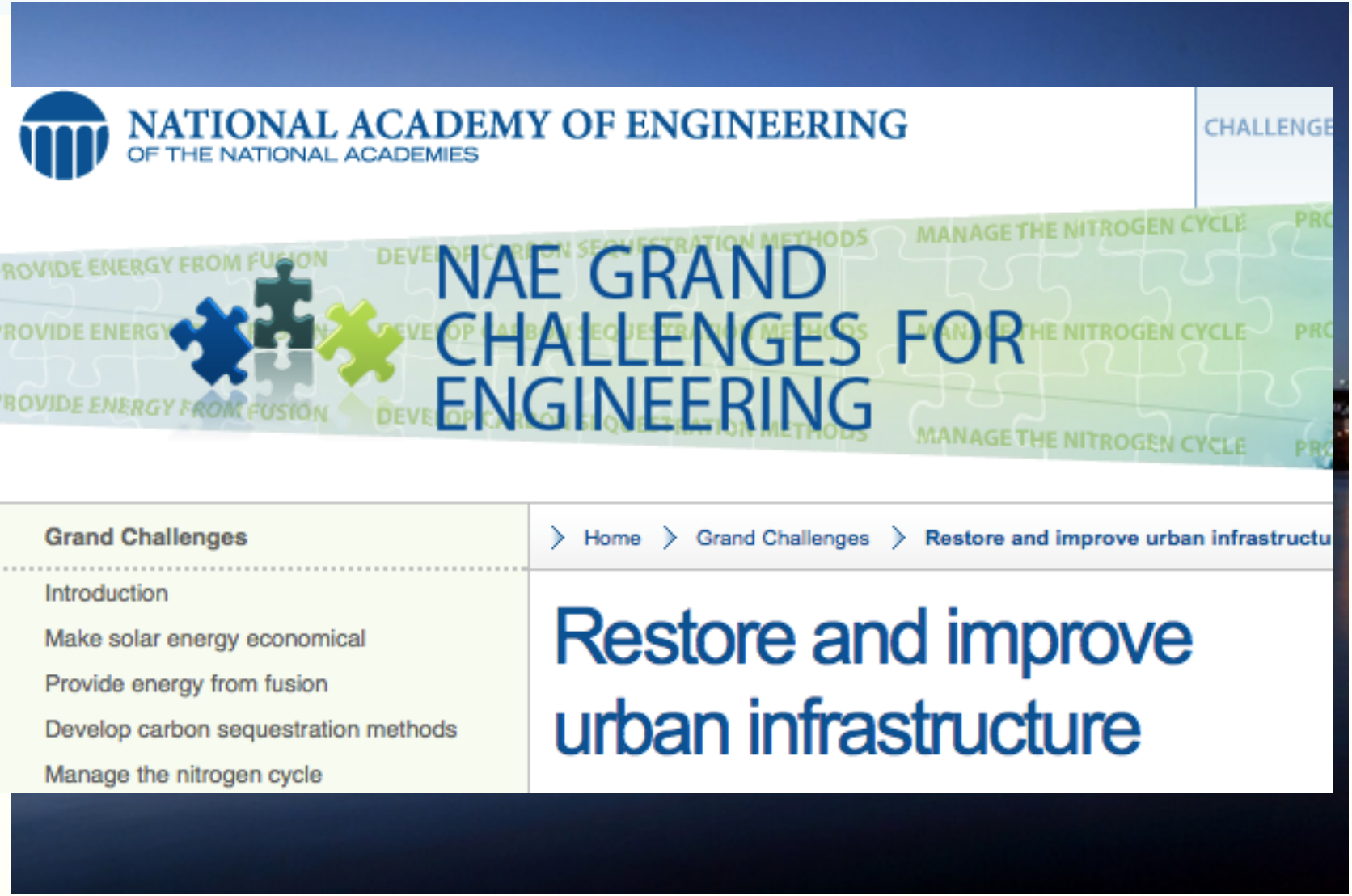
ENGG1811 Computing for Engineers

Week 9A: Algorithms

An engineering challenge



http://www.zdnet.com/sydneys-harbour-bridge-gets-sensor-tech-7000000296/
http://www.engineeringchallenges.org/cms/8996/9136.aspx

Maintenance and mor

is a grand r

Limitations of computation

- Your computer can do almost 100 billion multiplications in one second
- Tiny computers can do far less
  - Need **efficient** or **new** algorithms

- In any case, we want efficient algorithms

This week

- Efficiency in algorithms
- Python programming
  - while
  - More numpy functions
- Computer science concepts
  - Efficient algorithms/computational complexity

Algorithms

- A sequence of instructions for the computation

- Two important criteria
  - Correctness
  - Efficiency

- Example: An algorithm for multiplying 2 integers
  - Correctness means the algorithm returns the correct answer all the time
  - Efficiency: How many multiplications the algorithm can do in a given amount of time

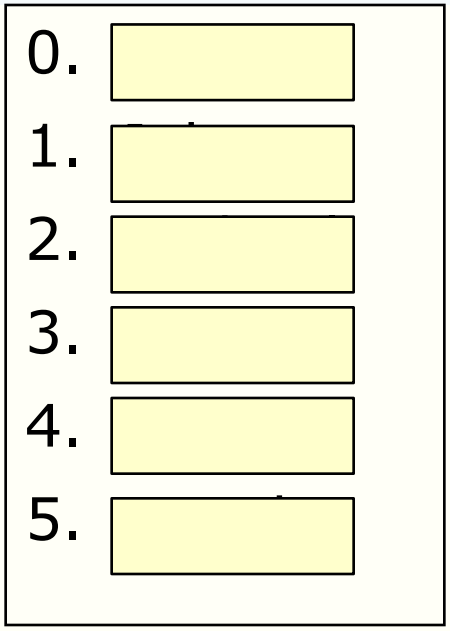
- An efficient algorithm takes a shorter time to arrive at the correct outcome

Challenge: Locate a name in a sorted list of names

- You are given:
  - A list of names arranged in alphabetical order
  - Names are indexed with 0, 1, 2 etc. in their order
- Rules
  - You are not allowed to see the list
  - You can choose an index and query what the name at that index is
- The challenge:
  - Given a name, what is the minimum number of indices that you need to query to locate that name?

