Student Name: Daniel Alconchel Vázquez
Semester: 1
Academic Year: 2022/2023

Faculty of Applied Physics and Mathematics
Institute of Physics and Applied Computer Science
QuickSort

---

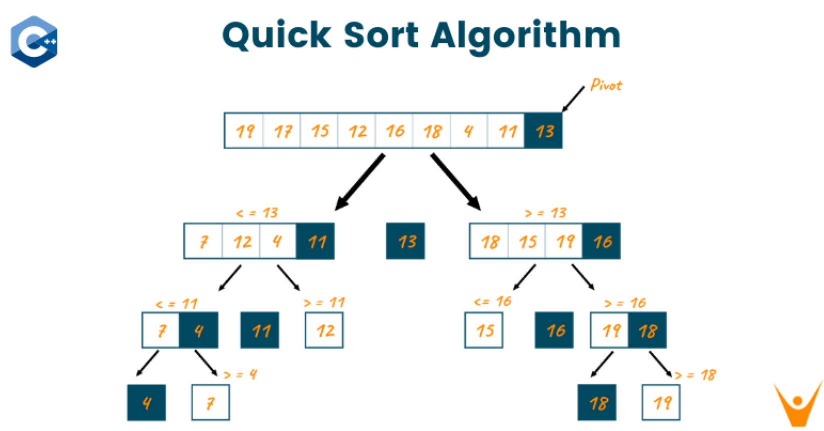
1. **Theoretical Introduction**

QuickSort is a Divide and Conquer algorithm that consists of 3 steps for sorting a typical sub-array $A[p...r]$. The idea is to pick an element as a pivot and partition the given array around the picked pivot. This partition can be recursively applied to the sub-arrays to form smaller partitioned sub-arrays. Let´s see the steps:

(a) **Divide:** Partition (rearrange) the array $A[p...r]$ into two (possibly empty) sub-arrays $A[p...q-1]$ and $A[q+1...r]$ such that each element of $A[p...q-1]$ is less than or equal to $A[q]$, which is, in turn, less than or equal to each element of $A[q+1...r]$. Compute the index q as a part of this partitioning procedure.

(b) **Conquer:** Sort the two sub-arrays $A[p...q-1]$ and $A[q+1...r]$ by recursive calls to quicksort.

(c) **Combine:** Because the sub-arrays are already sorted, no work is needed to combine them.



The following procedure implements quicksort:

```
QUICKSORT(A, p, r)
  if p < r
      q = PARTITION(A, p, r)
      QUICKSORT(A, p, q - 1)
      QUICKSORT(A, q + 1, r)
```

To sort an entire array A, the initial call is $QUICKSORT(A, 1, A.length)$. The key to the algorithm is the $PARTITION$ procedure, which rearranges the sub-array $A[p...r]$ in place:

```
PARTITION(A, p, r)
x = A[r]
i = p − 1
for j = p to r − 1
    if A[j] ≤ x
        i = i + 1
        exchange A[i] with A[j]
exchange A[i + 1] with A[r]
return i + 1
```

2. Description of Implementation

(a) Algorithm

For the implementation we are going to consider the pseudo-code from the previous section, so we have:

```python
def PARTITION(A,p,r):
    x=A[r]
    i=p-1
    for j in range (p,r):
        if A[j]<=x:
            i=i+1
            (A[i],A[j])=(A[j],A[i])
    (A[i+1],A[r])=(A[r],A[i+1])
    return i+1
```

```python
def QUICKSORT(A,p,r):
    if p<r:
        q=PARTITION(A,p,r)
        QUICKSORT(A,p,q-1)
        QUICKSORT(A,q+1,r)
```

Let's see an example:

```
In [12]: A=[2,8,7,1,3,5,6,4]
         QUICKSORT(A,0,len(A)-1)
         A

Out[12]: [1, 2, 3, 4, 5, 6, 7, 8]
```

(b) Unit Test

Since the objective of quicksort is sorting the array, we can define a function that sees if the array is sorted or not:

```python
def SORTED(A):
    for j in range(1,len(A)-1):
        if A[j-1]>A[j]:
            print("False")
            return 1
    print("True")
```

