Lab01-AlgorithmAnalysis

CS214-Algorithm and Complexity, Xiaofeng Gao, Spring 2020.

∗ If there is any problem, please contact TA Shuodian Yu.
∗ Name:Haotian Xue Student ID: 518021910506 Email: xavihart@sjtu.edu.cn

1. Please analyze the time complexity of Alg. 1 with brief explanations.

Algorithm 1: PSUM

Input: $n = k^2$, k is a positive integer.
Output: $\sum_{i=1}^{j} i$ for each perfect square j between 1 and n.

```
1 k ← √n;
2 for j ← 1 to k do
3 |   sum[j] ← 0;
4 |   for i ← 1 to j^2 do
5 |   |   sum[j] ← sum[j] + i;
6 return sum[1 ··· k];
```

Solution. In the first **for** loop will be executed for k times, and for each j in the first loop, the inner loop will be executed for j^2 times, so the iteration number should be:

$$\sum_{j=1}^{k} j^2 = \frac{k(k+1)(2k+1)}{6} = \frac{\sqrt{n}(\sqrt{n}+1)(2\sqrt{n}+1)}{6}$$

So the time complexity is $\Theta(n^{\frac{3}{2}})$.

□

2. Analyze the **average** time complexity of QuickSort in Alg. 2.

Algorithm 2: QuickSort

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

```
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]);
```

Solution. As we can see from the pseudocode, the QuickSort Algorithm are executed recursively: suppose we aim to sort a array with a length of n, firstly we pick up one element as pivot, sencondly we divide the array by the pivot, lastly we execute the recursion.

We define the average time complexity to sort a n-element array $T(n)$, which is composed of two parts: the dividing process and the recursion process.

$$T(n) = T_{div}(n) + T_{rec}(n)$$

As the dividing process is a **for** loop which has been executed for $n-1$ times(if we only consider the swap operation). So

$$T_{div}(n) = n$$

To get $T_{rec}(n)$, we think about the pivot $A[n]$, if there are j ($j \in \{0, 1, 2, ..., n-1\}$) elements which are smaller than $A[n]$, then the recursion part $T_{rec}(n)$ should be $T_{rec}(j)+T_{rec}(n-j-1)$.

Suppose the probability $P(j = m) = \frac{1}{n}(m = 0, 1, ..., n-1)$.

Then we can get:

$$T(n) = n + \sum_{j=0}^{n-1} \frac{1}{n}(T(j) + T(n-j-1))$$

So we can get $T(n+1)$:

$$nT(n) = n^2 + 2(T(0) + T(1) + ...T(n-1))$$
$$(n+1)T(n+1) = (n+1)^2 + 2(T(0) + T(1) + ... + T(n))$$

After subtracting $T(n+1)$ by $T(n)$ and dividing the both side of the equation by $(n+2)(n+1)$:

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

$$\frac{T(n)}{n+1} = \sum_{j=1}^{n-1} \frac{2(j)}{(j+1)(j+2)} + T(1)$$

the right side is fitting into $O(ln(n))$ when n is big enough.(easy to prove)

So the average time complexity is $O(nlogn)$.

□

3. The BubbleSort mentioned in class can be improved by stopping in time if there are no swaps during an iteration. An indicator is used thereby to check whether the array is already sorted. Analyze the **average** and **best** time complexity of the improved BubbleSort in Alg. 3.

Algorithm 3: BubbleSort

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

```
1 i ← 1;sorted ← false;
2 while i ≤ n − 1 and not sorted do
3 |   sorted ← true;
4 |   for j ← n downto i + 1 do
5 |   |   if A[j] < A[j − 1] then
6 |   |   |   interchange A[j] and A[j − 1];
7 |   |   |   sorted ← false;
8 |   i ← i + 1;
```

Solution. It is obvious that the **best case** is $O(n)$: in the first loop of **while**, no interchange happened and the indicator is set to true, so the program just break out of the loop.

To calculate the **Average Time Complexity**, firstly I will use a strict way to the get the actually swap times in the optimized BubbleSort.

First we define the inversion pair $< A_i, A_j >$ and the inversion pair numbers $N(A)$ in a array:

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

$$N(A) = \left| \{ < i, j > | InversionPair < A_i, A_j > \} \right|$$

If we regard the swap times in array A is $T_s(n, A)$ and the compare times $T_c(n, A)$ in the process of algorithm, we can regard $T_c(n)$ as the deciding part because it is obvious that:

$$T_s(n, A) < T_c(n, A)$$

Then let us make a accurate calculation of $T_s(n)$: the happening of each swap operation make the InversionPair number $N(A)$ subtract by 1, so actually for every array $A[1, ..., n]$, the swapping times is equal to the number of inversion pairs, that is:

$$T_s(n, A) = N(A)$$

Without loss of generality, we make 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 the same.

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

So the time complexity for swap operations is $T_s(n)$:

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

actually the sum equation can be view from another angle:

$$\sum_{i=1}^{n!} N(A_i) = \sum_{\substack{i,j \in \{1,2,...,n\} \\ i>j}} M(< i, j >)$$

$M < i, j >$ is the number of pair $< i, j >$ appearing in the $n!$ arranges. So it is obvious that:

$$M(< i, j >) = C_n^2 (n-2)! = \frac{n!}{2}$$

$$T_s(n) = \frac{1}{n!} C_n^2 M(< i, j >) = \frac{1}{n!} \cdot \frac{n! n(n-1)}{2} = \frac{n(n-1)}{2} = O(n^2)$$

Then worst case is $O(n^2)$, so the final Average Time Complexity $T(n)$can be calculated by:

$$O(n^2) = T_s(n) \leq T(n) = T_c(n) \leq WorstCase = O(n^2)$$

So we get the Average Time Complexity $T(n) = O(n^2)$.

□

4. 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.

$2^{\lg n}$	$(\lg n)^{\lg n}$	n^2	$n!$	$(n+1)!$
2^n	n^3	$\lg^2 n$	e^n	2^{2^n}
$\lg\lg n$	$n \cdot 2^n$	n	$\lg n$	$4^{\lg n}$

Solution.

$$\lg\lg n \prec \lg n \prec (\lg n)^2 \prec n = 2^{\lg n} \prec n^2 = 4^{\lg n} \prec n^3$$
$$\prec \lg n^{\lg n} \prec 2^n \prec n2^n \prec e^n \prec n! \prec (n+1)! \prec 2^{2^n}$$

Explanations: In this part I will give some explanations of the above answers which are not that obvious.

1) $n^3 \prec \lg n^{\lg n} \prec 2^n$:

$$\lim_{n\to\infty} \lg \frac{\lg n^{\lg n}}{2^n} = \lim_{n\to\infty} (\lg n \lg\lg n - n) = -\infty$$
$$\lim_{n\to\infty} \lg \frac{\lg n^{\lg n}}{n^3} = \lim_{n\to\infty} \lg n(\lg\lg n - 3) = +\infty$$

2) $(n+1)! \prec 2^{2^n}$:

let $R(n) = \frac{(n+1)!}{2^{2^n}}$, then calculate $\frac{R(n+1)}{R(n)}$:

$$\frac{R(n+1)}{R(n)} = \frac{n}{2} \to +\infty$$

So $R(n) \to +\infty$.

3) $e^n \prec n!$:

According the Stirling's approximation:

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

so it is obvious that $n!$ much bigger than 2^n when n is great enough.

□

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