0. Abraham
1. Adam
2. Eve
3. Sarah

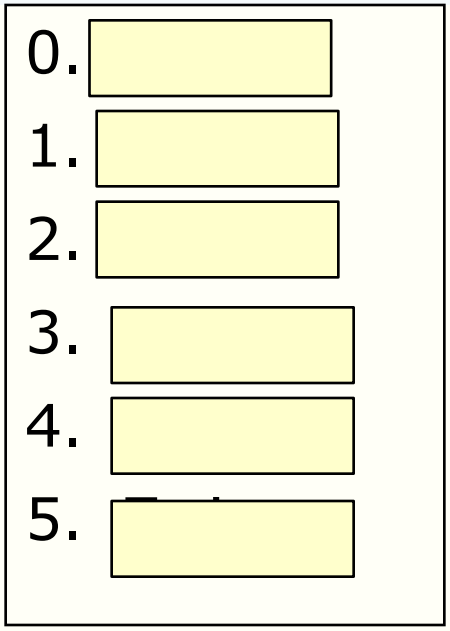
Example: Simple Scan



- Example:
  - There is a list with 6 names on the left
  - Given Peter is one of the names, you want to find which index it is at
- A simple algorithm is to scan the name one by one from the beginning until you have found the name

- Quiz: If the name that you want to locate has the index *k*, how many queries do you need to locate the name using simple scan?
- Which type of loop will you use to implement a simple scan?

Other possible algorithms



- Make a guess of where the name is and then start from there
  - Example: 6 names on the left. The name to locate is Yvonne. Since this name is near the end of the alphabet so we scan from the last name.

- What will be an efficient algorithms?

Towards a general principle

- Consider this game:
  - I think of a living person in this world
  - To win this game, you need to guess who this person is in as few questions as possible
- Consider two sets of questions below, which one will you ask and why?

Question set 1
• Is the person from Zambia? • Is the person from Fiji? • Is the person a current student of UNSW?

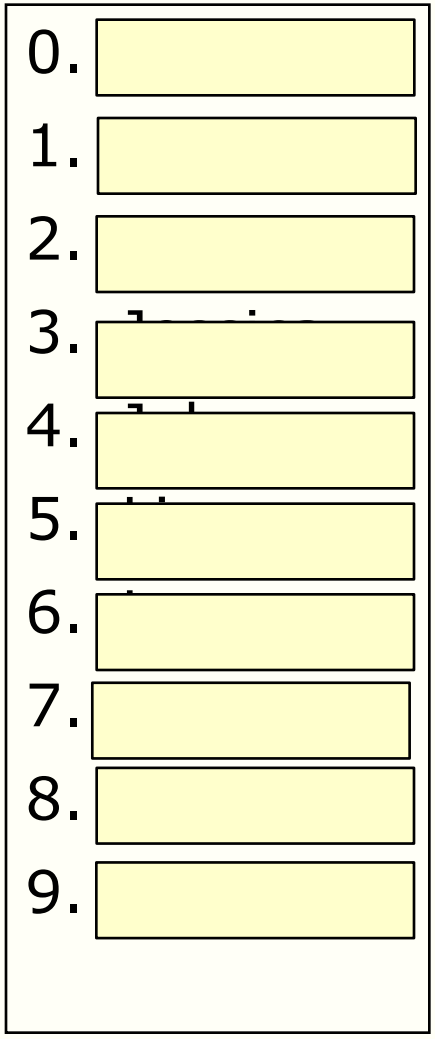
Question set 2
• Is the person a he or she? • Is the person from Asia? • Is the person from South America?

Name search using binary search

- The purpose of the query is to narrow down the possibilities as much as possible
  - Idea: Eliminate half of the possibilities with each query

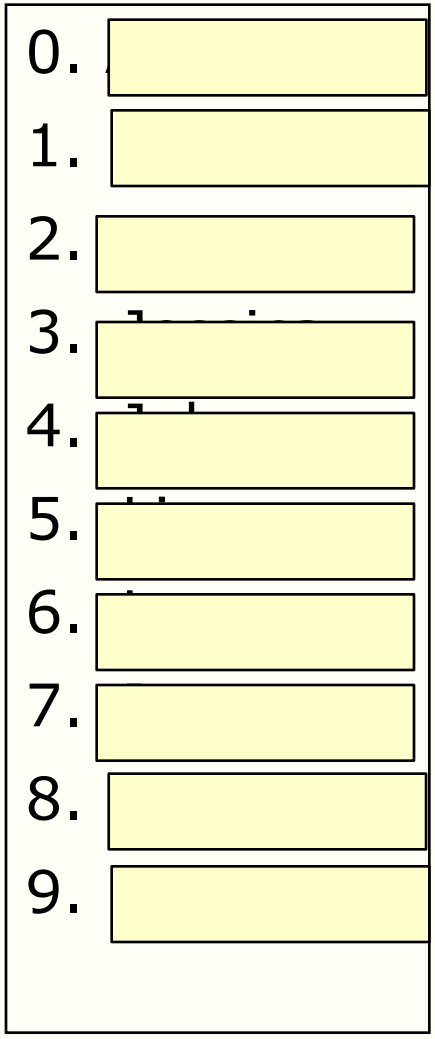
- Binary search:
  - Initialization: Query the name in the middle of the list
  - Eliminate nearly half of the possibilities with each additional query
  - Stop when the name is found

Binary search example: Problem set up

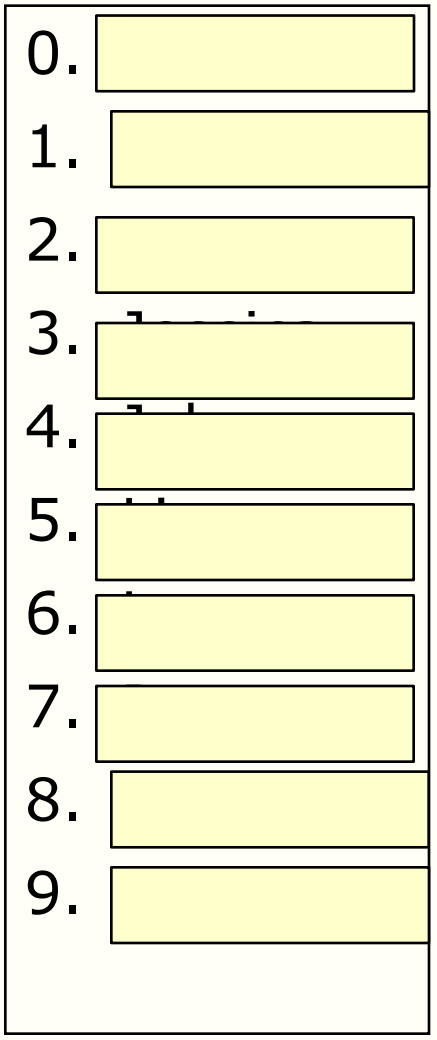


- Given a list of 10 names arranged in alphabetical order
- Aim: Use binary search to locate the name Peter

