Advanced Analysis of Algorithms

Dr. Qaiser Abbas
Department of Computer Science & IT,
University of Sargodha, Sargodha, 40100, Pakistan
qaiser.abbas@uos.edu.pk

Divide and Conquer

- Like Greedy and Dynamic Programming, Divide and Conquer is an algorithmic paradigm. A typical Divide and Conquer algorithm solves a problem using following three steps.
  - **1.** *Divide:* Break the given problem into subproblems of same type.
  - **2.** *Conquer:* Recursively solve these subproblems
  - **3.** *Combine:* Appropriately combine the answers

Divide and Conquer

- Following are some standard algorithms that are Divide and Conquer algorithms.
  - **1)** **Binary Search** is a searching algorithm. In each step, the algorithm compares the input element x with the value of the middle element in array. If the values match, return the index of middle. Otherwise, if x is less than the middle element, then the algorithm recurs for left side of middle element, else recurs for right side of middle element.
  - **2)** **Quicksort** is a sorting algorithm. The algorithm picks a pivot element, rearranges the array elements in such a way that all elements smaller than the picked pivot element move to left side of pivot, and all greater elements move to right side. Finally, the algorithm recursively sorts the subarrays on left and right of pivot element.

Divide and Conquer

- **3) Merge Sort** is also a sorting algorithm. The algorithm divides the array in two halves, recursively sorts them and finally merges the two sorted halves.
- **4) Closest Pair of Points** The problem is to find the closest pair of points in a set of points in x-y plane. The problem can be solved in O(n^2) time by calculating distances of every pair of points and comparing the distances to find the minimum. The Divide and Conquer algorithm solves the problem in O(nLogn) time.
- **5) Strassen's Algorithm** is an efficient algorithm to multiply two matrices. A simple method to multiply two matrices need 3 nested loops and is O(n^3). Strassen's algorithm multiplies two matrices in O(n^2.8974) time.

Divide and Conquer

- 6) **Cooley–Tukey Fast Fourier Transform (FFT) algorithm** is the most common algorithm for FFT. It is a divide and conquer algorithm which works in O(nlogn) time.
- 7) **Karatsuba algorithm for fast multiplication** it does multiplication of two *n*-digit numbers in at most single-digit multiplications in general (and exactly when *n* is a power of 2). It is therefore faster than the classical algorithm, which requires n^2 single-digit products. If $n = 2^{10} = 1024$, in particular, the exact counts are $3^{10} = 59,049$ and $(2^{10})^2 = 1,048,576$, respectively.

• We will study some of them in separate lectures. Binary and Merge sort (read it yourself) because it was the part of "Fundamentals of Algorithms" course.

Divide and Conquer

- ***Divide and Conquer (D & C) vs Dynamic Programming (DP)***
  Both paradigms (D & C and DP) divide the given problem into subproblems and solve subproblems. How to choose one of them for a given problem?
- Divide and Conquer should be used when same subproblems are not evaluated many times. Otherwise Dynamic Programming or Memoization should be used.
- For example, Binary Search is a Divide and Conquer algorithm, we never evaluate the same subproblems again. On the other hand, for optimal BST, Dynamic Programming should be preferred (See previous lectures for details).

Quick Sort

- Like Merge Sort, QuickSort is a Divide and Conquer algorithm.
- It picks an element as pivot and partitions the given array around the picked pivot. There are many different versions of quickSort that pick pivot in different ways.
  - 1) Always pick first element as pivot.
  - 2) Always pick last element as pivot (as in this lecture)
  - 3) Pick a random element as pivot.
  - 4) Pick median as pivot.
- The key process in quickSort is partition().
- Target of partitions is, given an array and an element x of array as pivot, put x at its correct position in sorted array and put all smaller elements (smaller than x) before x, and put all greater elements (greater than x) after x. All this should be done in linear time.

Quick Sort

- Here is the three step divide-and-conquer process for sorting a typical subarray A[p…r]:
  - **Divide:** Partition (rearrange) the array A[p…r] into two (possibly empty) subarrays A[p…q-1] and A[q+1…r] such that each element of A[p…q-1] is less than or equal to A[q], which is, in turn, less than or equal to each element of A[q+1…r]. Compute the index q as part of this partitioning procedure.
  - **Conquer:** Sort the two subarrays A[p…q-1] and A[q+1…r] by recursive calls to quicksort.
  - **Combine:** Because the subarrays are already sorted, no work is needed to combine them: the entire array A[p…r] is now sorted.

