Presentation for use with the textbook Data Structures and Algorithms in Java, 6th edition, by M. T. Goodrich, R. Tamassia, and M. H. Goldwasser, Wiley, 2014

Analysis of Algorithms

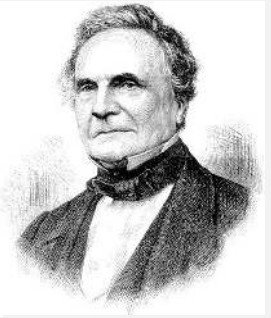
*Important Chapter

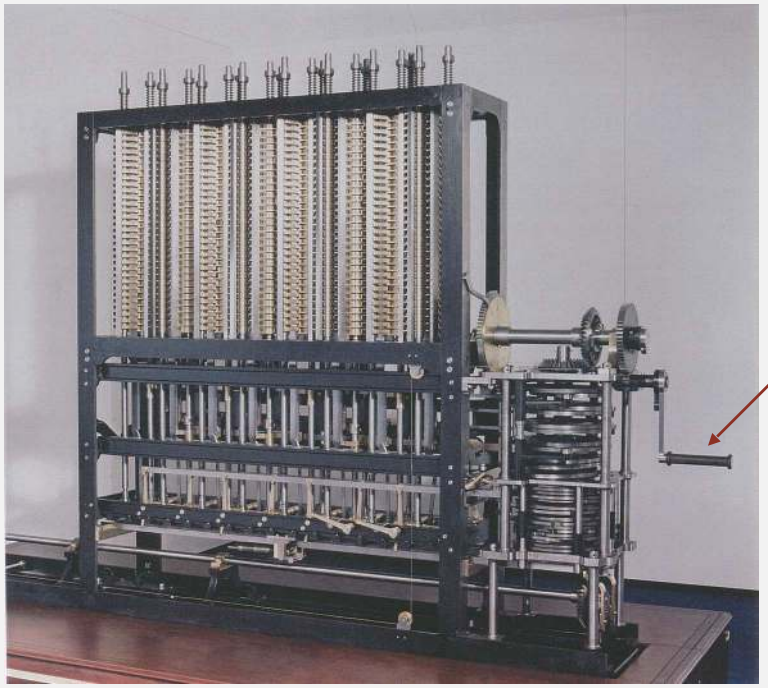
O(2^n)
can کل
بنخسم مرتین



Running time

" As soon as an Analytic Engine exists, it will necessarily guide the future course of the science. Whenever any result is sought by its aid, the question will arise—By what course of calculation can these results be arrived at by the machine in the shortest time? " — *Charles Babbage (1864)*





how many times do you have to turn the crank?

Analytic Engine

Measuring the running time

Q. How to time a program?

A. Manual.



```
% java ThreeSum 1Kints.txt
```



```
70
% java ThreeSum 2Kints.txt
```



```
528
% java ThreeSum 4Kints.txt
```



```
4039
```

Measuring the running time

Q. How to time a program?

A. Automatic.

public class Stopwatch	(part of `stdlib.jar`)
Stopwatch()	*create a new stopwatch*
double elapsedTime()	*time since creation (in seconds)*

```
public static void main(String[] args)
{
   In in = new In(args[0]);
   int[] a = in.readAllInts();
   Stopwatch stopwatch = new Stopwatch();
   StdOut.println(ThreeSum.count(a));
   double time = stopwatch.elapsedTime();
   StdOut.println("elapsed time " + time);
}
```

Measuring the running time

Q. How to time a program?

A. Automatic.

public class Stopwatch		
	Stopwatch()	*create a new stopwatch*
double	elapsedTime()	*time since creation (in seconds)*

```
public class Stopwatch
{
   private final long start = System.currentTimeMillis();

   public double elapsedTime()
   {
      long now = System.currentTimeMillis();
      return (now - start) / 1000.0;
   }
}
```

implementation (part of stdlib.jar)

```
public static String repeat1(char c, int n) {
  String answer = "";
  for (int j=0; j < n; j++)
    answer += c;
  return answer;
}
```

→ begian

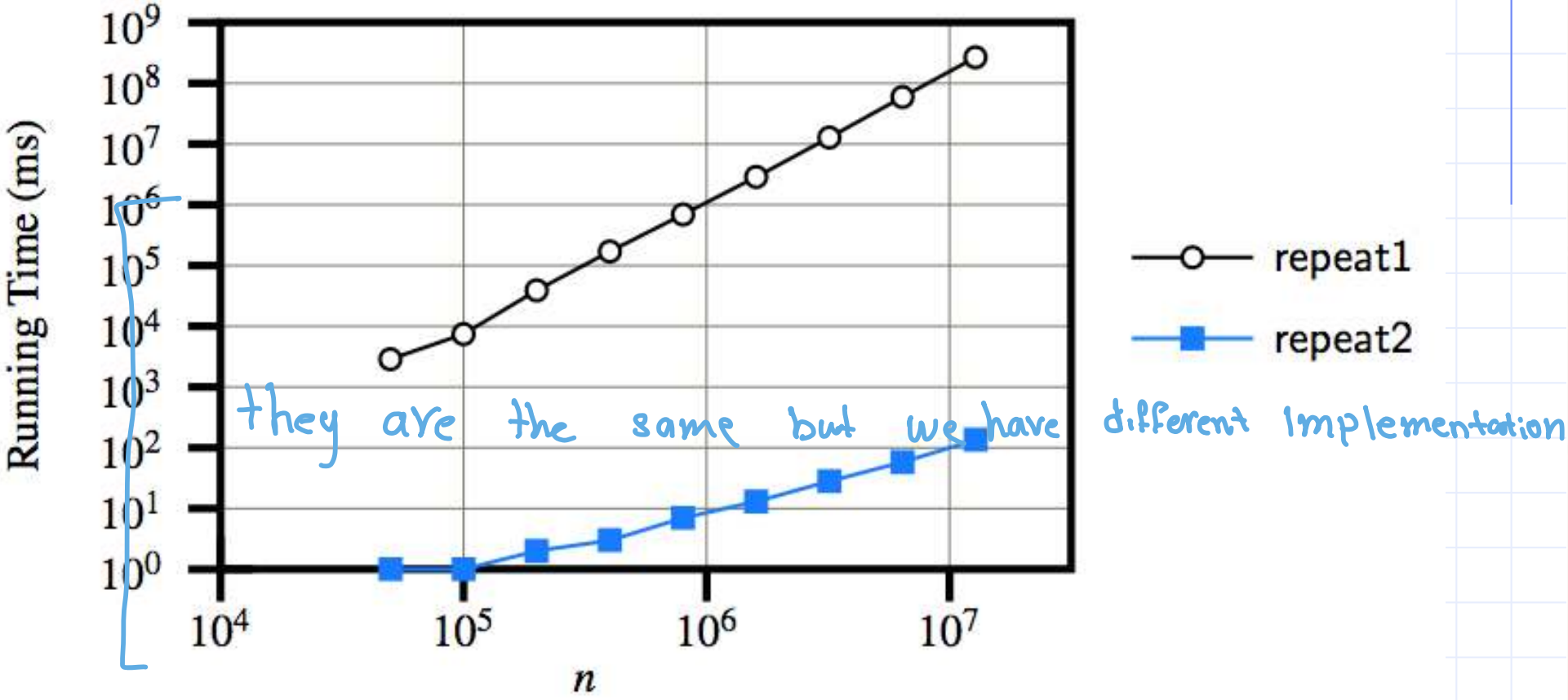
→ end time

end - beg

```
public static String repeat2(char c, int n) {
  StringBuilder sb = new StringBuilder( );
  for (int j=0; j < n; j++)
    sb.append(c);
  return sb.toString( );
}
```

