SE274 Data Structure

Lecture 2: Algorithm Analysis

Mar 09, 2020

Instructor: Sunjun Kim

Information&Communication Engineering, DGIST

Course Information

- Lectures: Monday 14:30 – 16:00 / Wed 15:00 – 16:30 (1.5h each)
- Office hour: Thursday 13:00 – 14:00 / contact: sunjun_kim@dgist.ac.kr
- TA:
  - 장용주 (dracol@dgist.ac.kr)
  - 배지훈 (greg1121@dgist.ac.kr)
- Prerequisite
  - Python (SE213 Programming)
- Textbook
  - **Data Structures and Algorithms in Python,**
    by Michael T. Godrich et al, Wiley (2013)
- Additional Reading
  - Introduction to Algorithms
    Thomas H. Cormen et al., MIT Press

Course Information (2)

- Grading
  - Attendance: 10%
  - Homework: 30% (4 assignments + homework)
  - Midterm: 30%
  - Final exam: 30%
- Note
  - The programming language used in this course is now Python (previously C++).
  - This course is a prerequisite for SE380 Computer Algorithm

Course Information (3)

- Lecture + Exercise structure
  - Monday: lecture on a topic.
  - Wednesday: remaining lecture + in-class exercise (feat. Jupyter Notebook)
    - I highly recommend you to bring (or use) your own laptop during the Wed classes.

- Your reactions and feedback are essential!

Schedule

- Note: the schedules are subject to change at any time

Week 1: Environment Setting

Week 2: Introduction, Primitive types, Array, List, Linked List

Week 3: Algorithm Analysis

Week 4: Stack, Queues, Deques (HW 1)

Week 5: Trees

Week 6: Heaps and Priority Queus (HW 2)

Week 7: Hash Tables, Maps, and Skip Lists

Week 8: Midterm exam

Week 9: Search Trees (Binary Search Trees, AVL Trees)

Week 10: Search Trees (2,4 Trees, Red-Black Trees)

Week 11: Sorting (HW3)

Week 12: Graph

Week 13: Shortest Paths (HW4)

Week 14: Text Processing

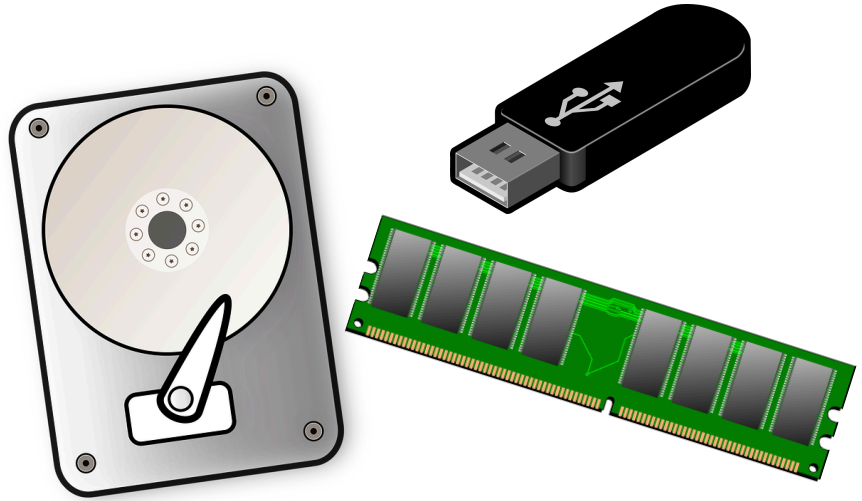
Week 15: Memory Management and B-Trees

Week 16: Final Exam

The need for Data Structure

Data, in a structure

0100101010111011…



In-class exercise

- Q1) Represent a decimal number **13** in binaries.

In-class exercise

- Q1) Represent a decimal number **13** in binaries.

- Binary converting

=8	=4	=2	=1	(is equal to)
2^3	2^2	2^1	2^0	(order)
1	***1***	***0***	***1***	***(binary = bits)***

= 8 + 4 + 1 = 13

In-class exercise

- Q2) Represent a decimal number **0.25** in binaries.

In-class exercise

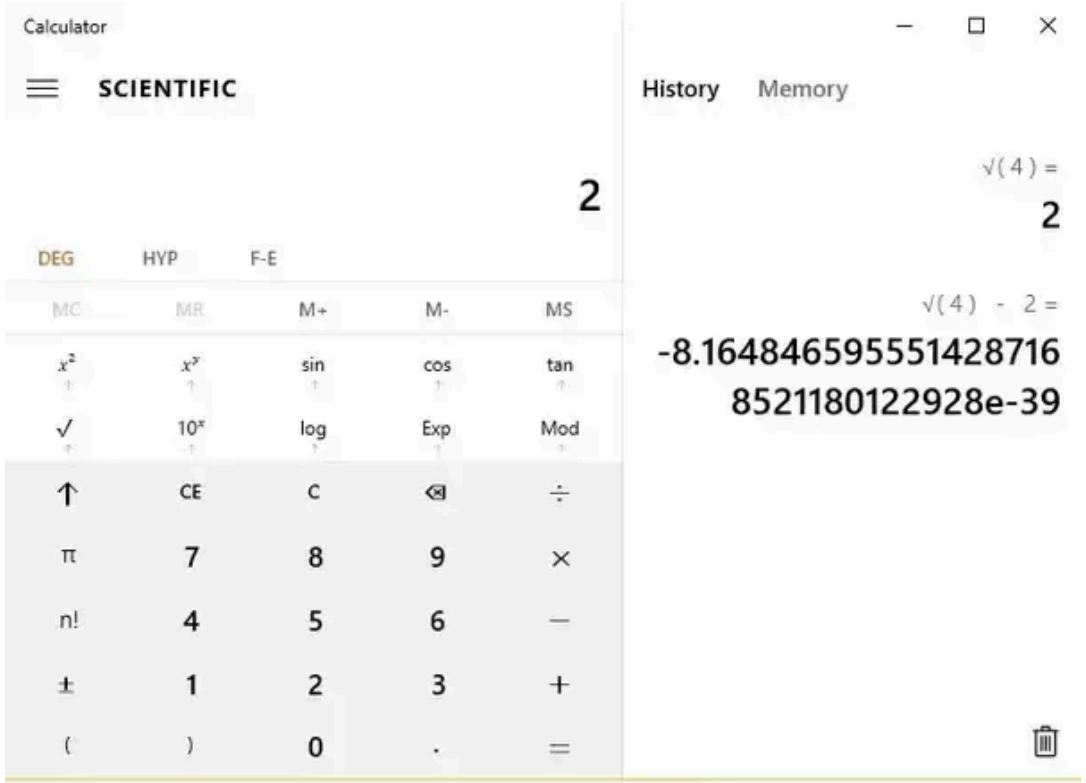
- Q2) Represent a decimal number **2.25** in binaries.

- Binary converting

=2	=1	=1/2	=1/4	(is equal to)
2^1	2^0	2^{-1}	2^{-2}	(order)
1	***0.***	***0***	***1***	***(binary = bits)***

= 2 + ¼ = 2.25

An error inherent in a data representation



https://www.quora.com/Why-hasn%E2%80%99t-Microsoft-ever-fixed-the-square-root-bug-in-Windows-Calculator

