Computational Thinking and Algorithms

159.171

Searching and Sorting Algorithms

Amjed Tahir

a.tahir@massey.ac.nz

Previous contributors: Catherine McCartin & Giovanni Moretti

Reading from a file

useful to know how to read data from a file
allows a program to search large sample data sets easily

```
# Read in data from a file and put it into a Python List

file = open("example_sorted_names.txt")

name_list = []
for line in file:
    line=line.strip()
    name_list.append(line)

file.close()
```

Reading from a file

```
# Read in data from a file and put it into a Python List
file = open("example_sorted_names.txt")

name_list = []
for line in file:
    line=line.strip()
    name_list.append(line)

file.close()
```

Verify file read correctly

```
print( "There were " + str(len(name_list)) +" names in the file.")

for line in name_list:
    print(line)
```

Linear or Sequential Search

Starts at the first element, and keeps comparing elements until it finds the desired element or runs out of elements to check

```
# Linear search

i = 0
while i < len(name_list) and name_list[i] != "Morgiana the Shrew":
    i += 1

if i == len(name_list):
    print( "The name was not in the list." )
else:
    print( "The name is at position", i)
```

Binary Search

Consider the number guessing game. (guessing the number 92!)

```
Guess a number 1 to 128: 64
Too low.
Guess a number 1 to 128: 96
Too high.
Guess a number 1 to 128: 80
Too low.
Guess a number 1 to 128: 88
Too low.
Guess a number 1 to 128: 92
Correct!
```

After each reply, the guesser is able to eliminate one half of the problem space by getting a “high” or “low” as a result of the guess.

Binary Search

Binary search only works if the items in the list are in order.

In a binary search, it is necessary to track an upper and a lower bound of the list that the answer can be in. The computer or number-guessing human picks the midpoint of those elements. Revisiting the example:

lower bound = 1, upper bound = 128, mid point = (1+128)/2=64.5

Guess a number 1 to 128: 64

Too low.

Binary Search

lower bound = 65, upper bound = 128, mid point = (65+128)/2=96.5

Guess a number 1 to 128: 96

Too high.

Binary Search

lower bound = 65, upper bound = 95, mid point = (65+95)/2=80

```
Guess a number 1 to 128: 80
Too low.
```

Binary Search

lower bound = 81, upper bound = 95, mid point = (81+95)/2=88

```
Guess a number 1 to 128: 88
Too low.
```

Binary Search

lower bound = 89, upper bound = 95, mid point = (89+95)/2=92

Guess a number 1 to 128: 92

Correct.

Binary Search

Requires significantly fewer guesses than a linear search.

Allow an extra guess = double the range of numbers that can be handled.
For a range of size ***n***, guesser will need **log n** guesses in the worst case.

log n means "log to the base 2 of n"

if n is a power of 2, **log n** is "the number of times I can cut n in half before I get to 1".

- if n is not a power of 2, then **log n** lies between two integer values
- take the higher of those values as the worst case number of guesses

If n = 47 then n lies between $32 = 2^5$ and $64 = 2^6$, so I might need 6 guesses.

Binary Search - *list must be sorted*

```
target = ”Alex"
lower_bound = 0
upper_bound = len(name_list)-1
found = False

while lower_bound <= upper_bound and found == False:
    middle_pos = (lower_bound + upper_bound) // 2  # // is an integer division
# 4 / 2 = 1.333  -> 4//3 = 1
    if name_list[middle_pos] < target:
        lower_bound = middle_pos+1
    elif name_list[middle_pos] > target:
        upper_bound = middle_pos-1
    else:
        found = True

if found:
    print( "The name is at position", middle_pos)
else:
    print( "The name was not in the list." )
```

Binary Search – Recursive version

```
# Recursive Binary search, list must be sorted

def binary_search(the_list, lower, upper, item):

    if lower > upper:
       print( "The name was not in the list." )
       return

    middle_pos = (lower + upper) // 2

    if the_list[middle_pos] < item:
       lower = middle_pos+1
       binary_search(the_list, lower, upper, item)
    elif the_list[middle_pos] > item:
       upper = middle_pos - 1
       binary_search(the_list, lower, upper, item)
    else:
       print( "The name is at position", middle_pos)
       return
```

Binary Search – recursive version

```
# Binary search, list must be sorted

def binary_search(the_list, lower, upper, item):

    if lower > upper:
       print( "The name was not in the list." )
       return

    middle_pos = (lower + upper) // 2

    if the_list[middle_pos] < item:
       lower = middle_pos+1
       binary_search(the_list, lower, upper, item)
    elif the_list[middle_pos] > item:
       upper = middle_pos - 1
       binary_search(the_list, lower, upper, item)
    else:
       print( "The name is at position", middle_pos)
       return
```

```
>>> binary_search(name_list, 0, len(name_list)-1, "Morgiana the Shrew")
```

Sorting

Sorting algorithm puts elements of a list into a certain order.

