted/general our instance are
- implementation details
- hardware it runs on

A better definition would be implementation independent:

- count number of "steps"
- bound the algorithm's worst-case performance

Efficiency

Efficiency

Def. 2: An algorithm is *efficient* if it achieves (analytically) qualitatively better worst-case performance than a brute-force approach.

Efficiency

Def. 2: An algorithm is *efficient* if it achieves (analytically) qualitatively better worst-case performance than a brute-force approach.

This is better but still has some issues:

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

Efficiency

Efficiency

Def. 3: An algorithm is *efficient* if it runs in polynomial time; that is, on an instance of size n, it performs $p(n)$ steps for some polynomial

$p(x) = a_d x^d + a_{d-1} x^{d-1} + \cdots + a_0$

Efficiency

Def. 3: An algorithm is *efficient* if it runs in polynomial time; that is, on an instance of size n, it performs $p(n)$ steps for some polynomial $p(x) = a_d x^d + a_{d-1} x^{d-1} + \cdots + a_0$

Notice that if we double the size of the input, then the running time would roughly increase by a factor of 2^d.

Efficiency

Def. 3: An algorithm is *efficient* if it runs in polynomial time; that is, on an instance of size n, it performs $p(n)$ steps for some polynomial $p(x) = a_d x^d + a_{d-1} x^{d-1} + \cdots + a_0$

Notice that if we double the size of the input, then the running time would roughly increase by a factor of 2^d.

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

Asymptotic growth analysis

Note: c is a constant independent of n.

Asymptotic growth analysis

Let $T(n)$ be the worst-case number of steps of our algorithm on an instance of "size" n. We say that $T(n) = O(f(n))$ if

there exist n_0 and $c > 0$ such that $T(n) \leq c\, f(n)$ for all $n > n_0$

Note: c is a constant independent of n.

Asymptotic growth analysis

Let $T(n)$ be the worst-case number of steps of our algorithm on an instance of "size" n. We say that $T(n) = O(f(n))$ if

there exist n_0 and $c > 0$ such that $T(n) \leq c\, f(n)$ for all $n > n_0$

Also, we say that $T(n) = \Omega(f(n))$ if

there exist n_0 and $c > 0$ such that $T(n) > c\, f(n)$ for all $n > n_0$

Note: c is a constant independent of n.

Asymptotic growth analysis

Let $T(n)$ be the worst-case number of steps of our algorithm on an instance of "size" n. We say that $T(n) = O(f(n))$ if

there exist n_0 and $c > 0$ such that $T(n) \leq c\, f(n)$ for all $n > n_0$

Also, we say that $T(n) = \Omega(f(n))$ if

there exist n_0 and $c > 0$ such that $T(n) > c\, f(n)$ for all $n > n_0$

Finally, we say that $T(n) = \Theta(f(n))$ if

$T(n) = O(f(n))$ and $T(n) = \Omega(f(n))$

Note: c is a constant independent of n.

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 $f = O(h)$ and $g = O(h)$, then $f + g = O(h)$
- If $f = \Omega(h)$ and $g = \Omega(h)$, then $f + g = \Omega(h)$
- If $f = \Theta(h)$ and $g = \Theta(h)$, then $f + g = \Theta(h)$
- BEWARE: If we have n functions $f_1, f_2, \ldots f_n = O(h)$, then their sum is not $O(h)$ but $O(nh)$.

Properties of asymptotic growth

Let $T(n) = a_d n^d + \cdots + a_0$ be a poly. with $a_d > 0$, then $T(n) = \Theta(n^d)$

Let $T(n) = \log_a n$ for constant $a > 1$, then $T(n) = \Theta(\log n)$

For every $b > 1$ and $d > 0$, we have $n^d = O(b^n)$

The reason we use asymptotic analysis is that allows us to ignore unimportant details and focus on what's important, on what dominates the running time of an algorithm.

Survey of common running times

Let n be the size of the input, and let $T(n)$ be the running time of our algorithm.

