Algorithm Analysis

Asymptotic Notations

Haitham A. El-Ghareeb

December 6, 2019

Faculty of Computers and Information Sciences
Mansoura University
Egypt
helghareeb@mans.edu.eg

Contacts

- https://www.haitham.ws
- https://youtube.com/helghareeb
- https://www.github.com/helghareeb
- http://eg.linkedin.com/in/helghareeb
- helghareeb@mans.edu.eg

Algorithms

Algorithms

- Algorithms are designed to solve problems.
- A problem can have multiple solutions.
- Question
  - How do we determine which solution is the most efficient?

Algorithms

Execution Time

Execution Time

- Measure execution time:
  - construct a program for a given solution.
  - execute the program.
  - time it using a “wall clock”.

Example

```
$ python3 -m timeit '[print(x) for x in range(100)]'
100 loops, best of 3: 11.1 msec per loop
$ python3 -m timeit '[print(x) for x in range(10)]'
1000 loops, best of 3: 1.09 msec per loop
# We can see that the time per loop changes depending on the input!
```

Dependent on

- amount of data
- type of hardware and time of day
- programming language and compiler

Algorithms

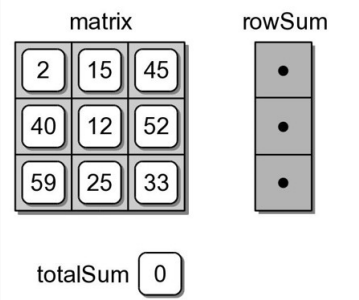
Complexity Analysis

Complexity Analysis

- What if we examine the solution itself and measure critical operations:
  - logical comparisons
  - assignments
  - arithmetic operations

Example Algorithm: Sum of Entire Matrix

- Given a matrix of size n x n, compute the:
  - sum of each row of a matrix.
  - overall sum of the entire matrix.



Example Algorithm: 01

```
total_sum = 0
for i in in range(n):
  row_sum[i] = 0
  for j in range(n):
    row_sum[i] = row_sum[i] + matrix[i, j]
    total_sum = total_sum + matrix[i,j]
```

Example Algorithm: 02

```
total_sum = 0
for i in range(n):
  row_sum[i] = 0
  for j in range(n):
    row_sum[i] = row_sum[i] + matrix[i,j]
  total_sum = total_sum + row_sum[i]
```

Compare The Results

- Number of additions:
  - v1: $2n^2$
  - v2: $n^2 + n$
- Second version has fewer additions ($n > 1$)
  - Will execute faster than the first.
  - Difference will not be significant.

Algorithms

Growth Rates

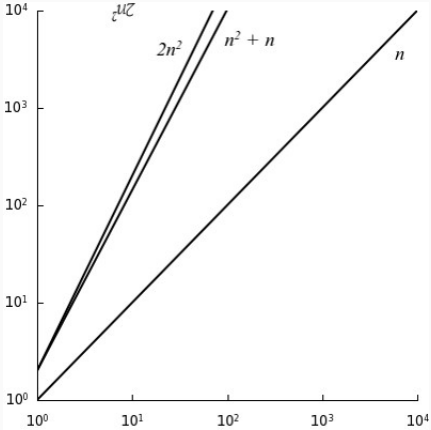
Growth Rates

- As n increases, both algorithms increase at approx the same rate:

Table 1: Growth rate comparisons for different input sizes

n	$2n^2$	$n^2 + n$
10	200	110
100	20,000	10,100
1000	2,000,000	1,001,000
10,000	200,000,000	100,010,000
100,000	20,000,000,000	10,000,100,000

Growth Rates



Asymptotic Analysis

Asymptotic Analysis

Compare Two Algorithms

Asymptotic Analysis

- Given two algorithms for a task, how do we find out which one is better?
  - implement both the algorithms and run the two programs on your computer for different inputs and see which one takes less time
  - There are many problems with this approach for analysis of algorithms
    1. It might be possible that for some inputs, first algorithm performs better than the second. And for some inputs second performs better.
    2. It might also be possible that for some inputs, first algorithm perform better on one machine and the second works better on other machine for some other inputs.

