Algorithms and Growth of Functions

- Algorithms
- The growth of functions
- Complexity of Algorithms

Algorithms

- An ***algorithm*** is a finite sequence of precise instructions for performing a computation or for solving a problem.

An algorithm

- is defined on specified inputs and generates an output
- stops after finitely many instructions are executed.

A Recipe is an **Algorithm**

The set of steps to assemble a Piece of Furniture is an **Algorithm.**

Properties of Algorithms

1. Input
2. Output
3. Definiteness
4. Correctness
5. Effectiveness
6. Finiteness
7. Generality

An algorithm has input values from a specified set.

Properties of Algorithms

1. Input
2. Output
3. Definiteness
4. Correctness
5. Effectiveness
6. Finiteness
7. Generality

From each set of input values, an algorithm produces output values from a specified set. The output values are the solution to the problem.

Properties of Algorithms

1. Input
2. Output
3. Definiteness
4. Correctness
5. Effectiveness
6. Finiteness
7. Generality

From each

of i

The steps of an algorithm must be defined precisely.

Properties of Algorithms

1. Input
2. Output
3. Definiteness
4. Correctness
5. Effectiveness
6. Finiteness
7. Generality

From each t
of i

An algorithm should produce the correct output values for each set of input values.

Properties of Algorithms

1. Input
2. Output
3. Definiteness
4. Correctness
5. Effectiveness
6. Finiteness
7. Generality

It must be possible to perform each step of an algorithm exactly and in a finite amount of time.

Properties of Algorithms

1. Input
2. Output
3. Definiteness
4. Correctness
5. Effectiveness
6. Finiteness
7. Generality

An algorithm should produce the desired output after a finite (but perhaps large) number of steps for any input in the set.

Properties of Algorithms

1. Input
2. Output
3. Definiteness
4. Correctness
5. Effectiveness
6. Finiteness
7. Generality

The procedure should be applicable for all problems of the desired form, not just for a particular set of input values.

How to express an Algorithm

C code

```c
int is_prime(int m)
{
int i;
for (i=2; i<m;i++)
 if (m % i ==0)
      return 0;
return 1;
}
```

Java code

```java
class SpecialInt
{
int  m;
boolean  is_prime()
{
for (i=2; i<m; i++)
if (m % i == 0)
      return false;
return true;
}  }
```

Pseudo-code

```
procedure is_prime(m)
   for  i: = 2  to m−1 do
      if  m mod i = 0
      then return(false)
      endif
   endfor
   return(true)
end is_prime
```

Psuedocode

- **Psuedocode** is an intermediate between an English description and an implementation in a particular language of an algorithm.

Advantages of using pseudo-code

- Pseudo-code has a structure similar to most computer languages.
- No need to worry about the precise syntax.
- Not specific to any particular computer language.

Example

- Example: Write an algorithm that finds the largest element in a finite sequence

$s_1, s_2, \ldots , s_n$

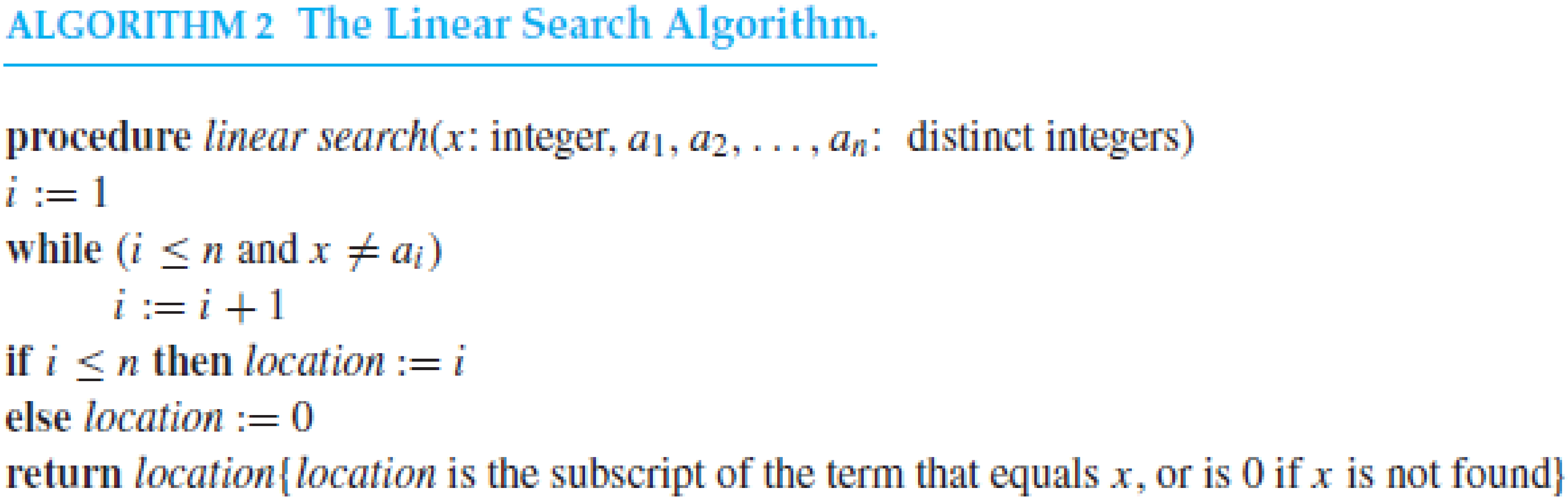
```
procedure find_large(s, n)
        large := $s_1$
        i := 2
        while $i \le n$ do
            if $s_i >$ large then large := $s_i$ endif
            i := i + 1
        endwhile
        return(large)
end find_large
```

Search Algorithms

- **Search**
  - Find a given element in a list. Return the location of the element in the list (index), or 0 if not found.
- **Linear Search**
  - Compare key (element being searched for) with each element in the list until a match is found, or the end of the list is reached.
- **Binary Search**
  - Compare key only with elements in certain locations. Split list in half at each comparison. *Requires list to be sorted.*

Linear Search

- Find the location of an element X in an array of possible unsorted items



ALGORITHM 2 The Linear Search Algorithm.

```
procedure linear search($x$: integer, $a_1, a_2, \ldots, a_n$: distinct integers)
$i := 1$
while ($i \leq n$ and $x \neq a_i$)
        $i := i + 1$
if $i \leq n$ then location $:= i$
else location $:= 0$
return location{location is the subscript of the term that equals $x$, or is 0 if $x$ is not found}
```

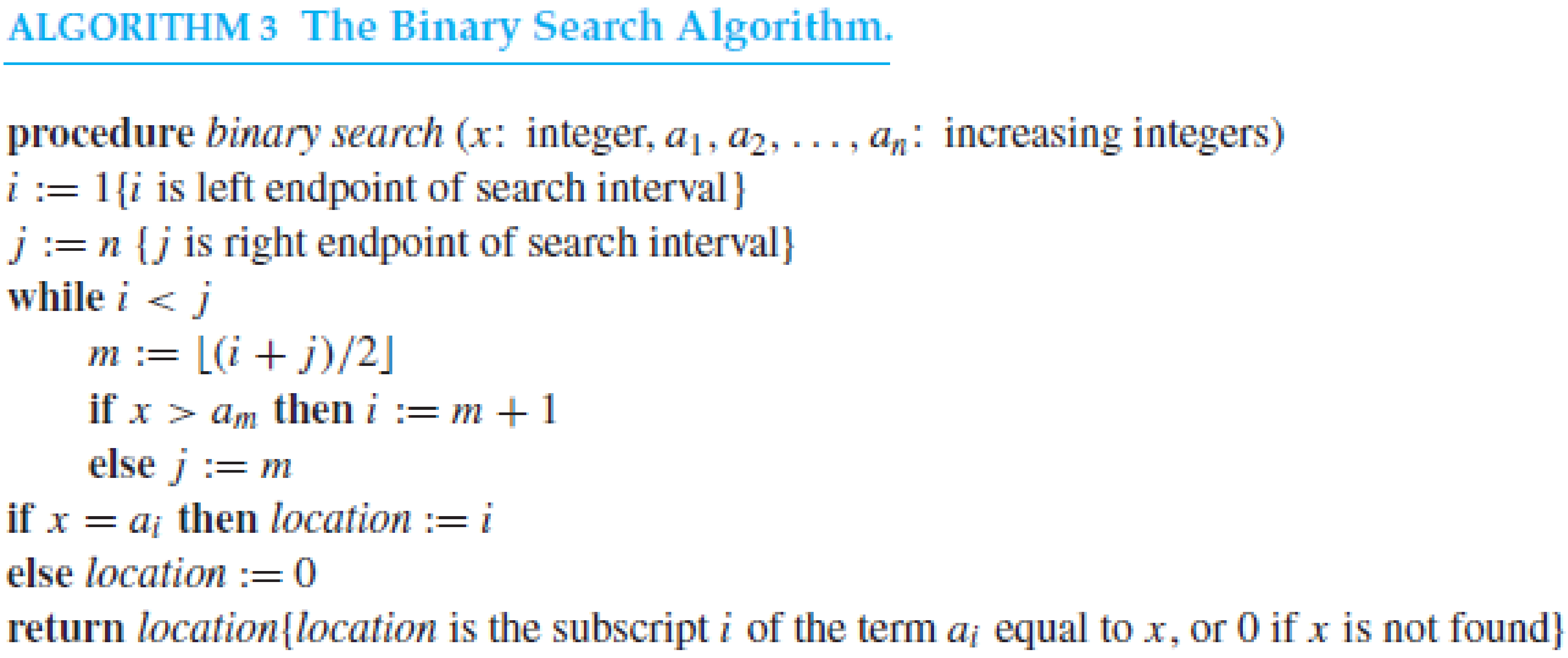
Linear Search Exercise

19, 1, 17, 2, 11, 13, 7, 9, 10, 5, 15, 6, 14, 20, 16, 12, 4, 18, 3, 8

- How many comparisons to find:
  - 17?
  - 21?

Binary Search

- Find the location of an element X in an array of sorted items



ALGORITHM 3 The Binary Search Algorithm.

```
procedure binary search ($x$: integer, $a_1, a_2, \ldots, a_n$: increasing integers)
$i := 1$ {$i$ is left endpoint of search interval}
$j := n$ {$j$ is right endpoint of search interval}
while $i < j$
    $m := \lfloor (i + j)/2 \rfloor$
    if $x > a_m$ then $i := m + 1$
    else $j := m$
if $x = a_i$ then location $:= i$
else location $:= 0$
return location {location is the subscript $i$ of the term $a_i$ equal to $x$, or 0 if $x$ is not found}
```

Binary Search Exercise

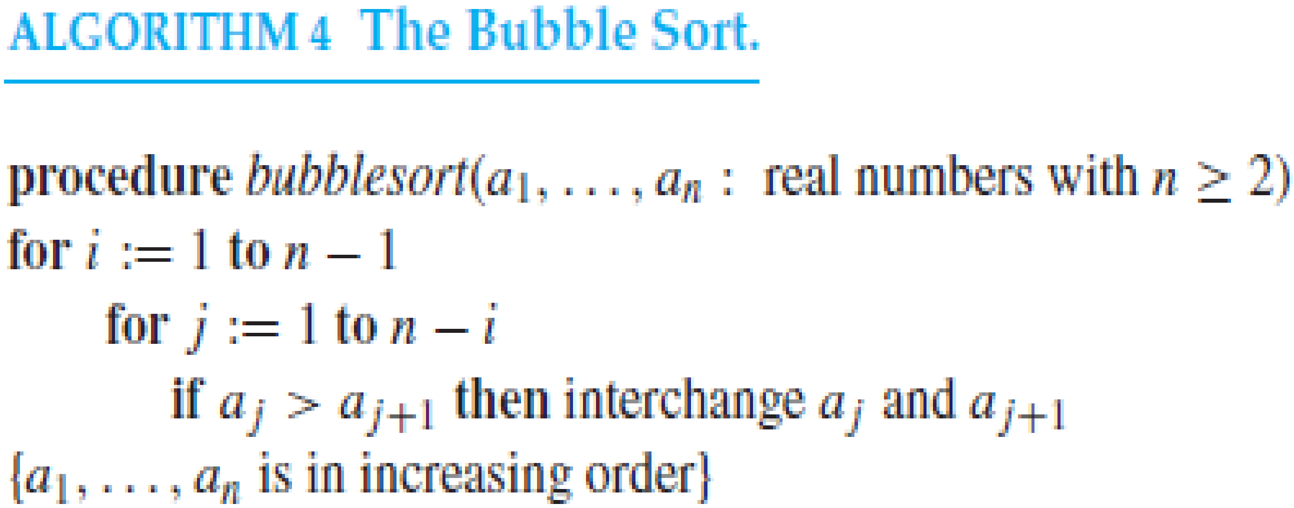
1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20

- How many comparisons to find:
  - Find 7
  - Find 21

Sort

- **Sort**
  - Put the elements of a list in ascending order
- **Bubble Sort**
  - Compare every element to its neighbor and swap them if they are out of order. Repeat until list is sorted.
- **Insertion Sort**
  - For each element of the unsorted portion of the list, insert it in sorted order in the sorted portion of the list.

