

CDS Scribing: Lecture 10

Algorithms: Searching, Sorting

Algorithm

An algorithm is a sequence of computational steps performed on a data input to generate a required result or solve a problem. It is an abstraction of program to be executed on a physical machine. There can be more than one solution (more than one algorithm) to solve a given problem. An algorithm can be implemented using different programming languages on different platforms. Analysis of algorithm is an important part of the broader computational complexity theory that provides theoretical estimates for the resources needed by an algorithm, solving a given computational problem.

Algorithm Efficiency

In theoretical analysis of algorithm, the efficiency is generally measured in the asymptotic sense, i.e. to estimate the complexity for an arbitrary large input. The asymptotic measures are helpful because different implementation of the same algorithm may differ in efficiency by a hidden factor. The performance of an algorithm are analysed from three aspects –

1. Time – It is the time taken by an algorithm to complete an operation. Time Complexity analysis typically expresses the runtime of an algorithm in terms of the size of the input using the Big O notation.
2. Space – This section is concerned with the amount of main memory (RAM) utilised while executing the algorithm on a specific data set, using the space-complexity analysis. The four aspects to consider are –
 - a. The amount of memory needed to hold the program.
 - b. The amount of memory needed for the input data
 - c. The amount of memory needed for the output data.
 - d. The amount of memory needed while executing the code of any underlying algorithm.
3. Cost – It computes the physical resources used by an algorithm and the cost of establishment and maintenance. It is used to compute the cost of ownership of hardware and equipment dedicated for an algorithm.

Cost Models

Time efficiency estimates depend on what we define to be a step. For the analysis to correspond usefully to the actual execution time, the time required to perform a step must be guaranteed to be bounded above by a constant. Two cost models are generally used:

1. Uniform Cost Model, also called uniform-cost measurement (and similar variations), assigns a constant cost to every machine operation, regardless of the size of the numbers involved
2. Logarithmic Cost Model, also called logarithmic cost measurement (and variations thereof), assigns a cost to every machine operation proportional to the number of bits involved

The latter is more cumbersome to use, so it's only employed when necessary, for example in the analysis of arbitrary-precision arithmetic algorithms, like those used in cryptography.

Algorithm Performance (Growth Rates)

Informally, an algorithm can be said to exhibit a growth rate on the order of a mathematical function if beyond a certain input size n , the function $f(n)$ times a positive constant provides an upper bound or limit for the run-time of that algorithm. In other words, for a given input size n greater than some n_0 and a constant c , the running time of that algorithm will never be larger than $c \times f(n)$. This concept is frequently expressed using Big O notation. For example, since the run-time of insertion sort grows quadratically as its input size increases, insertion sort can be said to be of order $O(n^2)$. Two other notational constructs used by computer scientists in the analysis of algorithms are Θ (Big Theta) notation and Ω (Big Omega) notation.

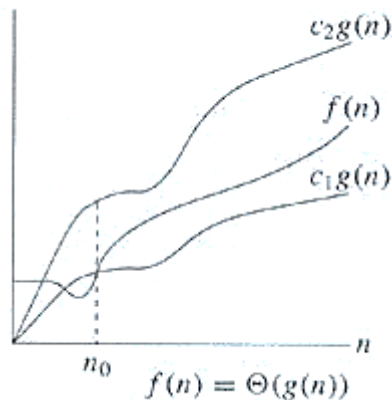
1. Θ -Notation (Same order)

This notation bounds a function to within constant factors. We say $f(n) = \Theta(g(n))$ if there exist positive constants n_0 , c_1 and c_2 such that to the right of n_0 the value of $f(n)$ always lies between $c_1 g(n)$ and $c_2 g(n)$ inclusive.

In the set notation, we write as follows:

$$\Theta(g(n)) = \{f(n) : \text{there exist positive constants } c_1, c_2, \text{ and } n_0 \text{ such that } 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \text{ for all } n \geq n_0\}$$

We say that $g(n)$ is an asymptotically tight bound for $f(n)$.



Graphically, for all values of n to the right of n_0 , the value of $f(n)$ lies at or above $c_1 g(n)$ and at or below $c_2 g(n)$. In other words, for all $n \geq n_0$, the function $f(n)$ is equal to $g(n)$ to within a constant factor. We say that $g(n)$ is an asymptotically tight bound for $f(n)$. In the set terminology, $f(n)$ is said to be a member of the set $\Theta(g(n))$ of functions. In other words, because $\Theta(g(n))$ is a set, we could write the following to indicate that $f(n)$ is a member of $\Theta(g(n))$.

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

Instead, we write the following to express the same notation.

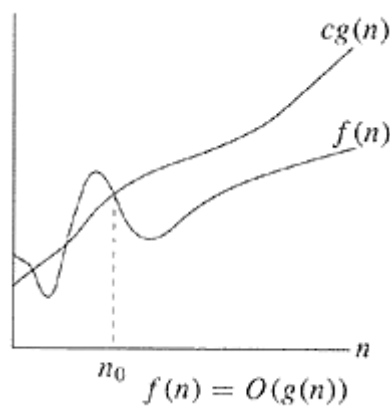
$$f(n) = \Theta(g(n))$$

2. O-Notation (Upper Bound)

This notation gives an upper bound for a function to within a constant factor. We write $f(n) = O(g(n))$ if there are positive constants n_0 and c such that to the right of n_0 , the value of $f(n)$ always lies on or below $c g(n)$. In the set notation, we write as follows:
For a given function $g(n)$, the set of functions,

$$O(g(n)) = \{f(n): \text{there exist positive constants } c \text{ and } n_0 \text{ such that } 0 \leq f(n) \leq c g(n) \text{ for all } n \geq n_0\}$$

We say that the function $g(n)$ is an asymptotic upper bound for the function $f(n)$. We use O-notation to give an upper bound on a function, to within a constant factor.