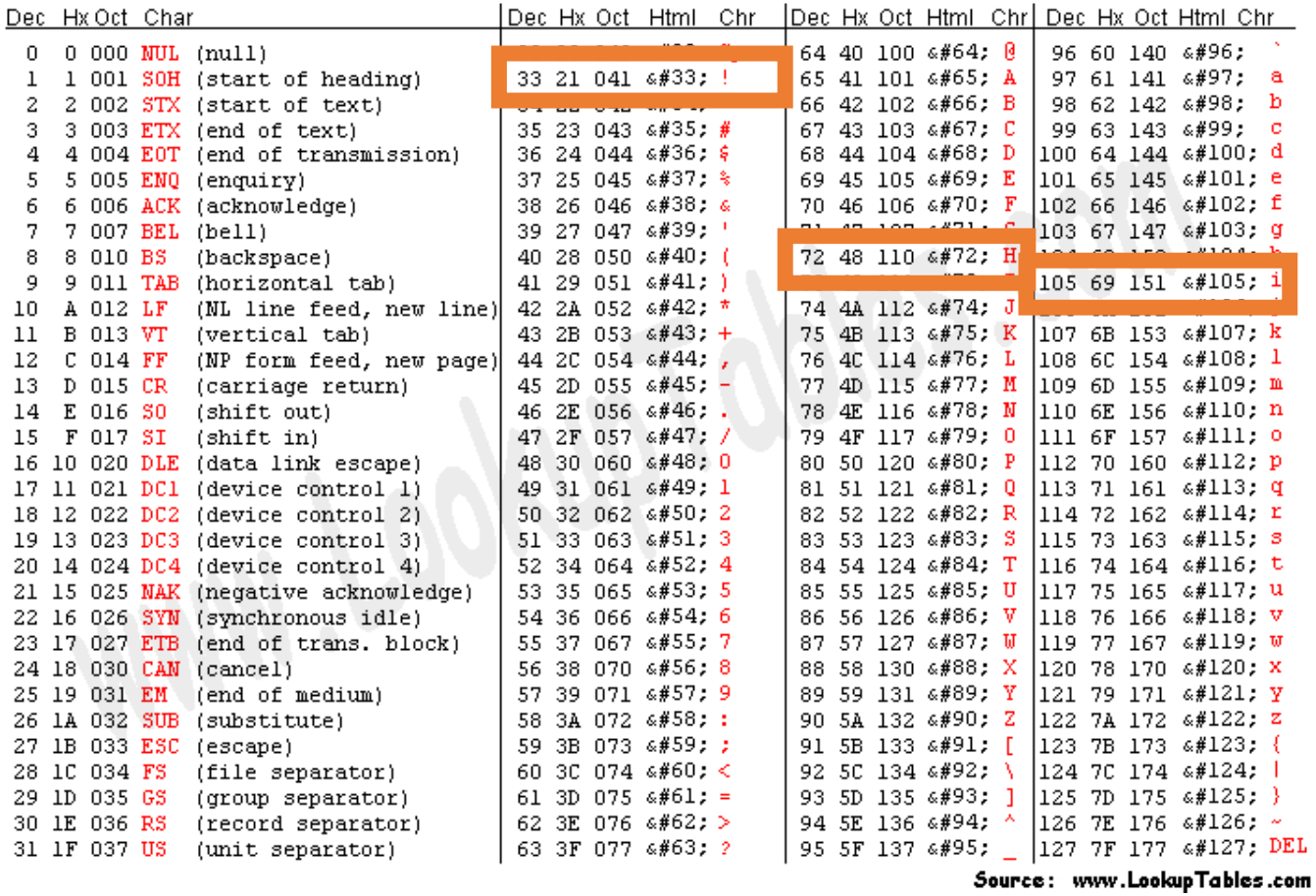
- E.g.,) 1.0 = 1/2 + 1/4 + 1/8 + 1/16 + 1/32 + … = 0.111111111111 (bin)

In-class exercise

- Q3) Represent [Hi!] in binaries.

In-class exercise

- Q3) Represent [Hi!] in binaries.



Dec	Hx	Oct	Char	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL (null)						64	40	100	@	@	96	60	140	`	`
1	1	001	SOH (start of heading)	33	21	041	!	!	65	41	101	A	A	97	61	141	a	a
2	2	002	STX (start of text)						66	42	102	B	B	98	62	142	b	b
3	3	003	ETX (end of text)	35	23	043	#	#	67	43	103	C	C	99	63	143	c	c
4	4	004	EOT (end of transmission)	36	24	044	$	$	68	44	104	D	D	100	64	144	d	d
5	5	005	ENQ (enquiry)	37	25	045	%	%	69	45	105	E	E	101	65	145	e	e
6	6	006	ACK (acknowledge)	38	26	046	&	&	70	46	106	F	F	102	66	146	f	f
7	7	007	BEL (bell)	39	27	047	'	'						103	67	147	g	g
8	8	010	BS (backspace)	40	28	050	(	(	72	48	110	H	H					
9	9	011	TAB (horizontal tab)	41	29	051	)	)						105	69	151	i	i
10	A	012	LF (NL line feed, new line)	42	2A	052	*	*	74	4A	112	J	J					
11	B	013	VT (vertical tab)	43	2B	053	+	+	75	4B	113	K	K	107	6B	153	k	k
12	C	014	FF (NP form feed, new page)	44	2C	054	,	,	76	4C	114	L	L	108	6C	154	l	l
13	D	015	CR (carriage return)	45	2D	055	-	-	77	4D	115	M	M	109	6D	155	m	m
14	E	016	SO (shift out)	46	2E	056	.	.	78	4E	116	N	N	110	6E	156	n	n
15	F	017	SI (shift in)	47	2F	057	/	/	79	4F	117	O	O	111	6F	157	o	o
16	10	020	DLE (data link escape)	48	30	060	0	0	80	50	120	P	P	112	70	160	p	p
17	11	021	DC1 (device control 1)	49	31	061	1	1	81	51	121	Q	Q	113	71	161	q	q
18	12	022	DC2 (device control 2)	50	32	062	2	2	82	52	122	R	R	114	72	162	r	r
19	13	023	DC3 (device control 3)	51	33	063	3	3	83	53	123	S	S	115	73	163	s	s
20	14	024	DC4 (device control 4)	52	34	064	4	4	84	54	124	T	T	116	74	164	t	t
21	15	025	NAK (negative acknowledge)	53	35	065	5	5	85	55	125	U	U	117	75	165	u	u
22	16	026	SYN (synchronous idle)	54	36	066	6	6	86	56	126	V	V	118	76	166	v	v
23	17	027	ETB (end of trans. block)	55	37	067	7	7	87	57	127	W	W	119	77	167	w	w
24	18	030	CAN (cancel)	56	38	070	8	8	88	58	130	X	X	120	78	170	x	x
25	19	031	EM (end of medium)	57	39	071	9	9	89	59	131	Y	Y	121	79	171	y	y
26	1A	032	SUB (substitute)	58	3A	072	:	:	90	5A	132	Z	Z	122	7A	172	z	z
27	1B	033	ESC (escape)	59	3B	073	;	;	91	5B	133	[	[	123	7B	173	{	{
28	1C	034	FS (file separator)	60	3C	074	<	<	92	5C	134	\	\	124	7C	174			|
29	1D	035	GS (group separator)	61	3D	075	=	=	93	5D	135	]	]	125	7D	175	}	}
30	1E	036	RS (record separator)	62	3E	076	>	>	94	5E	136	^	^	126	7E	176	~	~
31	1F	037	US (unit separator)	63	3F	077	?	?	95	5F	137	_	_	127	7F	177	&#127;	DEL

Source: www.LookupTables.com

Ascii Table

H i !