Bubble Sort



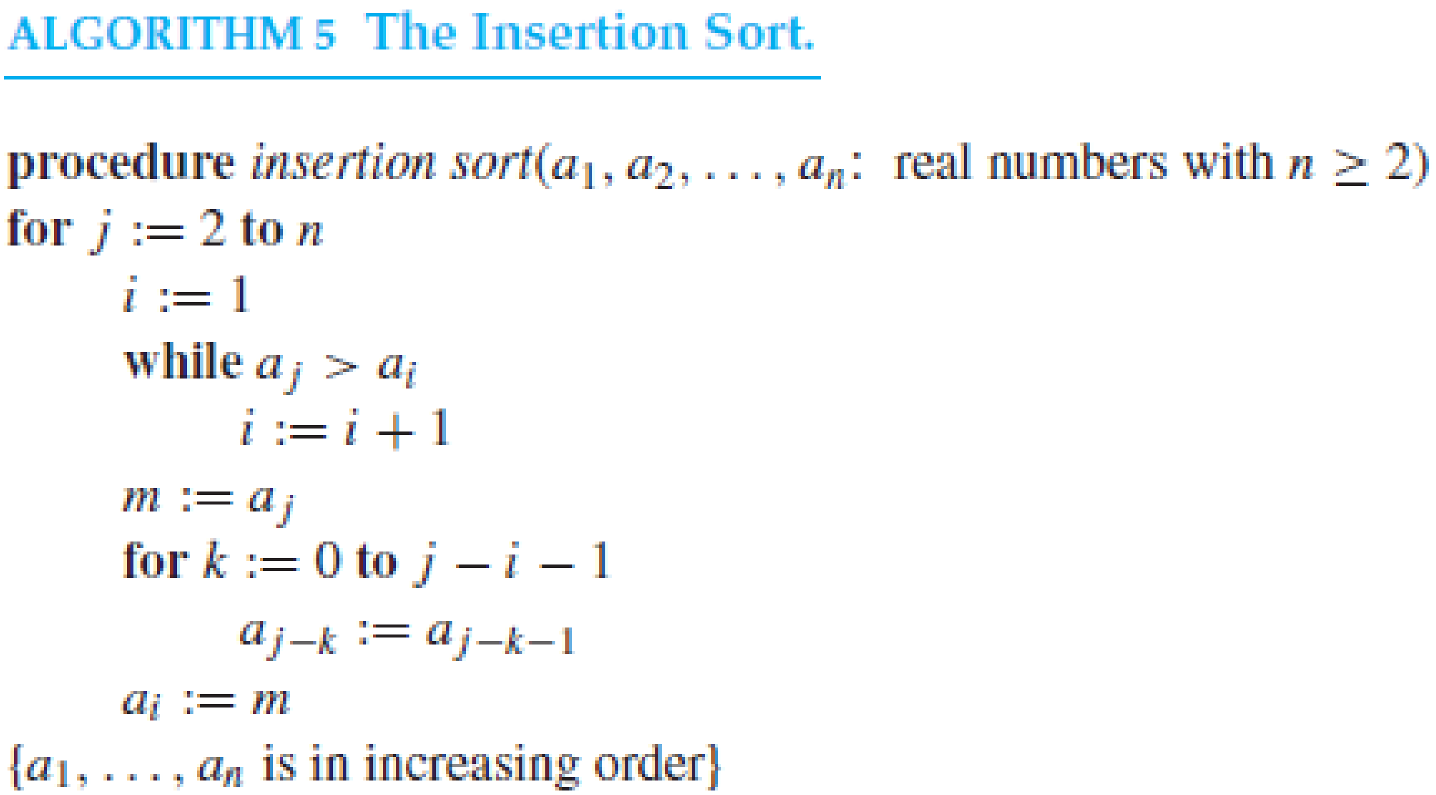
ALGORITHM 4 The Bubble Sort.

```
procedure bubblesort($a_1, \ldots, a_n$ : real numbers with $n \geq 2$)
for $i$ := 1 to $n - 1$
    for $j$ := 1 to $n - i$
        if $a_j > a_{j+1}$ then interchange $a_j$ and $a_{j+1}$
{$a_1, \ldots, a_n$ is in increasing order}
```

Bubble Sort Exercise

10, 2, 1, 5, 3

Insertion Sort



ALGORITHM 5 The Insertion Sort.

```
procedure insertion sort($a_1, a_2, \ldots, a_n$: real numbers with $n \geq 2$)
for $j := 2$ to $n$
    $i := 1$
    while $a_j > a_i$
        $i := i + 1$
    $m := a_j$
    for $k := 0$ to $j - i - 1$
        $a_{j-k} := a_{j-k-1}$
    $a_i := m$
{$a_1, \ldots, a_n$ is in increasing order}
```

Insertion Sort Exercise

10, 2, 1, 5, 3

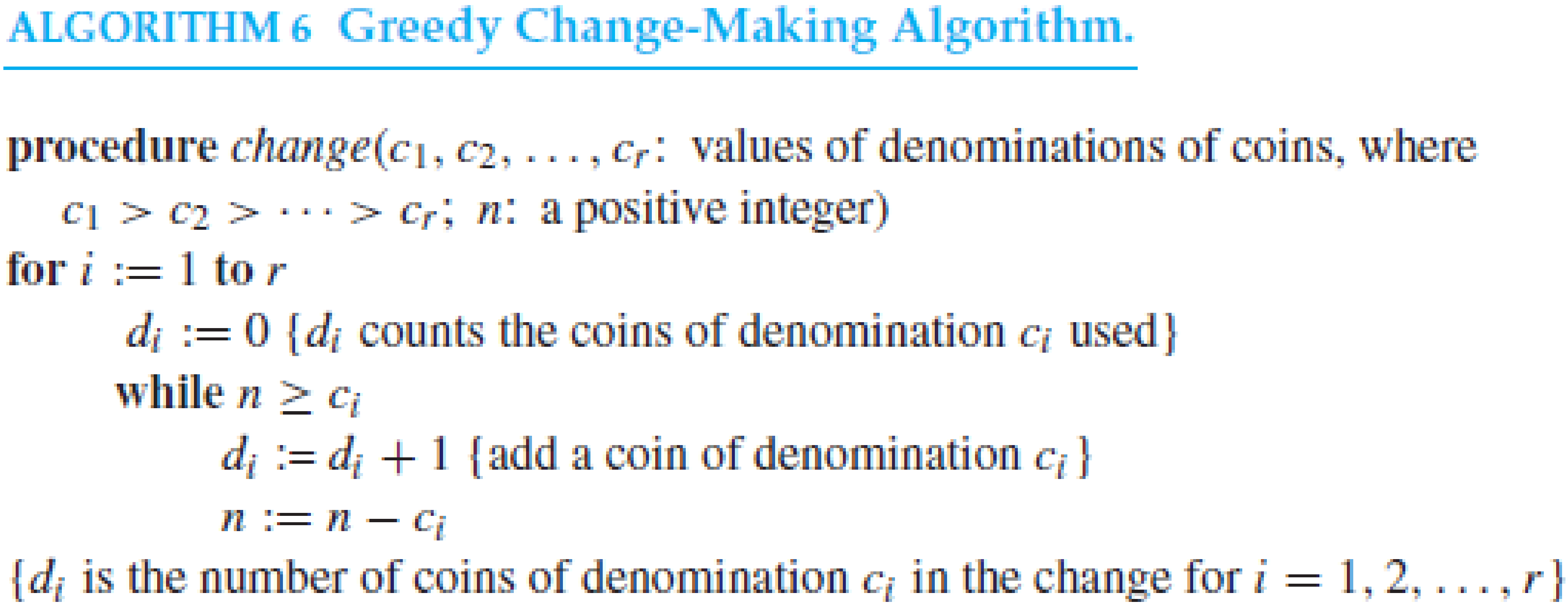
Greedy Algorithms

The goal of an **optimization problem** is to maximize or minimize an objective function.

One of the simplest approaches to solving optimization problems is to select the “best” choice at each step.

Greedy Change-Making

- Give an algorithm for making n cents change with quarters, dimes, nickels, and pennies, and using the least total number of coins.



ALGORITHM 6 Greedy Change-Making Algorithm.

```
procedure change($c_1, c_2, \ldots, c_r$: values of denominations of coins, where
    $c_1 > c_2 > \cdots > c_r$; $n$: a positive integer)
for $i := 1$ to $r$
        $d_i := 0$ {$d_i$ counts the coins of denomination $c_i$ used}
        while $n \geq c_i$
                $d_i := d_i + 1$ {add a coin of denomination $c_i$}
                $n := n - c_i$
{$d_i$ is the number of coins of denomination $c_i$ in the change for $i = 1, 2, \ldots, r$}
```

Make Change

69 cents:

56 cents:

The Halting Problem

Is there a procedure that does the following:

Takes as input *a program and input* to that program and determines whether that program will eventually stop when run on that input, for any program and input

No, there is no such program.

Algorithms and Growth of Functions

- Algorithms
- The growth of functions
- Complexity of Algorithms

Algorithms

- An ***algorithm*** is a finite sequence of precise instructions for performing a computation or for solving a problem.

An algorithm

- is defined on specified inputs and generates an output
- stops after finitely many instructions are executed.

Search Algorithms

- **Search**
  - Find a given element in a list. Return the location of the element in the list (index), or 0 if not found.
- **Linear Search**
  - Compare key (element being searched for) with each element in the list until a match is found, or the end of the list is reached.
- **Binary Search**
  - Compare key only with elements in certain locations. Split list in half at each comparison. *Requires list to be sorted.*

Sort

- **Sort**
  - Put the elements of a list in ascending order
- **Bubble Sort**
  - Compare every element to its neighbor and swap them if they are out of order. Repeat until list is sorted.
- **Insertion Sort**
  - For each element of the unsorted portion of the list, insert it in sorted order in the sorted portion of the list.

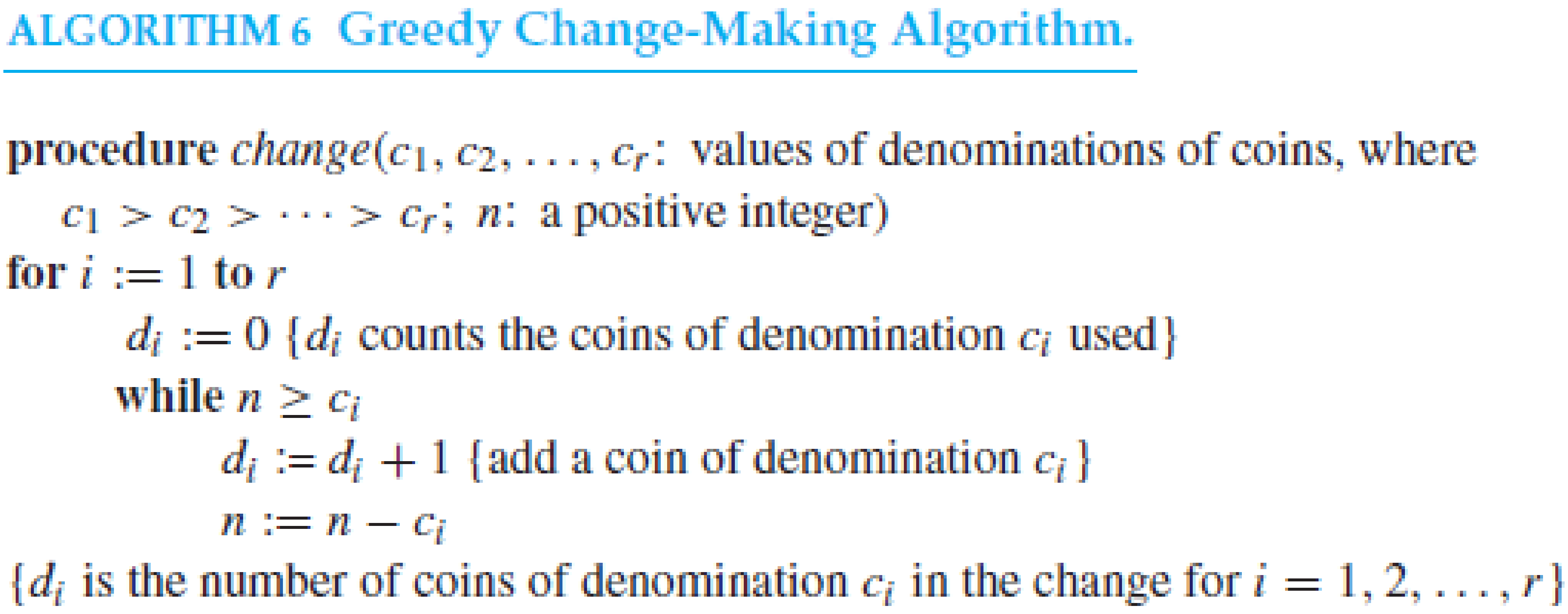
Greedy Algorithms

The goal of an **optimization problem** is to maximize or minimize an objective function.

