Week 03: Selection

FanFly

March 22, 2020

Order Statistics

First, let us introduce the selction problem.

Order Statistics

First, let us introduce the selction problem.

The Selection Problem

- Input: An array A of n numbers and an index k with $0 \leq k < n$.
- Output: The element at index k in the sorted array reordered from A.

Order Statistics

First, let us introduce the selction problem.

The Selection Problem

- Input: An array A of n numbers and an index k with $0 \leq k < n$.
- Output: The element at index k in the sorted array reordered from A.

Note that the output is the $(k + 1)$-th smallest number of A, which is the $(k + 1)$-th **order statistic** of A.

Order Statistics

First, let us introduce the selction problem.

The Selection Problem

- Input: An array A of n numbers and an index k with $0 \leq k < n$.
- Output: The element at index k in the sorted array reordered from A.

Note that the output is the $(k+1)$-th smallest number of A, which is the $(k+1)$-th **order statistic** of A.

- It is useful in many cases, e.g., the $\lfloor (n-1)/2 \rfloor$-th order statistic is the **lower median**.

The First Attempt

One can sort the array and then output the element at index k.

```
def sort_and_select(A, k):
    B = sorted(A)
    return B[k]
```

The First Attempt

One can sort the array and then output the element at index k.

```
def sort_and_select(A, k):
    B = sorted(A)
    return B[k]
```

What is its time complexity?

The First Attempt

One can sort the array and then output the element at index k.

```
def sort_and_select(A, k):
    B = sorted(A)
    return B[k]
```

What is its time complexity?

- One needs $\Theta(n \log n)$ time to sort an array with length n.

The First Attempt

One can sort the array and then output the element at index k.

```
def sort_and_select(A, k):
    B = sorted(A)
    return B[k]
```

What is its time complexity?

- One needs $\Theta(n \log n)$ time to sort an array with length n.
- Indexing takes time $\Theta(1)$.

The First Attempt

One can sort the array and then output the element at index k.

```
def sort_and_select(A, k):
    B = sorted(A)
    return B[k]
```

What is its time complexity?

- One needs $\Theta(n \log n)$ time to sort an array with length n.
- Indexing takes time $\Theta(1)$.

Thus, this algorithm runs in $\Theta(n \log n)$ time.

The Second Attempt

If k is small, we can find only the $(k+1)$ smallest numbers.

```python
def partial_sort_and_select(A, k):
    B = list(A)
    for i in range(k + 1):
        for j in range(i + 1, len(A)):
            if B[j] < B[i]:
                B[i], B[j] = B[j], B[i]
    return B[k]
```

The Second Attempt

If k is small, we can find only the $(k+1)$ smallest numbers.

```
def partial_sort_and_select(A, k):
    B = list(A)
    for i in range(k + 1):
        for j in range(i + 1, len(A)):
            if B[j] < B[i]:
                B[i], B[j] = B[j], B[i]
    return B[k]
```

What is its time complexity?

The Second Attempt

If k is small, we can find only the $(k+1)$ smallest numbers.

```
def partial_sort_and_select(A, k):
    B = list(A)
    for i in range(k + 1):
        for j in range(i + 1, len(A)):
            if B[j] < B[i]:
                B[i], B[j] = B[j], B[i]
    return B[k]
```

What is its time complexity?

- Swaping two elements takes $\Theta(1)$ time.

The Second Attempt

If k is small, we can find only the $(k+1)$ smallest numbers.

```
def partial_sort_and_select(A, k):
    B = list(A)
    for i in range(k + 1):
        for j in range(i + 1, len(A)):
            if B[j] < B[i]:
                B[i], B[j] = B[j], B[i]
    return B[k]
```

What is its time complexity?

- Swaping two elements takes $\Theta(1)$ time.
- The number of the inner iterations is

$$(n-1)+(n-2)+\cdots+(n-k-1)=\left(n-\frac{k+2}{2}\right)(k+1).$$

The Second Attempt

If k is small, we can find only the $(k+1)$ smallest numbers.

```python
def partial_sort_and_select(A, k):
    B = list(A)
    for i in range(k + 1):
        for j in range(i + 1, len(A)):
            if B[j] < B[i]:
                B[i], B[j] = B[j], B[i]
    return B[k]
```

What is its time complexity?

- Swaping two elements takes $\Theta(1)$ time.
- The number of the inner iterations is

$$(n-1)+(n-2)+\cdots+(n-k-1)=\left(n-\frac{k+2}{2}\right)(k+1).$$

Thus, this algorithm runs in $\Theta(nk)$ time.

