Lab01-Algorithm Analysis

CS214-Algorithm and Complexity, Xiaofeng Gao, Spring 2021.

∗ If there is any problem, please contact TA Haolin Zhou. Also please use English in homework.

∗ Name: Zilong Li Student ID: 518070910095 Email: logcreative-lzl@sjtu.edu.cn

1. *Complexity Analysis.* Please analyze the time and space complexity of Alg. 1 and Alg. 2.

Algorithm 1: QuickSort

Input: An array $A[1, \cdots, n]$
Output: $A[1, \cdots, n]$ sorted nondecreasingly

```
pivot ← A[n]; i ← 1;
for j ← 1 to n − 1 do
    if A[j] < pivot then
        swap A[i] and A[j];
        i ← i + 1;
swap A[i] and A[n];
if i > 1 then
    QuickSort(A[1, ··· , i − 1]);
if i < n then
    QuickSort(A[i + 1, ··· , n]);
```

Algorithm 2: CocktailSort

Input: An array $A[1, \cdots, n]$
Output: $A[1, \cdots, n]$ sorted nonincreasingly

```
i ← 1; j ← n;sorted ← false;
while not sorted do
    sorted ← true;
    for k ← i to j − 1 do
        if A[k] < A[k + 1] then
            swap A[k] and A[k + 1];
            sorted ← false;
    j ← j − 1;
    for k ← j downto i + 1 do
        if A[k − 1] < A[k] then
            swap A[k − 1] and A[k];
            sorted ← false;
    i ← i + 1;
```

(a) Fill in the blanks and **explain** your answers. You need to answer when the best case and the worst case happen.

Algorithm	Time Complexity[1]	Space Complexity
QuickSort	$\Omega(n \log n)$, $O(n \log n)$, $O(n^2)$	$O(\log n)$
CocktailSort	$\Omega(n)$, $O(n^2)$, $O(n^2)$	$O(1)$

[1] The response order can be given in *best*, *average*, and *worst*.

(b) For Alg. 1, how to modify the algorithm to achieve the same expected performance as the **average** case when the **worst** case happens?

Solution. (a) The problem only needs the proof of best and worst cases. So, I will not prove the conclusion of the average case. Instead, I will perform a series of test and note some thoughts to show that the conclusion is correct.

Algorithm 1 – QuickSort:

Best Case $\Omega(n \log n)$ Appears when every time the pivot seperates the array into two equally-sized subarrays. Then the call could be a binary tree with approximately $\log n$ layers. In Layer j, there are 2^j partitions and $\frac{n}{2^j}$ elements every partition. Thus, the complexity comes to:

$$T(n) \approx \sum_{j=1}^{\log n} \frac{n}{2^j} \times 2^j = n \log n = \Omega(n \log n)$$

Worst Case $O(n^2)$ Happens when every time the pivot always separates the array into 1 and $n-1$ sized subarrays. This may occur if the pivot happens to be the smallest element in every partition, for example, all the element are equal.
Then, a linear chain of partitioning is produced and the time complexity comes to:

$$T(n) = n + (n-1) + \cdots + 1 = \frac{(n+1)n}{2} = O(n^2)$$

$\boxed{1}$	3	$\underline{2}$
1	$\underline{3}$	$\underline{2}$
1	$\boxed{3}$	$\boxed{2}$
$\boxed{1}$	2	3
1	2	$\boxed{3}$
1	2	3

Best Case for QuickSort

$\underline{3}$	3	$\underline{3}$
3	$\underline{3}$	$\underline{3}$
$\boxed{3}$	3	$\boxed{3}$
3	$\underline{3}$	$\underline{3}$
3	$\boxed{3}$	$\boxed{3}$
3	3	$\boxed{3}$
3	3	3

Worst Case for QuickSort

Average Case $O(n \log n)$ The proof of the average case could be done by the master theorem for divide-and-conquer recurrences in the future.

Space Complexity $O(\log n)$ The algorithm is done in place for every partition, which is $O(1)$ space, but the sort of every partition should be done recursively before the other in the same layer could be performed. Thus the call stack depth is bounded by $O(\log n)$ spaces.