One of the simplest approaches to solving optimization problems is to select the “best” choice at each step.

Greedy Change-Making

- Give an algorithm for making n cents change with quarters, dimes, nickels, and pennies, and using the least total number of coins.



ALGORITHM 6 Greedy Change-Making Algorithm.

```
procedure change($c_1, c_2, \ldots, c_r$: values of denominations of coins, where
    $c_1 > c_2 > \cdots > c_r$; $n$: a positive integer)
for $i := 1$ to $r$
        $d_i := 0$ {$d_i$ counts the coins of denomination $c_i$ used}
        while $n \geq c_i$
                $d_i := d_i + 1$ {add a coin of denomination $c_i$}
                $n := n - c_i$
{$d_i$ is the number of coins of denomination $c_i$ in the change for $i = 1, 2, \ldots, r$}
```

Make Change

69 cents:

56 cents:

The Halting Problem

Is there a procedure that does the following:

Takes as input *a program and input* to that program and determines whether that program will eventually stop when run on that input, for any program and input

No, there is no such program.

The Growth of functions

- The time required to solve a problem using a procedure depends on:
  - Number of operations used
    - Depends on the size of the input
  - Speed of the hardware and software
    - Does not depend on the size of the input
    - Can be accounted for using a constant multiplier

- The growth of functions refers to the number of operations used by the function to solve the problem.

Complexity of Algorithms

The **complexity** of an algorithm refers to the amount of time and space required to execute the algorithm.

Computing the amount of time and space used without having the actual program requires one to focus on the essential features that affect performance.

Analyzing algorithm find_largest

- Time of execution depends on the number of iterations of the while loop.
- Performance does not generally depend on the values of the elements.
- How many iterations are executed?

n–1

The time needed is linearly proportional to n.

Example

```
for  i := 1  to  n  do
        for  j:=1  to  n  do
                $s_i := s_i + s_j$
```

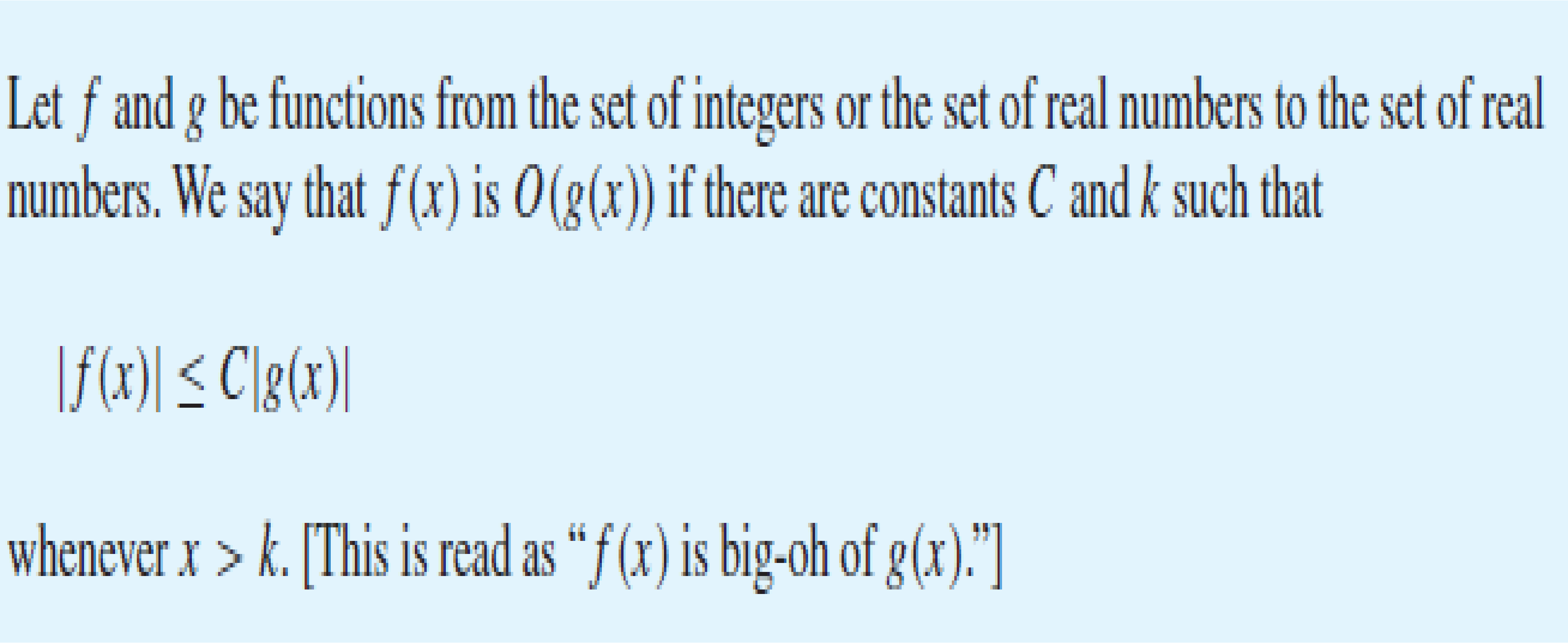
number of iterations executed: n^2

time needed: proportional to n^2

Big-*O* Notation

- Estimate the growth of a function without worrying about constant multipliers or smaller order terms.
  - Do not need to worry about hardware or software used
- Assume that different operations take the same time.
  - Addition is actually much faster than division, but for the purposes of analysis we assume they take the same time.

Big-*O*



Let f and g be functions from the set of integers or the set of real numbers to the set of real numbers. We say that $f(x)$ is $O(g(x))$ if there are constants C and k such that

$$|f(x)| \leq C|g(x)|$$

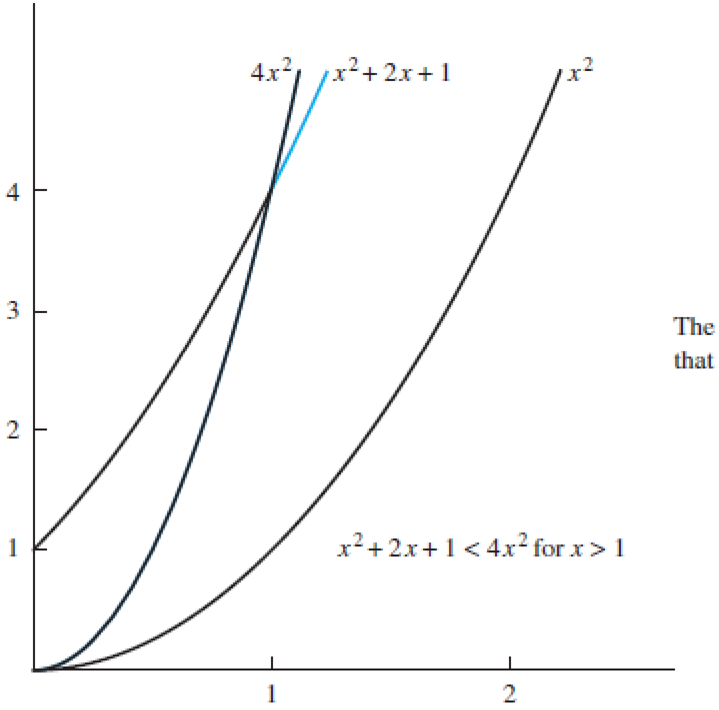
whenever $x > k$. [This is read as "$f(x)$ is big-oh of $g(x)$."]

Example

Show that $f(x) = x^2 + 2x + 1$ is $O(x^2)$.

- $X^2 + 2x + 1 <= x^2 + 2x^2 + x^2$ for $x >= 1$
- $x^2 + 2x^2 + x^2 = 4x^2$

- Witness
- C = 4
- K = 1



The part of the graph of $f(x) = x^2 + 2x + 1$ that satisfies $f(x) < 4x^2$ is shown in blue.

Example

Show that n^2 is not $O(n)$.

- Assume n^2 is $O(n)$
- Then $\exists$ C,k $\forall$ n>k, $n^2 \le Cn$
- $n \le C$
- But no constant is bigger than all n
- contradiction

Big-O for Polynomials

Let $f(x) = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0$.

Then, $f(x)$ is $O(x^n)$.

Example

- Give a big-O estimate for
- f(x) = $5x^2-18x+20$

- Solution
  - $5x^2-18x+20 \le 5x^2+20$ for $x>0$
  - $5x^2+20 \le 5x^2+20x^2$ for $x>1$
  - $5x^2+20x^2 = 25x^2 \le Cg(x)$ for $x>1$
  - Let $g(x)=x^2$
  - $\boldsymbol{f(x)}$ **is** $\boldsymbol{O(x^2)}$**.** $\boldsymbol{C=25,\ k=1}$

Example

- Give a big-O estimate for the sum of the first n positive integers

- Solution
- $1+2+\cdots+n \leq n+n+\cdots+n=n^2$
- $1+2+\cdots+n$ is $O(n^2)$, $C=1, k=1$

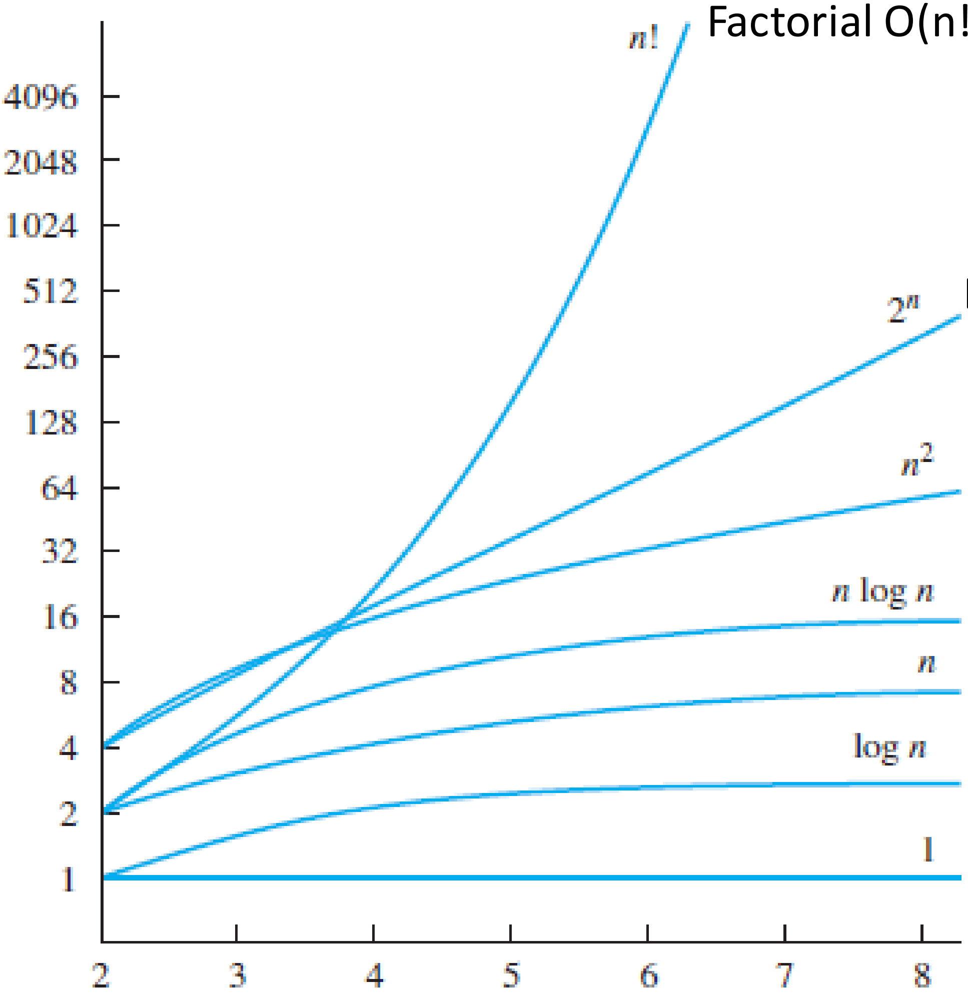
Example

- Give a big-O estimate for the factorial function f(n)=n!
- Give a big-O estimate for the logarithm of the factorial function

- Solution
- $n! = 1 \cdot 2 \cdot 3 \cdot \cdots \cdot n \leq n \cdot n \cdot n \cdot \cdots \cdot n = n^n$
- $n!$ is $O(n^n)$

- $\log(n!) \leq \log(n^n) = n \log n$
- $\log(n!)$ is $O(n \log n)$

Basic Growth Functions



Factorial O(n!)

