Selection Problem

The *selection* problem is formally defined as follows: given an array $A[1 \cdots n]$ and an integer k, $1 \leq k \leq n$, to find the k-th smallest number in A. Here we assume that all numbers in A are distinct. The following algorithm we design can be easily extended to allow duplicated numbers in A.

A straightforward algorithm is sorting A; then the k-th element of the sorted array is exactly the k-th smallest number of A. This algorithm runs in $\Theta(n \log n)$ time. Can we do better? Notice that, when $k = 1$, this problem is to seek the smallest number in A; when $k = n$, this problem is to seek the largest number in A. In either case, we know that it can be done in linear time. In fact, the general case can also be solved in linear time, using a divide-and-conquer approach, which is the focus of this lecture. Note also that, when $k = n/2$ the selection problem seeks the *medain* of A.

Pivot-based Divide-and-Conquer Algorithm

We will design a *pivot-based* divide-and-conquer algorithm. The idea is to first choose a number in A, called pivot, denoted as x. We then partition A, using x, into 3 parts, (A_1, x, A_2), where A_1 (resp. A_2) stores the numbers in A that are smaller (resp. larger) than x. Specifically, we initialize two empty lists A_1 and A_2, we then examine all numbers: for each $1 \leq i \leq n$, we put $A[i]$ to A_1 if $A[i] < x$, and we put $A[i]$ to A_2 if $A[i] > x$.

Now, with A_1 and A_2 in hand, we can locate which part contains the k-th smallest number of A (and therefore discard the other two parts). Specifically, if $k \leq |A_1|$, then we know that the k-th smallest number of A must be in A_1, and it is exactly the k-th smallest number of A_1; if $k = |A_1| + 1$, then we know that the k-th smallest number of A must be x; if $k > |A_1| + 1$, then we know that the k-th smallest number of A must be in A_2, and it is exactly the $(k - |A_1| - 1)$-th smallest number of A_2.

Example. Let $A = [15, 5, 2, 20, 1, 9, 4, 13, 8]$ and $k = 7$. Suppose that we are given pivot $x = 8$. We partion A into A_1, x, A_2, where $A_1 = [5, 2, 1, 4]$ and $A_2 = [15, 20, 9, 13]$. As $k = 7 > |A_1| + 1 = 5$, the 7th smallest number of A must be the 2nd (i.e, $7 - 5$) smallest number of A_2.

The above analyis is illustrated with the following pseudo-code.

```
Algorithm selection (A, k)
    x = find-pivot (A);
    for i = 1 to |A|
        if A[i] < x: add A[i] to A1
        if A[i] > x: add A[i] to A2
    end
    if k ≤ |A1|: return selection (A1, k);
    else if k = |A1| + 1: return x;
    else: return selection (A2, k − |A1| − 1);
end algorithm;
```

We don't know how to find a (good) pivot yet. But no matter how we do it, the algorithm is always correct, i.e., it always find the k-th smallest number of A. The choice of x only affects the running time of this algorithm, as it affects $|A_1|$ and $|A_2|$.

Let's first have a look at the running time of above algorithm to find a clue what pivot we need to target. Let $T(n)$ be the running time of selection (A, k) when $|A| = n$. We note that this definition is independent of k; in other words, it is the running time for the *worst* choice of k. We use $P(n)$ to denote the running time of find-pivot (A,k) with $|A| = n$. Partitioning A into A_1, x, A_2 clearly takes $\Theta(n)$ time. For the remaining if-else-if-else part, again we assume the *worst-case* scenario, i.e., the larger array among A_1 and A_2 will always be chosen. Combined, we have this recurrence: $T(n) = P(n) + \Theta(n) + T(\max\{|A_1|, |A_2|\})$.

Finding a Good Pivot

A good choice of pivot should result in A_1 and A_2 as balanced as possible, as in this case $\max\{|A_1|, |A_2|\}$ will be minimized. The best case, of course, is that x is the median of A which gives $|A_1| = |A_2|$. However, calculating the median of A is kind of as hard as solving the selection problem. Instead, we can try to pick a pivot x that is close to the median of A, using the idea called "median of medians".