Graphically, for all values of n to the right of n_0 , the value of the function $f(n)$ is on or below $g(n)$. We write $f(n) = O(g(n))$ to indicate that a function $f(n)$ is a member of the set $O(g(n))$ i.e.

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

Note that $f(n) = \Theta(g(n))$ implies $f(n) = O(g(n))$, since Θ -notation is a stronger notation than O-notation.

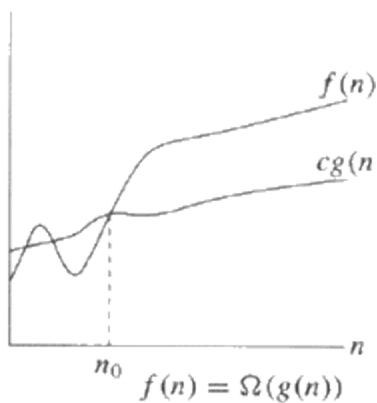
3. Ω -Notation (Lower Bound)

This notation gives a lower bound for a function to within a constant factor. We write $f(n) = \Omega(g(n))$ if there are positive constants n_0 and c such that to the right of n_0 , the value of $f(n)$ always lies on or above $c g(n)$. In the set notation, we write as follows:

For a given function $g(n)$, the set of function,

$$\Omega(g(n)) = \{f(n) : \text{there exist positive constants } c \text{ and } n_0 \text{ such that } 0 \leq c g(n) \leq f(n) \text{ for all } n \geq n_0\}$$

We say that the function $g(n)$ is an asymptotic lower bound for the function $f(n)$. The intuition behind Ω -notation is shown below.



Common Growth Rate Function:

There are some common categories of growth rate function like -

Function	Growth Rate Name
c	Constant
$\log N$	Logarithmic
$\log^2 N$	Log-squared
N	Linear
$N \log N$	
N^2	Quadratic
N^3	Cubic
2^N	Exponential

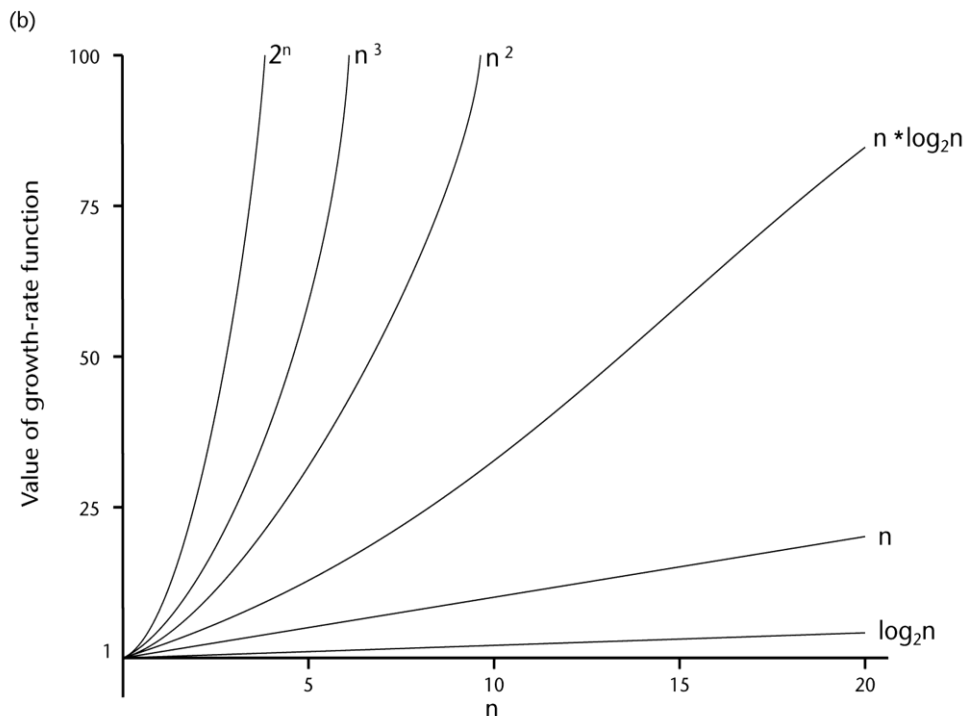


Figure 1: Growth rate functions with respect to input size (n)

Properties of Growth-Rate Functions

1. We can ignore low-order terms in an algorithm's growth-rate function.
 - a. If an algorithm is $O(n^3 + 4n^2 + 3n)$, it is also $O(n^3)$.
 - b. We only use the higher-order term as algorithm's growth-rate function.
2. We can ignore a multiplicative constant in the higher-order term of an algorithm's growth-rate function.
 - a. If an algorithm is $O(5n^3)$, it is also $O(n^3)$.
3. $O(f(n)) + O(g(n)) = O(f(n) + g(n))$
 - a. We can combine growth-rate functions.
 - b. If an algorithm is $O(n^3) + O(4n^2)$, it is also $O(n^3 + 4n^2)$. So, it is $O(n^3)$.
 - c. Similar rules hold for multiplication.

Algorithm Analysis

The complexity of an algorithm is a function $g(n)$ that gives the upper bound of the number of operation (or running time) performed by an algorithm when the input size is n .

There are two interpretations of upper bound-

1. Worst-case Complexity: The running time for any given size input will be lower than the upper bound except possibly for some values of the input where the maximum is reached.
2. Average-case Complexity: The running time for any given size input will be the average number of operations over all problem instances for a given size.