We say T(n) is…	if…
logarithmic	$T(n) = \Theta(\log n)$
linear	$T(n) = \Theta(n)$
“almost” linear	$T(n) = \Theta(n \log n)$
quadratic	$T(n) = \Theta(n^2)$
cubic	$T(n) = \Theta(n^3)$
exponential	$T(n) = \Theta(c^n)$ for some $c > 1$

Comparison of running times

Assume machine can run a million "steps" per second

size	n	n log n	n^2	n^3	2^n	n!
10	<1 s	<1 s	<1 s	<1 s	<1 s	3 s
30	<1 s	<1 s	<1 s	<1 s	17 m	WTL
50	<1 s	<1 s	<1 s	<1 s	35 y	WTL
100	<1 s	<1 s	<1 s	1 s	WTL	WTL
1000	<1 s	<1 s	1 s	15 m	WTL	WTL
10,000	<1 s	<1 s	2 m	11 d	WTL	WTL
100,000	<1 s	1 s	2 h	31 y	WTL	WTL
1,000,000	1 s	10 s	4 d	WTL	WTL	WTL

WTL = way too long

Recap: Asymptotic analysis

Establish the asymptotic number of "steps" our algorithm performs in the worst case

Each "step" represents constant amount of real computation

Asymptotic analysis provides the right level of detail

Efficiency = polynomial running time

Keep in mind hidden constants inside your O-notation

Naive algorithm

```
def naive(A):

  def evaluate(a,b)
     return A[a] + ... + A[b]

  n = size of A
  answer = (0,0)
  for i = 0 to n-1
      for j = i to n-1
      if evaluate(i,j) > evaluate(answer[0],answer[1])
        answer = (i,j)
   return answer
```

Naive algorithm

```
def naive(A):
    def evaluate(a,b)
        return A[a] + ... + A[b]
    n = size of A
    answer = (0,0)
    for i = 0 to n-1
        for j = i to n-1
        if evaluate(i,j) > evaluate(answer[0],answer[1])
            answer = (i,j)
     return answer
```

Θ(n) time

Naive algorithm

```
def naive(A):
    def evaluate(a,b)
        return A[a] + ... + A[b]

    n = size of A
    answer = (0,0)
    for i = 0 to n-1
        for j = i to n-1
        if evaluate(i,j) > evaluate(answer[0],answer[1])
            answer = (i,j)
    return answer
```

$\Theta(n)$ time

$\Theta(n^2)$ calls to evaluate

Naive algorithm

```
def naive(A):
    def evaluate(a,b)
        return A[a] + ... + A[b]

    n = size of A
    answer = (0,0)
    for i = 0 to n-1
        for j = i to n-1
        if evaluate(i,j) > evaluate(answer[0],answer[1])
            answer = (i,j)
    return answer
```

$\Theta(n)$ time

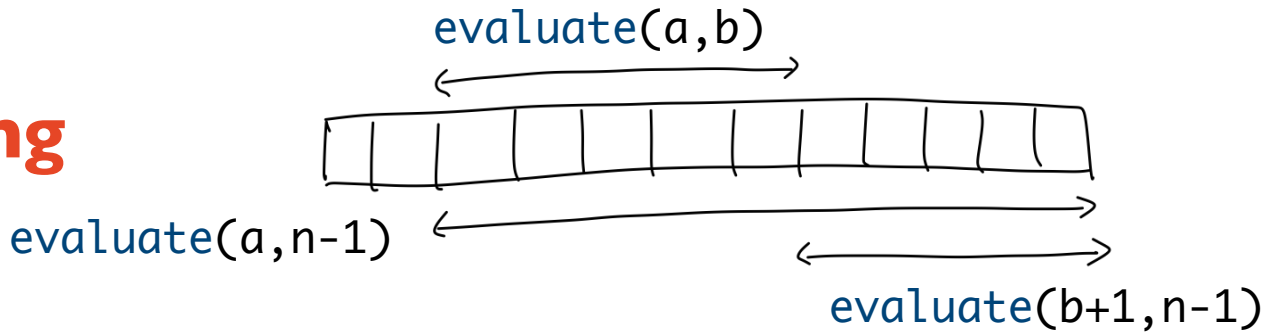
$\Theta(n^2)$ calls to evaluate

Obs.