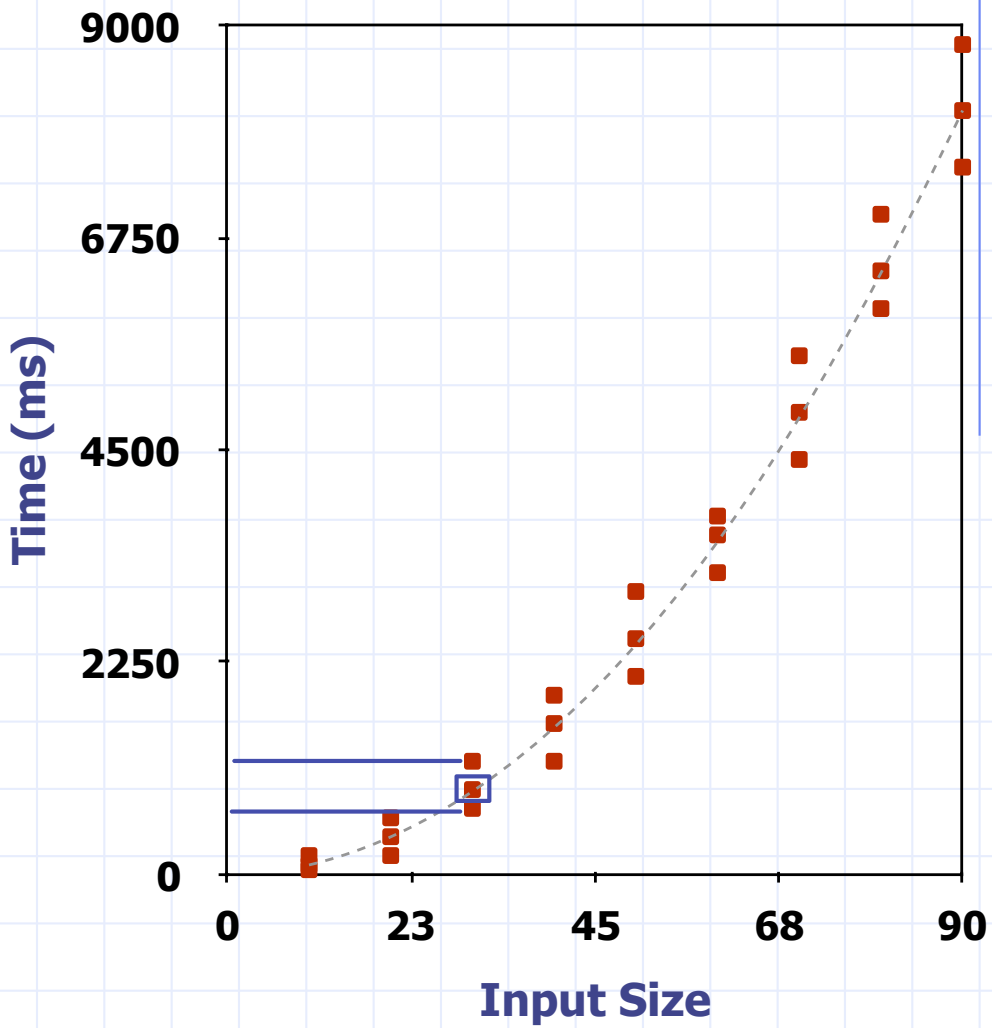
→ جيد اسرع

n	repeat1 (in ms)	repeat2 (in ms)
50,000	2,884	1
100,000	7,437	1
200,000	39,158	2
400,000	170,173	3
800,000	690,836	7
1,600,000	2,874,968	13
3,200,000	12,809,631	28
6,400,000	59,594,275	58
12,800,000	265,696,421	135


Running Time (ms)
10^9
10^8
10^7
10^6
10^5
10^4
10^3
10^2
10^1
10^0
10^4
10^5
10^6
10^7
n
repeat1
repeat2
they are the same but we have different Implementation

Experimental Studies

- Write a program implementing the algorithm
- Run the program with inputs of varying size and composition, noting the time needed:
- Plot the results




```
1  long startTime = System.currentTimeMillis( );      // record the starting time
2  /* (run the algorithm) */
3  long endTime = System.currentTimeMillis( );        // record the ending time
4  long elapsed = endTime − startTime;                // compute the elapsed time
```

Limitations of Experiments

- It is necessary to implement the algorithm, which may be difficult
- Results may not be indicative of the running time on other inputs not included in the experiment.
- In order to compare two algorithms, the same hardware and software environments must be used



Challenges

- Experimental running times of two algorithms are difficult to directly compare unless the experiments are performed in the same hardware and software environments.
- Experiments can be done only on a limited set of test inputs; hence, they leave out the running times of inputs not included in the experiment (and these inputs may be important).
- An algorithm must be fully implemented in order to execute it to study its running time experimentally.

Goals of Algorithm analysis

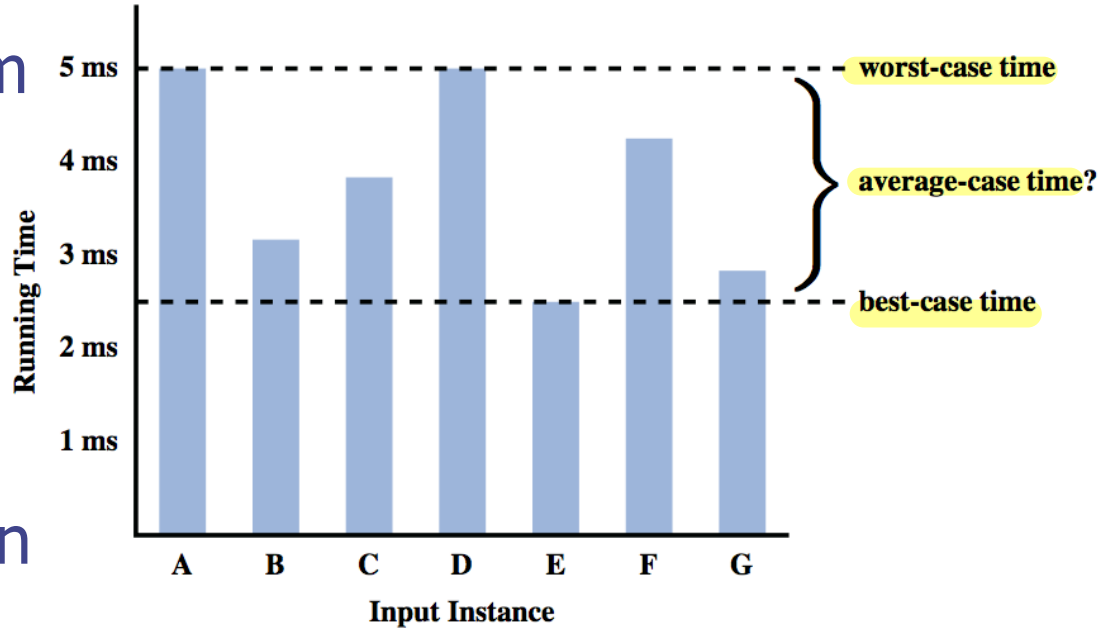
1. Allows us to evaluate the relative efficiency of any two algorithms in a way that is independent of the hardware and software environment.

2. Is performed by studying a high-level description of the algorithm without need for implementation.

3. Takes into account all possible inputs.

Running Time

- Most algorithms transform input objects into output objects.
- The running time of an algorithm typically grows with the input size.
- Average case time is often difficult to determine.
- We focus on the worst case running time.
  - Easier to analyze
  - Crucial to applications such as games, finance and robotics



Theoretical Analysis



مايعتمد على لغه معينه

- Uses a high-level description of the algorithm instead of an implementation
- Characterizes running time as a function of the input size, n
- Takes into account all possible inputs
- Allows us to evaluate the speed of an algorithm independent of the hardware/software environment

Primitive Operations



- Basic computations performed by an algorithm
- Identifiable in pseudocode
- Largely independent from the programming language
- Exact definition not important (we will see why later)
- Assumed to take a constant amount of time in the RAM model

- Examples:
  - Evaluating an expression
  - Assigning a value to a variable `=`
  - Indexing into an array `[ ]`
  - Calling a method
  - Returning from a method

Counting Primitive Operations

- By inspecting the pseudocode, we can determine the maximum number of primitive operations executed by an algorithm, as a function of the input size

```
1  /** Returns the maximum value of a nonempty array of numbers. */
2  public static double arrayMax(double[ ] data) {
3    int n = data.length;
4    double currentMax = data[0];          // assume first entry is biggest (for now)
5    for (int j=1; j < n; j++)             // consider all other entries
6      if (data[j] > currentMax)           // if data[j] is biggest thus far...
7        currentMax = data[j];             // record it as the current max
8    return currentMax;
9  }
```

- Step 3: 2 ops,
  4: 2 ops,
  5: 2n ops,
  6: 2n ops,
  7: 0 to n ops,
  8: 1 op

* عدد التكرارات لمن تبدا من (1) لـ (n) :

Ex :

اذا ابدا من واحد ← n

1- for (i ← 1 ; i <= n ; i++) → 1, 2, 3 → عدد التكرار ← 3 → n

اذا اكبر من او اصغر (بدون مساواه) ← n

2- for (i ← 1 ; i <= n+1 ; i++) → 1, 2, 3, 4 → عدد التكرار ← 4

آخر المساواه تزيد بواحد .

اذا اكبر من او تساوي او اصغر من او تساوي ← (زادت وحدة)

3- for (i ← 0 ; i < n+1 ; i++) → 0, 1, 2, 3 → صناما يشتغل

بدن فيه صفر وتزيده من قدام .

4- for (i ← 0 ; i <= n+2 ; i++) → 0, 1, 2, 3, 4

(بدن خيه مساوه و صفر)

* عدد التكرارات لمن تبدا من (0) لـ (n) :

بتزيد واحد

* الفكرة :

Body of the loop : ← البودي كم مره اشتغل

header تبع اللوب اكيد فيه زياده وحدة عشان تست ما بتدخل او ماتتوقف

1- n+1 ←

دائم البودي تبع اللوب اقل منه بواحد من اللوب (الاصلي)

2 n ← الاربعه بتعمل التست عشان ماداح تشتغل + فيه مساواه

Cost of basic operations

Challenge. How to estimate constants.

operation	example	nanoseconds †
integer add	a + b	2.1
integer multiply	a * b	2.4
integer divide	a / b	5.4
floating-point add	a + b	4.6
floating-point multiply	a * b	4.2
floating-point divide	a / b	13.5
sine	Math.sin(theta)	91.3
arctangent	Math.atan2(y, x)	129
...	...	...

