COMPSCI 107
Computer Science Fundamentals

Lecture 10 – Algorithm Analysis

Fibonacci numbers

- Next number is sum of previous two numbers
  - 1, 1, 2, 3, 5, 8, 13, 21 …
- Mathematical definition

$$F(n) = \begin{cases} n & \text{if } n = 0 \text{ or } n = 1; \\ F(n-1) + F(n-2) & \text{if } n \geq 2. \end{cases}$$

Question

- Which of the following would you choose?

```
def fib_a(n):
    if n == 0 or n == 1:
        return n
    if n >= 2:
        return fib_a(n - 1) + fib_a(n - 2)
```

```
def fib_b(n):
    if n == 0 or n == 1:
        return n
    prev = 1
    prev_prev = 0
    for i in range(2, n+1):
        temp = prev + prev_prev
        prev_prev = prev
        prev = temp
    return prev
```

Empirical testing

- How long will it take for fib_a(100) to execute?

Fibonacci number	Time taken	
	`fib_a(n)`	`fib_b(n)`
10	< 0.001 second	< 0.001 second
20	< 0.001 second	< 0.001 second
30	1 second	< 0.001 second
100	???	< 0.001 second

Fibonacci numbers

2	1
3	2
4	3
5	5
6	8
7	13
8	21
9	34
10	55

20	6,765
30	832,040
40	102,334,155
50	12,586,269,025
60	1,548,008,755,920
70	190,392,490,709,135
80	23,416,728,348,467,685
90	2,880,067,194,370,816,120
100	354,224,848,179,261,915,075

Comparing Algorithms

- Analyse performance
  - How much of a given resource do we use?
  - Space (memory)
  - Time

- We are going to be mainly interested in how long our programs take to run, as time is generally a more precious resource than space.

Analysing time to run an algorithm

- Three considerations
  - How are the algorithms encoded?
  - What computer will they be running on?
  - What data will be processed?

- Analysis should be independent of specific
  - Coding,
  - Computers, or
  - Data

- How do we do it?
  - Count the number of basic operations and generalise the count

Example

- Sum the first 10 element of a list

```
def count_operations1(items):
    sum = 0
    index = 0
    while index < 10:
        sum = sum + items[index]
        index += 1

    return sum
```

1 assignment
1 assignment
11 comparisons
10 assignments
10 assignments

1 return

Total: 34

Example

- Sum the elements in a list

```
def count_operations2(items):
    sum = 0                            1 assignment
    index = 0                          1 assignment
    while index < len(items):          N + 1 comparisons
        sum = sum + items[index]       N assignments
        index += 1                     N assignments

    return sum                         1 return
```

Total: 3n + 5

- We express the time as a function of **problem size**

Exercise

- For each of the following, how many operations are required (express in terms of N where possible)?
  1. Adding an element to the beginning of a list containing n elements
  2. Printing each element of a list containing n elements,
  3. Adding a single element to a list using the append() function.
  4. Performing a nested loop where the outer loop is executed n times and the inner loop is executed 10 times. For example, printing out the times tables for all integer values between 1 and n