Algorithm 2 – CocktailSort:

Best Case $\Omega(n)$ Appears when the array has already been sorted nonincreasingly. The *sorted* state will stay *true* and scanning the array is the only thing to be done:

$$T(n) = n - 1 + n - 2 = 2n - 3 = \Omega(n)$$

Worst Case $O(n^2)$ Happens when the array is sorted nondecreasingly. The first **for** loop will push the smallest element among $A[i] \cdots A[j]$ to $A[j]$, while the second **for** loop will push the largest element among $A[i] \cdots A[j-1]$ to $A[i]$. And in the worst case, every loop will not be ignored in order to fix an element due to a complete anti-sorted array. The procedure will continue until there is only one element to be sorted. The time complexity will be

$$T(n) = n - 1 + n - 2 + \cdots + 1 = \frac{(n-1)n}{2} = O(n^2)$$

$\underline{4}$	$\underline{3}$	2	1
4	$\underline{3}$	$\underline{2}$	1
4	3	$\underline{2}$	$\underline{1}$
4	$\underline{3}$	$\underline{2}$	1
$\underline{4}$	$\underline{3}$	2	1
4	3	2	1

Best Case for CocktailSort

$\boxed{1}$	$\boxed{2}$	3	4
2	$\boxed{1}$	$\boxed{3}$	4
2	3	$\boxed{1}$	$\boxed{4}$
2	$\boxed{3}$	$\boxed{4}$	1
$\boxed{2}$	$\boxed{4}$	3	1
4	$\boxed{2}$	$\boxed{3}$	1
4	3	2	1

Worst Case for CockTailSort

Average Case $O(n^2)$ Because it doesn't use any tree structure. And by only calculating the average between the best and worst case, it is $\frac{(n-1)n}{4} = O(n^2)$. As a matter of fact,

it could be more than that due to the scanning process is more than the swaping process. And it could be improved if most of the elements are ordered and the complexity drops down to $O(n)$.

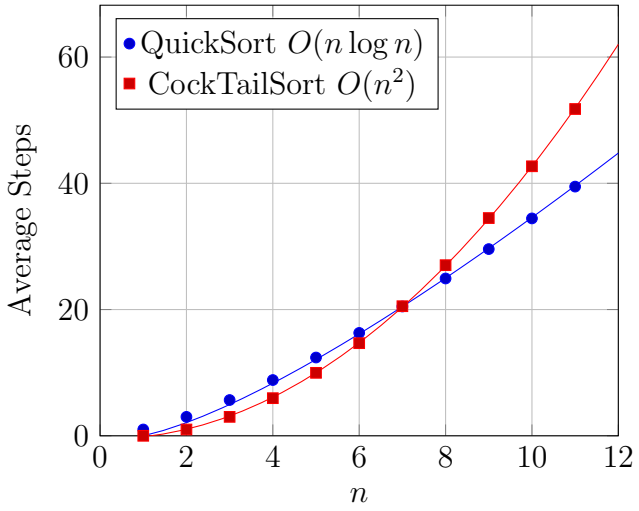
Space Complexity $O(1)$ No extra space is needed during CocktailSort. The sorting is always performed in-place.

(b) Choose the pivot as a **random element** in the array. To achieve this, one could choose a random index or the middle index or the median of the first, middle and last element of the partition for the pivot. This could make the choice of the privot to be randomrized and let the worst case to be the average case, when the order of the elements is unknown in the array.

$$pivot \leftarrow \texttt{randomElement}(A[1, \cdots, n]);$$

where

$$\begin{aligned} \texttt{randomElement}(A[1, \cdots, n]) &= A[\mathsf{randint}(1, n)] \\ \text{or} \quad &= A\left[\left\lfloor \frac{n}{2} \right\rfloor\right] \\ \text{or} \quad &= \mathsf{median}\left(A[1], A\left[\left\lfloor \frac{n}{2} \right\rfloor\right], A[n]\right) \end{aligned}$$



Algorithm	Fitted Function[1]	R^2
QuickSort	$1.502n \log n$	0.9984
CocktailSort	$0.4453n^2 - 0.1365n - 0.473$	0.9999

[1] The fitting is only based on the data from $n = 1$ to 11.

The test for average case is done as follows: Traverse all the permutation within the array of size n, then test all $n!$ permutations for different algorithms. Count the step for every permutation and calculate the average number of steps, which reflects the time complexity of the algorithm.

`permutation.h` provides a way to get all the permutation and perform the test for the function pointer of the algorithm on each arrangement.

The permutation is traversed in a swaping and recursive way. And any destruction on the original array will break the traverse of permutation. Thus, the replica of the array is sent for the sorting algorithm after the generation of the permutation.

