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:Xinhao Zheng Student ID:519021910528 Email: void_zxh@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

```
1  pivot ← A[n]; i ← 1;
2  for j ← 1 to n − 1 do
3      if A[j] < pivot then
4          swap A[i] and A[j];
5          i ← i + 1;
6  swap A[i] and A[n];
7  if i > 1 then
       QuickSort(A[1, ··· , i − 1]);
8  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

```
1  i ← 1; j ← n;sorted ← false;
2  while not sorted do
3      sorted ← true;
4      for k ← i to j − 1 do
5          if A[k] < A[k + 1] then
6              swap A[k] and A[k + 1];
7              sorted ← false;
8      j ← j − 1;
9      for k ← j downto i + 1 do
10         if A[k − 1] < A[k] then
11             swap A[k − 1] and A[k];
12             sorted ← false;
13     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	**Best:** $\Omega(n \log n)$, **Average:** $O(n \log n)$, **Worst:** $O(n^2)$	$O(\log n)$
CocktailSort	**Best:** $\Omega(n)$, **Average:** $O(n^2)$, **Worst:** $O(n^2)$	$O(1)$

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

Explanation:

I.QuickSort

Time Complexity

This QuickSort algorithm are recursive.To be convenient we define $T(n)$ the time complexity to sort a n-element array and we only consider the compare operation between $A[j]$ and *pivot*.

1) Best case: The best case appears when every time the pivot separates the array into equally-sized subarrays. Thus, in this case, the process of recursion consist of a **for** loop which has been executed for $n-1$ times and the recursion of two equall-sized subarrays which has the size of $\lfloor \frac{n}{2} \rfloor$.

Thus, we get:

$$T(n) = 2T(\lfloor \frac{n}{2} \rfloor) + n - 1$$

That means the QuickSort algorithm will separate the array for approximately $\log n$ times.Besides, when $n = 1$, there is no compare operation between $A[j]$ and *pivot* so that $T(1) = 0$.

Then, we sum it up and get:

$$T(n) = \sum_{i=0}^{\log n} 2^i \times \lfloor \frac{n}{2^i} \rfloor - \log n = (n-1)\log n = \Omega(n \log n)$$

So the time complexity of the best case is $\Omega(n \log n)$

(PS: We can also use Master Theorem to get the time complexity more easily. According to $T(n) = 2T(\lfloor \frac{n}{2} \rfloor) + n - 1$, by Master Theorem, we get $a = 2$, $b = 2$, $d = 1$, $d = \log_b a$ and thus the time complexity should be $\Omega(n \log n)$)

2) Worst case: The worst case happens when every time the pivot always divides the array into 1 and $n-1$ sized subarrays.Thus, in this case, the recursive process works like the double **for** loops and each level of recursion is composed of a **for** loop that works for $n - 1$ (There n means the size of the array) times.

So, we have:

$$T(n) = T(n-1) + T(1) + n - 1$$

This means the QuickSort algorithm will separate the array for n times and same as the best case, $T(1) = 0$.

Then, we get:

$$\begin{aligned} T(n) &= T(n-1) + T(1) + n - 1 \\ &= nT(1) + \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2} = O(n^2) \end{aligned}$$

Thus, the time complexity of the worst case is $O(n^2)$

3) Average case:

We consider the process of the QuickSort algorithm. When we start the recursion to sort a n-element array, we choose one element as the pivot, separate the array into two subarrays and then execute the recursion of these two subarrays.

Without loss of generality, we deliver this assumption that **all elements are equal to be picked up as the pivot in each level of recursion.**

Considering the recursion of the array with the length of n, we think that there are $x(x \in 0, 1, ..., n-1)$ elements smaller than the pivot and this array will be divide into a x-element part and a part with the size of $n - x - 1$.

Then, we have

$$T(n) = T(x) + T(n - x - 1) + n - 1$$

Besides, the probability $P(x = i) = \frac{1}{n}(i \in 0, 1, ..., n-1)$ and obviously $T(1) = 0$ as well as $T(0) = 0$.

Thus ,we can get this recurrence function after considering the probability:

$$\begin{aligned} T(n) &= n - 1 + \sum_{i=0}^{n-1} P(x = i) \times (T(i) + T(n - i - 1)) \\ &= n - 1 + \frac{2}{n} \sum_{i=0}^{n-1} T(i) \end{aligned}$$

So we can get:

$$nT(n) = n^2 - n + 2 \sum_{i=0}^{n-1} T(i) \tag{1}$$

$$(n+1)T(n+1) = (n+1)^2 - n - 1 + 2 \sum_{i=0}^{n} T(i) \tag{2}$$

After subtracting (2) by (1) and divding both sides of the equation, we have:

$$\begin{aligned} \frac{T(n+1)}{n+2} &= \frac{T(n)}{n+1} + \frac{2n}{(n+1)(n+2)} \\ \frac{T(n)}{n+1} &= \frac{T(n-1)}{n} + \frac{2(n-1)}{n(n+1)} \\ &\vdots \\ \frac{T(2)}{3} &= \frac{T(1)}{2} + \frac{2}{2 \times 3} \end{aligned}$$

