Simple Complexity and Algorithms

SEARCHING, SORTING

Measuring Program Efficiency

Computers are fast and getting faster, so why is program efficiency important?

- Data sets can be very large, think about the data searched and retrieved by Google, so simple solutions will not scale for such large data sets.

There can be many algorithms for one problem, how do we decide which is the most efficient?

Separate the time and space efficiency of a program.

There is a trade-off between the time a program takes, and the memory required for a program:

- Sometimes we can pre-compute and store results to lookup more quickly

We will focus on time efficiency.

Measuring Program Efficiency

For example, we want to answer the question, how long will the following function take to run?

```
def f(i):
    """Assumes i is an int and i >= 0"""
    answer = 1
    while i >= 1:
        answer *= i
        i -= 1
    return answer
```

We could run with some input and time it, however the results would be inconsistent, because they would depend on the speed of the computer, the efficiency of the Python implementation and the value of the input

Timing programs is inconsistent…

Goal: to evaluate the efficiency of different algorithms:

- Running time varies between algorithms
- Running time varies between implementations.
- Running time varies between computers.
- Running time is not predictable based on small inputs.

Time varies for different inputs but we cannot express a relationship between inputs and time.

Measuring Program Efficiency

A better way to measure the time in terms of the number of basic steps executed by the program.

A **step** is an operation that takes a fixed amount of time, such as:

- Assignment (binding a variable to an object).
- Making a comparison.
- Executing an arithmetic operation.
- Accessing an object in memory.

COUNTING OPERATIONS

- assume these steps take **constant time**:
  - mathematical operations
  - comparisons
  - assignments
  - accessing objects in memory
- then count the number of operations executed as function of size of input

```
def c_to_f(c):
    return c*9.0/5 + 32

def mysum(x):
    total = 0
    for i in range(x):
        total += i
    return total
```

3 ops

1 op

loop x times

1 op

2 ops

1 op

mysum → **2** +3x ops

Measuring Program Efficiency

Different inputs change the efficiency of a program.

When evaluating the complexity of a program, there are 3 broad cases to consider:

- **Best case:** running time of the algorithm when the inputs are as favorable.
- **Worst case:** running time is the maximum running time over all the possible inputs of a given size.
- **Average case (expected case):** running time is the average running time over all possible inputs of a given size.

Often programmers consider the worst case because it provides an upper bound of the running time.

Program Complexity

Let's look at the worst-case running time of the factorial program:

```
def fact(n):
    """Assumes n is an int and n >= 0"""
    answer = 1
    while n >= 1:
        answer *= i
        n -= 1
    return answer
```

The number of steps is 2 + 5n

- 2: 1 for the initial assignment and 1 for the return.
- 5n:
  - 1 for the test in the while,
  - 2 for the first statement in the loop (assignment and multiplication)
  - 2 for the second statement in the loop (assignment and subtraction)

Program Complexity

In the factorial example, how many steps are there if n is equal to 1000?

The answer is 5002 steps, as we can see there is no difference between 5000 and 5002 steps, as n gets larger the difference between 5n and 5n+2 is unimportant.

Therefore, we generally can ignore additive constants, as they do not impact the result.

Big ‘O’ Notation (Asymptotic Notation)

Big O notation is a way to formally describe the relationship between the running time of an algorithm and the size of its inputs.

Used to give the upper bound of the growth of a function.

Describes the complexity of an algorithm as the size of its inputs approaches infinity.

Asymptotic Notation - 09_asymptoticNotation.py

```
def f(x):
    """Assume x is an int > 0"""
    ans = 0
    #Loop that takes constant time
    for i in range(1000):
        ans += 1
    print('Number of additions so far', ans)
    #Loop that takes time x
    for i in range(x):
        ans += 1
    print( 'Number of additions so far', ans )
    #Nested loops take time x**2
    for i in range(x):
        for j in range(x):
            ans += 1
            ans += 1
    print('Number of additions so far', ans )
    return ans
```

Big ‘O’ Notation and Complexity

Running time: $1000 + x + 2x^2$

```
f(10)

Number of additions so far 1000

Number of additions so far 1010

Number of additions so far 1210


f(1000)

Number of additions so far 1000

Number of additions so far 2000

Number of additions so far 2002000
```

The executions above show that for small values of x, the constant term dominates, when x is large, the last term (nested loop) that dominates.

