Python in Practice

Overview

* Introduction to Data Structure and Algorithm
* Data Structure
  * Searching and Sorting
  * List
  * Stack and Queue
  * Linked List
  * Dictionary
  * Binary Search Tree
  * Heap
* Algorithm
  * Divide and Conquer
  * Dynamic Programming
  * Greedy

Data Application Lab

What is Data Structure

* *Data structure*: a particular way of organizing data in a computer so that it can be used efficiently.
  * All about efficiency
  * More powerful computers ➔ More complex applications
* *Abstract Data Type (ADT)*: array, list, tree, table, etc.
  * a description of some type of data (or a collection of data) and the operations on that data
* ADTs have clean interfaces, and the implementation details are hidden.

Data Application Lab

What is Algorithm

* *Algorithm*: a step-by-step procedure for solving a problem in a FINITE amount of time.
* Efficiency
  * Space
  * Time
* Objective
  * Be able to identify the functionality required of the program in order to solve the task at hand
  * Design data structures and algorithms which express this functionality in an efficient way
  * Be able to evaluate a given implementation in terms of its efficiency and correctness

Data Application Lab

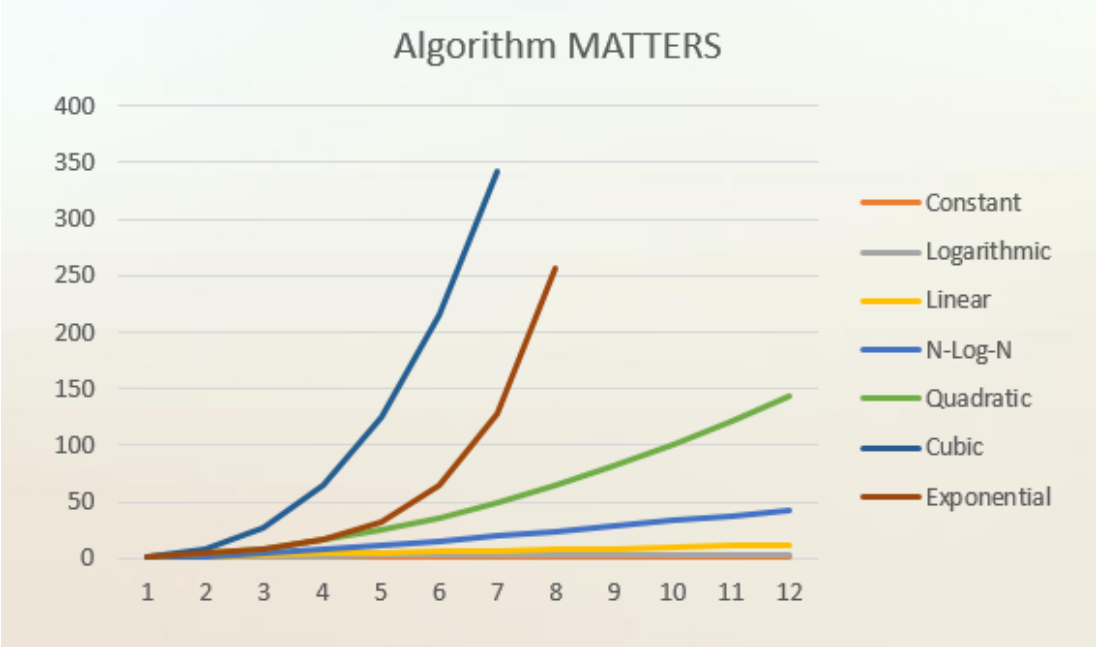
A Simple Example

* *Find Missing Number*
  * You are given a list of n-1 integers and these integers are in the range of 1 to n. There are no duplicates in list. One of the integers is missing in the list. Write an efficient code to find the missing integer. XOR
* *Find Top k Most Searched Keywords from Amazon*
  * *Amazon! How many records?*
  * *Boss wants a quick answer*
  * *How about in real time? Like every 5 mins*

Data Application Lab

Algorithm Analysis

* Running Time (Time Complexity)
  * The running time typically grows with the input size
  * Best Case: Lower bound on cost
    * Determine by “easiest” input
    * Provides a goal for all inputs
  * Worse Case: Upper bound on cost
    * Determine by “most difficult” input
    * Provides a guarantee for all inputs
  * Average case: Expected cost for random input
    * Need a model for “random” input
    * Provides a way to predict performance
  * Average case time is often difficult to determine
  * We focus on worst case running time



Theoretical Analysis

* Uses a high-level description of the algorithm instead of an implementation
* Characterizes running time as a function of the input size, n.
* Takes into account all possible inputs
* Allows us to evaluate the speed of an algorithm independent of the hardware/software environment
* Pseudocode
  * Find Max Element of an Array

