Lecture 6 - Selection Problem

Fall 2025, Korea University

Instructor: Gabin An (gabin_an@korea.ac.kr)

Course Outline (Before Midterm)

- Part 1: Basics
  - ~~Divide and Conquer (w/ Integer Multiplication)~~ ✅
  - ~~Basic Sorting Algorithms (Insertion Sort & Merge Sort)~~ ✅
  - ~~Asymptotic Analysis (Big-O, Big-Theta, Big-Omega)~~ ✅
  - ~~Solving Recurrences Using Master Method~~ ✅
- Part 2: Advanced Selection and Sorting
  - **Median and Selection Algorithm** 👈
  - Solving Recurrences Using Substitution Method
  - Quick Sort, Counting Sort, Radix Sort
- Part 3: Data Structures
  - Heaps, Binary Search Trees, Balanced BSTs

Review

Do you remember what we covered in the last class? We looked at how to solve recurrence relations of the form:

$$T(n) = aT\left(\frac{n}{b}\right) + O(n^d)$$

- **Master Method** (last class)
  A powerful shortcut for solving divide-and-conquer recurrences - **but** it only works when all subproblems are of **equal size**.
- **Substitution Method** (next class)
  A more flexible, general-purpose technique. It can handle a wider variety of recurrence forms and is often used when the Master Method doesn't apply.

Let's Review Master Method 🧙

Today's Goals

- Define the **selection problem**
- Learn a clever algorithm for the selection problem
- Analyze the running time!

Selection Problem: Finding the k-th Smallest Element

> Input: An unsorted array of size n (without duplicates)
>
> Output: Return the k-th smallest element ($1 \le k \le n$)

Example

Array: `[7, 2, 1, 8, 6, 3, 5, 4]` ($n = 8$)

- *k = 1* → 1st smallest = `1`
- *k = 3* → 3rd smallest = `3`
- *k = 8* → 8th smallest = `8`

A Naive Approach & Lower Bound Insight

- A naive approach
  - Sort the array in ascending order and return the element at the k-th position
  - For example, with MergeSort, it's $O(n \log n)$!
  - It is corret but not optimal.
- Any correct algorithm must inspect every element at least once, i.e., $\Omega(n)$.
- But can we actually do it in $\Theta(n)$ time?

A Surprisingly Clever Linear-Time Algorithm

We'll see an algorithm that finds the *k*-th smallest element in **O(n)** time!

No need for:

- ❌ Sorting
- ❌ Heaps
- ✅ Just smart **divide-and-conquer** with **careful grouping and selection**

High Level Strategy

We'll try to do **Binary Search** over an unsorted array.

At each step:

- Partition the array into:
  - elements smaller than some pivot
  - elements larger than that pivot
- Decide which side contains the k-th smallest
- Recurse only on that part

🔍 Example: `Select([7, 2, 1, 8, 6, 3, 5, 4], k=2)`

- Pivot = `3`
- Partition:
  - 🔻 Smaller: `[2, 1]` (size = 2)
  - 📍 Pivot: `3` (rank = 3)
  - 🔺 Larger: `[7, 8, 6, 5, 4]` (size=5)
- Since we want the **2nd smallest**
  - It's in the **smaller** part
    - Recurse on `Select([2, 1], k=2)`
- If `k` was `6` ...
  - It's in the **larger** part.
    - Recurse on `Select([7, 8, 6, 5, 4], k=?)` (Guess the new `k` !)

Selection Algorithm

```python
def select(A, k):
    assert 1 <= k <= len(A)
    if len(A) == 1:
        return A[0]
    p = choose_pivot(A)
    A_less = [x for x in A if x < p]
    A_greater = [x for x in A if x > p]
    if len(A_less) == k - 1:
        return p
    elif len(A_less) > k - 1:
        return select(A_less, k)
    else:
        return select(A_greater, k - len(A_less) - 1)
```

Proving Correctness (Using Strong Induction!)

Claim: For every array A of n distinct elements and every $k \in [1, n]$, `select(A, k)` returns the k-th smallest element of A.

