Quicksort summary

# Overview of Quicksort

The Quicksort algorithm was developed by Tony Hoare in 1959. Quicksort is very popular and can be much faster than Merge sort and Heapsort.

Quicksort requires little additional memory when sorting an array, it is thus considered similar to selection sort.

On average, Quicksort takes O(n log n) comparisons to sort n items. Worst case performance is O(n2), but this is rare.

# Run-through of algorithm

Consider an array of 10 integers that are not ordered. The first thing that needs to be done is choose a pivot. The pivot is a number that will satisfy the following rule. All values lower than the pivot will be put at the beginning of the array and all values high at the end.

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 7 | 4 | 2 | 5 | 9 | 0 | 1 | 3 | 6 | 8 |

There are ways to choose a pivot optimally, but that will be discussed later. For this example the pivot is 7. Next we place the pivot at either the beginning or end of the array and mark its index. In this case pivot remains at index 0.

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 7 | 4 | 2 | 5 | 9 | 0 | 1 | 3 | 6 | 8 |

^ ^

Low High

In the next step we set an iterator that will go through the array from the left. It will stop when it finds an integer value that is greater than our pivot.

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 7 | 4 | 2 | 5 | 9 | 0 | 1 | 3 | 6 | 8 |

^ ^

Low High

In this case that is the value 9 at index 4. Then another iterator is set to through the array from the right. It will stop when it finds a value that is smaller than the pivot. In this case the integer 6 at index 8.

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 7 | 4 | 2 | 5 | 9 | 0 | 1 | 3 | 6 | 8 |

^ ^

Low High

In the next step we swap these items. We have thus fulfilled a part of the pivot rule. The indices of iterators ‘Low’ and ‘High’ are retained. The comparison operation continues until index for ‘Low’ is greater than ‘High’.

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 7 | 4 | 2 | 5 | 6 | 0 | 1 | 3 | 9 | 8 |

^ ^

High Low

The array is not partly sorted. Quicksort is recursive and we will now redo the entire operation again. This time the index of ‘Low’ is set to 1 and index for High is set ‘High - 1’. For the values larger than the original pivot index ‘Low’ is set to ‘High’ and index ‘High is set to 9.

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 7 | 4 | 2 | 5 | 6 | 0 | 1 | 3 | 9 | 8 |

^ ^ ^ ^

Low High Low High

As you can see the array is divided into two after each run of the algorithm. Each time the array becomes more sorted. The base case is reached when a sub-array is of size 2 – a pivot and an index. We place the pivot at the beginning. If ‘Low’ value is smaller than the pivot but the ‘High’ value is at the same index than we swap ‘Low’ for the pivot value.

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 7 | 4 | 2 | 5 | 6 | 0 | 1 | 3 | 8 | 9 |

^ ^

Low High

In the example above the last part of the array is not correctly sorted.

#### Choice of pivot

The optimal pivot is one that divides the array in half. One method is median-of-three. We look at the first, middle and last elements of the array. We sort them and then chose the middle value as the pivot.

#### Run-time and memory use

Though Quicksort is considered an in-place sort – meaning all operations are performed on the same stored in memory. Nonetheless choices pivots as well as indices ‘Low’ and ‘High’ require variables.

# Implementation in Python and Java

Below is my implementation Python

import sys

class Quicksort(object):

def \_\_init\_\_(self):

print()

def quicksort(self, unsorted, low, high):

if low < high:

p = self.partition(unsorted, low, high)

self.quicksort(unsorted, low, p - 1)

self.quicksort(unsorted, p + 1, high)

return unsorted

def partition(self, unsorted, low, high):

pivot = unsorted[high]

i = low - 1

for j in range(low, high+1):

if unsorted[j] <= pivot:

i = i + 1

if i != j:

swap = unsorted[i]

unsorted[i] = unsorted[j]

unsorted[j] = swap

return i

def main():

unsorted = [7, 3, 8, 2, 1, 9, 4, 6, 5, 0]

quick = Quicksort()

quick.quicksort(unsorted, 0, 9)

print(unsorted)

if \_\_name\_\_ == "\_\_main\_\_":

main()

And now in Java