† Running OS X on Macbook Pro 2.2GHz with 2GB RAM

Cost of basic operations

Observation. Most primitive operations take constant time.

→ WCS

operation	example	nanoseconds †
variable declaration	int a	c_1
assignment statement	a = b	c_2
integer compare	a < b	c_3
array element access	a[i]	c_4
array length	a.length	c_5
1D array allocation	new int[N]	$c_6 N$
2D array allocation	new int[N][N]	$c_7 N^2$

Caveat. Non-primitive operations often take more than constant time.

novice mistake: abusive string concatenation

Example: 1-SUM

Q. How many instructions as a function of input size N?

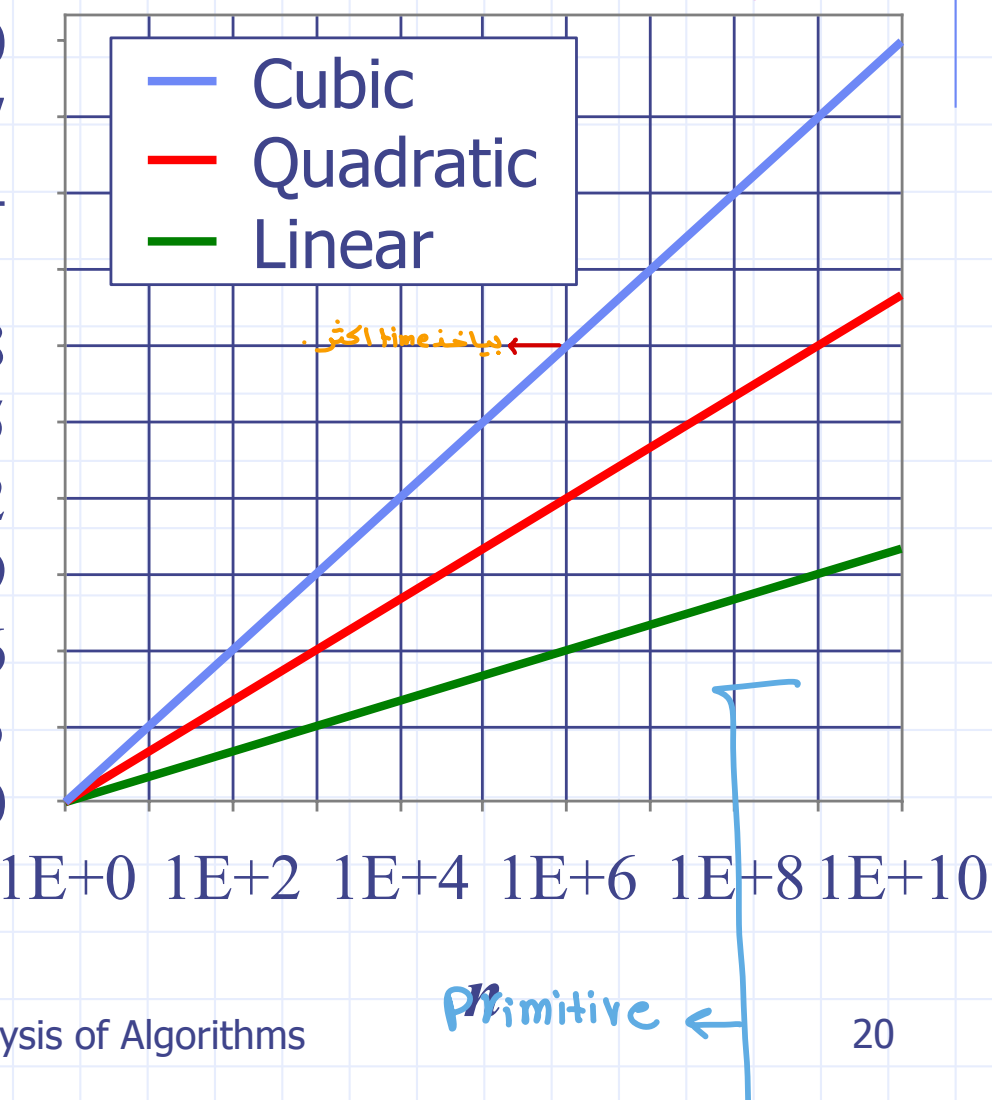
```
int count = 0;
for (int i = 0; i < N; i++)
    if (a[i] == 0)
        count++;
```

N array accesses

operation	frequency
variable declaration	2
assignment statement	2
less than compare	$N + 1$
equal to compare	N
array access	N
increment	N to $2\,N$

Seven Important Functions

- Seven functions that often appear in algorithm analysis:
  - Constant ≈ 1
  - Logarithmic ≈ log n
  - Linear ≈ n
  - N-Log-N ≈ n log n
  - Quadratic ≈ n^2
  - Cubic ≈ n^3
  - Exponential ≈ 2^n
- In a log-log chart, the slope of the line corresponds to the growth rate

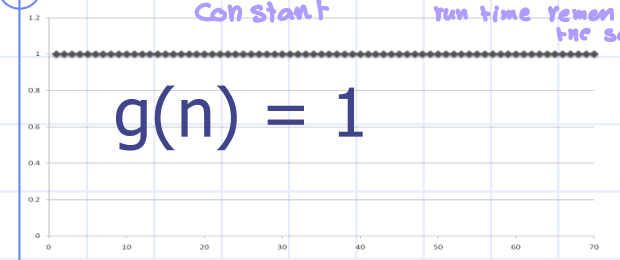


Functions Graphed Using "Normal" Scale

Slide by Matt Stallmann included with permission.

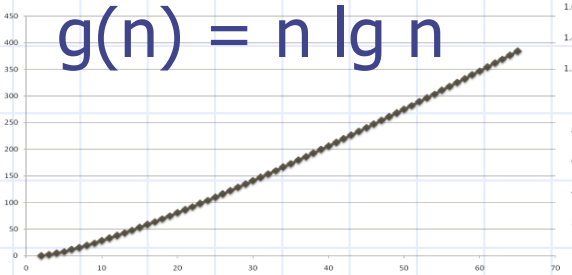
Constant

run time remain the same

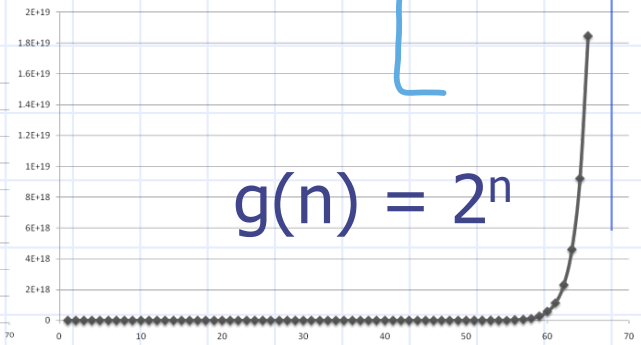


$g(n) = 1$

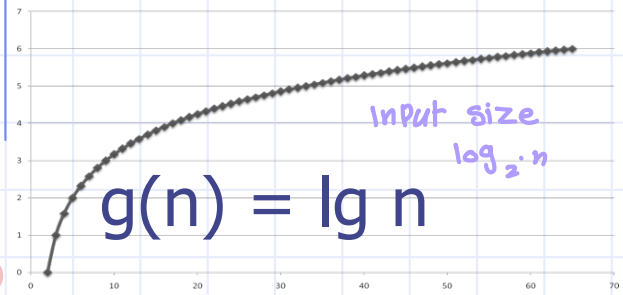
ex: merge sort



$g(n) = n \lg n$



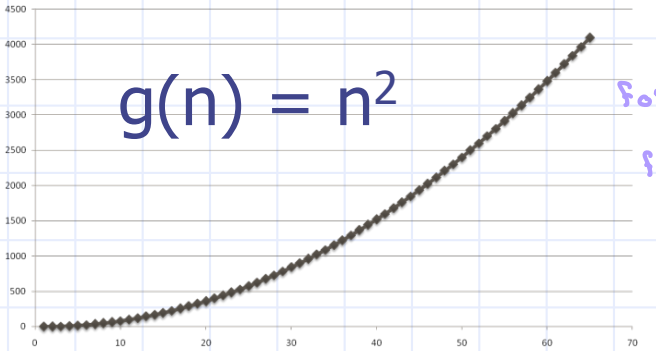
$g(n) = 2^n$



input size

$\log_2 n$

$g(n) = \lg n$



$g(n) = n^2$

```
for (i = 0 ..... ) → n
for (j = 0 ..... ) → n
```

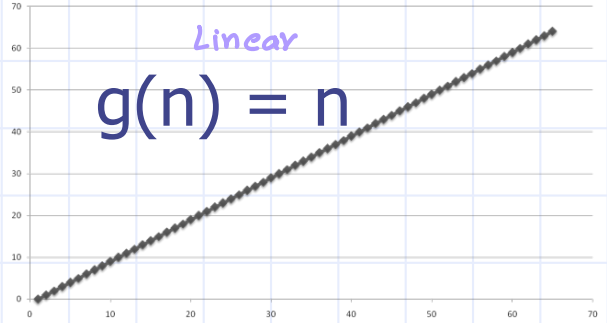
insertion sort

creat 2D array.