Output needs to satisfy two conditions:

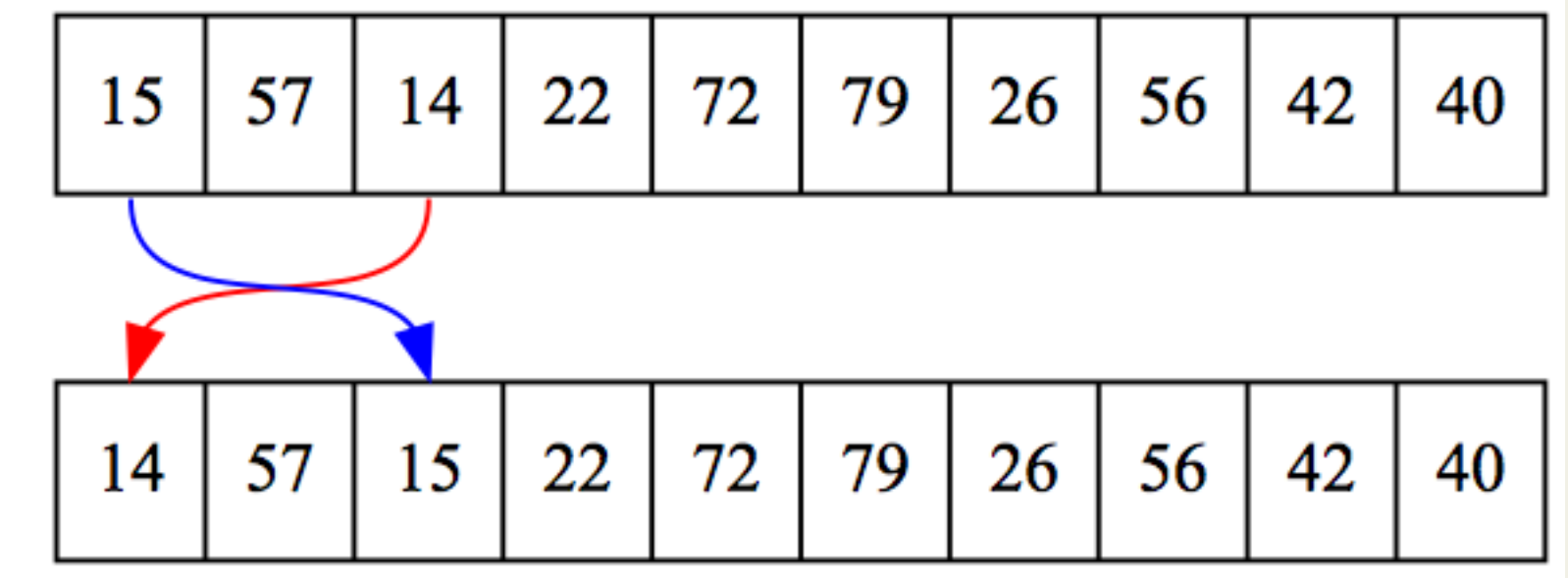
1. **Output is in non-decreasing order**

   each element is no smaller than the previous element according to the desired ordering

2. **Output is a permutation, or reordering, of the input**
   - we must not add or lose any elements
   - if there are duplicates in the input, they remain after sorting