Comparison of Two Attempts

Let us compare the two algorithms.

Comparison of Two Attempts

Let us compare the two algorithms.

- The first attempt runs in $\Theta(n \log n)$ time.

Comparison of Two Attempts

Let us compare the two algorithms.

- The first attempt runs in $\Theta(n \log n)$ time.
- The second attempt runs in $\Theta(nk)$ time.

Comparison of Two Attempts

Let us compare the two algorithms.

- The first attempt runs in $\Theta(n \log n)$ time.
- The second attempt runs in $\Theta(nk)$ time.

Which one is faster?

Comparison of Two Attempts

Let us compare the two algorithms.

- The first attempt runs in $\Theta(n \log n)$ time.
- The second attempt runs in $\Theta(nk)$ time.

Which one is faster? It depends on the size of k.

Comparison of Two Attempts

Let us compare the two algorithms.

- The first attempt runs in $\Theta(n \log n)$ time.
- The second attempt runs in $\Theta(nk)$ time.

Which one is faster? It depends on the size of k.

- If k is large (i.e., $k = \Omega(\log n)$), we can choose the first method.

Comparison of Two Attempts

Let us compare the two algorithms.

- The first attempt runs in $\Theta(n \log n)$ time.
- The second attempt runs in $\Theta(nk)$ time.

Which one is faster? It depends on the size of k.

- If k is large (i.e., $k = \Omega(\log n)$), we can choose the first method.
- If k is small (i.e., $k = O(\log n)$), we can choose the second method.

Comparison of Two Attempts

Let us compare the two algorithms.

- The first attempt runs in $\Theta(n \log n)$ time.
- The second attempt runs in $\Theta(nk)$ time.

Which one is faster? It depends on the size of k.

- If k is large (i.e., $k = \Omega(\log n)$), we can choose the first method.
- If k is small (i.e., $k = O(\log n)$), we can choose the second method.

Thus, we have an algorithm that runs in $\Theta(\min\{n \log n, nk\})$ time.

Quicker, Quicker!

However, there exists faster algorithms that solve the selction problem.

Quicker, Quicker!

However, there exists faster algorithms that solve the selction problem.

- A deterministic algorithm, called the **median of medians** algorithm, solves the problem in $O(n)$ time.

Quicker, Quicker!

However, there exists faster algorithms that solve the selction problem.

- A deterministic algorithm, called the **median of medians** algorithm, solves the problem in $O(n)$ time.
- A randomized algorithm, called the **quick select** algorithm, solves the problem in expected $O(n)$ time.

Quicker, Quicker!

However, there exists faster algorithms that solve the selction problem.

- A deterministic algorithm, called the **median of medians** algorithm, solves the problem in $O(n)$ time.
- A randomized algorithm, called the **quick select** algorithm, solves the problem in expected $O(n)$ time.

We are going to introduce the latter one, and thus we need to deal with **randomization** first.

Randomization

We use the package `random` to generate pseudo-random numbers.

Randomization

We use the package `random` to generate pseudo-random numbers.

```
>>> import random
```

Randomization

We use the package `random` to generate pseudo-random numbers.

```
>>> import random
```

```
>>> random.random()     # generate number in [0.0, 1.0)
0.5476045155149599
```

Randomization

We use the package `random` to generate pseudo-random numbers.

```
>>> import random
```

```
>>> random.random()    # generate number in [0.0, 1.0)
0.5476045155149599
>>> random.random()
0.630346861646684
```

Randomization

We use the package `random` to generate pseudo-random numbers.

```
>>> import random
```

```
>>> random.random()    # generate number in [0.0, 1.0)
0.5476045155149599
>>> random.random()
0.630346861646684
```

```
>>> random.randrange(5)    # generate integer in [0, 5)
2
```

Randomization

We use the package `random` to generate pseudo-random numbers.

```
>>> import random
```

```
>>> random.random()    # generate number in [0.0, 1.0)
0.5476045155149599
>>> random.random()
0.630346861646684
```

```
>>> random.randrange(5)    # generate integer in [0, 5)
2
>>> random.randrange(5)
4
```

Randomization

We use the package `random` to generate pseudo-random numbers.

```
>>> import random
```

```
>>> random.random()    # generate number in [0.0, 1.0)
0.5476045155149599
>>> random.random()
0.630346861646684
```

```
>>> random.randrange(5)    # generate integer in [0, 5)
2
>>> random.randrange(5)
4
>>> random.randrange(5)
3
```

Randomization

We use the package `random` to generate pseudo-random numbers.

```
>>> import random
```

```
>>> random.random()    # generate number in [0.0, 1.0)
0.5476045155149599
>>> random.random()
0.630346861646684
```

```
>>> random.randrange(5)    # generate integer in [0, 5)
2
>>> random.randrange(5)
4
>>> random.randrange(5)
3
```

See the offcial documentation page for more examples.