Asymptotic Analysis

Asymptotic Analysis Idea

Asymptotic Analysis

- Asymptotic Analysis is the big idea that handles above issues in analyzing algorithms
- In Asymptotic Analysis, we evaluate the performance of an algorithm in terms of input size (we don't measure the actual running time)
- We calculate, how does the time (or space) taken by an algorithm increases with the input size.

Asymptotic Analysis

It is Not Perfect

Challenge 01

- Not Perfect
- However, The best way available for analyzing algorithms
- Example
  - Say there are two sorting algorithms that take $1000nLogn$ and $2nLogn$ time respectively on a machine
  - Both of these algorithms are asymptotically same (order of growth is $nLogn$)
- We can't judge which one is better as we ignore constants

Challenge 02

- we always talk about input sizes larger than a constant value
- It might be possible that
  - those large inputs are never given to your software
  - and an algorithm which is asymptotically slower, always performs better for your particular situation
- So, you may end up choosing an algorithm that is Asymptotically slower but faster for your software.

Asymptotic Analysis

Three Cases

Three Cases

We can have three cases to analyze an algorithm:

1. Worst Case
2. Average Case
3. Best Case

Asymptotic Analysis

Search as an Example

Searching

- The process of selecting particular information from a collection of data based on specific criteria.
  - Can be performed on different data structures.
  - sequence search – search within a sequence.
  - search key (or key) – identifies a specific item.
  - compound key – consists of multiple parts.

Asymptotic Analysis

Linear Search

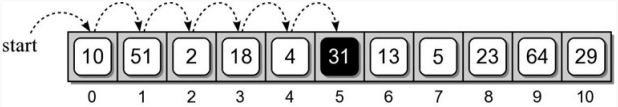
Linear Search

- Iterates over the sequence, item by item, until the specific item is found or the list is exhausted.
  - The simplest solution to sequence search problem.
  - Python's in operator: find a specific item.

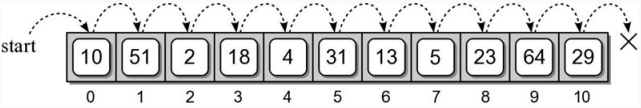
```
if key in the_array :
  print( 'The key is in the array.' )
else :
  print( 'The key is not in the array.' )
```

Linear Search Example

- Searching for 31



- Searching for 8



Linear Search - Example 01

```
def linear_search( the_values, target ) :
  n = len( the_values )
  for i in range( n ) :
    if the_values[i] == target
      return True
```

Linear Search - Example 02

```
def linear_search(arr, n, x):
    i = 0
    for i in range(i, n):
        if (arr[i] == x):
            return i
    return -1

arr = [1, 10, 30, 15]
x = 30
n = len(arr)
print(x, "is present at index",
             linear_search(arr, n, x))

30 is present at index 2
```

Asymptotic Analysis

Worst Case Analysis - Big-O

Worst Case - Big O

- Calculate **upper bound** on running time of an algorithm
- We must know the case that causes maximum number of operations to be executed
- For Linear Search, the worst case happens when the element to be searched is not present in the array
- When x is not present, the *linear_search*() function compares it with all the elements of $arr[i]$ one by one
- Worst case time complexity of linear search would be $O(n)$.

Asymptotic Analysis

Average Case Analysis - Big Theta

Average Case - Big Theta

- Take all possible inputs and calculate computing time for all of the inputs
- Sum all the calculated values and divide the sum by total number of inputs
- We must know (or predict) distribution of cases
- For the linear search problem, let us assume that all cases are uniformly distributed (including the case of x not being present in array)
- So we sum all the cases and divide the sum by (n+1)
- Following is the value of Average Case Time (ACT) complexity

$$ACT = \frac{\sum_{i=1}^{n+1} \theta(i)}{(n+1)} = \theta(n) \quad (1)$$

Asymptotic Analysis

Best Case Analysis - Big Omega

Best Case - Big Omega

- We calculate lower bound on running time of an algorithm
- We must know the case that causes minimum number of operations to be executed
- In the linear search problem, the best case occurs when x is present at the first location
- The number of operations in the best case is constant (not dependent on n)
- So time complexity in the best case would be $\omega(1)$

General Notes

1. Most of the times, we do worst case analysis to analyze algorithms. In the worst analysis, we guarantee an upper bound on the running time of an algorithm which is good information.
2. The average case analysis is not easy to do in most of the practical cases and it is rarely done. In the average case analysis, we must know (or predict) the mathematical distribution of all possible inputs.
3. The Best Case analysis is bogus. Guaranteeing a lower bound on an algorithm doesn't provide any information as in the worst case, an algorithm may take years to run.
4. For some algorithms, all the cases are asymptotically same, i.e., there are no worst and best cases.