Because, it is quite difficult to estimate the statistical behaviour of the input, most of the time we content ourselves to a worst case behaviour. Most of the time, the complexity of $g(n)$ is approximated by its family $O(f(n))$ where $f(n)$ is one of the following functions- n (linear complexity), $\log n$ (logarithmic complexity), n^a where $a \geq 2$ (polynomial complexity), a^n (exponential complexity).

Basic Data Structures

In computer science, a data structure is a way to represent data that can be used efficiently for the future purposes. Mostly, data structures are of abstract data types which can be implemented in various ways. Different kinds of data structures are suited to different kinds of applications, and some are highly specialized to specific tasks. For example, databases use B-tree indexes for small percentages of data retrieval, and compilers and databases use dynamic hash tables as look-up tables.

Some useful data structures in our purpose are -

Array

An array (also called list) is a number of elements in a specific order, typically all of the same type. Elements are accessed using an integer index to specify which element is required (although the elements may be of almost any type). Typical implementations allocate contiguous memory words for the elements of arrays (but this is not always a necessity). Arrays may be fixed-length or resizable. Similar in built data structure defined in Python library is **List**. Advantage of using lists over arrays is that it can contain data of various data types like - integer, string, character etc. For example,

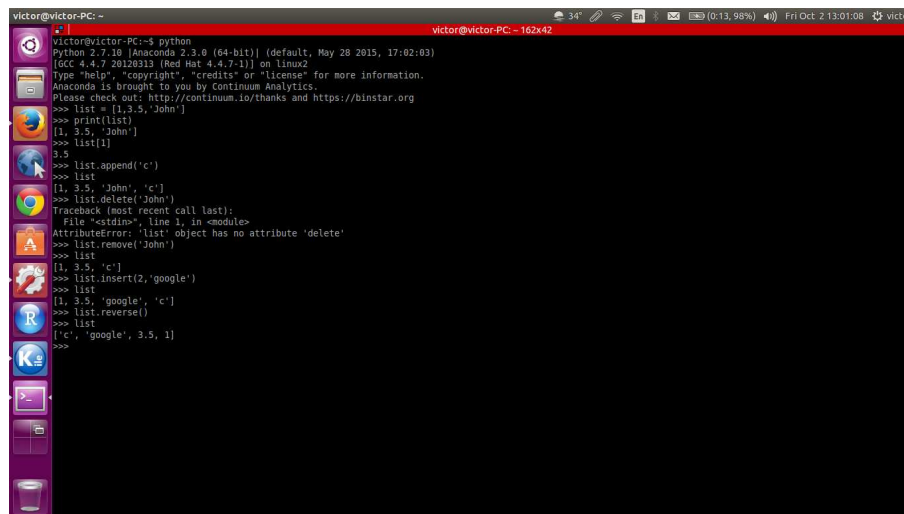
A screenshot of a terminal window titled 'victor@victor-PC: ~'. The terminal shows the execution of a Python script. The script starts with 'python', followed by 'list = [1, 3.5, 'John']'. It then prints the list, appends 'c', deletes 'John', removes 'John', inserts 'google' at index 2, reverses the list, and finally prints the list. The output shows the list evolving through these operations. The terminal window has a dark background with a light-colored text. The system tray at the bottom shows the date and time as 'Fri Oct 2 13:01:08 victor'.

Figure 1: Python lists

Stack

In computer science, a stack or LIFO (last in, first out) is an abstract data type that serves as a collection of elements, with two principal operations: push, which adds an element to the collection, and pop, which removes the last element that was added. Initially, we index the top of the stack as -1. As we push elements we increment the index and decrement the top index when we pop.

Operations on Stack			
	Stack's contents	TOP value	Output
1. Init_stack()	<empty>	-1	
2. Push('a')	a	0	
3. Push('b')	a b	1	
4. Push('c')	a b c	2	
5. Pop()	a b	1	c
6. Push('d')	a b d	2	c
7. Push('e')	a b d e	3	c
8. Pop()	a b d	2	c e
9. Pop()	a b	1	c e d
10. Pop()	a	0	c e d b
11. Pop()	<empty>	-1	c e d b a

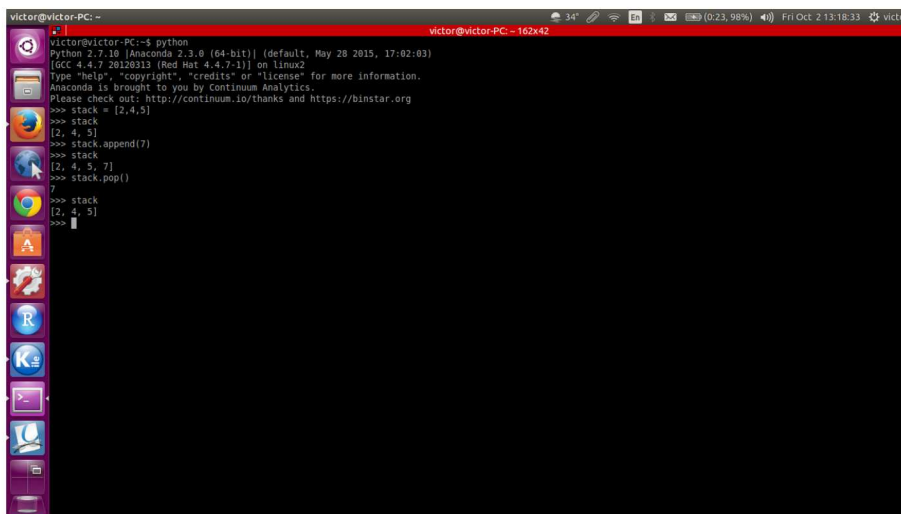
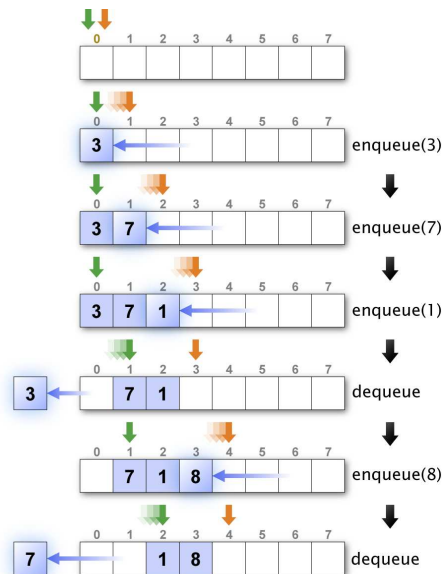


