Order Statistics

CS 5633 Analysis of Algorithms

Computer Science
University of Texas at San Antonio

October 2, 2024

Order Statistics with Divide and Conquer

Order Statistics

- Suppose we are given an array of n distinct elements. The ith **order statistic** is the ith smallest element.
  - $i = 1$: minimum element
  - $i = n$: maximum element
  - $i = \lfloor (n+1)/2 \rfloor$: median
- Trivial algorithm: Sort the input using merge sort. Return the element at index i. Worst case running time is $\Theta(n \log n)$.
- Can we develop an algorithm which has a better worst-case running time than the trivial algorithm?

A Divide and Conquer Solution

- Use a randomized partitioning scheme similar to the one used in quicksort (choose a pivot and partition the elements into two sets "around" the pivot).
- Let k be the index of the pivot after partitioning.
- If $i = k$, then the pivot is the element we are looking for, and we return the pivot.
- If $i < k$, then we know that the element we are looking for is in the first subarray, and we recursively find the ith smallest element in this subarray.
- If $i > k$, then we now that the element we are looking for is in the second subarray, and we recursively find the element in position $i - k$ in this subarray.

A Divide and Conquer Solution cont.

- The
  pseudo-code of finding the i'th element in array $A[p : r]$:

Algorithm 1: DnD Order Statistics

```
Function Randomized_Select(A, p, r, i)
    if p == r then
        return A[p];// base case;
    q = Random_Partition(A, p, r);
    // k represents the pivot item's position after partition.;
    k = q − p + 1;
    if k == i then
        return A[q]; // we found the i'th item;
    else if i < k then
        return Randomized_Select(A, p, q − 1, i); // search in the first
          half;
    else
        return Randomized_Select(A, q + 1, r, i − k); // search in the first
          half;
```

Worst Case Run Time of Randomized_Select

- The non-recursive cost of each invocation,
  - For each invocation, line 4 always requires $\Theta(n)$ or $c \cdot n$ time to execute.
- The recursive cost depends on the size of the sub-array at either line 10 or 12.
- In the worst case, we always select the smallest or largest item as the pivot.
  - And, we always have to find the target in the $n-1$ array.
  - That is, the execution time is, $T(n) = T(n-1) + c \cdot n$.
  - It is fairly easy to solve this equation with recursive tree, or expanding the equation or induction, which all give $T(n) = \Theta(n^2)$.

Typical Run Time of Randomized_Select

- If we choose a good pivot whose largest subarray is of size at most $9n/10$:
  - The run time is $T(n) \leq T(9n/10) + c \cdot n$
  - This equation can be easily solved with master theorem case 3.
  - The final run time is then $T(n) = \Theta(n)$.
- This example shows that we can expect this algorithm to have a linear run time.

Expected Average Run Time of Randomized_Select

- The running time will depend on the sizes of the subproblems we generate. The two subarrays computed by partition will be of size $(k-1, n-k)$ for some $k \in \{1, \ldots n\}$.
  - Either line 10 or line 12 of Randomized_Sort is executed.
  - Therefore,the sub-problem will have a size of $k-1$ (line 10), or a size of $n-k$ (line 12).
- To obtain an upper bound on the running time, we will assume that the ith element always falls in the larger subarray (i.e. the subproblem size will be $\max(k, n-k)$).
- Thus we can express the running time of the algorithm in the following way:
  - $T(n) = T(max(k-1, n-k)) + c \cdot n$.

Expected Average Run Time of Randomized_Select cont.

- Thus we can express the running time of the algorithm in the following way:
  - $T(n) = T(max(k-1, n-k)) + c \cdot n$.
  - The problem is, however, k could be an value between 1 and n.
- Let's define an indicator random variable to handle k.

$$I_k = \begin{cases} 1, & \text{if pivot is at } k \\ 0, & \text{otherwise} \end{cases}$$

- Note that, based on the slides from randomized algorithms, we have $E(I_k) = Pr(I_k = 1) = Pr(\text{the pivot is } k) = \frac{1}{n}$.
  - $Pr(\text{the pivot is } k) = \frac{1}{n}$, as each item has an even chance becoming the pivot.

