COMP90038 Algorithms and Complexity

Tutorial 2 Analysis of Algorithm Complexity

Tutor: Chuang(Frank) Wang

Prerequisite

- ❑ **Chapter 2.** *Levitin, A. (2012). Introduction to the design & analysis of algorithms. (3rd Ed.)*
- ❑ ***Week 2 Lectures***

1. Review

Complexity Analysis

Complexity:

how do the resource requirements of a program or algorithm scale, i.e., what happens as the size of the problem(input) being solved becomes incredibly large?

Time

Space

Time Complexity:

Time complexity is principally measured as a function of its input size by counting the number of times its **basic operation** is executed.

Basic Operation
- the most important operation
- it is usually the most time-consuming operation in the algorithm's innermost loop

e.g. $f(n) = 7n^3 + 28n^2 + 60$

n - the size of the input

Big-O Notation

- A way that we use to measure the complexity by quantifying performance as the input size goes to infinity.
- It gives an upper bound of the complexity in the worst case. e.g. to sequentially search a target in a list.

item	89	7	...	26	50
index	0	1	...	1billion -2	1billion-1

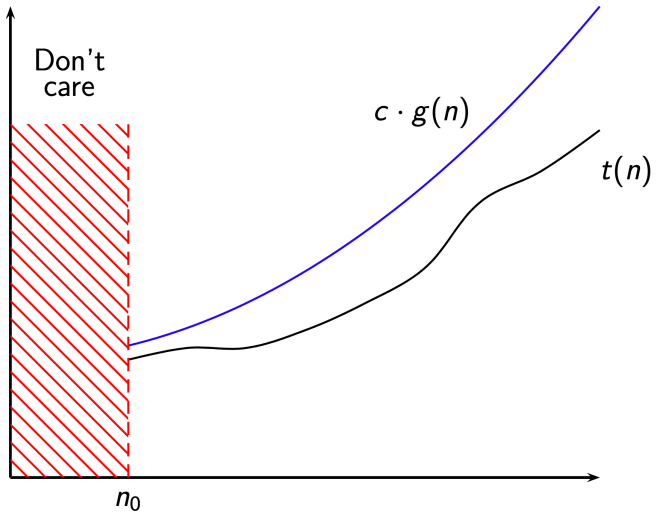
- $O(g(n))$ denotes the set of functions that grow no faster than g, asymptotically.

- Formal Definition:

  $t(n) \in O(g(n))$

  For some c and n_0

  $n > n_0 \Rightarrow t(n) \in c \cdot g(n)$



Big-O Properties

- As we are interested in the growth rate as the input size grows to infinity:
- We ignore small input sizes
- We ignore constant added to big-o notation.

$$O(n + c) = O(n)$$

- We ignore multiplicative factors.