Algorithm *arrayMax*(*A*, *n*)	# operations
currentMax ← *A*[0]	2
for *i* ← 1 **to** *n* − 1 **do**	2*n*
if *A*[*i*] > *currentMax* **then**	2(*n* − 1)
currentMax ← *A*[*i*]	2(*n* − 1)
{ increment counter *i* }	2(*n* − 1)
return *currentMax*	1
Total	8*n* − 3

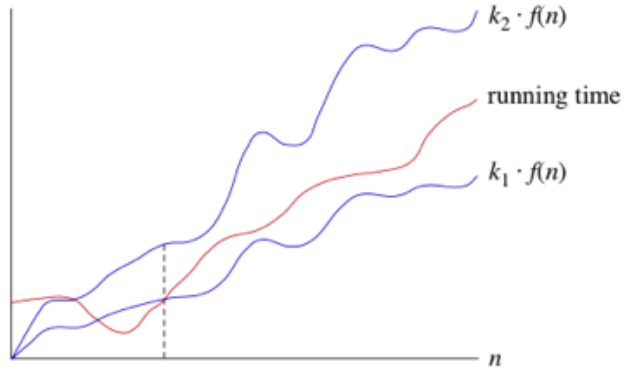
* Primitive Operations
  * Evaluating an expression
  * Assigning a value to a variable
  * Indexing into an array
  * Calling a method
  * Returning from a method
* By inspecting the pseudocode, we can determine the maximum number of primitive operations executed by an algorithm, as a function of the input size

Estimating Running Time

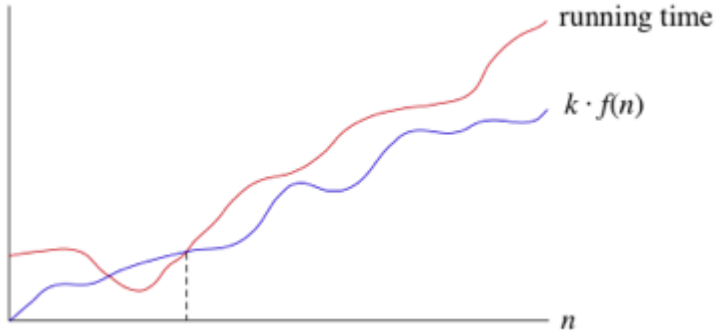
* Algorithm *arrayMax* executes $8n - 3$ primitive operations in the worst case. Define:
  * a = Time taken by the fastest primitive operation
  * b = Time taken by the slowest primitive operation
* Let $T(n)$ be worst-case time of arrayMax. Then
  * $a(8n - 3) \leq T(n) \leq b(8n - 3)$
* Hence, the running time $T(n)$ is bounded by two linear functions

Asymptotic Notation

* **Big-θ (Big-Theta) notation**
  * When we say that a particular running time is Θ(*n*), we're saying that once *n* gets large enough, the running time is at least $k_1 \cdot n$ and at most $k_2 \cdot n$ for some constants k_1 and k_2.

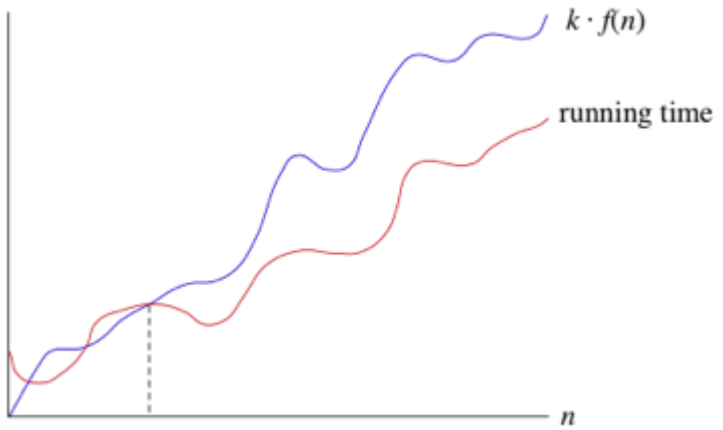


* **Big-Ω (Big-Omega) notation**
  * Sometimes, we want to say that an algorithm takes at least a certain amount of time, without providing an upper bound. We use big-Ω notation
  * If a running time is Ω(f(n)), then for large enough n, the running time is at least k·f(n) for some constant k.



Big Oh Notation

* **Big-O notation**
* We use $\Theta(n)$ notation to asymptotically bound the growth of a running time to within constant factors above and below. Sometimes we want to bound from only above.
* Although the worst-case running time of binary search is $\Theta(lgn)$, it would be incorrect to say that binary search runs in $\Theta(lgn)$ time in all cases.
* The running time of binary search is never worse than $\Theta(lgn)$, but it's sometimes better.