Expected Average Run Time of Randomized_Select cont.

- The run-time can then be expressed as,
  $$T(n) = \sum_{k=1}^{n} I_k \cdot T(max(k-1, n-k)) + c \cdot.$$
- The expected run time of Randomized_Select, $E(T(n))$, is then,

$$\begin{aligned} E(T(n)) &= E(\sum_{k=1}^{n} I_k \cdot T(max(k-1, n-k)) + c \cdot n) \\ &= E(\sum_{k=1}^{n} I_k \cdot T(max(k-1, n-k))) + c \cdot n \\ &= \sum_{k=1}^{n} E(I_k) \cdot E(T(max(k-1, n-k))) + c \cdot n \\ &= \sum_{k=1}^{n} \frac{1}{n} \cdot E(T(max(k-1, n-k))) + c \cdot n \end{aligned} \tag{1}$$

Expected Average Run Time of Randomized_Select cont.

- Next, we need to treat $max(k-1, n-k)$.
  - if $k > \lceil \frac{n}{2} \rceil$, $max(k-1, n-k) = k-1$
  - if $k \leqslant \lceil \frac{n}{2} \rceil$, $max(k-1, n-k) = n-k$
  - Considering we are just partitioning the array, the above two cases are actually equivalent.
  - Therefore, we can use the first case to estimate the second case.
- If only the first case is considered,

$$
\begin{aligned}
E(T(n)) &= \sum_{k=1}^{n} \frac{1}{n} \cdot E(T(max(k-1, n-k))) + c \cdot n \\
&= \sum_{k=\lfloor \frac{n}{2} \rfloor}^{n} 2 \cdot \frac{1}{n} \cdot E(T(k-1)) + c \cdot n \\
&= \sum_{k=\lfloor \frac{n}{2} \rfloor}^{n-1} 2 \cdot \frac{1}{n} \cdot E(T(k)) + c \cdot n
\end{aligned} \tag{2}
$$

Expected Average Run Time of Randomized_Select cont.

- Now $E(T(n))$ is represented with an equation that does not require any special operators.
- We can now prove $E(T(n)) = O(n)$ with induction.
  - Clearly, when $n = 2$, $E(T(2)) = T(1) + 2 \cdot c <= d \cdot 2 (d > 4c)$
  - Assume $E(T(k)) = O(k)$ for any $k < n$. For $E(T(n))$, we have,

$$
\begin{aligned}
E(T(n)) &= \sum_{k=\lfloor \frac{n}{2} \rfloor}^{n-1} \frac{2}{n} \cdot E(T(k)) + c \cdot n \leqslant \frac{2}{n} \sum_{k=\lfloor \frac{n}{2} \rfloor}^{n-1} d \cdot k + c \cdot n \\
&= \frac{2}{n} \left(\sum_{k=1}^{n-1} d \cdot k - \sum_{k=1}^{k=\lfloor \frac{n}{2} \rfloor - 1} d \cdot k \right) + c \cdot n \\
&<= \frac{2}{n} \left(\frac{dn(n-1)}{2} - \frac{dn(n-1)}{8} \right) + c \cdot n \\
&= dn - d - \frac{dn}{4} + \frac{d}{4} + c \cdot n <= d \cdot n (d > 4c)
\end{aligned}
\tag{3}
$$

Stable Order Statistics

Worst-case Order Statistics

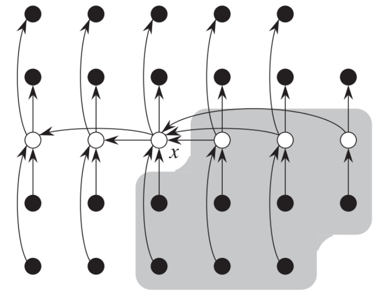
- The randomized algorithm described is excellent in practice (linear expected running time); however, the worst case running time ($\Theta(n^2)$) is slower than the trivial algorithm.
- Is it possible to obtain an algorithm whose worst-case running time is better than the $\Theta(n \log n)$ running time of merge sort?
- Answer is yes [Blum, Floyd, Pratt, Rivest, and Tarjan - 1973].
- The idea is to recursively generate a good pivot.