So we can generate random arrays, apply quicksort to them and see if it is sorted or not, so let's do the following unit test:

```python
import numpy as np

def unit_test():
    for i in range (10):
        A = np.random.random(1024)
```

```
A.tolist()
print("Array 1", A)
QUICKSORT(A,0,len(A)-1)
print("Result:")
SORTED(A)
```

```
unit_test()
```

```
Array 1 [0.41138196 0.57569209 0.46863333 ... 0.93398863 0.21333774 0.72794098]
Result:
True
Array 1 [0.69686112 0.04776689 0.48293928 ... 0.37234248 0.25611569 0.95494013]
Result:
True
Array 1 [0.31908404 0.14046774 0.29376551 ... 0.16484978 0.55765199 0.29728541]
Result:
True
Array 1 [0.37575534 0.24119999 0.80202207 ... 0.35638797 0.40937311 0.24029256]
Result:
True
Array 1 [0.51784735 0.03105887 0.42503681 ... 0.80189685 0.34210684 0.64277637]
Result:
True
Array 1 [0.39638079 0.81621566 0.03053212 ... 0.81694526 0.89471081 0.87092932]
Result:
True
Array 1 [0.817976   0.49592901 0.55765885 ... 0.01127768 0.48444268 0.04060839]
Result:
True
Array 1 [0.02819151 0.90493479 0.23199143 ... 0.3582075  0.37596609 0.28264843]
Result:
True
Array 1 [0.03847609 0.87346996 0.84894843 ... 0.22880364 0.27768125 0.33919016]
Result:
True
Array 1 [0.1621792  0.18155124 0.66147092 ... 0.37779171 0.10661048 0.10163773]
Result:
True
```

3. **Description of unit test and computational complexity**

   (a) **Unit Test** Already explained in 2b.

   (b) **Parameter N of the algorithm**

   The **worst-case behavior** for quicksort occurs when the partitioning routine produces one sub-problem with $n-1$ elements and one with 0 elements. The partitioning code costs $\Theta(n)$ time. Since the recursive call on an array of size 0 just returns, $T(0) = \Theta(1)$ and the recurrence for the running time is

   $$T(n9 = T(n-1) + T(0) + \Theta(n) = T(n-1) + \Theta(n)$$

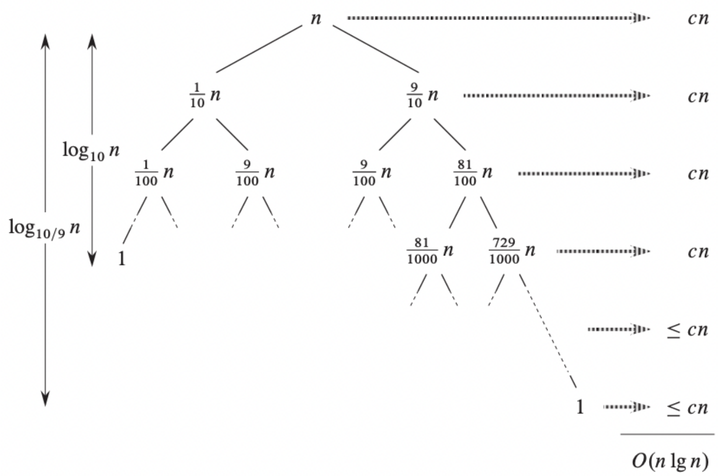
   Intuitively, if we sum the costs incurred at each level of the recursion, we get an arithmetic series which evaluates to $\Theta(n^2)$, so by substitution we have that $T(n) = \Theta(n^2)$ in the **worst-case behavior**.

   Now let's see the **best-case partitioning**. In the most even possible split, $PARTITION$ produces two sub-problems, each of size no more than $n/2$, since one is of size $\lfloor n/2 \rfloor$ and one of size $\lceil n/2 \rceil - 1$. In this case, quicksort runs much faster. The recurrence for the running time is then:

   $$T(n) = 2T(n/2) + \Theta(n)$$

   where we tolerate the sloppiness from ignoring the floor and ceiling and from sub-tracting 1. By case 2 of the master theorem this recurrence has the solution is $T(n) = \Theta(n\log(n))$

   Finally, in the **average-case running**, it is much closer to the best case than to the worst case



Suppose, for example, that the partitioning algorithm always produces a 9-to-1 proportional split, which at first blush seems quite unbalanced. We then obtain the recurrence

$$T(n) = T(9n/10) + T(n/10) + cn$$

on the running time of quicksort, where we have explicitly included the constant c hidden in the $\Theta(n)$ term. The previous figure shows the recursion tree for this recurrence. Notice that every level of the tree has cost cn, until the recursion reached a boundary condition at depth $\log_{10} n = \Theta(\log n)$. The recursion terminates at depth $\log_{10/9} n = \Theta(\log n)$, so the total cost of quicksort is $\Theta(n \log n)$.