Notation Summary

notation	provides	example	shorthand for	used to
Big Theta	asymptotic order of growth	$\Theta(N^2)$	$\frac{1}{2} N^2$ $10 N^2$ $5 N^2 + 22 N \log N + 3N$ ⋮	classify algorithms
Big Oh	$\Theta(N^2)$ and smaller	$O(N^2)$	$10 N^2$ $100 N$ $22 N \log N + 3 N$ ⋮	develop upper bounds
Big Omega	$\Theta(N^2)$ and larger	$\Omega(N^2)$	$\frac{1}{2} N^2$ N^5 $N^3 + 22 N \log N + 3 N$	develop lower bounds

Master Theorem

The Master Theorem applies to recurrences of the following form:

$$T(n) = aT(n/b) + f(n)$$

where $a \geq 1$ and $b > 1$ are constants and $f(n)$ is an asymptotically positive function.

There are 3 cases:

1. If $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$.
2. If $f(n) = \Theta(n^{\log_b a} \log^k n)$ with[1] $k \geq 0$, then $T(n) = \Theta(n^{\log_b a} \log^{k+1} n)$.
3. If $f(n) = \Omega(n^{\log_b a + \epsilon})$ with $\epsilon > 0$, and $f(n)$ satisfies the regularity condition, then $T(n) = \Theta(f(n))$.
   Regularity condition: $af(n/b) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n.

$T(n) = 3T(n/2) + n^2 \Longrightarrow T(n) = \Theta(n^2)$ (Case 3)

$T(n) = 4T(n/2) + n^2 \Longrightarrow T(n) = \Theta(n^2 \log n)$ (Case 2)

$T(n) = T(n/2) + 2^n \Longrightarrow \Theta(2^n)$ (Case 3)

$T(n) = 16T(n/4) + n \Longrightarrow T(n) = \Theta(n^2)$ (Case 1)

$T(n) = 2T(n/2) + n \log n \Longrightarrow T(n) = n \log^2 n$ (Case 2)

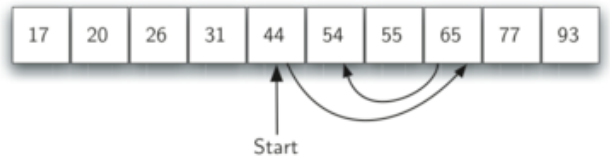
https://math.dartmouth.edu/archive/m19w03/public_html/Section5-2.pdf
http://www.cs.cornell.edu/courses/cs3110/2011sp/lectures/lec19-master/mm-proof.pdf

Search

* Sequential Search

Case	Best Case	Worst Case	Average Case
item is present	1	n	$\frac{n}{2}$
item is not present	n	n	n

* Binary Search



Comparisons	Approximate Number of Items Left
1	$\frac{n}{2}$
2	$\frac{n}{4}$
3	$\frac{n}{8}$
...	
i	$\frac{n}{2^i}$

Data Application Lab

Sort Overview

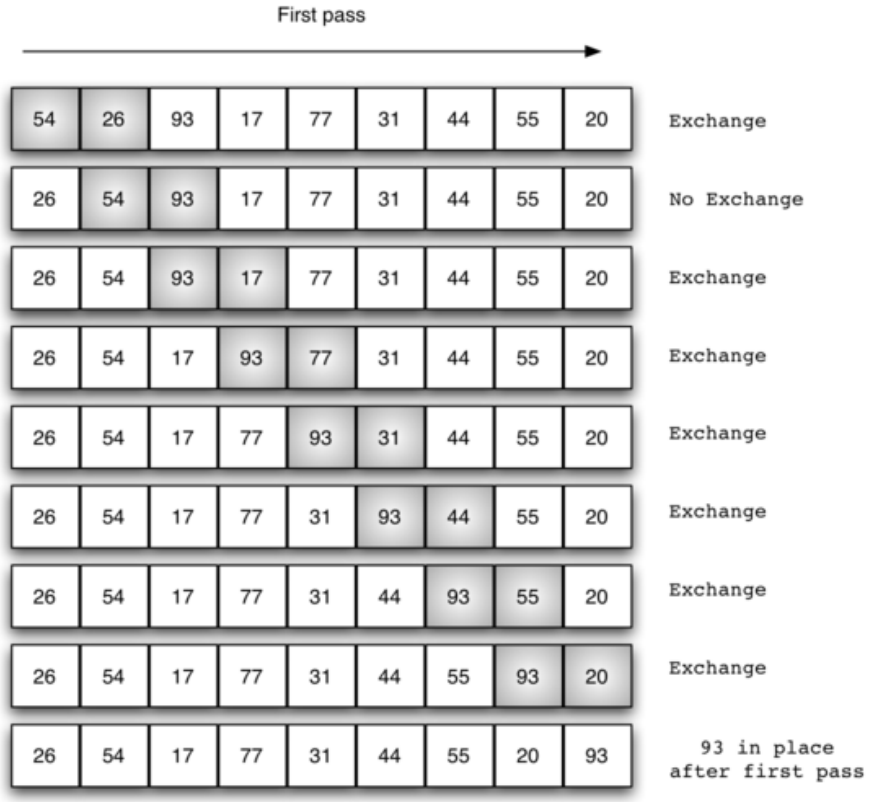
* Bubble Sort
* Selection Sort
* Insertion Sort
* Count Sort
* Merge Sort
* Quick Sort
* Heap Sort / BST Sort
* Python Lib
* Applications