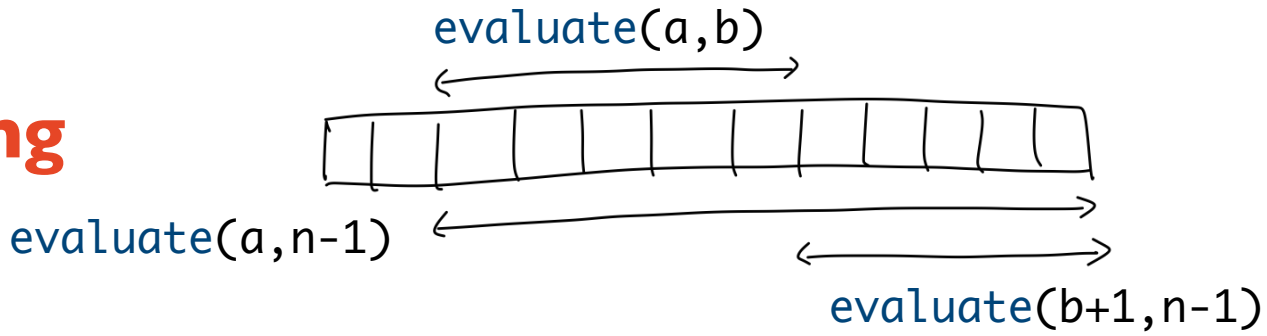
`naive` runs in $\Theta(n^3)$ time

Pre-processing

Pre-processing

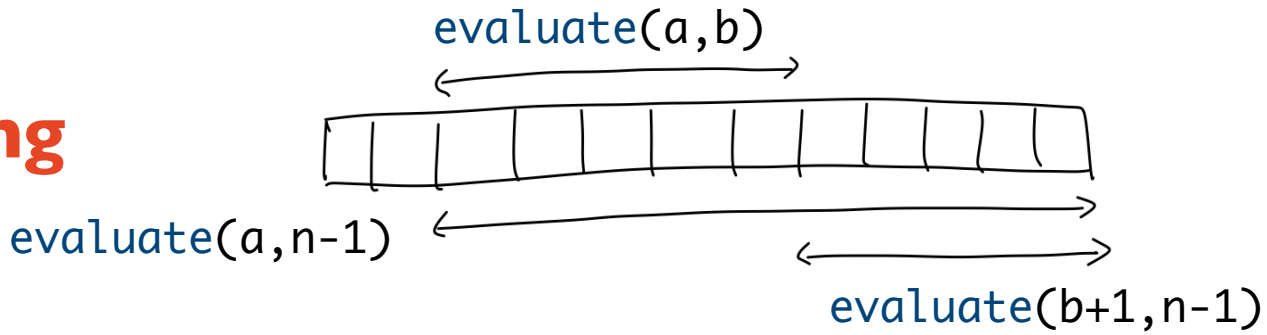


Pre-processing



Speed up "evaluate"
subroutine by
pre-computing for all i:
B[i] = A[i] + ... + A[n-1]

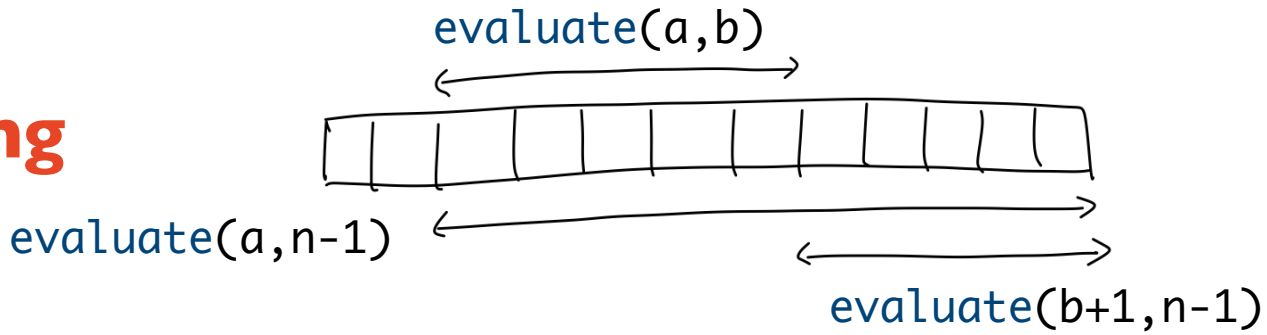
Pre-processing



Speed up "evaluate"
subroutine by
pre-computing for all i:
B[i] = A[i] + ... + A[n-1]