Here is the procedure. We partition A into $n/5$ subarrays, each of which has a size of 5. We then calculate the median (i.e., the 3rd smallest number) of each subarray. We collect these medians with an array M. Clearly, $|M| = n/5$. We then calculate the *median* of M, which will be the pivot we will use.

$M = (m_1,\ m_2,\ m_3,\ m_4)$

Figure 1: Finding pivot x using median of medians. The median of each subarray (of size 5) is marked with m_i. We then collect these medians and denote it as M. The median of M will be the pivot x.

There are two questions here. First, how to calculate the median of each subarray? As each subarray is of size 5, we can use any algorithm. For example, we can sort it, using time of $5 \cdot \log 5$, to get its median. Note, as there are $n/5$ subarrays, the total running time will be $n/5 \cdot 5 \cdot \log 5 = \log 5 \cdot n = \Theta(n)$. Second, how to calcuate the median of M? The answer is a recursive call: selection $(M,\ |M|/2)$. We will see later on that, the resulting algorithm still runs in linear time.

The pseudo-code for find-pivot function is given below.

```
function find-pivot (A)
    if |A| < 5: find (e.g., by sorting A) and return the median of A;
    partition A into n/5 subarrays of size 5;
    calculate the median of each subarray;
    let M be the array that includes all medians;
    return selection (M, |M|/2);
end
```

Running Time of the Selection Algorithm

By definition, the find-pivot functions takes time $\Theta(n) + T(|M|) = \Theta(n) + T(n/5)$, as $|M| = n/5$. Therefore, the total running time, in the form of a recurrence, is $T(n) = \Theta(n) + T(n/5) + \max\{T(|A_1|), T(|A_2|)\}$.

We now bound $\max\{|A_1|, |A_2|\}$. Think: how many numbers in A are *guaranteed* smaller than x (this number then gives a lower bound of $|A_1|$)? First, in these $n/5$ medians, half of them, i.e., $n/10$ numbers, are smaller

than x, as x is the median of these medians. Second, consider these $n/10$ subarrays whose median is smaller than x: clearly in each of these subarrays, at least two numbers are smaller than x. This is because the median of this subarray (with 5 numbers) is smaller than x and there are two numbers in this subarray that are smaller than its median. Combined, we have a lower bound of $|A_1|$: $|A_1| \geq n/10 + 2 \cdot n/10 = 3n/10$. This gives an upper bound of $|A_2|$: $|A_2| = n - |A_1| \leq 7n/10$.

Symmetrically, we have that $|A_2| \geq 3n/10$ and hence $|A_1| = n - |A_2| \leq 7n/10$. This is because, in these $n/5$ medians, $n/10$ of them are larger than x, and in these corresponding $n/10$ subarrays whose median is larger than x, there are in total $2 \cdot n/10$ numbers larger than x.

Combined, we have $\max\{|A_1|, |A_2|\} \leq 7n/10$. The above recurrence becomes $T(n) \leq \Theta(n) + T(n/5) + T(7n/10)$. Note that the recurrence is $T(n) \leq \cdots$ instead of $T(n) = \cdots$, as we only have an upper bound for $\max\{|A_1|, |A_2|\}$. (In other words, we don't have that $\max\{|A_1|, |A_2|\} = 7n/10$.) How to solve this recurrence? First, we can use a conclusion stated and proved in the next subsection. Suppose the recurrence is "=", i.e., $T'(n) \leq \Theta(n) + T'(n/5) + T'(7n/10)$. Applying the conclusion given below, we have $c_1 + c_2 = 1/5 + 7/10 < 1$. Hence its running time $T'(n) = \Theta(n)$. Now come back to $T(n)$. Since $T(n) \leq T'(n)$, we know that $T(n) = O(n)$. To show that $T(n) = \Omega(n)$, we need the fact that the algorithm requires the partition step which takes $\Theta(n)$ time. Hence, $T(n) = \Omega(n)$. Combined, we have $T(n) = \Theta(n)$.