Data Application Lab

http://www.sorting-algorithms.com/

Bubble Sort

* Repeatedly steps through the list to be sorted, compares each pair of adjacent items and swaps them if they are in the wrong order
* On each pass, the largest element that is left unsorted, has been "bubbled" to its rightful place at the end of the array
* The pass through the list is repeated until no swaps are needed, which indicates that the list is sorted
* Properties:
  * Stable
  * $O(1)$ extra space
  * $O(n^2)$ comparisons and swaps
  * Adaptive: $O(n)$ when nearly sorted



Data Application Lab

Bubble Sort

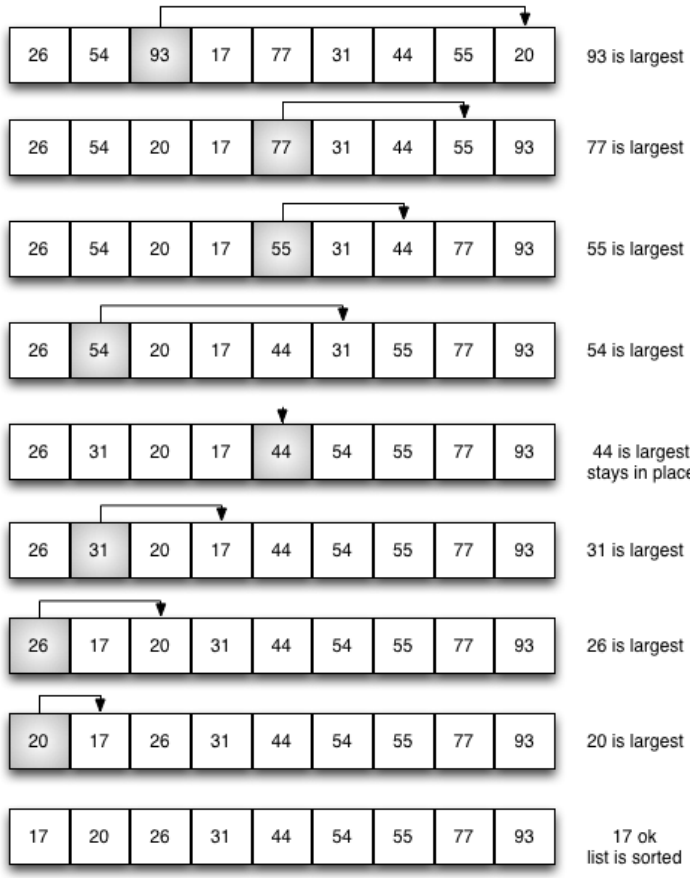
```
def sort(self, array):
    for i in range(len(array)):
        is_sorted = True
        for j in range(1, len(array) - i):
            if (array[j] < array[j-1]):
                # swap
                array[j], array[j-1] = array[j-1], array[j]
                is_sorted = False

        if (is_sorted): break
```

Sort	Best	Average	Worst	Memory	Stable*	Method
Bubble	n	n^2	n^2	1	Yes	Exchanging

Data Application Lab

Selection Sort

* 2 linear passes on the list
* On each pass, it selects the smallest value
* Swaps it with the last unsorted element
* Properties
  * Not Stable
  * $O(1)$ extra space
  * $O(n^2)$ comparisons
  * $O(n)$ swaps
  * Not Adaptive