Big-O

Big-O

Definition

Big-O Definition - 01

- Given a function T(n)
  - No. of steps required for an input of size n.
  - Ex: $T_2(n) = n^2 + n$
- Suppose there exist a function f(n) for all integers $n > 0$ such that

$$T(n) \leq cf(n)$$

- for some constant c and for all large values of $n > m$ (a constant).

Big-O Definition - 02

- Then, the algorithm has a time-complexity of or executes "on the order of" $f(n)$
  - We use the notation: $O(f(n))$
  - Big-O is intended for large values of n.

Big-O

Examples

Example 01

- Consider the previous sample algorithms (Matrix)
- Version 1: $T_1(n) = 2n^2$

$$T1(n) < cf(n) \tag{2}$$

$$2n^2 < 2n^2 \tag{3}$$

Let c = 2

$$O(n^2) \tag{4}$$

Example 01

- Consider the previous sample algorithms (Matrix)
- Version 2: $T_1(n) = n^2 + n$

$$T2(n) < cf(n) \tag{5}$$

$$n^2 + n \leq 2n^2 \tag{6}$$

Let c = 2

$$O(n^2) \tag{7}$$

Upper Bound

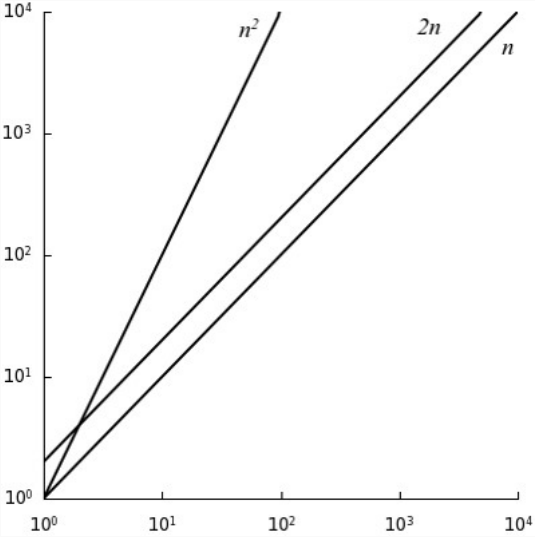
- There is more than one f(n) for an algorithm
- $$n^2 + n < cf(n)$$
- n^2 is not the only choice
- $f(n)$ could be n^2, n^3, n^4
- **Objectives**
  - find an f(n) that provides the tightest (lowest) upper bound

Constant of Proportionality

- Is it important?
- Consider two algorithms:
  - $O(n^2)$, with c $= 1$
  - $O(2n)$, with c $= 2$

n	n^2	$2n$
10	100	20
100	10,000	200
1000	1,000,000	2,000
10,000	100,000,000	20,000
100,000	10,000,000,000	200,000

Constant of Proportionality



Big-O

Examples

Constructing T(n)

- We don't count total number of specific instructions (math operations, comparisons, etc)
  - Assume each basic statement takes the same time, constant time.
  - Total number of steps required:

$$T(n) = f1(n) + f2(n) + \cdots + fk(n) \tag{8}$$

Choosing the Function

- Given $T(n)$, chose the dominant term

$$T(n) = n^2 + log_2 n + 3n$$

- n^2 dominates the other terms (for $n > 3$)

$$n^2 + log_2 n + 3n \leq n^2 + n^2 + n^2$$

$$n^2 + log_2 n + 3n \leq 3n^2$$

Choosing the Function

- What is the dominant term for the following expression?