Base Case (n=1)

If $|A| = 1$ (hence $k = 1$), the algorithm returns $A[0]$, which is trivially the 1st smallest. ✅

Induction Hypothesis Assume the claim holds for all arrays of size $\leq m$ (for all valid k).

Inductive Step ($n = m + 1$) (Prove Case 3 by yourself 🙂)

- Case 1 ($|A_{less}| = k - 1$): Exactly $k - 1$ elements are smaller than p. Hence p is the k-th smallest, and `select` returns p. Correct.
- Case 2 ($|A_{less}| > k - 1$): In this case, the k-th smallest of A_{less} is the k-th smallest of A. Since $|A_{less}| \leq m$, the recursive call returns the k-th smallest of A_{less}. Correct.

Choosing Pivot

```
def select(A, k):
    assert 1 <= k <= len(A)
    if len(A) == 1:
        return A[0]
    p = choose_pivot(A) # ⭐⭐⭐⭐⭐⭐⭐
    A_less = [x for x in A if x < p]
    A_greater = [x for x in A if x > p]
    if len(A_less) == k - 1:
        return p
    elif len(A_less) > k - 1:
        return select(A_less, k)
    else:
        return select(A_greater, k - len(A_less) - 1)
```

- The pivot only affects **runtime**, not correctness.
  - To show this, suppose that `choose_pivot` runs in $\Theta(n)$.

👎 Bad Pivot: Worst-Case Runtime

Proposition

> If the pivot is the **minimum or maximum**, then `Select` runs in $\Theta(n^2)$.

Why?

- Each recursive call only removes **1 element**
- Running `ChoosePivot` and creating subarrays takes linear time.
- Recurrence: $T(n) = T(n-1) + \Theta(n)$
- $T(n) = \Theta(n^2)$ since

$$T(n) \leq c_1 n + c_1(n-1) + c_1(n-2) + \ldots + c_1 = c_1 n(n+1)/2$$
$$T(n) \geq c_2 n + c_2(n-1) + c_2(n-2) + \ldots + c_2 = c_2 n(n+1)/2$$

👍 Good Pivot: Best-Case Runtime

Proposition

> If the pivot is the **median**, then `Select` runs in $O(n)$.

Why?

- Each call removes **half the elements**
- Recurrence: $T(n) \leq T(\frac{n}{2}) + \Theta(n) \leq T(\frac{n}{2}) + cn$
- $T(n) = O(n)$ since

$$T(n) \leq cn\left(1 + \frac{1}{2} + \frac{1}{2^2} + \ldots + \frac{1}{n}\right) \leq cn\left(1 + \frac{1}{2} + \frac{1}{2^2} + \ldots\right) = cn \cdot \left(\frac{1}{1 - 1/2}\right) = 2cn$$

How Do We Find a Good Pivot?

We saw that choosing a **good pivot** is critical to achieving linear time!

- Idea #1: Choose a pivot that creates the most “balanced” split
  - No … 🙅 This is exactly selection problem we are tyring to solve, with $k = n/2$.
- Idea #2: **Find a pivot "close enough" to the median**
  - We don’t need the exact median.
  - We just need a pivot that ensures
    - Subarrays shrink quickly
    - Worst-case size of recursive calls drops.
  - 💡 Trick: Use the **Median of Medians**

Median of Medians: High-Level Idea

- We want a pivot that's *not too far* from the true median, to ensure good balance in the recursive split.
- In 1973, Blum, Floyd, Pratt, Rivest, and Tarjan came up with the Median of Medians algorithm.
  - This method guarantees that, in worst case, we eliminate **a fixed fraction of elements** each time.