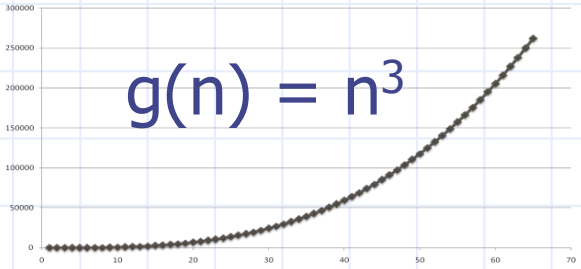
min

max

Linear



$g(n) = n$



$g(n) = n^3$

for .... → n

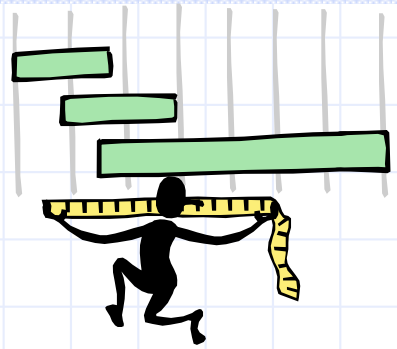
for .... → n

for .... → n

Common order-of-growth classifications

order of growth	name	typical code framework	description	example	$T(2N) / T(N)$
1	**constant**	`a = b + c;`	statement	add two numbers	1
$\log N$	**logarithmic**	`while (N > 1) { N = N / 2; ... }`	divide in half	binary search	~ 1
N	**linear**	`for (int i = 0; i < N; i++) { ... }`	loop	find the maximum	2
$N \log N$	**linearithmic**	[see mergesort lecture]	divide and conquer	mergesort	~ 2
N^2	**quadratic**	`for (int i = 0; i < N; i++) for (int j = 0; j < N; j++) { ... }`	double loop	check all pairs	4
N^3	**cubic**	`for (int i = 0; i < N; i++) for (int j = 0; j < N; j++) for (int k = 0; k < N; k++) { ... }`	triple loop	check all triples	8
2^N	**exponential**	[see combinatorial search lecture]	exhaustive search	check all subsets	$T(N)$

Estimating Running Time



- Algorithm arrayMax executes $5n + 5$ primitive operations in the worst case, $4n + 5$ in the best case. Define:
  - a = Time taken by the fastest primitive operation
  - b = Time taken by the slowest primitive operation
- Let $T(n)$ be worst-case time of arrayMax. Then

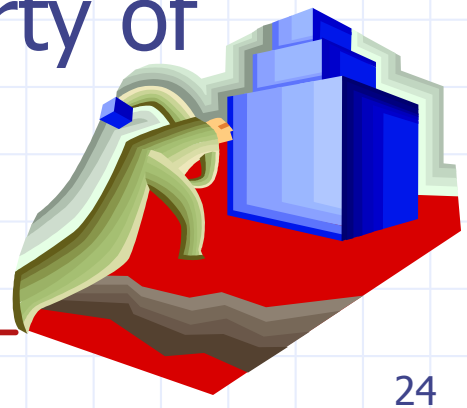
$$a\,(4n + 5) \leq T(n) \leq b(5n + 5)$$

- Hence, the running time $T(n)$ is bounded by two linear functions

Growth Rate of Running Time

- Changing the hardware/ software environment
  - Affects $T(n)$ by a constant factor, but
  - Does not alter the growth rate of $T(n)$
- The linear growth rate of the running time $T(n)$ is an intrinsic property of algorithm arrayMax

array Max : $5+7n$
↳ Linear



Slide by Matt Stallmann included with permission.

Why Growth Rate Matters

if runtime is...	time for n + 1	time for 2 n	time for 4 n
c lg n	c lg (n + 1)	c (lg n + 1)	c(lg n + 2)
c n	c (n + 1)	2c n	4c n
c n lg n	~ c n lg n + c n	2c n lg n + 2cn	4c n lg n + 4cn
c n^2	~ c n^2 + 2c n	**4c n^2**	16c n^2
c n^3	~ c n^3 + 3c n^2	8c n^3	64c n^3
c 2^n	c 2^{n+1}	c 2^{2n}	c 2^{4n}

runtime quadruples when problem size doubles

Practical implications of order-of-growth

growth rate	problem size solvable in minutes			
	1970s	1980s	1990s	2000s
1	any	any	any	any
log N	any	any	any	any
N	millions	tens of millions	hundreds of millions	billions
N log N	hundreds of thousands	millions	millions	hundreds of millions
N^2	hundreds	thousand	thousands	tens of thousands
N^3	hundred	hundreds	thousand	thousands
2^N	20	20s	20s	30

Bottom line. Need linear or linearithmic alg to keep pace with Moore's law.

Practical implications of order-of-growth

growth rate	problem size solvable in minutes				time to process millions of inputs			
	1970s	1980s	1990s	2000s	1970s	1980s	1990s	2000s
1	any	any	any	any	instant	instant	instant	instant
log N	any	any	any	any	instant	instant	instant	instant
N	millions	tens of millions	hundreds of millions	billions	minutes	seconds	second	instant
N log N	hundreds of thousands	millions	millions	hundreds of millions	hour	minutes	tens of seconds	seconds
N^2	hundreds	thousand	thousands	tens of thousands	decades	years	months	weeks
N^3	hundred	hundreds	thousand	thousands	never	never	never	millennia

run time remain the same.

Practical implications of order-of-growth

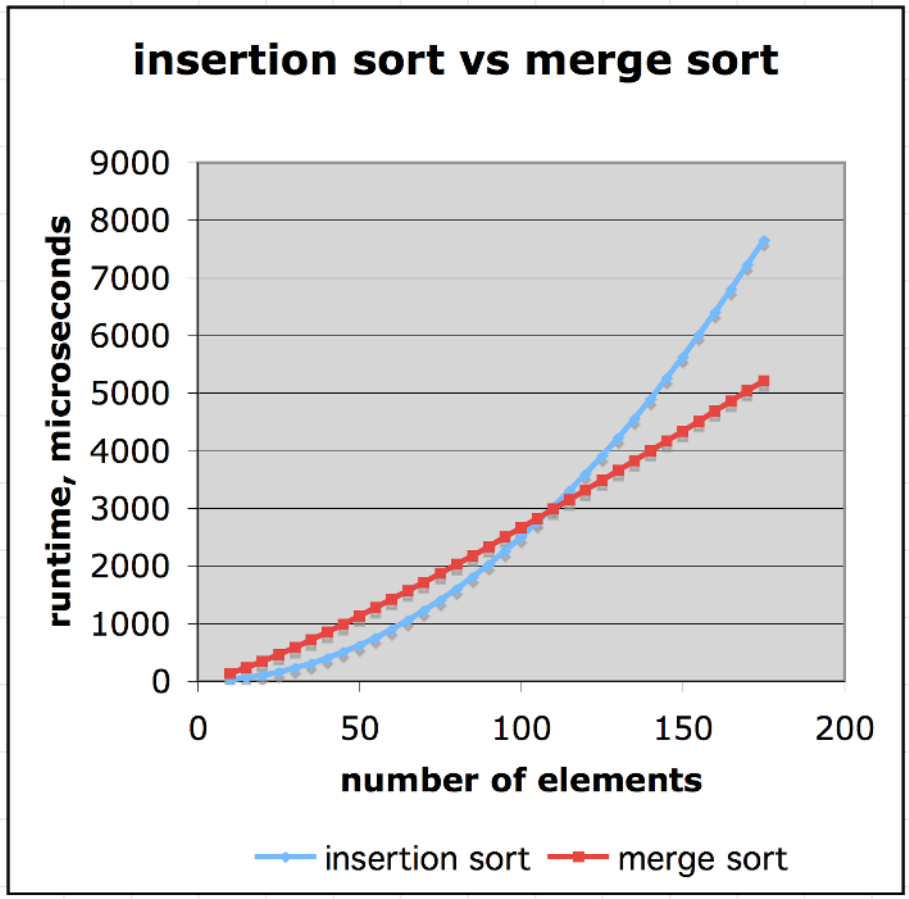
growth rate	name	description	effect on a program that runs for a few seconds	
			time for 100x more data	size for 100x faster computer
1	constant	independent of input size	–	–
log N	logarithmic	nearly independent of input size	–	–
N	linear	optimal for N inputs	a few minutes	100x
N log N	linearithmic	nearly optimal for N inputs	a few minutes	100x
N^2	quadratic	not practical for large problems	several hours	10x
N^3	cubic	not practical for medium problems	several weeks	4–5x
2^N	exponential	useful only for tiny problems	forever	1x

Slide by Matt Stallmann
included with permission.