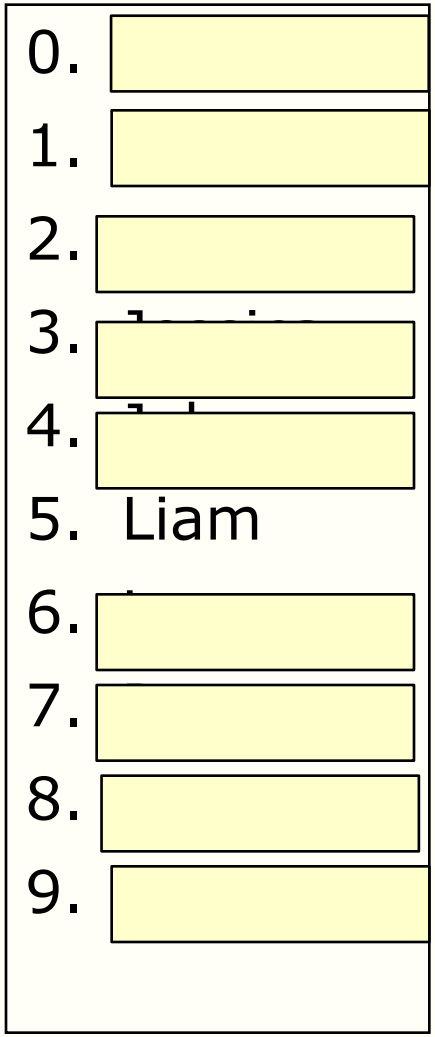
Binary search example (1)



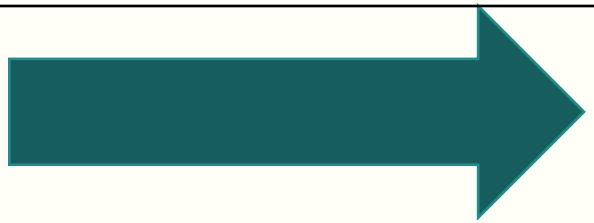
- To eliminate half of the possibilities, pick the name in the middle
- Middle of 0 and 9 = (0+9)/2 = 4.5
- Let us round up
- Initialisation: Query 5

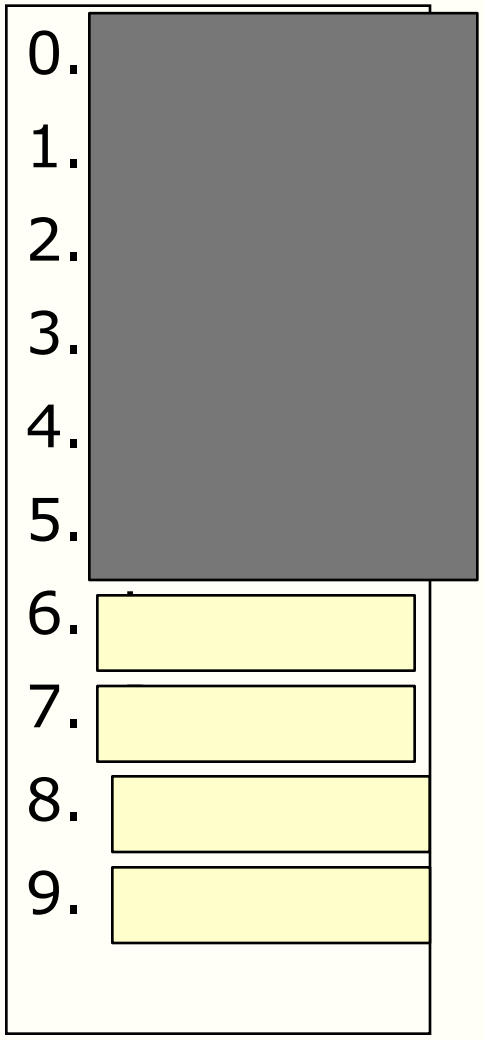


Binary search example (2)