The rest is as before

Pre-processing



Speed up "evaluate" subroutine by pre-computing for all i:

B[i] = A[i] + ... + A[n-1]

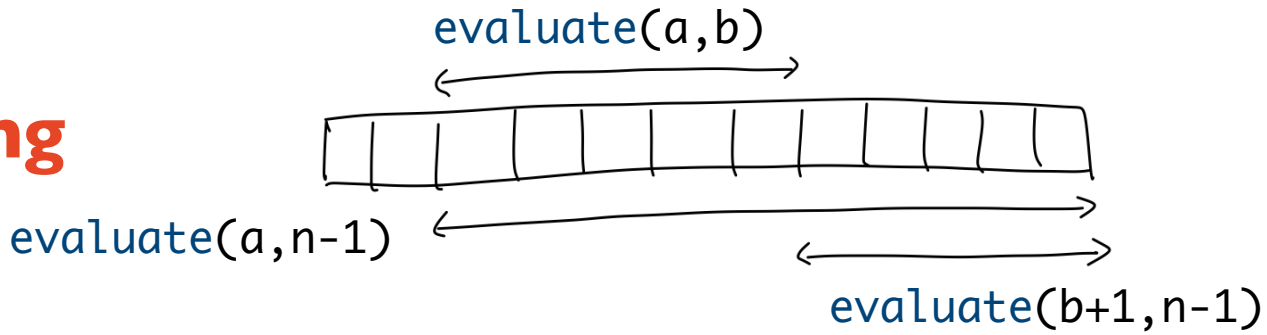
The rest is as before

```
def preprocessing(A):

  def evaluate(a,b)
     return B[a] - B[b+1]

  n = size of A
  B = array of size n+1
  B[n] = 0
  for i in 0 to n-1
    B[i] = A[i] + ... A[n-1]
  answer = (0,0)
  for i = 0 to n-1
      for j = i to n-1
       if evaluate(i,j) >
evaluate(answer[0],answer[1])
         answer = (i,j)
   return answer
```

Pre-processing



Speed up "evaluate" subroutine by pre-computing for all i:

B[i] = A[i] + ... + A[n-1]

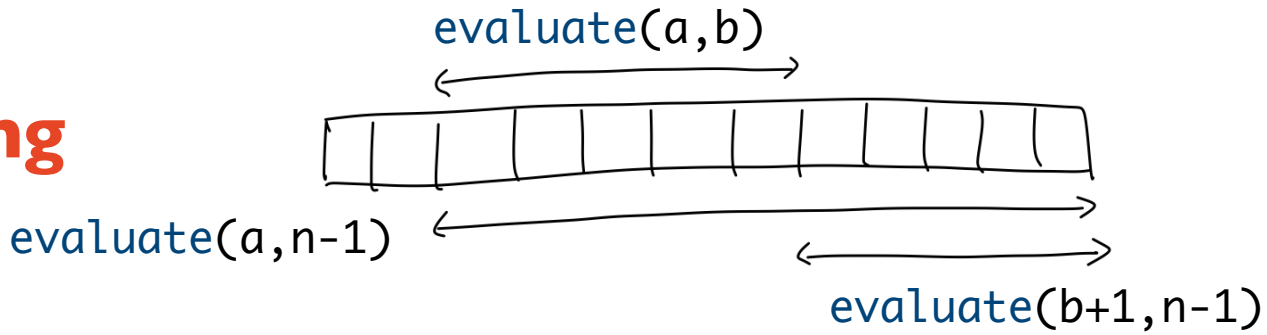
The rest is as before

```
def preprocessing(A):

  def evaluate(a,b)
     return B[a] - B[b+1]

  n = size of A
  B = array of size n+1
  B[n] = 0
  for i in 0 to n-1
    B[i] = A[i] + ... A[n-1]
  answer = (0,0)
  for i = 0 to n-1
      for j = i to n-1
       if evaluate(i,j) >
evaluate(answer[0],answer[1])
         answer = (i,j)
   return answer
```