$$O(cn) = O(n), c > 0$$

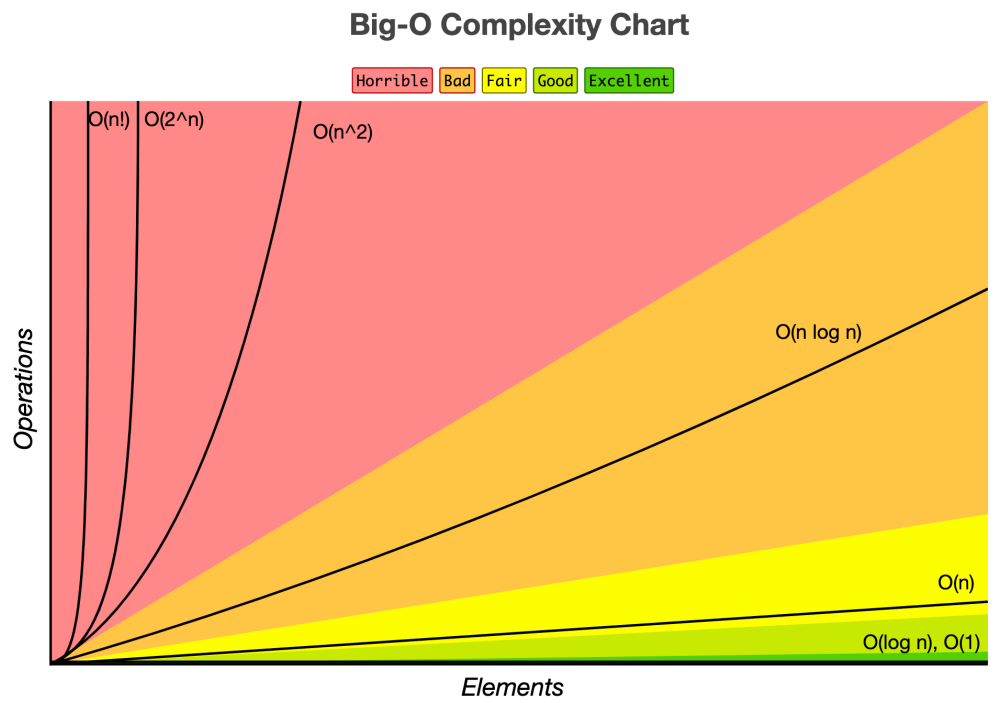
e.g. $f(n) = 7n^3 + 28n^2 + 60$

$$O(f(n)) = O(n^3)$$

Asymptotic Analysis

- Establishing growth rate
- $$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \begin{cases} 0 \rightarrow f \text{ grows asymptotically slower than } g \\ c \rightarrow f \text{ and } g \text{ have same order of growth} \\ \infty \rightarrow f \text{ grows asymptotically faster than } g \end{cases}$$

Question 11 in this tutorial!



Source

- Constant Time: O(1)
- Logarithmic Time: O(log n)
- Linear Time: O(n)
- Linearithmic Time: O(n logn)
- Quadratic Time: O(n^2)
- Cubic Time: O(n^3)
- Exponential Time: O(b^n), b>1
- Factorial Time: O(n!)

2. Tutorial Questions

Question 11

11. For each of the following pairs of functions, indicate whether the components have the same order of growth, and if not, which grows faster.

(a) $n(n+1)$ and $2000n^2$ (b) $100n^2$ and $0.01n^3$

(c) $\log_2 n$ and $\ln n$ (d) $\log_2^2 n$ and $\log_2 n^2$

(e) 2^{n-1} and 2^n (f) $(n-1)!$ and $n!$

Here ! is the factorial function, and $\log_2^2 n$ is the way we usually write $(\log_2 n)^2$.

Solution: L'Hôpital's Rule

$$\lim_{h\to\infty} \frac{t(n)}{g(n)} = \lim_{h\to\infty} \frac{t'(n)}{g'(n)}$$

where t' and g' are the derivatives of t and g

(a) $\lim_{n\to\infty} \frac{n^2+n}{2000\,n^2} = \lim_{n\to\infty} \frac{2n+1}{4000n} = \lim_{n\to\infty} \frac{2}{4000} = constant$

Same growth rate

(b) $\lim_{n\to\infty} \frac{100n^2}{0.01n^3} = \lim_{n\to\infty} \frac{200n}{0.03n^2} = \lim_{n\to\infty} \frac{200}{0.03n} = 0$

$0.01n^3$ grows faster than $100n^2$

(c) $\lim_{n\to\infty} \frac{\log_2 n}{\ln n} = \lim_{n\to\infty} \frac{\frac{1}{n\ln 2}}{\frac{1}{n}} = \lim_{n\to\infty} \frac{1}{\ln 2} = constant$

Same growth rate

(d) $\lim_{n\to\infty} \frac{(\log_2 n)^2}{\log_2 n^2} = \lim_{n\to\infty} \frac{(\log_2 n)\,(\log_2 n)}{2\log_2 n} = \lim_{n\to\infty} \frac{1}{2}\log_2 n = \infty$

$(\log_2 n)^2$ grows faster than $\log_2 n^2$

(e) $\lim_{n\to\infty} \frac{2^{n-1}}{2^n} = \lim_{n\to\infty} \frac{1}{2} = constant$

Same growth rate

(f) $\lim_{n\to\infty} \frac{(n-1)!}{n!} = \lim_{n\to\infty} \frac{1}{n} = 0$

$n!$ grows faster than $(n-1)!$

Question 10

10. Gaussian elimination, the classical algorithm for solving systems of n linear equations in n unknowns, requires about $\frac{1}{3}n^3$ multiplications, which is the algorithm's basic operation.