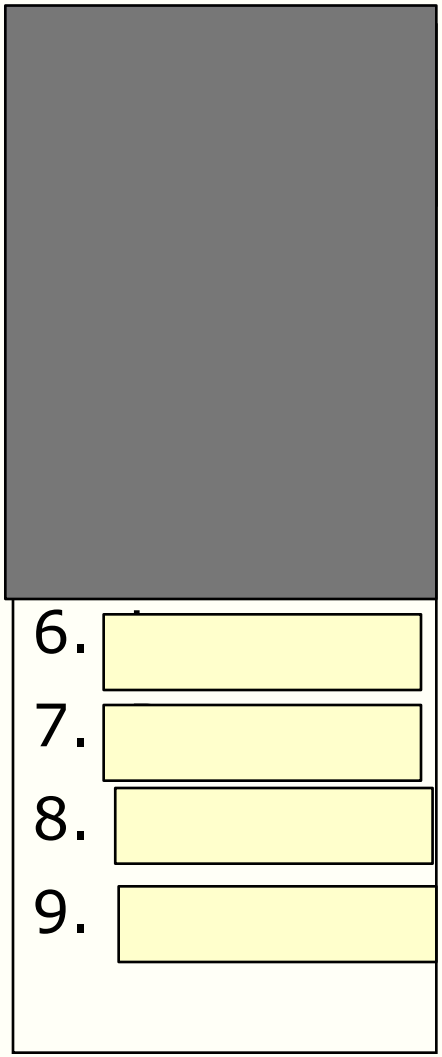


- Where should we look next?
- Can forget indices 0-5

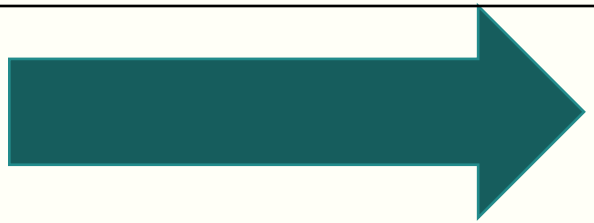


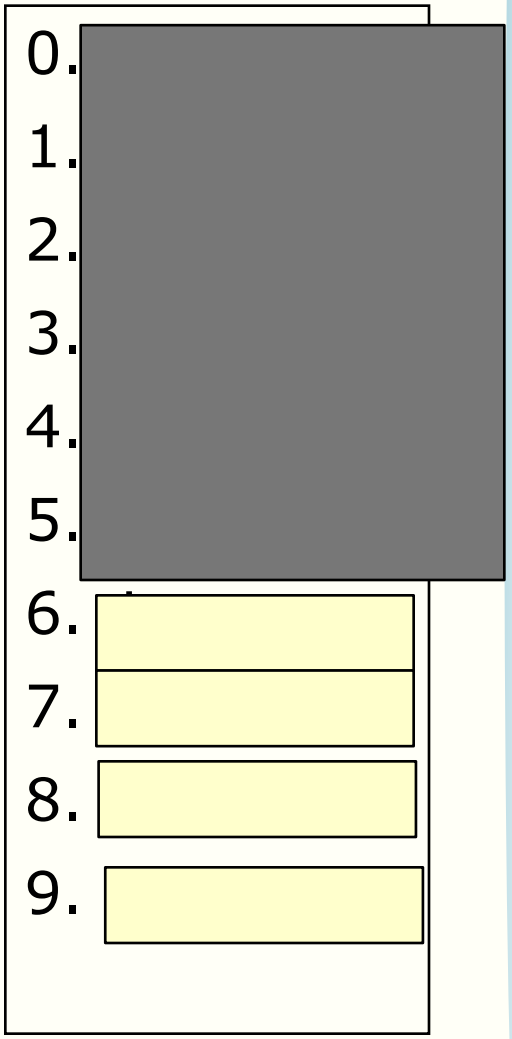


Binary search example (3)

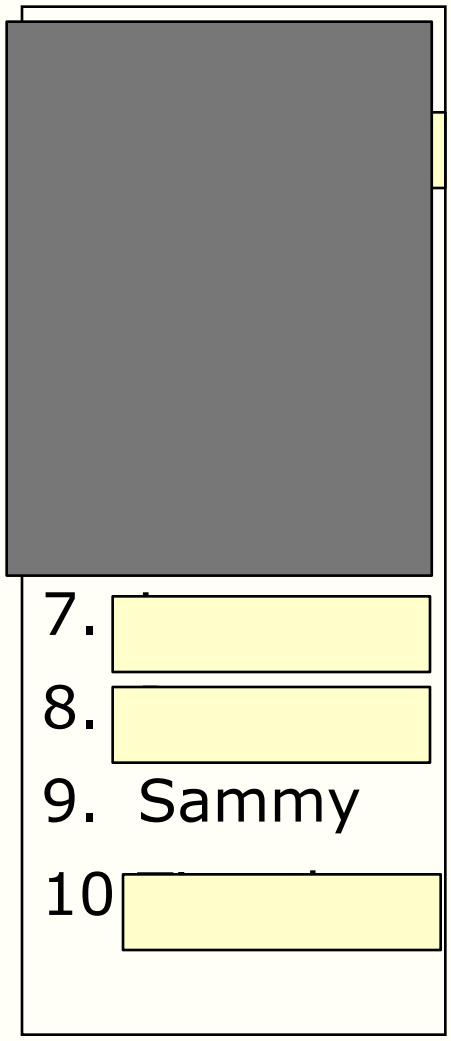


- Search between 6-9
- Middle of 6 and 9
  = (6+9)/2 = 7.5
- Let us round up
- Query index 8

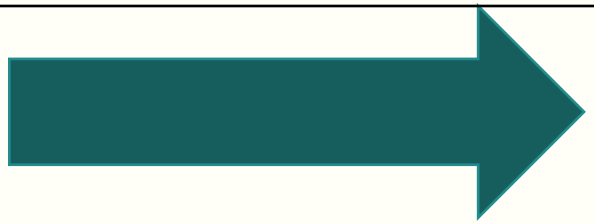


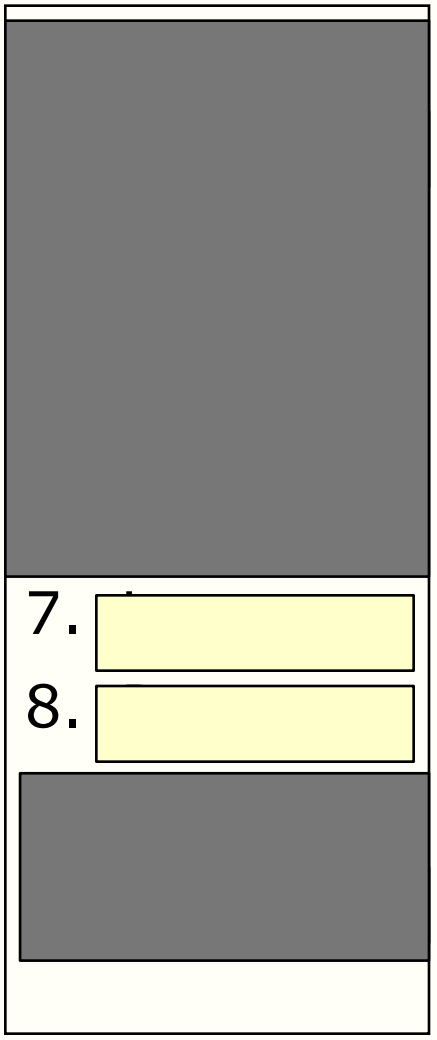


Binary search example (4)