Solving the Recurrence

Consider recurrence relation $T(n) = \Theta(n) + T(a \cdot n) + T(b \cdot n)$, $T(1) = 1$, $0 < a < 1$, $0 < b < 1$. We now prove the following:

1. $T(n) = \Theta(n)$, if $a + b < 1$.
2. $T(n) = \Theta(n \cdot \log n)$, if $a + b = 1$.

Assume that the term $\Theta(n)$ in the recursion admits a lower bound of $d_1 n$ and upper bound of $d_2 n$, for large enough n. That is $T(n) \geq d_1 n + T(an) + T(bn)$ and $T(n) \leq d_2 n + T(an) + T(bn)$.

1. We first prove that $T(n) = O(n)$, by induction. Assume that $T(n) \leq c_2 n$ for large enough n, where $c_2 = d_2/(1-a-b)$. Now we prove that $T(n+1) \leq c_2(n+1)$. We can write

$$T(n+1) \leq d_2(n+1) + T(a(n+1)) + T(b(n+1)).$$

   We have $a(n+1) \leq n$ and $b(n+1) \leq n$ for large enough n as $a < 1$ and $b < 1$. By the inductive assumption we have

$$\begin{aligned} T(n+1) &\leq d_2(n+1) + c_2 a(n+1) + c_2 b(n+1) \\ &\leq d_2(n+1) + c_2(a+b)(n+1) \\ &= (d_2 + c_2(a+b))(n+1) \\ &= c_2(n+1). \end{aligned}$$

   It's easy to verify the last equation, i.e., $d_2 + c_2(a+b) = c_2$ with $c_2 = d_2/(1-a-b)$. In fact this is where we determine the value of c_2. You can also see why $a + b < 1$ is required here. We then

prove that $T(n) = \Omega(n)$, by induction. Assume that $T(n) \geq c_1 n$ for large enough n, where $c_1 = d_1/(1-a-b)$. Now we prove that $T(n+1) \geq c_1(n+1)$. We can write

$$\begin{aligned} T(n+1) &\geq d_1(n+1) + T(a(n+1)) + T(b(n+1)) \\ &\geq d_1(n+1) + c_1 a(n+1) + c_1 b(n+1) \\ &\geq d_1(n+1) + c_1(a+b)(n+1) \\ &= (d_1 + c_1(a+b))(n+1) \\ &= c_1(n+1). \end{aligned}$$

The last equation holds as $d_1 + c_1(a+b) = c_1$ with $c_1 = d_1/(1-a-b)$.

2. We first prove that $T(n) = O(n\log n)$, by induction. Assume that $T(n) \leq c_2 n\log n$ for large enough n, where $c_2 = -d_2/(a\log a + b\log b)$. Now we prove that $T(n+1) \leq c_2(n+1)\log(n+1)$. We can write

$$\begin{aligned} T(n+1) &\leq d_2(n+1) + T(a(n+1)) + T(b(n+1)) \\ &\leq d_2(n+1) + c_2 a(n+1)\log(a(n+1)) + c_2 b(n+1)\log(b(n+1)) \\ &\leq d_2(n+1) + c_2(a\log a + b\log b)(n+1) + c_2(a+b)(n+1)\log(n+1) \\ &= (d_2 + c_2(a\log a + b\log b))(n+1) + c_2(n+1)\log(n+1) \\ &= c_2(n+1)\log(n+1). \end{aligned}$$

