SUMMARY

- One of the top algorithms used in practice
- O(n log n) runtime "on average"
- Works in place (minimal extra memory needed, unlike merge sort needing an extra array to store sorted elements.)

HIGH LEVEL WALKTHROUGH

GIVEN: An array of unsorted numbers

3	8	2	5	1	4	7	6

GOAL: Sort given array in ascending order

1	2	3	4	5	6	7	8

ASSUME: All entry arrays distinct, no repeated values.

1) Select an element in the array to be a pivot element. There is some thought into selecting an appropriate pivot element, but we will select the first value in the array for simplicity.

   PIVOT ELEMENT = 3

2) Rearrange the array so that all elements:
   a) left of the pivot are less than the pivot element
   b) right of the pivot are greater than the pivot element

3	8	2	5	1	4	7	6

PIVOT

$\longrightarrow$

2	1	3	7	4	5	8	6

$\underbrace{2\ 1}_{<\text{PIVOT}}\ 3\ \underbrace{7\ 4\ 5\ 8\ 6}_{>\text{PIVOT}}$

Notice the values are partitioned onto appropriate sides of the pivot element, but are not yet sorted. Also, the pivot element is in its correct position in the array.

3) For each partitioned sub-array (< pivot elements and > pivot elements) perform steps 1 and 2 until all elements are in position.

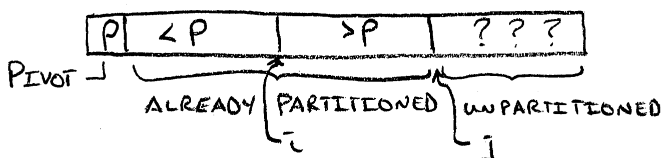
   The base case is that a single element is by default sorted since it cannot be greater than or less than itself.

Partitioning Around a Pivot (in-place implementation)

Happens in O(n), linear, runtime.

For simplicity, assume the pivot element is the first element in the array. If the pivot element is not the first element just swap the pivot with the first element and you start with the original assumption.

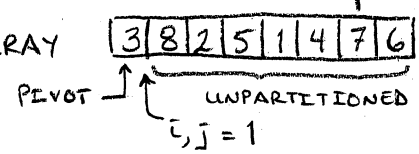
View of arbitrary array after arbitrary number of comparisons



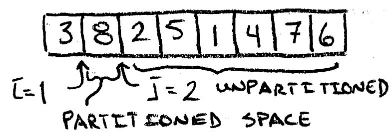
Let i track the index of the boundary between elements less than the pivot value and elements greater than the pivot value

Let j track the index of the boundary between elements that have already been scanned and compared to the pivot element and those elements that have not yet been placed in their appropriate subarray of greater than the pivot element's value or less than the pivot element's value.

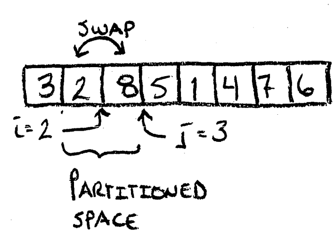
INITIAL ARRAY



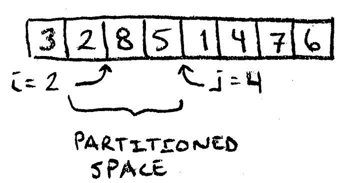
advance j



8 > 3, leave in place
advance j



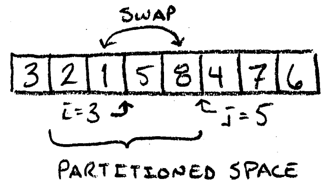
2 < 3, swap 2 and 8
advance i
advance j



5 > 3, leave in place
advance j

(con't on next sheet)

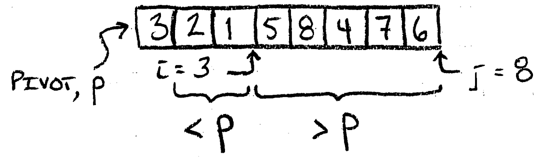
PARTITIONING AROUND A PIVOT (con't)



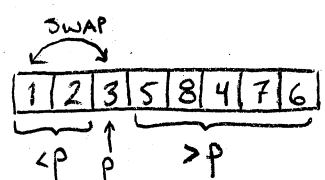
1 < 3, swap 1 and 8
advance i
advance j

Since the last three elements are greater than the pivot advance j to the end of the array as there are no swaps.