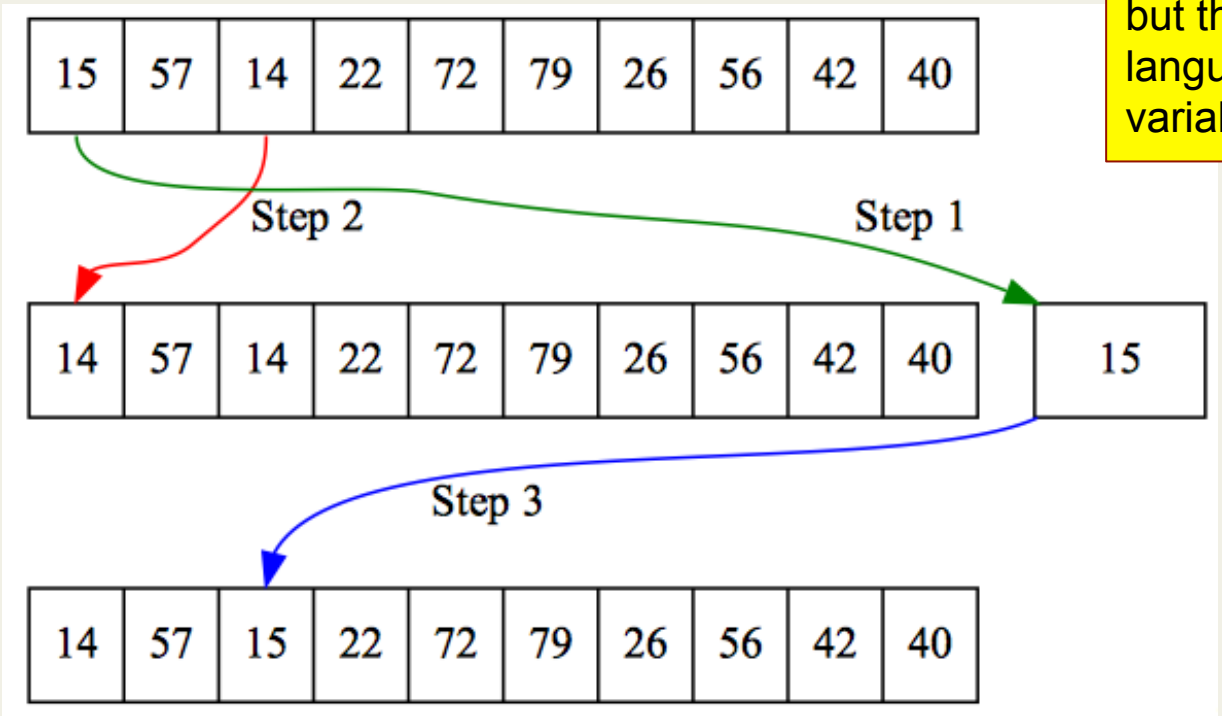
Swapping values

Want to swap positions 0 and 2



Swapping values

```
temp = list[0]
list[0] = list[2]
list[2] = temp
```



For swapping values, Python allows

list[0], list[[2] = list[2], list[0]

but this doesn't work in most languages, so a temporary variable (temp) is used.

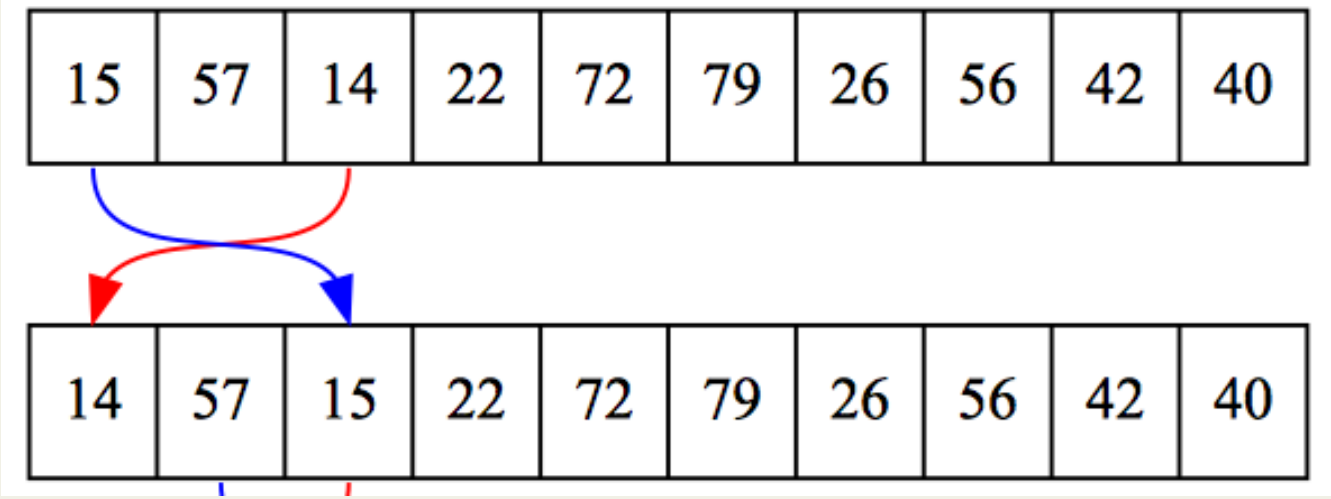
Selection sort

***n* sort passes**

first "pass" through the list = scan the list to find the smallest element

- then smallest element is swapped into the first location

(next "pass" will start from the second element)