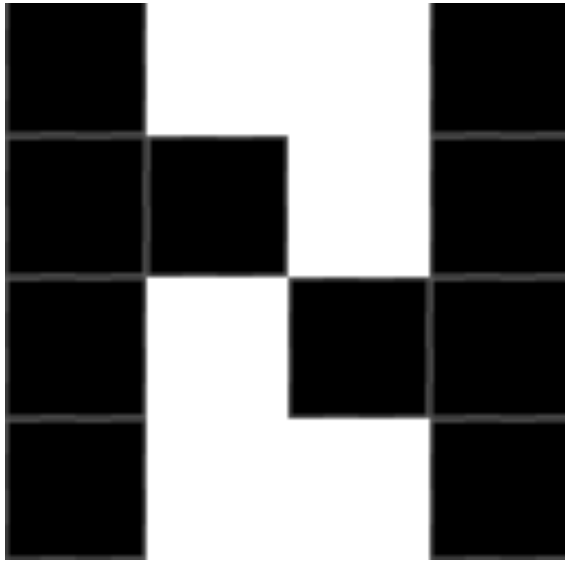
Char	Hex	Bin
H	0x48	0100 1000
i	0x69	0110 1001
!	0x21	0010 0001
(null)	0x00	0000 0000

= 8 bits

= 1 byte

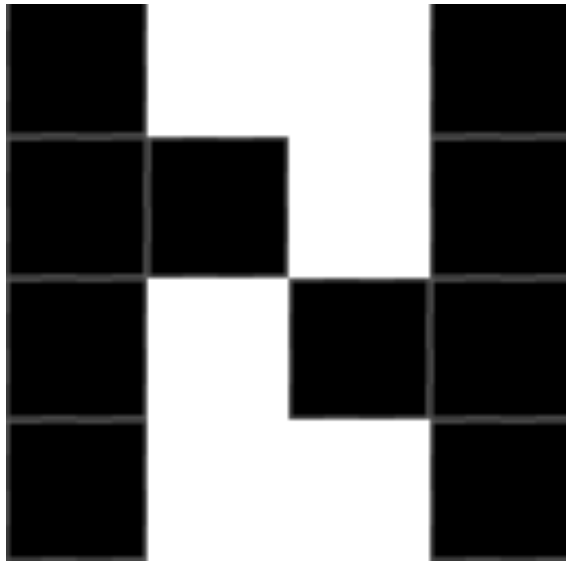
In-class exercise

- Q4) Represent the following image in binaries.



In-class exercise

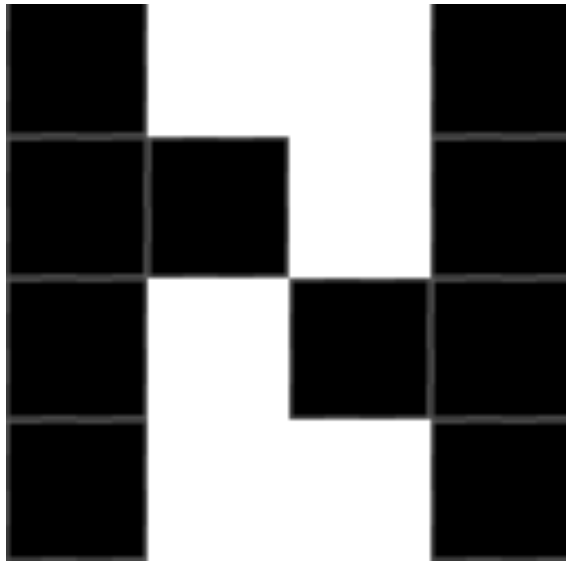
- Q4) Represent the following image in binaries.



1	0	0	1
1	1	0	1
1	0	1	1
1	0	0	1

In-class exercise

- Q4) Represent the following image in binaries.



0	1	0	0
0	1	0	0
1	0	0	1
1	1	0	1
1	0	1	1
1	0	0	1

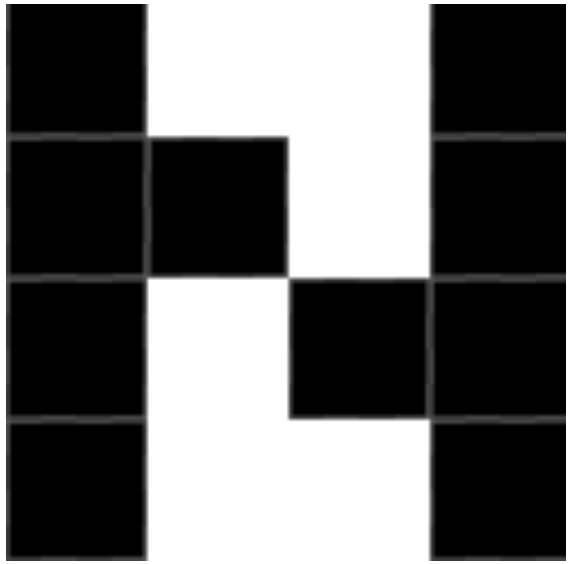
= 4 (width)

= 4 (height)

= 0100 0100 1001 1101 1011 1001

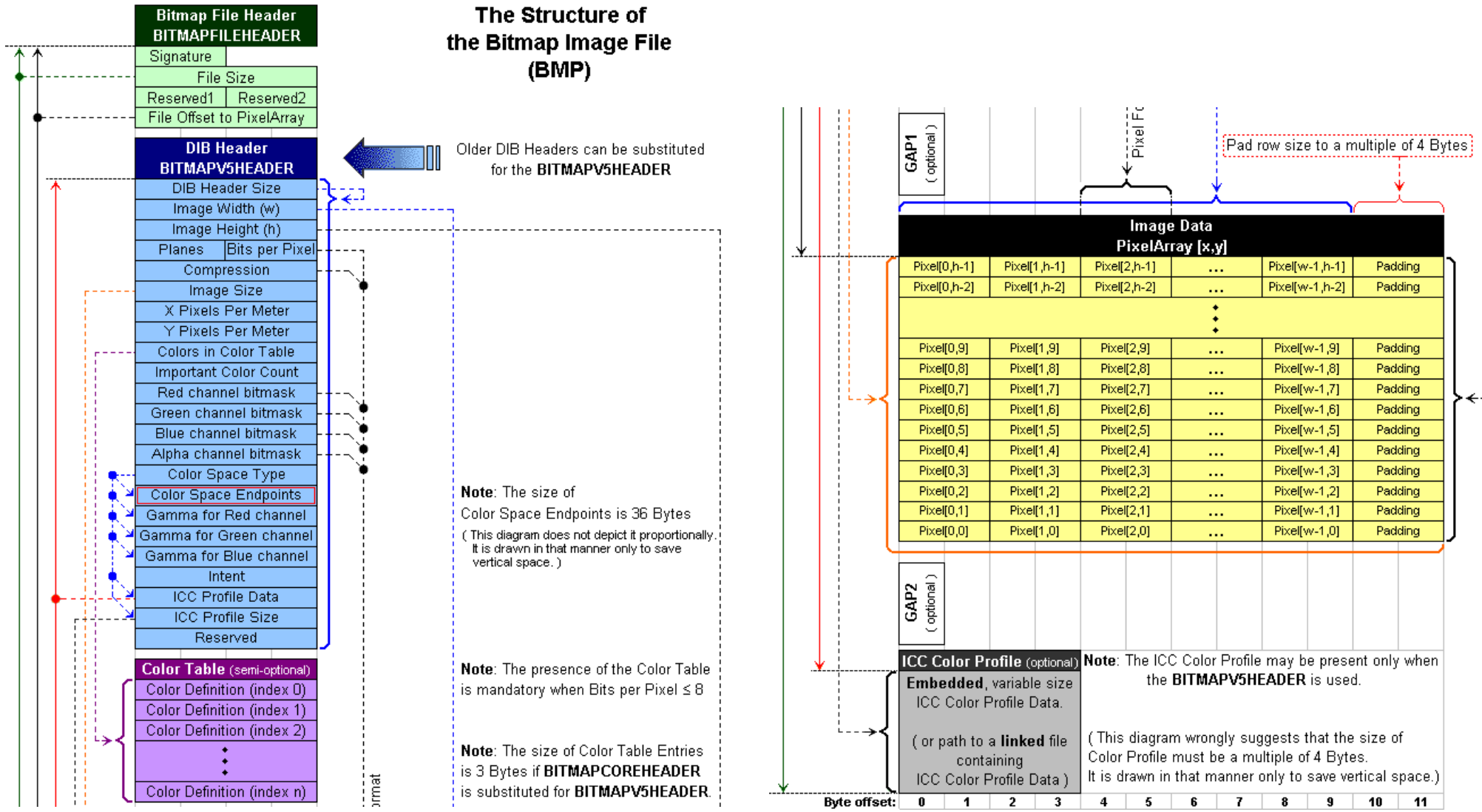
In-class exercise

- Q4) Represent the following image in binaries.



0	1	0	0	= 4 (width)
0	1	0	0	= 4 (height)
1	0	0	1	
1	1	0	1	
1	0	1	1	
1	0	0	1	

File format specification: BMP



What is data structure?

