2.3.7 Execution time for algorithms with given time complexites

Assume n= 1 takes .00001 second

Time complexity function	Size n: 10	20	30	40	50	60
n	.00001 second	.00002 second	.00003 second	.00004 second	.00005 second	.00006 second
n^2	.0001 second	.0004 second	.0009 second	.0016 second	.0025 second	.0036 second
n^3	.001 second	.008 second	.027 second	.064 second	.125 second	.216 second
n^5	.1 second	3.2 seconds	24.3 seconds	1.7 minutes	5.2 minutes	13.0 minutes
2^n	.001 second	1.0 second	17.9 minutes	12.7 days	35.7 years	366 centuries
3^n	.059 second	58 minutes	6.5 years	3855 centuries	2×10^8 centuries	1.3×10^{13} centuries

Let us purchase 1000 times faster laptop

Size of Largest Problem Instance Solvable in 1 Hour

Time complexity function	With present computer	With computer 100 times faster	With computer 1000 times faster
n	N_1	$100\ N_1$	$1000\ N_1$
n^2	N_2	$10\ N_2$	$31.6\ N_2$
n^3	N_3	$4.64\ N_3$	$10\ N_3$
n^5	N_4	$2.5\ N_4$	$3.98\ N_4$
2^n	N_5	$N_5+6.64$	$N_5+9.97$
3^n	N_6	$N_6+4.19$	$N_6+6.29$

Listed from slowest to fastest growth:

- 1
- log n
- n
- n log n
- n^2
- n^3
- 2^n
- n!

Figure 2.20: Execution time

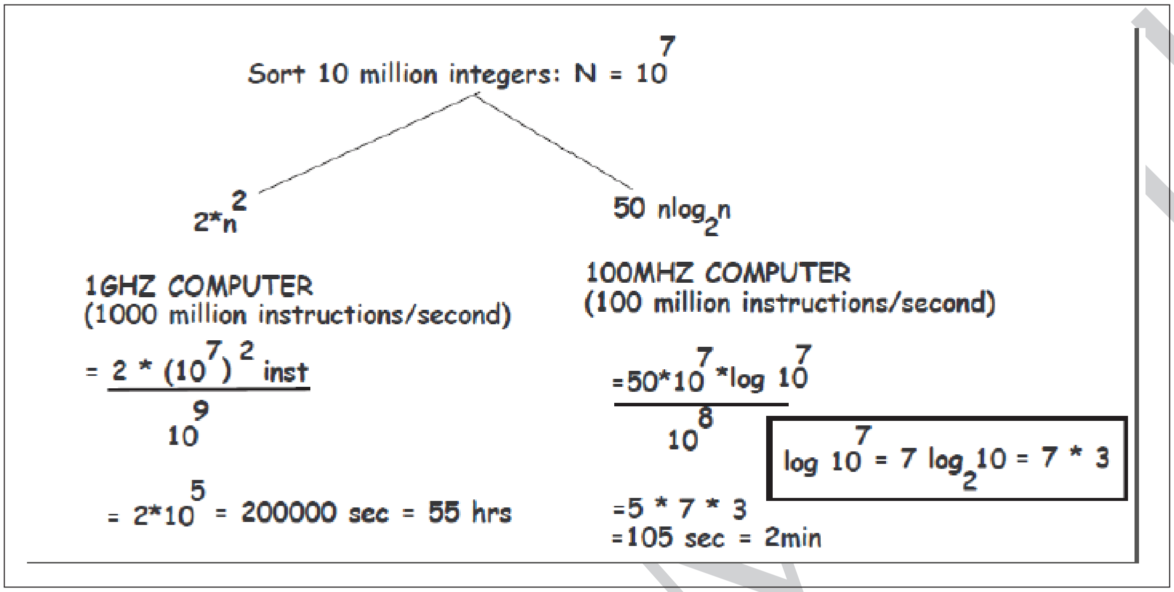


Figure 2.21: Cpu time difference between n^2 and $nlog_2n$ algorithms

2.3.8 O(log log n) algorithms

O(log log n) algorithm

Take the number 65,536.
How many times do we have to divide this by 2 until we get down to 1?

16 steps

65,536 / 2 = 32,768
32,768 / 2 = 16,384
16,384 / 2 = 8,192
8,192 / 2 = 4,096
4,096 / 2 = 2,048
2,048 / 2 = 1,024
1,024 / 2 = 512
512 / 2 = 256
256 / 2 = 128
128 / 2 = 64
64 / 2 = 32
32 / 2 = 16
16 / 2 = 8
8 / 2 = 4
4 / 2 = 2
2 / 2 = 1

4 steps

$\sqrt{65,536} = 256$
$\sqrt{256} = 16$
$\sqrt{16} = 4$
$\sqrt{4} = 2$

$\log_2 ? = 65536 \quad 2^{16} = 65536$

$\log_2 65536 = 16 = O(\log n)$

We require 4 steps instead of 16

$2^4 = 16$

$2^k = n$

$k = \log n$

We require $O(\log k) = O(\log \log n)$ steps

$O(\log \log 65536)$
$= O(\log 16)$
$= 4$

$$\sqrt{n} = n^{\frac{1}{2}}$$

$$\log \sqrt{n} = \log n^{\frac{1}{2}}$$

$$\log \log \sqrt{n} = \log (0.5 \log n)$$

$$\log \log \sqrt{n} = \log(0.5) + \log \log n$$

$$\log \log \sqrt{n} = O(\log \log n)$$

Figure 2.22: O(log log n) algorithm

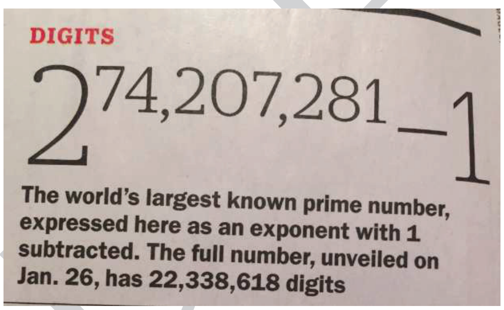
2.3.9 Generating prime numbers

HOW MANY PRIME NUMBERS ARE THERE BETWEEN 1 TO N ?