Data Application Lab

Selection Sort

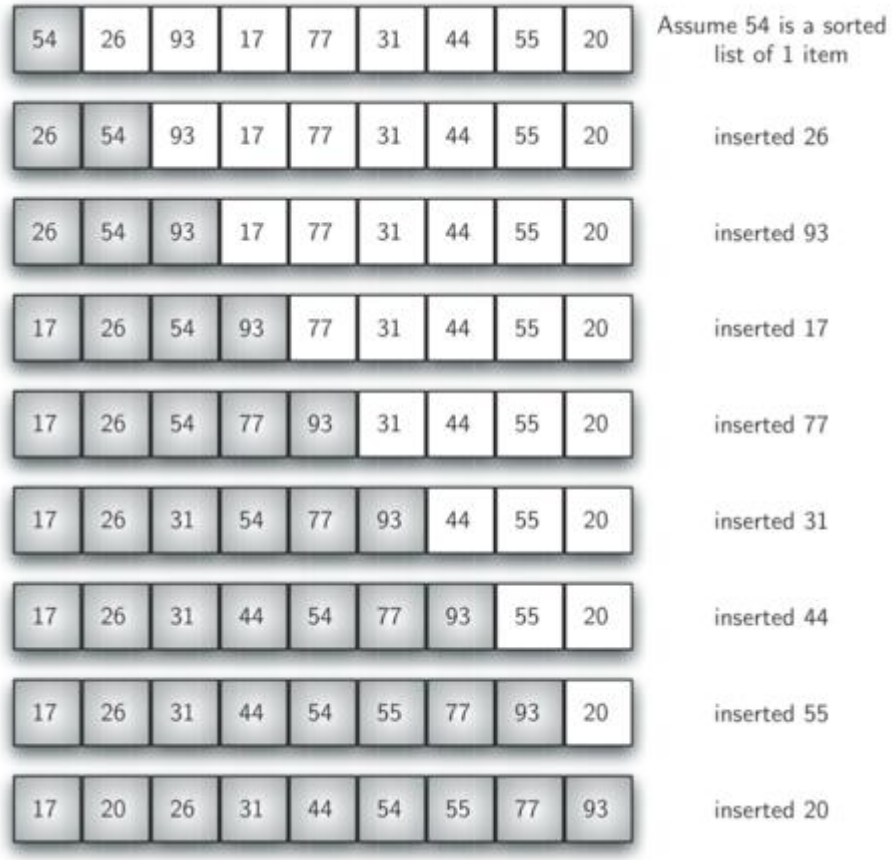
```
def sort(self, array):
    for i in range(len(array)):
        pos_min = i
        for j in range(i+1, len(array)):
            if (array[j] < array[pos_min]):
                pos_min = j

        array[i], array[pos_min] = array[pos_min], array[i]
```

Sort	Best	Average	Worst	Memory	Stable*	Method
Selection	n^2	n^2	n^2	1	No	Selection

Data Application Lab

Insertion Sort

- On each pass the current item is inserted into the sorted section of the list
- It starts with the last position of the sorted list, and moves backwards until it finds the proper place of the current item
- That item is then inserted into that place, and all items after that are shuffled to the left to accommodate it
- Binary Insertion Sort
  - instead of doing a linear search each time for the correct position, it does a binary search, which is $O(\log n)$ instead of $O(n)$
  - This brings the cost of the best cast up to $O(n \log n)$
  - The only problem is that it always has to do a binary search even if the item is in its current position
  - Due to the possibility of having to shuffle all other elements down the list on each pass, the worst case running time remains at $O(n^2)$



Insertion Sort

```
def sort(self, array):
    for i in range(1, len(array)):
        for j in range(0, i):
            if (array[i] < array[j]):
                array[i], array[j] = array[j], array[i]
```

Sort	Best	Average	Worst	Memory	Stable*	Method
Insertion	n	n^2	n^2	1	Yes	Insertion
Binary Insertion	$n \log n$	n^2	n^2	1	Yes	Insertion

Count Sort

* The input to counting sort consists of a collection of n items, each of which has a non-negative integer key whose maximum value is at most k

```
def sort(self, array):
    mmax, mmin = array[0], array[0]
    for i in range(1, len(array)):
        if (array[i] > mmax): mmax = array[i]
        elif (array[i] < mmin): mmin = array[i];

    nums = mmax - mmin + 1
    counts = [0] * nums
    for i in range (len(array)):
        counts[array[i] - mmin] = counts[array[i] - mmin] + 1

    pos = 0
    for i in range(nums):
        for j in range(counts[i]):
            array[pos] = i + mmin
            pos += 1
```