Exponential $O(n^n)$

Polynomial $O(n^2)$

Linearithmic O(n log n)

Linear O(n)

Logarithmic O(log n)

Constant O(1)

Useful Big-O Estimates

- n^c is $O\left(n^d\right)$, but n^d is **not** $O(n^c)$, $d > c > 1$
- $(\log_b n)^c$ is $O\left(n^d\right)$, but n^d is **not** $O\left((\log_b n)^c\right)$, $b > 1, c, d > 0$
- n^d is $O(b^n)$, but b^n is **not** $O\left(n^d\right)$, $d > 0, b > 1$
- b^n is $O(c^n)$, but c^n is **not** $O(b^n)$, $c > b > 1$

The Growth of Combinations of Functions

- Suppose $f_1(n)$ is $O(g_1(n))$ and $f_2(n)$ is $O(g_2(n))$
  - $(f_1 + f_2)(n)$ is $O(\max(g_1(n), g_2(n)))$
    - If $g_1(n) = g_2(n) = g(n)$, then $(f_1 + f_2)(n)$ is $O\left(g(n)\right)$

  - $(f_1 f_2)(n)$ is $O(g_1(n) g_2(n))$

Example

- Give a big-O estimate for
  $$f(n) = 3n\log(n!) + (n^2 + 3)\log n$$

- $O(n^2 \log n)$

Big-Ω

- Big-O
  - $\exists C, k \ \forall n > k \ \ f(n) \leq Cg(n)$
- Big- Ω (big omega)
  - $\exists C, k \ \forall n > k \ f(n) \geq Cg(n)$
  - C must be **positive.**
  - $f(n)$ is $\Omega(g(n)) \leftrightarrow g(n)$ is $O(f(n))$
  - "$f(x)$ is bounded below by $g(x)$"

Example

- Show that $8x^3 + 5x^2 + 7$ is $\Omega(x^3)$

- $8x^3+5x^2+7 \geq 8x^3$ for x > 0

- C=8, k =0

Big-Θ

- Big- Θ (big theta)
  - $f(n)$ is $O\big(g(n)\big)$ and $\Omega\big(g(n)\big)$
  - $f(n)$ is $O\big(g(n)\big)$ and $g(n)$ is $O\big(f(n)\big)$
  - $f(n)$ is $\Theta\big(g(n)\big) \leftrightarrow \mathrm{g}(n)$ is $\Theta\big(f(n)\big)$
  - $\exists C_1, C_2, k \;\; \forall n > k \quad C_1 g(n) \le f(n) \le C_2 g(n)$
  - $f(n)$ is of *order* $g(n)$
  - $f(n)$ and $g(n)$ are of the *same order*

Example

- Show that $3x^2 + 8x\log x$ is $\Theta(x^2)$

- Big-o
- $3x^2 + 8x \log x <= 11x^2$
  - $C=11, k=1$

- Big-omega
- $x^2 <= 3x^2 + 8x \log x$

Big-Θ for Polynomials

- Let $f(x) = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0$.
- Then, $f(x)$ is of order x^n.
  - "$f(x)$ is bounded [above and below] by $g(x)$"
- Example:
  - $3x^8 + 10x^7 + 221x^2 + 1444$ is of order x^8

1. LINEAR SEARCH (p. 194 of text)

INPUT: List of n numbers, and
a number to search for

OUTPUT: Position of number in list,
or 0 if number is not in list.

LIST

7	1	6	4	8	3	21

$i = 1$ $i = 7$ $n = 7$

Say we are searching for 9

1. Move loop index i from 1 to 7
2. check List[i] for equality with 9

[this check is a COMPARISON OPERATION. A comparison is often the "basic operation" we count in algorithms.

ie. how many comparisons in this linear search?

3. If we have a match, output the index position i.

If after $i = 7$ there is no match, output 0.

-1

Q: How many comparisons in step 2?

A: 7 comparisons

For an n-elemt list, n comparisons.

Q: why only count comparisons? why not also count other stuff in loop, such as $i \leftarrow i+1$ etc.?

A:

```
      i ← 0
next: i ← i + 1
      If L[i] is 9 then
          output i
          stop
      Else
          If (i < n) then go-to next
          Else stop.
```

we only count how many times this is done. Why?

old programming languages used to have this construct. (Bad idea to use it!)

Suppose you have a loop :

```
n      [ statement 1
times  [ statement 2
       [     :
       [ If (List(i) == item) then.....
       [     :
       [ statement 99
```

Say time to execute each statement = 1

⇒ time to make one pass thru loop = 100 x 1

⇒ " " run complete loop = $n \times \underbrace{100} \times \underbrace{1}$

depends on speed of machine

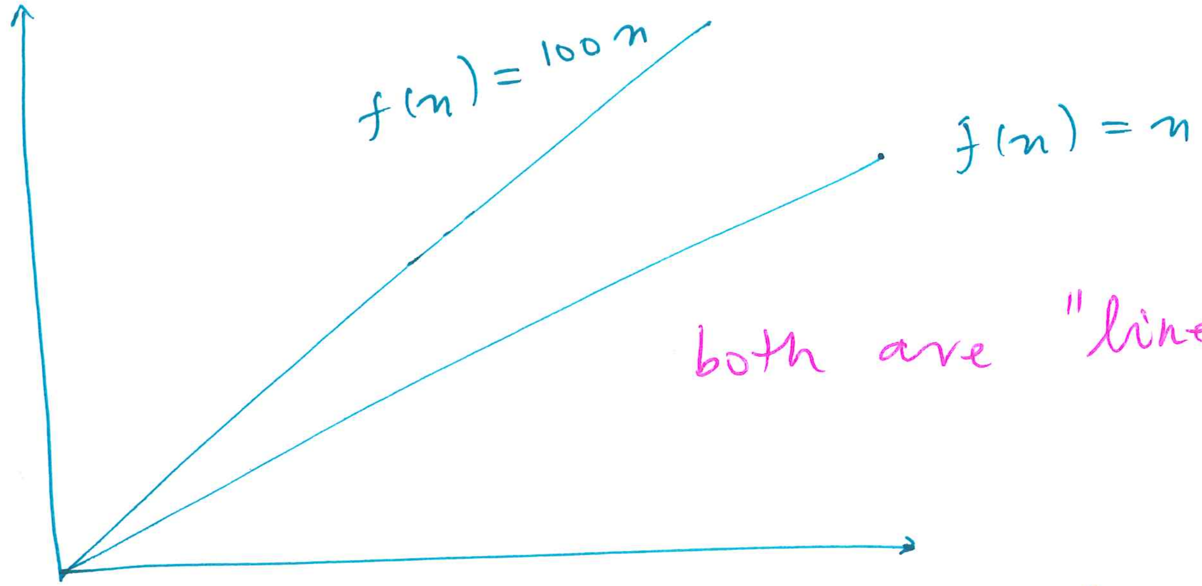
n is input to program

Some constant number of statements in program

$$\underbrace{f(n)}_{\text{run-time of algorithm (or program)}} = n \cdot 100 \cdot 1$$

Q: What really influences $f(n)$?

A: n
not 100
not 1.



both are "linear"

To capture the behaviour of $f(n)$ we simply focus on a basic operation (e.g. if List(i) == 9) inside the loop, and ignore all the other statements in the loop.

So for Linear Search,

$f(n) \hat{=}$ run-time

$=$ total # of basic operations (and here a basic opn. is a "comparison")

$= n$.

$$\boxed{\text{Linear Search :} \quad f(n) = n}$$

<u>Binary Search</u> : (p. 195 of text)

INPUT : List of n items in <u>increasing</u> order, item to search for.

OUTPUT : List posn. of item, if it is in list.
or output 0 if not in list.

Say $n = 9$, searching for 12 in

List :

1	3	5	7	9	11	14	19	20

step 1 : jump to item in middle of list, compare it to 12.

$$\text{middle} \leftarrow \left\lfloor \frac{1+n}{2} \right\rfloor = 5$$

<u>Basic Idea</u>: Is 12 in List[5]?

YES ⇒ Found it! Output posn. 5.

No ⇒ Our original problem of size n now becomes a new problem of size $\frac{n}{2}$. How?

(a) Because list L is ordered, and

(b) $12 > 9$, we only need to now work with the sublist on the right of number 9.

New problem : Find if 12 is in

List :

11	14	19	20

Q: When will the algorithm finish?

A: we find 12 or we output 0.

Q: How many basic operations?

A: count the # of comparisons.

(comparison = basic operation)

Here is the idea:

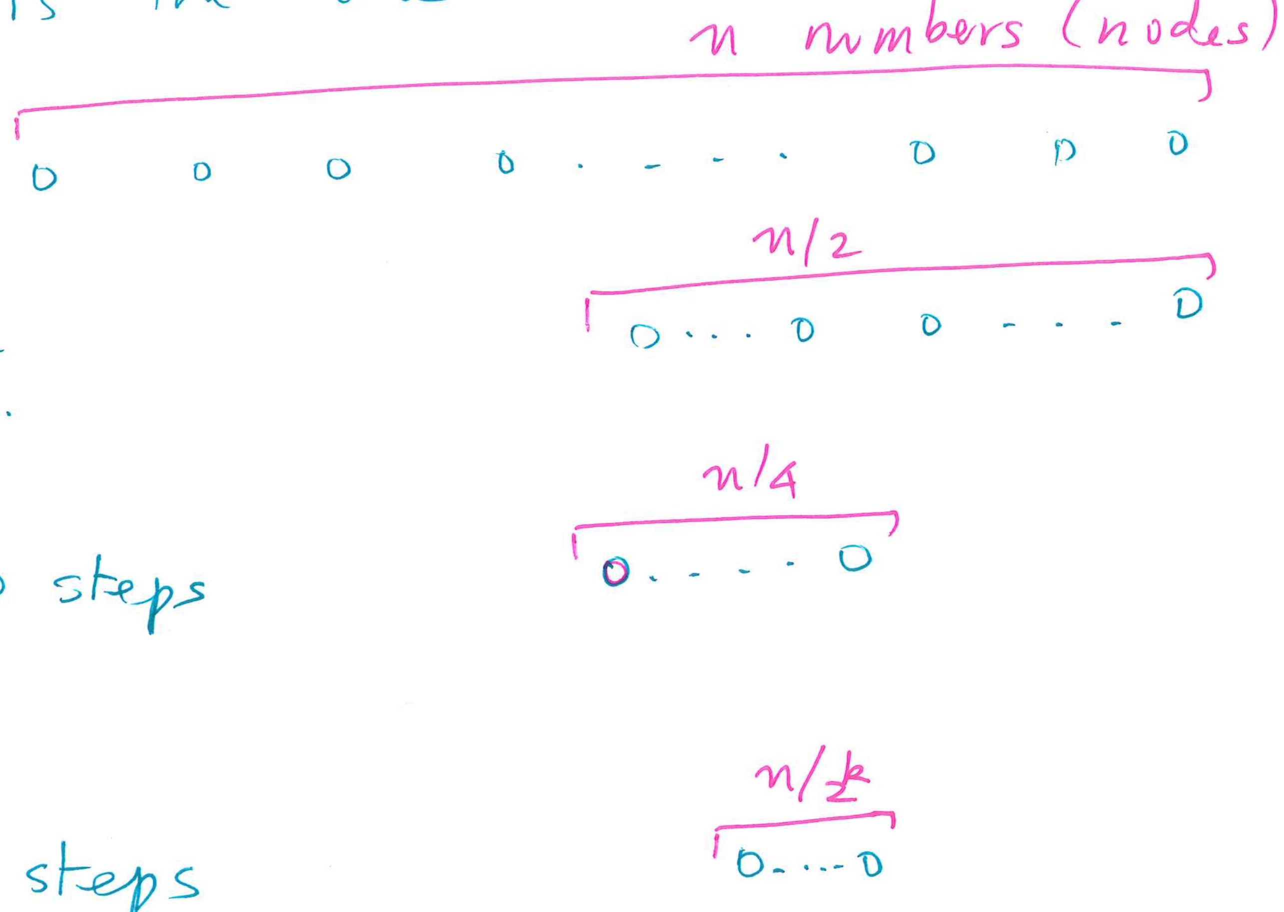
List:

In one step.

In two steps

⋮

In k steps



In k steps we need to end up with a list of size 1 and settle the problem with "found" or "not found".

But what is value of k ?

Well, $\frac{n}{2^k} = 1$

we need this!