Pre-processing



Speed up "evaluate" subroutine by pre-computing for all i:

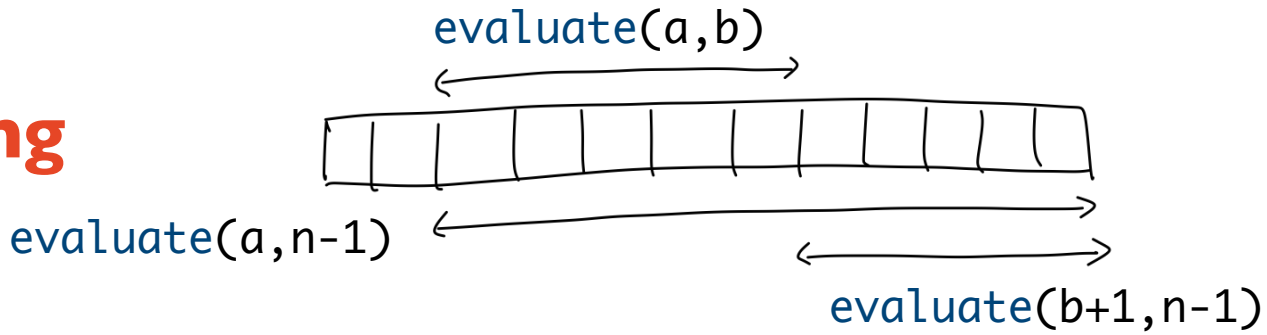
B[i] = A[i] + … + A[n-1]

The rest is as before

```
def preprocessing(A):
    def evaluate(a,b)
        return B[a] - B[b+1]
    n = size of A
    B = array of size n+1
    B[n] = 0
    for i in 0 to n-1
      B[i] = A[i] + ... A[n-1]
    answer = (0,0)
    for i = 0 to n-1
        for j = i to n-1
        if evaluate(i,j) >
evaluate(answer[0],answer[1])
            answer = (i,j)
    return answer
```

Θ(1) time

Pre-processing



Speed up "evaluate" subroutine by pre-computing for all i:

$B[i] = A[i] + \ldots + A[n-1]$

The rest is as before

```
def preprocessing(A):

    def evaluate(a,b)
        return B[a] - B[b+1]

    n = size of A
    B = array of size n+1
    B[n] = 0
    for i in 0 to n-1
      B[i] = A[i] + ... A[n-1]
    answer = (0,0)
    for i = 0 to n-1
        for j = i to n-1
        if evaluate(i,j) >
evaluate(answer[0],answer[1])
            answer = (i,j)
    return answer
```