Figure 2: Python lists as stack

Queue

Queue is a FIFO (first in, first out) data structure where the first element added to the queue will be the first one to be removed. This is equivalent to the requirement that once a new element is added, all elements that were added before have to be removed before the new element can be removed.



```

victor@victor-PC: ~$ python
Python 2.7.10 [Anaconda 2.3.0 (64-bit)] (default, May 28 2015, 17:02:03)
[GCC 4.4.7 20120313 (Red Hat 4.4.7-1)] on linux2
Type "help()", "copyright()", "credits()" or "license()" for more information.
Anaconda is brought to you by Continuum Analytics.
Please check out: http://continuum.io/thanks and https://binstar.org

>>> from collections import deque
>>> queue = deque(['john', 'richard', 'mary'])
>>> queue
deque(['john', 'richard', 'mary'])
>>> queue.append('sarah')
>>> queue
deque(['john', 'richard', 'mary', 'sarah'])
>>> queue.popleft()
'john'
>>> queue
deque(['richard', 'mary', 'sarah'])
>>> queue.popleft()
'richard'
>>> queue
deque(['mary', 'sarah'])
>>>

```

Figure 3: Python lists as queue

Recursive Algorithm

A recursive algorithm is an algorithm which calls itself with "smaller (or simpler)" input values, and which obtains the result for the current input by applying simple operations to the returned value for the smaller (or simpler) input. More generally if a problem can be solved utilizing solutions to smaller versions of the same problem, and the smaller versions reduce to easily solvable cases, then one can use a recursive algorithm to solve that problem. For example, the elements of a recursively defined set, or the value of a recursively defined function can be obtained by a recursive algorithm.

Example:-

```
void hanoi(int n, char source, char dest, char spare) {
    if (n > 0) {
        hanoi(n-1, source, spare, dest);
        cout << "Move top disk from pole " << source
              << " to pole " << dest << endl;
        hanoi(n-1, spare, dest, source);
    }
}
```

The time-complexity function $T(n)$ of a recursive algorithm is defined in terms of itself, and this is known as recurrence equation for $T(n)$. To find the growth-rate function for a recursive algorithm, we have to solve its recurrence relation.

Searching Algorithms

Searching algorithms are important in all sorts of applications that we meet every day. In text editors we might want to search through a very large document. Let's assume a text file with a million characters for the occurrence of a given string (maybe dozen of characters). In text retrieval tools, we might want to search through thousands of such documents (though normally these files would be indexed, making this unnecessary). Other applications might require matching algorithms as part of a more complex algorithm.

Searching algorithms are closely related to the concept of dictionaries. Dictionaries are data structures that support search, insert and delete operations. Typically a simple function is applied to the key to determine its place in the directory. Another efficient search algorithm on sorted tables is binary search.

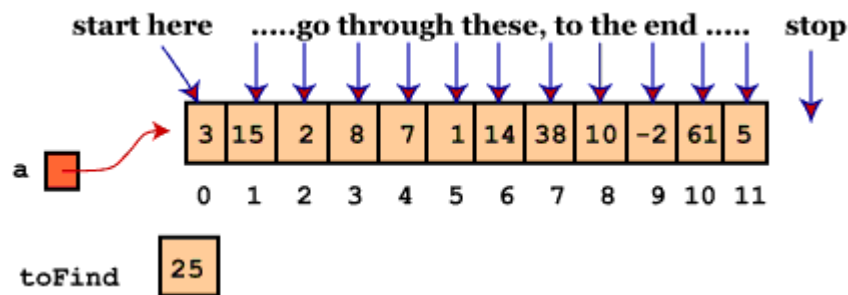
The search algorithm that are primarily used are:

1. Linear Search
2. Binary Search

Linear Search

We can think this as two different ways of finding our way in phonebook. A linear search is starting at the beginning, reading every name until we find what we are looking for. A binary search on the other hand is when we open the book (usually in the middle), look at the name on

top of the page and decide if the name we are looking for is bigger or smaller than the one we are looking for. If the same we are looking for is bigger than we continue searching the upper part of the book in this very fashion.



Pseudocode:

```
# Input: Array D, integer key
# Output: first index of key in D, or -1 if not found
For i = 0 to last index of D:
    if D[i] equals key:
        return
return -1
```

Asymptotic Analysis

Since this algorithm compares every element to find the required one its complexity in all the cases remains order of n i.e. $O(n)$ (where n is number of elements in the list) and its expected cost is also proportional to n provided that searching and comparing cost of all the elements is same.