This $\Rightarrow$ $n = 2^k$

$\Rightarrow$ $k = \log_{\textcircled{2}} n$

NOTE: We made a implicit assumption i.e., list L had n as some power of 2 at the start.

But in reality, n could be any number, not necc. a power of 2.

So our "analysis" is APPROXIMATE.

For arbitrary n,

$\left.\begin{array}{l}\text{\# of basic opns} \\ \text{reqd. by Binary Search}\end{array}\right\} \approx \log_2 n$

$$\boxed{f(n) \simeq \log_2 n}$$

# of comparisons, or run-time of Binary Search.

SORTING ALGORITHM

INPUT : A list L of numbers,
n of them

OUTPUT : List L is sorted order
(increasing or decreasing)

usually this.

BUBBLE SORT. (p. 197 of text)

Basic Idea:

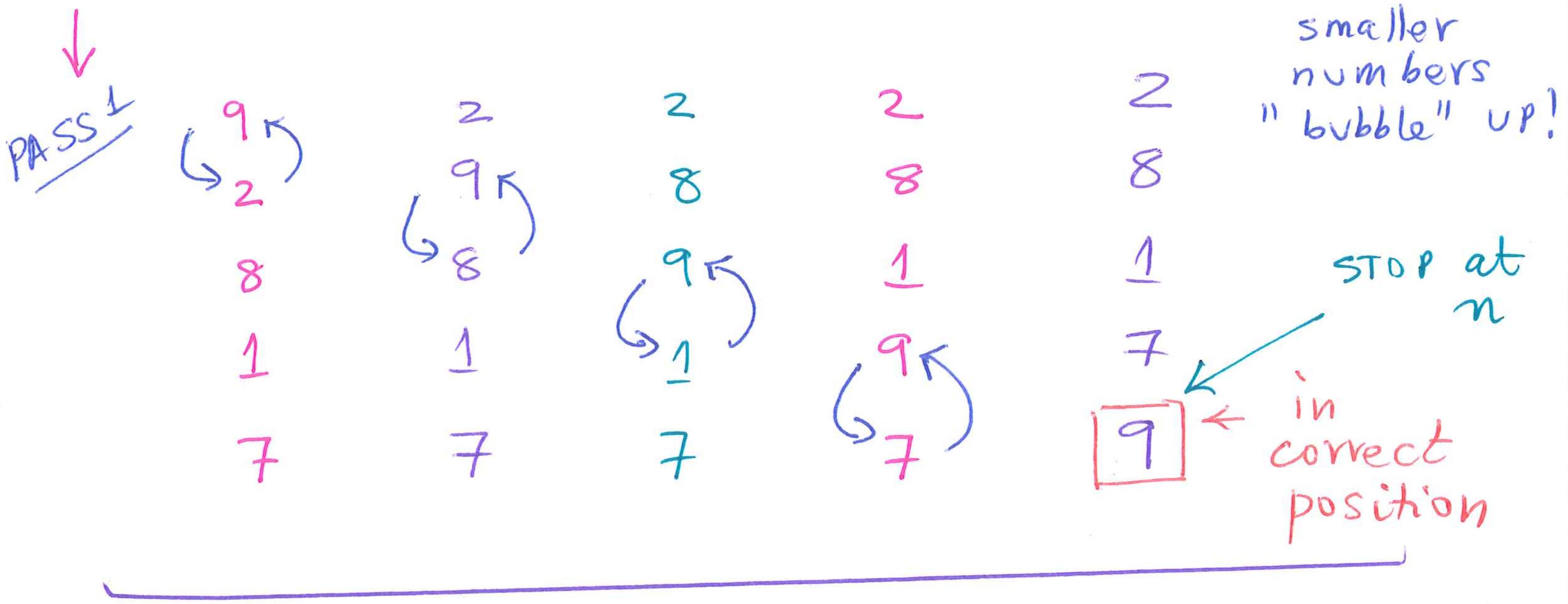
Pass 1 : compare (n-1) pairs. Swap if necessary

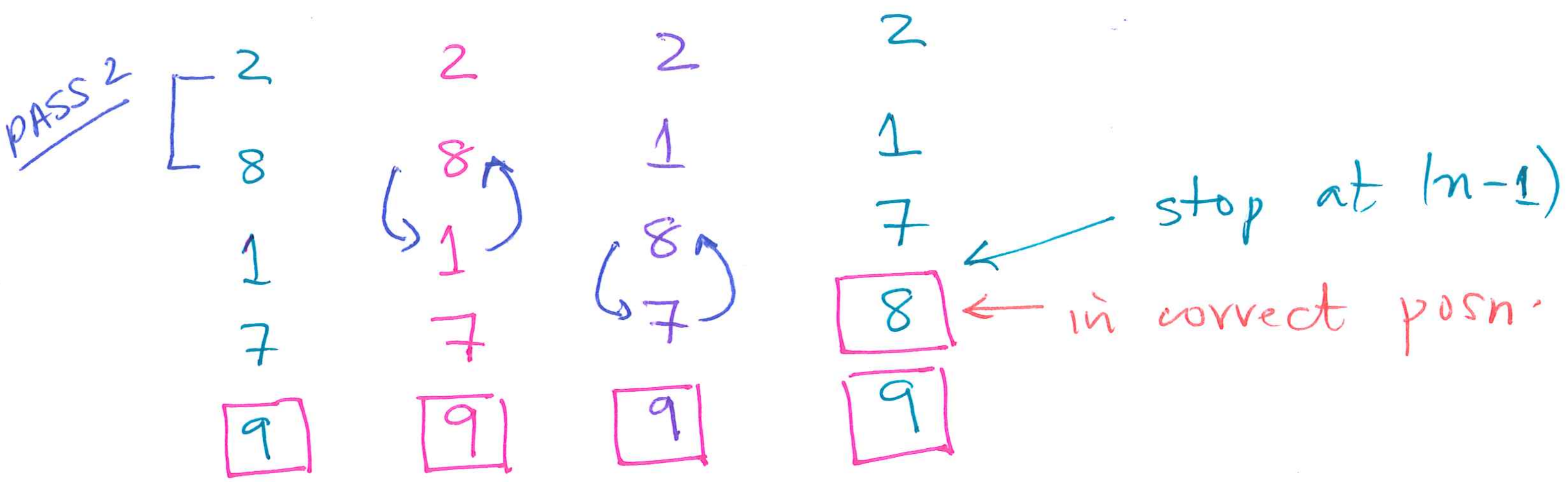
Pass 2 : compare (n-2) pairs. Swap if necessary

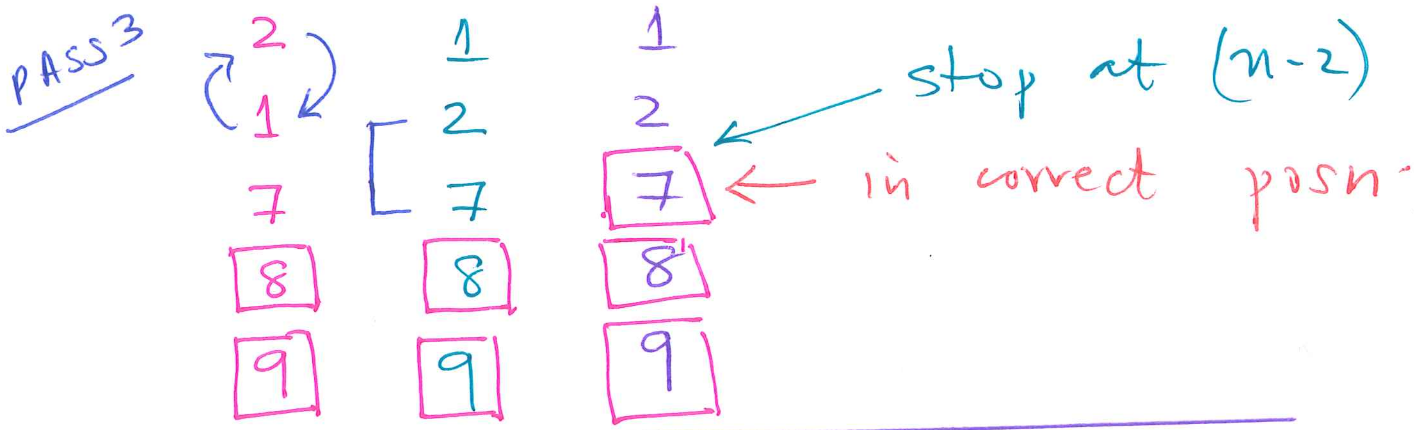
⋮

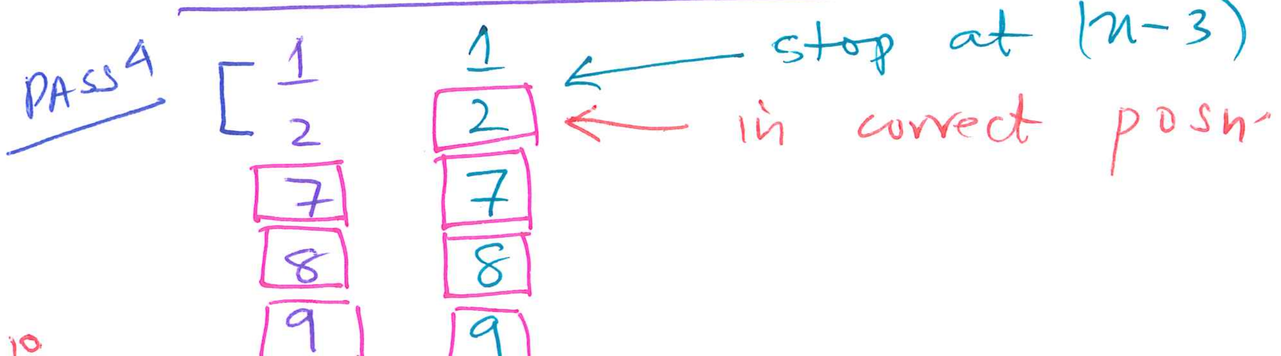
pass (n-1) : compare 1 pair. Swap if necessary.

List L : 9 2 8 1 7


PASS 1
9 2 8 1 7
2 9 8 1 7
2 8 9 1 7
2 8 1 9 7
2 8 1 7 9
smaller numbers "bubble" up!
STOP at n
in correct position


PASS 2
2 8 1 7 9
2 8 1 7 9
2 1 8 7 9
2 1 7 8 9
stop at (n-1)
in correct posn.


PASS 3
2 1 7 8 9
1 2 7 8 9
1 2 7 8 9
stop at (n-2)
in correct posn.


PASS 4
1 2 7 8 9
1 2 7 8 9
stop at (n-3)
in correct posn.

List L has n elements.

Pass # 1 : $(n-1)$ comparisons

Pass # 2 : $(n-2)$ "

⋮

pass # $(n-1)$: $\underline{1}$ comparisons.

Total # of comparisons $= 1 + 2 + 3 + \cdots + (n-1)$

$$= \frac{(n-1)n}{2}.$$

$$\boxed{f(n) = \frac{(n-1)n}{2}}$$

# of comparisons or run-time of B~~inary~~UBBLE Sort alg.

NOTE:

If you keep track of the # of swaps made in each pass, then you can stop the algorithm in any pass where 0 swaps are made. This means L is sorted!

INSERTION SORT

INPUT : List L of n numbers

OUTPUT : Sorted list L of n numbers.

Basic Idea :

Imagine a hand of 12 cards
J, Q, 2, 5, Ace, K, 3, 7, 9 etc.
↑
Going from left to right

2, J, Q, 5, Ace, K, ...
↓
2, 5, J, Q, Ace, K, . . .

K.
2, 5, J, Q, ~~Ace~~ [illegible], ...
2 5 J Q K A

As you move from left to right you will get sorted lists of size 2, size 3, size 4, etc.

If list L has n elements,

$$\text{Total \# of comparisons} = \underset{\substack{\uparrow \\ \text{step 1}}}{2} + \underset{\substack{\uparrow \\ \text{step 2 etc.}}}{3} + 4 + \cdots + \underset{\substack{\uparrow \\ \text{step } (n-1)}}{n}$$

$$= \widetilde{(1+2+\cdots+n)} - 1$$

$$= \frac{n(n+1)}{2} - 1.$$

$$\boxed{f(n) = \frac{n(n+1)}{2} - 1}$$

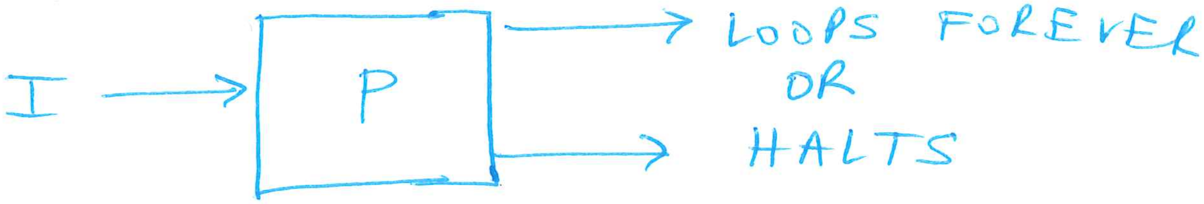
INSERTION SORT
# of compns.
or run-time.