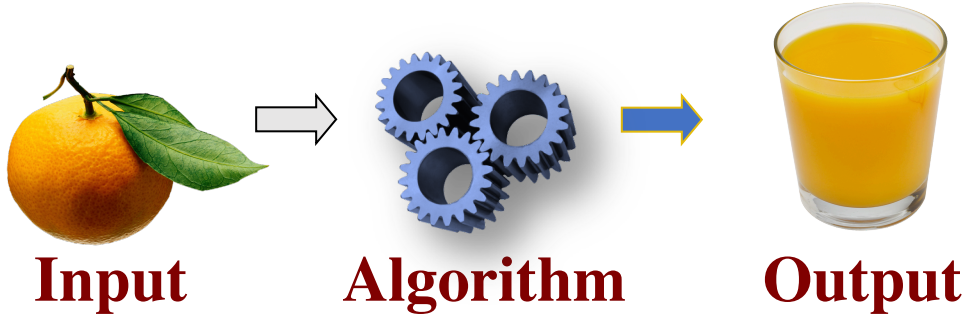
- Computer programs: all about data manipulation
- The core of many areas in CS such as web, OS, computer graphics, vision, and AI lies on ***HOW TO REPRESENT AND PROCESS DATA***.
- Therefore, it's essential to organize data for processing a massive amount of data ***efficiently***.

- In this course, we will study some of fundamental data structures such as lists, queues, hashes, tables, trees, graphs, etc.
- We will also study (a bit of) computational algorithms based on such data structures (e.g., search, sorting, etc.) and ***how to analyze their efficiency***.

What is *GOOD* data structure?

- Data structure: a systematic way of organizing and accessing data.
- Algorithm: a step-by-step procedure for performing some task (mainly, data manipulation) in a finite amount of time.
- ***Good data structure*** (and algorithm) is a structure that accelerate the processing time of a given task.

→ We need a ***precise*** way to analyze the "***goodness***" of the data structure and algorithm.



Week 3: Analysis of Algorithms

In-class exercise

- Q5) Sorting
  - Get a pen and some paper.
  - Write 10 random numbers between 1 – 1000.

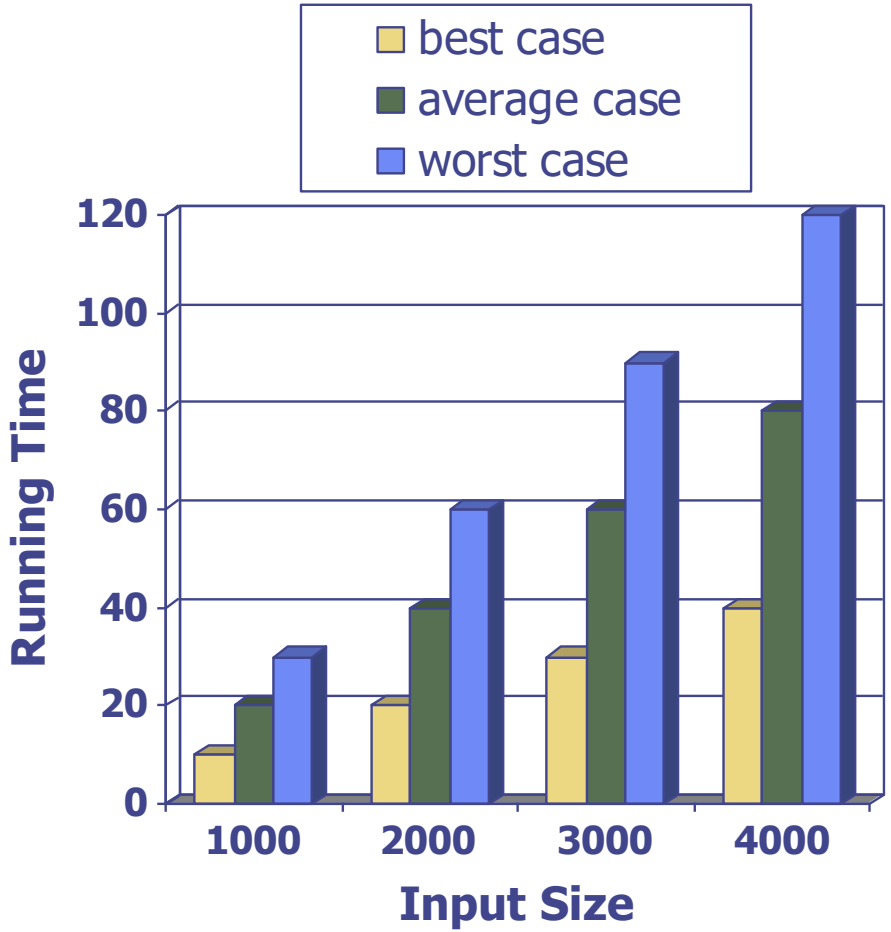
In-class exercise

- Q5) Sorting
  - Get a pen and some paper.
  - Write 10 random numbers between 1 – 1000.

  - Now, make your numbers sorted.
    - Explain how you sort those numbers (raise your hand).

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

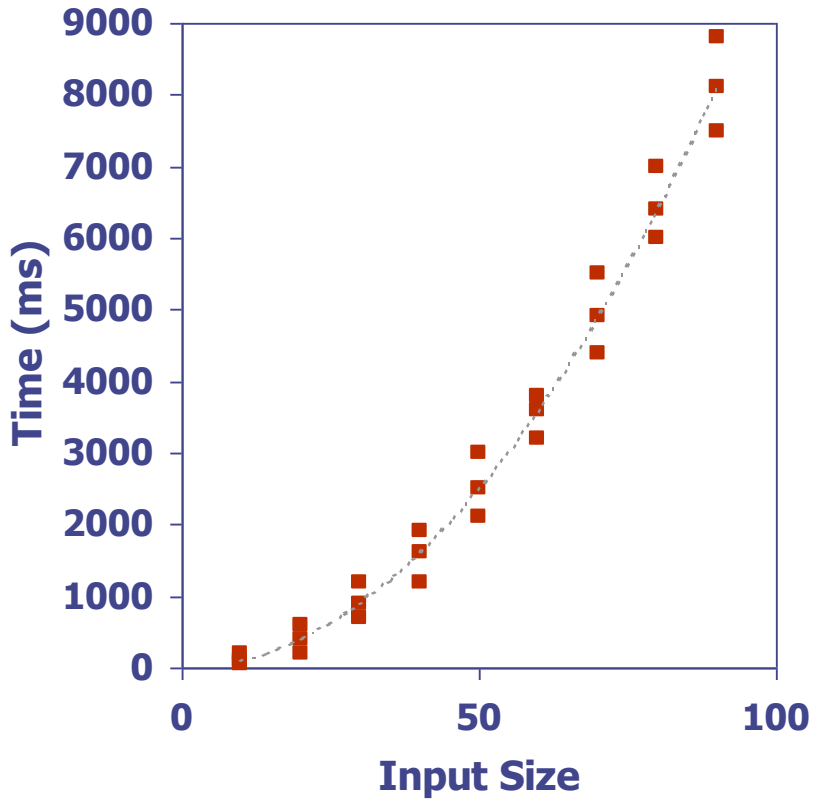


Experimental Studies

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

```
from time import time
start_time = time( )
run algorithm
end_time = time( )
elapsed = end_time − start_time
```

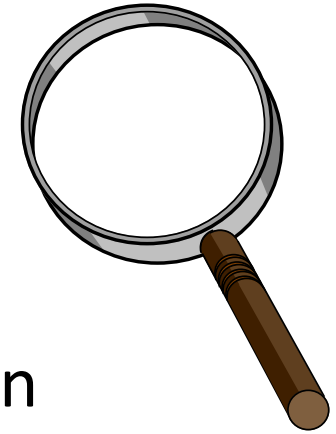
- Plot the results



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

Theoretical Analysis



- 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

Pseudocode

- High-level description of an algorithm
- More structured than English prose
- Less detailed than a program
- Preferred notation for describing algorithms
- Hides program design issues

In-class exercise

- Q5) Sorting (cont.)
  - Get a pen and some paper.
  - Write 10 random numbers between 1 – 1000.
  - Now, make your numbers sorted.
    - Explain how you sort those numbers.
  - Try to write a pseudocode for your algorithm.
    - raise your hand when you’re done.

