Design and Analysis of Algorithm [R1UC407B]

Module-III: Divide and Conquer

Dr. A K Yadav

GALGOTIAS UNIVERSITY

School of Computer Science and Engineering
Plat No 2, Sector 17A, Yamuna Expressway
Greater Noida, Uttar Pradesh - 203201

February 14, 2025

Contents

Divide and Conquer Technique

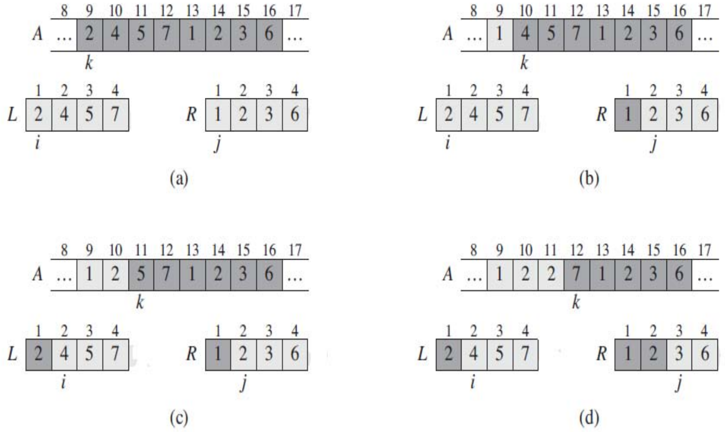
- The divide-and-conquer technique involves taking a large-scale problem and dividing it into similar sub-problems of a smaller scale.
- Each sub-problem is a non-overlapping sub-problems.
- Solve each of these sub-problems recursively.
- Generally, a problem is divided into sub problems repeatedly until the resulting sub-problems are very easy to solve.
- This technique can be divided into the following three parts:
- **Divide:** This involves dividing the problem into smaller sub-problems.
- **Conquer:** Solve sub-problems by calling recursively until solved.
- **Combine:** Combine the sub-problems to get the final solution of the whole problem.

Merge Sort

```
MERGE-SORT(A, p, r)
  if p < r
      q = (p + r)/2
      MERGE-SORT(A, p, q)
      MERGE-SORT(A, q+1, r)
      MERGE(A,p,q,r)
MERGE(A, p, q, r)
  n_1 = q − p + 1
  n_2 = r − q
  Let L[1..n_1] and R[1..n_2] be new arrays
  for i = 1 to n_1
      L[i] = A[p + i − 1]
  for j = 1 to n_2
```

```
        R[j] = A[q + j]
i = 1
j = 1
for k = p to r
    if L[i] ≤ R[j]
        A[k] = L[i]
        i = i + 1
    else
        A[k] = R[j]
        j = j + 1
```

Working of Merge



Analysis:

- First call of merge-sort will be MERGE-SORT(A, 1, n) for input size n.
- q is half of n,$q = \frac{n}{2}$ in step 2.
- Every call of merge-sort divides the size in half and double the sub-problems.
- So there will be only $\lg n$ calls of merge-sort and sum of size of all sub-problems is n
- There is only one call for each merge-sort call
- So total calls of the merge will be $\lg n$
- Step 6 to 8 will be executed $\lg n$ times
- Step 9 will be executed $n_1 \lg n$ times
- Step 10 will be executed $n_1 \lg n$ times
- Step 11 will be executed $n_2 \lg n$ times
- Step 12 will be executed $n_2 \lg n$ times

- Total of Step 9 to 12 will be executed $2(n_1 + n_2) \lg n$ times
- Step 13 & 14 will be executed $\lg n$ times
- For each call of $\lg n$ step 15 will be executed $n \lg n$ times
- Step 16 will be executed $n \lg n$ times
- Every time either step 17 & 18 or step 20 & 21 will be executed and sum of these will be $n \lg n$.
- After adding cost of all these step $T(n) = an \lg n + bn + c = O(n \lg n)$

Quick Sort

```
QUICK-SORT(A, p, r)
  if p < r
      q=PARTITION(A,p,r)
      QUICK-SORT(A, p, q-1)
      QUICK-SORT(A, q+1, r)
PARTITION(A, p, r)
  x = A[r]
  i = p
  for j = p to r − 1
      if A[j] ≤ x
          if(i ≠ j) swap(A[i], A[j])
          i = i + 1
  swap(A[i], A[r])
```

12 return i

Analysis:

- The number of calls of PARTITION depends on QUICK-SORT
- The number of calls of QUICK-SORT depends on q
- If PARTITION returns q as middle value for every call
- Then array will be divided in two half every time
- So after $\lg n$ times, $p = r$ and process will terminate.
- This will be **Best Case** of the algorithm.
- $\therefore T(n) = O(n \lg n)$ for best case
- But if PARTITION find $A[r]$ fit at its current location after checking p to r-1 elements then it return q as an end position and making partition of size 0 and $n - 1$.
- So every time the array size is reduced by only 1.

- The process will terminate after n operations.
- This will be **Worst Case** of the algorithm and happens when input is already sorted.
- Step 7 will be executed n times for every value of q in step 2.
- $\therefore T(n) = O(n^2)$ for worst case

Heap Sort

- Like merge sort but unlike insertion sort, Heap sort running time is $O(nlogn)$.
- Like insertion sort but unlike merge sort, heap sort sorts in place: only a constant number of array elements are stored outside the input array at any time.
- So heap sort combines the better attributes of the two sorting algorithms.