Sort	Best	Average	Worst	Memory	Stable*	Method
Count	n	n	n	k	No	Counting

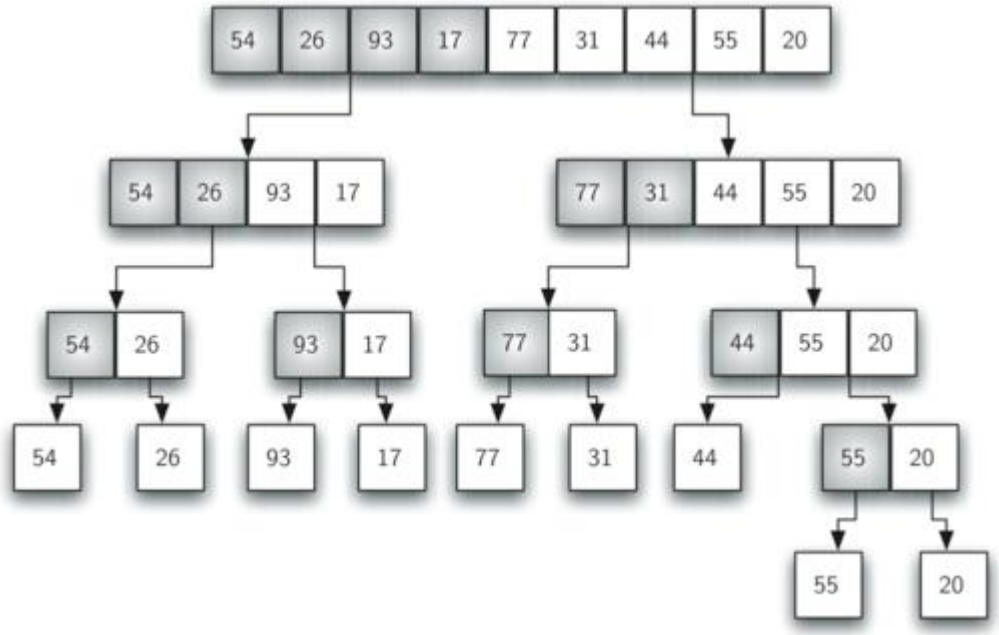
Data Application Lab

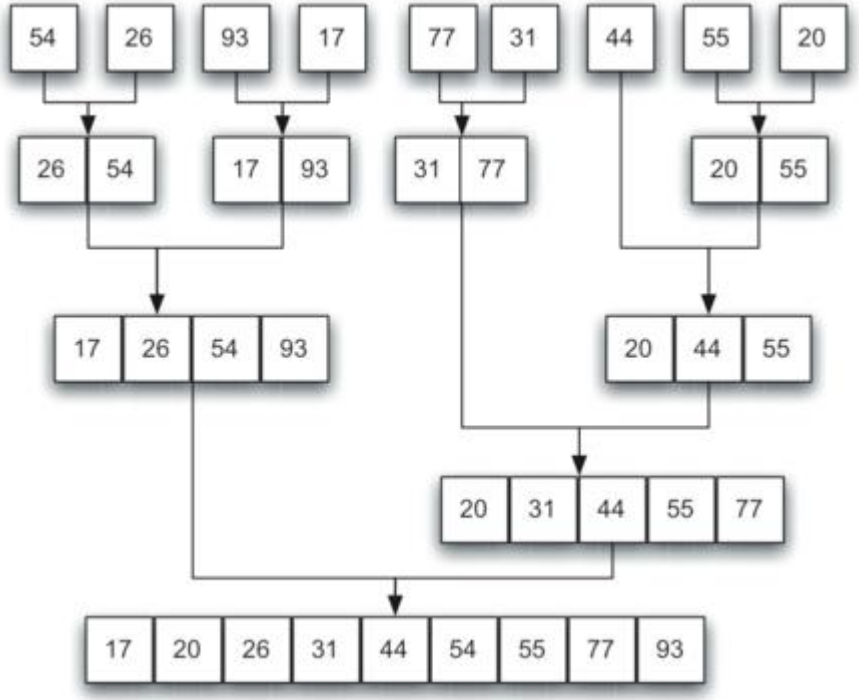
Merge Sort

* Divide and Conquer
* Divide
  * Recursively splits up the array until it is fragmented into pairs of two single element arrays
  * Each of those single elements is then merged with its pairs, and then those pairs are merged with their pairs and so on, until the entire list is united in sorted order
* Conquer
  * It is simple to take 2 sorted lists, and combine into another sorted list,
  * simply by going through, comparing the heads of each list, removing the smallest to join the new sorted list
  * $O(n)$ operation

Data Application Lab

Merge Sort





Data Application Lab

Divide and Conquer

* Mathematical analysis. In the worst case, merge sort makes ~ $N \lg N$ compares and the running time is linearithmic.
* Quadratic-linearithmic chasm. The difference between N^2 and $N \lg N$ makes a huge difference in practical applications.
* Divide-and-conquer algorithms. The same basic approach is effective for many important problems
* Reduction to sorting. A problem A reduces to a problem B if we can use a solution to B to solve A.
  * For example, consider the problem of determining whether the elements in an array are all different.
  * This problem reduces to sorting because we can sort the array, the make a linear pass through the sorted array to check whether any entry is equal to the next (if not, the elements are all different.)