Pseudocode Details

- Control flow
  - **if** … **then** … [**else** …]
  - **while** … **do** …
  - **repeat** … **until** …
  - **for** … **do** …
  - Indentation replaces braces
- Method declaration

  Algorithm ***method*** (***arg*** [, ***arg***…])

  Input …

  Output …

- Method call

  method (***arg*** [, ***arg***…])
- Return value

  return ***expression***
- Expressions:

  ←Assignment

  = Equality testing

  n^2 Superscripts and other mathematical formatting allowed

In-class exercise

- Q5) Sorting (cont.)
  - Get a pen and some paper.
  - Write 10 random numbers between 1 – 1000.
  - Now, make your numbers sorted.
    - Explain how you sort those numbers.
  - Try to write a pseudocode for your algorithm.
    - raise your hand when you're done.

```
1:  function SELECTION-SORT(A, n)
2:      for i = 1 to n-1 do
3:          min ← i
4:          for j = i + 1 to n do
5:              if A[j] < A[min] then
6:                  min ← j
7:              end if
8:          end for
9:          swap A[i], A[min]
10:     end for
11: end function
```

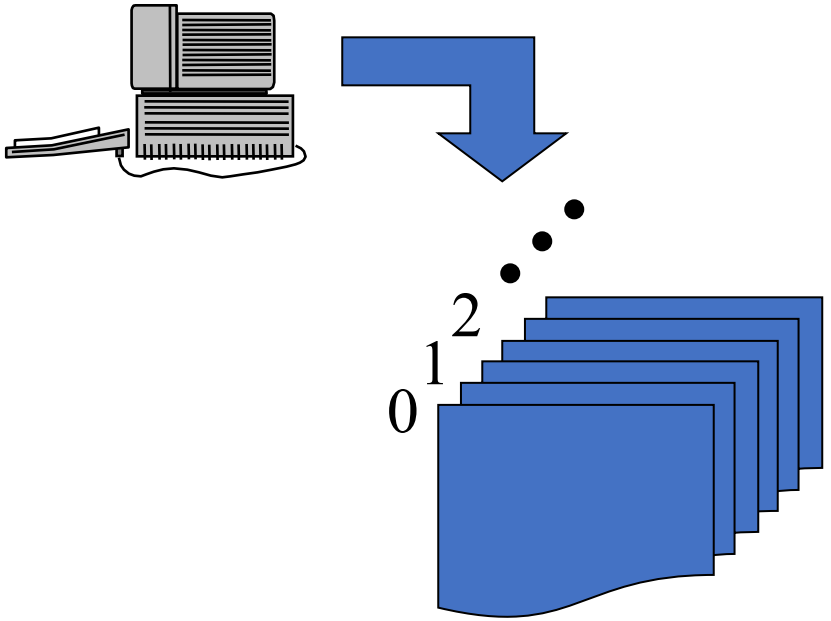
A = 30, 412, 21, 404, 42 (n = 5)

```
1:  function SELECTION-SORT(A, n)
2:      for i = 1 to n-1 do
3:          min ← i
4:          for j = i + 1 to n do
5:              if A[j] < A[min] then
6:                  min ← j
7:              end if
8:          end for
9:          swap A[i], A[min]
10:     end for
11: end function
```

The Random Access Machine (RAM) Model

- A **CPU**

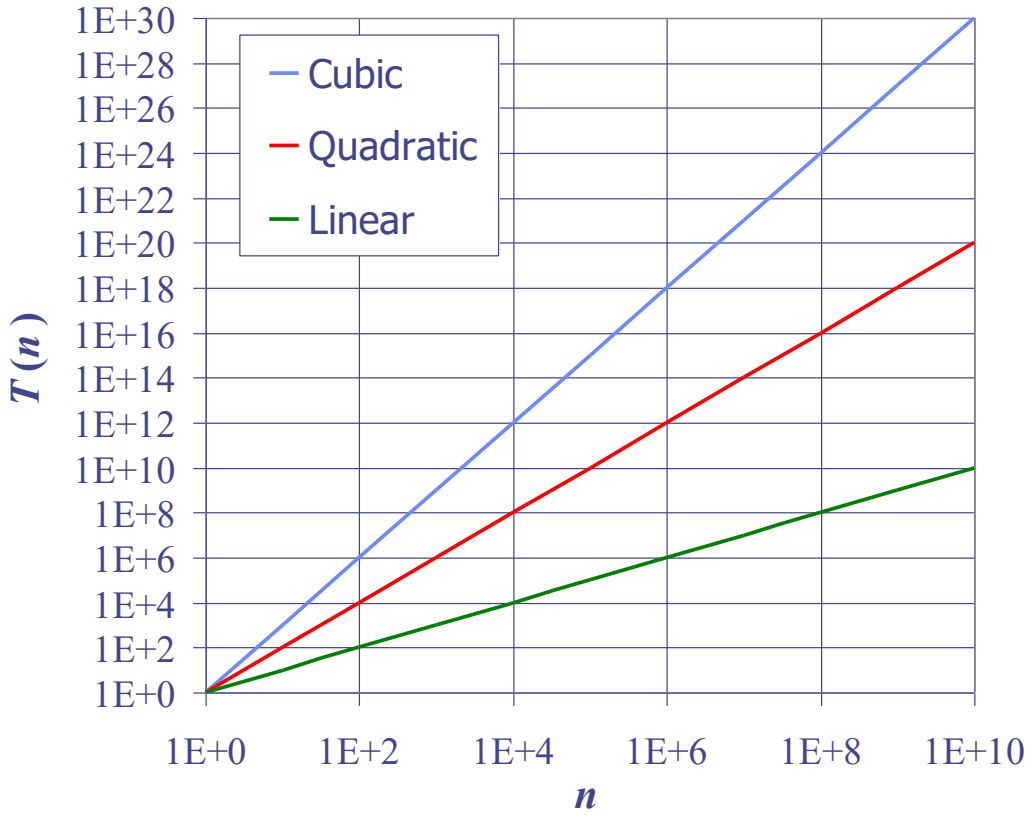
- An potentially unbounded bank of **memory** cells, each of which can hold an arbitrary number or character



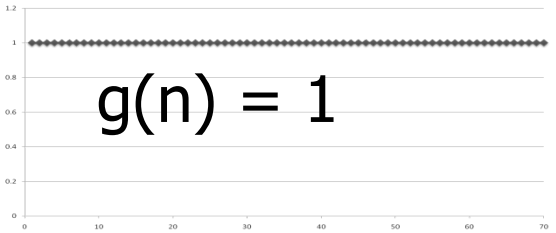
- Memory cells are numbered and accessing any cell in memory takes **unit time.**

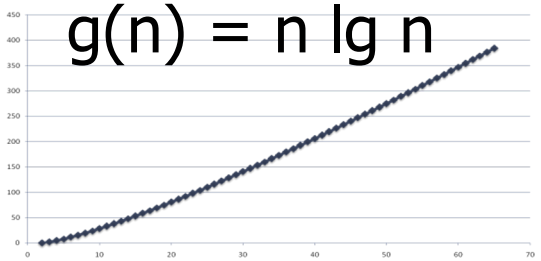
Seven Important Functions

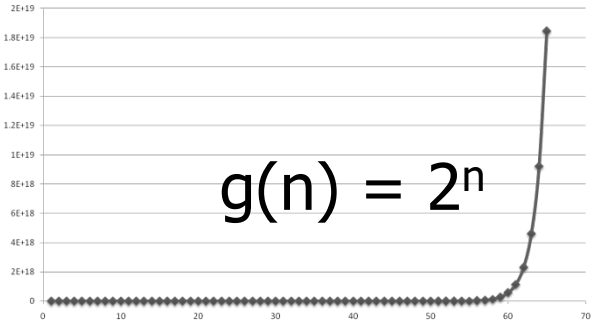
- Seven functions that often appear in algorithm analysis:
  - Constant ≈ 1
  - Logarithmic ≈ log n
  - Linear ≈ n
  - N-Log-N ≈ n log n
  - Quadratic ≈ n^2
  - Cubic ≈ n^3
  - Exponential ≈ 2^n
- In a log-log chart, the slope of the line corresponds to the growth rate