Then, we sum them up and get (c is a non-zero constant):

$$\begin{aligned} \frac{T(n)}{n+1} &= \frac{T(1)}{2} + \sum_{i=1}^{n-1} \frac{2i}{(i+1)(i+2)} \\ T(n) &= (n+1)\frac{T(1)}{2} + (n+1) \sum_{i=1}^{n-1} \frac{2i}{(i+1)(i+2)} \\ &= (n+1) \sum_{i=1}^{n-1} (\frac{4}{i+2} - \frac{2}{i+1}) \\ &\xrightarrow{n \to \infty} (n+1) \times c \log n = O(n \log n) \end{aligned}$$

So the average time complexity is $O(n \log n)$

Space Complexity

Due to the recursive process of the QuickSort algorithm, its space complexity consists of the space cost in each level of the recursion.

In each depth of the recursion, the space cost cost only comes from the local variable and the maximum depth is $\log n$.

Thus, the space complexity is $O(\log n)$.

II.CocktaiSort

Time Complexity

Same as the QuickSort algorithm, we define $T(n)$ the time complexity to sort a n-element array and we only consider the compare operation between two elements.

1) Best case: The best case happens when the array have already been sorted. Then, in this case, the outer loop have only been executed for once and the time complexity just consists of two inner **for** loop.Thus we have

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

So the time complexity of the best case is $\Omega(n)$.

2) Worst case: The worst case appears when the array are in reverse order.Thus, in this case, the outer loop will be executed until $i = j$ and every time the outer loop is executed, $i = i+1, j = j-1$ have happened and two inner **for** loops have been executed for totally $2n-3$ times ($n = j - i + 1$ before two **for** loops). Thus we have:

$$T(n) = T(n-2) + \max(0, n-1) + \max(0, n-2)$$

Besides, obviously, $T(1) = 0$ and then we can get:

$$T(n) = T(n-1) + \max(0, n-1)$$
$$T(n) = T(1) + \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2} = O(n^2)$$

Thus, the time complexity of the worst time is $O(n^2)$

3) Average case:

To begin with, we introduce the concept of Inversion Pair firstly. The inversion pair $< A_i, A_j >$ is defined below and we regard $C(n, A)$ as the number of inversion pairs in the array A.

$$< A_i, A_j > \quad iff \quad (A_i > A_j) \wedge (i < j)$$

Then we find that the swap operation only happens between two nearby elements so after the operation, $C(n, A') = C(n, A) - 1$ (A' means the array A after the operation).

Thus, we can have this obvious outcome:

$$T(n) \geq C(n, A)$$

Then we calculate the accurate number of $C(n, A)$.

Before that, we deliver the assumption that the elements in A is $1, 2, ..., n$, and the probability of each of the $n!$ arranges$(A_1, A_2, ..., A_{n!})$ is equal.

$$P(A = A_i) = \frac{1}{n!} \quad (i = 1, 2, ...n!)$$

So, we can get:

$$C(n, A) = \sum_{i=1}^{n!} P(A = A_i) C(n, A_i) = \frac{1}{n!} \sum_{i=1}^{n!} C(n, A_i)$$

But when we think in another perspective, we have this equation which means we consider the contribution to $C(n, A)$ one exact inversion pair in the array A:

$$\sum_{i=1}^{n!} C(n, A_i) = \sum_{\substack{i,j \in \{1,2,...,n\} \\ i>j}} C_n^2 (n-2)!$$

Thus, we have

$$C(n,A) = \frac{1}{n!}\sum_{i=1}^{n!} C(n,A_i) = \frac{1}{n!}\sum_{\substack{i,j\in\{1,2,\ldots,n\}\\ i>j}} C_n^2(n-2)! = \frac{1}{n!}\sum_{\substack{i,j\in\{1,2,\ldots,n\}\\ i>j}} \frac{n!}{2} = \frac{n(n-1)}{2} = O(n^2)$$

Besides, it's obvious thar $T(n)$ of the average case is small than $T(n)$ of the worst case. So, we get $T(n) \leq O(n^2)$

Thus, all in all, we have:

$$O(n^2) = C(n,A) \leq T(n) \leq O(n^2)$$

So, the average time complexity is $O(n^2)$.

Space Complexity

The CocktailSort algorithm doesn't have any recursive process and the space cost only results from the local variable.Thus, the space complexity is $O(1)$

(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. The worst case results from the worst choice of pivot in each level of recursion. So, in order to optimize the algorithm, we should have some better policies to choose the pivot.

Here are some approaches to modify the algorithm.

1)Randomized QuickSort

Considering the process of calculating the average time complexity, we transform this computational process into practice and get this pseudocode below:

Algorithm 3: Randomized QuikSort

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

```
1  p ← Rand(1, n);
2  swap(A[n], A[p]);
3  pivot ← A[n]; i ← 1;
4  for j ← 1 to n − 1 do
5      if A[j] < pivot then
6          swap A[i] and A[j];
7          i ← i + 1;
8  swap A[i] and A[p];
9  if i > 1 then QuickSort(A[1, ··· , i − 1]);
10 if i < n then QuickSort(A[i + 1, ··· , n]);
```

In this pseudocode, the function **Rand** means the random function which can generate a random integer between a and b. With this function, we can achieve the same expected performance as the average in the worst case theoretically.