Exercise

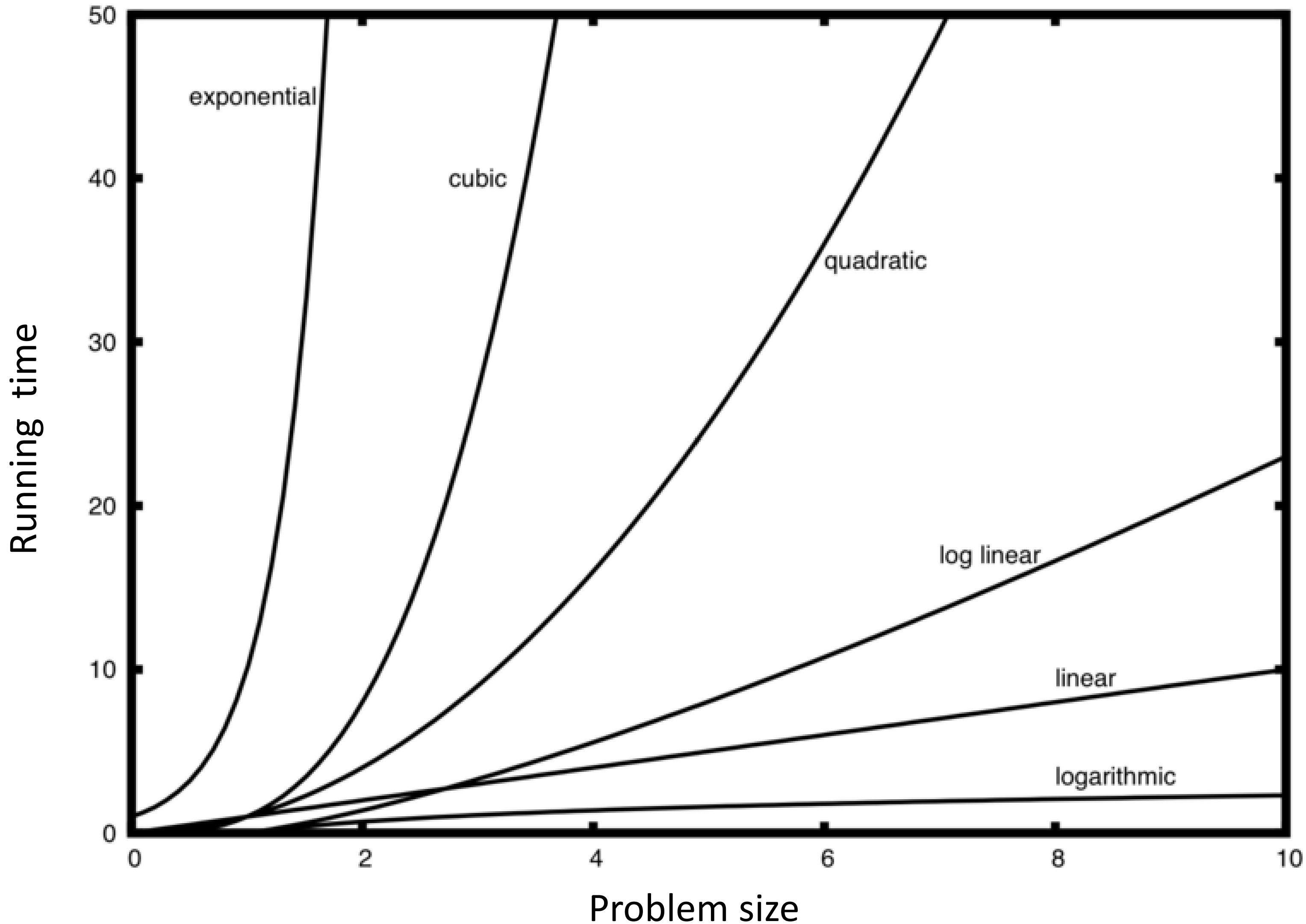
- Assume that we have 5 different algorithms that are functionally equivalent. The time taken to execute each algorithm is described by the respective functions below. Which algorithm would you choose and why?

- (a) $T(n) = n^3 + 4n + 7$
- (b) $T(n) = 20n + 2$
- (c) $T(n) = 3n^2 + 2n + 23$
- (d) $T(n) = 1{,}345{,}778$
- (e) $T(n) = 3\log_2 n + 2n$

Efficiency – we care most about scalability

- For small problem sizes, most code runs extremely fast
  - When we do care about small problem sizes, we can do detailed analysis and measure empirically
- However, running well when the problem is small doesn't mean the code will run well when the problem gets bigger
  - Scalability is critically important
  - Interested in the order of magnitude of the running time
- Can analyse in different levels of detail
  - Crude estimates are good enough