Comparison of Two Algorithms

insert → n^2

merge sort → $n \log_2 n$



insertion sort is
$n^2 / 4$

merge sort is → recurgen ⇄
$2 n \lg n$

sort a million items?

insertion sort takes
roughly 70 hours

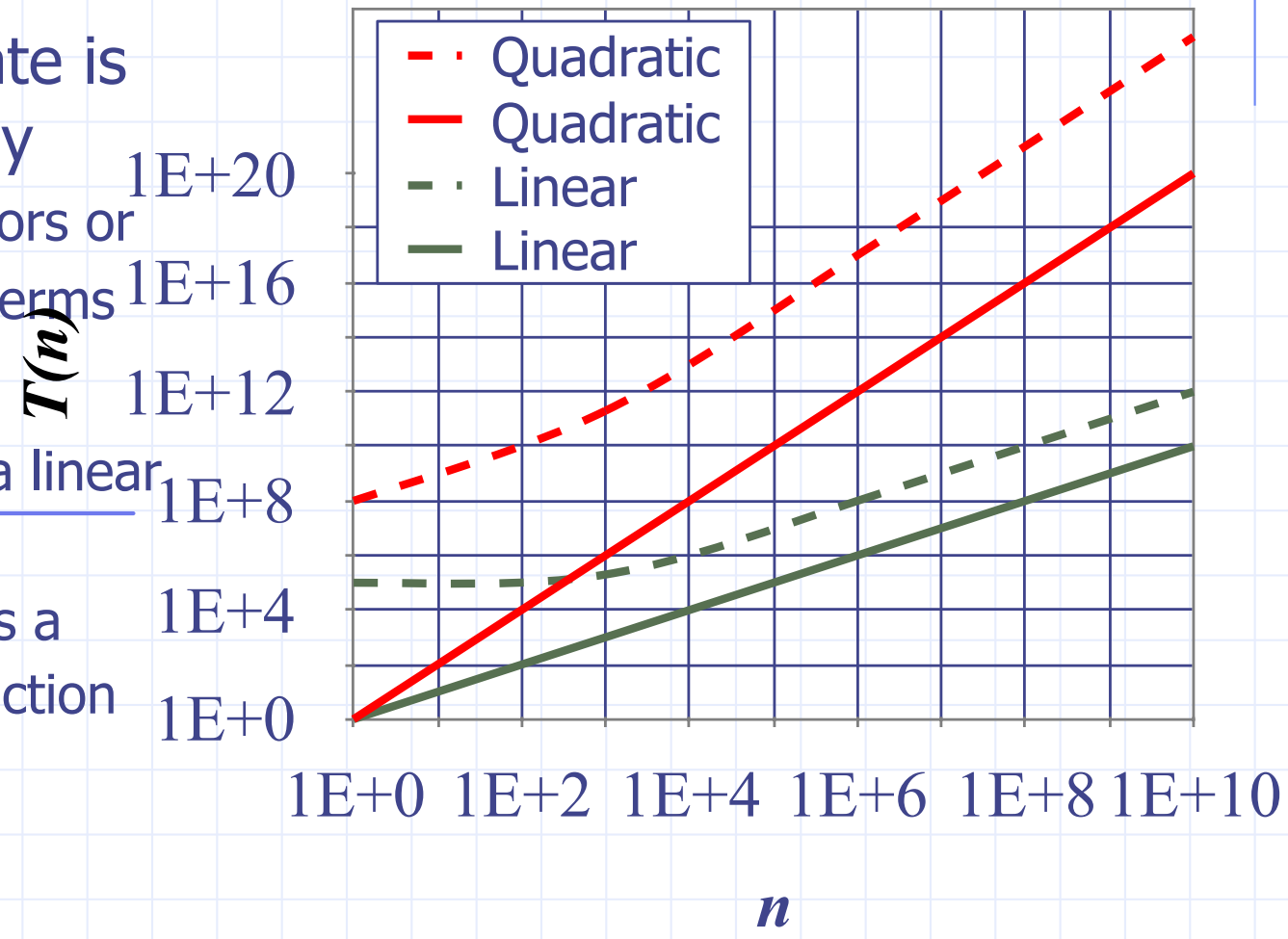
while

merge sort takes
roughly 40 seconds

This is a slow machine, but if
100 x as fast then it's 40 minutes
versus less than 0.5 seconds

Constant Factors

- The growth rate is not affected by
  - constant factors or
  - lower-order terms
- Examples
  - $10^2 n + 10^5$ is a linear function
  - $10^5 n^2 + 10^8 n$ is a quadratic function



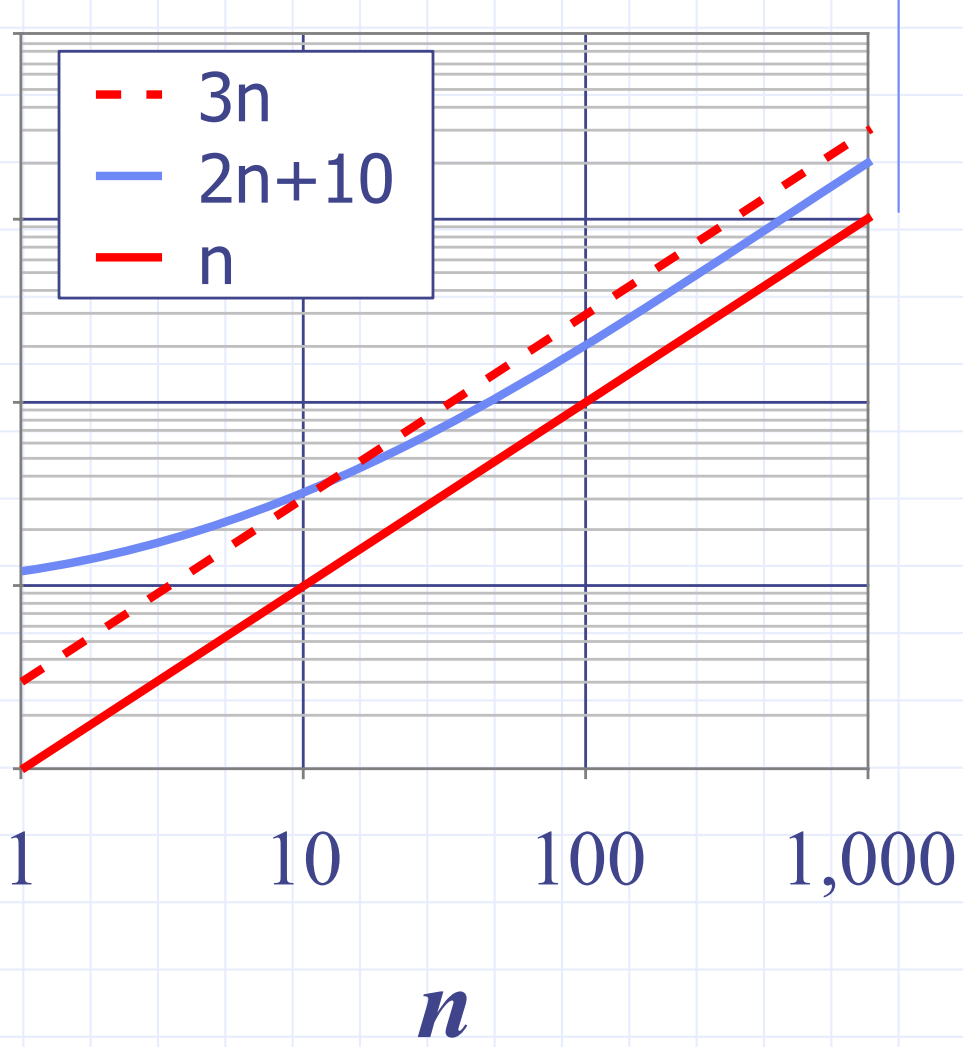
T(n) = 5+7n

Big-Oh Notation

- Given functions $f(n)$ and $g(n)$, we say that $f(n)$ is $O(g(n))$ if there are positive constants c and n_0 such that

  $f(n) \leq cg(n)$ for $n \geq n_0$

- Example: $2n + 10$ is $O(n)$
  - $2n + 10 \leq cn$
  - $(c - 2)\, n \geq 10$
  - $n \geq 10/(c - 2)$
  - Pick $c = 3$ and $n_0 = 10$



A

a	c	x	e

- Search ('a') : 1 time → B.c.s
- Search ('x') : ≈ 1/2 of the input time → A.cc
- Search ('y') : input size time → w.c.s

---

f(n) is o(g(n))

f(n) = 2n + 10 g(n) = n

$f(n) \leq c \cdot g(n)$, $c > 1$, $n \geq 1$

$2n + 10 \leq c \cdot n$

$10 \leq c \cdot n - 2n$

$10 \leq n(c-2)$

$c = 3$ $10 \leq n$

or

$3n + 10 \leq cn$ $c = 12$, $n = 1$

$2n + 10 \leq 12n$

$2(1) + 10 \leq 12(1)$

$12 \leq 12$ ✓

Ex: if f(n) = 3n + 2 and g(n) = n

then Prove f(n) = o(g(n)) ?