Worst case time complexity -- $O(n)$

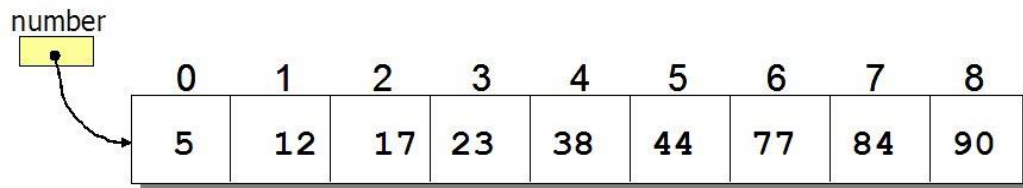
Average case time complexity – $O(n)$

So the idea is-

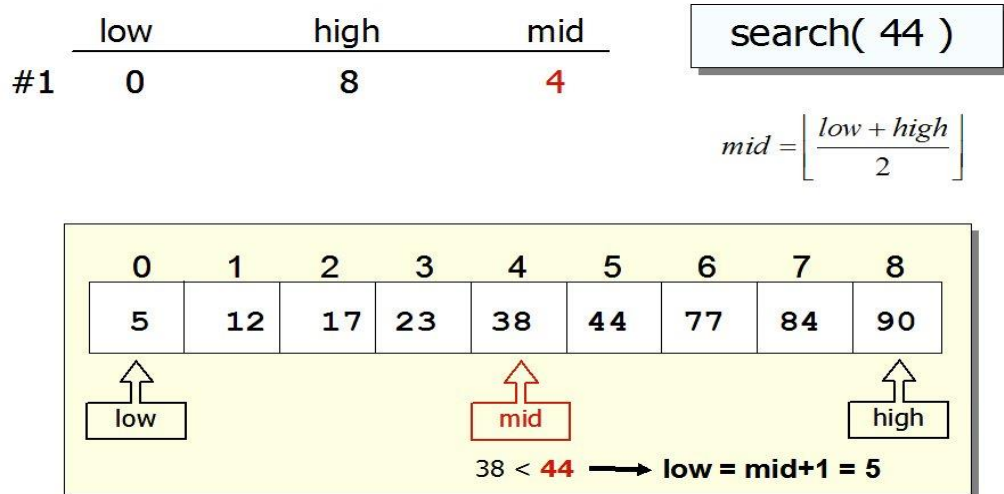
1. Start with the first element in the array or list.
2. Compare it with the given key, if key and value at current index are same, return the current index.
3. Else increase the index value and repeat step 2 until end of list is reached.

Binary Search:

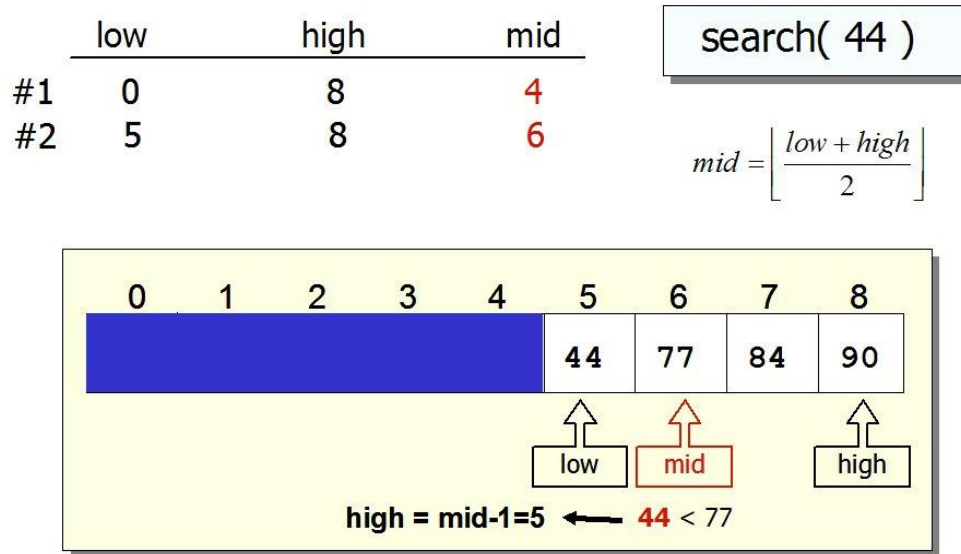
The Binary search algorithm depends on the array being already sorted.



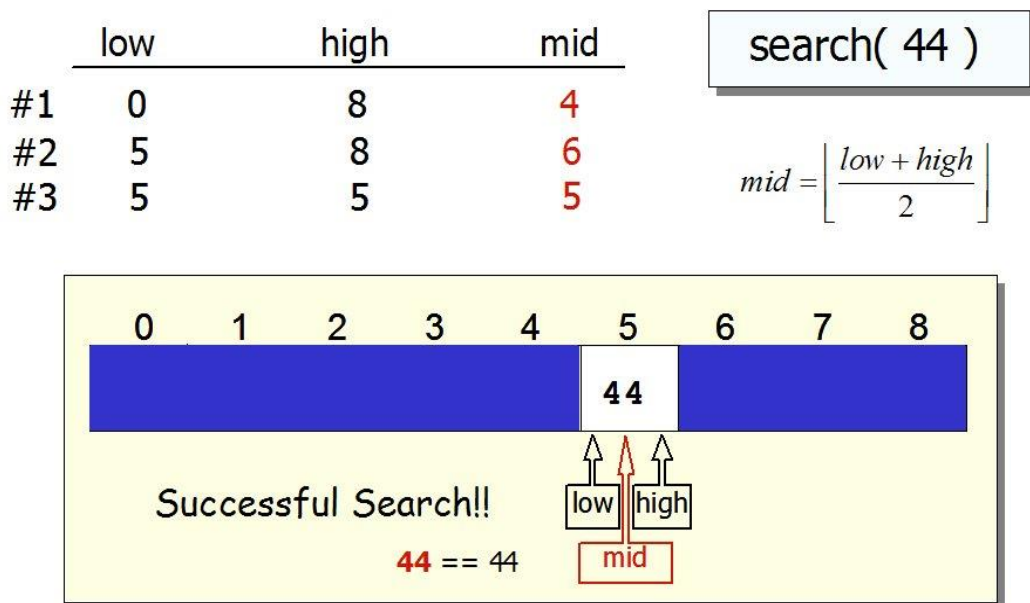
We need to look at three index low point, high point and mid point to perform a successful binary search. Please note that we need to use integer division to find the midpoint. First, we compare the value at the mid point to see if it is the value we are looking for (44). It is not in our case. So, we confirm that the value at the mid point is higher or lower than our search value? In this example, it is lower. Now, since the array is sorted, we know that the value we are searching for must be in the UPPER HALF of the array, since it is larger than the midpoint element value! So in one comparison, we have discarded the lower half of the array as elements that we need to search! This is a powerful tool for searching large arrays!