Rules for calculating asymptotic complexity

Keep the term with the largest growth rate in the sum of multiple terms and drop the others.

If the remaining term is a product, drop any constants.

According to this rule, the running time of f(), is $O(x^2)$, because as shown in the sample runs, it has the largest growth rate as the inputs increase.

ORDERS OF GROWTH

Goals:

- want to evaluate program's efficiency when **input is very big**
- want to express the **growth of program's run time** as input size grows
- want to put an **upper bound** on growth – as tight as possible
- do not need to be precise: **"order of" not "exact"** growth
- we will look at **largest factors** in run time (which section of the program will take the longest to run?)
- **thus, generally we want tight upper bound on growth, as function of size of input, in worst case**

SIMPLIFICATION EXAMPLES

- drop constants and multiplicative factors
- focus on **dominant terms**

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

$O(n^2)$: $n^2 + 100000n + 3^{1000}$

$O(n)$: $\log(n) + n + 4$

$O(n \log n)$: $0.0001*n*\log(n) + 300n$

$O(3^n)$: $2n^{30} + 3^n$

Most Common Big O Measures

Assuming n is the number of inputs, the following measures are used:

$O(1)$: constant running time.

$O(\log n)$: logarithmic running time, reduce problem in half each time through processes.

$O(n)$: linear running time, simple iterative or recursive programs

$O(n \log n)$: log-linear running time

$O(n^k)$: polynomial running time (k is a constant)

$O(c^n)$: exponential running time.

Constant Complexity O(1)

Asymptotic complexity is independent of the size of the inputs.

Example: finding the length of a python list, multiplying two floating point numbers, displaying a message, accessing an element in an array.

Constant complexity does not mean there are no loops, no recursive calls, but that the number of iterations/recursive calls are independent of the size of the input.

Logarithmic Complexity O(log(n))

An example is the binary search algorithm which we will talk about in detail.

```
def intToStr(i):
    digits = '0123456789'
    if i == 0:
        return '0'
    res = ''
    while i > 0:
        res = digits[i%10] + res
        i = i//10
    return res
```

only have to look at loop as no function calls

within while loop, constant number of steps

how many times through loop?

- how many times can one divide i by 10?
- *O(log(i))*

Linear Complexity O(n)

Access each element in a sequence a constant number of times greater than zero.

For example, searching a list for an element, or adding characters of a string assumed to be composed of decimal digits.

```
def addDigits(s):
    val = 0
    for c in s:
        val += int(c)
    return val
```

Recursive Factorial O(n)

Same as linear complexity.

There are no loops, but the number of recursive calls determine the complexity level.

Although the memory usage is greater, here the complexity reflects the timing.

Log-Linear Complexity O(n log(n))

Both terms depend on the size of the inputs in the product of two terms.

Example, mergesort, which we will look at in detail.

Polynomial Complexity $O(n^k)$

The complexity grows as the square of the size of their inputs.

The most common polynomial algorithms are quadratic $O(n^2)$

The following algorithm determines if one list is a subset of the second.

```
def isSubset(L1, L2):
    for e1 in L1:
        matched = False
        for e2 in L2:
            if e1 == e2:
                matched = True
                break
        if not matched:
            return False
    return True
```

outer loop executed len(L1) times

each iteration will execute inner loop up to len(L2) times, with constant number of operations

*O(len(L1)*len(L2))*

worst case when L1 and L2 same length, none of elements of L1 in L2

$O(len(L1)^2)$

Polynomial Complexity $O(n^k)$

Find the intersection of two lists.

```
def intersect(L1, L2):
    tmp = []
    for e1 in L1:
        for e2 in L2:
            if e1 == e2:
                tmp.append(e1)
    res = []
    for e in tmp:
        if not(e in res):
            res.append(e)
    return res
```