<u>Note:</u> In all 4 examples we used "worst-case" analysis.

In insertion sort, for example, if list was mostly sorted, then searching and rearranging would take less time than worst case.

HALTING PROBLEM [ALAN TURING 1940]

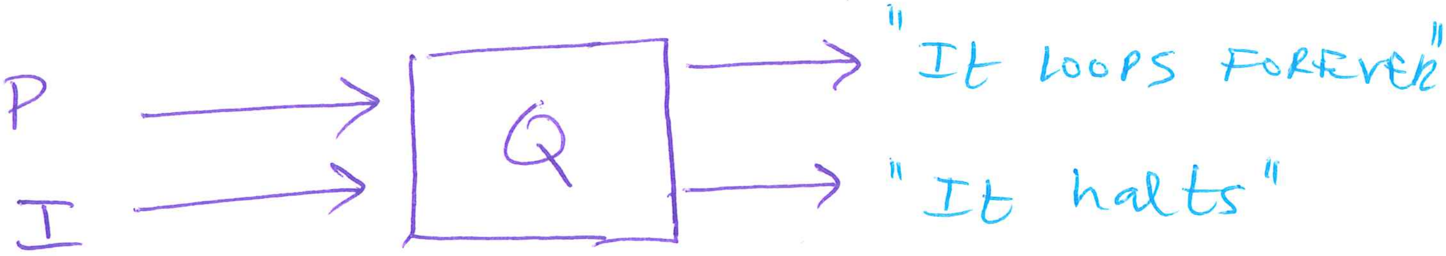
Amy writes program P that runs on input I.



So Amy's program does one of two things. But which?

Her friend Poindexter says he can writ a program Q that takes (P, I) as input and tell what happens.

Poindexter's claim:



Q: Is he correct? Can he do it?

Let's write our own program E (E for Einstein) and wrap it around Poindexter's program Q.

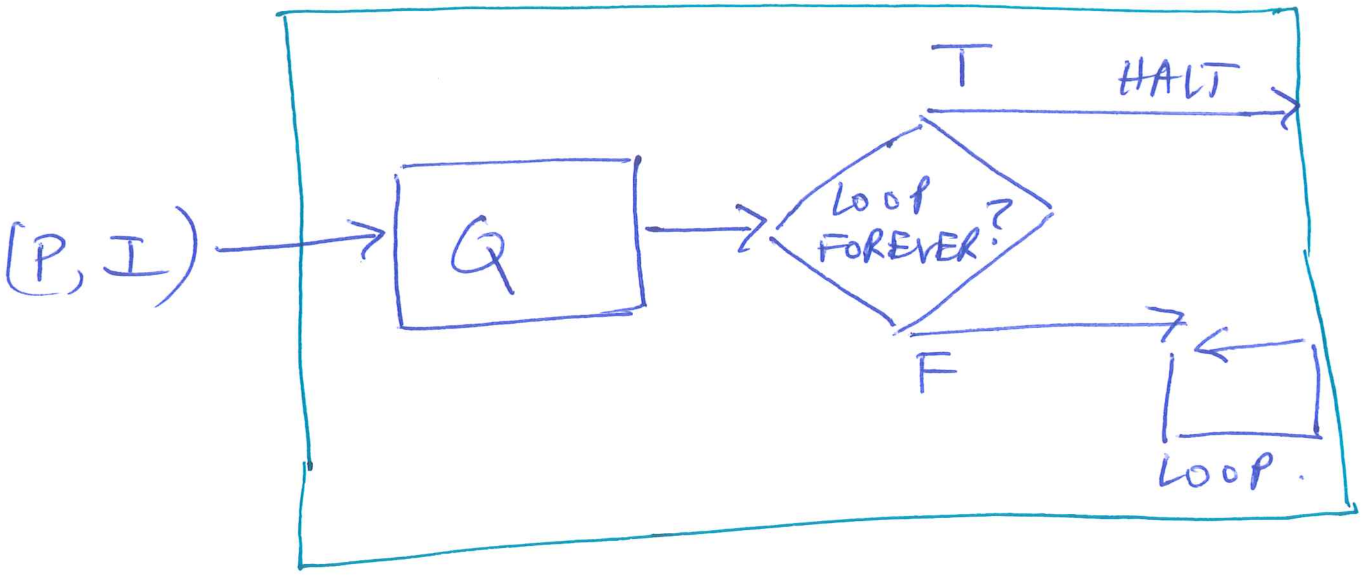
```
function E() {
    If ( Q() == "loop forever")
        return;  // halt
    Else
        While(true);  // loop forever
}
```

What does program E() do?

1. If Q() outputs "loop forever" then E() halts.

2. If Q() outputs "it halts", then E() loops forever.

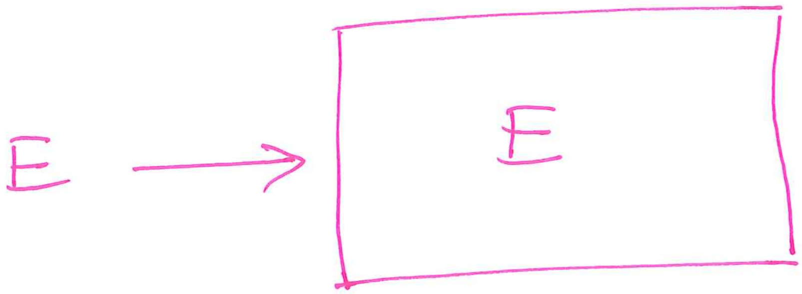
E() does the opposite of Q()!



This is E( )

Now E is a program like P.
What happens if we feed
E to itself?

E takes a program E as
input and watches it run.



What will
E conclude?

1. If input program E loops forever, then E says it halts.
2. If input program E halts, then E says it loops forever.

We constructed an input E that made program Q fail.

ie. there can be cases where Q gives the wrong answer.

So Poindexter was wrong!

Conclusion

It is not possible for any program Q to examine another program P and decide if it will halt or loop forever.

This is called the Halting Problem.

GROWTH OF FUNCTIONS

1. $\mathcal{A}$ is some algorithm. Run it on an input of size n and get its run-time $f(n)$.
(i.e. get some expression for $f(n)$)

2. Things we do not care about:
   - hardware
   - software
   - speed of computer
   - constants

3. Things we care about:
   (a) How fast does $f(n)$ grow?
   (b) Can we relate its growth to the growth of some function we know?

E.g. is $f(n)$ linear?
is $f(n)$ cubic?

4. When we analyze A to get $f(n)$ we count "basic operations" (we focus on one particular operation such as a "comparison" and count how many times A does comparisons to solve the problem for an input of size n)

Note: In chap 3.2, the text assumes $f(\,)$ is any function, not necessarily the run-time of an algorithm.

f: Integers → Real numbers

or f: Real #s → Real #s

Because of this, the definitions in chap 3.2 are a little more general than what we present here. We will focus on $f(n) =$ run-time of A.

N = set of natural numbers = $\{0, 1, 2, \ldots\}$

n is size of A's input, $n \in N$

R = set of real numbers, positive

$f : N \longrightarrow R$ $\Big\}$ run-time of A is some real number.

Q: How fast does $f()$ grow?

Idea: Can we relate $f()$'s growth rate to the growth rate of some other function?

Let $g()$ be this other function.

$g : N \longrightarrow R$

We do not care where $g()$ comes from.

Perhaps $g()$ is:

(a) the ~~g~~ run-time of some other algorithm B, or

(b) just some "reference" function we know: n, n^2, n^3, $\log n$, 2^n

Defn. [Big-Oh]

If $\exists\ c > 0$ and integer $k > 0$ s.t

$$f(n) \leq c\, g(n) \quad \forall\ n \geq k$$

then

$$f(n) \in \underbrace{O(g(n))}.$$

set of functions
that are $O(g(n))$
(ie. "order of $g(n)$")
grow "slower" than $g(n)$

Note: Sometimes you see
the notation

$$f(n) = O(g(n))$$

Do not use it. The idea
"seems" okay, but

LHS: function (f)

RHS: some set

function = set ! X

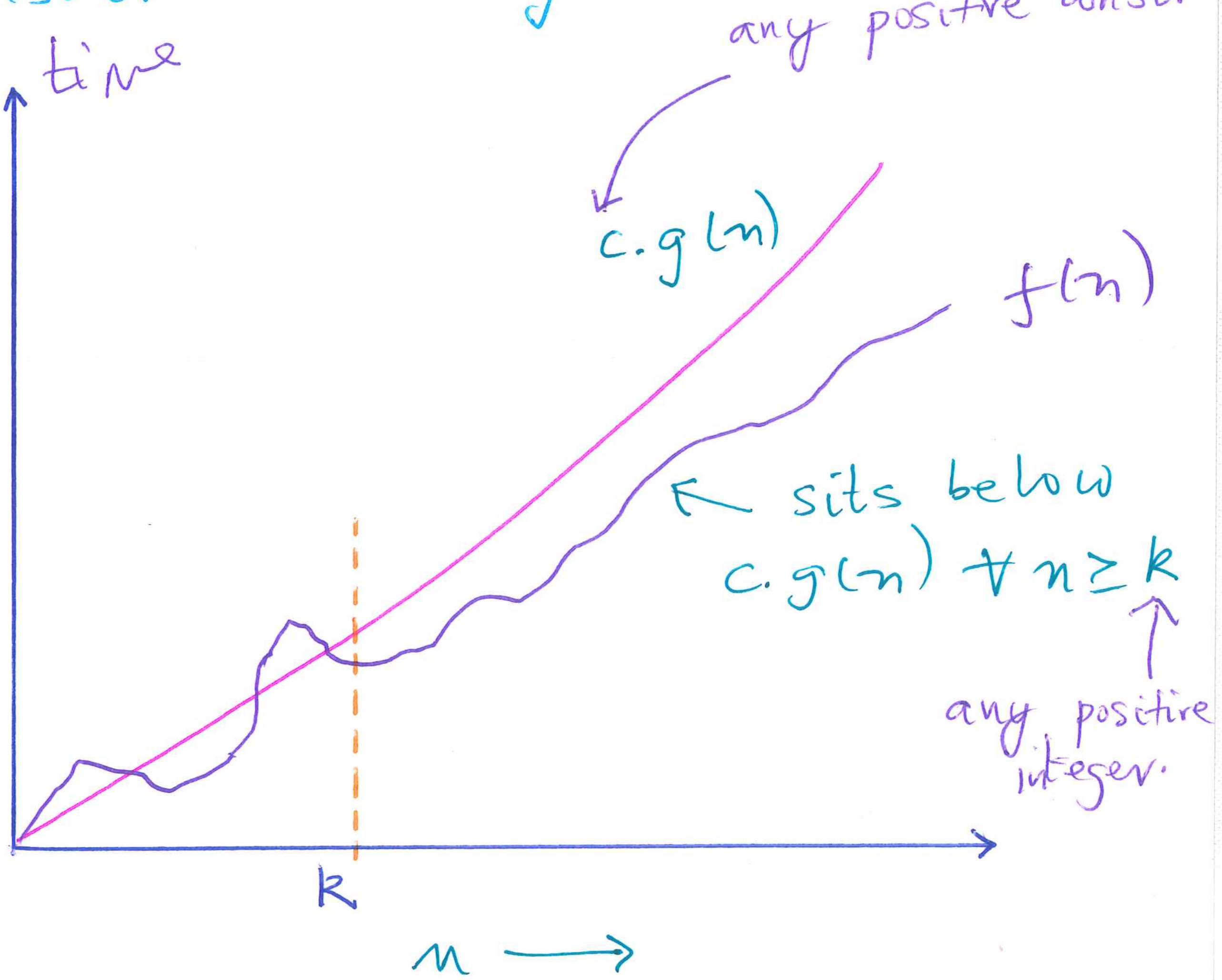
$f(n) \in O(g(n))$ means:

"for large n, $g(n)$ or some constant times $g(n)$ grows faster than $f(n)$".

$\Rightarrow$ $c \cdot g(n)$ acts as an upper bound for $f(n)$, $\forall\, n \geq k$

$\Rightarrow$ we know f cannot grow faster than g.

1.
(Example)



S.

2. $n^3 \notin O(n^2)$. why?

Use contradiction.