$f(n) \leq c \cdot g(n)$

$3n + 2 \leq c \cdot n$

↳ زود واحد

$3n + 2 \leq 4n$

$-3n \quad -3n$

$2 \leq 4n - 3n$

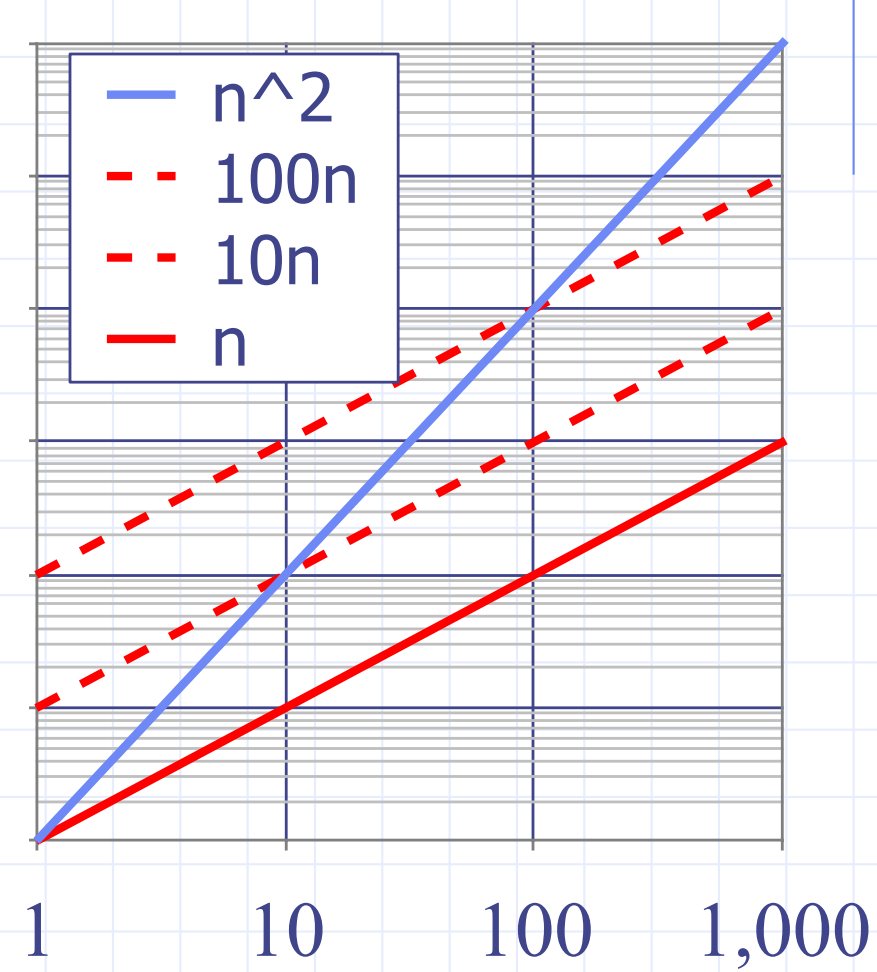
$2 \leq n$

This is true for $c = 4$ and $n_0 = 2$

f(n) = n² , g(n) = n

Big-Oh Example

- Example: the function n^2 is not $O(n)$
  - $n^2 \leq cn$ → not constant
  - $n \leq c$ (some constant)
  - The above inequality cannot be satisfied since c must be a constant



Constant or O(1) Complexity

- Example for Constant function O(1)

```
# O(1)
num = 10

def deductOne(num):
    num -= 1
    return num
```

```
# O(1)
num = 100

def deductOne(num):
    num -= 1
    return num
```

- Result

```
O(1):
9
```

```
O(1):
99
```

- Regardless of how big the input is, only one single conatatn operation is performed by algorithm that is num - 1

Link: https://www.youtube.com/watch?v=0PPR5RWhoFI

الوقت ثابت لانه ماخذ loop .

Log-n or O(log n) Complexity

- Example for Constant function O(logn)

```
# O(log n)
num = 10

def divide(num):
    while num > 1:
        num /= 2
        print(num)
    return num

print "O(log n): "
divide(num)
```

```
# O(log n)
num = 100

def divide(num):
    while num > 1:
        num /= 2
        print(num)
    return num

print "O(log n): "
divide(num)
```

- Result

```
O(log n):
5
2
1
```

time

Time Complexity is 3

```
O(log n):
50
25
12
6
3
1
```

time

Time Complexity is 6

Link: https://www.youtube.com/watch?v=0PPR5RWhoFI

لازم نشوف اول الـ counter نشوف التغير

الـ loop اللي فيها ضرب او قسمة

$\log_2 n$

Linear-n or O(n) Complexity

Example for Constant function O(n)

```
# O(n)
num = 10

def addOnesToTestList(num):
    testList = []
    for i in range(0, num):
        testList.append(1)
        print(testList)
    return testList

print "O(n): "
addOnesToTestList(num)
```

Result

```
O(n):
[1]
[1, 1]
[1, 1, 1]
[1, 1, 1, 1]
[1, 1, 1, 1, 1]
[1, 1, 1, 1, 1, 1]
[1, 1, 1, 1, 1, 1, 1]
[1, 1, 1, 1, 1, 1, 1, 1]
[1, 1, 1, 1, 1, 1, 1, 1, 1]
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
```

Tester Part

Input num is 10 and Complexity is also 10

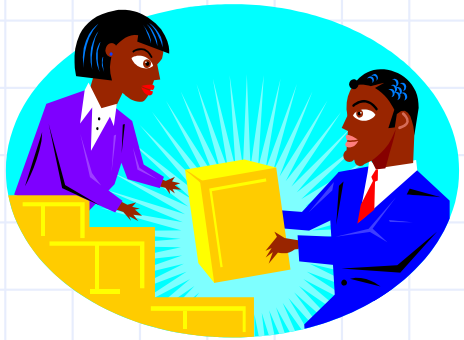
Following are more complex:

- N-Log-N **O(n log n) -> Complexity**
- Quadratic **O(n²) / O(n * n) -> Complexity**
- Cubic **O(n ** 3) -> Complexity**
- Exponential **O(n**n) -> Complexity**

Link: https://www.youtube.com/watch?v=D9PRSIWhoFI

ال loop بيعمل عليها انزيش .

More Big-Oh Examples



- $7n - 2$

  $7n-2$ is $O(n)$

  need $c > 0$ and $n_0 \geq 1$ such that $7n - 2 \leq cn$ for $n \geq n_0$

  this is true for $c = 7$ and $n_0 = 1$

$7n-2 \leq cn$

$7n-2 \leq 8n \rightarrow -2 \leq 8n-7n \rightarrow -2 \leq n$

- $3n^3 + 20n^2 + 5$

  $3n^3 + 20n^2 + 5$ is $O(n^3)$

  need $c > 0$ and $n_0 \geq 1$ such that $3n^3 + 20n^2 + 5 \leq cn^3$ for $n \geq n_0$

  this is true for $c = 4$ and $n_0 = 21$

$3n^3 + 20n^2 + 5 \leq cn^3$

$20n^2 + 5 \leq 4n^3 - 3n^2$

- $3 \log n + 5$

  $3 \log n + 5$ is $O(\log n)$

  need $c > 0$ and $n_0 \geq 1$ such that $3 \log n + 5 \leq c \log n$ for $n \geq n_0$

  this is true for $c = 8$ and $n_0 = 2$

$3 \log n + 5 \leq 8 \log n$

$8 \leq 8 \log n \quad , \quad 1 \leq \log n$

$n = 2$

عندنا جزء قبل الاختيار (c) يا ازيد اول رقم او اجمع كل الارقام الموجودة.

$3n^3 + 20n^2 + 5$ is $o(n^3)$

$f(n)$ $g(n)$

$c > 1$
$n \geq 1$

$f(n) \leq c(g(n))$

$3n^3 + 20n^2 + 5 \leq cn^3$

$5 \leq cn^3 - 3n^3 - 20n^2$

$5 \leq n^3(c-3) - 20n^2$

$c = 4$

$5 \leq n^3 - 20n^2$

T

$n = 1 \rightarrow 5 \leq (1)^3 - 20(1)^2$
$5 \leq -19$ ✗

$n = 2 \rightarrow 5 \leq (2)^3 - 20(2)^2$
$8 - 80$

$n = 10 \rightarrow 5 \leq (10^3) - 20(100)$
$1000 - 2000$
$5 \leq -1000$

$n = 20$ $5 \leq 8000 - 20(20)^2$
$5 \leq 8000 - 8000$

$3\log_2 n + 5$ is $o(\log_2 n)$

$f(n) = 3\log_2 n + 5$, $g(n) = \log_2$

$f(n) <$, $o(g(n))$

$f(n) \leq c.(g(n))$

* $2^3 = 8 \rightarrow \log_2 8 = 3$

Example: Find big-oh

$16^5 n^2 + 16^2 n$

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

$10n^2 + 4n + 2 = O(n^2)$

$f(n) \quad g(n)$

$f(n) \leq c \cdot g(n)$

$10n^2 + 4n + 2 \leq cn^2$

لـ نزود واحد

$10n^2 + 4n + 2 \leq 11n^2$

$4n + 2 \leq 11n^2 - 10n^2$

$4n + 2 \leq n^2$

this true for $n_0 = 5$ call, $n \geq 5$

Big-Oh and Growth Rate

- The big-Oh notation gives an upper bound on the growth rate of a function
- The statement "$f(n)$ is $O(g(n))$" means that the growth rate of $f(n)$ is no more than the growth rate of $g(n)$
- We can use the big-Oh notation to rank functions according to their growth rate

	$f(n)$ is $O(g(n))$	$g(n)$ is $O(f(n))$
$g(n)$ grows more	Yes	No
$f(n)$ grows more	No	Yes
Same growth	Yes	Yes

Big-Oh Rules



معلومه مهمة:
big-oh لازم يكون صو حد من الحدود متساوي واحد من الموجودين في المعادله أو اكبر منة.