(c) **Some Examples**

Let's see more examples:

```
def unit_test2():
    size=1024
    for i in range (10):
        A = np.random.random(size)
        A.tolist()
        print("Array 1", A)
        QUICKSORT(A,0,len(A)-1)
        print("Result:")
        SORTED(A)
        size=size*2
```

```
unit_test2()
```

```
Array 1 [0.6775493  0.12556696 0.23261311 ... 0.12456557 0.04413183 0.2
1266269]
Result:
True
Array 1 [0.15905901 0.72184597 0.54809616 ... 0.3327998  0.0484294  0.6
2596972]
Result:
True
Array 1 [0.75131703 0.53151424 0.63609716 ... 0.26380774 0.90787177 0.7
4049898]
Result:
True
Array 1 [0.44459964 0.26356737 0.7802722  ... 0.04477238 0.67390688 0.1
3234603]
Result:
True
Array 1 [0.06764037 0.17813707 0.3822946  ... 0.74185882 0.08271806 0.8
2154976]
Result:
True
Array 1 [0.47038258 0.55226827 0.14487484 ... 0.926356   0.59531344 0.4
6073317]
Result:
True
Array 1 [0.70301919 0.00352703 0.1918506  ... 0.91922753 0.59408773 0.2
3321084]
Result:
True
Array 1 [0.92905584 0.71463358 0.00717379 ... 0.02200004 0.30439719 0.7
6689952]
Result:
True
Array 1 [0.79048932 0.98897536 0.30578156 ... 0.42048557 0.94613337 0.7
1188031]
Result:
True
Array 1 [0.66227647 0.3743658  0.9783583  ... 0.34213012 0.75697747 0.0
7690143]
Result:
True
```

4. **Results**

   Let's prepare a new script:

```
from timeit import default_timer as timer

N=10
for i in range (15):
    A=np.random.random(N)
    A.tolist()

    times = []
    for j in range (10):
        start = timer()
        QUICKSORT(A,0,len(A)-1)
        end = timer()
        time = end-start
        times.append(time)

    average = np.mean(times)
    deviation = np.std(times)

    print("Size of N:", N)
    print("Average Time:", average)
    print("Standard Deviation:", deviation)
    print("\n")
    N+=100
```

   We get the following output (*not all the output*):

```
Size of N: 100
Average Time: 0.0026351265999892347
Standard Deviation: 0.0010235340742359676


Size of N: 200
Average Time: 0.010390990400060219
Standard Deviation: 0.004433795081502104


Size of N: 300
Average Time: 0.023191566600007717
Standard Deviation: 0.00960129928796335
```

Now, we can create a table with those data:

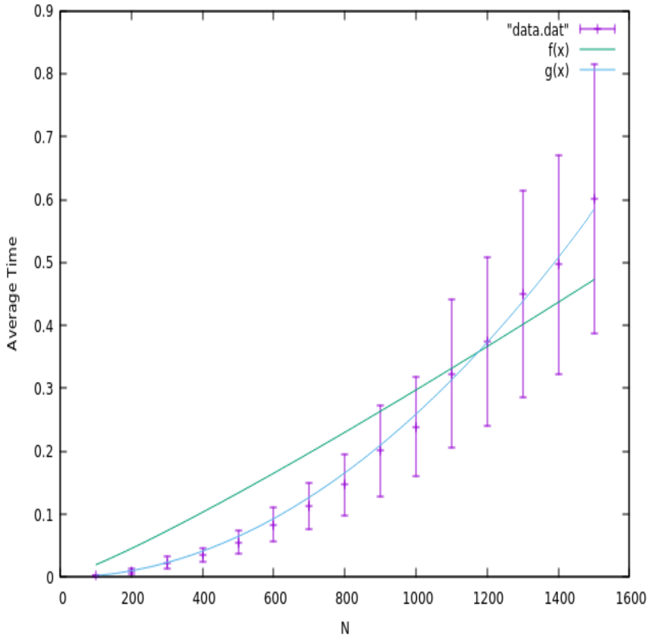
N	Average Time	Standard Deviation
100	0.0026351265999892347	0.0010235340742359676
200	0.010390990400060219	0.004433795081502104
300	0.023191566600007717	0.00960129928796335
400	0.0353624888000013	0.01131113519921623
500	0.05548215869996511	0.017993138556731746
600	0.08358199629997216	0.02734790036674195
700	0.11298645040001247	0.03745186523885189
800	0.1467695115999959	0.048235712308892086
900	0.20074805709994054	0.07310429969212746
1000	0.23943876980001733	0.0796041332294781
1100	0.32323740439999255	0.11763969637628843
1200	0.3748904607999748	0.1340223953607623
1300	0.44969641949996914	0.16392417886713767
1400	0.49649058170000443	0.1733668688009081
1500	0.6010589021000442	0.21368952893608614

Let's use gnuplot for making a representation of the data:

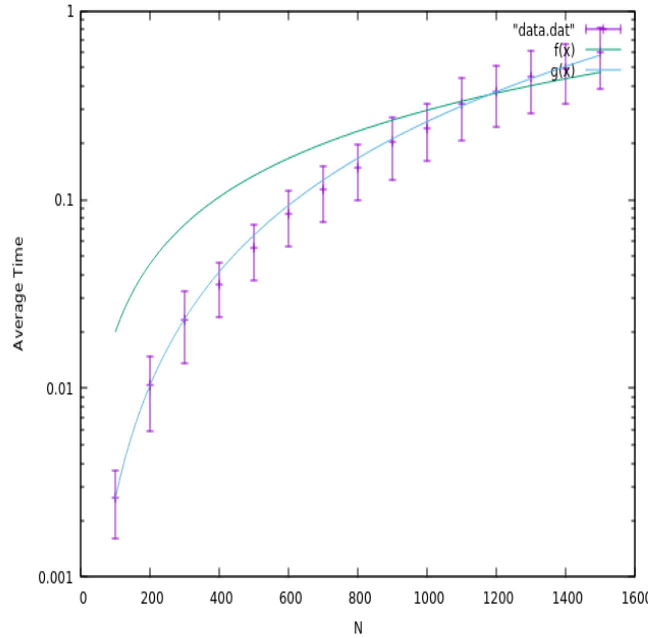
```
gnuplot> set xlabel "N"
gnuplot> set ylabel "Average Time"
gnuplot> f(x) = a*x*log(x)
gnuplot> g(x) = b*x**2
gnuplot> fit f(x) "data.dat" u 1:2 via a
iter      chisq       delta/lim  lambda   a
   0 6.1097474697e+08   0.00e+00  6.38e+03    1.000000e+00
   1 2.3866201706e+06  -2.55e+07  6.38e+02    6.254038e-02
   2 1.1248457315e+00  -2.12e+11  6.38e+01    8.470565e-05
   3 6.5538565874e-02  -1.62e+06  6.38e+00    4.306881e-05
   4 6.5538565827e-02  -7.18e-05  6.38e-01    4.306853e-05
iter      chisq       delta/lim  lambda   a

After 4 iterations the fit converged.
final sum of squares of residuals : 0.0655386
rel. change during last iteration : -7.18351e-10

degrees of freedom    (FIT_NDF)                        : 14
rms of residuals      (FIT_STDFIT) = sqrt(WSSR/ndf)    : 0.0684202
variance of residuals (reduced chisquare) = WSSR/ndf   : 0.00468133
```



(a) Results Plot



(b) Log Scale Y

```
Final set of parameters            Asymptotic Standard Error
=======================            ==========================
a               = 4.30685e-05      +/- 2.768e-06    (6.427%)
gnuplot> fit g(x) "data.dat" u 1:2 via b
iter      chisq        delta/lim  lambda   b
   0 1.7831190760e+13   0.00e+00  1.09e+06    1.000000e+00
   1 6.9653088906e+10  -2.55e+07  1.09e+05    6.250024e-02
   2 3.0915695656e+04  -2.25e+11  1.09e+04    4.189799e-05
   3 1.9401430799e-03  -1.59e+12  1.09e+03    2.593705e-07
   4 1.9387690675e-03  -7.09e+01  1.09e+02    2.590930e-07
   5 1.9387690675e-03  -3.36e-11  1.09e+01    2.590930e-07
iter      chisq        delta/lim  lambda   b

After 5 iterations the fit converged.
final sum of squares of residuals : 0.00193877
rel. change during last iteration : -3.35533e-16

degrees of freedom    (FIT_NDF)                        : 14
rms of residuals      (FIT_STDFIT) = sqrt(WSSR/ndf)    : 0.0117679
variance of residuals (reduced chisquare) = WSSR/ndf   : 0.000138484

Final set of parameters            Asymptotic Standard Error
=======================            ==========================
b               = 2.59093e-07      +/- 2.787e-09    (1.076%)
gnuplot> plot "data.dat" with yerrorbars, f(x), g(x)
gnuplot> set log y
gnuplot> replot
```

We can see that YerrorBars are quite huge in this case, that's because some cases takes $\Theta(n^2)$ and other times $\Theta(n\log(n))$.

5. **Conclusion**

As we have seen, QuickSort is a fast algorithm in practice, even with its worst case complexity $\Theta(n^2)$ begin bigger than more some other algorithm like HeapSort or MergeSort. It is also a versatile algorithm, as it can be implemented in so many ways just by changing the choice of pivot. However, it has some disadvantages, like it is not a stable algorithm, because it is recursive, and it requires a quadratic time for the worst case.

References

[1] QuickSort in C++

[2] QuickSort