The prime numbers until 25 are
1 2 3 4 5 6 7 8 9
2, 3, 5, 7, 11, 13, 17, 19, 23

#(P25) = 9

	x	pi(x)
1	10	4
2	100	25
3	1,000	168
4	10,000	1,229
5	100,000	9,592
6	1,000,000	78,498
7	10,000,000	664,579
8	100,000,000	5,761,455
9	1,000,000,000	50,847,534
10	10,000,000,000	455,052,511
11	100,000,000,000	4,118,054,813
12	1,000,000,000,000	37,607,912,018
13	10,000,000,000,000	346,065,536,839
14	100,000,000,000,000	3,204,941,750,802
15	1,000,000,000,000,000	29,844,570,422,669
16	10,000,000,000,000,000	279,238,341,033,925
17	100,000,000,000,000,000	2,623,557,157,654,233
18	1,000,000,000,000,000,000	24,739,954,287,740,860
19	10,000,000,000,000,000,000	234,057,667,276,344,607
20	100,000,000,000,000,000,000	2,220,819,602,560,918,840
21	1,000,000,000,000,000,000,000	21,127,269,486,018,731,928
22	10,000,000,000,000,000,000,000	201,467,286,689,315,906,290
23	100,000,000,000,000,000,000,000	1,925,320,391,606,803,968,923
24	1,000,000,000,000,000,000,000,000	18,435,599,767,349,200,867,866
25	10,000,000,000,000,000,000,000,000	176,846,309,399,143,769,411,680



Jan 26 2016 (Time Jan 2016 issue)

#Primes of n < n
O(n)
but we need a tighter bound

The Prime Number Theorem

#number of prime for n

$$\sim= \frac{n}{(\log_e n) - 1}$$

n	#P(n)	$\frac{n}{(\log_e n)-1}$
100	25	27
1000	168	169
10000	1229	1217
100000	9592	9512
1000000	78498	78030
10000000	654579	661458
100000000	5761455	5740303