- If is $f(n)$ a polynomial of degree d, then $f(n)$ is $O(n^d)$, i.e.,
  1. Drop lower-order terms → الحدود الصغيرة بعملها.
  2. Drop constant factors → وناخذ الحد الكبير
- Use the smallest possible class of functions
  - Say "$2n$ is $O(n)$" instead of "$2n$ is $O(n^2)$"
- Use the simplest expression of the class
  - Say "$3n + 5$ is $O(n)$" instead of "$3n + 5$ is $O(3n)$"

القاعدة بتقول:
اذا اخترت function من هون وهو (n).
وقلت الـ O(n) بتخبط. بس العكس مش صحيح
اختار (n) واعطيك حد اصغر منه واقول هذا big-oh خطا لازم يساوية او اكبر منه ويخفل دائم يساوية.

Asymptotic Algorithm Analysis

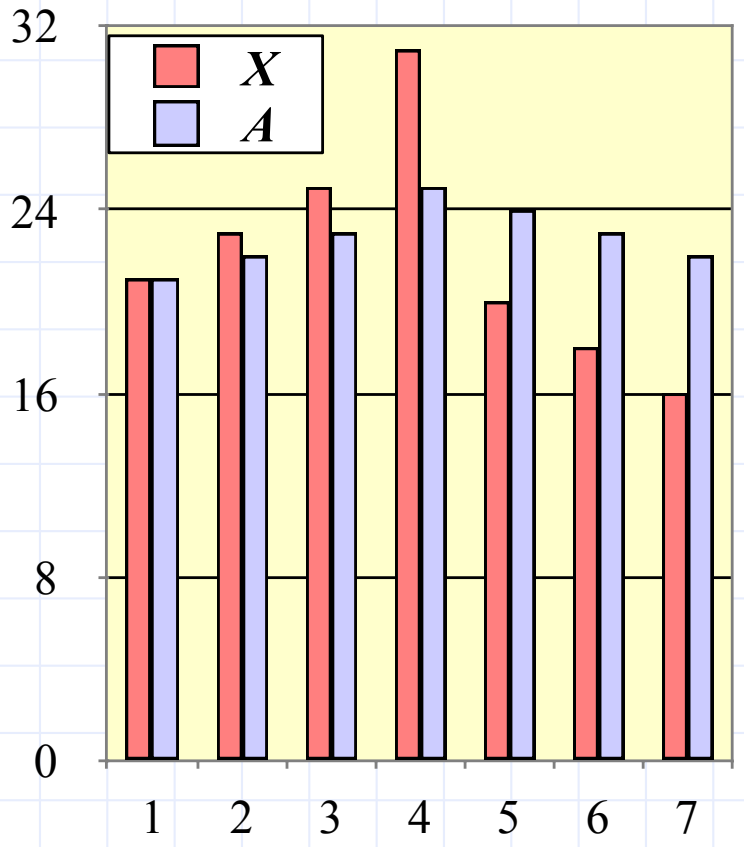
- The asymptotic analysis of an algorithm determines the running time in big-Oh notation
- To perform the asymptotic analysis
  - We find the worst-case number of primitive operations executed as a function of the input size
  - We express this function with big-Oh notation
- Example:
  - We say that algorithm arrayMax "runs in $O(n)$ time"
- Since constant factors and lower-order terms are eventually dropped anyhow, we can disregard them when counting primitive operations

Computing Prefix Averages

- We further illustrate asymptotic analysis with two algorithms for prefix averages
- The i-th prefix average of an array X is average of the first $(i + 1)$ elements of X:

$$A[i] = (X[0] + X[1] + \ldots + X[i])/(i+1)$$

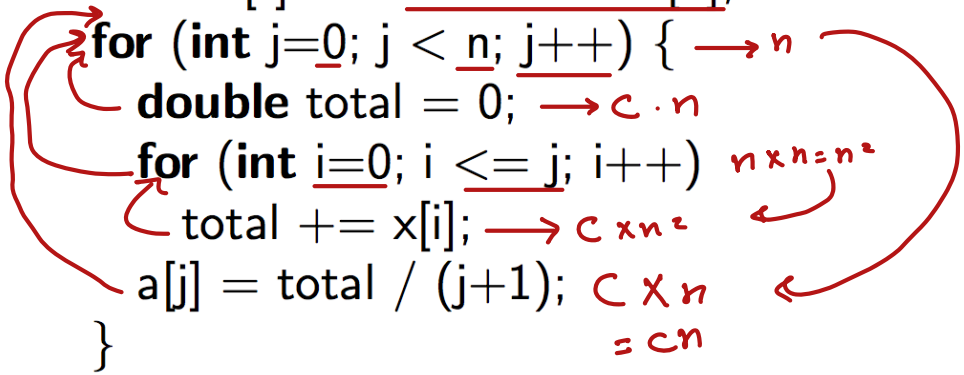
- Computing the array A of prefix averages of another array X has applications to financial analysis



Prefix Averages (Quadratic)

The following algorithm computes prefix averages in quadratic time by applying the definition

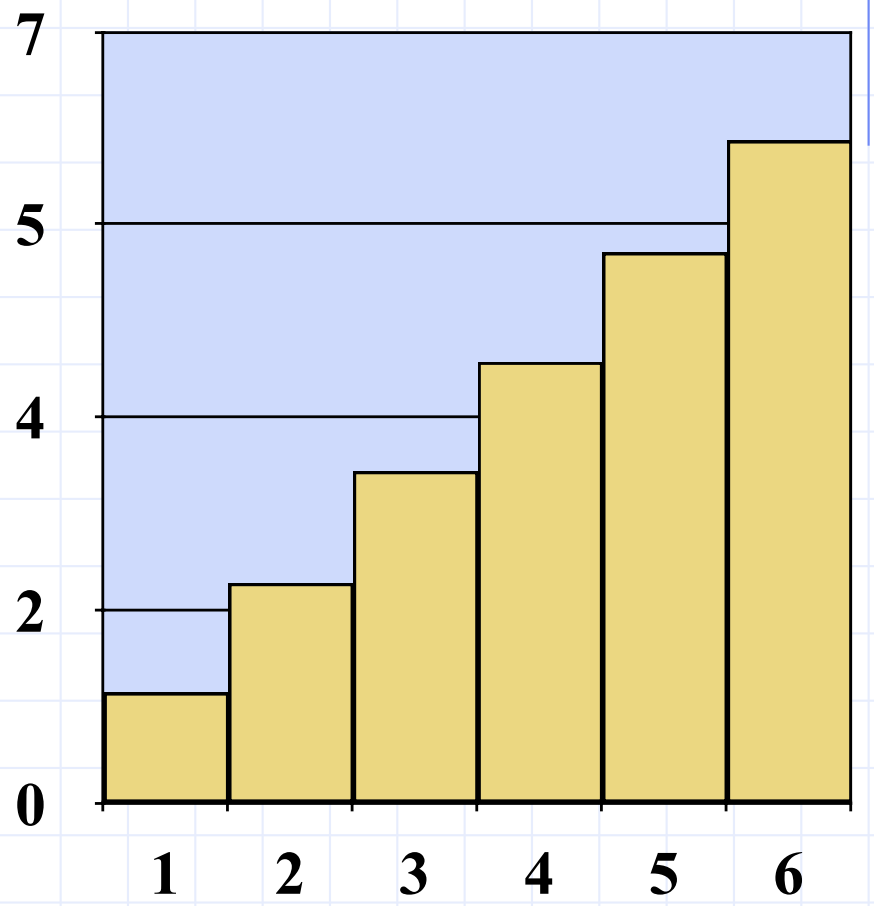
```
1  /** Returns an array a such that, for all j, a[j] equals the average of x[0], ..., x[j]. */
2  public static double[ ] prefixAverage1(double[ ] x) {
3    int n = x.length;
4    double[ ] a = new double[n];                  // filled with zeros by default
5    for (int j=0; j < n; j++) {
6      double total = 0;                           // begin computing x[0] + ... + x[j]
7      for (int i=0; i <= j; i++)
8        total += x[i];
9      a[j] = total / (j+1);                       // record the average
10   }
11   return a;
12 }
```