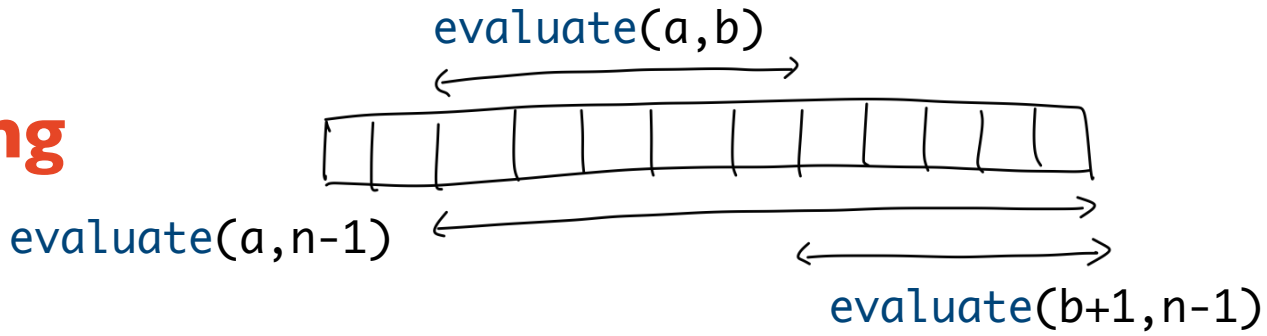
Θ(1) time

Θ(n^2) time

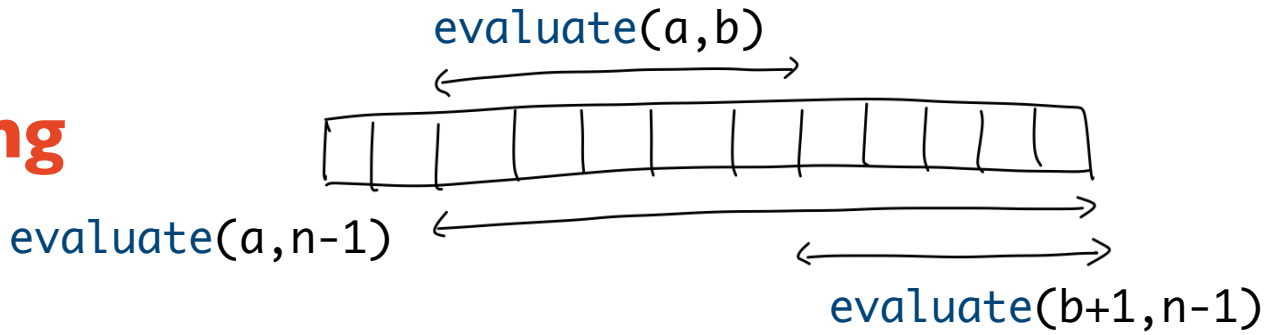
Θ(n^2) time

Pre-processing

Pre-processing



Pre-processing



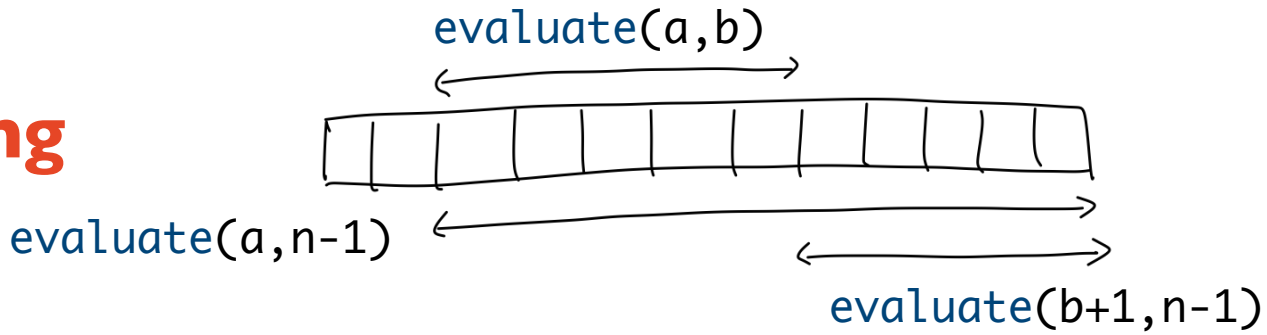
Speed up "evaluate"

subroutine by

pre-computing for all i:

B[i] = A[i] + ... + A[n-1]