$$T(n) = 2n^2 + 15n + 500$$

Answer

- When $n < 16$, 500 dominates
- When $n > 16$, n^2 is the dominate term

Big-O

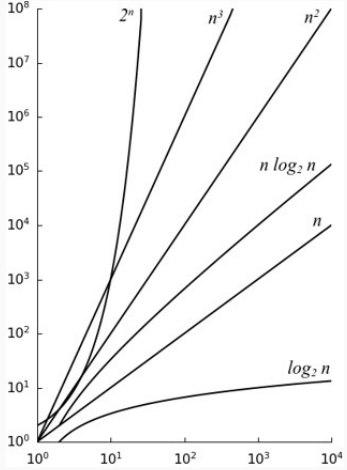
Classes of Algorithms

Classes of Algorithms

- Many algorithms have a time-complexity selected from a common set of functions

$f(n)$	Common Name
1	constant
log n	logarithmic
n	linear
n log n	log linear
n^2	quadratic
n^3	cubic
a^n	exponential

Classes of Algorithms



Evaluating String Operations

True / False

1. Most of the string operations have a time-complexity that is proportional to the length of the string
2. For most problems that do not involve string processing, string operations seldom have an impact on the run time of an algorithm

Evaluating Python Code

Evaluating Python Code

- Basic operations only require constant time:
  - x = 5
  - z = x + y * 6
  - if $x > 0$ and $x < 100$
- What about function calls?
  - y = ex1(n)

Answer

- To determine the run time of the previous statement, we must know the cost of the function call ex1(n)
- The time required by a function call is the time it takes to execute the given function

Evaluating Python Code

ex_01.py

ex_01.py

```
def ex1( n ):
    total = 0
    for i in range( n ) :
        total += i
    return total
```

Answer

- Time required to execute a loop depends on the number of iterations performed and the time needed to execute the loop body during each iteration
- Loop will be executed n times and the loop body only requires constant time since it contains a single basic instruction
- Underlying mechanism of the for loop and the range() function are both O(1)
- $T(n) = n * 1$ for a result of $O(n)$
- What about the other statements in the function?

Answer

- The first line of the function and the return statement only require constant time
- It is common to omit the steps that only require constant time and instead focus on the critical operations (those that contribute to the overall time)
- In most instances, this means we can limit our evaluation to repetition and selection statements and function and method calls since those have the greatest impact on the overall time
- Since the loop is the only non-constant step, the function $ex1()$ has a run time of $O(n)$
- Statement $y = ex1(n)$ requires linear time

Evaluating Python Code

ex_02.py

ex_02.py

```
def ex2( n ):
    count = 0
    for i in range( n ) :
        count += 1
    for j in range( n ) :
        count += 1
    return count
```

Answer

- To evaluate the function, we have to determine the time required by each loop
- The two loops each require $O(n)$ time as they are just like the loop in function $ex1()$ earlier
- If we combine the times, it yields $T(n) = n + n$ for a result of $O(n)$

Evaluating Python Code

ex_03.py

ex_03.py

```
def ex3( n ):
    count = 0
    for i in range( n ) :
        for j in range( n ) :
            count += 1
    return count
```

Answer

- Nested Loops
- Both loops will be executed n times, but since the inner loop is nested inside the outer loop, the total time required by the outer loop will be $T(n) = n * n$, resulting in a time of $O(n^2)$ for the ex3() function
- Question
  - Not all nested loops result in a quadratic time

Evaluating Python Code

ex_04.py

ex_04.py

```
def ex4( n ):
    count = 0
    for i in range( n ) :
        for j in range( 25 ) :
            count += 1
    return count
```

ex_04.py

- $O(n)$
- The function contains a nested loop, but the inner loop executes independent of the size variable n
- Since the inner loop executes a constant number of times, it is a constant time operation
- The outer loop executes n times, resulting in a linear run time

Evaluating Python Code

ex_05.py

ex_05.py

```
def ex5( n ):
    count = 0
    for i in range( n ) :
        for j in range( i+1 ) :
            count += 1
    return count
```

- How many times does the inner loop execute?