Assume $n^3 \in O(n^2)$

$\Rightarrow \exists\ c > 0$ and integer $k \geq 0$ s.t.

$$n^3 \leq c \cdot n^2 \quad \forall\ n \geq k$$

Since $n > 0$, divide both sides by n

$$\Rightarrow \quad n \leq c \quad \forall\ n \geq k$$

a contradiction! //

3. $f(n) \in O(g(n)) \not\Rightarrow g(n) \in O(f(n))$

$$f(n) = 4n^4, \quad g(n) = n^5.$$

$f(n) \in O(g(n))$?

YES! Because:

$$4n^4 \leq 1 \cdot n^5 \quad \forall\ c > 4$$
$$\forall\ n > 1$$

$\uparrow$

use $c = 4$

Q: Does $g(n) \in O(f(n))$?

Use contradiction.

Assume $g(n) \in O(f(n))$

i.e., $n^5 \le c \cdot 4n^4 \quad \forall n > k, c > 0$

$\Rightarrow \quad n \le 4c \quad \forall n > k, c > 0$

A contradiction!

So $g(n) \notin O(f(n))$

Theorem.

Let $d > 0$ be any constant.

$$f(n) = a_n d^n + a_{n-1} d^{n-1} + \cdots + a_1 d + a_0$$

for $a_0, a_1, a_2, \ldots, a_n \in R$.

Then $f(n) \in O(d^n)$.

Proof:

$$f(n) = a_n d^n + a_{n-1} d^{n-1} + \cdots + a_1 d + a_0$$
$$\le a_n d^n + a_{n-1} d^n + \cdots + a_1 d^n + a_0$$
$$= d^n (a_n + a_{n-1} + \cdots + a_1 + a_0)$$

$\Rightarrow f(n) \le c \cdot d^n$ for any c,

$c \geq \sum_{j=0}^{n} a_j$

HW.

Read Thm 1 on p 209 of text.

The author does it for

$$f(x) = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0$$

↑

note x is in R, instead of N

so $f(x)$ is any function, not necc. the run-time of an algorithm.

When $f(x)$ is any function, values can be positive or negative, so $|f(x)|$ is used to capture "size" of function $f()$ without sign.

4. S_n = sum of 1^{st} n natural #s

$$= 1 + 2 + 3 + 4 + \cdots + n$$

$$\le n + n + n + n + \cdots + n$$

$$= n \cdot n = n^2$$

i.e. $S_n \le \overset{c}{1} \cdot n^2 \quad \forall\, n \ge \overset{k}{1}$

$\Rightarrow S_n \in O(n^2)$

5. Let $f(n) = n!$

How fast does $f(n)$ grow?

$n! = 1 \cdot 2 \cdot 3 \cdot 4 \cdots n$

$0! = 1$.

$$n! = 1 \cdot 2 \cdot 3 \cdot 4 \cdots n$$

$$\le n \cdot n \cdot n \cdot n \cdots n$$

$$= n^n$$

$\Rightarrow \boxed{n! \le n^n}$ for $c = 1$, $k \ge 1$

Take "log" of both sides

(usually we mean $\log_2$ but we don't have to be specific until we compute something)

$$\log n! \leq \log n^{\boxed{n}} = n \log n$$

$$\Rightarrow \log n! \in O(n \log n)$$

$c = 1$, $k = 1$

---

6. Show $f(n) = n \in O(2^n)$

$$n \leq c.2^n \text{ for } c = 1, \ \forall n \geq \underline{1}$$

k.

$$\Rightarrow n \in O(2^n)$$

7. If $n \in O(2^n)$ then it follows that

$$\log n \in O(n)$$

Why?

$n \le c \cdot 2^n \quad \forall n \ge 1$, for $c = 1$

i.e. $n \le 2^n \quad \forall n \ge 1$

Take "$\log_2$" of both sides

$$\log_2 n \le n \boxed{\log_2 2} \longrightarrow = 1$$

$$\Rightarrow \log_2 n \le n.$$

$$\Rightarrow \log n \in O(n)$$

} use $c = 1$, $k = 1$

HW. see example 7, p. 211 on why you can use any base b, and not just 2.

HW. P. 211. on an x-y graph try to plot the functions

$f_1(n) = 1$, $f_2(n) = \log n$, $f_3(n) = n$

$f_4(n) = n \log n$, $f_5(n) = n^2$, $f_6(n) = 2^n$

$f_7(n) = n!$ to see how they grow in relation to one another.

can you show:

$$[f_1(n)+f_2(n)] \in O(g_1(n)+g_2(n))\ ?$$

Q: What happens to growth-rate when we <u>combine</u> functions?

run-time of Alg A + Alg B.

Combine means:

a) add two functions ←

b) multiply two functions ←

Loop 1

Loop 2

I. Let $f_1(n) \in O(g_1(n))$

and $f_2(n) \in O(g_2(n))$

$$[f_1(n) + f_2(n)] \in \ ?$$

$$f_1(n) \leq c_1 \cdot g_1(n) \quad \forall\ n \geq k_1$$

$$f_2(n) \leq c_2\, g_2(n) \quad \forall\ n \geq k_2$$

$$\Rightarrow f_1(n) + f_2(n) \leq c_1 g_1(n) + c_2 g_2(n)$$

$$\forall\ n \geq \max(k_1, k_2)$$

$$\leq c_1 \cdot \max(g_1(n), g_2(n))$$

$$+ c_2 \cdot \max(g_1(n), g_2(n))$$

$$\forall\ n \geq \max(k_1, k_2)$$

$$= \underbrace{(c_1 + c_2)}_{c} \max(g_1(n), g_2(n))$$

$$\forall\ n \geq \max\underbrace{(k_1, k_2)}_{k}$$

$$= c \cdot \max(g_1(n), g_2(n))$$

$$\forall\ n \geq k$$

$$\Rightarrow [f_1(n) + f_2(n)] \in O(\max(g_1(n)), g_2(n)))$$

Theorem 1

If $f_1 \in O(g_1)$

and $f_2 \in O(g_2)$

then $f_1 + f_2 \in O(\max(g_1, g_2))$

II. Let $f_1(n) \in O(g_1(n))$
and $f_2(n) \in O(g_2(n))$.

Then $[f_1(n) \cdot f_2(n)] \in$?

simple product

$$f_1(n) \le c_1 g_1(n) \quad \forall n \ge k_1$$

$$f_2(n) \le c_2 g_2(n) \quad \forall n \ge k_2$$

$$\Rightarrow f_1(n) \cdot f_2(n) \le \underbrace{c_1 c_2}_{C} g_1(n) g_2(n)$$

$$\forall n \ge \max(\underbrace{k_1, k_2}_{K})$$

$$\Rightarrow f_1(n) \cdot f_2(n) \le C \cdot g_1(n) \cdot g_2(n)$$

$$\forall n \ge K$$

$$\Rightarrow f_1 f_2 \in O(g_1 g_2)$$

Theorem:

If $f_1(n) \in O(g_1(n))$ and
$f_2(n) \in O(g_2(n))$

Then $f_1(n) \cdot f_2(n) \in O(g_1(n) \cdot g_2(n))$

Examples .

$$f(n) = \underbrace{3n \cdot \log n!}_{f_1(n)} + \underbrace{(n^2+3)\log n}_{f_2(n)} .$$

Question : $f(n) \in O(?)$

$$f_1(n) = \underbrace{3n}_{\uparrow \; O(n)} \cdot \underbrace{\log n!}_{\uparrow \; O(n\log n)}$$

product of two fns.

$$\text{Thm 2.} \Longrightarrow \quad f_1(n) \in O(n^2 \log n)$$

$$\begin{aligned} f_2(n) &= n^2 \log n + 3 \log n && \forall n \geq 1 \\ &\leq n^2 \log n + n^2 \log n && \forall n \geq 1 \\ &= 2n^2 \log n && \forall n \geq 1 \end{aligned}$$

$$\Rightarrow \quad f_2(n) \in O(n^2 \log n)$$

Now $f_1 \in O(n^2 \log n)$, $f_2 \in O(n^2 \log n)$

$$\xRightarrow{\text{Thm 1}} \quad f_1(n) + f_2(n) \in O(n^2 \log n)$$

because $\max(n^2 \log n, n^2 \log n) = n^2 \log n$

Defn. [Big- Omega].

Big-Oh uses " $\leq$ "

Big- Omega uses " $\geq$ "

If $\exists$ $c > 0$ and integer $k > 0$ s.t.

$$f(n) \geq c \cdot g(n) \quad \forall \; n \geq k$$

then $f(n) \in \underbrace{\Omega(g(n))}_{\uparrow}$

set of functions that
are $\Omega(g(n))$
i.e. grow "faster" than $g(n)$

Relationship: $O(\,)$ and $\Omega(\,)$

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

$$\iff \quad g(n) \in \Omega(f(n))$$

(simply apply the definition
and this becomes clear).

Examples:

1. $f(n) = 6n^3 + 5n^2 + 4n$:

Q: $f(n) \in \Omega(?)$

(a) $f(n) \in \Omega(n^3)$?

$$6n^3 + \underbrace{5n^2 + 4n}_{\text{positive}} \geq 6n^3 \quad \forall n \geq 1$$

$$\Rightarrow \quad 6n^3 + 5n^2 + 4n \in \Omega(n^3)$$

and $\quad n^3 \in O(6n^3 + 5n^2 + 4n)$

Note: You have some $f(n)$ by analyzing some algorithm A.

Big-Oh gives you a reference function that acts as an upper bound for $f(n)$

Big-Omega gives a reference function that acts as a lower bound for $f(n)$

So $\boxed{} \le f(n) \le \boxed{}$

f grows faster than this function

f grows slower than this function

This does not help us much if the upper and lower bounds are "far".

Say $f(n) = n^2$

$$n \le f(n) \le 2^n$$

they do not tell us much about what f(n) is really like, or how fast it grows.

These "bounds" are "loose".

How to "narrow" things down?

Idea: A's run-time is $f(n)$

You want to find some function $g(n)$ such that

(a) $f(n) \in O(g(n))$ and

(b) $f(n) \in \Omega(g(n))$

means "f sits in-between g above and g below"

$\Rightarrow$ so f grows just as fast as g.

Defn: [Big-Theta]

Alg. A has a run-time $f(n)$.

Let $g(n)$ be some function.

If $f(n) \in O(g(n))$ and
$f(n) \in \Omega(g(n))$

then $f(n) \in \Theta(n)$

20 (i.e. f and g have same "order", or same growth rate.

1. $S_n = \sum_{i=1}^{n} i = 1+2+\dots+n = \frac{n(n+1)}{2}$.

Q: $S_n \in \Theta(n^2)$?

$$\frac{n(n+1)}{2} = \frac{n^2}{2} + \frac{n}{2} \le \boxed{1} \cdot n^2 \quad \forall n > \underline{1}$$

$$\Rightarrow S_n \in O(n^2)$$

$$\frac{n(n+1)}{2} = \frac{n^2}{2} + \frac{n}{2} \ge \frac{n^2}{2} = \boxed{\frac{1}{2}} n^2 \quad \forall n \ge \underline{1}$$

$$\Rightarrow S_n \in \Omega(n^2)$$

Hence $S_n \in \Theta(n^2)$

<u>Meaning:</u>

$(1+2+3+\dots+n)$ is a sum that grows exactly as fast as n^2.

Remember that $O(\ )$, $\Omega(\ )$ and $\Theta(\ )$ are all DIFFERENT SETS OF FUNCTIONS !

ASYMPTOTIC GROWTH

Given a number of functions, can we rank them in order of increasing growth rate?

Let $f(n)$, $g(n)$ be two functions we compare.

Defn. "$\prec$" (because we can't use "$<$")

$$\underbrace{f(n) \prec g(n)}_{\text{g(n) grows faster than f(n) as } n \uparrow \infty} \iff \lim_{n \to \infty} \frac{f(n)}{g(n)} = 0$$

$\prec$ is transitive. Why?

$$f(n) \prec g(n) \text{ and } g(n) \prec h(n)$$

$$\Rightarrow f(n) \prec h(n)$$

Eg: $n \prec n^2$, $n^2 \prec n^4$

$$\Rightarrow n \prec n^4$$

Here is an ordering: epsilon

$$\boxed{1 \prec \log\log n \prec \log n \prec n^{\varepsilon} \prec n^{c} \prec n^{\log n} \prec c^{n} \prec n^{n} \prec c^{c^{n}}}$$

Examples

$$\sum_{i=1}^{n} i^2 = 1^2 + 2^2 + 3^2 + \cdots + n^2$$

$$\leq n^2 + n^2 + n^2 + \cdots + n^2 = n \cdot n^2$$

$$= n^3$$

$$\Rightarrow \sum_{i=1}^{n} i^2 \in O(n^3)$$

$$\sum_{i=1}^{n} i^2 = 1^2 + 2^2 + 3^2 + \cdots + n^2$$

$$= 1^2 + 2^2 + 3^2 + \cdots + \underbrace{\left(\frac{n}{2}\right)^2 + \cdots + n^2}_{\substack{\text{use only these} \\ \text{last } n/2 \text{ terms}}}$$

$$\geq \underbrace{\left(\frac{n}{2}\right)^2 + \left(\frac{n}{2}\right)^2 + \cdots + \left(\frac{n}{2}\right)^2}_{n/2 \text{ terms}}$$

$$= \frac{n}{2} \cdot \left(\frac{n}{2}\right)^2 = \frac{n^3}{8}$$

$$\Rightarrow \sum_{i=1}^{n} i^2 \in \Omega(n^3)$$

* Observe how we found "tight" bounds.

so clearly $\sum_{i=1}^{n} i^2 \in \Theta(n^3)$ //.