$T(n) = c + 2cn + n + 2cn^2 + n^2 + 1 \rightarrow O(n^2)$

Arithmetic Progression

- The running time of prefixAverage1 is $O(1 + 2 + \ldots + n)$
- The sum of the first n integers is $n(n + 1) / 2$
  - There is a simple visual proof of this fact
- Thus, algorithm prefixAverage1 runs in $O(n^2)$ time



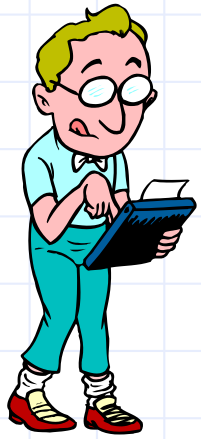
Prefix Averages 2 (Linear)

The following algorithm uses a running summation to improve the efficiency

```
1   /** Returns an array a such that, for all j, a[j] equals the average of x[0], ..., x[j]. */
2   public static double[ ] prefixAverage2(double[ ] x) {
3     int n = x.length;
4     double[ ] a = new double[n]; // filled with zeros by default
5     double total = 0;            // compute prefix sum as x[0] + x[1] + ...
6     for (int j=0; j < n; j++) {
7       total += x[j];             // update prefix sum to include x[j]
8       a[j] = total / (j+1);      // compute average based on current sum
9     }
10    return a;
11  }
```

Algorithm prefixAverage2 runs in $O(n)$ time!

Math you need to Review



- Summations
- Powers
- Logarithms
- Proof techniques
- Basic probability

- Properties of powers:
  - $a^{(b+c)} = a^b a^c$
  - $a^{bc} = (a^b)^c$
  - $a^b / a^c = a^{(b-c)}$
  - $b = a^{\log_a b}$
  - $b^c = a^{c*\log_a b}$
- Properties of logarithms:
  - $\log_b(xy) = \log_b x + \log_b y$
  - $\log_b (x/y) = \log_b x - \log_b y$
  - $\log_b xa = a\log_b x$
  - $\log_b a = \log_x a / \log_x b$

Relatives of Big-Oh



big-Omega

- f(n) is $\Omega(g(n))$ if there is a constant $c > 0$ and an integer constant $n_0 \geq 1$ such that

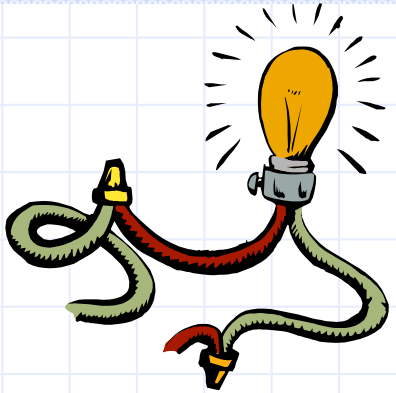
$$f(n) \geq c\, g(n) \text{ for } n \geq n_0$$

big-Theta

- f(n) is $\Theta(g(n))$ if there are constants $c' > 0$ and $c'' > 0$ and an integer constant $n_0 \geq 1$ such that

$$c'g(n) \leq f(n) \leq c''g(n) \text{ for } n \geq n_0$$

Intuition for Asymptotic Notation



big-Oh

- f(n) is O(g(n)) if f(n) is asymptotically less than or equal to g(n)

big-Omega

$f(n) \geq c \cdot g(n)$

- f(n) is $\Omega(g(n))$ if f(n) is asymptotically greater than or equal to g(n)

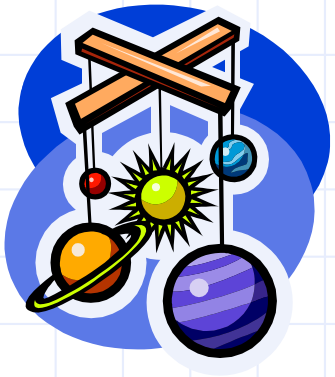
big-Theta

- f(n) is $\Theta(g(n))$ if f(n) is asymptotically equal to g(n)

$c_1 \cdot g(n) < f(n) < c_2 \cdot g(n)$

from the same class

Example Uses of the Relatives of Big-Oh



- **$5n^2$ is $\Omega(n^2)$**

  $f(n)$ is $\Omega(g(n))$ if there is a constant $c > 0$ and an integer constant $n_0 \geq 1$ such that $f(n) \geq c\, g(n)$ for $n \geq n_0$

  let $c = 5$ and $n_0 = 1$

- **$5n^2$ is $\Omega(n)$**

  $f(n)$ is $\Omega(g(n))$ if there is a constant $c > 0$ and an integer constant $n_0 \geq 1$ such that $f(n) \geq c\, g(n)$ for $n \geq n_0$

  let $c = 1$ and $n_0 = 1$

- **$5n^2$ is $\Theta(n^2)$**

  $f(n)$ is $\Theta(g(n))$ if it is $\Omega(n^2)$ and $O(n^2)$. We have already seen the former, for the latter recall that $f(n)$ is $O(g(n))$ if there is a constant $c > 0$ and an integer constant $n_0 \geq 1$ such that $f(n) \leq c\, g(n)$ for $n \geq n_0$

  Let $c = 5$ and $n_0 = 1$

Example 4.14: $3n\log n - 2n$ *is* $\Omega(n\log n)$.

Justification: $3n\log n - 2n = n\log n + 2n(\log n - 1) \geq n\log n$ for $n \geq 2$; hence, we can take $c = 1$ and $n_0 = 2$ in this case. ■

↳ $\log n$ 이상

$(2n\log n + n\log n) - 2n$ $\qquad$ $2n(\log \leq n-1)$

In general, we should use the big-Oh notation to characterize a function as closely as possible. While it is true that the function $f(n) = 4n^3 + 3n^2$ is $O(n^5)$ or even $O(n^4)$, it is more accurate to say that $f(n)$ is $O(n^3)$. Consider, by way of analogy,

---

Example 4.9: $5n^2 + 3n\log n + 2n + 5$ *is* $O(n^2)$.

Justification: $5n^2 + 3n\log n + 2n + 5 \leq (5+3+2+5)n^2 = cn^2$, for $c = 15$, when $n \geq n_0 = 1$. ■

Example 4.10: $20n^3 + 10n\log n + 5$ *is* $O(n^3)$.

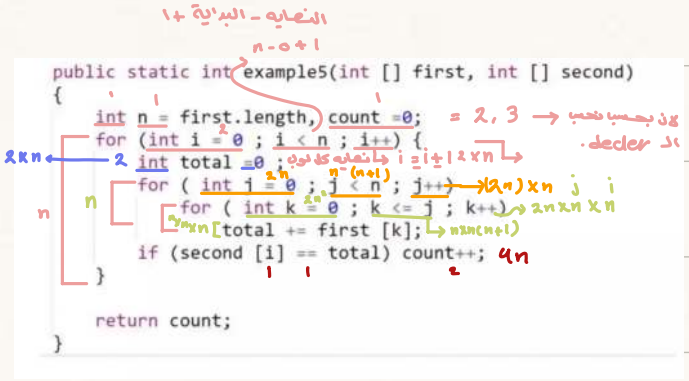
Justification: $20n^3 + 10n\log n + 5 \leq 35n^3$, for $n \geq 1$. ■

Example 4.11: $3\log n + 2$ *is* $O(\log n)$.

Justification: $3\log n + 2 \leq 5\log n$, for $n \geq 2$. Note that $\log n$ is zero for $n = 1$. That is why we use $n \geq n_0 = 2$ in this case. ■

Example 4.12: 2^{n+2} *is* $O(2^n)$.

Justification: $2^{n+2} = 2^n \cdot 2^2 = 4 \cdot 2^n$; hence, we can take $c = 4$ and $n_0 = 1$ in this case. ■



$O(n^3)$

depend loop:

$j = 0 \; ; \; j < n$

$K = 0 \; ; \; K <= j$

n=9 j	K				
0	0				stop 1
1	0	1			
2	0	1	2		2
3	0	1	2	3	4
4 → stop					

اذا كان اللوب الي جواء يعتمد على اللوب الي حوا

عدد تكرارات اللوب الي حوا وجوده في اللوب الي حوا

$\frac{n(n+1)}{2}$

$\frac{4(n+1)}{4}$

* $(n-2)+(n-1)+n$

$\rightarrow n^2$

$\frac{7n^2 + 3n\log_2 n \leq c \cdot n^2}{n}$

$7n + 3\log_2 n \leq c \cdot n^2$

$c=1 \quad 7(1)+3(0) \leq (1)^2 \times$

$n=1 \quad 7(2)+3(1) \leq (2)^2 \times$

$n=2 \quad 7(4)+3(2) \leq (4)^2 \times$

$n=4 \quad 7(8)+3(3) \leq (8)^2 \rightarrow 65 \leq 64$

$n=8 \quad 7(16)+3(4) \leq (16^2)$

$n=16$