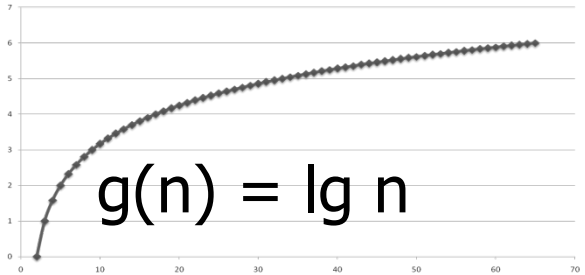


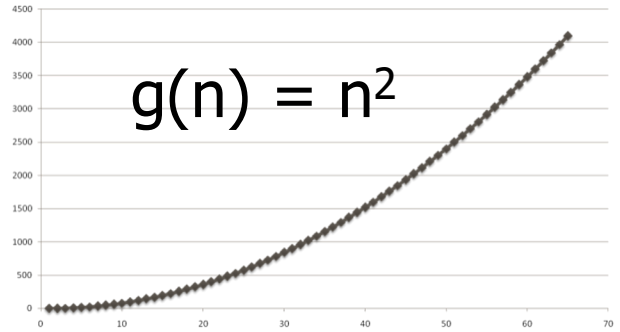
Functions Graphed Using "Normal" Scale

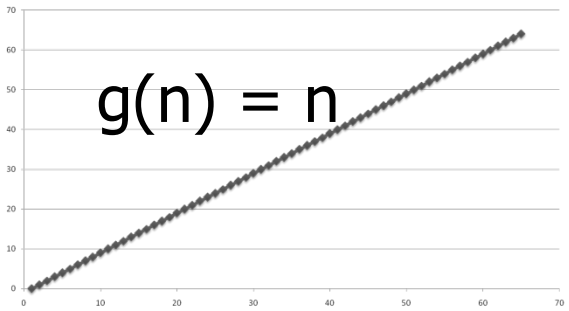


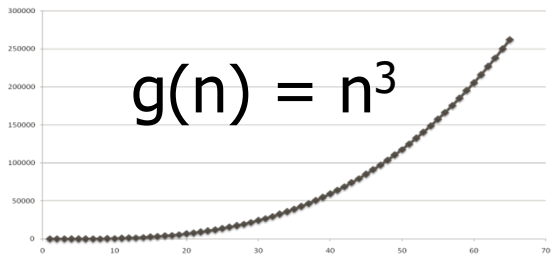












* Slide by Matt Stallmann included with permission.

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

```
# Ops
      2  1  def find_max(data):
         2    """Return the maximum element from a nonempty Python list."""
      2  3    biggest = data[0]                  # The initial value to beat
     2n  4    for val in data:                   # For each value:
     2n  5      if val > biggest                 # if it is greater than the best so far,
   0 – n 6        biggest = val                  # we have found a new best (so far)
      1  7    return biggest                     # When loop ends, biggest is the max
```

* n = len(data)

Estimating Running Time

- Algorithm **find_max** 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 **find_max.** Then

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

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

```
# Ops
  2     1  def find_max(data):
        2    """Return the maximum
  2     3    biggest = data[0]
 2n     4    for val in data:
 2n     5      if val > biggest
0 – n   6        biggest = val
+ 1     7    return biggest
```

5n + 5 (when step 6 = n, worst case)
4n + 5 (when step 6 = 0, best case)

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 **find_max**

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

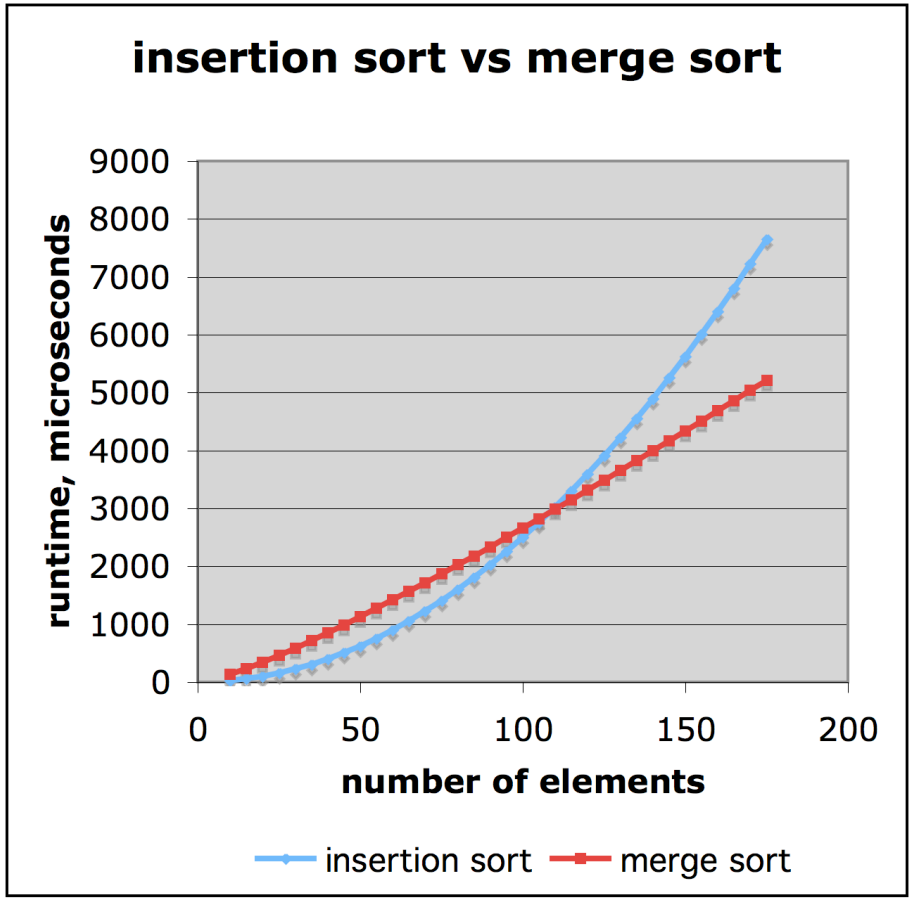
Why Growth Rate Matters

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

* Slide by Matt Stallmann included with permission.

Comparison of Two Sorting Algorithms



insertion sort is
$n^2 / 4$

merge sort is
2 n lg n

sort a million items?

