CS 2500: Algorithms

Lecture 18: Divide-and-Conquer: QuickSelect and Strassen's Matrix Multiplication

Shubham Chatterjee

Missouri University of Science and Technology, Department of Computer Science

October 17, 2024

Selecting the k-th Smallest Element

- The goal of the QuickSelect algorithm is to find the **k-th smallest element** in an array.
- It uses the `Partition` function, similar to Quick sort:
  - Partitions the array into elements smaller than or equal to the pivot and elements greater than the pivot.
  - Depending on the pivot's position, the algorithm decides whether to search the left or right part.

Selecting the k-th Smallest Element

Algorithm QuickSelect(a, n, k)

$low \leftarrow 1$, $up \leftarrow n+1$

$a[n+1] \leftarrow \infty$ ▷ Set the sentinel value

repeat

  $j \leftarrow \text{Partition}(a, low, up)$

  if $k = j$ **then**

    return ▷ k-th smallest element is found

  else if $k < j$ **then**

    $up \leftarrow j$ ▷ Search in the left part

  else

    $low \leftarrow j+1$ ▷ Search in the right part

  end if

until false

Example: QuickSelect

- **Array:** $a = [65, 70, 75, 80, 85, 60, 55, 50, 45]$
- **Goal:** Find the **7th smallest element** $(k = 7)$.
- The algorithm repeatedly calls `Partition` to rearrange the array until it finds the 7th smallest element.

Step 1: First Partition Call

Partition(1, 9)

- Pivot element: 65
- After partitioning, the array becomes:

$$a = [45, 55, 50, 60, 65, 80, 85, 75, 70]$$

- Pivot 65 is placed at position 5.
- Since $k = 7$ and $7 > 5$, the 7th smallest element must be in the **right half**.
- Next call: **Partition(6, 10)**

Step 2: Second Partition Call

Partition(6, 10)

- Subarray to partition: $a[6:10] = [80, 85, 75, 70, \infty]$
- Pivot element: 80
- After partitioning, the array becomes:

$$a[6:10] = [70, 75, 80, 85, \infty]$$

- Pivot 80 is placed at position 8.
- Since $k = 7$ and $7 < 8$, search continues in the **left part**.
- Next call: **Partition(7, 8)**

Step 3: Third Partition Call

Partition(7, 8)

- Subarray to partition: $a[7:8] = [75, 70]$
- Pivot element: 75
- After partitioning, the array becomes:

$$a[7:8] = [70, 75]$$

- Pivot 75 is placed at position 8.
- The 7th smallest element is found at position 7: $a[7] = 70$.

Summary of Execution

- **Initial Array:**

$$[65, 70, 75, 80, 85, 60, 55, 50, 45]$$

- **After 1st Partition:**

$$[45, 55, 50, 60, 65, 80, 85, 75, 70]$$

- **After 2nd Partition:**

$$[70, 75, 80, 85, \infty]$$

- **After 3rd Partition:**

$$[70, 75]$$

- The 7th smallest element is 70.

Time Complexity Analysis of QuickSelect

- The worst-case time complexity is $O(n^2)$.
- In the worst case, the partitioning process might reduce the search space by only one element at each step.
- Average-case time complexity is $O(n)$.

Partition Function and Assumptions

- The Partition function is based on:
  - All elements in the input are distinct.
  - The partitioning element has an equal probability of being any element.
- The time for each partition call is $O(p - m)$, where p and m are the current boundaries.
- On each call:
  - Either the lower bound increases by at least one.
  - Or the upper bound decreases by at least one.
- At most n partition calls are made.

Introduction to Matrix Multiplication

- Given two matrices A and B of size $n \times n$, the product $C = A \times B$ is also an $n \times n$ matrix.
- Each element of C is calculated as:

$$C(i,j) = \sum_{k=1}^{n} A(i,k) \cdot B(k,j)$$

- This conventional algorithm takes $\Theta(n^3)$ time due to the three nested loops.

Divide-and-Conquer Strategy

Matrix Multiplication using Submatrices

- Assume that n is a power of 2. If not, pad the matrices with zeros to the nearest power of 2.
- Split each matrix into four equal-sized submatrices of size $\frac{n}{2} \times \frac{n}{2}$:

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}, \quad B = \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

Matrix Multiplication using Submatrices

- The product matrix C can be expressed as:

$$C = A \times B = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix}$$

- Each element is calculated as:

$$C_{11} = A_{11}B_{11} + A_{12}B_{21}$$

$$C_{12} = A_{11}B_{12} + A_{12}B_{22}$$

$$C_{21} = A_{21}B_{11} + A_{22}B_{21}$$

$$C_{22} = A_{21}B_{12} + A_{22}B_{22}$$

Matrix Multiplication using Submatrices

- Multiplication using submatrices requires:
  - 8 multiplications of $\frac{n}{2} \times \frac{n}{2}$ matrices.
  - 4 additions of $\frac{n}{2} \times \frac{n}{2}$ matrices.
- Since two $\frac{n}{2} \times \frac{n}{2}$ matrices can be added in time cn^2 for some constant c, the overall computing time $T(n)$ for the resulting divide-and-conquer algorithm is:

$$T(n) = \begin{cases} b & \text{if } n \leq 2 \\ 8T\left(\frac{n}{2}\right) + cn^2 & \text{if } n > 2 \end{cases}$$

  where b and c are constants.
- Solving this recurrence gives as $T(n) = O(n^3)$.
- **No improvement over the conventional method.**