Growth rates of common time-complexity functions



Describing time-complexity

- We can describe the running time of an algorithm mathematically
  - Simply count the number of instructions executed

```
def peek(a):
    #return the first item
    return a[0]
```

```
def pop(a):
    #remove and return the first item
    firstItem = a[0]
    for i in range(1,len(a)):
       a[i-1] = a[i]
    return firstItem
```

Ignore constant factors

- Linear time algorithm takes An + B
  - Where A and B are implementation-specific constants
- When n is large, An is a good approximation
- Since we know the relationship is linear, we can work out A for a particular implementation if we need it.
- For large n, the difference between different order of magnitude is huge – the other factors are insignificant
- Therefore, we don’t need fine distinctions, only crude order of magnitude

Big-O

- We use Big O notation (capital letter O) to specify the complexity of an algorithm e.g., $O(n^2)$, $O(n^3)$, $O(n)$.

- If a problem of size *n* requires time that is directly proportional to *N*, the problem is $O(n)$

- If the time requirement is directly proportional to n^2, the problem is $O(n^2)$

Common big-O functions

f(n)	Name
$O(1)$	Constant
$O(\log n)$	Logarithmic
$O(n)$	Linear
$O(n \log n)$	Log Linear
$O(n^2)$	Quadratic
$O(n^3)$	Cubic
$O(2^n)$	Exponential