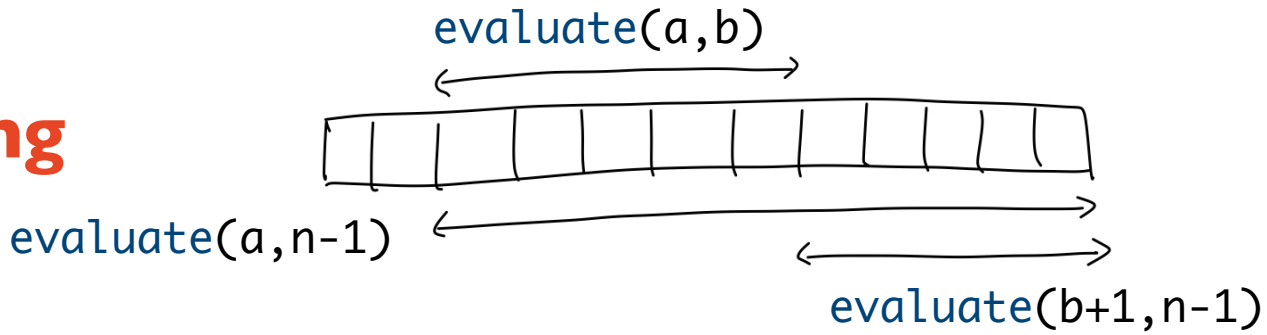
Pre-processing



Speed up "evaluate"
subroutine by
pre-computing for all i:
B[i] = A[i] + ... + A[n-1]

The rest is as before

Pre-processing



Speed up "evaluate" subroutine by pre-computing for all i:

B[i] = A[i] + ... + A[n-1]

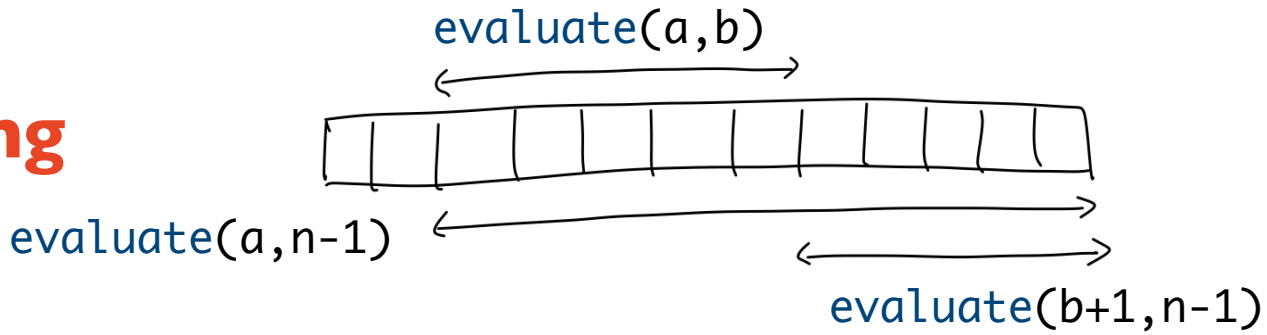
The rest is as before

```
def preprocessing(A):

  def evaluate(a,b)
     return B[a] - B[b+1]

  n = size of A
  B = array of size n+1
  B[n] = 0
  for i in 0 to n-1
    B[i] = A[i] + ... A[n-1]
  ⋮
```

Pre-processing



Speed up "evaluate" subroutine by pre-computing for all i:

B[i] = A[i] + ... + A[n-1]

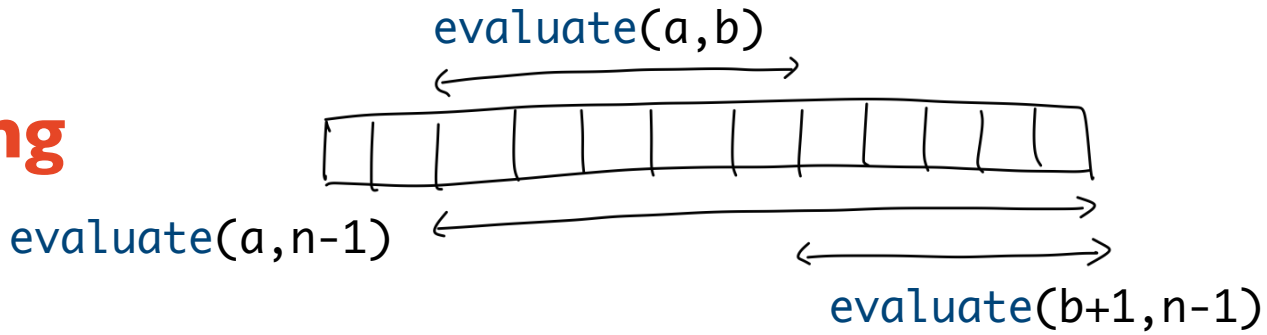
The rest is as before

```
def preprocessing(A):

  def evaluate(a,b)
     return B[a] - B[b+1]

  n = size of A
  B = array of size n+1
  B[n] = 0
  for i in 0 to n-1
    B[i] = A[i] + ... A[n-1]
  ⋮
```