first nested loop takes *len(L1)*len(L2)* steps

second loop takes at most *len(L1)* steps

determining if element in list might take *len(L1)* steps

if we assume lists are of roughly same length, then

$O(len(L1)^2)$

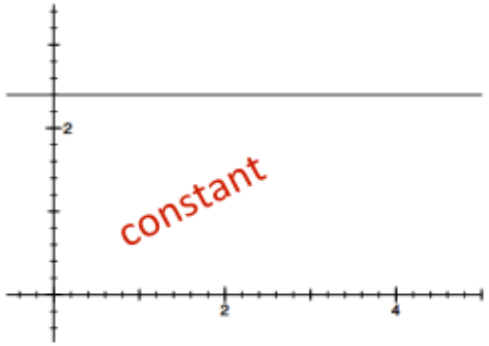
Exponential Complexity $O(c^n)$

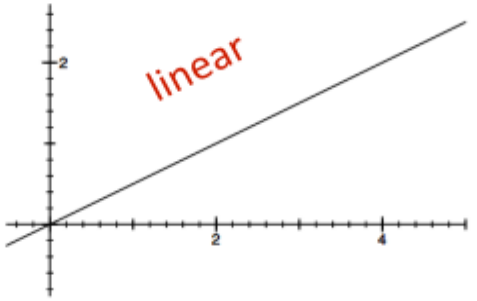
Many important problems are exponential.

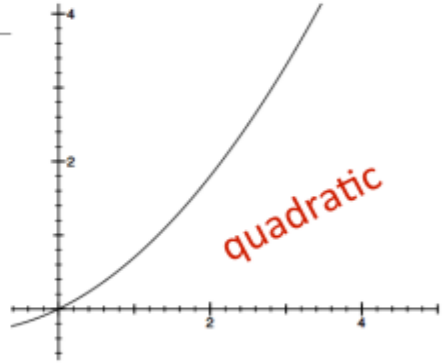
Because the complexity grows as an exponent of the size of the problem, exponentially complex programs may not be useful.

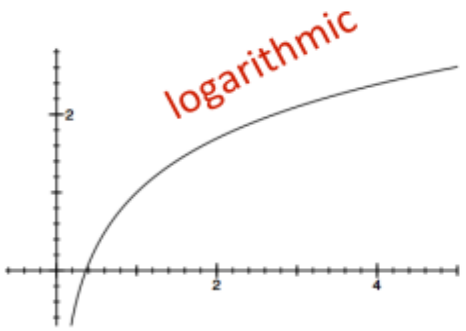
Less complex algorithms may be found instead.

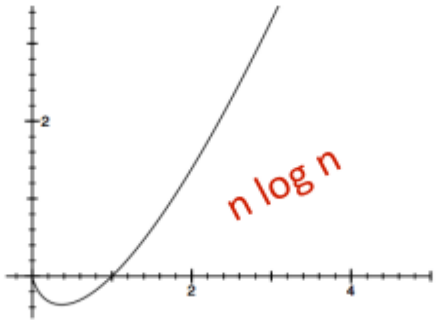
TYPES OF ORDERS OF GROWTH

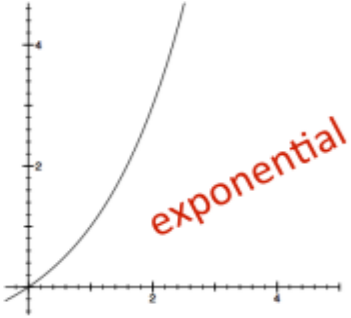












Simple Algorithms

Most common types of algorithms used in programs: searching and sorting algorithms.

Python has many built in functions that allow us to quickly and easily search and sort lists of data.

Understanding these algorithms help to illustrate program complexity and algorithmic efficiency.

Algorithms are language independent and may be implemented in any programming language.

Search Algorithms

A search algorithm is a method for finding an item or group of items with specific properties within a collection of items.

The collection could be implicit:

- example – find square root as a search problem
- exhaustive enumeration

The collection could be explicit:

- example – is a student record in a stored collection of data?

Search Algorithms

There are two general search algorithms, linear search and binary / bisection search.

Linear search:

- Called a brute force search.
- Starts from the beginning or the end and searches until target value is found or until there is no more data to be searched (value is not found).
- The list does not need to be sorted.

Binary / bisection search:

- The list must be sorted to find the correct result.
- Divide and conquer where each time you discard half of the values in the search data.