Comparison of Growth Rates

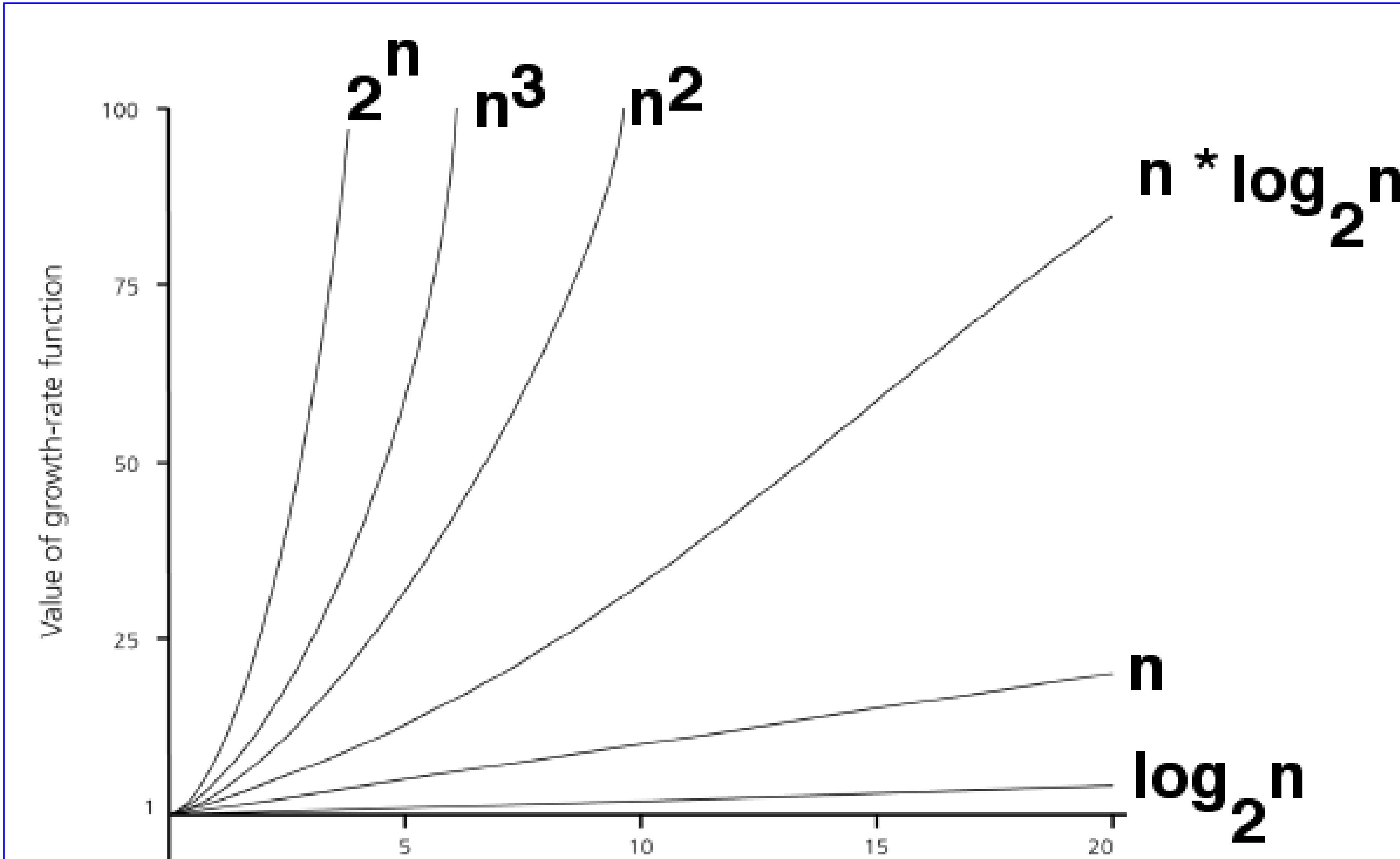
Problem size

n

Number Of Operations

Function	10	100	1,000	10,000	100,000	1,000,000
1	1	1	1	1	1	1
$\log_2 n$	3	6	9	13	16	19
n	10	10^2	10^3	10^4	10^5	10^6
$n * \log_2 n$	30	664	9,965	10^5	10^6	10^7
n^2	10^2	10^4	10^6	10^8	10^{10}	10^{12}
n^3	10^3	10^6	10^9	10^{12}	10^{15}	10^{18}
2^n	10^3	10^{30}	10^{301}	$10^{3,010}$	$10^{30,103}$	$10^{301,030}$

Comparison of Growth Rates



Properties of Big O

- When considering the Big O for an algorithm, the Big O's can be combined e.g.

$$O(n^2) + O(n) = O(n^2 + n)$$

$$O(n^2) + O(n^4) = O(n^2 + n^4)$$

Properties of Big O

When considering the Big O for an algorithm, **any lower order terms** in the growth function can be ignored e.g.

$$O(n^3 + n^2 + n + 5000) = O(n^3)$$

$$O(n + n^2 + 5000) = O(n^2)$$

$$O(1500000 + n) = O(n)$$

Properties of Big O

- When considering the Big O for an algorithm, any **constant multiplications** in the growth function can be ignored e.g.

$$O(254 * n^2 + n) = O(n^2)$$

$$O(546 * n) = O(n)$$

$$O(n / 456) = O((1/456) * n) = O(n)$$

Exercise

- What is the Big O of the following growth functions?

a) $T(n) = n + \log(n)$

b) $T(n) = n^4 + n*\log(n) + 3000n^3$

c) $T(n) = 300n + 60 * n * \log(n) + 342$

Worst-case and average-case analyses

- An algorithm can require different times to solve different problems of the same size. For example, search for a particular element in an array.

  Best-case analysis: the minimum amount of time that an algorithm requires to solve problems of size n

  Worst-case analysis: the maximum amount of time that an algorithm requires to solve problems of size n

  Average-case analysis: the average amount of time that an algorithm requires to solve problems of size n

Average performance and worst-case performance are the most commonly used in algorithm analysis.