    (a) How much longer should we expect Gaussian elimination to spend on 1000 equations, compared with 500 equations?

    (b) You plan to buy a computer that is 1000 times faster than what you currently have. By what factor will the new computer increase the size of systems solvable in the same amount of time as on the old computer?

Solution:

(a) 8 times longer. $\frac{\frac{1}{3}(2n)^3}{\frac{1}{3}(n)^3} = \frac{8n^3}{n^3} = 8$

(b)

- t_{new} → units of time that the new machine needs for a job
- t_{old} → units of time that the old machine needs for a job

$$t_{old} = 1000 t_{new}$$

Give a fixed time period

- m → the number of equations that the new machine can handle
- n → the number of equations that the old machine can handle
- $\frac{1}{3}m^3$ → time the new machine needs to handle m equations
- $\frac{1}{3}n^3$ → time the old machine needs to handle n equations

$$\frac{1}{3}m^3 = 1000\frac{1}{3}n^3 \qquad m = \sqrt[3]{1000n^3} = 10n$$

So we can now solve systems that are 10 times larger.

Question 8

8. Design an algorithm to check whether two given words are anagrams, that is, whether one can be obtained from the other by permuting its letters. For example, *garner* and *ranger* are anagrams.

Def: Two strings are said to be anagrams of one another if you can turn the first string into the second by rearranging its letters. For example, “table” and “bleat” are anagrams, as are “tear” and “rate.”

Solution:

Q: Can we do “for each letter in the first word, check that it occurs in the second, and vice versa.”?

A: ✗ **Duplicate letters! e.g. garner vs. range**

Approach 1: Brute force
1) list off all permutations of the first string
2) see if any of them are equal to the second string

egg **gge**
geg
gge

Approach 2: Sorting
1) sort the characters in each string
2) test whether the sorted strings are equal

tear **rate**
aert **aert**

✓ ***Good performance!***

Approach 2: Counting Characters
1) count how many times each character appears in each string
2) test if each string has the same number of each character as the other.

garner `d1 = {“a”:1, “e”:1, “g”:1, “n”:1, “r”:2}`

range `d2 = {“a”:1, “e”:1, “g”:1, “n”:1, “r”:1}`

Question 9

9. Here we consider a function `first_occ`, such that `first_occ`(d, s) returns the first position (in string s) which holds any symbol from the collection d (and returns -1 if there is no such position). For simplicity let us assume both arguments are given as strings. For example, `first_occ`("`aeiou`", s) means "find the first vowel in `s`." For s = "`zzzzzzzzzzzz`" this should return -1, and for s = "`Phlogiston`", it should return 3 (assuming we count from 0).

   Assuming strings are held in arrays, design an algorithm for this and find its complexity as a function of the lengths of d and s.

   Suppose we know that the set of possible symbols that may be used in d has cardinality 256. Can you find a way of utilising this to speed up your algorithm? Hint: Find a way that requires only one scan through d and only one through s.

Approach 1: Brute force

```
function first_occ(d[0..len_d - 1], s[0..len_s - 1])
    for i ← 0 to len_s - 1 do
        for j ← 0 to len_d - 1 do
            if s[i] == d[j]:
                return i
    return -1
```

array **s** ->

f	o	r	c	e

array **d** ->

a	e	i	o	u

Complexity: O(len_d * len_s)

- len_d = the length of d
- len_s = the length of s

Question 9

9. Here we consider a function `first_occ`, such that `first_occ`(d, s) returns the first position (in string s) which holds any symbol from the collection d (and returns -1 if there is no such position). For simplicity let us assume both arguments are given as strings. For example, `first_occ`("`aeiou`", s) means "find the first vowel in `s`." For s = "`zzzzzzzzzzz`" this should return -1, and for s = "`Phlogiston`", it should return 3 (assuming we count from 0).

   Assuming strings are held in arrays, design an algorithm for this and find its complexity as a function of the lengths of d and s.

   Suppose we know that the set of possible symbols that may be used in d has cardinality 256. Can you find a way of utilising this to speed up your algorithm? Hint: Find a way that requires only one scan through d and only one through s.

