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: Qinwei Yang Student ID: 519030910145 Email: yinwai@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	$\Omega(n \log n), \mathrm{O}(n \log n), \mathrm{O}(n^2)$	$\Omega(\log n), \mathrm{O}(\log n), \mathrm{O}(n)$
CocktailSort	$\Omega(n), \mathrm{O}(n^2), \mathrm{O}(n^2)$	$\Theta(1), \Theta(1), \Theta(1)$

[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.

The Analysis of Quicksort

In *QuickSort*, we set $f(r, n)$ and $g(r, n)$ respectively as the number of computations and space consumption, when there are n items in the array and A[n] is the r-th one of the array.

Also, we set $r_m(n)$ and $r_M(n)$ respectively as the r which make $f(r, n)$ reach the minimum and maximum, then we have:

$$f_m(n) = f(r_m(n), n), \quad f_M(n) = f(r_M(n), n)$$

Firstly let's talk about the best case. We have:

$$f_m(n) = f_m(r_m(n) - 1) + f_m(n - r_m(n)) + n$$

$$f_m(n) = f_m(r_m(n) - 1) + (r_m(n) - 1) + f_m(n - r_m(n)) + (n - r_m(n)) + 1$$

And we can draw a tree of recursion:

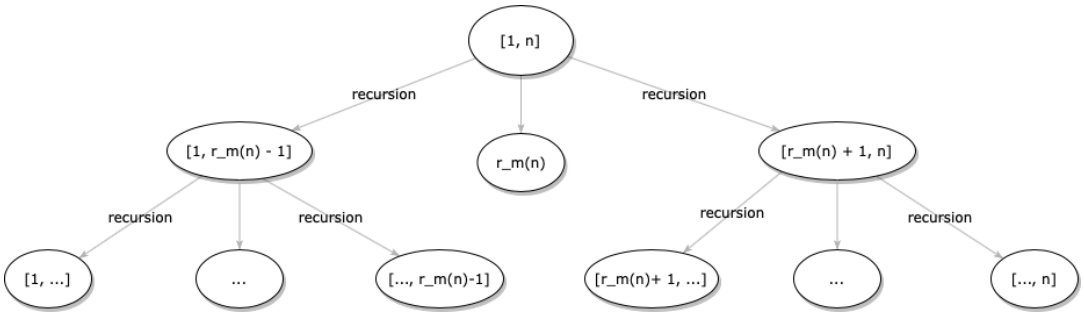

Figure 1: Australian wildfire scene

If we set $h(i)$ as the height of A[i] in the tree, then we can conclude that:

$$f_m(n) = \min(\sum_{i=1}^{n} h(i))$$

Using mathematical induction, we can easily prove that the best case is obtained when **the recursive tree is a balanced binary tree** (without considering the leaf nodes). So we have:

$$r_m(n) = \left\lfloor \frac{n}{2} \right\rfloor$$

Similarly, the worst case is taken when **the recursive tree degenerates to a straight chain** (without considering the leaf nodes). So we have:

$$r_M(n) = 1 \;||\; r_M(n) = n$$

Then we can analyze the time complexity:

$$f_m(n) = 2f_m(\frac{n}{2}) + n$$

$$\frac{f_m(n)}{n} = \frac{f_m(\frac{n}{2})}{\frac{n}{2}} + 1 = \log_2 n, \quad f_m(n) = n \log_2 n$$

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

$$f_M(n) = f_M(n-1) + n = \sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

So in the worst case the time complexity is $\mathrm{O}(n^2)$.

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

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

$$nE(f(n)) - (n-1)E(f(n-1)) = 2n - 1 + 2E(f(n-1))$$

$$\frac{E(f(n))}{n+1} - \frac{E(f(n-1))}{n} = (2n-1)(\frac{1}{n} - \frac{1}{n+1})$$

$$\frac{E(f(n))}{n+1} = \sum_{i=1}^{n} \frac{2n-1}{n} - \sum_{i=2}^{n+1} \frac{2n-3}{n} = 1 - \frac{2n-3}{n+1} + 2(\frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n}) = \mathrm{O}(\log n)$$

So in the average case the time complexity is $\mathrm{O}(n \log n)$.

And we can analyze the space complexity

$$g_m(n) = g_m(\frac{n}{2}) + 1, \quad g_m(n) = \log_2 n$$

So in the best case the space complexity is $\Omega(\log n)$.

$$g_M(n) = g_M(n-1) + 1 = \sum_{i=1}^{n} 1 = n$$

So in the worst case the space complexity is $\mathrm{O}(n)$.

$$E(g(n)) = 1 + \frac{1}{n}\sum_{i=0}^{n-1} \max\{E(g(i)), E(g(n-i-1))\} \approx 1 + \frac{1}{n}\sum_{i=0}^{n-1} \frac{E(g(i)) + E(g(n-i-1))}{2}$$

$$E(g(n)) \approx 1 + \frac{1}{n}\sum_{i=0}^{n-1} E(g(i))$$

$$nE(g(n)) \approx n + \sum_{i=0}^{n-1} E(g(i))$$

$$(n-1)E(g(n-1)) \approx (n-1) + \sum_{i=0}^{n-2} E(g(i))$$

$$nE(g(n)) - (n-1)E(g(n-1)) \approx 1 + E(g(n-1))$$

$$E(g(n)) - E(g(n-1)) \approx \frac{1}{n}$$

$$E(g(n)) \approx \sum_{i=1}^{n} \frac{1}{n} = \mathrm{O}(\log n)$$

So in the average case the space complexity is $\mathrm{O}(\log n)$.