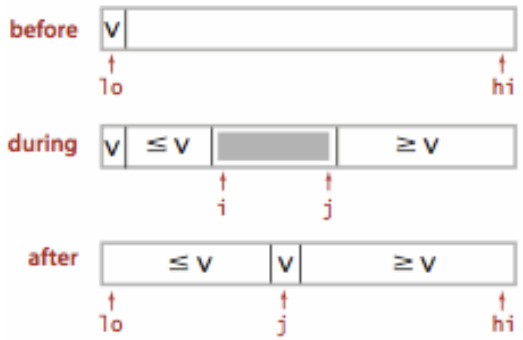
Sort	Best	Average	Worst	Memory	Stable*	Method
Merge	$n \log n$	$n \log n$	$n \log n$	Depends	Yes	Merging

Improvements

* Use insertion sort for small subarrays
  * We can improve most recursive algorithms by handling small cases differently
  * Improve the running time of a typical merge sort implementation by 10 to 15 percent
* Test whether array is already in order
  * We can reduce the running time to be linear for arrays that are already in order by adding a test to skip call to merge() if a[mid] is less than or equal to a[mid+1]
  * With this change, we still do all the recursive calls, but the running time for any sorted subarray is linear.

Quick Sort

* Another Divide-and-Conquer
* Partitioning an array into two parts, then sorting the parts independently
  * Firstly a pivot is selected, and removed from the list (hidden at the end)
  * Then the elements are partitioned into 2 sections. One which is less than the pivot and one that is greater. This partitioning is achieved by exchanging values
  * Then the pivot is restored in the middle, and those 2 sections are recursively quick sorted