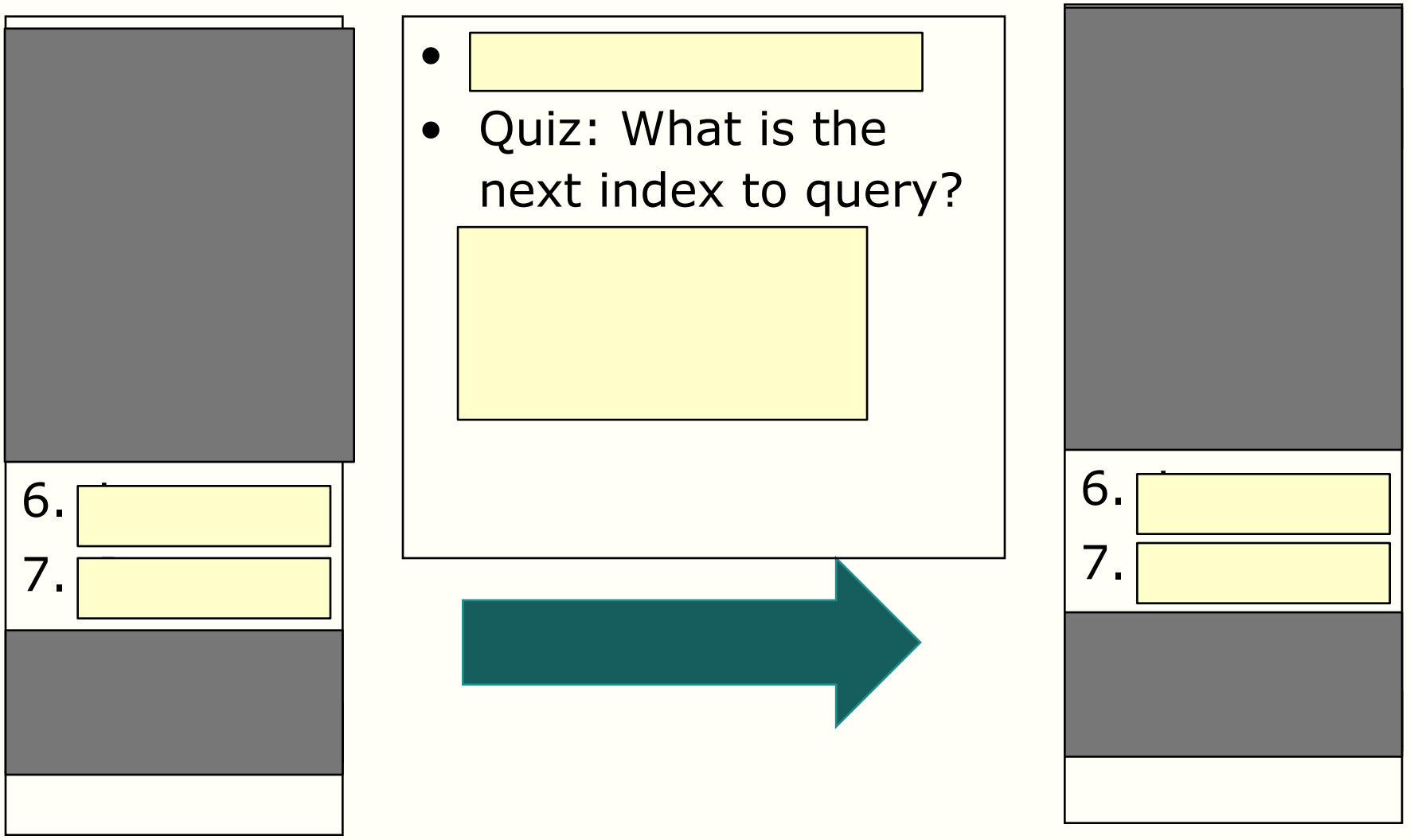


- Which section can we forget now?





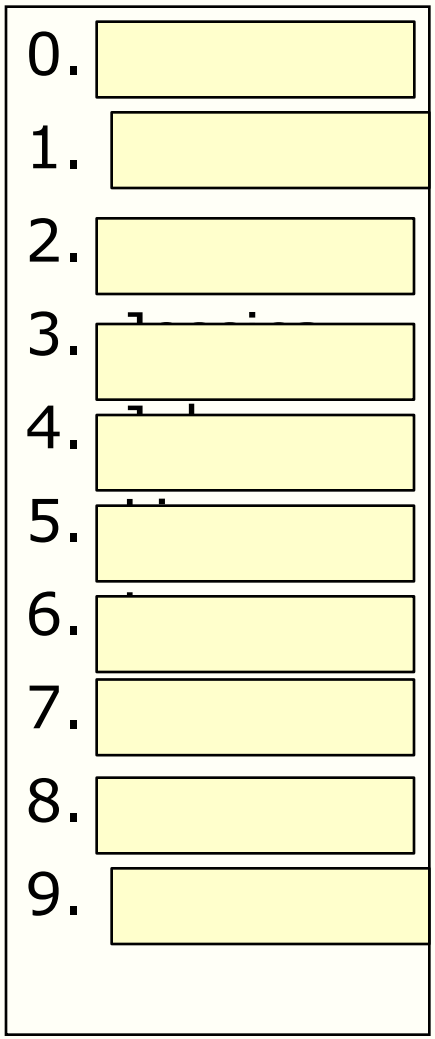
Binary search example (5)



Binary search: A detail

- Note that when we select the mid-point, we have chosen to round up
- We can also choose to round down
- As long as one consistent rounding method is used throughout the algorithm, that’s fine

Algorithmic complexity (+ ALP)



- Computer scientists are very interested in how the complexity of algorithms
  - Roughly speaking, higher complexity translates to a longer running time on the same computer
- Computer scientists like to derive efficient algorithms
- For the name search example earlier:
  - Binary search needs 3 queries
  - Simple scan needs **?** queries
- The difference does not appear to be a lot for this example, but let us increase the size of the list

Demo

- I took all the first names of the all students enrolled in ENGG1811 in 16s2
- Remove all duplicates and sort the names
- There are 484 unique names
- A Python program
  - Will randomly pick 10 names
  - Uses simple scan and binary search to locate those 10 names
  - The function will also report the number of queries made by each method
- There are a number of points that I'd like you to think about when you watch the demo (next slide)
- **Note**: We haven't given you the source code for this demo because you will be writing Python programs on simple scan and binary search in your lab next week

A number of questions

- Is binary search always better?
- What is the largest number of queries required by
  - simple scan
  - binary search

Name	Scan	Binary
Arunkumar	39	6
Duy	94	8
Rachel	305	9
Siyu	356	9
Yuhan	450	8
Yongmin	441	9
Casper	58	7
Justin	194	9
Yuemeng	449	9
Miriam	255	7

Number of queries required by binary search

- Each query reduces the number of possibilities by half

# queries	Remaining # possibilities after the query
1	**484 * (1/2)**
2	**484 * (1/2) * (1/2)**
3	**484 * (1/2) * (1/2) * (1/2)**

- After n queries, # possibilities = $484 * (1/2)^n$
- Finished when only one possibility left

$$484 * (1/2)^n \leq 1 \rightarrow n \geq \log_2(484) \rightarrow n \geq 9$$

- Maximum queries needed = 9

Worst case complexity

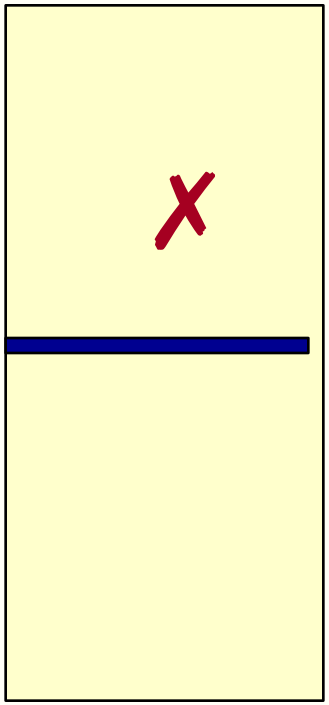
- A way to measure the efficiency of an algorithm is to look at its worst case complexity