Linear Search Algorithm – Unsorted List

```
def linear_search(L, e):
    found = False
    for i in range(len(L)):
        if e == L[i]:
            found = True
    return found
```

Access list elements in constant time.

Must look through all elements to decide if the value is not there.

You can stop searching/return when the value is found – only small impact on efficiency.

Best/worst case?

09_LinearSearch.py

Linear Search Algorithm – Sorted List

```
def search(L, e):
    for i in range(len(L)):
        if L[i] == e:
            return True
        if L[i] > e:
            return False
    return False
```

If the list that we are searching is sorted, we can make our algorithm more efficient.

In this case we only need to search the list until the current element of the list is greater than the value that we are searching for.

Linear Search Complexity

Best Case for Linear Search?

- Element is the first element in the list (1) and is not dependent on the list size.

Average Case for Linear Search?

- Half the time it will be in the first half of the list, half the time it will be in the second half of the list, therefore the average case is n/2.

Worst Case for Linear Search?

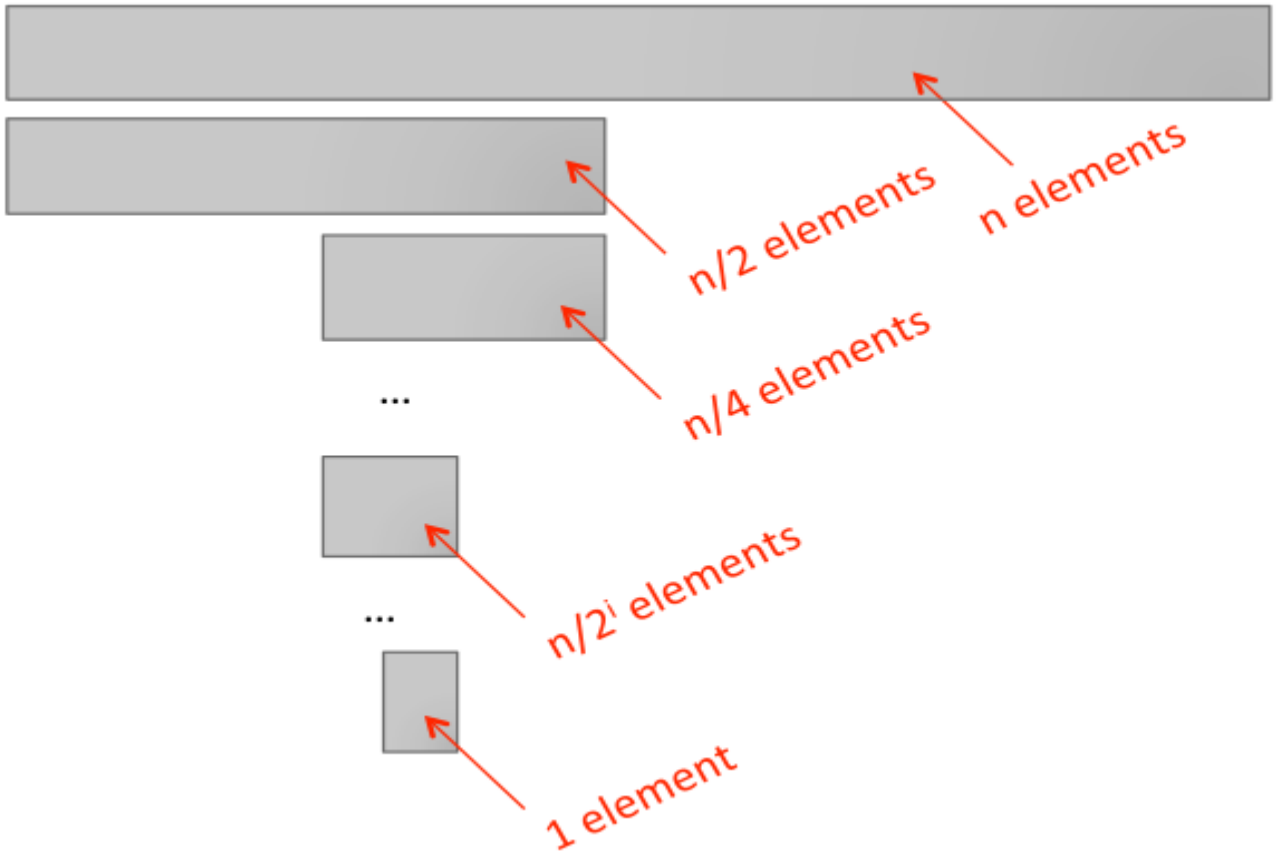
- If the value that we are searching for is not in the list, n elements will need to be accessed. Therefore the running time is linear (n) and dependent on the size of the list.