We recalculate the midpoint, and using integer division, $(5+8)/2$ will give 6 as the midpoint index to use. So now we will repeat the process.



For our third pass, we reset the HIGH pointer since our search value was lower than the value of the element at the midpoint. In the figure below, we can see that we reset the high pointer to point to one less than the previous mid pointer (since we already knew that the mid pointer did not point to our value). We leave the low pointer alone. Note that now, low and high both point to element 5, and so $(5+5)/2 = 5$, and now the mid pointer will point to 5 as well. So now we see if the element in the array that mid is pointing to contains the value that we are searching for. And it does! We have successfully searched for and found our value in three comparison steps.



Pseudocode:

```
# Initially called with low = 0 , high = N - 1
BinarySearch_Right(A[0..N-1], value, low, high) {
    value < A[i] for all i > high
    If high < low
        Return low
    mid = low + ((high - low) / 2)
    if (A[mid] > value )
        return BnarySearch_Right(A, value, low, mid-1)
    elsehht
        return BinarySearch_Right(A, value, mid+1, high)
}
```

Asymptotic Analysis

Since this algorithm halves the no of elements to be checked after every iteration it will take logarithmic time to find any element i.e. **$O(\log n)$** (where n is number of elements in the list) and its expected cost is also proportional to **$\log n$** provided that searching and comparing cost of all the elements is same.

Data structure used – Array

Worst case performance – $O(\log n)$

Best case performance – $O(1)$

Average case performance – $O(\log n)$

Worst case space complexity – $O(1)$

So the idea is-

1. Compare the key (element to be searched) with the mid element.
2. If key matches with middle element, we return the mid index.
3. Else If key is greater than the mid element, then key can only lie in right half subarray after the mid element. So we recur for right half.
4. Else (x is smaller) recur for the left half until there are no more elements left in the array.

Sorting

A sorting algorithm is an algorithm that puts elements of a list in a certain order. The most-used orders are numerical order and lexicographical order. Some popular sorting algorithms are -

Bubble Sort

The algorithm works by comparing each item in the list with the item next to it, and swapping them if required. In other words, the largest element has bubbled to the top of the array. The algorithm repeats this process until it makes a pass all the way through the list without swapping any items. Example -

Let us take the array of numbers "5 1 4 2 8", and sort the array from lowest number to greatest number using bubble sort. In each step, elements written in **bold** are being compared. Three passes will be required.

First Pass:

(**5** 1 4 2 8) → (**1** **5** 4 2 8), Here, algorithm compares the first two elements, and swaps since $5 > 1$.

(**1** **5** 4 2 8) → (**1** **4** **5** 2 8), Swap since $5 > 4$

(**1** **4** **5** 2 8) → (**1** **4** **2** **5** 8), Swap since $5 > 2$

(**1** **4** **2** **5** 8) → (**1** **4** **2** **5** 8), Now, since these elements are already in order ($8 > 5$), algorithm does not swap them.

Second Pass:

(**1** **4** **2** 5 8) → (**1** **4** **2** 5 8)

(**1** **4** **2** 5 8) → (**1** **2** **4** 5 8), Swap since $4 > 2$

(**1** **2** **4** 5 8) → (**1** **2** **4** 5 8)

(**1** **2** **4** 5 8) → (**1** **2** **4** 5 8)

Now, the array is already sorted, but the algorithm does not know if it is completed. The algorithm needs one **whole** pass without **any** swap to know it is sorted.

Third Pass:

(**1** **2** **4** 5 8) → (**1** **2** **4** 5 8)

(**1** **2** **4** 5 8) → (**1** **2** **4** 5 8)

(**1** **2** **4** 5 8) → (**1** **2** **4** 5 8)

(**1** **2** **4** 5 8) → (**1** **2** **4** 5 8)

```
procedure bubbleSort( A : list of sortable items )
  n = length(A)
  repeat
    swapped = false
    for i = 1 to n-1 inclusive do
      /* if this pair is out of order */
      if A[i-1] > A[i] then
        /* swap them and remember something changed */
        swap( A[i-1], A[i] )
        swapped = true
      end if
    end for
  until not swapped
end procedure
```

If the array is already sorted, then we need 1 iteration with $n - 1$ comparisons. In the worst case, we need total $n - 1$ iterations with total time complexity

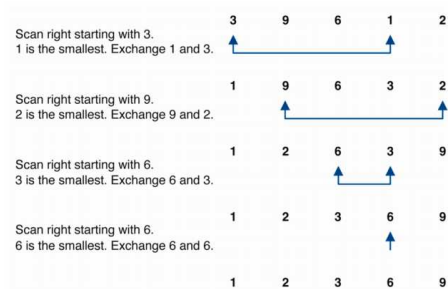
$$n - 1 + n - 2 + \dots + 1 = \frac{n(n-1)}{2}$$

Average-Case Time Complexity : A pair $(A[i], A[j])$ (resp. (i, j)) is inverted

if $i < j$ and $A[i] > A[j]$. Assuming our algorithm performs one swap for each inversion, the running time of your algorithm will depend on the number of inversions. Calculating the expected number of inversions in a uniform random permutation is easy. Let P be a permutation, and let $R(P)$ be the reverse of P . For example, if $P = 2, 1, 3, 4$ then $R(P) = 4, 3, 1, 2$. For each pair of indices (i, j) there is an inversion in exactly one of either P or $R(P)$. Since the total number of pairs is $\frac{n(n-1)}{2}$, and the total number and each pair is inverted in exactly half of the permutations, assuming all permutations are equally likely, the expected number of inversions is $\frac{n(n-1)}{4}$.