ALL ELEMENTS IN APPROPRIATE PARTITION SPACE EXCEPT PIVOT



swap the rightmost element of the partition space < p with the pivot element.



Partitioning is complete

PSEUDOCODE TO IMPLEMENT PARTITION

```
Partition(Array, A; Leftmost index, l; rightmost index, r)
  p = A[l]
  i = l + 1
  for j = i to r
    if A[j] < p
      swap A[j] and A[i]
      i++
  swap A[l] and A[i-1]
```

CHOOSING A GOOD PIVOT

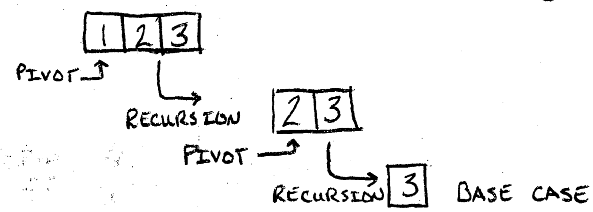
Choosing a good pivot affects the runtime of quicksort. This is why quicksort runs in $O(n \log n)$ on average.

CASE: ALWAYS SELECTING FIRST ELEMENT IN GIVEN ARRAY

PRO: Simple to implement

CON: Leads to $O(n^2)$ runtime if the given array is already sorted.

For each recursion all elements other than the pivot, $n-1$ elements, will continue to recurse until the n^{th} element is left while the array is still sorted. So instead of saving on recursions by dividing the input array up, there are n recursions that touch all n elements leading to $O(n^2)$ runtime



n	$n+(n-1)+1$ OPERATIONS
$n-1$	$2n$ OPS (n RECURSIONS)
1	$2n^2 \rightarrow O(n^2)$

CASE: PIVOT SPLITS GIVEN ARRAY 50/50

PRO: Best case scenario for quicksort $O(n \lg n)$

CON: Probability of selecting perfect median out of a large input array is very low.

USING MASTER METHOD (equal subproblems)

Let $T(n)$ be the running time on an array of size n

$$\underbrace{2}_{a\text{ = NUMBER OF RECURSIONS GENERATED}} \quad T(n) \leq 2T(n/\underbrace{2}_{b\text{ = DIVIDES ARRAY OF SIZE } n \text{ IN } 2}) + \Theta(n)\ \} \text{ CHOOSE PIVOT AND PARTITION IS LINEAR, } d=1$$

$$a = 2,\ b^d = 2 \rightarrow a = b^d \rightarrow T(n) = O(n^d \lg n) \rightarrow$$

$$T(n) = O(n^1 \lg n)$$

(con't on next sheet)

3-0235 — 50 SHEETS — 5 SQUARES
3-0236 — 100 SHEETS — 5 SQUARES
3-0237 — 200 SHEETS — 5 SQUARES
3-0137 — 200 SHEETS — FILLER
COMET

CHOOSING A GOOD PIVOT (con't)

CASE: SELECT RANDOM PIVOTS

That is, in every recursive call, choose the pivot randomly (each element is equally likely to be selected as the pivot element, aka, uniform distribution.) There are other ways of randomly selecting a pivot, but selecting out of a uniform distribution leads to satisfactory results without being overly complex.

PRO: On average leads to $O(n \lg n)$ runtime
Not reliant on arrangement of values within input data

HIGH LEVEL ANALYSIS

A 25%-75% split in the input data gets us close enough to $O(n \lg n)$ runtime

Since an average is about 50% of the data, with a uniform distribution used to select a pivot element 50% of the input array elements are likely to be chosen as the pivot element as those potential pivot elements reside within $0.25n$ and $0.75n$.