Efficiency of Linear Search

In the worst case—which occurs when the value you're searching for comes at the end of the array or does not appear at all—linear search requires *N* steps. On average, it takes approximately half that.

O (N)

Binary Search Algorithm



- finish looking through list when

  $1 = n/2^i$

  so $i = \log n$

- complexity of recursion is **O(log n) – where n is len(L)**

Binary (Bisection) Search

```
def binary_search(L, e):
    first = 0
    last = len(L)-1

    while first <= last:
        mid = (first + last)//2
        if e < L[mid]:
            last = mid -1
        elif e > L[mid]:
            first = mid + 1
        else:
            return mid
    return -1
```

The binary search algorithm is more efficient than the linear search algorithm for most lists.

Binary search may only be applied if the data in the list is sorted.

09_binarySearch.py

Binary Search - Recursive

```
def binarySearch(arr, sVal, startInd, endInd):
    if(startInd > endInd):
        return -1
    else:
        mid = (startInd + endInd)//2
        if(arr[mid] == sVal):
            return mid

        elif(arr[mid] > sVal):
            return binarySearch(arr, sVal, startInd, mid-1)
        else:
            return binarySearch(arr, sVal, mid+1, endInd)
```

See: 09_binarySearchRecursive.py

Efficiency of Binary Search

The complexity for both the recursive and iterative versions of binary search are the same.

On each step in the process, the binary search algorithm rules out half of the remaining possibilities.

In the worst case, the number of steps required is equal to the number of times you can divide the original size of the array in half until there is only one element remaining. In other words, what you need to find is the value of *k* that satisfies the following equation:

$$1 = N / 2 / 2 / 2 / 2 \ldots / 2$$

k times

You can simplify this formula using basic mathematics:

$$1 = N / 2^k$$

$$2^k = N$$

$$k = \log_2 N$$

O (log N)

Sorting Algorithms

Sorting data is one of the most important computing applications

Sorting is the process of arranging a list of items in a particular order

9 2 4 5 8 1 3

After sorting in ascending order:

1 2 3 4 5 8 9

There are many algorithms, which vary in efficiency, for sorting a list of items

We will examine three algorithms, Bubble Sort, Selection Sort and Merge Sort.

See Visual Sort

Bubble Sort

It’s called bubble sort because

- larger values gradually “bubble” their way upward to the end of the array like air bubbles rising in water.

The technique is to make several passes through the array.

- On each pass, pairs of adjacent elements are compared.
- If the pair is in increasing order, we leave the values as they are.
- If the pair is in decreasing order, we swap the values.

Pass 1

30, 50, 40, 80, 70, 10, 90, 60

30, 40, 50, 80, 70, 10, 90, 60

30, 40, 50, 80, 70, 10, 90, 60

30, 40, 50, 70, 80, 10, 90, 60

30, 40, 50, 70, 10, 80, 90, 60

30, 40, 50, 70, 10, 80, 90, 60

30, 40, 50, 70, 10, 80, 60, 90

Pass 2

30, 40, 50, 70, 10, 80, 60, 90

30, 40, 50, 70, 10, 80, 60, 90

30, 40, 50, 70, 10, 80, 60, 90

30, 40, 50, 10, 70, 80, 60, 90

30, 40, 50, 10, 70, 80, 60, 90

30, 40, 50, 10, 70, 60, 80, 90

Continue..