$$\frac{n}{(\log_e n)-1} = O\left(\frac{n}{\log_e n}\right)$$

Figure 2.23: Computing numbers of prime numbers

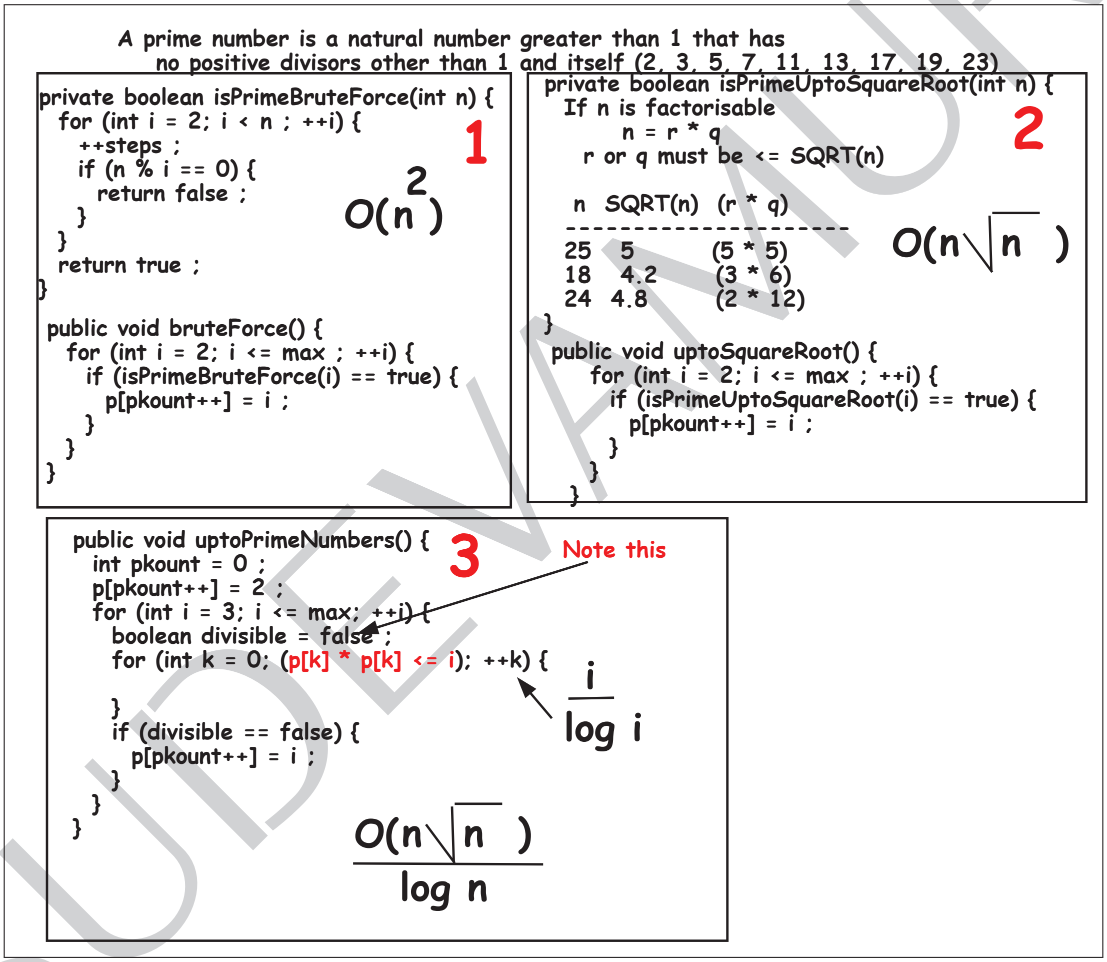


Figure 2.24: Three algorithms for genearting prime numbers

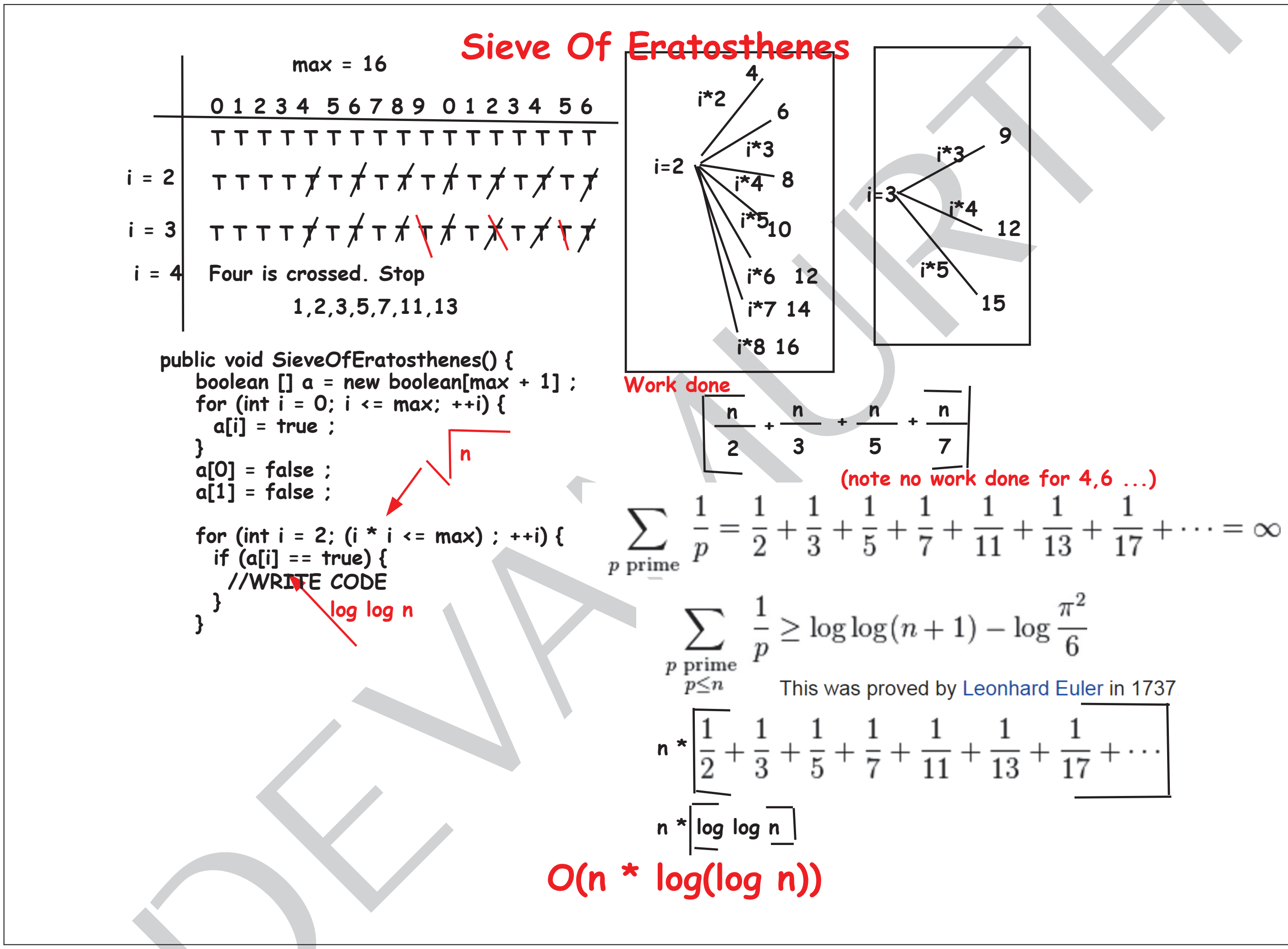


Figure 2.25: Sieve of Eratosthenes algorithm

n	#prime	$O(n^2)$	$O(n\sqrt{n})$	$O(n\sqrt{n}) / \log n$	O(n * log(log n))
16	6	40	17	17	10
1000	168	78022	5288	2801	1411
50000	5133	-	-	313588	93276
500000	41538	-	-	5709008	1033917