Answer: Quadratic

- It depends on the current iteration of the outer loop
  - On the first iteration of the outer loop, the inner loop will execute one time
  - on the second iteration, it executes two times
  - on the third iteration, it executes three times
  - and so on until the last iteration when the inner loop will execute n times
- The time required to execute the outer loop will be the number of times the increment statement $count+ = 1$ is executed
- Since the inner loop varies from 1 to n iterations by increments of 1, the total number of times the increment statement will be executed is equal to the sum of the first n positive integers:

$$T(n) = \frac{n(n+1)}{2} = \frac{n^2 + n}{2} \tag{9}$$

Logarithmic Time Examples

Logarithmic Time Examples

Example 01

Example 01

- Contains a single loop, the change to the modification step is incrementing (or decrementing) by one
- it cuts the loop variable in half each time through the loop

```
def ex6( n ):
    count = 0
    i = n
    while i >= 1 :
        count += 1
        i = i // 2
    return count
```

Explanation

- When the size of the input is reduced by half in each subsequent iteration, the number of iterations required to reach a size of one will be equal to

$$\lfloor log_2 n \rfloor + 1$$

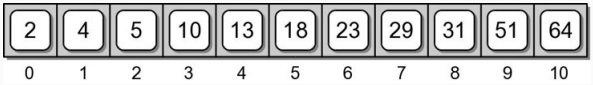
- or the largest integer less than $log_2 n$, plus 1
- In our example of $n = 16$, there are $log_2 16 + 1$, or four iterations

Logarithmic Time Examples

Example 02 - Binary Search

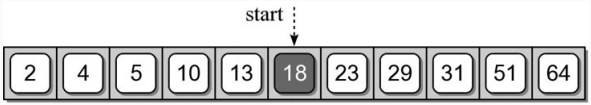
Binary Search

- The linear search has a linear time-complexity
  - We can improve the search time if we modify the search technique itself
  - Use a divide and conquer strategy
  - Requires a sorted sequence



Binary Search Algorithm

- Examine the middle item (searching for 10):



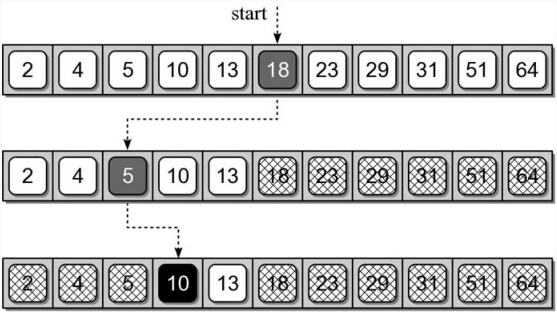
- One of three possible conditions:
  - target is found in the middle item
  - target is less than the middle item
  - target is greater than the middle item

Binary Search Algorithm

- Since the sequence is sorted, we can eliminate half the values from further consideration

Binary Search Algorithm

- Repeat the process until either the target is found or all items have been eliminated



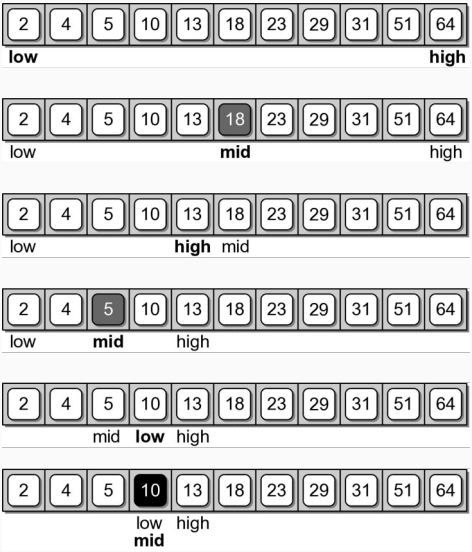
Binary Search Implementation

```
def binary_search( the_values, target ) :
  low = 0
  high = len(the_values) - 1

  while low <= high :
    mid = (high + low) // 2
    if theValues[mid] == target :
      return True
    elif target < the_values[mid] :
      high = mid - 1
    else :
      low = mid + 1

  return False
```

Binary Search Utilization

```
if __name__ == '__main__':
    vals = range(0,20,2)
    print(binary_search(vals, 5))
    print(binary_search(vals, 10))
    print(binary_search(vals, 20))

False
True
False
```