This can be proven in low level detail. I do not believe it is necessary.

Python Implementation of Quicksort

```
# quicksort_example.py
# A program to show how to implement quicksort in an
#  average of O(n lg n) runtime.

from random import randint

def main():
  target_list = [0, 7, 1, 2, 5, 8, 6, 3, 9, 4]

  print("The initial unsorted list is:", target_list)

  target_list = quicksort(target_list, 0, len(target_list)-1)

  print("The sorted list is:", target_list)

def quicksort(input_list, left_index, right_index):
  # Base case: A difference of 1 or 0 between the left and
  # right index means the element is sorted.
  if right_index - left_index < 1:
    return input_list

  pivot_index = randint(left_index, right_index)
  pivot = input_list[pivot_index]

  # Swap the pivot element with the element in the leftmost
  # index position.
  input_list = swap(input_list, left_index, pivot_index)

  # Set pointers for partitions
  # i is the pointer for the index where all elements in positions
  # less than index i are less than the pivot.
  # j is the pointer to the index where all elements in positions
  # greater than index j have not yet been compared to the pivot.
  i = j = left_index + 1 # The pivot element is in the first position.

  while j <= right_index:
    if input_list[j] < pivot
      input_list = swap(input_list, i, j)
      i += 1
    j += 1
```

(con't on next sheet)

PYTHON IMPLEMENTATION OF QUICKSORT (cont)

```
    #Swap the pivot element into its rightful position
    input_list = swap(input_list, i-1, left_index)

    # Sort the elements less than the pivot.
    input_list = quicksort(input_list, left_index, i-2)
    # Sort the elements greater than the pivot.
    input_list = quicksort(input_list, i , len(input_list)-1)

    return input_list

def swap(L, i, j):
    # Swap the elements at i and j in the list, L.
    tmp = L[i]
    L[i] = L[j]
    L[j] = tmp

    return L

if __name__ == "__main__":
    main()
```

Go Implementation of Quicksort

```
package main

// A program to show how to implement quicksort in an
// average of O(n lg n) runtime.

import (
        "fmt"
        "math/rand"
        "time"
)

func main() {
  targetSlice := []int {0, 7, 1, 2, 5, 8, 6, 3, 9, 4}

  fmt.Println("The initial unsorted slice is: \t", targetSlice)

  targetSlice = quicksort(targetSlice, 0, len(targetSlice)-1)

  fmt.Println("The quicksorted slice is: \t", targetSlice)
}

func quicksort( xint []int, leftIndex, rightIndex int) []int {
  // Base case: A difference of 0 or 1 between the left and
  // right index means the element is sorted.
  if rightIndex - leftIndex < 1 {
    return xint
  }

  r := rand.New(rand.NewSource(time.Now().UnixNano()))

  pivotIndex := rightIndex - r.Intn( rightIndex - leftIndex + 1)
  pivot := xint[pivotIndex]

  // Swap the pivot element with the element in the leftmost
  // index position
  xint = swap(xint, leftIndex, pivotIndex)

```

(con't on next sheet)

Go Implementation of Quicksort (cont)

```
/* Set pointers for partitions
   i is the pointer for the index where all elements in positions
   less than index i are less than the pivot.
   j is the pointer to the index where all elements in positions
   greater than index j have not yet been compared to the pivot.
*/

i := leftIndex +1 // The pivot element is in the first position
j := i

for j <= rightIndex {
  if xint[j] < pivot {
    xint = swap(xint, i, j)
    i++
  }
  j++
}

// Swap the pivot element into its rightful position
xint = swap(xint, i-1, leftIndex)

// Sort the elements less than the pivot
xint = quicksort(xint, leftIndex, i-2)
// Sort the elements greater than the pivot
xint = quicksort(xint, i, len(xint)-1)

return xint
}

func swap(xint []int, i, j int) []int {
  // Swap the elements at i and j in the slice, xint.
  tmp := xint[i]
  xint[i] = xint[j]
  xint[j] = tmp

  return xint
}
```