- For the problem of locating a name in a sorted list of *n* names
  - Worst case complexity = maximum number of queries ever needed to locate the name
  - Worst case complexity for
    - Simple scan is
    - Binary search

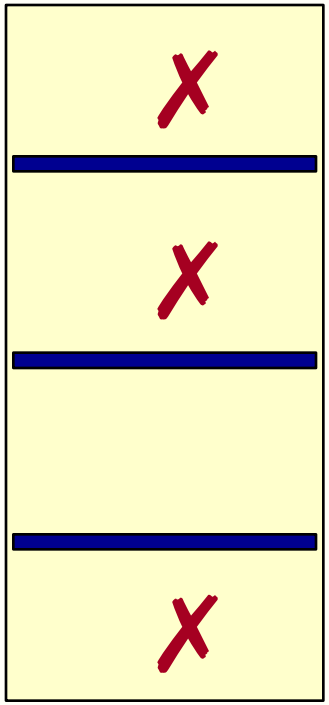
- Computer scientists also use other ways to measure complexity, such as average complexity

Which is more efficient?

Binary: query one name and eliminate half of the names at a time



"Quad"-nary: query three names and eliminate 3 quarters of the names at a time



round / ceil / floor

- Python has 3 functions for rounding
  - **round(x): round to the nearest integer of x**
    - **Note: round(x) is not part of math library**
  - **math.ceil(x): round to the nearest integer bigger than or equal to x**
  - **math.floor(x): : round to the nearest integer smaller than or equal to x**

```
round(1.4)     # = 1
round(1.5)     # = 2

math.ceil(1.4)  # 2
math.ceil(1.5)  # 2
math.ceil(1)    # 1
```

```
math.floor(1.4)     # = 1
math.floor(1.5)     # = 1
```

numpy.random.randint()

- Python numpy function numpy.random.randint() generates random intergers
- For example: The following command generates a random integer in the interval [0,10), i.e. 10 not included

```
np.random.randint(0,10)
```

- See the manual page for more examples

Finding roots

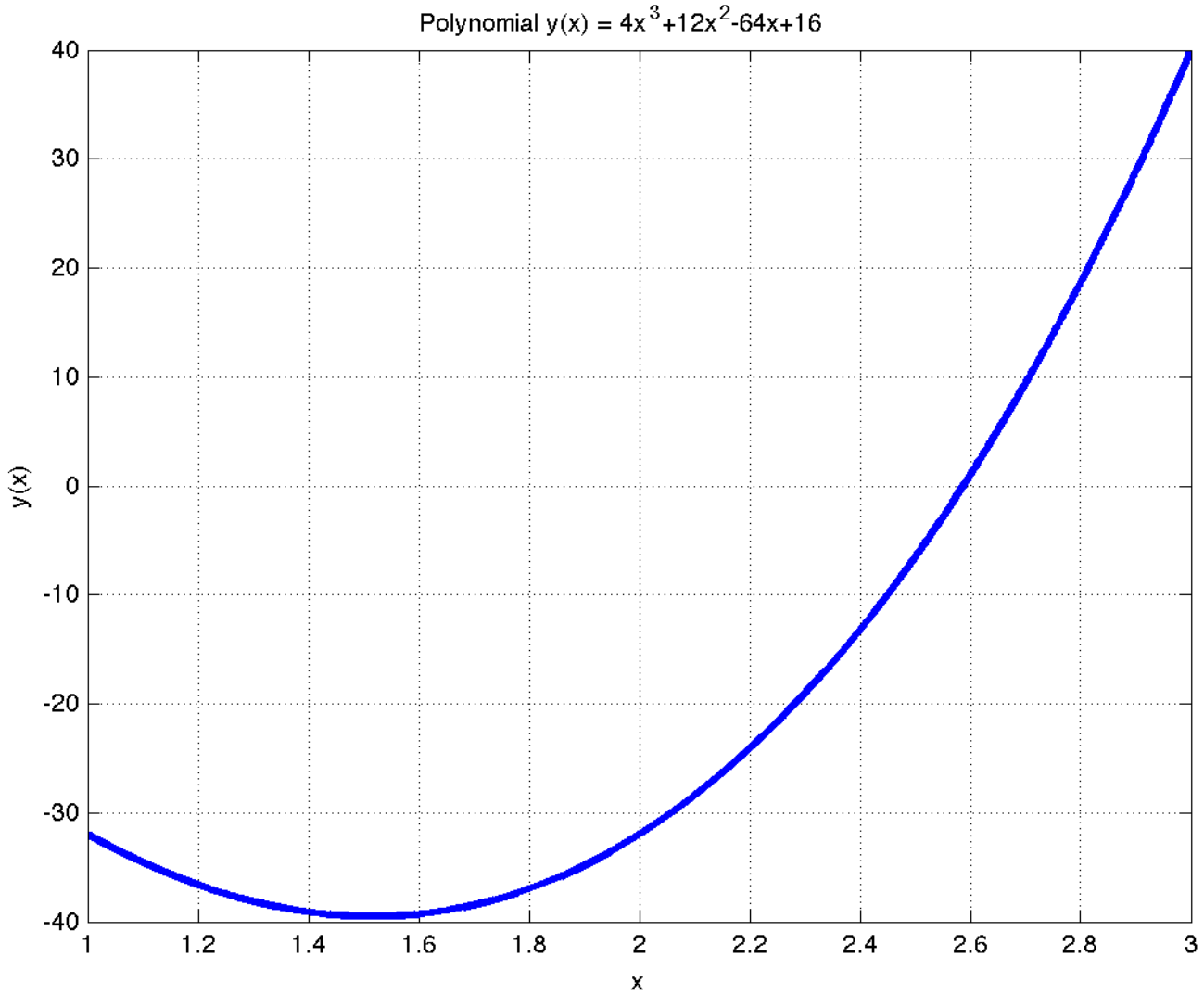
- It's easy to find the roots of polynomial equations using numpy
- Example: To find the roots of the polynomial equation

$$4x^3 + 12\,x^2 - 64\,x + 16$$

```
import numpy as np
poly_roots = np.roots([4, 12, -64, 16])
```

- For the purpose of learning algorithms, we look at how to use bisection method to find roots in Python