Stable Order Statistics

- The stable order statistics algorithm SELECT has the following steps:
  1. Partition the *n* items into groups each with 5 items. There are $\lceil \frac{n}{5} \rceil$ groups.
  2. Find the median for each group with any sorting algorithm. Note that since the number of items per group is fixed (five), the run time of finding the median of a group is constant.
  3. Recursively used SELECT to find the median *x* of the $\lceil \frac{n}{5} \rceil$ medians from Step 2.
  4. Partition the array with *x*. If *x* is the *i*'th item, then return *x*. Otherwise, use SELECT to recursively find the *i*'th item in the corresponding sub-array.

An Illustration of the SELECT Algorithm

- Illustration of SELECT:



Run Time of the SELECT Algorithm

- Let the run time be $T(n)$.
  - Step 1 takes $\Theta(n)$ time to partition the array.
  - Step 2 takes $\Theta(n)$ time, since there are $\Theta(\lceil \frac{n}{5} \rceil)$ groups, and sorting one 5-item group takes constant time.
  - Step 3 recursively calls SELECT with $\lceil \frac{n}{5} \rceil$ items. Therefore, step 3 takes $T(\lceil \frac{n}{5} \rceil)$ time.
  - Step 4 partitions the array with $\Theta(n)$ time.
  - Step 5 recursively calls SELECT on one of the sub-arrays. Let m be the size of the largest sub-array, the run time of Step 5 is then $T(m)$. To determine m,
    - About half of the $\lceil \frac{n}{5} \rceil$ groups have medians larger than x. These groups have at least 3 items larger that x.
    - For the group that contains x, there are at least two items larger than x.
    - Therefore, the number of elements larger than x is at least $3(\frac{1}{2}\lceil \frac{n}{5} \rceil) + 2 \geqslant \frac{3n}{10} + 2$

Run Time of the SELECT Algorithm cont.

- Let the run time be $T(n)$.
  - Step 5 recursively calls SELECT on one of the sub-arrays. Let m be the size of the largest sub-array, the run time of Step 5 is then $T(m)$. To determine m,
    - About half of the $\lceil \frac{n}{5} \rceil$ groups have medians larger than x. These groups have at least 3 items larger that x.
    - For the group that contains x, there are at least two items larger than x.
    - Therefore, the number of items larger than x is at least $3(\frac{1}{2}\lceil \frac{n}{5} \rceil) + 2 = \frac{3n}{10} + 2$
    - The number of items less than x is at most $n - \frac{3n}{10} + 2 = \frac{7n}{10} - 2$. That is, m is at most $\frac{7n}{10} - 2$.
  - Summing the time of all steps, we have $T(n) = T(\lceil \frac{n}{5} \rceil) + T(\frac{7n}{10} - 2) + \Theta(n)$.
  - It is fairly easy to prove that $T(n) = O(n)$ with induction.

The Intuition Behind SELECT and Randomized_Select

- Clearly, finding the *i*'th item does not require sorting the whole array. Therefore, the run time of order statistics algorithm should be much smaller than $O(n \lg n)$.
  - SELECT algorithm better demonstrates this fact by limiting the sorting within each five-item group.
- SELECT also achieves better partitioning by cleverly using the median of medians.
  - Note that, when partitioning the array with *x*, many items are guaranteed to be smaller than *x*, and do not need to be compared with *x* again.