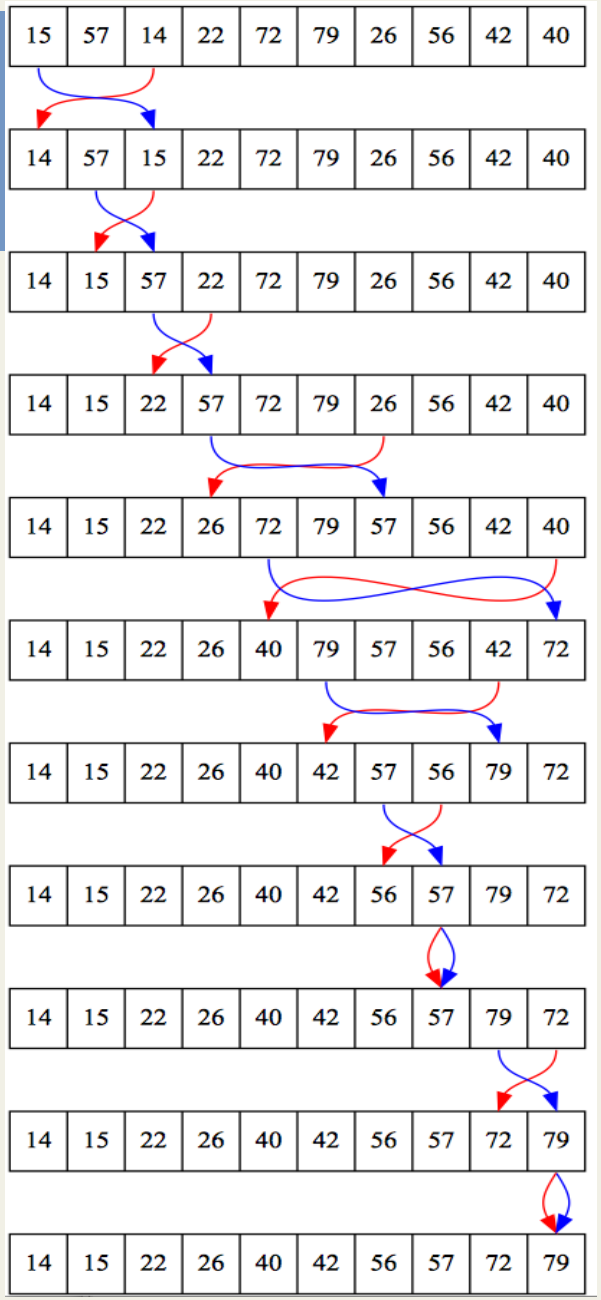


Selection sort

After **n-1** passes through the list,

smallest n-1 elements are in the first n-1 locations,

largest element is in the last location.



Selection sort

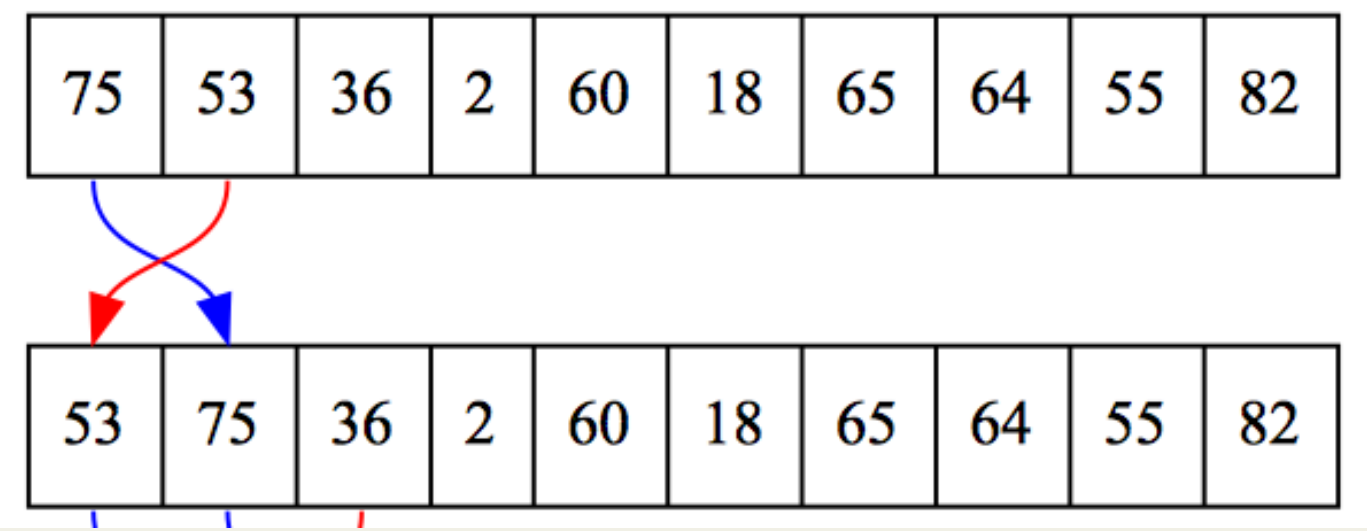
```
def selection_sort(list):

    for i in range( len(list) ):        # Loop through the entire array

        # Find the position that has the smallest number
        # Start with the current position
        smallestPos= i

        for j in range(i+1, len(list) ):     # Scan what's left

            if list[j] < list[smallestPos]:    # Is this position smallest?
                smallestPos= j             # Smallest so far, save position
        # Swap the two values
        temp = list[smallestPos]
        list[smallestPos] = list[i]
        list[i] = temp
```

Insertion sort

n sort passes

each "pass" through the list = pick up element to right of sorted portion
insert this element in correct position of sorted portion

first "pass" starts at the second element.