insertion sort takes roughly 70 hours

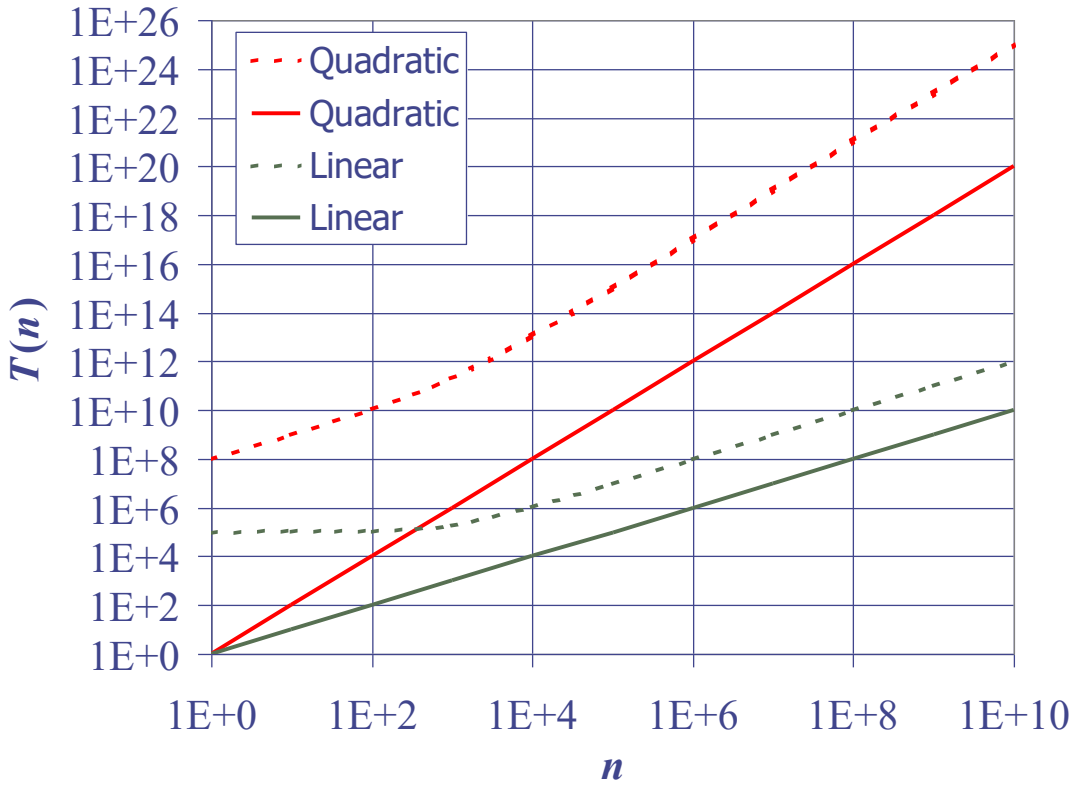
while

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

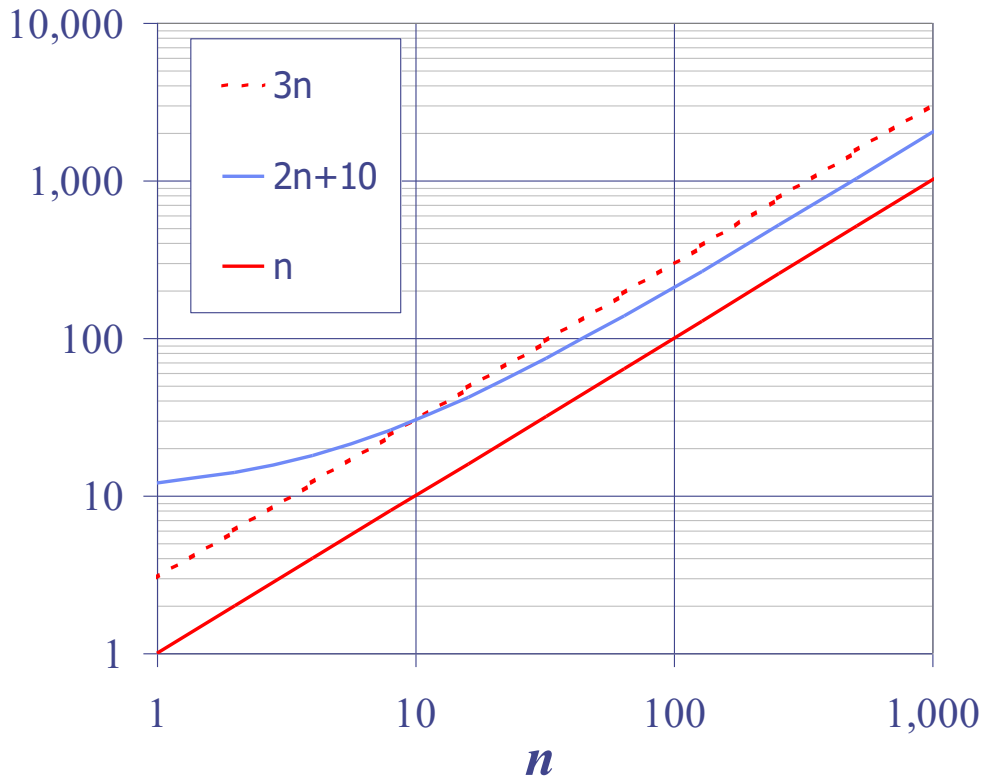


Big-Oh Notation

- 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

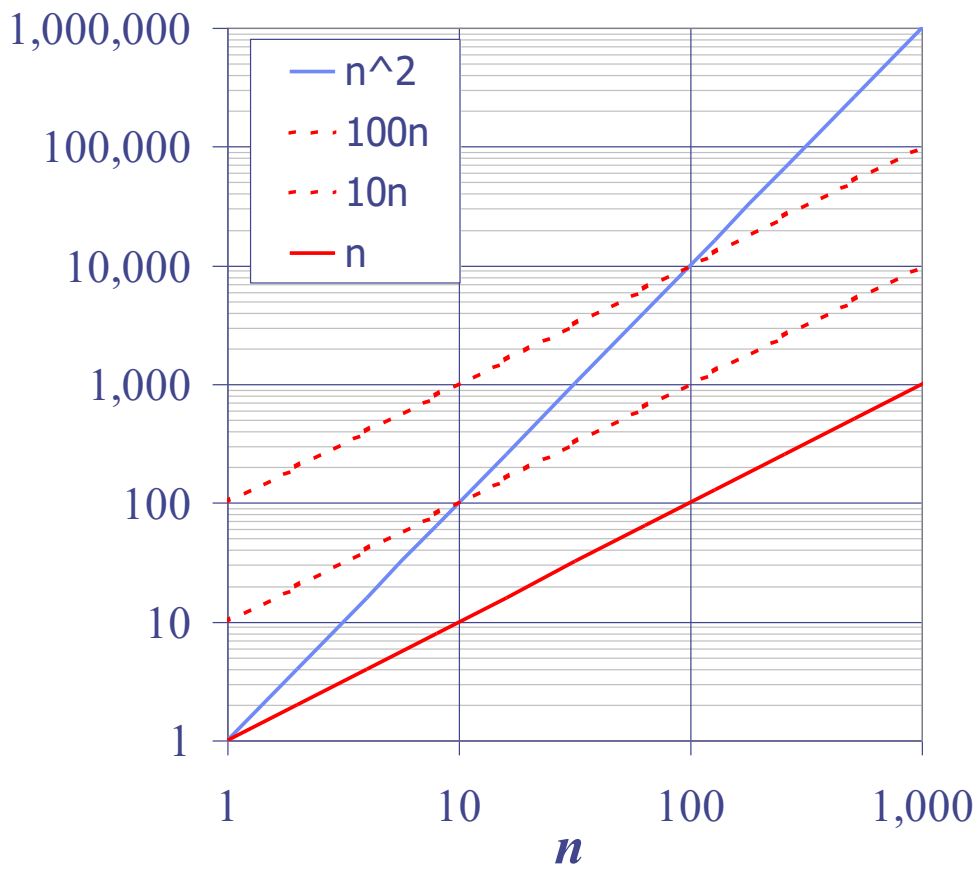
  $f(n) \leq cg(n)$ for $n \geq n_0$

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



Big-Oh Example

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



More Big-Oh Examples

- 7n-2

  7n-2 is O(n)

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

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

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

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

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

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

- 3 log n + 5

  3 log n + 5 is O(log n)

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

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

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

- If is $f(n)$ a polynomial of degree d, then $f(n)$ is $O(n^d)$, i.e.,
  1. Drop lower-order terms
  2. Drop constant factors
- 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 $O(3n)$"

In-class exercise

- Q5) Sorting (cont.)
  - Get a pen and some paper.
  - Write 10 random numbers between 1 – 1000.
  - Now, make your numbers sorted.
    - Explain how you sort those numbers.
  - Try to write a pseudocode for your algorithm.
    - raise your hand when you're done.
  - Represent your algorithm's running time with Big-Oh notation.

```
1:  function SELECTION-SORT(A, n)
2:      for i = 1 to n-1 do
3:          min ← i
4:          for j = i + 1 to n do
5:              if A[j] < A[min] then
6:                  min ← j
7:              end if
8:          end for
9:          swap A[i], A[min]
10:     end for
11: end function
```

In-class exercise

- Q5) Sorting (cont.)
  - The selection sort algorithm runs in $O(n^2)$ time.
  - $f(n) \leq cg(n)$ for $n \geq n_0$
  - $6n^2 + 3n + 4 \leq 7n^2$ when $n \geq 5$