Bubble Sort

```
def bubbleSort (data):
    issorted = False
    j = 0
    while(j < len(data)-1 and not issorted):
        issorted = True
        for k in range(len(data)-j-1):
                if data[k] > data[k+1]:
                    issorted = False
                    temp = data[k]
                    data[k] = data[k+1]
                    data[k+1] = temp
        j = j + 1
```

See: 09_bubbleSort.py

Bubble Sort Complexity

```
def bubble_sort(L):
    swap = False
    while not swap:
        swap = True
        for j in range(1, len(L)):
            if L[j-1] > L[j]:
                swap = False
                temp = L[j]
                L[j] = L[j-1]
                L[j-1] = temp
```

O(len(L))

O(len(L))

- inner for loop is for doing the **comparisons**
- outer while loop is for doing **multiple passes** until no more swaps
- **$O(n^2)$ where n is len(L)**
  to do len(L)-1 comparisons and len(L)-1 passes

Selection Sort

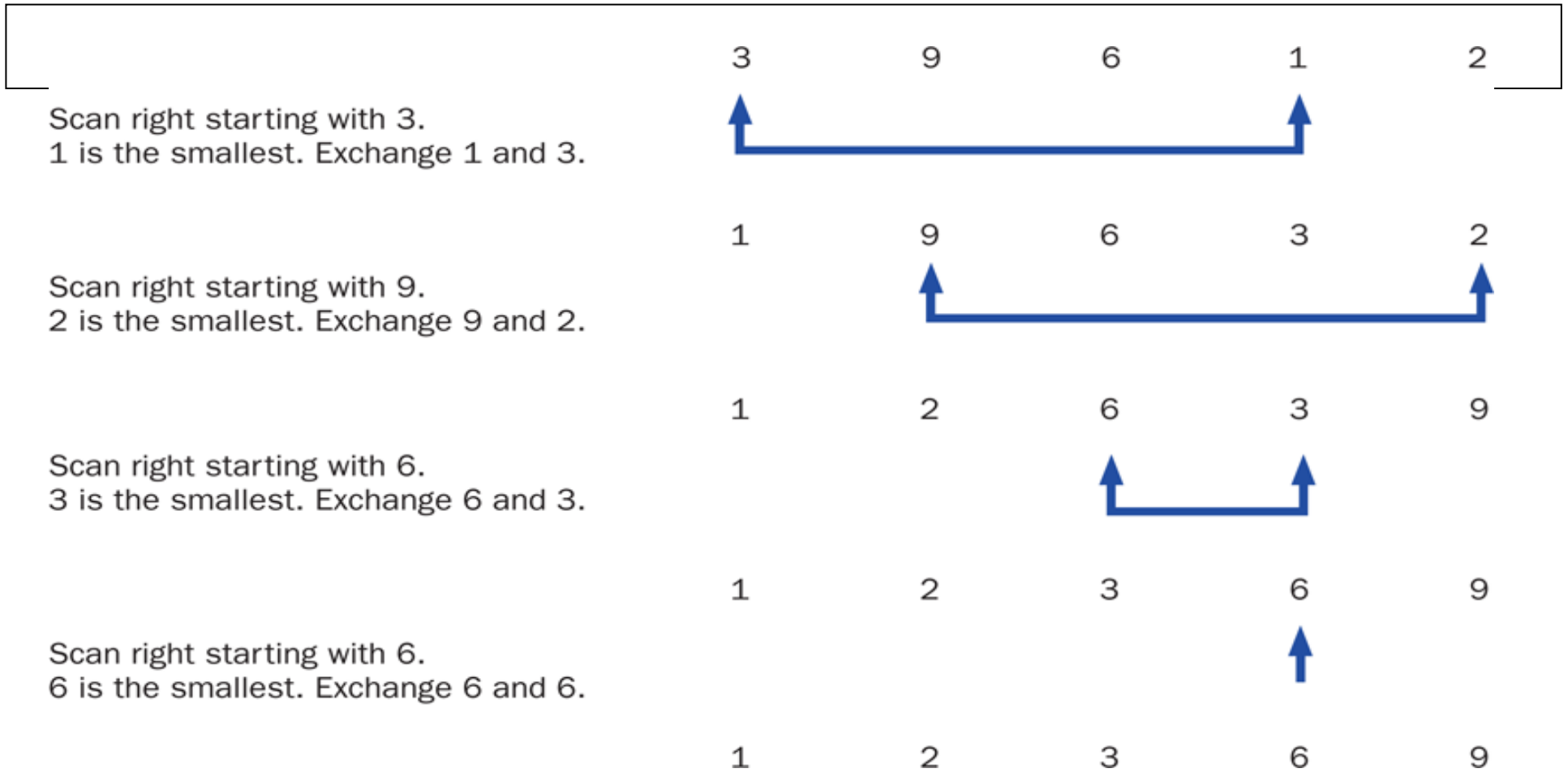
The strategy of Selection sort:

- select a value and put it in its final place in the list
- repeat for all other values

In more detail:

- find the smallest value in the list
- switch it with the value in the first position
- find the next smallest value in the list
- switch it with the value in the second position
- repeat until all values are in their proper places

Selection Sort



Selection Sort

	8
	5
	2
	6
	9
	3
	1
	4
	0
	7

Selection Sort Algorithm

```
def selectionSort(L):
    suffixStart = 0
    while suffixStart != len(L):
        for i in range(suffixStart, len(L)):
            if L[i] < L[suffixStart]:
                L[suffixStart], L[i] = L[i], L[suffixStart]
        suffixStart += 1
```

See: 09_selectionSort.py

Selection Sort Complexity

```
def selection_sort(L):
    suffixSt = 0
    while suffixSt != len(L):
        for i in range(suffixSt, len(L)):
            if L[i] < L[suffixSt]:
                L[suffixSt], L[i] = L[i], L[suffixSt]
        suffixSt += 1
```

len(L) times → O(len(L))

len(L) – suffixSt times → O(len(L))

- outer loop executes len(L) times
- inner loop executes len(L) – i times
- complexity of selection sort is **$O(n^2)$ where n is len(L)**

Lambda Functions/Expressions

Python supports the creation of inline functions, using the reserved word `lambda`.

A lambda function is a small anonymous function.

A lambda function can take any number of arguments, but can only have one expression

```
lambda < sequence of variable names >: <expression>
```

Example:

```
x = lambda a : a + 10
print(x(5))
```

The variable `x` is assigned the lambda function, which adds 10 to the parameter value.

The statement `x(5)`, `a` is bound to 5, and after adding 10, the value 15 is returned by the function.

MergeSort Algorithm

Merge sort is a divide and conquer algorithm.

It is easily described recursively:

1. If the list is of length 0 or 1, it is already sorted.
2. If the list has more than one element, split the list into two lists, and use merge sort to sort each of the lists.
3. Merge the lists.

See: `09_mergeSortRecursive.py`

```
def merge(left, right, compare):
    """Assumes left and right are sorted lists, compare defines an ordering on the elements.
    Returns a new sorted (by compare) list containing same elements as (left + right) would contain."""
    result = []
    i,j = 0, 0
    while i < len(left) and j < len(right):
        if compare(left[i], right[j]):
            result.append(left[i])
            i += 1
        else:
            result.append(right[j])
            j += 1
    while (i < len(left)):
        result.append(left[i])
        i += 1
    while (j < len(right)):
        result.append(right[j])
        j += 1
    return result
def mergeSort(L, compare = lambda x,y:x < y):
    """L is a list, compare defines an ordering on elements of L Returns new sorted list containing same
       elements as L"""
    if len(L) < 2:
        return L[:]
    else:
        middle = len(L)//2
        left = mergeSort(L[:middle], compare)
        right = mergeSort(L[middle:], compare)
        return merge(left, right, compare)
```

Efficiency of Merge Sort

Merge sort has a log-linear complexity, nlog(n), where n is the size or length of the list being sorted.

When comparing merge sort with selection and bubble sort , merge sort is better than the other two.

For example, if L has 10000 elements, selection sort has a complexity of 100 million, but merge sort is approximately 130000.

Although it is more efficient, merge sort requires more memory for the copies of the list (sorting is not done in place).

O (nlog(n))

Sorting and Searching

Amortization means to take the cost of a search and spread it across multiple searches.

If a list is sorted, the time to search a value can be reduced, but sorting a list comes at a cost.

If the list will be searched many times, it makes sense to pay the overhead of sorting the list once and amortize the cost of the sort over many searches.

Terms of Use

- This presentation was adapted from lecture materials provided in MIT Introduction to Computer Science and Programming in Python.
- Licenced under terms of Creative Commons License.