QUICK-SORT(A, p, r)

1 if $p < r$

 2 q=PARTITION(A,p,r)
 3 QUICK-SORT(A, p, q-1)
 4 QUICK-SORT(A, q+1, r)

PARTITION(A, p, r)

$x = A[r]$

$i = p$

for $j = p$ to $r - 1$

    if $A[j] \leq x$

        if$(i \neq j)$ swap$(A[i], A[j])$

        $i = i + 1$

swap$(A[i], A[r])$

return i

Analysis:

- The number of calls of PARTITION depends on QUICK-SORT
- The number of calls of QUICK-SORT depends on q
- If PARTITION returns q as middle value for every call
- Then array will be divided in two half every time

- So after $\lg n$ times, $p = r$ and process will terminate.
- This will be **Best Case** of the algorithm.
- $\therefore T(n) = O(n \lg n)$ for best case
- But if PARTITION find $A[r]$ fit at its current location after checking p to r-1 elements then it return q as an end position and making partition of size 0 and $n - 1$.
- So every time the array size is reduced by only 1.
- The process will terminate after n operations.
- This will be **Worst Case** of the algorithm and happens when input is already sorted.
- Step 7 will be executed n times for every value of q in step 2.
- $\therefore T(n) = O(n^2)$ for worst case

Randomized Quick Sort

- To avoid the worst performance of the Quick Sort for sorted input, we use Randomized Quick Sort.
- In this we choose pivot element randomly.

Algorithm:

RANDOMIZED-QUICK-SORT(A, p, r)

```
if p < r
    q=RANDOMIZED-PARTITION(A,p,r)
    RANDOMIZED-QUICK-SORT(A, p, q-1)
    RANDOMIZED-QUICK-SORT(A, q+1, r)
```

RANDOMIZED-PARTITION(A, p, r)

```
i = Random(p, r)
swap(A[i], A[r])
x = A[r]
```

```
i = p
for j = p to r − 1
    if A[j] ≤ x
        if(i ≠ j) swap(A[i], A[j])
        i = i + 1
swap(A[i], A[r])
return i
```

This algorithm performs worst case only when $Random(p, r)$ always gives the location of the largest/smallest number which is very rare.

Strassen's algorithm for Matrix Multiplications

- if we multiply two matrices $C_{n\times m} = A_{n\times l}B_{l\times m}$
- Total number of scaler multiplications will be $n \times l \times m$
- If we take matrices of size $n \times n$ where n is power of 2 i.e $n = 2^i$
- Total number of scaler multiplications will be n^3
- If we divide size by 2 then the matrices will be

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}$$

$$B = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}$$

$$C = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}$$

- $$\begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}$$
- $$C_{11} = A_{11}.B_{11} + A_{12}.B_{21}$$
  $$C_{12} = A_{11}.B_{12} + A_{12}.B_{22}$$
  $$C_{21} = A_{21}.B_{11} + A_{22}.B_{21}$$
  $$C_{22} = A_{21}.B_{12} + A_{22}.B_{22}$$
- So there are 8 multiplication and 4 submissions of size $n/2$

- $$T(n) = 8T\left(\frac{n}{2}\right) + 4(n/2)^2$$

- Solution of the recurrence will be $\Theta(n^3)$ using master theorem. So no benefit of dividing the main problem in subproblems.
- Strassen's gives 18 sum and 7 products of size $n/2$ as follows:

$$S_1 = B_{12} - B_{22}$$

$$S_2 = A_{11} + A_{12}$$

$$S_3 = A_{21} + A_{22}$$

$$S_4 = B_{21} - B_{11}$$

$$S_5 = A_{11} + A_{22}$$

$$S_6 = B_{11} + B_{22}$$

$$S_7 = A_{12} - A_{22}$$

$$S_8 = B_{21} + B_{22}$$

$$S_9 = A_{11} - A_{21}$$

$$S_{10} = B_{11} + B_{12}$$

- 7 products will be

$$P_1 = A_{11}.S_1$$

$$P_2 = S_2.B_{22}$$

$$P_3 = S_3.B_{11}$$

$$P_4 = A_{22}.S_4$$

$$P_5 = S_5.S_6$$

$$P_6 = S_7.S_8$$

$$P_7 = S_9.S_{10}$$

- Now the the matrix will be

$$C_{11} = P_5 + P_4 - P_2 + P_6$$

$$C_{12} = P_1 + P_2$$

$$C_{21} = P_3 + P_4$$

$$C_{22} = P_5 + P_1 - P_3 - P_7$$

- There are 7 multiplication and 18 submissions of size $n/2$
-

$$T(n) = 7T\left(\frac{n}{2}\right) + 18(n/2)^2$$

- The solution of the recurrence will be $\Theta(n^{\lg 7})$ using master theorem which is less then $\Theta(n^3)$

- So Strassen's algorithm is faster then divide and conquer.
- Ques.1 Find the product of the following matrices using Strassen's algorithm.

$$\begin{pmatrix} 1 & 3 \\ 7 & 5 \end{pmatrix} \begin{pmatrix} 6 & 8 \\ 4 & 2 \end{pmatrix}$$

Medians and Order Statistics

- The i^{th} **Order Statistic** of a set of n elements is the i^{th} smallest element.
- The 1^{st} **Order Statistic** is the **minimum** of the set.
- The n^{th} **Order Statistic** is the **maximum** of the set.
- **Median** is the middle element of the set.
- If n is odd then unique median will be at $(n+1)/2$
- If n is even then two median **lower** and **upper** will be at $n/2$ and $n/2+1$ respectively.
- So **lower median** will be at $\lfloor (n+1)/2 \rfloor$ and **upper median** will be at $\lceil (n+1)/2 \rceil$ for both n even or odd.

Thank you

Please send your feedback or any queries to
ashokyadav@galgotiasuniversity.edu.in