  → c=7, $n_0 = 5$

# Ops		
2	1:	**function** SELECTION-SORT(A, n)
2n	2:	**for** i = 1 to n-1 **do**
2	3:	min ← i
$2n^2$	4:	**for** j = i + 1 to n **do**
$2n^2$	5:	**if** $A[j] < A[min]$ **then**
$0 – 2n^2$	6:	min ← j
-	7:	**end if**
-	8:	**end for**
n	9:	swap A[i], A[min]
-	10:	**end for**
-	11:	**end function**

$6n^2 + 3n + 4$ (when step 6 = n^2, worst case)
$4n^2 + 3n + 4$ (when step 6 = 0, best case)

Asymptotic Algorithm Analysis

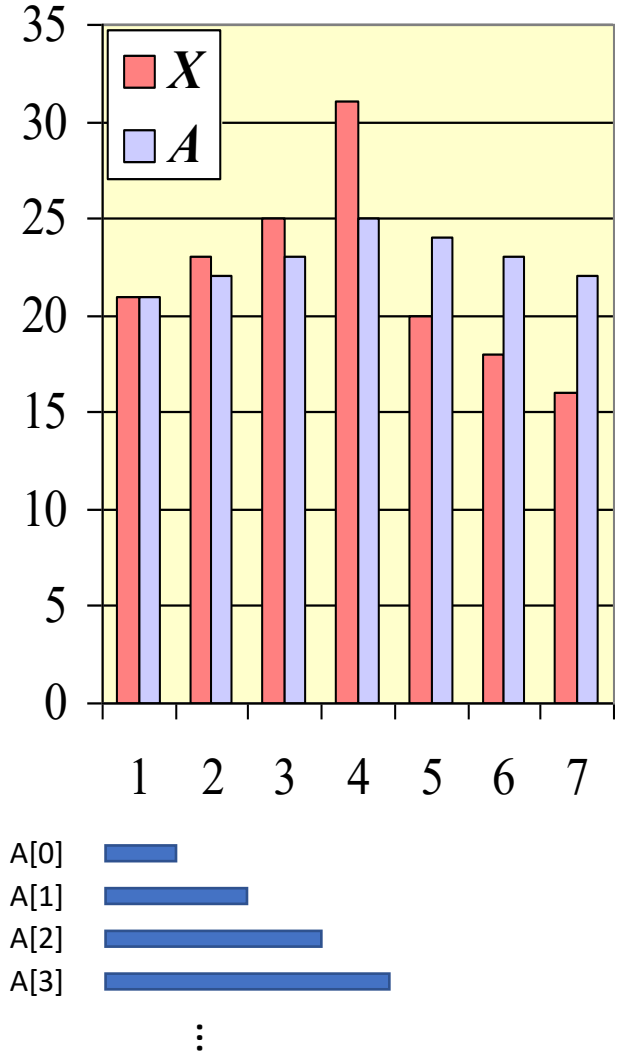
- The asymptotic analysis of an algorithm determines the running time in big-Oh notation
- To perform the asymptotic analysis
  - We find the worst-case number of primitive operations executed as a function of the input size
  - We express this function with big-Oh notation
- Example:
  - We say that algorithm **find_max** "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



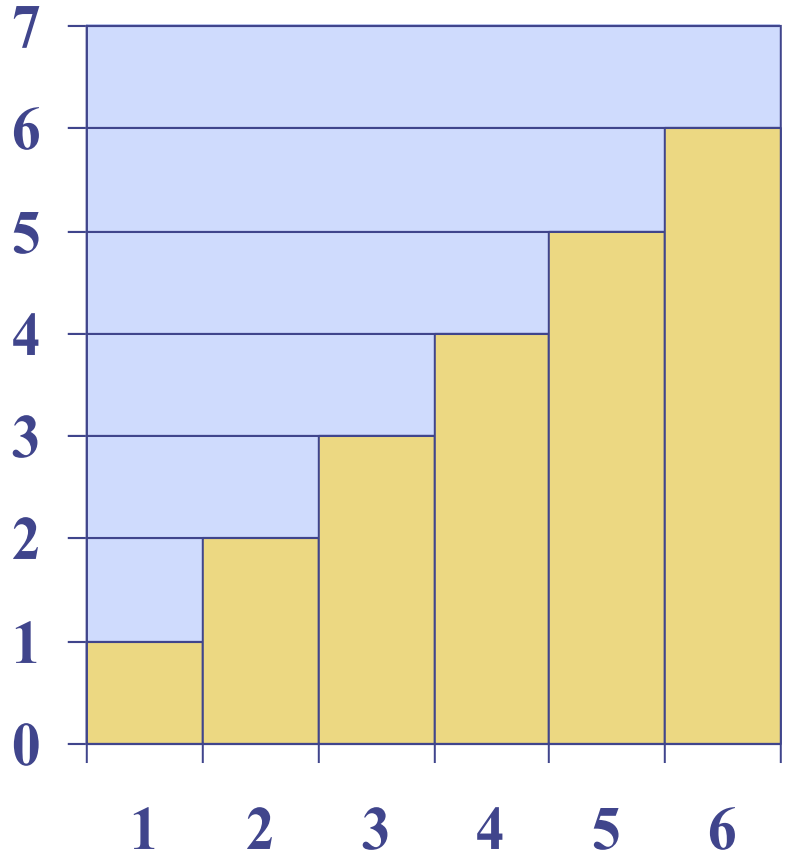
Prefix Averages (Quadratic)

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

```
1  def prefix_average1(S):
2    """Return list such that, for all j, A[j] equals average of S[0], ..., S[j]."""
3    n = len(S)
4    A = [0] * n                          # create new list of n zeros
5    for j in range(n):
6      total = 0                          # begin computing S[0] + ... + S[j]
7      for i in range(j + 1):
8        total += S[i]
9      A[j] = total / (j+1)               # record the average
10   return A
```

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 (Looks Better)

- The following algorithm uses an internal Python function to simplify the code

```
1  def prefix_average2(S):
2    """Return list such that, for all j, A[j] equals average of S[0], ..., S[j]."""
3    n = len(S)
4    A = [0] * n                          # create new list of n zeros
5    for j in range(n):
6      A[j] = sum(S[0:j+1]) / (j+1)       # record the average
7    return A
```

- Algorithm *prefixAverage2* still runs in $O(n^2)$ time!

Prefix Averages 3 (Linear Time)

- The following algorithm computes prefix averages in linear time by keeping a running sum

```
1  def prefix_average3(S):
2    """Return list such that, for all j, A[j] equals average of S[0], ..., S[j]."""
3    n = len(S)
4    A = [0] * n                    # create new list of n zeros
5    total = 0                      # compute prefix sum as S[0] + S[1] + ...
6    for j in range(n):
7      total += S[j]                # update prefix sum to include S[j]
8      A[j] = total / (j+1)         # compute average based on current sum
9    return A
```

  - Algorithm ***prefixAverage3*** runs in $O(n)$ time

Math you need to Review

- Summations
- Logarithms and Exponents

- Proof techniques
- Basic probability

- **properties of logarithms:**

  $\log_b(xy) = \log_b x + \log_b y$

  $\log_b (x/y) = \log_b x - \log_b y$

  $\log_b xa = a\log_b x$

  $\log_b a = \log_x a/\log_x b$

- **properties of exponentials**:

  $a^{(b+c)} = a^b a^c$

  $a^{bc} = (a^b)^c$

  $a^b / a^c = a^{(b-c)}$

  $b = a^{\log_a b}$

  $b^c = a^{c*\log_a b}$

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 $f(n) \geq c \bullet g(n)$ 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 $c' \bullet g(n) \leq f(n) \leq c'' \bullet g(n)$ 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) is Ω(g(n)) if f(n) is asymptotically **greater than or equal** to g(n)

big-Theta

- f(n) is Θ(g(n)) if f(n) is asymptotically **equal** to g(n)

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 $f(n) \geq c \bullet g(n)$ 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 $f(n) \geq c \bullet g(n)$ 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 $f(n) \leq c \bullet g(n)$ for $n \geq n_0$

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