Median of Medians: `choose_pivot` Algorithm

```python
def choose_pivot(A):
    # Base case: if small enough, just sort and return median
    if len(A) <= 5:
        return sorted(A)[len(A) // 2]

    # Step 1: Divide A into groups of 5
    groups = [A[i:i+5] for i in range(0, len(A), 5)]

    # Step 2: Sort each group and collect their medians
    medians = [sorted(group)[len(group) // 2] for group in groups]

    # Step 3: Recursively find the median of the medians
    return select(medians, k=(len(medians)+1)//2)
```

```python
A = [13, 5, 2, 8, 9, 4, 7, 1, 6, 3, 10, 12, 11, 15, 14]
```

Example

```
A = [13, 5, 2, 8, 9, 4, 7, 1, 6, 3, 10, 12, 11, 15, 14]
```

1. Group into 5s

   ```
   [13, 5, 2, 8, 9] # median = 8
   [4, 7, 1, 6, 3] # median = 4
   [10, 12, 11, 15, 14] # median = 12
   ```

2. Medians list

   ```
   [8, 4, 12]
   ```

3. Pivot (median of medians)

   ```
   select([8, 4, 12], k=2) # pivot = 8
   ```

Why Median of Medians Ensures Progress (The 30-70 Lemma)

The median-of-medians pivot guarantees a split of 30%-70% or better of the input array.

- At least 30% of elements are less than or equal to the pivot.
- At least 30% of elements are greater than or equal to the pivot.

30-70 Lemma

> For every input array of length $n \geq 2$, the subarray passed to the selection recursive call has length at most $\frac{7}{10}n$.

Proof of the 30-70 Lemma

Suppose that n is a multiple of 5, and $g = n/5$ (i.e., # groups).
Since the p is the **median** of g medians, at least $\lceil \frac{g}{2} \rceil - 1$ medians are **less than** p, which means at least three elements in those $\lceil \frac{g}{2} \rceil - 1$ groups are less than p.

Example: $n = 15, g = 3$. Here, 8 is the median of medians.
At least $\lceil \frac{3}{2} \rceil - 1 = 1$ median (here, 4) is less than 8.
Therefore, at least 3 elements (here, 1, 3, 4) in 4's group are less than 8.

```
[  1 <  3 <  4 <  6 <  3 ] # median = 4
             v
[  2 <  5 <  8 <  9 < 10 ] # median = 8
             v
[ 10 < 11 < 12 < 14 < 15 ] # median = 12
```

> Since the p is the **median** of g medians, at least $\lceil \frac{g}{2} \rceil - 1$ medians are **less than** p, which means at least three elements in those $\lceil \frac{g}{2} \rceil - 1$ groups are less than p.

Therefore, in A, **at least** $3(\lceil \frac{g}{2} \rceil - 1) + 2$ elements are less than p.

$$\begin{aligned} |A_{greater}| &= \#\text{ of elements greater than } p \\ &\leq \#\text{ total elements except pivot} - \#\text{ of elements less than } p \\ &= (n-1) - 3 \cdot \left(\left\lceil \frac{g}{2} \right\rceil - 1 \right) - 2 \\ &= n - 3 \left\lceil \frac{n}{10} \right\rceil \\ &\leq n - 3 \cdot \frac{n}{10} \;\; = \frac{7}{10} n \end{aligned}$$

By symmetry, $|A_{less}| \leq \frac{7}{10} n$ as well. ■

Running Time Analysis of `select`

We want to analyze the runtime of `select(A, k)` when `choose_pivot` uses Median of Medians. Let $T(n)$ be the time to run `select` on an input of size n.

We split the analysis into three parts:

1. **Divide into groups of 5 and find their medians**: $O(n)$ (Sort $\frac{n}{5}$ subarrays with size 5)
2. **Recursively find median of medians**: $T(\frac{n}{5})$
3. **Partition around pivot**: Worst case, the pivot splits into:
    - $\frac{7}{10}n$ on the larger side
    - So next recursive call is at most $T(\frac{7n}{10})$

✍️ **Final Recurrence**: $T(n) \le T(\frac{n}{5}) + T(\frac{7n}{10}) + cn$

Running time of `select`

$$T(n) \leq T(\frac{n}{5}) + T(\frac{7n}{10}) + cn$$

We can't apply the Master Method because subproblems are of different sizes.

In the next class, we're going to learn **Substitution Method**, which is a more flexible, general-purpose technique.

Credits & Resources

Lecture materials adapted from:

- Stanford CS161 slides and lecture notes
  - https://stanford-cs161.github.io/winter2025/
- *Algorithms Illuminated* by Tim Roughgarden
  - https://algorithmsilluminated.com/