Figure 2.26: Number of steps with all the four methods

2.4 Big O Notation

2.4.1 $O()$ definition

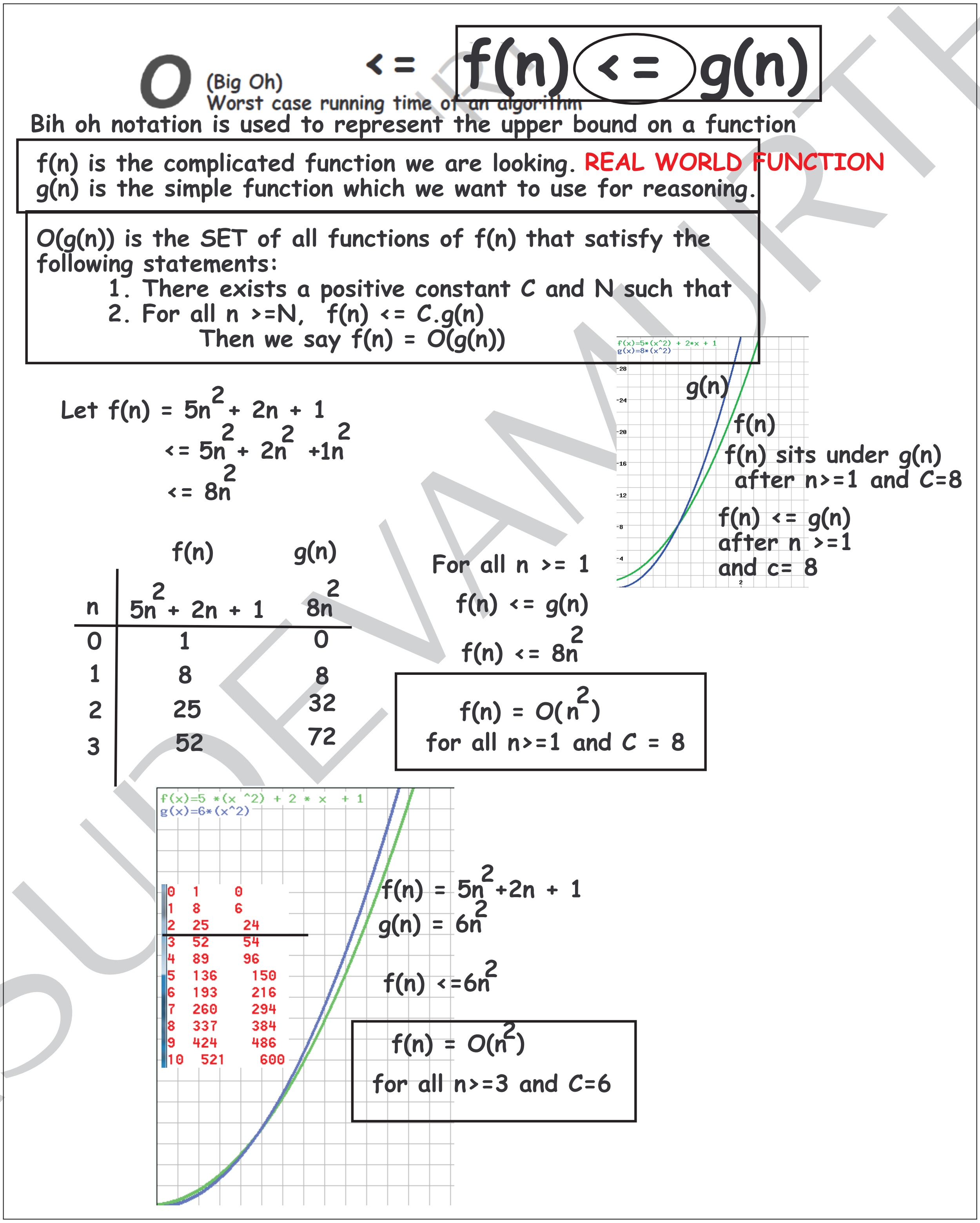


Figure 2.27: $O()$ definition

2.4.1.1 $O()$ Example 1

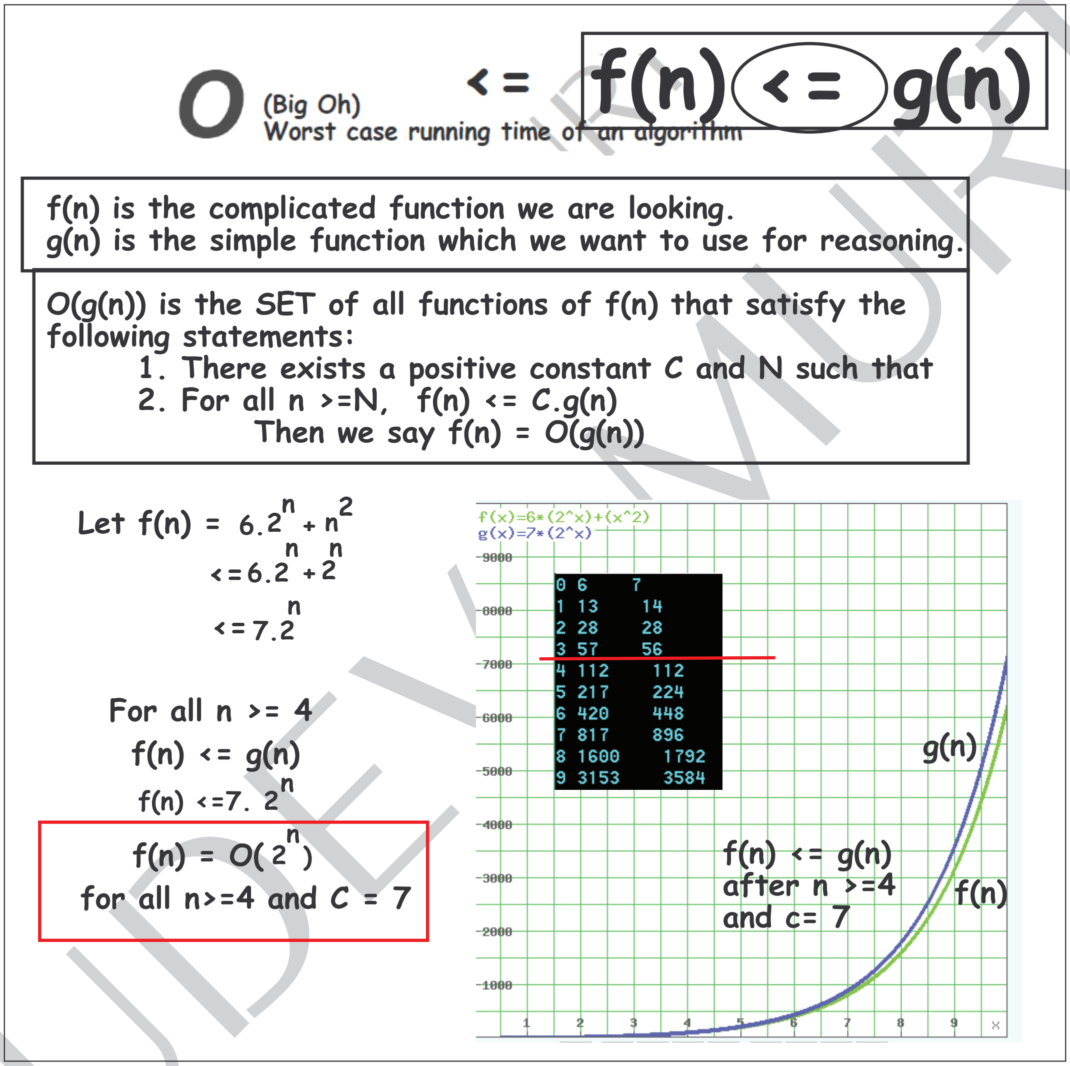


Figure 2.28: $O()$ example

2.4.2 $O()$ Fact 1

O (Big Oh) <=
Worst case running time of an algorithm

Fact1: Big-Oh notation does not care about constant factors in terms of running time

f(n) = 1,000,000 n
<= 1,000,000 n
<= g(n)

$\boxed{\text{million n <= million n. This is always true}}$

f(n) = O(n)
for all n>=0 and C=1,000,000

Complicated 1,000,000 n is written in simplest possible form O(n)

Consider a program written by 3 programmers

n	n/2	n	100n
100	50	100	100000
200	100	200	200000
ratio	2	2	2

For all the programs, as n is doubled from 100 to 200, the growth rate is constant factor 2.