Selection Sort

The selection sort works as follows: you look through the entire array for the smallest element, once you find it you swap it (the smallest element) with the first element of the array. Then you look for the smallest element in the remaining array (an array without the first element) and swap it with the second element. Then you look for the smallest element in the remaining array (an array without first and second elements) and swap it with the third element, and so on.



```
# selection sort algorithm
def selection_sort(a):
    for k in range(0, len(a)-1):
        # getting the minimum element and its index from the sublist from k to len(a) where k runs from 0, 1, ..., len(a)-1
        j = 0
        b = a[k:len(a)]
        minimum = min(b)
        j = b.index(minimum)
        # swapping the minimum element with the first element of the sublist
        a[k], a[j+k] = a[j+k], a[k]
        print "after %s steps the array is : " % (k+1), a
    return a
```

Figure 1: Python code of Selection Sort

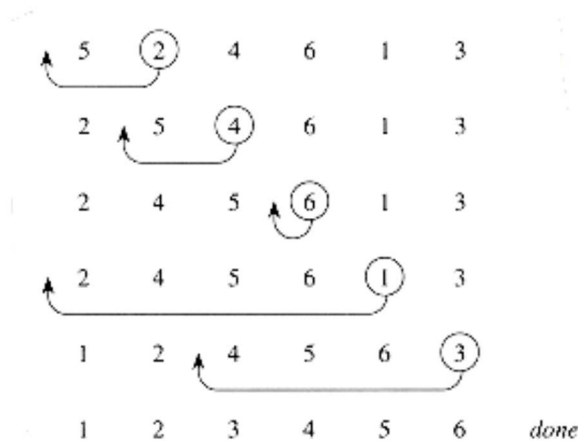
Even if the array is sorted, the algorithm finds the minimum element of the subarrays in i time for $i = n, n-1, \dots, 2$. So, the time complexity in the best case is

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

Similarly, in the worst case, finding the minimum and placing it in the proper position will take $O(n)$ time in each step. So, the whole algorithm will take $O(n^2)$ time.

Insertion Sort

Insertion sort iterates, consuming one input element each repetition, and growing a sorted output list. Each iteration, insertion sort removes one element from the input data, finds the location it belongs within the sorted list, and inserts it there. It repeats until no input elements remain. The pseudo code for insertion



sort is -

```

for i ← 1 to length(A) - 1
  j ← i
  while j > 0 and A[j-1] > A[j]
    swap A[j] and A[j-1]
    j ← j - 1
  end while
end for

```

If the array is already sorted, then this algorithm will never enter into the while loop of 3rd line. For the outer for loop it takes $O(n)$ time complexity.

For the worst case, when the array is completely in decreasing order, in each i th step, the i th element swaps with exactly i elements and each swap takes constant time. So, the total time complexity of this algorithm in worst case is -

$$\sum_{i=1}^{n-1} i = \frac{(n-1)n}{2} = O(n^2)$$

Similarly, it can also be proved using probability theory, that the average time complexity of insertion sort is $O(n^2)$.

Merge Sort

Merge-sort is based on the divide-and-conquer paradigm. It involves the following three steps:

- Divide the array into two subarrays
- Sort each subarray
- Merge them into one

Example. Consider the following array of numbers

```
27 10 12 25 34 16 15 31
divide it into two parts
27 10 12 25          34 16 15 31
divide each part into two parts
27 10          12 25          34 16          15 31
divide each part into two parts
27    10    12    25    34    16    15    31

merge (cleverly-!) parts
10 27    12 25    16 34    15 31
merge parts
10 12 25 27          15 16 31 34
merge parts into one
10 12 15 16 25 27 31 34
```

```
def merge_sort(a):
    # base case when the array is of size <= 1, then the array is already sorted
    if len(a) <= 1:
        return a
    # otherwise, use recursion to partition the list, then we merge two sorted sublists to get the original sorted list
    else:
        pivot = len(a)//2
        b = a[:pivot]
        c = a[pivot:len(a)]
        merge_sort(b)
        merge_sort(c)
        i = 0
        j = 0
        k = 0
        # we create another temporary array to concat the parts
        while (i < len(b)) & (j < len(c)):
            if b[i] < c[j]:
                a[k] = b[i]
                i += 1
            else:
                a[k] = c[j]
                j += 1
            k += 1
        while (i < len(b)):
            a[k] = b[i]
            i += 1
            k += 1
        while (j < len(c)):
            a[k] = c[j]
            j += 1
            k += 1
        print "The current merged array is : ",a
        return a
```

Figure 2: Python code

If the array contains n elements, then to completely break down the array into subarrays of size 1 will take $O(\log n)$ time, as there are exactly $\lfloor \log n \rfloor$ levels of divisions. In each level, if there are k many sorted subarrays of length j , then the total time taken in merge is $\approx kj/2 \in O(n)$. So, in each of the $\log n$ step, merge sort algorithm takes $O(n)$ time. Which gives us the time complexity of merge sort to be $O(n \log n)$.