The Analysis of CocktailSort

Since the while loop is executed at least 1 time, in which the for loop is executed $n-1$ times, so in the best case the time complexity is $\Omega(n)$, which can happen when **the input array is already sorted**.

The while loop is executed at most $\frac{n}{2}$ times, which can happen when **the input array is sorted reversely**, and the for loop is executed sequently $n-1, n-2, \ldots$ times. So we can find that the worst case's time complexity is $\mathrm{O}(n^2)$.

Then we can examine the lower bound of the time complexity under average conditions. In order to facilitate the calculation, we first calculate the number of exchanges among them.

We can examine the changes in the array state brought about by the exchange and introduce the concept of inverse ordinal numbers.

Definition 1. ***Reverse pair*** *is a binary subsequence of an array which is sorted reversely.* ***Inverse ordinal numbers*** *is the number of an array's reverse pairs.*

So obviously, each exchange in the sorting exactly reduces the inverse ordinal number of the array by one.

For each possible array, we construct an array that reverses it. Obviously, the original array and the reversed array have a one-to-one correspondence. The reverse pair of the original array and the forward pair of the reverse array, the forward pair of the original array and the reverse pair of the reverse array are all in one-to-one correspondence.

So the average inverse ordinal number of a n-element array is $\frac{1}{2}C_n^2 = \frac{n(n-1)}{4}$, which means the time complexity of exchanges is $\Omega(n^2)$, and the time complexity under average conditions is also $\Omega(n^2)$.

And according to the time complexity under the worst condition, the average time complexity is $\mathrm{O}(n^2)$.

Since the algorithm only use three variables, the space complexity is $\Theta(1)$.

Modify the QuickSort algorithm

Initial position randomization is a way to optimize quick sort, but it just bypasses the worst-case scenario. It is difficult to optimize the worst case to $\mathrm{O}(n \log n)$ on the basis of quick sort, but we can think in the opposite direction and change the average complexity to $\Theta(n^2)$.

In order to ensure that we can always get the worst case, we can traverse the array once and find the minimum value of the array within the time of $\Theta(n)$. Next, we also use the previous sorting method, but replace A[n] with the minimum value we found.

Algorithm 3: ModifiedQuickSort

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

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

□

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 $\ln n$.

$$\begin{array}{ccccc} 1 & n & \log n & \log(\log n) & n \log n \\ \log_4 n & 2^n & 4^n & 2^{\log n} & 2^{2^n} \\ \log(n!) & n! & (2n)! & n^{1/2} & n^2 \end{array}$$

Solution.

$$\begin{array}{ccccccccccc} 1 & \prec & \log(\log n) & \prec & \log_4 n & = & \log n & \prec & n^{\frac{1}{2}} & \prec \\ 2^{\log n} & \prec & n & \prec & \log(n!) & = & n \log n & \prec & n^2 & \prec \\ 2^n & \prec & 4^n & \prec & n! & \prec & (2n)! & \prec & 2^{2^n} & \end{array}$$

The previous relationship is mathematically obvious, so we only prove $\log(n!) = n \log n$ and $(2n)! \prec 2^{2^n}$.

Comparing n^n, $(2n)!$ and $(2n)^{2n}$, we have:

$$n^n < (n+1) \cdot (n+2) \cdot \cdots \cdot 2n < (2n)! < (2n)^{2n}$$

So we have:

$$(\frac{n}{2})^{\frac{n}{2}} < n! < n^n$$

$$\frac{n}{4} \log n = \frac{n}{2} \log \sqrt{n} \leq \frac{n}{2} \log \frac{n}{2} \leq \log(n!) \leq n \log n$$

And we get that $\log(n!) = n \log n$.

Firstly, let's check the relationship between $(2n)!$ and $(2n)^{2n}$. We have:

$$0 \leq \lim_{n\to\infty} \frac{(2n)!}{(2n)^{2n}} \leq \lim_{n\to\infty} \frac{(2n)^{2n-1}}{(2n)^{2n}} = \lim_{n\to\infty} \frac{1}{n} = 0, \quad \lim_{n\to\infty} \frac{(2n)!}{(2n)^{2n}} = 0$$

So $(2n)! \prec (2n)^{2n}$. And we have:

$$0 \leq \lim_{n\to\infty} \frac{(2n)^{2n}}{2^{2^n}} = \lim_{n\to\infty} e^{2n \ln 2n - 2^n \ln 2} < \lim_{n\to\infty} e^{0.1 \cdot 2^n - 2^n \ln 2} < \lim_{n\to\infty} e^{-0.1 \cdot 2^n} = 0$$

$$\lim_{n\to\infty} \frac{(2n)^{2n}}{2^{2^n}} = 0$$

So $(2n)! \prec (2n)^{2n} \prec 2^{2^n}$. □

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