Quick Sort

- The following procedure implements quicksort:

```
QUICKSORT(A, p, r)
1  if p < r
2      q = PARTITION(A, p, r)
3      QUICKSORT(A, p, q - 1)
4      QUICKSORT(A, q + 1, r)
```

```
PARTITION(A, p, r)
1  x = A[r]
2  i = p - 1
3  for j = p to r - 1
4      if A[j] ≤ x
5          i = i + 1
6          exchange A[i] with A[j]
7  exchange A[i + 1] with A[r]
8  return i + 1
```

- To sort an entire array A, the initial call is QUICKSORT(A,1,A.length())
- The key to the algorithm is the PARTITION procedure, which rearranges the subarray A[p…r] in place.
- The running time of PARTITION on the subarray A[p…r] is O(n)

Quick Sort

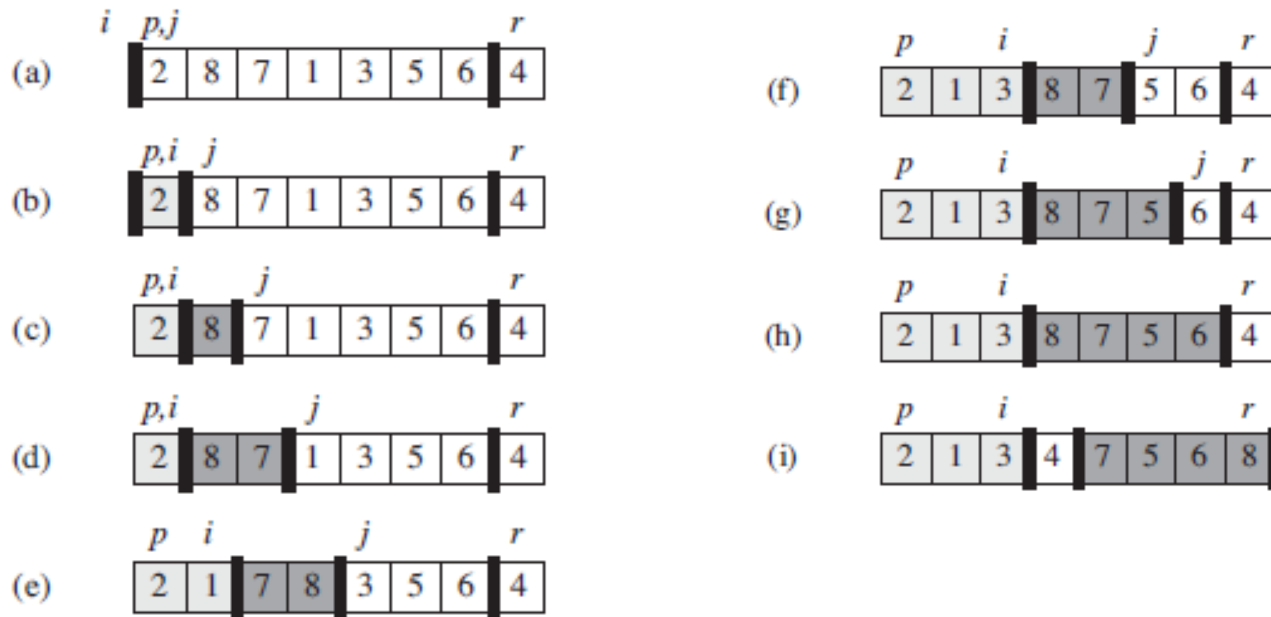

Figure 7.1 The operation of PARTITION on a sample array. Array entry $A[r]$ becomes the pivot element x. Lightly shaded array elements are all in the first partition with values no greater than x. Heavily shaded elements are in the second partition with values greater than x. The unshaded elements have not yet been put in one of the first two partitions, and the final white element is the pivot x. **(a)** The initial array and variable settings. None of the elements have been placed in either of the first two partitions. **(b)** The value 2 is "swapped with itself" and put in the partition of smaller values. **(c)–(d)** The values 8 and 7 are added to the partition of larger values. **(e)** The values 1 and 8 are swapped, and the smaller partition grows. **(f)** The values 3 and 7 are swapped, and the smaller partition grows. **(g)–(h)** The larger partition grows to include 5 and 6, and the loop terminates. **(i)** In lines 7–8, the pivot element is swapped so that it lies between the two partitions.

Quick Sort

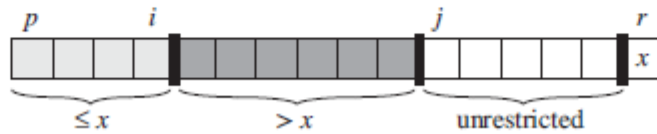

Figure 7.2 The four regions maintained by the procedure PARTITION on a subarray $A[p..r]$. The values in $A[p..i]$ are all less than or equal to x, the values in $A[i+1..j-1]$ are all greater than x, and $A[r] = x$. The subarray $A[j..r-1]$ can take on any values.