This can also be shown using the recursion $T(n) = 2T(n/2) + O(n)$.

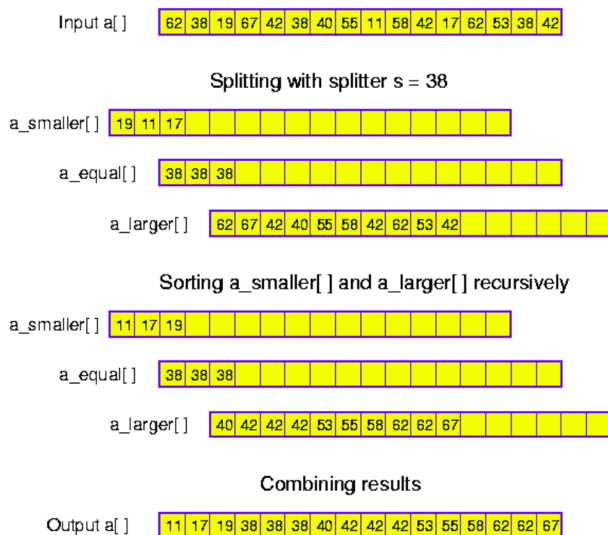
Quick Sort

Quicksort is a divide and conquer algorithm. Quicksort first divides a large array into two smaller sub-arrays: the low elements and the high elements. Quicksort can then recursively sort the sub-arrays.

The steps are:

- Pick an element, called a pivot, from the array.
- Reorder the array so that all elements with values less than the pivot come before the pivot, while all elements with values greater than the pivot come after it (equal values can go either way). After this partitioning, the pivot is in its final position. This is called the partition operation.
- Recursively apply the above steps to the sub-array of elements with smaller values and separately to the sub-array of elements with greater values.

QUICK SORT



```
def quick_sort(a):
    # base case: when length of the array is less than or equal 1, then it is already sorted. so, our algorithm should return the same
    if len(a) <= 1:
        return a

    # else, we choose the pivot to be the last element and do the partition
    else:
        # we use one counter to check the elements greater than the pivot element and swap them
        pivot = len(a)-1
        i = -1
        for j in range(0,pivot):
            if a[j] <= a[pivot]:
                i += 1
                a[i],a[j] = a[j],a[i]
        a[i+1],a[pivot] = a[pivot],a[i+1]
        pivot = i+1
        print "The pivot element is %s and the left and right arrays are : " % a[pivot],a[0:pivot],a[pivot+1:len(a)]
        # after partition, we apply recursion to compute the sorted array
        return quick_sort(a[0:pivot]) + [a[pivot]] + quick_sort(a[pivot+1:len(a)])
```

Figure 3: Python code of quicksort

If the array is in decreasing order and we choose our pivot to be the first element, then we will need total $O(n^2)$ time. (The recursion would be $T(n) = T(n-1) + n - 1$)

For the best and average case one can prove intuitively, that partitioning require $O(n)$ time and our recursion becomes

$$T(n) = 2T(n/2) + O(n) \tag{1}$$

This gives us the time complexity to be $O(n \log n)$.

Comparisons of algorithms

Comparisons of Sorting Algorithms						
Algorithm	Time (Best)	Time (Average)	Time (Worst)	Space (Worst)	In place	Stable
Bubble	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$	Yes	Yes
Selection	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	Yes	Yes
Insertion	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$	Yes	Yes
Merge	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(n)$	No	Yes
Quick	$O(n \log n)$	$O(n \log n)$	$O(n^2)$	$O(n)$	No	No

Some Notes

- A sorting algorithm is said to be **stable** if two elements with equal value appear in the same order in sorted output as they appear in the input unsorted array.
- An algorithm is **in place** if it uses negligible extra storage.
- Insertion sort is relatively efficient for small lists and mostly sorted lists. Shell sort is a variation which is more efficient for larger lists.
- Selection sort is noted for its simplicity, and also has performance advantages over more complicated algorithms in certain situations. It does no more than n swaps, and thus is useful where swapping is very expensive.
- Merge sort is very efficient for large lists and is used for the standard sort routine function in Python, Java.
- Typically, merge sort takes extra storage, but there are several in place merge sorting algorithms. But in this algorithm the worst case time complexity increases to $O(n^2)$, though it is faster than the classical selection sort or, insertion sort.
- The important caveat about quicksort is that its worst-case performance. But good choice of pivots yields $O(n \log n)$ performance, which is asymptotically optimal. For example, if at each step the median is chosen as the pivot then the algorithm works in $O(n \log n)$. Finding the median, such as by the median of medians selection algorithm is however an $O(n)$ operation on unsorted lists and therefore exacts significant overhead with sorting. In practice choosing a random pivot almost certainly yields $O(n \log n)$ performance. When implemented well, it can be about two or three times faster than its main competitors, merge sort and heapsort.
- Another, less common, not-in-place, version of quicksort uses $O(n)$ space for working storage and can implement a stable sort.

References

- [1] <http://www.softpanorama.org/Algorithms/searching.shtml>.
- [2] <http://www.csit.parkland.edu/~mbrandyberry/CS1Java/Lessons/Lesson27/BinarySearch.htm>.
- [3] <https://www.cs.cmu.edu/~adamchik/15-121/lectures/Sorting%20Algorithms/sorting.html>.
- [4] <https://en.wikipedia.org/>.
- [5] http://www.cs.odu.edu/~toida/nerzic/content/recursive_/alg/rec_/alg.html.
- [6] <http://www.personal.kent.edu/~rmuhamma/Algorithms/MyAlgorithms/intro.htm>.
- [7] <http://www.personal.kent.edu/~rmuhamma/Algorithms/MyAlgorithms/intro.htm>.