Hence all the 3 algorithms are in O(n)

n : n/2

$\frac{n}{n/2}$ = 2 Times better

Does not depend on n

n : 100n

$\frac{n}{100n}$ =(1/100) worst

n	100n	n^2
1	100	1
5	500	25
10	1000	100
20	2000	400
50	5000	2500
80	8000	6400
100	10000	10000
150	15000	22500
200	20000	40000
1000	100000	1000000

n^2algorithm is better than n algorirhm
until n = 100

After that n^2 algorithms doubles.
It is not a constant factor like above

$\frac{n^2}{100n} = \frac{n}{100}$ worst

depends on n

BIG-OH is an UPPER BOUND ONLY
You can say how FAST your algorithm is,
You cannot say how SLOW is your algorithm

Figure 2.29: $O()$ fact 1

2.4.3 $O()$ Fact 2

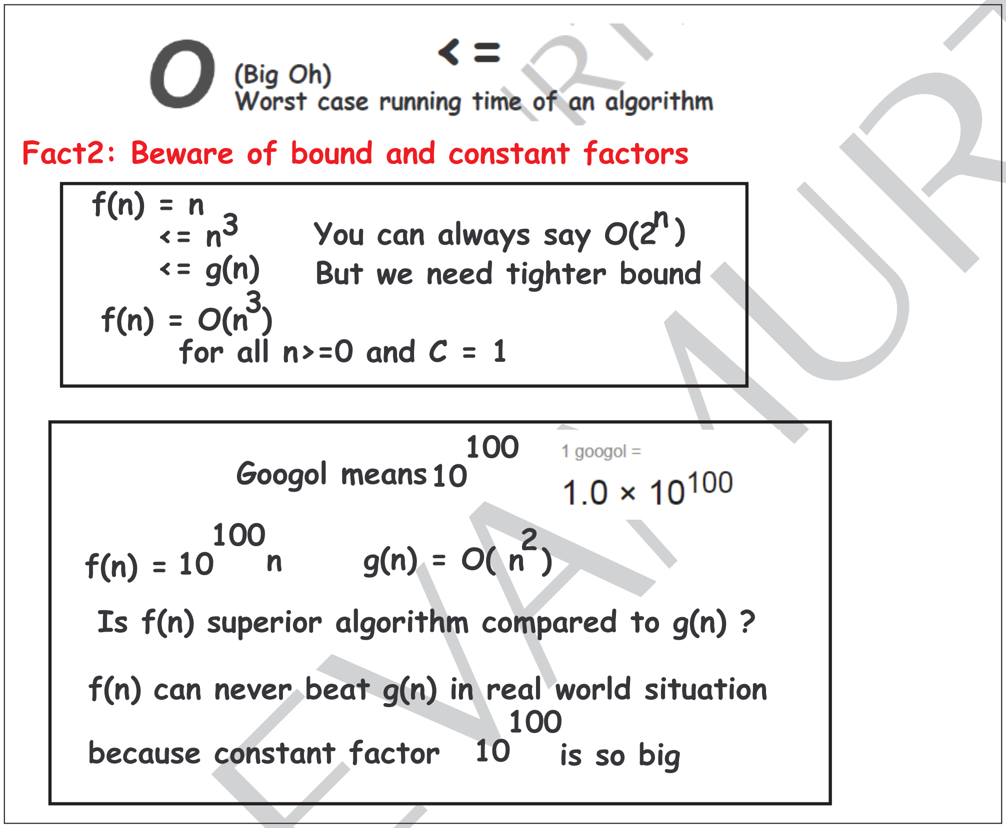


Figure 2.30: $O()$ fact 2

2.4.4 $O()$ Fact 3

Fact3: Big-Oh notation is used to pick dominating terms

$f(n) = n^3 + n^2 + 100\,n + 2000$;

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

As n -> infinity, n^3 dominates, or grow much faster than n^2 and 100 n

We are telling our algorithm is as fast n^3 algorithm

Note that Big-Oh tells how fast is your algorithm.
It cannot say how slow is your algorithm.

Theorem 1:

$f(n) = a0 + a1\ n + a2\ n^2 + a3\ n^3 + \ldots\ldots\ldots\ldots + ak\ n^k$

$\le a0\ n^k + a1\ n^k + a2\ n^k + a3\ n^k + ak\ n^k$

$\le (a0 + a1 + a2 + a3 + \ldots\ldots + ak)\ n^k$ for $n \ge 1$

$\le Cn^k$

$f(n) = O(n^k)$ for $n \ge 1$
and $C = (a0+a1+a2+..+ak)$

Theorem 2;

Let us say say have two phases in your algorithm.
f1(n) = O(g1(n)) Phase1
f2(n) = O(g2(n)) Phase2
then f1(n) + f2(n) = O(max(g1(n).g2(n)) NOTE IT IS NOT SUM

proof:
f1(n) =O(g1(n)) means f1(n) <= C.g1(n) for all n > n1
f2(n) =O(g2(n)) means f2(n) <= D.g2(n) for all n > n2

Let us choose n3 = max(n1,n2);
M = max(C,d) ;

That means for n>=n3, f1+f2 <= C.g1+D.g2
<= M.g1 + M.g2
<= M(g1+g2)
<= M(2.MAX(g1,g2))
<=2M(MAX(g1,g2))

Now C = 2*M
n = n3
For n>=n3, and C= 2*M, f1(n)+f2(n) = O(max(g1(n),g2(n))

This also proves, if you have 'n' phases in your algorithm
the only phase that dominates is the one which takes maximum time.