Example.

n! is not so easy.

$$n! = 1 \cdot 2 \cdot 3 \cdots n \le n^n$$

$$\Rightarrow n! \in O(n^n)$$

$$n! = 1 \cdot 2 \cdot 3 \cdots n$$

$$= 1 \cdot 2 \cdot 3 \cdots \underbrace{\frac{n}{2} \cdots n}_{\text{use only these } n/2 \text{ terms}}$$

~~use~~

$$\ge \underbrace{\frac{n}{2} \cdot \frac{n}{2} \cdot \frac{n}{2} \cdots \frac{n}{2}}_{n/2 \text{ times}}$$

$$= \left(\frac{n}{2}\right)^{n/2}$$

$$\Rightarrow n! \in \Omega\left(\left(\frac{n}{2}\right)^{n/2}\right)$$

STIRLING APPROXIMATION

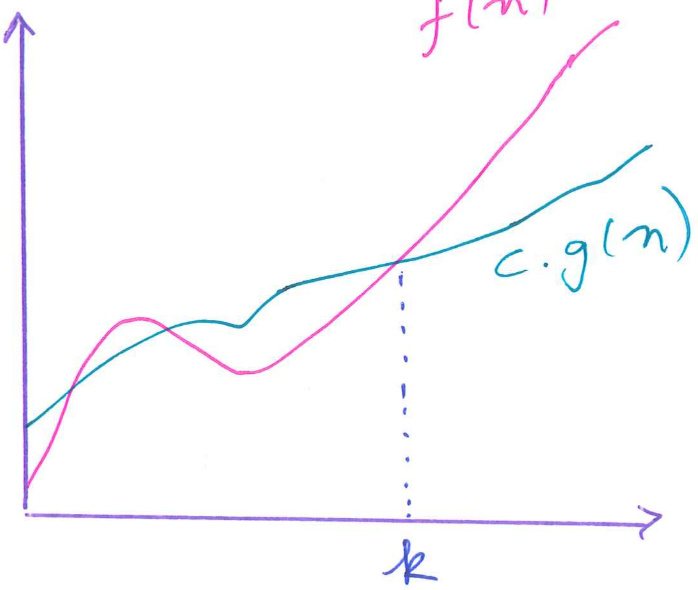
$$n! = \sqrt{2\pi n}\cdot\left(\frac{n}{e}\right)^{n}\left(1+O\left(\frac{1}{n}\right)\right)$$

$$\simeq \sqrt{2\pi n}\cdot\left(\frac{n}{e}\right)^{n} \quad //$$

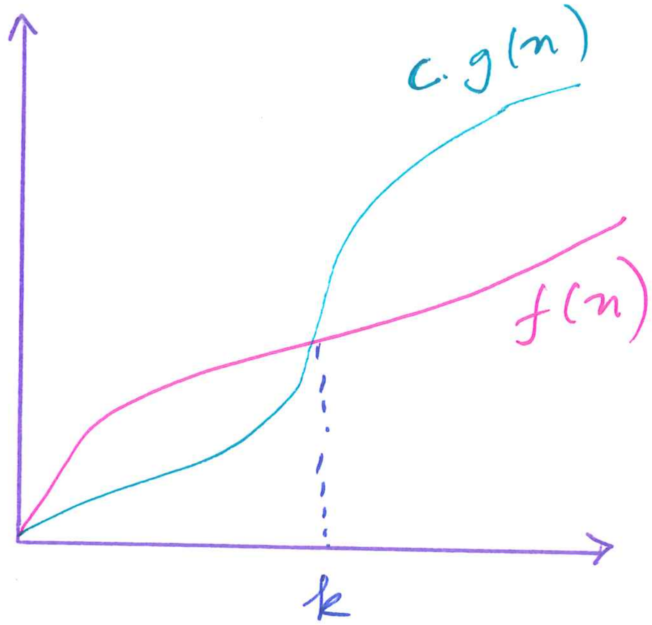
LANDAU SYMBOLS

$$\underset{\checkmark}{O}\;,\quad \underset{\checkmark}{\Omega}\;,\quad \underset{\checkmark}{\Theta}\;,\quad \underset{?}{\overset{\text{little-oh}\ \swarrow}{o}}\;,\quad \underset{?}{\overset{\text{little omega}\ \swarrow}{\omega}}$$

SYMBOL	NAME	USAGE	ANALOGY	
O	Big-oh	upper bound	$f \in O(g)$	$a \leq b$
Ω	Big-Omega	lower bound	$f \in \Omega(g)$	$a \geq b$
Θ	Theta	same order	$f \in \Theta(g)$	$a = b$
o	Little-oh	STRICT upper bound	$f \in o(g)$	$a < b$
ω	little-omega	STRICT lower bound	$f \in \omega(g)$	$a > b$



$f \in \Omega(g)$



$f \in O(g)$

$\exists$ $c > 0$, k posutive integer

s.t. $f(n) \geq c.g(n)$

$\forall$ $n \geq k$.

$\Rightarrow f(n) \in \Omega(g(n))$

$\exists$ $c > 0$, k posctive integer

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

$\forall$ $n \geq k$

$\Rightarrow f(n) \in O(g(n))$

same function

THETA Θ

$\exists$ $c_1 > 0$, $c_2 > 0$ and k positive integer

s.t. $c_1 g(n) \leq f(n) \leq c_2 g(n)$

$\forall$ $n \geq k$

$\Rightarrow f(n) \in \Theta(g(n))$

TIGHT BOUND

EXAMPLE: $3n^2 + 16 \in \Theta(n^2)$. WHY?

$$3n^2 \le 3n^2 + 16 \le 4n^2 \quad \text{for } n^2 \ge 16$$
or $n \ge 4$ //.

HOW to CALCULATE Big-oh relations?

Let $\lim_{n \to \infty} \frac{f(n)}{g(n)} = L$

1. $L = 0 \Rightarrow f(n) \in O(g(n))$

   In fact, $f(n) \in o(g(n))$

2. $L = \infty \Rightarrow f(n) \in \Omega(g(n))$

   In fact, $f(n) \in \omega(g(n))$

3. $0 < L < \infty \Rightarrow f(n) \in \Theta(g(n))$

TROUBLE! Sometimes $\lim_{n \to \infty} \frac{f(n)}{g(n)}$ is hard to calculate.

Say $f(n) = n^2$, $g(n) = 2^n$

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = \lim_{n \to \infty} \frac{n^2}{2^n} = \frac{\infty}{\infty} \; !$$

THEOREM [L'HOPITAL'S RULE]

If $\lim_{n \to \infty} f(n) = \infty$

and $\lim_{n \to \infty} g(n) = \infty$

and $f'(n)$ and $g'(n)$ exist
(i.e., first derivatives exist) then

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = \underbrace{\lim_{n \to \infty} \frac{f'(n)}{g'(n)}}$$

L'Hopital's rule says work with the ratio of first derivatives

Example: $f(n) = n^2, \quad g(n) = 2^n$

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = \lim_{n \to \infty} \frac{n^2}{2^n} = ?$$

L'HOPITAL

How to differentiate 2^n to get this?

$$= \lim_{n \to \infty} \frac{2n}{\boxed{2^n \ln 2}}$$

L'HOPITAL AGAIN!

$$= \lim_{n \to \infty} \frac{2}{2^n \ln 2 . \ln 2} = \lim_{n \to \infty} \frac{2}{2^n (\ln 2)^2}$$

$$= 0 \Rightarrow n^2 \in O(2^n)$$

Let $y = 2^n$

$\log_e$: $\ln y = n \ln 2$

differentiate: $\frac{1}{y} \frac{dy}{dn} = \ln 2$

$$\Rightarrow \frac{dy}{dn} = y \ln 2 = 2^n . \ln 2 .$$

Sometimes we may run into trouble even with L'Hopital's rule.

Example $f(n) = n^n$, $g(n) = 2^n$

clearly $n^n > 2^n$, so $f(n) \in \omega(g(n))$

Try L'Hopital:

$$y = n^n$$

$\log_e$: $\ln y = n \ln n$

$$\frac{1}{y} \cdot \frac{dy}{dn} = n \cdot \frac{1}{n} + (\ln n)\, 1$$

$$\Rightarrow \quad \frac{dy}{dn} = y\left(1 + \ln n\right)$$

$$= n^n (1 + \ln n)$$

So $f'(n) = n^n (1 + \ln n)$

$g'(n) = 2^n \cdot \ln 2$

$$\lim_{n \to \infty} \frac{f'(n)}{g'(n)} = \lim_{n \to \infty} \frac{n^n}{2^n} = \lim_{n \to \infty} \frac{n^n (1 + \ln n)}{2^n \ln 2}$$

.... trouble! $\frac{\infty}{\infty}$

This may help.

THEOREM

If $f(n)$ and $g(n)$ are positive and monotone over $[0, \infty)$, and if

$$\lim_{n \to \infty} f(n) = \infty \quad \text{and} \quad \lim_{n \to \infty} g(n) = \infty,$$

then if

$$\lim_{n \to \infty} \frac{\lg f(n)}{\lg g(n)} = L,$$

$$L < 1 \Rightarrow f(n) \in o(g(n))$$

$$L > 1 \Rightarrow f(n) \in \omega(g(n)) \quad //$$

"lg" means $\log_2$, but log to any base will work.

Example.

$$\lim_{n \to \infty} \frac{n^2}{n^3} \longrightarrow \lim_{n \to \infty} \frac{2 \lg n}{3 \lg n} = \frac{2}{3} < 1$$

$$\Rightarrow n^2 \in o(n^3), \quad n^2 \in O(n^3)$$

<u>Example</u>: Now apply this to the previous example where we got stuck.

$$f(n) = n^n, \quad g(n) = 2^n$$

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{n^n}{2^n}$$

$$\to \lim_{n\to\infty} \frac{\lg n^n}{\lg 2^n}$$

$\lg \equiv \log_2$

$$= \lim_{n\to\infty} \frac{n \lg n}{n \cdot \cancelto{1}{\lg 2}}$$

$$= \lim_{n\to\infty} \frac{\cancel{n} \lg n}{\cancel{n}}$$

$$= \lim_{n\to\infty} \lg n = \infty$$

$\Rightarrow$ $f(n)$ grows much faster than $g(n)$

$\Rightarrow$ $f(n) \in \omega(g(n))$.

//

Example. $f(n) = 2^{\sqrt{2 \lg n}}$ $\qquad g(n) = \lg n!$

[redacted]

$$\lg f(n) = \sqrt{2 \lg n} \cdot \cancelto{1}{\lg_2 2}$$

$$= \sqrt{2 \lg n}$$

$$\lg g(n) = \lg(\underbrace{\lg n!}) = \lg \lg n!$$

we know $\lg n! \in \Theta(n \lg n)$

So, $\lg \lg n! \in \Theta(\lg \underbrace{(n \lg n)})$

$$\lg(n . \lg n) = \lg n + \underbrace{\lg(\lg n)}_{\text{smaller than } \lg(n)}$$

$$\Rightarrow \lg(n \lg n) \in \Theta(\lg n)$$

Hence, $\lim_{n \to \infty} \frac{\lg f(n)}{\lg g(n)} = \lim_{n \to \infty} \frac{\sqrt{2} . \sqrt{\lg n}}{\lg n} = \lim_{n \to \infty} \frac{\sqrt{2}}{\sqrt{\lg n}}$

$$\lim_{n \to \infty} \frac{\lg f(n)}{\lg g(n)} = \lim_{n \to \infty} \frac{\sqrt{2}\sqrt{\lg n}}{\lg n}$$

$$= \lim_{n \to \infty} \frac{\sqrt{2}}{\sqrt{\lg n}} = 0$$

$\Rightarrow$ $2^{\sqrt{2\lg n}}$ grows much slower than $\lg n!$

$\Rightarrow$ $2^{\sqrt{2 \lg n}} \in o(\lg n!)$

Note: When $L = 1$ in the last method using log, the method fails. When $L = 1$ we cannot tell which of $f(n)$ or $g(n)$ grows faster. We'll need another way to tell.