Considering that $12! = 4.7 \times 10^8$ is a huge number, C++ will not provide the correct integer when I add up the count (it will become a negative one). So, I only perform the precise test from 1 to 11, and to avoid overflow, `long long int` is used to save the permutation count and the average number of steps is counted through a divided way, highlighted in line 25:

$$\bar{s}' = \frac{\bar{s} \times N + s}{N+1} = \bar{s} \times \frac{N}{N+1} + \frac{s}{N+1}$$

`permutation.h`

```
1 #include <iostream>
2
```

```cpp
int steps;
double averagestep;
long long int permucount;

void output(int *a, int n){
    for(int l = 0; l<n; ++l) std::cout<<a[l]<<' ';
    std::cout<<'\n';
}

void swap(int& a, int &b){
    int temp = a;
    a = b;
    b = temp;
}

// Generate permutation by swaping
void permutation(int a[], int n, int cur, void (*pf)(int*a, int n)){
    if(cur==n){
        steps = 0;
        int* b = new int[n];
        for(int i = 0;i<n;  ++i) b[i] = a[i];
        (*pf)(b,n); // The array is broken when it is put into the function.
        averagestep = averagestep * (1.0 * permucount / (permucount + 1)) + 1.0 * steps / (permucount + 1);     // avoid overflow
        ++permucount;
        delete[] b;
    }
    else for(int i = cur; i < n; i++) {
                swap(a[cur], a[i]);
                permutation(a, n, cur+1, *pf);
                swap(a[cur], a[i]);
        }
}

double permutation(int*a, int n, void (*pf)(int*a, int n)){
    averagestep = 0;
    permucount = 0;
    permutation(a,n,0,*pf);
    return averagestep;
}
```

According to Algorithm 1, `quickSort.cpp` provides the algorithm in that way. The count on steps is based on the `if` statement and the `swap` directive.

`quickSort.cpp`

```cpp
#include "permutation.h"

void QuickSort(int* a, int n){
    int pivot = a[n-1]; int i = 0;
    for(int j = 0; j <= n-2; ++j){
        if(a[j]<pivot){
            swap(a[i],a[j]);
            ++i;
        }
        ++steps;
    }
    swap(a[i],a[n-1]);
    ++steps;
    if(i>0) QuickSort(a,i);
    if(i<n-1) QuickSort(a+i+1,n-i-1);
}

int main(){
    for(int n = 1; n<=20; ++n){
        int* a = new int[n];
        for(int i = 0; i<n; ++i) a[i] = i+1;
        double res = permutation(a,n,QuickSort);
        std::cout << n << '\t' << res << '\n';
        delete[] a;
    }
    return 0;
}
```

According to Algorithm 2, `cockTailSort.cpp` provides the algorithm in that way. The count on steps is based on the `if` statement.

`cockTailSort.cpp`

```cpp
#include "permutation.h"

void CockTailSort(int* a, int n){
    int i = 0, j = n-1; bool sorted = false;
    while(!sorted){
        sorted = true;
        for(int k = i; k<=j-1; ++k){
            if(a[k]<a[k+1]){
                swap(a[k],a[k+1]);
                sorted = false;
            }
            ++steps;
        }
        --j;
        for(int k = j; k>=i+1; --k){
            if(a[k-1]<a[k]){
                swap(a[k-1],a[k]);
                sorted = false;
            }
            ++steps;
        }
        ++i;
    }
}

int main(){
    for(int n = 1; n<=20; ++n){
        int* a = new int[n];
        for(int i = 0; i<n; ++i) a[i] = i+1;
        steps = 0;
        double res = permutation(a,n,CockTailSort);
        std::cout << n << '\t' << res << '\n';
        delete[] a;
    }
    return 0;
}
```

As a result, a precise figure of the average number of steps for the two algorithms is shown in the previous page. I fitted the curve base on the previous analysis by using `Matlab`. The fitting result is quite satisfying, which indicates that the time complexity analysis on average cases is correct.

The coefficients here is not pretty accurate when n close up to infinity. For example, QuickSort acutually has a time complexity of $1.39n \log_2 n$ based on the master theorem for divide-and-conquer recurrences. Only math deduction in the future could show the real shape of the complexity. Machine can not get a precise result when $n!$ is close up to an insane number.

□

2. *Growth Analysis.* Rank the following functions by order of growth with brief explanations: that is, find an arrangement $g_1, g_2, \ldots, g_{15}$ of the functions $g_1 = \Omega(g_2), g_2 = \Omega(g_3), \ldots, g_{14} = \Omega(g_{15})$. Partition your list into equivalence classes such that functions $f(n)$ and $g(n)$ are in the same class if and only if $f(n) = \Theta(g(n))$. Use symbols "$=$" and "$\prec$" to order these functions appropriately. Here $\log n$ stands for $\ln n$.