Binary Search Illustration



Back to Python Cases

Back to Python Cases

ex_07.py

ex_07.py

```
def ex7( n ):
    count = 0
    for i in range( n )
        count += ex6( n )
    return count
```

ex_07.py

```
def ex7( n ):
    count = 0
    for i in range( n )
        count += ex6( n )
    return count
```

- O(n log n)

Different Cases

Different Cases

```
def findNeg( intSeq ):
  n = len( intSeq )
  for i in range( n ) :
    if intSeq[i] < 0 :
      return i
  return None

L = [ 72, 4, 90, 56, 12, 67, 43, 17, 2, 86, 33 ]
p = findNeg( L )

L = [ -12, 50, 4, 67, 39, 22, 43, 2, 17, 28 ]
p = findNeg( L )
```

Evaluating the Python List

Evaluating the Python List

Why Python List

Why Python List Performance

- We used the list to implement many of our ADTs
- Their efficiency depends on the efficiency of Python's list

Evaluating the Python List

List Traversal

List Traversal: O(n)

- Iterates over the contiguous elements of the underlying array

```
# Sum the elements of a list.
sum = 0
for value in valueList :
  sum = sum + value

# Alternate version.
sum = 0
n = len(valueList)
for i in range( n ) :
  sum = sum + valueList[i]
```

Evaluating the Python List

List Allocation

List Allocation: O(1), O(n)

- Creating a non-empty list is not constant

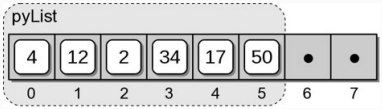
```
temp = list() # O(1)
listX = [ 0 ] * n # O(n)
valueList = [ 4, 8, 20, 2, 15, 89, 60, 75 ] # O(n)
```

Evaluating the Python List

Appending to List

Python List: Appending - O(1)

- When space is available, the item is stored in the next slot



- What if the underlying array is full?

Python List: Expansion - O(n)

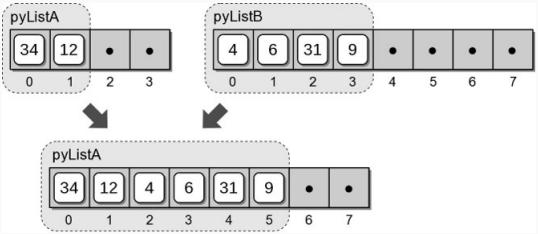
1. Step 1: create a new array, double the size
2. Step 2: copy the items from original array to the new array
3. Step 3: replace the original array with the new array
4. Step 4: store the new value the next slot of the new array

Evaluating the Python List

Extending

Extending Python List - O(n)

- Adds the contents of a source list to the end of the destination list



Evaluating the Python List

Python Lists - Time Complexities

Time Complexities

List Operation	Worst Case
v = list()	O(1)
len(v)	O(1)
v = [0] * n	O(n)
v[i] = x	O(1)
v.append(x)	O(n)
v.extend(w)	O(n)
v.insert(x)	O(n)
v.pop()	O(1)
traversal	O(n)

Amortized Cost

Amortized Cost

- Consider a sequence of n append operations

```
L = list()
for i in range( 1, n+1 ):
  L.append( i )
```

- What is the worst-case running time?

Special Case

- The append() method introduces a special case
  - available capacity: O(1)
  - expansion required: O(n)
- How many times does append require O(n) time?

Amortized Analysis

- Given a sequence of operations, compute the time-complexity by computing the **average cost** over the entire sequence
  - Cost per operation must be known
  - Cost must vary, with many ops contributing little cost
  - only a few ops contributing high cost

Aggregate Method

- Determine upper bound total cost: $T(n)$
- Calculate average cost: $T(n) / n$
- Example: sequence of n append operations
  - Storage of a single item: $O(1)$
  - Expansion only occurs when $(i - 1)$ is a power of 2
  - Cost of the expansion based on current array size

Amortized Analysis

- The append() operation:
  - worst-case time: O(n)
  - amortized cost: O(1)
- Can only be used for a long sequence of append operations