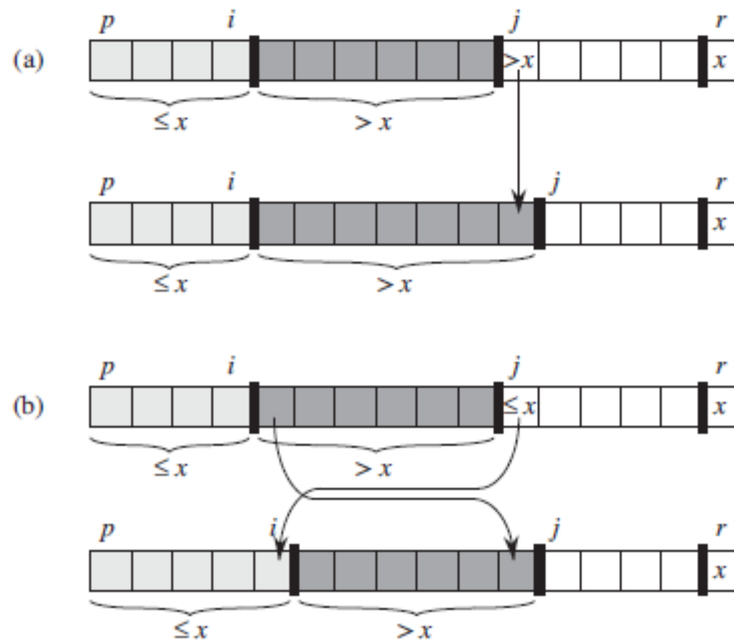

Figure 7.3 The two cases for one iteration of procedure PARTITION. **(a)** If $A[j] > x$, the only action is to increment j, which maintains the loop invariant. **(b)** If $A[j] \le x$, index i is incremented, $A[i]$ and $A[j]$ are swapped, and then j is incremented. Again, the loop invariant is maintained.

Quick Sort

- **Analysis of QuickSort**: Time taken by QuickSort in general can be written as following.
  - $T(n) = T(k) + T(n-k-1) + O(n)$
  - The first two terms are for two recursive calls, the last term is for the partition process.
  - k is the number of elements which are smaller than pivot.
- The time taken by QuickSort depends upon the input array and partition strategy.

Quick Sort

- ***Worst Case:*** The worst case occurs when the partition process always picks greatest or smallest element as pivot.
- If we consider previous partition strategy where last element is always picked as pivot, the worst case would occur when the array is already sorted in increasing or decreasing order.
- In this case, partitioning produces one subproblem with n-1 elements and one with 0 elements. So following would be the recurrence for worst case.
  - $T(n) = T(0) + T(n-1) + O(n)$
  - which is equivalent to $T(n) = T(n-1) + O(n)$
  - The solution of above recurrence is $O(n^2)$ by eq. of Arithmetic Series in A.2

Quick Sort

- ***Best Case:*** In the most even possible split, PARTITION produces two subproblems, each of size no more than n/2, since one is of size $\lfloor n/2 \rfloor$ and one of size $\lceil n/2 \rceil$-1.
- In this case, quicksort runs much faster Or the best case occurs when the partition process always picks the middle element as pivot.
- Following is recurrence for best case.
  - $T(n) = 2T(n/2) + O(n)$
  - The solution of above recurrence is O(nLogn). It can be solved using case 2 of Master Theorem.

Quick Sort

- Although the worst case time complexity of QuickSort is $O(n^2)$ which is more than many other sorting algorithms like Merge Sort and Heap Sort.
- QuickSort is faster in practice, because its inner loop can be efficiently implemented on most architectures, and in most real-world data.
- QuickSort can be implemented in different ways by changing the choice of pivot, so that the worst case rarely occurs for a given type of data..

Strassen's Matrix Multiplication

- Given two square matrices A and B of size n x n each, find their multiplication matrix.

Naive Method

Following is a simple way to multiply two matrices.

```
void multiply(int A[][N], int B[][N], int C[][N])
{
    for (int i = 0; i < N; i++)
    {
        for (int j = 0; j < N; j++)
        {
            C[i][j] = 0;
            for (int k = 0; k < N; k++)
            {
                C[i][j] += A[i][k]*B[k][j];
            }
        }
    }
}
```

Time Complexity of above method is $O(N^3)$.