Partition

Now we introduce the partition algorithm, which is really useful.

Partition

Now we introduce the partition algorithm, which is really useful.

- First, we choose the last element as **pivot**.

Partition

Now we introduce the partition algorithm, which is really useful.

- First, we choose the last element as **pivot**.
- Then we reorder the array such that elements less than pivot come before the pivot, while elements greater than pivot come after the pivot.

Partition

Now we introduce the partition algorithm, which is really useful.

- First, we choose the last element as **pivot**.
- Then we reorder the array such that elements less than pivot come before the pivot, while elements greater than pivot come after the pivot.

```
def partition(A, l, r):
    i = l
    for j in range(l, r - 1):
        if A[j] < A[r - 1]:
            A[i], A[j] = A[j], A[i]
            i += 1
    A[i], A[r - 1] = A[r - 1], A[i]
    return i
```

Partition (cont.)

Step 1	3	1	4	1	5	9	2

Partition (cont.)

Step 1	3	1	4	1	5	9	2
Step 2	3	1	4	1	5	9	2

Partition (cont.)

Step 1	3	1	4	1	5	9	2
Step 2	3	1	4	1	5	9	2
Step 3	1	3	4	1	5	9	2

Partition (cont.)

Step 1	3	1	4	1	5	9	2
Step 2	3	1	4	1	5	9	2
Step 3	1	3	4	1	5	9	2
Step 4	1	3	4	1	5	9	2

Partition (cont.)

Step 1	3	1	4	1	5	9	2
Step 2	3	1	4	1	5	9	2
Step 3	1	3	4	1	5	9	2
Step 4	1	3	4	1	5	9	2
Step 5	1	1	4	3	5	9	2

Partition (cont.)

Step 1	3	1	4	1	5	9	2
Step 2	3	1	4	1	5	9	2
Step 3	1	3	4	1	5	9	2
Step 4	1	3	4	1	5	9	2
Step 5	1	1	4	3	5	9	2
Step 6	1	1	4	3	5	9	2

Partition (cont.)

Step 1	3	1	4	1	5	9	2
Step 2	3	1	4	1	5	9	2
Step 3	1	3	4	1	5	9	2
Step 4	1	3	4	1	5	9	2
Step 5	1	1	4	3	5	9	2
Step 6	1	1	4	3	5	9	2
Step 7	1	1	4	3	5	9	2

Partition (cont.)

Step 1	3	1	4	1	5	9	2
Step 2	3	1	4	1	5	9	2
Step 3	1	3	4	1	5	9	2
Step 4	1	3	4	1	5	9	2
Step 5	1	1	4	3	5	9	2
Step 6	1	1	4	3	5	9	2
Step 7	1	1	4	3	5	9	2
Step 8	1	1	2	3	5	9	4

Quick Select

With the partition procedure, one can select the any order statistic really quickly!

```
def quick_select(A, l, r, k):
    i = partition(A, l, r)
    if k == i:
        return A[k]
    elif k > i:
        return quick_select(A, i + 1, r, k)
    else:
        return quick_select(A, l, i, k)
```

Analysis of Quick Select

Let us analyze the time complexity of quick select.

Analysis of Quick Select

Let us analyze the time complexity of quick select.

- The partition procedure runs in $\Theta(n)$ time when the array has n elements.

Analysis of Quick Select

Let us analyze the time complexity of quick select.

- The partition procedure runs in $\Theta(n)$ time when the array has n elements.
- If the partition is balanced, then we have

$$T(n) = T(n/2) + \Theta(n)$$

for $n \geq 2$, implying $T(n) = \Theta(n)$.

Analysis of Quick Select

Let us analyze the time complexity of quick select.

- The partition procedure runs in $\Theta(n)$ time when the array has n elements.
- If the partition is balanced, then we have

$$T(n) = T(n/2) + \Theta(n)$$

  for $n \geq 2$, implying $T(n) = \Theta(n)$.
- However, if the partition is unbalanced, then we may have

$$T(n) = T(n-1) + \Theta(n)$$

  for $n \geq 2$, implying $T(n) = \Theta(n^2)$.