Overall running time of an algorithm depend on the solving this
BOTTLE NECK phase.

Figure 2.31: $O()$ fact 3

2.4.5 Sum and product rules

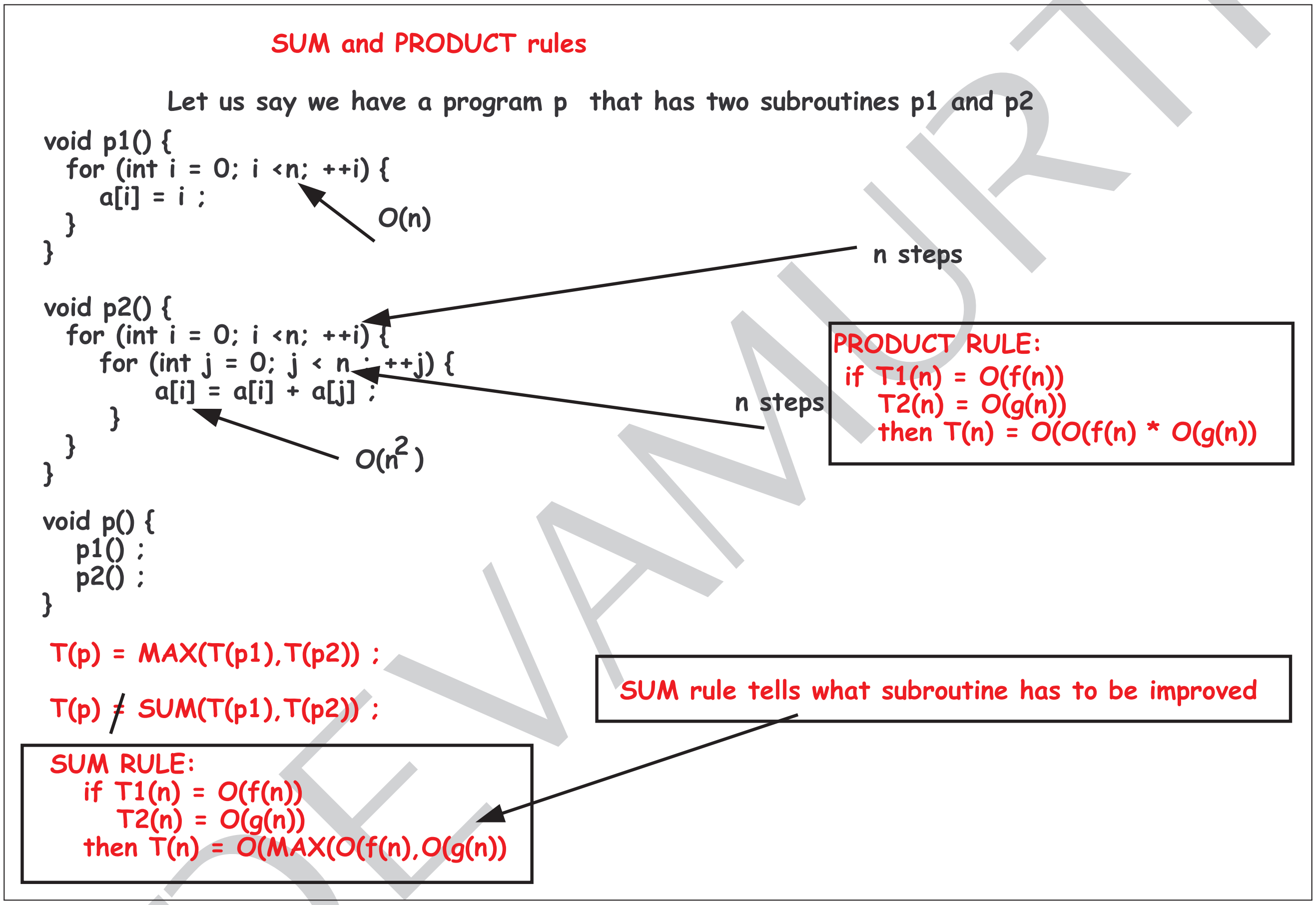


Figure 2.32: Sum and product rules

2.5 Big $\Omega()$ notation

2.5.1 $\Omega()$ definition

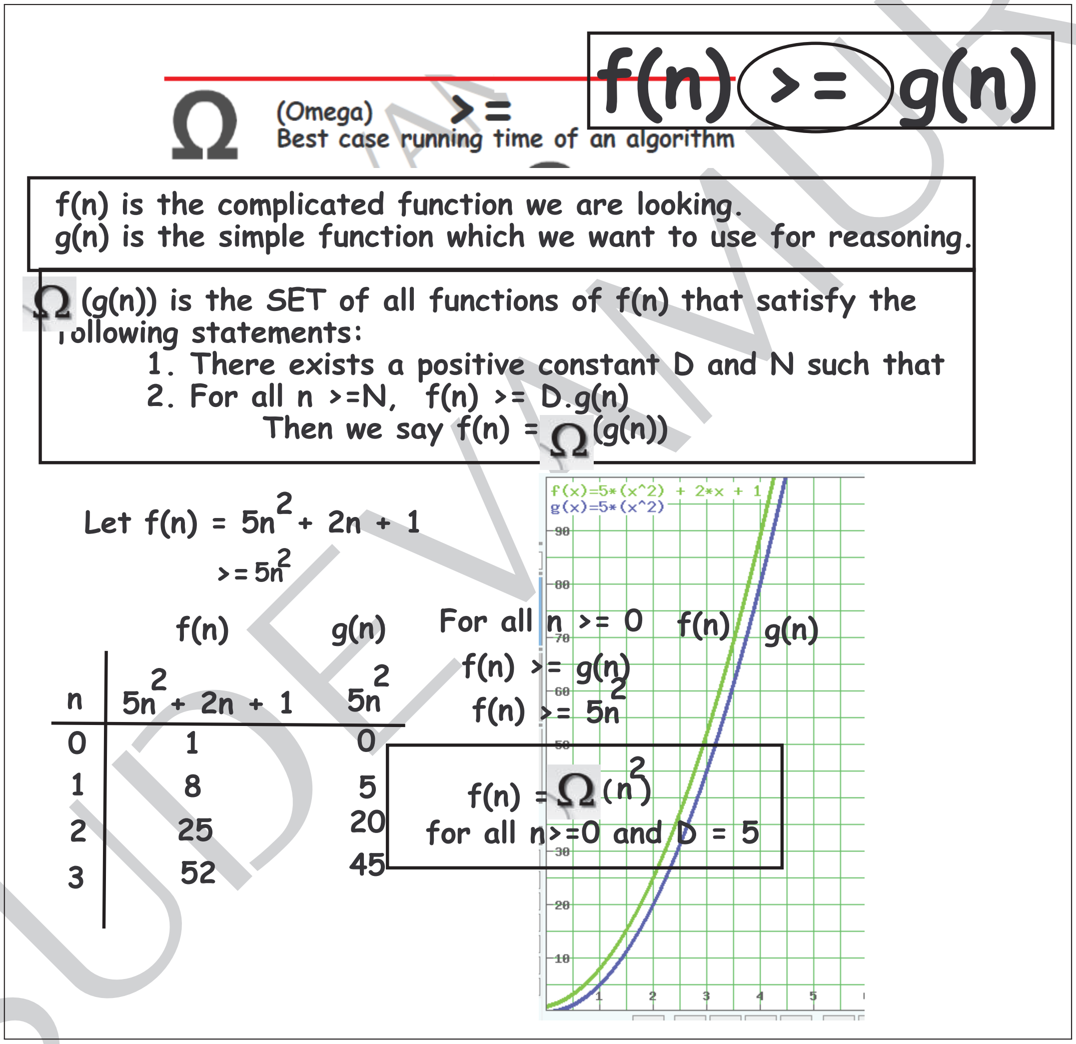


Figure 2.33: Ω() definition

2.6 Big Θ() notation

2.6.1 Θ() definition

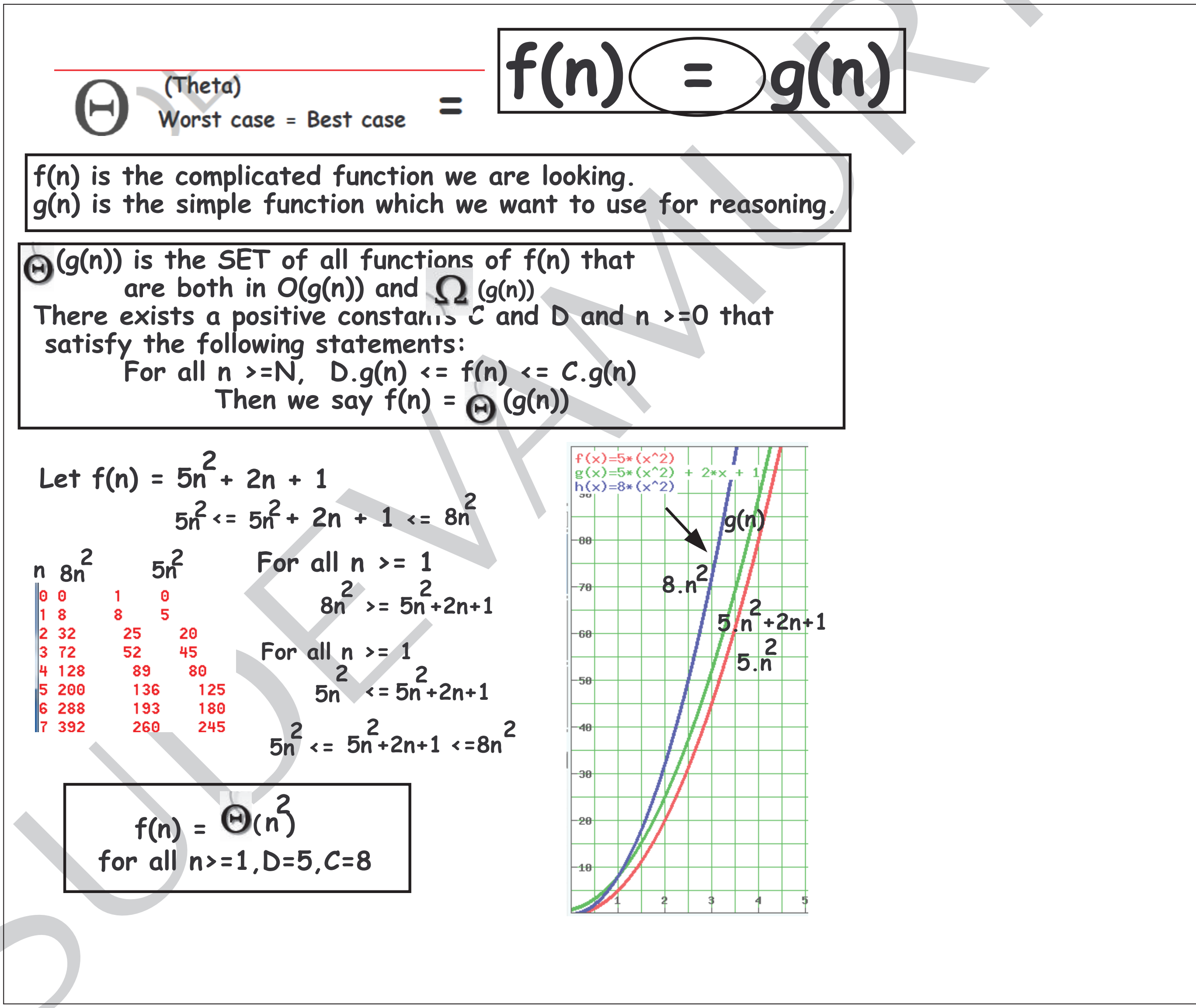


Figure 2.34: Θ() definition

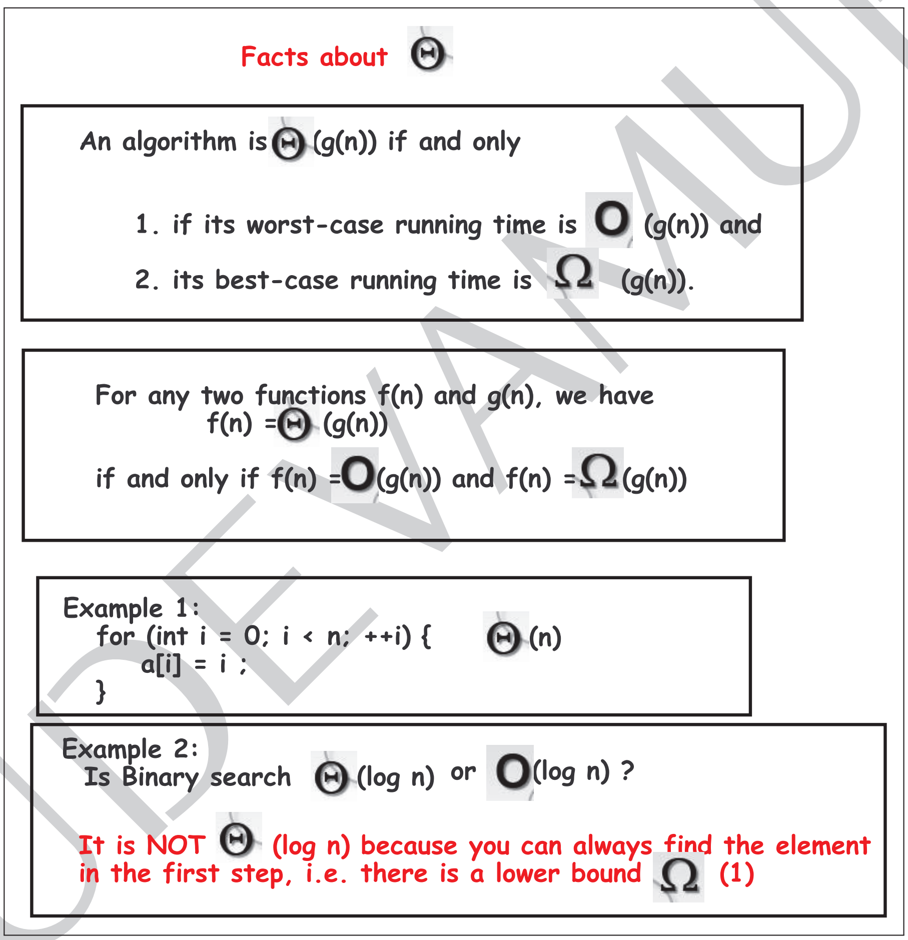


Figure 2.35: Facts about Θ() notation

https://www.khanacademy.org/computing/computer-science/algorithms/asymptotic-notation/

Linear search

set up time

Worst case: c1*n + c2 ;

As we've argued, the constant factor c_1 and the low-order term c_2 don't tell us about the rate of growth of the running time. What's significant is that the worst-case running time of linear search grows like the array size n. The notation we use for this running time is $\Theta(n)$. That's the Greek letter "theta," and we say "big-Theta of n" or just "Theta of n."

Worst case running is THETA(n)
Best case running is THETA(1)
RUNNING TIME: O(n)

https://www.khanacademy.org/computing/computer-science/algorithms/asymptotic-notation/

Binary search