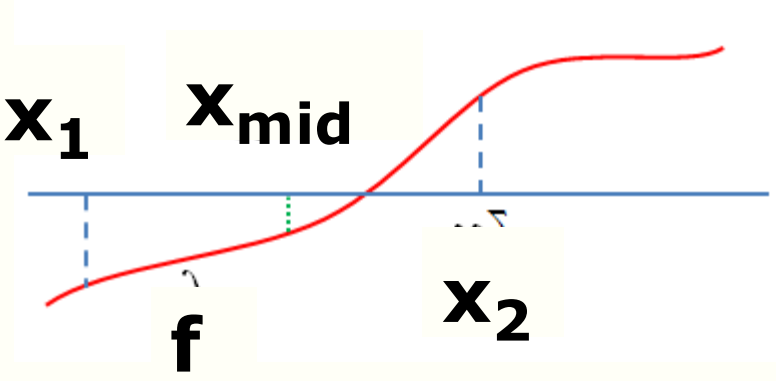
Polynomial $4x^3 + 12\,x^2 - 64\,x + 16$



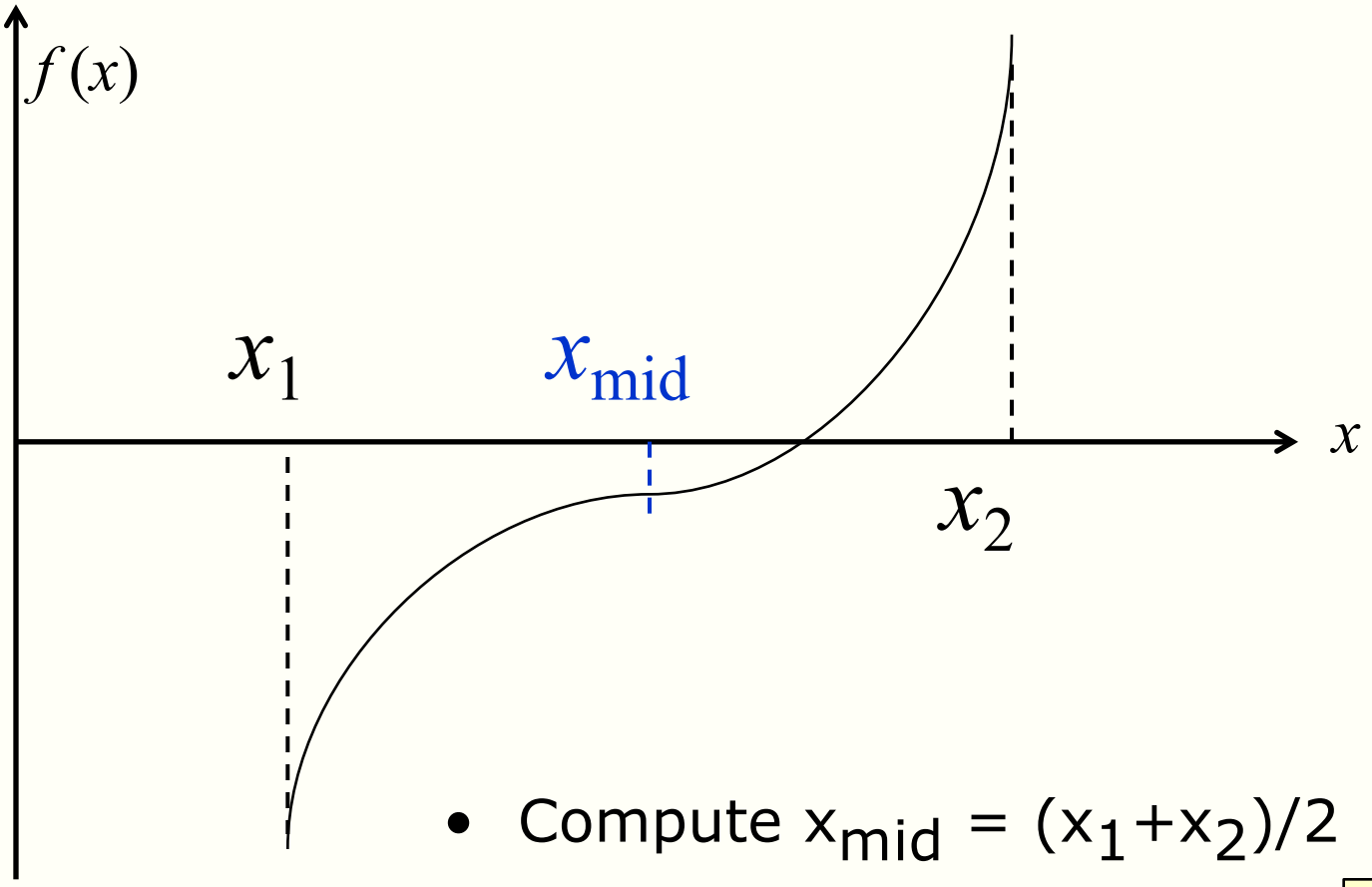
Another application of bisection method

- Bisection method can be used to find the root of a continuous function f(x)
- Assuming you know two numbers x_1 and x_2 such that $f(x_1) < 0$ and $f(x_2) > 0$ then there is a root of $f(x) = 0$ between x_1 and x_2
- We don't know where the root is, we find the mid-point of x_1 and x_2 , and eliminate half of the interval