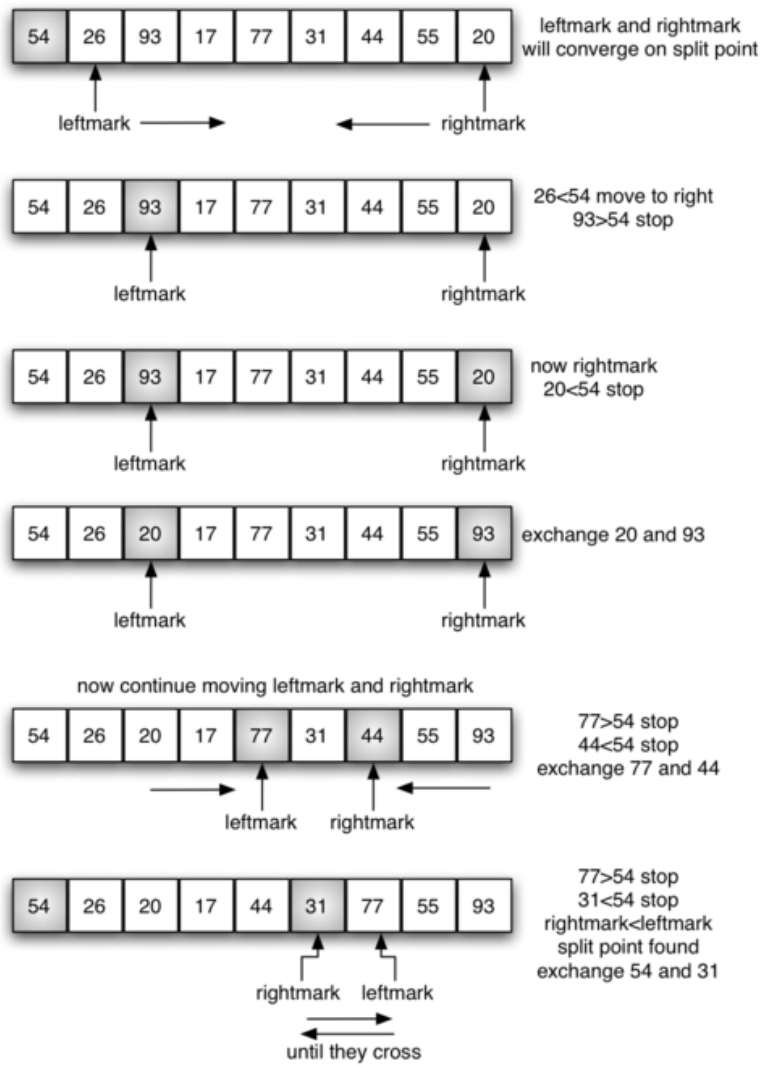


Quicksort partitioning overview

Data Application Lab

Quick Sort

```
def sort(self, alist):
    self.helper(alist, 0, len(alist) - 1)

def helper(self, alist, first, last):
    if (first < last):
        splitpoint = self.partition(alist, first, last)

        self.helper(alist, first, splitpoint - 1)
        self.helper(alist, splitpoint + 1, last)


def partition(self, alist, first, last):
    pivotvalue = alist[first]
    leftmark = first + 1
    rightmark = last

    done = False
    while not done:
        while leftmark <= rightmark and alist[leftmark] <= pivotvalue:
            leftmark = leftmark + 1

        while alist[rightmark] >= pivotvalue and rightmark >= leftmark:
            rightmark = rightmark - 1

        if rightmark < leftmark:
            done = True
        else:
            alist[leftmark], alist[rightmark] = alist[rightmark], alist[leftmark]

    alist[first], alist[rightmark] = alist[rightmark], alist[first]

    return rightmark
```



Data App

Improvements

* Cutoff to insertion sort
  * As with merge sort, it pays to switch to insertion sort for tiny arrays
  * The optimum value of the cutoff is system-dependent
  * but any value between 5 and 15 is likely to work well in most situations.
* Median-of-three partitioning
  * A second easy way to improve the performance of quicksort is to use the median of a small sample of items taken from the array as the partitioning item
  * Doing so will give a slightly better partition, but at the cost of computing the median
  * It turns out that most of the available improvement comes from choosing a sample of size 3 (and then partitioning on the middle item)

Other Sorting

* Other Sorting Algorithm
  * Binary Search Tree
  * Heapsort
  * Pancake Sorting
    * Sorting a disordered stack of pancakes in order of size when a spatula can be inserted at any point in the stack and used to flip all pancakes above it
    * In 1979, Bill Gates gave an upper bound of 5/3n
    * This was improved, thirty years later, to 18/11n by a team of researchers at the University of Texas at Dallas, led by Founders Professor Hal Sudborough
* Applications
  * Sort a customize object: Transaction

Python Lib

* Sorting Basic
  * sorted()
  * list.sort()
* Key Functions
  * The value of the key parameter should be a function that takes a single argument and returns a key to use for sorting purposes
  * Lamda
  * Named Attribute
* Operator Module Functions
  * Itemgetter, Attrgetter, Methodcaller
* Old Way Using the cmp Parameter
* Timsort
  * Hybrid Sort (insertion + merge)
  * Stable

Data Application Lab

Summary

algorithm	stable?	inplace?	growth rate to sort N items		notes
			running time	extra space	
selection sort	no	yes	N^2	1	
insertion sort	yes	yes	between N and N^2	1	depends on order of input keys
shellsort	no	yes	$N^{6/5}$?	1	
quicksort	no	yes	$N \lg N$	$\lg N$	probabilistic guarantees, depend on distribution of input key values
3-way quicksort	no	yes	between N and $N \lg N$	$\lg N$	
mergesort	yes	no	$N \lg N$	N	
heapsort	no	yes	$N \lg N$	1	

Performance characteristics of sorting algorithms

Applications

* Two Pointers
* Binary Search
* Others

Data Application Lab

Two Pointers

* Two Sum (1E)
* Merge Two Sorted Array into One (88E)
  * You are given two sorted arrays, A and B, and A has a large enough buffer at the end to hold B. Write a method to merge B into A in sorted order Others
* Minimum Difference Between Two Sorted Arrays
  * Given two arrays sorted in ascending order, find the absolute minimum difference between any pair of elements |a-b| such that a is from one array and b is from another array
* Valid Palindrome (125E)
  * Given a string, determine if it is a palindrome, considering only alphanumeric characters and ignoring cases.
* Find Min in Rotated Sorted Array (153M)
  * Suppose an array sorted in ascending order is rotated at some pivot unknown to you beforehand. Find the minimum element.

Data Application Lab

Binary Search

* Recover Rotated Sorted Array
  * [4, 5, 1, 2, 3] ➔ [1, 2, 3, 4, 5]
* Search Insert Position (35M)
  * Given a sorted array and a target value, return the index if the target is found. If not, return the index where it would be if it were inserted in order. You may assume no duplicates in the array.
* Search Peak Element (162M)
  * The array has no duplicates, may contain multiple peaks, return the index to *any* one of the peaks.
  * You may imagine that num[-1] = num[n] = -∞.

Data Application Lab

Others

* Find in Row/Column Sorted Matrix (74M) (240M)
  * Given a matrix in which each row and each column is sorted, write a method to find an element in it
* Valid Anagram (242E)
  * Given two strings s and t, write a function to determine if t is an anagram of s.

Python in Practice
The End