cpt_s 350

Homework 3

11641327 Yu-Chieh Wang

2020/2/20

1. Given: An array A of n elements.

   Question: Find an algorithm can select both maximal and minimal elements by running at most 1.5 n comparisons.

A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]	A[8]	A[9]	A[10]	A[11]	A[12]	…	A[n]

Solution 1:

If n is less than 3, use quick sort on the array A. Then, pick the maximal and minimal elements directly. The sorting time of quick sort is $nlogn$, which $log_2 n \leq 1.5$ when n is less than 3.

Solution 2:

Bigger value:	A[1]	A[4]	A[5]	A[7]	A[10]	A[11]		A[n-1]
	|	|	|	|	|	|	…	|
Smeller value:	A[2]	A[3]	A[6]	A[8]	A[9]	A[12]		A[n]

First, We make every two elements a smell group, and do one comparison in each group. we will have 0.5n groups, so it's going to run 0.5n comparisons. Then, we make two large groups: One to combine the bigger value in each smell group. The other one combine the smeller value. Now, each large group has 0.5 n elements. Finally, when we do select number, we only compare to a large group (0.5n comparisons). As a result, we spend 0.5n comparisons on separating an array to two groups, 0.5n comparisons on selecting the largest element, and 0.5n comparisons on selecting the minimal element, which is totally 1.5n comparisons.

2. Compare the average-case complexities of two algorithm.

   Consider to the times we select numbers C, we can show the difference between two algorithms S and T. The running time complexities are as follow:

   $T_S = C \cdot n$ (Every time we select, we need to run in linear time again.)

   $T_T = nlogn + C$ (We sort the array first, and then pick the number directly.)

   To compare the two algorithm, we get the following equation:

   $nlogn + C \leq C \cdot n$, which means that T_T spend less time than T_S for some C.

   From the equation, we can see that, if we want to select more times, T_T is better than T_S because when the number of select times C is up to n, the time complexity of S will become n^2. However, if we only select few times, T_S is better than T_T because it spends too much time on sorting.

3. The worst-case complexities for k=3 and k=7.

   k=3:

   Step 1: cut an array $A[1,\ldots,n]$ into 3/n groups: each has 3 numbers.

   Step 2: sort each group (sort 3 numbers).

   Step 3: each group has a median number. We have totally n/3 medians.

   Run linear-select (recursively) to select the n/6-th smelliest from the n/3 medians; the element selected is called MM.

   Step 4: We go back to the original input array $A[1,\ldots,n]$, and recall that MM is an element in the array swap MM with the first element in A (as a result, A has MM as its first element).

Step 5: Run partition on the A.

low part **(r-1)**	**r**	**high part** **(n-r)**

Step 6: if i==r, return A[r]

If i<r, run linear select on the Low to select i-th smallest.

If i>r, run linear select on the High to select (i-r)-th smallest.

Notice that, we have n/3 medians, n/6 medians that are $\leq$ the MM, and 2 elements in the group that the median belongs that are $\leq$ the medians, so in the original A, we have at least $2 \cdot (n/6)$ numbers that $\leq$ MM. Finally, we know the numbers in the low parts and high part are totally $\leq$ n, and we get at most $2 \cdot (n/6)$ numbers in the low part, so we have at most $4 \cdot (n/6)$ numbers in the high part.

As a result, we count the time complexity as follow:

We spend O(n) on step 2 to sort each group, spend $T(n/3)$ on step 3 to select the medians, and run $T(2n/6)$ on the low part to select the minimal number or $T(4n/6)$ on the high part to select the maximal number. As a result, we count the worst-case by O(n) + $T(n/3)$ + $T(4n/6)$. Assume the answer is linear time, we need to prove the following equation: O(n) + $T(2n/6)$ + $T(4n/6) \leq$ O(n).

$\rightarrow a \cdot n + c \cdot (n/3) + c \cdot (4n/6) \leq c \cdot n$

$\rightarrow a \cdot n + c \cdot n \leq c \cdot n$ where $c >> a$.

k=7

Notice that, we have n/7 medians, n/14 medians that are $\leq$ the MM, and 4 elements in the group that the median belongs that are $\leq$ the medians, so in the original A, we have at least $4 \cdot (n/14)$ numbers that $\leq$ MM. Finally, we know the numbers in the low parts and high part are totally $\leq$ n, and we get at most $4 \cdot (n/14)$ numbers in the low part, so we have at most $10 \cdot (n/14)$ numbers in the high part.

As a result, we count the time complexity as follow:

We spend O(n) on step 2 to sort each group, spend $T(n/7)$ on step 3 to select the medians, and run $T(4n/14)$ on the low part to select the minimal number or $T(10n/14)$ on the high part to select the maximal number. As a result, we count the worst-case by O(n) + $T(n/7)$ + $T(10n/14)$. Assume the answer is linear time, we need to prove the following equation: O(n) + $T(n/7)$ + $T(10n/14)$ $\leq$ O(n).

$\rightarrow a \cdot n + c \cdot (n/7) + c \cdot (10n/14) \leq c \cdot n$

$\rightarrow a \cdot n + (6/7)c \cdot n \leq c \cdot n$ where $c >> a$.

4. Given: an algorithm called ilselect(A, n, i).

   Question: Find the worst-case and the average-case complexities.

   We learn the time complexities of following algorithms, so we can do it directly.

	Best-case	**Worst-case**	**Average-case**
Quickselect	$O(1)$	$O(n^2)$	$O(n)$
Linearselect	$O(1)$	$O(n)$	$O(n)$

First of all, we separate the process of ilselect to three steps:

(1) Do partition.

(2) Run quickselect on the low part.

(3) Run linearselect on the hight part.

low part **(r-1)**	**r**	**high part** **(n-r)**

Then, we count the complexities by these steps:

(a) The worst-case complexities:

(1) $T_{WORST}(n) = O(n)$

(2) $T_{WORST}(n) = \min_{1 \le r \le n} \{T_{WORST}(r-1)\} = O(r^2)$

(3) $T_{WORST}(n) = \min_{1 \le r \le n} \{T_{WORST}(n-r)\} = O((n-r)^2)$

$$T_{WORST}(n) = O(n) + O(r^2) + O((n-r)^2) = O(n^2)$$

(b) The average-case complexities:

(1) $T_{AVG}(n) = O(n)$

(2) $T_{AVG}(n) = \frac{1}{n}\sum_{r=1}^{n} \{T_{AVG}(r-1)\} = O(r)$

(3) $T_{AVG}(n) = \frac{1}{n}\sum_{r=1}^{n} \{T_{AVG}(n-r)\} = O(n-r)$

$$T_{AVG}(n) = O(n) + O(r) + O(n-r))\} = O(n)$$

5. 5com

Before the array of n numbers is sorted, the amount of information on the ordering is $log_2 n!$ bits (since we have n! orderings of the n numbers). Each use of 5com decrease the amount by at most $log_2 5!$ bits. After the array is sorted, the amount of information on the ordering is zero. Therefore, the number of 5 com uses is at least $(log_2 n! / log_2 5!)$