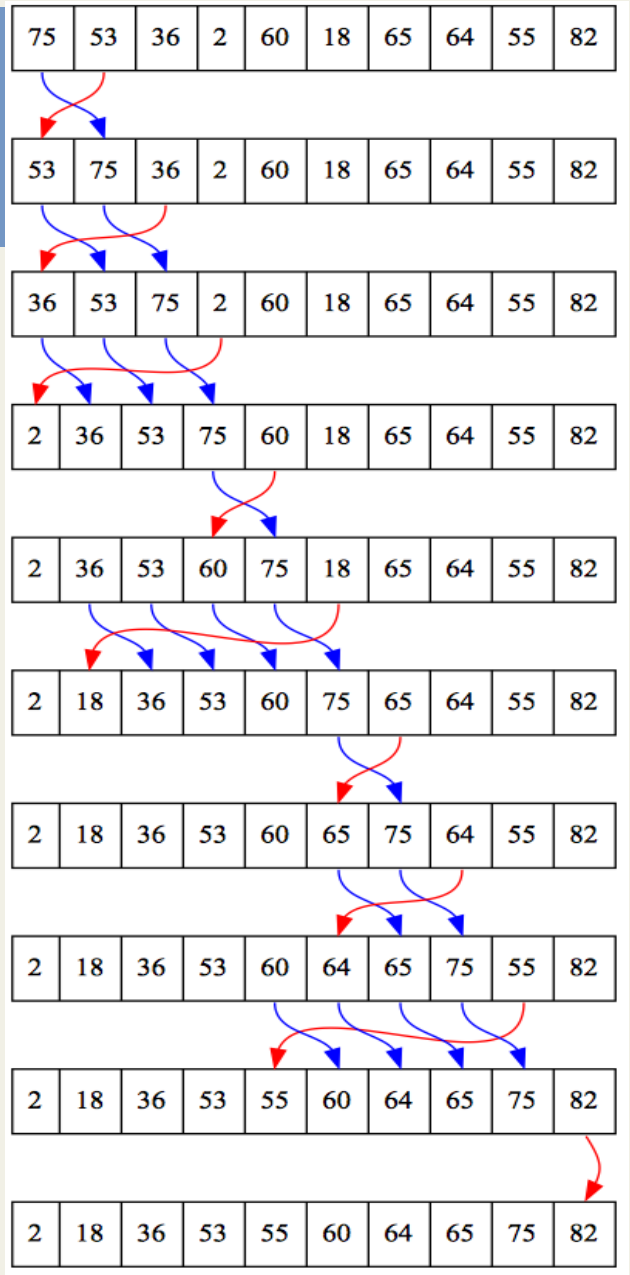
Insertion sort

Breaks the list into two sections,
the "sorted" half and the "unsorted" half

On each pass through the list,
algorithm will grab next unsorted element,
insert it into sorted portion

Sorted portion starts off as singleton `the_list`

After n-1 passes through the list,
all n items in correct position



Insertion sort

```
def insertion_sort(list):

    # Start at the second element (pos 1).
    # Insert this element into the sorted part of the list.
    for i in range(1, len(list)):

        keyValue = list[i]     # Get the value of the element to insert

        j = i - 1              # Scan to the left

        # Loop each element, moving them up until
        # we reach the position the element should go in
        while (j >= 0) and (list[j] > keyValue):
            list[j+1] = list[j]
            j = j - 1

        # Everything's been moved out of the way, insert
        # the key into the correct location
        list[j+1] = keyValue
```

Efficiency

How long does selection sort take to run?
How much storage space does selection sort use?

Measure amount of time/space required as a function of size of input

T(n) = max number of steps taken by the algorithm on any input of size n
$\mathbf{T_{avg}(n)}$ = average number of steps taken over all inputs of size n

S(n) = storage space required by the algorithm on any input of size n
(don't count the space used by the input data)

What should we measure to determine the size of the input?

Efficiency

What do we mean by a step?

Time efficiency measures the amount of work required by the nature of the algorithm itself and the approach that it uses,

not the exact number of machine instructions required to run the algorithm using a particular implementation on a particular machine.