1	n	$\log n$	$\log(\log n)$	$n \log n$
$\log_4 n$	2^n	4^n	$2^{\log n}$	2^{2^n}
$\log(n!)$	$n!$	$(2n)!$	$n^{1/2}$	n^2

Solution. Arrangement:

$$2^{2^n}, 2^{n^2}, (2n)!, n!, 4^n, 2^n, n^2, n \log n, \log(n!), n, 2^{\log n}, n^{1/2}, \log n, \log\log n, 1$$

Partition:

$$1 \prec \log\log n \prec \log n \prec n^{1/2} \prec 2^{\log n} \prec n \prec \log(n!)$$
$$= n \log n \prec n^2 \prec 2^n \prec 4^n \prec n! \prec (2n)! \prec 2^{n^2} \prec 2^{2^n}$$

Along the proof, relations (6) and (7) are considered as fundamentals. And Stirling's approximation is used:

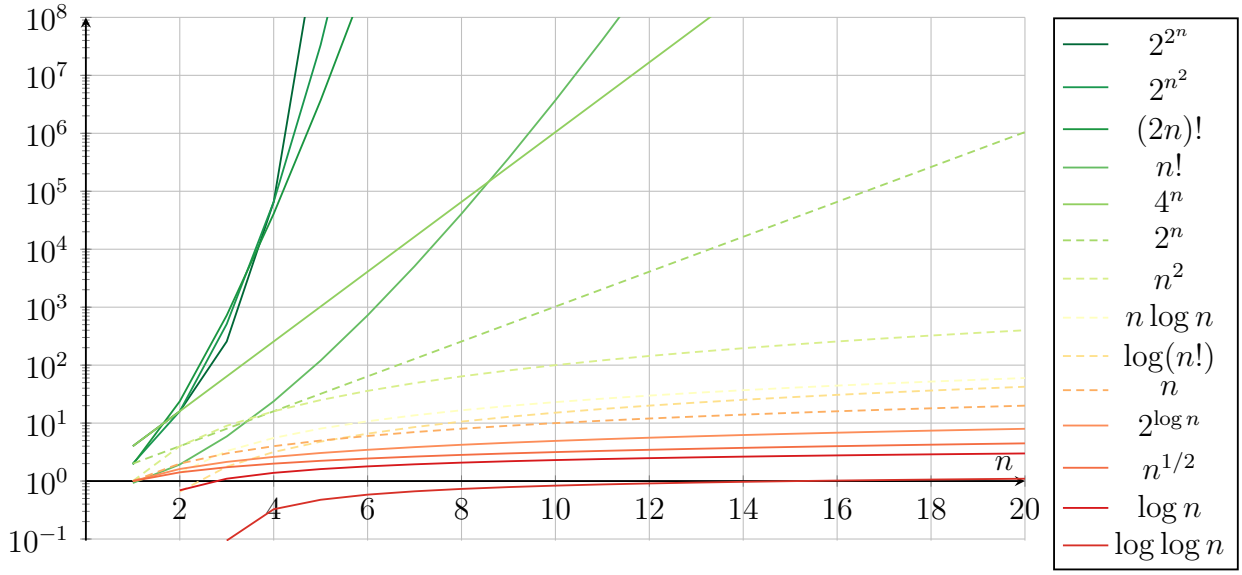
$$\lim_{n\to\infty} \frac{n!}{\sqrt{2\pi n}\left(\frac{n}{e}\right)^n} = 1$$

L'Hôpital's rule is used:

$$\lim_{n\to\infty} f(n) = \lim_{n\to\infty} g(n) = +\infty, g'(n) \neq 0, \exists \lim_{n\to\infty} \frac{f'(n)}{g'(n)} \Rightarrow \lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{f'(n)}{g'(n)}$$

The transformation rule of $\omega(\cdot)$ is used:

$$g(n) \neq 0 \Rightarrow \frac{\omega(f(x))}{g(x)} = \omega\left(\frac{f(x)}{g(x)}\right)$$



The proof is as follows:

$$2^{2^n} = \omega(2^{n^2}) \Leftarrow \quad \lim_{n\to\infty} \frac{2^{2^n}}{2^{n^2}} = \lim_{n\to\infty} 2^{2^n - n^2} = \lim_{n\to\infty} 2^{\omega(n^2)-n^2} = 2^{\infty} \tag{1}$$