Assignment

Evaluating the Set ADT

Set Operation	Worst Case
s = Set()	O(1)
len(s)	O(1)
x in s	O(n)
s.add(x)	O(n)
s.is_subset_of(t)	$O(n^2)$
s == t	$O(n^2)$
s.union(t)	$O(n^2)$
traversal	O(n)

Sorting

Sorting

Introduction

Sorting

- The process of arranging a collection of items such that each item and its successor satisfy a prescribed relationship
  - sequence sort – sorting within a sequence
  - sort key – values on which items are ordered
  - items arranged in ascending or descending order
  - sorted in place – within the same structure

Sorting

Bubble Sort

Bubble Sort

- A simple solution to the sorting problem.
- Arranges the items by
  - iterating over the sequence multiple times
  - larger values bubble to the top (or end)

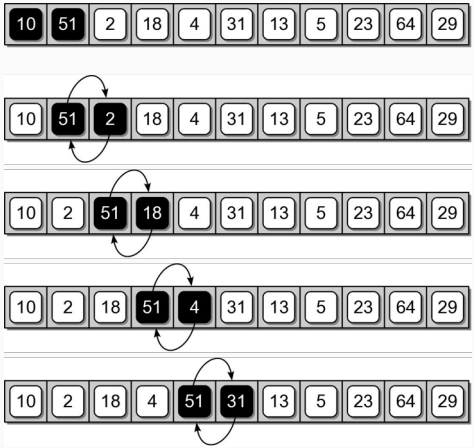
Sorting

Implementation

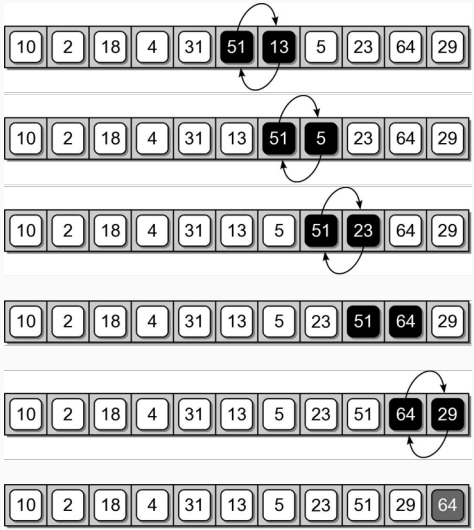
Bubble Sort Implementation

```
def bubble_sort( values ):
    n = len( values )
    for i in range( n ) :
        for j in range( n - 1) :
            if values[j] > values[j + 1] :
                values[j], values[j+1] = values[j+1], values[j]
        #print(values)
    return values


if __name__ == '__main__':
    val = [10, 51, 2, 18, 4, 31, 13, 5, 23, 64, 29]
    print(bubble_sort(val))
```

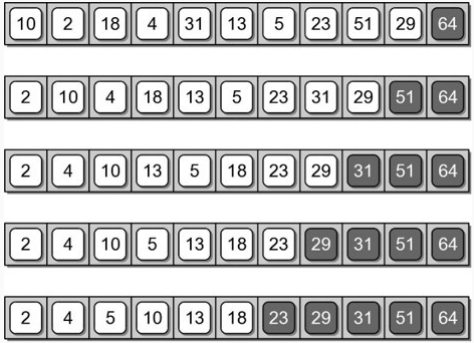
Bubble Sort - First Complete Iteration - 01



Bubble Sort - First Complete Iteration - 02



Bubble Sort - Results after each iteration of the outer loop



Sorting

Selection Sort

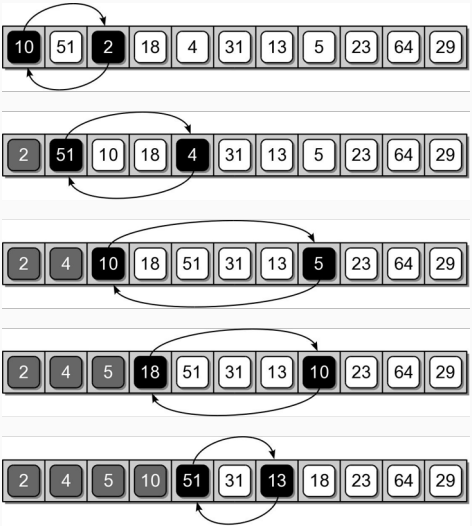
Selection Sort

- Improves on the bubble sort
- Works in a fashion similar to what a human may use to sort a sequence
- Instead of swapping many items,
  - repeatedly selects the next largest item from among the unsorted items
  - requires a search to select the smallest item