We need to identify the **fundamental units of work** done by an algorithm.

for sorting algorithms count…

(number of comparisons) + (number of moves)

Selection Sort

Number of comparisons and moves is independent of the arrangement of the data being sorted.

Requires **n(n-1)/2 compares** no matter how the data are initially ordered.

Requires **3(n-1) moves** (one swap in each iteration of outer loop) no matter how the data are initially ordered.

The outside loop will **always run *n* times**. The inside loop will run **(n-(i+1))** times, for each *i* from 0 to *n*-1. This will be the case regardless if the list is in order or not.

Good method for:

large items, expensive to move

short key fields, easy to compare.

Insertion sort

Worst case: Requires **n(n-1)/2 compares** and **n(n-1)/2 moves**

On average, requires only half as many compares and moves.

Best case: n compares and **n** moves

On average, this means we need to look at **half** of the sorted list on each pass.

The inside loop will run an average of **i/2 times**, for ***i*** in the range 1 to ***n-1***, if the loop is randomly shuffled.

If the loop is close to a sorted loop already, then the inside loop does not run very much, and **the total sort time is closer to n.**

Efficiency

Large problems

- running time determines whether a given program can be used
- need a measure of running time that summarizes the program's performance on all inputs

We want to hide constant factors such as:

- average number of machine instructions a particular compiler generates
- average number of instructions a particular machine executes per second

Simplify:

instead of saying that *Selection sort does n(n-1)/2 comparisons and 3(n-1) moves*, say that ***Selection sort takes $O(n^2)$ time*.**

Order n^2 or **Big-Oh of n^2**
Big-O means worst case scenario!
informally, this means ***some constant times n^2***

Order notation

Two important principles:

- constant factors don't matter

- low-order terms don't matter

A program's execution time is **O(f(n))** if

T(n) – the execution time for inputs of size *n*

is at most **c** times **f(n)**, for some constant **c**, except maybe for some small values of n

f(n) is usually **1**, **n**, **n log(n)**, $\mathbf{n^2}$, $\mathbf{n^3}$, $\mathbf{2^n}$ …

Choose the tightest bound
e.g. for all **n** >= 1:
$\mathbf{n^2 <= n^3}$
so choose $\mathbf{n^2}$

Order notation

T(n) is O(f(n)

if there is an integer **m** and a constant **c** > 0 such that

for all **n** > **m** we have **T(n) <= c . f(n)**

Q: Running time is **$3n^2$** - this is still **$O(n^2)$?**

yes c is 3, for all n

Q: Running time is **$(n+1)^2$**

- is this still **$O(n^2)$**,

yes, as n => 1 (so m = 1) gives **$(n+1)^2$** <= **$4n^2$**

We need to find a constant **c** so that
$(n+1)^2 < cn^2$
for all n sufficiently large

Largest **c** is 4, when n is 1

Order notation

Use a Big-Oh expression as an upper bound on T(n)
want the tightest upper bound that we can show

One way to express that f(n) is a tight bound,

T(n) is **O(f(n))** and **T(n)** is **Ω(f(n))**
except for small values of **n,**
we have **T(n) < c . f(n)** and **f(n) < c . T(n)**

say T(n) is **Θ(f(n))** 'theta f(n)' in this case

Knowledge of Ω(f(n)) is optional & will NOT be examined

Big Oh running times

BIG-OH	INFORMAL NAME
$O(1)$	constant
$O(\log n)$	logarithmic
$O(n)$	linear
$O(n \log n)$	n log n
$O(n^2)$	quadratic
$O(n^3)$	cubic
$O(2^n)$	exponential

Fig. 3.4. Informal names for some common big-oh running times.

O(1) examples

O(1) describes an algorithm that will always execute in the same time (or space) regardless of the size of the input.

```
def getfirst(a_list):
    return a_list[0]== null

def getlast(a_list):
    return a_list[len(list)-1]
```

O(1) examples

O(1) describes an algorithm that will always execute in the same time (or space) regardless of the size of the input.