2)Median-of-three policy

Although we have the optimized Random QuickSort, we still have some algorithm working better. The Median-of-three policy is one of them and it's based on the target to choose the median as the pivot. In this policy, we regard the median in the three element:$A[1], A[\lceil\frac{1+n}{2}\rceil], A[n]$ as the median of the whole array and it works better in practical tests. Here is the pseudocode.

Algorithm 4: Median-of-three QuikSort

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

```
1  Median3(A,1,n);
2  p ← A[n];
3  pivot ← A[n]; i ← 1;
4  for j ← a to n − 1 do
5      if A[j] < pivot then
6          swap A[i] and A[j];
7          i ← i + 1;
8  swap A[i] and A[p];
9  if i > 1 then
       QuickSort(A[1, ··· , i − 1]);
10 if i < n then
       QuickSort(A[i + 1, ··· , n]);
```

Algorithm 5: Median3

Input: An array $A[low, \cdots, high]$, two integer low and high
Output: $A[low, \cdots, high]$ whose median of three is $A[high]$

```
1  mid ← ⌈(low + high)/2⌉;
2  if A[high] > A[mid] then
3      swap(A[mid], A[high]);
4  if A[mid] > A[low] then
5      swap(A[mid], A[low]);
6  if A[low] > A[high] then
7      swap(A[low], A[high]);
```

Actually, to get the better outcome, we can divide the array into m parts (often $m = \sqrt{n}$) called array $B_i (i \in 1, 2, ..., m)$ and use the Median-of -three policy to get the median of each part called Bm_i. Then Bm_i consists of a new array and we use this policy again and choose the median as the final median of the whole array A. Then the Median-of-three QuickSort will be executed. □

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

Explanation:

This explanation consist of two parts.

1) The arrangement of functions

In this part, we have:

$$\lim_{n\to\infty}\frac{\log(\log n)}{1}=\infty,\quad \lim_{n\to\infty}\frac{\log_4 n}{\log(\log n)}=\lim_{n\to\infty}\frac{\frac{1}{n\ln 4}}{\frac{1}{n\ln n}}=\infty,\quad \lim_{n\to\infty}\frac{n^{\frac{1}{2}}}{\log n}=\lim_{n\to\infty}\frac{\frac{1}{2}n^{-\frac{1}{2}}}{n^{-1}}=\infty$$

$$\lim_{n\to\infty}\frac{n^{\frac{1}{2}}}{\log n}=\lim_{n\to\infty}\frac{\frac{1}{2}n^{-\frac{1}{2}}}{n^{-1}}=\infty,\quad \lim_{n\to\infty}\frac{n}{2^{\log n}}=\lim_{n\to\infty}2^{\frac{\log_2 n}{\log_2 e}}=\lim_{n\to\infty}\frac{n}{n^{\log 2}}=\lim_{n\to\infty}n^{1-\log 2}=\infty$$

$$\lim_{n\to\infty}\frac{n^2}{\log(n!)}=\lim_{n\to\infty}\frac{n^2}{\sum_{i=1}^{n}\log i}\geq\lim_{n\to\infty}\frac{n^2}{\int_1^{n+1}\log x dx}=\lim_{n\to\infty}\frac{n^2}{(n+1)log(n+1)-n}=\infty$$

$$\lim_{n\to\infty}\frac{n\log n}{n}=\lim_{n\to\infty}\log n=\infty,\quad \lim_{n\to\infty}\frac{2^n}{n^2}=\infty,\quad \lim_{n\to\infty}\frac{4^n}{2^n}=\lim_{n\to\infty}2^n=\infty$$

$$\lim_{n\to\infty}\frac{n!}{4^n}=\infty,\quad \lim_{n\to\infty}\frac{(2n)!}{n!}=\infty,\quad \lim_{n\to\infty}\frac{2^{2^n}}{(2n)!}=\infty$$

2) The equivalence classes among the functions

To find some equivalence classes, we have:

$$\lim_{n\to\infty}\frac{\log n}{\log_4 n}=\lim_{n\to\infty}\frac{\frac{1}{n}}{\frac{1}{n\ln 4}}=\ln 4$$

$$\lim_{n\to\infty}\frac{log(n!)}{n\log n}=\lim_{n\to\infty}\frac{\sum_{i=1}^{n}\log i}{n\log n}=\lim_{n\to\infty}\frac{log(n+1)}{(n+1)\log(n+1)-n\log n}$$

$$=\lim_{n\to\infty}\frac{\frac{1}{n+1}}{\log(n+1)-\log n}=\lim_{n\to\infty}\frac{\frac{1}{(n+1)^2}}{\frac{1}{(n+1)n}}=1$$

All in all we get the final outcome. □

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