$$x_{mid} = \frac{x_1 + x_2}{2}$$

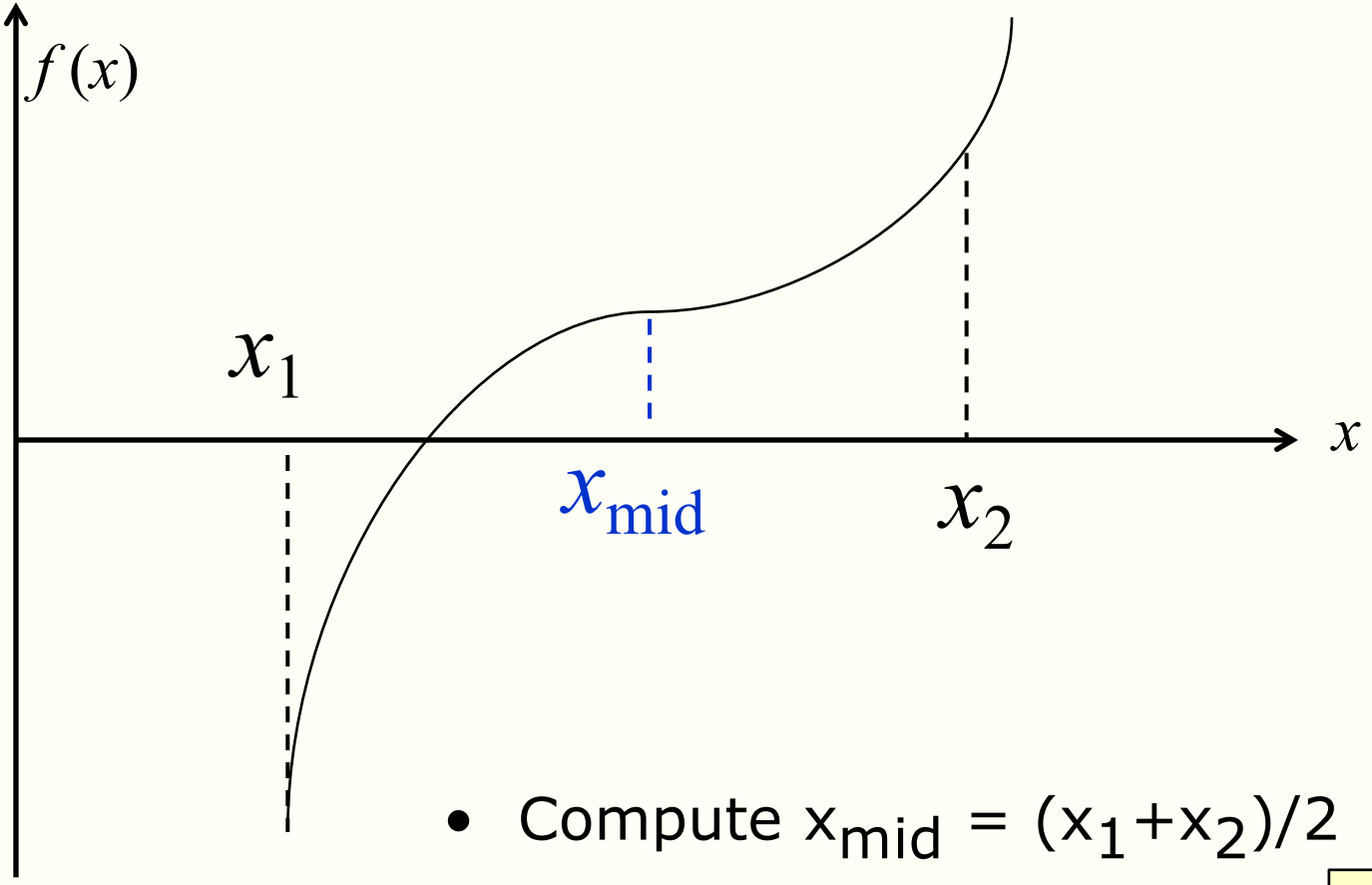


Bisection method: updating, case 1 (+ALP)



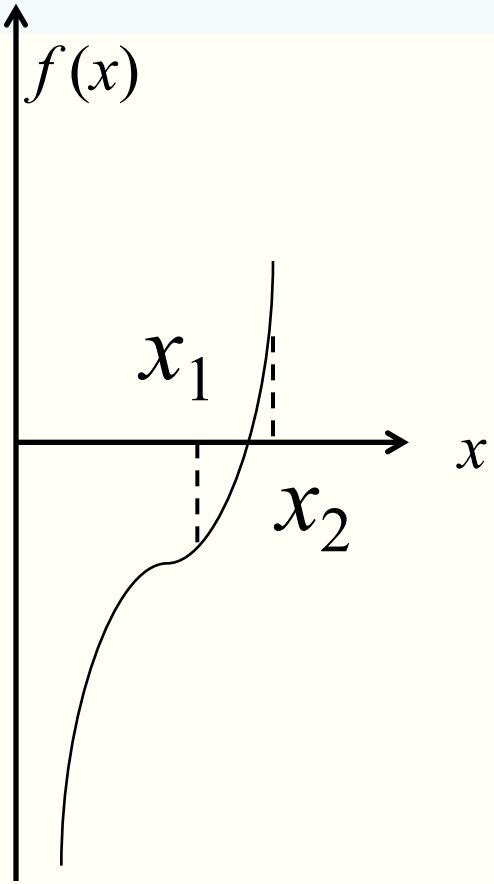
- Compute $x_{mid} = (x_1+x_2)/2$
- If $f(x_{mid}) < 0$, then replace [ ] by x_{mid}

Bisection method: updating, case 2



- Compute $x_{mid} = (x_1+x_2)/2$
- If $f(x_{mid}) > 0$, then replace [ ] by x_{mid}

Stopping criterion



- Stop when x_1 and x_2 are close enough
- Stop if abs(x_1 – x_2) <= TOL where TOL is a pre-defined tolerance
  - abs is short for absolute value
- Question: What is the loop-guard for the while statement?

```
while [          ]
      # continue to iterate
```

A) abs(x_1 – x_2) <= TOL

B) abs(x_1 – x_2) > TOL

Bisection

Pseudocode:

```
choose x1 and x2 such that f(x1) < 0 and f(x2) > 0
while abs(x1-x2) > TOL
        xmid = (x1 + x2) / 2
        if f(xmid) < 0
                x1 = xmid
        else
                x2 = xmid
```

Python implementation

- A Python implementation is in find_root_by_bisection_prelim.m
- We will complete it in the lecture
- It uses the numpy function polyval() to calculate the value of the polynomial y(x) at a given value of x
  - Example: Want to calculate the value of $y(x) = 4x^3 + 12\,x^2 - 64\,x + 16$ at $x = 2$
  - You can use

```
import numpy as np
value_at_x = np.polyval([4, 12, -64, 16],2)
```

Comments on

- The program has a limitation
  - The script assumes that `f(x1) < 0 and f(x2) > 0` but it is possible to write code so that it can work for either of the following:
    - `f(x1) < 0 and f(x2) > 0`
    - `f(x1) > 0 and f(x2) < 0`

Comments on root finding

- Using bisection method to find a root requires you to specify two points which bound the root

- There are algorithms which require you to specify one starting point and find the “closest” root. These algorithms require advance calculus to understand. You will learn them later years

Summary

- Algorithms play a very important role in computer science. Two key issues: Correctness and efficiency
- Algorithms are behind many great computing innovations
  - Computers, Internet, Face and speech recognition etc.
- Algorithms are everywhere in engineering too. Examples:
  - Autopilot, satellite navigation, traffic control, automation of mining, chemical and food production, power grid, robotics, control of combustion engines and many others
  - You may wish to watch the following two videos produced especially for ENGG1811 on application of algorithms in transport (1st video) and human hip tissue map (2nd video)
    - https://youtu.be/CR-bwYiT-IM
    - https://youtu.be/ZV3_ckI_4xw
- Next frontiers for algorithms: Reverse engineering the brain, personalised education, algorithms of living cells etc.