Data Structure & Algorithm

1. Sorting Algorithm

1.1 Quicksort

1.1.1 Brief

Quicksort is an efficient and commonly used sorting algorithm, serving as a method that puts one number of array in correct place for each recursion.
Quicksort is a comparison sort, which indicates that for any kind of dataset that the relation of less or greater than is defined it will work.
From efficiency aspect, quicksort is not a stable algorithm, meaning that the time complexity of quicksort is not same for average and worst case.
Consider quicksort is a kind of in-place algorithm, meaning that during sorting process there are only swap operations and no copy or cover operations, which indicates that only few addition memory will be needed in the process.

1.1.2 Algorithm

Suppose, we have an array of integers `A = [1,6,3,2,5,8,7,4]` as input and the supposed output after quicksort is `A=[1,2,3,4,5,6,7,8]`.
The main idea of quicksort is recursion. Our mission is to seperate the array into two subarray, less than and great than, then use same algorithm on subarray again.

1. We choose the last number of array as a `key`, in this case, `key = 4`, and `A[len(A)-1] = key`.

Index	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
Pointer								key
Value	1	6	3	2	5	8	7	4

2. We initialize an index `bar = -1`.

Index	**-1**	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
Pointer	bar								key
Value		1	6	3	2	5	8	7	4

3. For `i` in the range of indices, meaning `i++` for each step, do the loop:
   if the value of `A[i]` is less than `key`, `bar++` and then swap `A[bar]` and `A[i]`.

   In this instance, for `i=0`, `bar=-1`, `A[i]=1` is less than `key`, so `bar++`, `bar=0`, then swap `A[bar]` and `A[i]`. Because `i==bar` in this case, nothing will change. We have:

Index	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
Pointer	bar							key
Value	**1**	6	3	2	5	8	7	4

For `i=1`, `bar=0`, `A[i]=6` is great than `key`, so do nothing this step. We have:

Index	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
Pointer	bar							key
Value	**1**	6	3	2	5	8	7	4

For `i=2`, `bar=0`, `A[i]=3` is less than `key`, so `bar++`, `bar=1`, then swap `A[bar]` and `A[i]`. We have:

Index	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
Pointer		bar						key
Value	**1**	**3**	6	2	5	8	7	4

For `i=3`, `bar=1`, `A[i]=2` is less than `key`, so `bar++`, `bar=2`, then swap `A[bar]` and `A[i]`. We have:

Index	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
Pointer			bar					key
Value	**1**	**3**	**2**	6	5	8	7	4

For `i=4`, `bar=2`, `A[i]=5` is great than `key`, so do nothing in this step. Similarly, we do nothing for `i=5` and `i=6`.

Index	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
Pointer			bar					key
Value	**1**	**3**	**2**	6	5	8	7	4

4. When the loop reach `key`, which means it reach the last index of array, swap `A[bar+1]` and `A[key]`.

   For `i=7`, `bar=2`, the loop reach the end of the array, where we set the `key`. So, in this step, we swap `A[bar+1]` and `A[len(A)-1]`, we have:

Index	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
Pointer			bar	bar+1				
Value	**1**	**3**	**2**	4	5	8	7	6

   By the end of loop, the array have reached a state that the index less than `bar+1` holds smaller number than `A[bar+1]` and index great than `bar+1` holds larger number than `A[bar+1]`. And now we find number 4, the `A[bar+1]` and the previous `key`, on the correct position of the sorted array.

5. Recursion on subarrays `A[0:bar+1]` and `A[bar+2:len(A)]`.

   In this case, `A[0:bar+1]` is `[A[0],A[1],A[2]]`, `A[bar+2:len(A)]` is `[A[4],A[5],A[6],A[7]]`. Because number

4 is already find the correct place to stay in, so it won't join the recursion.

1.1.3 Python Code

Python
```
A = [1,6,3,2,5,8,7,4]

def part(A,s,r):
    # s: start point
    # r: rear point
    key=A[r] # Choose last number as key
    bar=s-1 # Set bar as start -1
    for i in range(s,r):
        if A[i]<=key:
            bar+=1
            A[bar], A[i] = A[i], A[bar] # Swap
    A[bar+1], A[r] = A[r], A[bar+1] # Swar key with bar+1
    return bar+1

def qsort(A,s,r):
    if s<r:
        m=part(A,s,r)
        qsort(A,s,m-1)
        qsort(A,m+1,r)

qsort(A,0,len(A)-1)
print A
```

1.1.4 C++ Code

```cpp
#include "iostream"
using namespace std;

template <class T>
void change(T &a,T &b){
    T temp;
    temp = a;
    a = b;
    b = temp;
}

int part(int A[],int s,int r){
    // s: start point
    // r: rear point
    int key=A[r]; // Choose last number as key
    int bar=s-1; // Set bar as start -1
    for (int i=s;i<r;i++){
        if (A[i]<=key){
            bar++;
            change(A[bar], A[i]); // Swap
        }
    }
    change(A[bar+1], A[r]); // Swar key with bar+1
    return bar+1;
}

void qsort(int A[],int s,int r){
    if(s<r){
        int m=part(A,s,r);
        qsort(A,s,m-1);
        qsort(A,m+1,r);
    }
}

int main(){
    int A[]={1,6,3,2,5,8,7,4};
    int lenth = sizeof(A)/sizeof(A[0]);
    qsort(A,0,lenth-1);
    for (int i=0;i<lenth;i++){
        cout << A[i];
    }
    cout << endl;
    return 0;
}
```

1.1.5 Time Complexity

Worst case

The worst case is the most unbalanced partition when we divide the list into two sublists of sizes 0 and $n-1$. This may occur if the key happens to be the smallest or largest element in the list.

If it happens to all the recursion steps, we will have every recursion is just dealing with the list whose size is 1 smaller than the previous one. In this case, we will have $n-1$ steps of recursion, each recursion dealing with $n-1$ numbers. So we could have the time complexity:

$$f(n) = \sum_{i=0}^{n-1} i = O(n^2)$$

Best case
The best case is the most balanced case, each recursion we divide the list into two nearly equal subarrays. This means each recursive call processes a list of half the size. It's easy to prove that we only have $\log n$ recursion step to get the size of array into 1. And we could prove that for each level of recursion, including all the subarrays in this level, it needs only $O(n)$. So we could have the time complexity:

$$f(n) = O(n) \cdot O(\log n) = O(n \log n)$$

Average case
We could use recursion function to solve the average time complexity. See the prove on Wikipedia: Quicksort

1.1.6 Optimization

By the definition of worst case, we could easily find that the sorted array (positive sequence or inverse) may cause the worst time complexity. To solve this problem and to optimize the algorithm, we could use following approaches:

1. Choose key randomly.
2. Choose middle index.
3. Choose the median of the first, middle and last element

If the number of elements in a subarray is quite small, under a threshold, the recursive method may not efficient anymore. We could switch to some non-recursive algorithm like insertion sort.