The last equation holds as $d_2 + c_2(a\log a + b\log b) = 0$ with $c_2 = -d_2/(a\log a + b\log b)$. We then prove that $T(n) = \Omega(n\log n)$, by induction. Assume that $T(n) \geq c_1 n\log n$ for large enough n, where $c_1 = -d_1/(a\log a + b\log b)$. Now we prove that $T(n+1) \geq c_1(n+1)\log(n+1)$. We can write

$$\begin{aligned} T(n+1) &\geq d_1(n+1) + T(a(n+1)) + T(b(n+1)) \\ &\geq d_1(n+1) + c_1 a(n+1)\log(a(n+1)) + c_1 b(n+1)\log(b(n+1)) \\ &\geq d_1(n+1) + c_1(a\log a + b\log b)(n+1) + c_1(a+b)(n+1)\log(n+1) \\ &= (d_1 + c_1(a\log a + b\log b))(n+1) + c_1(n+1)\log(n+1) \\ &= c_1(n+1)\log(n+1). \end{aligned}$$

The last equation holds as $d_1 + c_1(a\log a + b\log b) = 0$ with $c_1 = -d_1/(a\log a + b\log b)$.

Choices of the Size of Subarrays

Why partitioning A into subarrays of size 5? Does other size work (i.e., leading to a $\Theta(n)$ algorithm)? You will get yourself an answer by working on the assignment problems.

A Randomized Algorithm for Selection Problem

We now design a *randomized algorithm* for selection problem. The idea is simply pick the pivot uniformly at random from A. The pseudo-code is given below.

```
function find-pivot-random (A)
    pick pivot x uniformly at random from A;
end function ;
```

First, note that the selection algorithm combined with above random function to pick pivot is correct, i.e., it will still find the k-th smallest number of A. We now analyze its running time. Again, let $T(n)$ be the running time of selection (A, k), with above random function to select pivot, when $|A| = n$. Define random variable $Z := \max\{|A_1|, |A_2|\}$. Hence we can write $T(n) = \Theta(n) + T(Z)$. Again, here Z is a random variable, $T(Z)$ is a random variable, and therefore $T(n)$ is also a random variable.

We aim to calculate the expected running time, a common practice in analyzing randomized algorithms. We first estimate the distribution of Z. Think: what's the probability for event $Z \leq 3n/4$? Answer: at least $1/2$. Why? This is because we pick x uniformly at random from x. Therefore, the probability of event of $\{x$ is between 25-percentile and 75-percentile of $A\}$ is $1/2$. And this event is equivalent to the event that $Z \leq 3n/4$, according to the definition of Z. Hence, $\Pr(Z \leq 3n/4) = 1/2$.

We now calculate its expected running time. We start with recursion $T(n) = \Theta(n) + T(Z)$. We first take expectation over Z on both sides: $\mathcal{E}_Z[T(n)] = \Theta(n) + \mathcal{E}_Z[T(Z)]$. Note that $T(n)$ does not contain Z (although $T(n)$ is a random variable), we have $\mathcal{E}_Z[T(n)] = T(n)$. That is $T(n) = \Theta(n) + \mathcal{E}_Z[T(Z)]$.

We now estimate $\mathcal{E}_Z[T(Z)]$.

$$\begin{aligned}
\mathcal{E}_Z[T(Z)] &= \sum_{k=n/2}^{n} \Pr(Z=k) \cdot T(k) \\
&= \sum_{k=n/2}^{3n/4} \Pr(Z=k) \cdot T(k) + \sum_{k=3n/4}^{n} \Pr(Z=k) \cdot T(k) \\
&\leq T(3n/4) \cdot \sum_{k=n/2}^{3n/4} \Pr(Z=k) + T(n) \cdot \sum_{k=3n/4}^{n} \Pr(Z=k) \\
&= T(3n/4) \cdot \Pr(Z \leq 3n/4) + T(n) \cdot \Pr(Z \geq 3n/4) \\
&\leq T(3n/4) \cdot 1/2 + T(n) \cdot 1/2.
\end{aligned}$$

Hence, now we have $T(n) \leq \Theta(n) + T(3n/4)/2 + T(n)/2$, which gives $T(n) \leq \Theta(n) + T(3n/4)$. We now take expectation, over $T(n)$, on both sides: $\mathcal{E}_T[T(n)] = \Theta(n) + \mathcal{E}_T[T(3n/4)]$. By using master's theorem, we have that $\mathcal{E}_T[T(n)] = \Theta(n)$.