only above. For example, although the worst-case running time of binary search is $\Theta(\lg n)$, it would be incorrect to say that binary search runs in $\Theta(\lg n)$ time in *all* cases. What if we find the target value upon the first guess? Then it runs in $\Theta(1)$ time. The running time of binary search is never worse than $\Theta(\lg n)$, but it's sometimes better. It would be convenient to have a form of asymptotic notation that means "the running time grows at most this much, but it could grow more slowly." We use "big-O" notation for just such occasions.

Binary search worst case: THETA(log n)
Binary search best case: THETA(1)
RUNNING TIME:
It is in correct to say THETA(logn). So we say O(logn)

Printing an array
Worst case: THETA(n)
Best case: THETA(n)
Running time: THETA(n)

Figure 2.36: Facts about $\Theta()$ notation

What is Big Theta? When should I use Big Theta as opposed to big O?

https://www.quora.com/What-is-Big-Theta-When-should-I-use-Big-Theta-as-opposed-to-big-O

Formally, the only place you see bit-Theta is when the complexity is guaranteed and usually only in proofs. Adding two fixed-length numbers, for example, is $\Theta(1)$, no matter what. Printing or modifying every element of an array is $\Theta(n)$.

Informally, you'll generally call everything big-O, even if it isn't. It's technically valid, since big-O is the upper-bound and the upper-bound of a fixed function is obviously that function, but it can sound a little bit awkward.

https://cs.stackexchange.com/questions/23068/how-do-o-and-%CE%A9-relate-to-worst-and-best-case

Consider the following algorithm (or procedure, or piece of code, or whatever):

```
Contrive(n)
1. if n = 0 then do something Theta(n^3)
2. else if n is even then
3.    flip a coin
4.    if heads, do something Theta(n)
5.    else if tails, do something Theta(n^2)
6. else if n is odd then
7.    flip a coin
8.    if heads, do something Theta(n^4)
9.    else if tails, do something Theta(n^5)
```

What is the asymptotic behavior of this function?

In the best case (where n is even), the runtime is $\Omega(n)$ and $O(n^2)$, but not Θ of anything.

In the worst case (where n is odd), the runtime is $\Omega(n^4)$ and $O(n^5)$, but not Θ of anything.

In the case $n = 0$, the runtime is $\Theta(n^3)$.

This is a bit of a contrived example, but only for the purposes of clearly demonstrating the differences between the bound and the case. You could have the distinction become meaningful with completely deterministic procedures, if the activities you're performing don't have any known Θ bounds.

Figure 2.37: Facts about Θ() notation