Key Idea

- Matrix multiplications are more expensive than matrix additions ($O(n^3)$ vs $O(n^2)$). Thus, try to reformulate the equations for C_{ij} so that we have fewer multiplications.
- Strassen's algorithm introduces **7 matrix multiplications** instead of the usual 8.
- Define the following intermediate matrices:

$$
\begin{aligned}
P &= (A_{11} + A_{22})(B_{11} + B_{22}) \\
Q &= (A_{21} + A_{22})B_{11} \\
R &= A_{11}(B_{12} - B_{22}) \\
S &= A_{22}(B_{21} - B_{11}) \\
T &= (A_{11} + A_{12})B_{22} \\
U &= (A_{21} - A_{11})(B_{11} + B_{12}) \\
V &= (A_{12} - A_{22})(B_{21} + B_{22})
\end{aligned}
$$

Computing the Submatrices of C

- Using the intermediate matrices, the submatrices of C are computed as:

$$C_{11} = P + S - T + V$$

$$C_{12} = R + T$$

$$C_{21} = Q + S$$

$$C_{22} = P + R - Q + U$$

Strassen's Algorithm

Recurrence Relation for Time Complexity

- The recurrence relation for the time complexity of Strassen's algorithm is:

$$T(n) = \begin{cases} b & \text{if } n \leq 2 \\ 7T\left(\frac{n}{2}\right) + an^2 & \text{if } n > 2 \end{cases}$$

- Solving this recurrence relation gives:

$$T(n) = O\left(n^{\log_2 7}\right) \approx O(n^{2.81})$$

- Strassen's algorithm improves the time complexity over the conventional $O(n^3)$ approach.

Summary

- Strassen's algorithm is a faster way to multiply matrices by reducing the number of multiplications.
- It achieves a time complexity of $O(n^{2.81})$.
- However, the algorithm introduces more additions and subtractions, which can affect performance for smaller matrices.
- In practice, Strassen's algorithm is most effective for large matrices where multiplication dominates the computation.

Problem Statement

- Given two sorted arrays, `nums1` and `nums2`.
- The goal is to find the **median** of the combined array without explicitly merging them.
- **Challenge:** Achieve an efficient solution in $O(\log(\min(m, n)))$ time.

Key Idea: Binary Search for Partitioning

- Instead of merging the arrays, use binary search to find the correct partition.
- The partition ensures:
    - The left part contains the smaller half of elements.
    - The right part contains the larger half of elements.
- Once partitioned correctly, the median is computed based on elements around the partition.

Step 1: Search on the Smaller Array

- Let `m` be the length of the smaller array (`nums1`) and `n` the length of the larger array (`nums2`).
- If `nums1` is larger, swap them to always search on the smaller array.
- This ensures the algorithm runs in $O(\log(\min(m, n)))$ time.

Step 2: Binary Search Setup

- Define the search space on `nums1`:
  - Let i be the partition in `nums1`.
  - Let $j = \frac{m+n+1}{2} - i$ be the partition in `nums2`.
- This divides both arrays into two parts:
  - Left part: `nums1[0..i-1]` and `nums2[0..j-1]`
  - Right part: `nums1[i..m-1]` and `nums2[j..n-1]`

Weekly3: Median of Two Sorted Arrays

Step 3: Adjusting the Partition

- The partition is valid if:

$$\max(\text{nums1}[i-1], \text{nums2}[j-1]) \leq \min(\text{nums1}[i], \text{nums2}[j])$$

- If the partition is not valid:
  - If $nums1[i] < nums2[j-1]$, increase i (move partition right).
  - If $nums1[i-1] > nums2[j]$, decrease i (move partition left).

Weekly3: Median of Two Sorted Arrays

Step 4: Calculating the Median

- Once a valid partition is found:
  - **If the total number of elements is odd:**

$$\text{Median} = \max(\text{nums1}[i-1], \text{nums2}[j-1])$$

  - **If the total number of elements is even:**

$$\text{Median} = \frac{\max(\text{nums1}[i-1], \text{nums2}[j-1]) + \min(\text{nums1}[i], \text{nums2}[j])}{2}$$

Median of Two Sorted Arrays: Example

- **Given:**

$$\text{nums1} = [1, 3, 4, 5, 7], \quad \text{nums2} = [2, 6, 8]$$

- **Goal:** Find the median of the two arrays without merging them.
- **Idea:**Perform binary search on the smaller array to efficiently partition the two arrays.

Median of Two Sorted Arrays: Example

Step 1: Identify the Smaller Array

- Since `nums2` is smaller, we swap the arrays.
- Now:
  - `nums1 = [2, 6, 8]`
  - `nums2 = [1, 3, 4, 5, 7]`
- Perform binary search on `nums1` (the smaller array).

Step 2: Binary Search Setup

- Total elements: $m + n = 8$.
- Half the total length:

$$\text{half\_len} = \frac{m+n+1}{2} = 4$$

- Perform binary search on `nums1` to find the correct partition.

Median of Two Sorted Arrays: Example

Step 3: First Partition Call

- Set $i = 1$, $j = 3$.
- Partition the arrays:
  - **Left part:** `nums1[0:1] = [2]`, `nums2[0:3] = [1, 3, 4]`
  - **Right part:** `nums1[1:3] = [6, 8]`, `nums2[3:5] = [5, 7]`

Median of Two Sorted Arrays: Example

Step 4: Check Partition Validity

- Compute:

$$\max(\text{nums1}[i-1], \text{nums2}[j-1]) = \max(2, 4) = 4$$

$$\min(\text{nums1}[i], \text{nums2}[j]) = \min(6, 5) = 5$$

- Since $4 \leq 5$, the partition is valid.

Median of Two Sorted Arrays: Example

Step 5: Calculate the Median

- Total number of elements is even, so:

$$\text{Median} = \frac{\max(2,4) + \min(6,5)}{2} = \frac{4+5}{2} = 4.5$$