Pre-processing



Speed up "evaluate" subroutine by pre-computing for all i:

B[i] = A[i] + ... + A[n-1]

The rest is as before

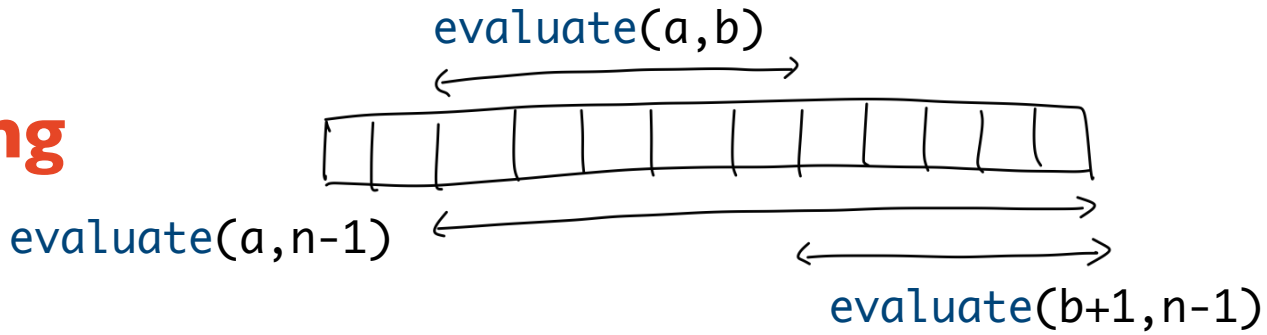
```
def preprocessing(A):

    def evaluate(a,b)
        return B[a] - B[b+1]

    n = size of A
    B = array of size n+1
    B[n] = 0
    for i in 0 to n-1
      B[i] = A[i] + ... A[n-1]
    ⋮
```

Θ(1) time

Pre-processing



Speed up "evaluate" subroutine by pre-computing for all i:

B[i] = A[i] + ... + A[n-1]

The rest is as before

```
def preprocessing(A):
    def evaluate(a,b)
        return B[a] - B[b+1]
    n = size of A
    B = array of size n+1
    B[n] = 0
    for i in 0 to n-1
      B[i] = A[i] + ... A[n-1]
    ⋮
```

Θ(1) time

Obs.

preprocessing runs in $\Theta(n^2)$ time

Dynamic Programming

Imagine trying to find the best window ending at a fixed index j:

$$OPT[j] = \max_{i \leq j} B[i] - B[j]$$

But we can also express $OPT[j]$ recursively in a way that allows us to compute, in $O(n)$ time, $OPT[j]$ for all j

Finally, in $O(n)$ time, find j maximizing $OPT[j]$

Dynamic Programming

Imagine trying to find the best window ending at a fixed index j:

$$OPT[j] = \max_{i \leq j} B[i] - B[j]$$

But we can also express $OPT[j]$ recursively in a way that allows us to compute, in $O(n)$ time, $OPT[j]$ for all j

Finally, in $O(n)$ time, find j maximizing $OPT[j]$

Obs.

There is an $\Theta(n)$ time algorithm for finding the optimal investment window

Recap: Algorithm analysis

`naive` runs in $\Theta(n^3)$ time

`preprocessing` runs in $\Theta(n^2)$ time

With a bit of ingenuity we can solve the problem in $\Theta(n)$ time

Why we separate problem, algorithm, and analysis?

- somebody can design a better algorithm to solves a given problem
- somebody can give a tighter analysis of an old algorithm

This week

Tutorial Sheet 1:

- Posted tonight
- Review of asymptotic analysis and graphs
- Make sure you work on it before the tutorial

Quiz 0

- 15 minutes long
- It won't count as assessment. It's just to review your knowledge of graphs
- Relevant materials on graphs uploaded as "Self-Review - Graphs" on Ed