$$\begin{aligned} 2^{n^2} = \omega((2n)!) \Leftarrow \quad \lim_{n\to\infty} \frac{2^{n^2}}{(2n)!} &= \lim_{n\to\infty} \frac{(2^n)^n}{\sqrt{2\pi(2n)}\left(\frac{n}{e}\right)^n} \\ &= \lim_{n\to\infty} \frac{1}{2\sqrt{\pi}} \left(e\frac{2^n}{n}\right)^n \cdot n^{-\frac{1}{2}} \\ &= \lim_{n\to\infty} \frac{1}{2\sqrt{\pi}} \left(e\frac{\omega(n^2)}{n}\right)^n \cdot n^{-\frac{1}{2}} \\ &= \lim_{n\to\infty} \frac{1}{2\sqrt{\pi}} e^n \omega\left(n^{n-\frac{1}{2}}\right) = \infty \end{aligned} \tag{2}$$

$$(2n)! = \omega(n!) \Leftarrow \quad \lim_{n\to\infty} \frac{(2n)!}{n!} = \lim_{n\to\infty} \prod_{i=n+1}^{2n} i = \infty \tag{3}$$

$$n! = \omega(4^n) \Leftarrow \quad \lim_{n\to\infty} \frac{n!}{4^n} = \lim_{n\to\infty} \frac{\sqrt{2\pi n}\left(\frac{n}{e}\right)^n}{4^n} = \infty \tag{4}$$

$$4^n = \omega(2^n) \Leftarrow \quad \lim_{n\to\infty} \frac{4^n}{2^n} = \lim_{n\to\infty} 2^n = \infty \tag{5}$$

$$\begin{aligned} 2^n = \omega(n^2) \Leftarrow \quad \lim_{n\to\infty} \frac{2^n}{n^2} &= \lim_{n\to\infty} \frac{e^{n\log 2}}{n^2} \\ &= \lim_{n\to\infty} \frac{1 + n\log 2 + \frac{(n\log 2)^2}{2} + \frac{(n\log 2)^3}{6} + \omega((n\log 2)^3)}{n^2} \\ &= \lim_{n\to\infty} \left[\alpha + \frac{n\log^3 2}{6} + \omega(n\log^3 2)\right] \ (\alpha > 0) = \infty \end{aligned} \tag{6}$$

$$n^2 = \omega(n\log n) \Leftarrow \quad \lim_{n\to\infty} \frac{n^2}{n\log n} = \lim_{n\to\infty} \frac{n}{\log n} = \lim_{n\to\infty} \frac{n'}{(\log n)'} = \lim_{n\to\infty} \frac{1}{\frac{1}{n}} = \infty \tag{7}$$

$$n\log n = \Theta(\log(n!)) \Leftarrow \quad \lim_{n\to\infty} \frac{n\log n}{\log(n!)} = \lim_{n\to\infty} \frac{n\log n}{\frac{1}{2}\log 2\pi n + n\log n - n} = 1 \tag{8}$$

$$\log(n!) = \omega(n) \Leftarrow \quad \lim_{n\to\infty} \frac{\log(n!)}{n} = \lim_{n\to\infty} \left(\frac{\pi}{n} + \frac{\log n}{2n} + \log n - 1\right) = \infty \tag{9}$$

$$n = \omega(2^{\log n}) \Leftarrow \quad \lim_{n\to\infty} \frac{n}{2^{\log n}} = \lim_{n\to\infty} \frac{n}{2^{\frac{\log_2 n}{\log_2 e}}} = \lim_{n\to\infty} n^{1-\frac{1}{\log_2 e}} = \lim_{n\to\infty} n^{0.31} = \infty \tag{10}$$

$$2^{\log n} = \omega(n^{1/2}) \Leftarrow \quad \lim_{n\to\infty} \frac{2^{\log n}}{n^{1/2}} = \lim_{n\to\infty} n^{\frac{1}{\log_2 e}-\frac{1}{2}} = \lim_{n\to\infty} n^{0.19} = \infty \tag{11}$$

$$n^{1/2} = \omega(\log n) \Leftarrow \quad \lim_{n\to\infty} \frac{n^{1/2}}{\log n} = \lim_{n\to\infty} \frac{\left(n^{1/2}\right)'}{(\log n)'} = \lim_{n\to\infty} \frac{n^{1/2}}{2} = \infty \tag{12}$$

$$\log n = \omega(\log\log n) \Leftarrow \quad \lim_{n\to\infty} \frac{\log n}{\log\log n} = \lim_{n\to\infty} \frac{(\log n)'}{(\log\log n)'} = \lim_{n\to\infty} \frac{1/n}{1/(n\log n)} = \infty \tag{13}$$

$$\log\log n = \omega(1) \Leftarrow \quad \lim_{n\to\infty} \log\log n = \infty \tag{14}$$

□

Remark: You need to include your .pdf and .tex files in your uploaded .rar or .zip file.