Remove method from Improved linked Queue class:

```
def remove(self):
    cargo     = self.head.cargo
    self.head = self.head.next
    self.length = self.length - 1
    # if list becomes empty, last must be set to None
    if self.length == 0:
      self.last = None
    return cargo
```

O(1) examples

O(1) describes an algorithm that will always execute in the same time (or space) regardless of the size of the input.

methods from Linked List class:

```
def addFirst(self, cargo):
    node = Node(cargo)
    node.next = self.head
    self.head = node
    self.length = self.length + 1

def removeFirst(self):
    cargo = self.head.cargo
    self.head = self.head.next
    self.length = self.length - 1
    return cargo
```

O(n) examples

O(n) describes an algorithm whose performance will **grow linearly** and in direct proportion to the size of the input.

```
# Linear search
def linear_search(a_list, item):
    i = 0
    while i < len(a_list) and a_list[i] !=item:
    i += 1

    if i == len(a_list):
       print( str(item) + " was not in the list." )
    else:
       print( str(item) + " is at position " + str(i))
```

O(n) examples

O(n) describes an algorithm whose performance will grow linearly and in direct proportion to the size of the input.

```
>>> list((1, 2, 3))
[1, 2, 3]

>>> list('john')
['j', 'o', 'h', 'n']

>>> newlst
['my ', 'big ', 'greek ', 'wedding ', 'next ', 'week ']

>>> newlst.insert(2, 'fat ')
>>> newlst
['my ', 'big ', 'fat ', 'greek ', 'wedding ', 'next ', 'week ']
```

O(n) examples

O(n) describes an algorithm whose performance will grow linearly and in direct proportion to the size of the input.

```
def insertLast(self, cargo):
    node = Node(cargo)
    node.next = None
    if self.head == None:
        # if list is empty the new node goes first
        self.head = node
    else:
        # find the last node in the list
        last = self.head
          while last.next is not None:
                last = last.next
        # append the new node
        last.next = node
    self.length = self.length + 1
```

$O(n^2)$ examples

$O(n^2)$ represents an algorithm whose performance is directly proportional to the **square of the size of the input**.
This is common with algorithms that involve nested iterations

```
def insertion_sort(list):

    for i in range(1, len(list)):
        keyValue = list[i]
        j = i - 1

        while (j >= 0) and (list[j] > keyValue):
            list[j+1] = list[j]
            j = j - 1

        list[j+1] = keyValue
```

$O(n^2)$ examples

$O(n^2)$ represents an algorithm whose performance is directly proportional to the square of the size of the input.

```
def selection_sort(list):

    for i in range( len(list) ):
        minPos = i

        for j in range(i+1, len(list) ):
            if list[j] < list[minPos]:
               minPos = j

        temp = list[minPos]
        list[minPos] = list[i]
        list[i] = temp
```

O(log n) example

O(log n) represents an algorithm whose performance is **directly proportional to log (base 2) of the size of the input**.

```
def binary_search(the_list, lower, upper, item):

    if lower > upper:
       print( "The item was not in the list." )
       return
    middle_pos = (lower + upper) // 2

    if the_list[middle_pos] < item:
       lower = middle_pos+1
       binary_search(the_list, lower, upper, item)
    elif the_list[middle_pos] > item:
       upper = middle_pos - 1
       binary_search(the_list, lower, upper, item)
    else:
       print( "The item is at position", middle_pos)
       return
```