Analysis of Quick Select (cont.)

If we'd like to make the partition balanced, we can use some tricks.

Analysis of Quick Select (cont.)

If we'd like to make the partition balanced, we can use some tricks.

- Instead of choosing the last element as pivot, we can choose a random element as pivot.

Analysis of Quick Select (cont.)

If we'd like to make the partition balanced, we can use some tricks.

- Instead of choosing the last element as pivot, we can choose a random element as pivot.
- With randomization, the partition is somehow balanced in most cases, and it leads to a expected linear time complexity.

Conclusion

The time complexity of the selction problem is $\Theta(n)$.

The Selection Problem

- Input: An array A of n numbers and an index k with $0 \leq k < n$.
- Output: The element at index k in the sorted array reordered from A.

Conclusion

The time complexity of the selction problem is $\Theta(n)$.

The Selection Problem

- Input: An array A of n numbers and an index k with $0 \leq k < n$.
- Output: The element at index k in the sorted array reordered from A.

The comparison among the algorithms that solves the selction problem is as follows.

Algorithm	Worst-case Time Complexity
Sorting (Merge Sort)	$\Theta(n \log n)$
Partial Selection Sort	$\Theta(nk)$
Quick Select	$\Theta(n)$ (expected)
Median of Medians	$\Theta(n)$

Exercise #1

If $T(n)$ satisfies

$$T(n) = \begin{cases} O(1), & \text{if } n \leq 1 \\ T(n-2) + \Theta(n), & \text{otherwise,} \end{cases}$$

what is the exact complexity of $T(n)$?

Exercise #1

If $T(n)$ satisfies

$$T(n) = \begin{cases} O(1), & \text{if } n \leq 1 \\ T(n-2) + \Theta(n), & \text{otherwise,} \end{cases}$$

what is the exact complexity of $T(n)$?

Solution

We have $T(n) = \Theta(n^2)$.

Exercise #2

If $T(n)$ satisfies

$$T(n) = \begin{cases} O(1), & \text{if } n \leq 1 \\ T(9n/10) + \Theta(n), & \text{otherwise,} \end{cases}$$

what is the exact complexity of $T(n)$?

Exercise #2

If $T(n)$ satisfies

$$T(n) = \begin{cases} O(1), & \text{if } n \leq 1 \\ T(9n/10) + \Theta(n), & \text{otherwise,} \end{cases}$$

what is the exact complexity of $T(n)$?

Solution

Use the master theorem, and we have $T(n) = \Theta(n)$.

Exercise #3

In fact, there is an algorithm called quick sort, that also uses the partition procedure.

Exercise #3

In fact, there is an algorithm called quick sort, that also uses the partition procedure.

```
def quick_sort(A, l, r):
    if r - l <= 1:
        return
    i = partition(A, l, r)
    quick_sort(A, l, i)
    quick_sort(A, i + 1, r)
```

Exercise #3

In fact, there is an algorithm called quick sort, that also uses the partition procedure.

```
def quick_sort(A, l, r):
    if r - l <= 1:
        return
    i = partition(A, l, r)
    quick_sort(A, l, i)
    quick_sort(A, i + 1, r)
```

If you are really unlucky such that the partition is unbalanced in each iteration, how much time will it takes to run quick sort?

Exercise #3

In fact, there is an algorithm called quick sort, that also uses the partition procedure.

```
def quick_sort(A, l, r):
    if r - l <= 1:
        return
    i = partition(A, l, r)
    quick_sort(A, l, i)
    quick_sort(A, i + 1, r)
```

If you are really unlucky such that the partition is unbalanced in each iteration, how much time will it takes to run quick sort?

- You can assume that the pivot is always the largest element.

Exercise #3

In fact, there is an algorithm called quick sort, that also uses the partition procedure.

```
def quick_sort(A, l, r):
    if r - l <= 1:
        return
    i = partition(A, l, r)
    quick_sort(A, l, i)
    quick_sort(A, i + 1, r)
```

If you are really unlucky such that the partition is unbalanced in each iteration, how much time will it takes to run quick sort?

- You can assume that the pivot is always the largest element.

Solution

It takes $\Theta(n^2)$ time.
(However, quick sort can run in expected $\Theta(n \log n)$ time if randomization is used.)