Exercise

- What is the big-O running time for the code:

```
def question(n):
    count = 0
    for i in range(n):
        count += 1
    for j in range(n):
        count += 1
    return count
```

  - (a) O(1)
  - (b) O(logn)
  - (c) O(n)
  - (d) O(n^2)
  - (e) None of the above

Exercise

- What is the big-O running time for the code:

```
def question(n):
    count = 0
    for i in range(n):
        count += 1
        for j in range(n):
            count += 1
    return count
```

  - (a) O(1)
  - (b) O(logn)
  - (c) O(n)
  - (d) O(n^2)
  - (e) None of the above

Exercise

- What is the big-O running time for the code:

```
def question(n):
    count = 0
    for i in range(n):
        count += 1
        for j in range(10):
            count += 1
    return count
```

  - (a) O(1)
  - (b) O(logn)
  - (c) O(n)
  - (d) O(n^2)
  - (e) None of the above

Exercise

- What is the big-O running time for the code:

```
def question(n):
    count = 0
    for i in range(n):
        count += 1
        for j in range(i+1):
            count += 1
    return count
```

- (a) O(1)
- (b) O(logn)
- (c) O(n)
- (d) O(n^2)
- (e) None of the above

Exercise

- What is the big-O running time for the code:

```
def question(n):
    i = 1
    count = 0
    while i < n:
        count += 1
        i = i * 2
    return count
```

  - (a) O(1)
  - (b) O(logn)
  - (c) O(n)
  - (d) O(n^2)
  - (e) None of the above

Exercise

- What is the big-O running time for the code:

```
def question(n):
    i = 1
    count = 0
    while i < n:
        count += 1
        i = i + 2
    return count
```

  - (a) O(1)
  - (b) O(logn)
  - (c) O(n)
  - (d) O(n^2)
  - (e) None of the above

Exercise

- What is the big-O running time for the code:

```
def question(n):
    count = 0
    for i in range (n):
        j = 0
        while j < n:
            count += 1
            j = j * 2
    return count
```

  - (a) O(1)
  - (b) O(logn)
  - (c) O(n)
  - (d) O(n^2)
  - (e) None of the above

Exercise

- What is the big-O running time for the following function?

```
def exampleA(n):
    s = "PULL FACES"

    for i in range(n):
        print("I must not ", s)

    for j in range(n, 0, -1):
        print("I must not ", s)
```

Exercise

- What is the big-O running time for the following function?

```
def exampleB(n):
        s = "JUMP ON THE BED"

        for i in range(n):
                for j in range(i):
                        print("I must not ", s)
```

Exercise

- What is the big-O running time for the following function?

```
def exampleC(n):
        s = "WHINGE"
        i = 1
        while i < n:
                for j in range(n):
                        print("I must not ", s)

                i = i * 2
```

Exercise

- What is the big-O running time for the following function?

```
def exampleD(n):
        s = "PROCRASTINATE"

        for i in range(n):
                for j in range(n, 0, -1):
                        outD(s, n / 2)

def outD(s, b):
        number_of_times = int(b % 10)
        for i in range(number_of_times):
                print(i, "I must not ", s)
```

Exercise

- What is the big-O running time for the following function?

```
def exampleF(n):
    s = "FORGET MY MOTHER’S BIRTHDAY"
    i = n
    while i > 0:
        outF(s)
        i = i // 2

def outF(s):
    for i in range(25, 0, -1):
        print(i, "I must not ", s)
```

Challenge Question

- If a particular quadratic time algorithm uses 300 elementary operations to process an input of size 10, what is the most likely number of elementary operations it will use if given an input of size 1000.

- (a) 300 000 000
- (b) 3 000 000
- (c) 300 000
- (d) 30 000
- (e) 3 000

Challenge Question

- You know that a given algorithm runs in $O(2^n)$ time. If your computer can process input of size 10000 in one year using an implementation of this algorithm, approximately what size input could you solve in one year with a computer 1000 times faster?

A. 10 100
B. 320 000
C. 10 010
D. 15 000
E. 10 000 000

ChallengeQuestion

The running time for the following code fragment is $\Theta(f(n))$.

for (int $i = 0; i < n; i$++)
    for (int $j = i - 10; j < i; j$++)
        for (int $k = 1; k < n; k = 4 * k$)
            System.out.println(i);
        end for
    end for
end for

A. $n \log n$
B. $n^2 \log n$
C. n^2
D. n^3
E. $n \log \log n$