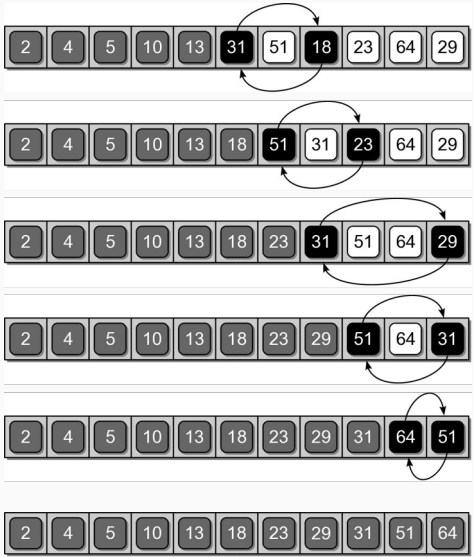
Implementation

```
def selection_sort( values ):
    n = len( values )
    for i in range( n - 1 ):
        smallNdx = i
        for j in range( i + 1, n ):
            if values[j] < values[smallNdx] :
                smallNdx = j

        if smallNdx != i :
            values[i], values[smallNdx] = values[smallNdx], values[i]
        print(values)
    return values
```

Selection Sort in Action - 01



Selection Sort in Action - 02



Sorting

Insertion Sort

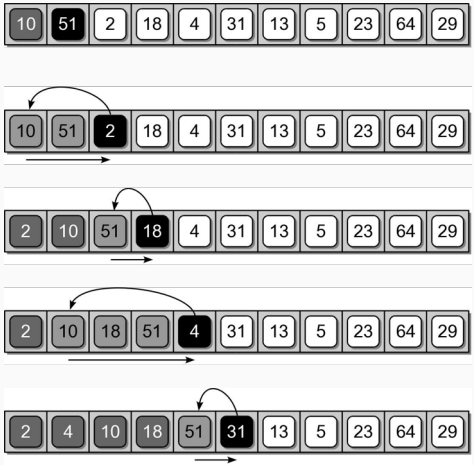
Insertion Sort

- Another commonly studied algorithm
- Arranges the items by
  - iterating over the sequence one complete time
  - inserts each unsorted item into its proper place

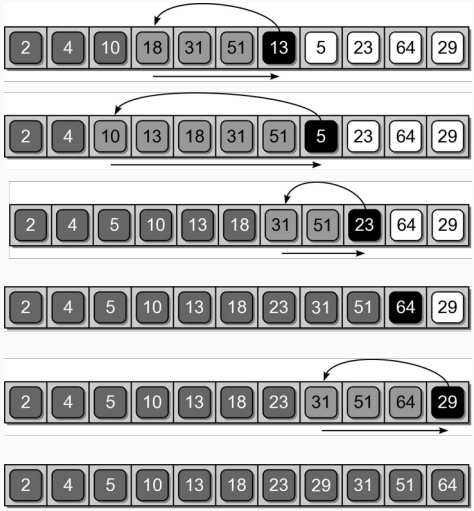
Insertion Sort Implementation

```
def insertion_sort( values ):
    n = len( values )
    for i in range( 1, n ) :
        value = values[i]
        pos = i
        while pos > 0 and value < values[pos - 1] :
            values[pos] = values[pos - 1]
            pos -= 1

        values[pos] = value

    return values
```

Insertion Sort in Action - 01



Insertion Sort in Action - 02



Summary

Summary

- Algorithms
- Execution Time
- Asymptotic Analysis
- Linear Search
- Big-O
- Evaluating Python Code
- Binary Search
- Amortized Analysis

Summary

- Sorting
- Bubble Sort
- Selection Sort
- Insertion Sort

https://www.haitham.ws