Approach 2:

array **s** ->

a	e	i	o	u

array **d** ->

f	o	r	c	e

helper array **first** -> to record all the symbols appearing in d

- indexed by the symbol's ASCII value (0...255)
- all elements are initialized with -1
- Scan through d. For each symbol x in d, if first[asc(x)] is -1, replace it with 1

elem	-1	...	1	...	1	1	...	1	...	1	...	-1
index	0	...	99	...	101	102	...	111	...	114	...	255

ASCII TABLE

Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char
0	0	[NULL]	32	20	[SPACE]	64	40	@	96	60	`
1	1	[START OF HEADING]	33	21	!	65	41	A	97	61	a
2	2	[START OF TEXT]	34	22	"	66	42	B	98	62	b
3	3	[END OF TEXT]	35	23	#	67	43	C	99	63	c
4	4	[END OF TRANSMISSION]	36	24	$	68	44	D	100	64	d
5	5	[ENQUIRY]	37	25	%	69	45	E	101	65	e
6	6	[ACKNOWLEDGE]	38	26	&	70	46	F	102	66	f
7	7	[BELL]	39	27	'	71	47	G	103	67	g
8	8	[BACKSPACE]	40	28	(	72	48	H	104	68	h
9	9	[HORIZONTAL TAB]	41	29	)	73	49	I	105	69	i
10	A	[LINE FEED]	42	2A	*	74	4A	J	106	6A	j
11	B	[VERTICAL TAB]	43	2B	+	75	4B	K	107	6B	k
12	C	[FORM FEED]	44	2C	,	76	4C	L	108	6C	l
13	D	[CARRIAGE RETURN]	45	2D	-	77	4D	M	109	6D	m
14	E	[SHIFT OUT]	46	2E	.	78	4E	N	110	6E	n
15	F	[SHIFT IN]	47	2F	/	79	4F	O	111	6F	o
16	10	[DATA LINK ESCAPE]	48	30	0	80	50	P	112	70	p
17	11	[DEVICE CONTROL 1]	49	31	1	81	51	Q	113	71	q
18	12	[DEVICE CONTROL 2]	50	32	2	82	52	R	114	72	r
19	13	[DEVICE CONTROL 3]	51	33	3	83	53	S	115	73	s
20	14	[DEVICE CONTROL 4]	52	34	4	84	54	T	116	74	t
21	15	[NEGATIVE ACKNOWLEDGE]	53	35	5	85	55	U	117	75	u
22	16	[SYNCHRONOUS IDLE]	54	36	6	86	56	V	118	76	v
23	17	[ENG OF TRANS. BLOCK]	55	37	7	87	57	W	119	77	w
24	18	[CANCEL]	56	38	8	88	58	X	120	78	x
25	19	[END OF MEDIUM]	57	39	9	89	59	Y	121	79	y
26	1A	[SUBSTITUTE]	58	3A	:	90	5A	Z	122	7A	z
27	1B	[ESCAPE]	59	3B	;	91	5B	[	123	7B	{
28	1C	[FILE SEPARATOR]	60	3C	<	92	5C	\	124	7C	|
29	1D	[GROUP SEPARATOR]	61	3D	=	93	5D	]	125	7D	}
30	1E	[RECORD SEPARATOR]	62	3E	>	94	5E	^	126	7E	~
31	1F	[UNIT SEPARATOR]	63	3F	?	95	5F	_	127	7F	[DEL]

```
function get_first(d[0.. len_d - 1])
    first[0..255] ← -1
    for i ← 0 to len_d - 1 do
        x_asc = asc(d[i])
        if first[x_asc] == -1:
            first[x_asc] == 1
    return first
```

array **s** -> a single scan to find the first position which holds any symbol from d

- Scan through s from the beginning. For each symbol x in s, if first[asc(x)] is 1, return x's position in s. If we check all elements in s without finding the element, return -1.

```
function first_occ(d[0.. len_d - 1], s[0.. len_s - 1])
    first = get_first(d[0.. len_d - 1])
    for i ← 0 to len_s - 1 do
        x_asc = asc(s[i])
        elem = first[x_asc]
        if elem == 1:
            return i
    return -1
```

Complexity: O(len_d + len_s)

- len_d = the length of d
- len_s = the length of s

Question 9

9. Here we consider a function `first_occ`, such that `first_occ`(d, s) returns the first position (in string s) which holds any symbol from the collection d (and returns -1 if there is no such position). For simplicity let us assume both arguments are given as strings. For example, `first_occ`("`aeiou`", s) means "find the first vowel in `s`." For s = "`zzzzzzzzzzzz`" this should return -1, and for s = "`Phlogiston`", it should return 3 (assuming we count from 0).