Comparing running times

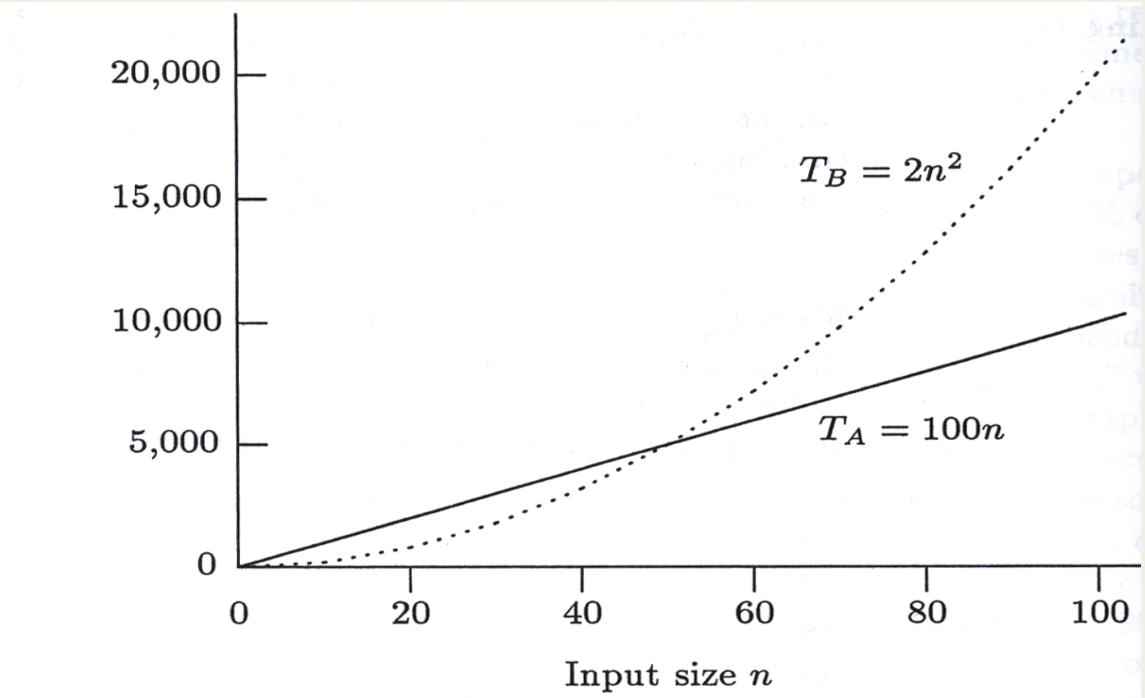
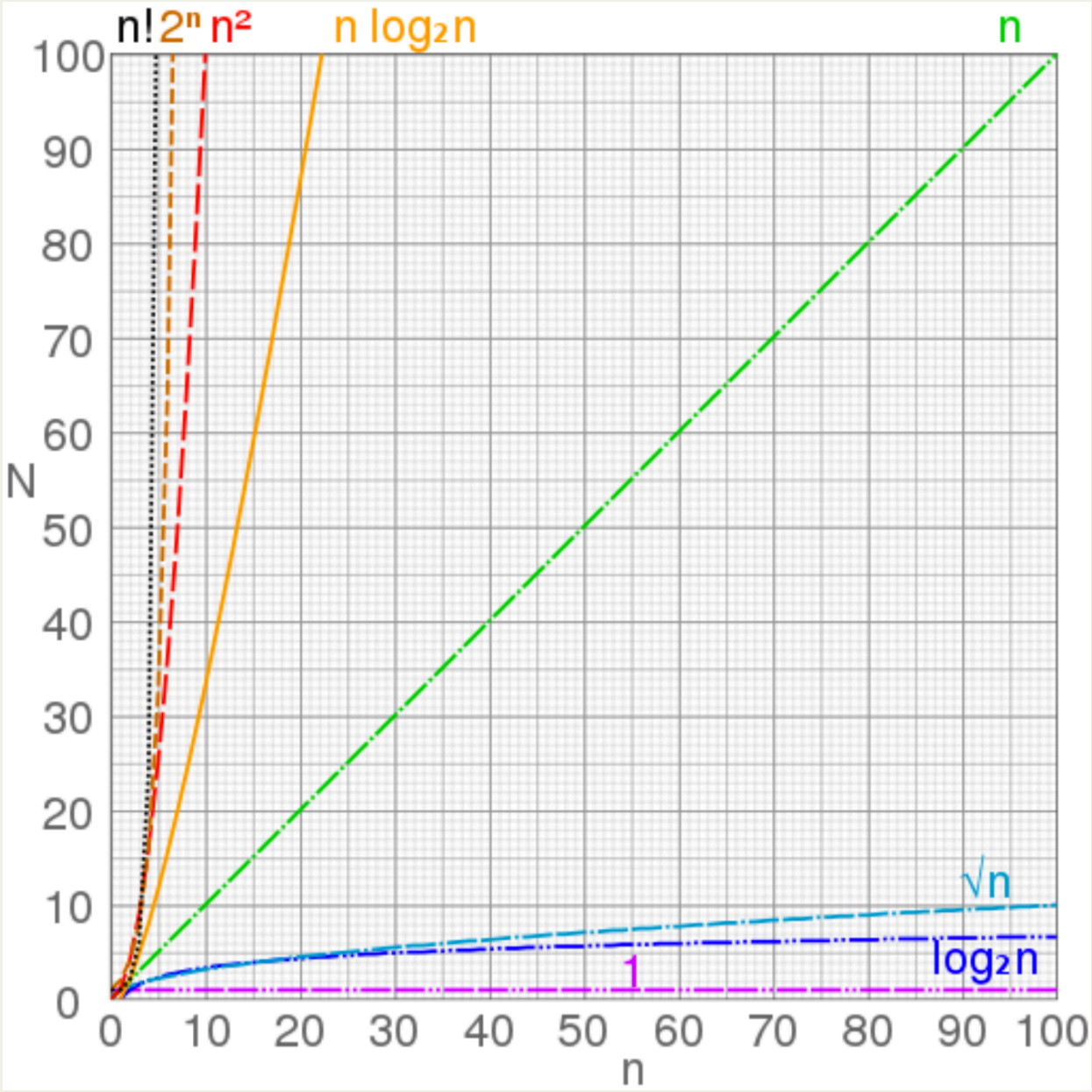


Fig. 3.2. Running times of a linear and a quadratic program.

The **functional form** of a program's running time determines how **big a problem** input we can solve with that program



No of operations (N) for various f(n) - slow growth is better

from Creator cmglee - https://en.wikipedia.org/wiki/Big_O_notation