   Assuming strings are held in arrays, design an algorithm for this and find its complexity as a function of the lengths of d and s.

   Suppose we know that the set of possible symbols that may be used in d has cardinality 256. Can you find a way of utilising this to speed up your algorithm? Hint: Find a way that requires only one scan through d and only one through s.

Approach 3 (Optional) :

array **s** ->

f	o	r	c	e

array **d** ->

a	e	i	o	u

helper array **first** -> to record the position of the first occurrence of each symbol in s

- indexed by the symbol's ASCII value (0...255)
- all elements are initialized with len_s (the length of s)
- Scan through s. For each symbol x in s, if first[asc(x)] is len_s, replace it with x's position.

elem	n	...	3	...	4	0	...	1	...	2	...	n
index	0	...	99	...	101	102	...	111	...	114	...	255

```
function get_first(s[0.. len_s - 1])
    first[0..255] ← len_s
    for i ← 0 to len_s - 1 do
        x_asc = asc(s[i])
        if first[x_asc] == len_s:
            first[x_asc] == i
    return first
```

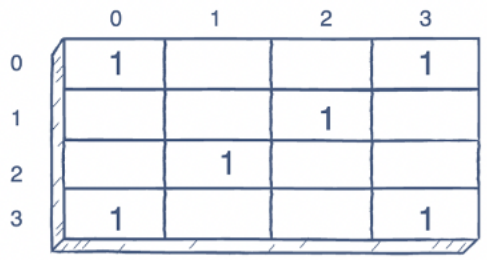
array **d** ->

- A single scan through d. find the minimum value of first[x_asc] for each symbol x in d.

```
function first_occ(d[0.. len_d - 1], s[0.. len_s - 1])
    first = get_first(s[0.. len_s - 1])
    min = Max_Int
    for i ← 0 to len_d - 1 do
        x_asc = asc(d[i])
        elem = first[x_asc]
        if elem < min:
            min = elem
    if min == n:
        return -1
    else:
        return min
```

Complexity: O(len_d + len_s)

Question 7



Source

7. One way of representing an undirected graph G is using an *adjacency matrix*. This is an $n \times n$ matrix, where n is the number of nodes in G. In this matrix, we label the rows and the columns with the names of G's nodes. For a pair (x, y) of nodes, the matrix has a 1 in the (x, y) entry if G has an edge connecting x and y; otherwise the entry has a 0. (As the graph is undirected, this is somewhat redundant: (x, y) and (y, x) will always be identical.) A *complete* graphs is one in which x and y are connected, for all x, y with $x \neq y$. We don't exclude the possibility of a *loop*, that is, an edge connecting a node to itself, although loops are rare.

How can you tell from its matrix whether an undirected graph

(a) is complete?

(b) has a loop?

(c) has an isolated node?

- An Adjacency Matrix is a 2D array of size V x V where V is the number of nodes in a graph.
- A cell matrix[i][j] = 1 indicates that there is an edge from node i to node j.

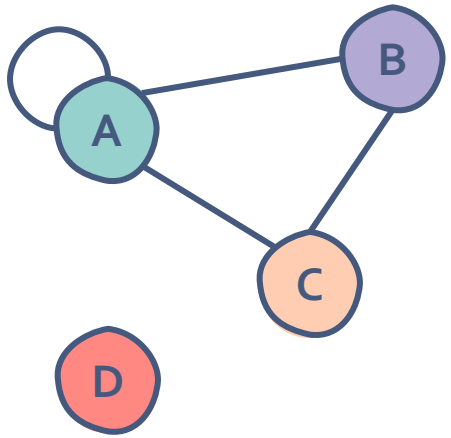
Solution:

(a) All elements (except those on the main diagonal) are 1.

(b) Some element on the main diagonal is 1.

(c) Some row (and hence some column) has all zeros.

	A	B	C	D
A	1	1	1	
B	1		1	
C	1	1		
D				



THE UNIVERSITY OF MELBOURNE

Next Week:

1st Algorithm Technique
- how to use brute force to solve algorithm problems

Thank you !