Strassen's Matrix Multiplication

- ***Divide and Conquer*** : Following is simple Divide and Conquer method to multiply two square matrices.
  - Divide matrices A and B in 4 sub-matrices of size N/2 x N/2 as shown in the below diagram.
  - Calculate following values recursively. ae + bg, af + bh, ce + dg and cf + dh.

$$\underset{A}{\begin{bmatrix} a & b \\ c & d \end{bmatrix}} \times \underset{B}{\begin{bmatrix} e & f \\ g & h \end{bmatrix}} = \underset{C}{\begin{bmatrix} ae + bg & af + bh \\ ce + dg & cf + dh \end{bmatrix}}$$

A, B and C are square metrices of size N x N
a, b, c and d are submatrices of A, of size N/2 x N/2
e, f, g and h are submatrices of B, of size N/2 x N/2

Strassen's Matrix Multiplication

```
SQUARE-MATRIX-MULTIPLY-RECURSIVE(A, B)
1   n = A.rows
2   let C be a new n × n matrix
3   if n == 1
4       c11 = a11 · b11
5   else partition A, B, and C as in equations (4.9)
6       C11 = SQUARE-MATRIX-MULTIPLY-RECURSIVE(A11, B11)
              + SQUARE-MATRIX-MULTIPLY-RECURSIVE(A12, B21)
7       C12 = SQUARE-MATRIX-MULTIPLY-RECURSIVE(A11, B12)
              + SQUARE-MATRIX-MULTIPLY-RECURSIVE(A12, B22)
8       C21 = SQUARE-MATRIX-MULTIPLY-RECURSIVE(A21, B11)
              + SQUARE-MATRIX-MULTIPLY-RECURSIVE(A22, B21)
9       C22 = SQUARE-MATRIX-MULTIPLY-RECURSIVE(A21, B12)
              + SQUARE-MATRIX-MULTIPLY-RECURSIVE(A22, B22)
10  return C
```

Strassen's Matrix Multiplication

- In the previous method, we do 8 multiplications for matrices of size N/2 x N/2 and 4 additions. Addition of two matrices takes $O(N^2)$ time. So the time complexity can be written as
  - $T(N) = 8T(N/2) + O(N^2)$
  - From Master's Theorem (4.5), time complexity of above method is $O(N^3)$ which is unfortunately same as the above naive method.

Strassen's Matrix Multiplication

- ***Simple Divide and Conquer also leads to O(N^3), can there be a better way?***
  - In the previous divide and conquer method, the main component for high time complexity is 8 recursive calls.
  - The idea of **Strassen's method** is to reduce the number of recursive calls to 7.
  - Strassen's method is similar to previous simple divide and conquer method in the sense that this method also divide matrices to sub-matrices of size N/2 x N/2 as shown in the previous diagram, but in Strassen's method, the four sub-matrices of result are calculated using following formulae.

Strassen's Matrix Multiplication

p1 = a(f - h)
p2 = (a + b)h
p3 = (c + d)e
p4 = d(g - e)
p5 = (a + d)(e + h)
p6 = (b - d)(g + h)
p7 = (a - c)(e + f)

The A x B can be calculated using above seven multiplications.
Following are values of four sub-matrices of result C

$$\underbrace{\begin{bmatrix} a & b \\ c & d \end{bmatrix}}_{A} \times \underbrace{\begin{bmatrix} e & f \\ g & h \end{bmatrix}}_{B} = \underbrace{\begin{bmatrix} p5 + p4 - p2 + p6 & p1 + p2 \\ p3 + p4 & p1 + p5 - p3 - p7 \end{bmatrix}}_{C}$$

A, B and C are square metrices of size N x N
a, b, c and d are submatrices of A, of size N/2 x N/2
e, f, g and h are submatrices of B, of size N/2 x N/2
p1, p2, p3, p4, p5, p6 and p7 are submatrices of size N/2 x N/2

Strassen's Matrix Multiplication

- **Time Complexity of Strassen's Method**: Addition and Subtraction of two matrices takes $O(N^2)$ time. So time complexity can be written as
  - $T(N) = 7T(N/2) + O(N^2)$
  - From Master's Theorem, time complexity of above method is $O(N^{Log7})$ which is approximately $O(N^{2.8074})$

Strassen’s Matrix Multiplication

- Generally Strassen’s Method is not preferred for practical applications for following reasons.
  - The constants used in Strassen’s method are high and for a typical application Naive method works better.
  - For Sparse matrices, there are better methods especially designed for them.
  - The sub-matrices in recursion take extra space.
  - Because of the limited precision of computer arithmetic on non-integer values, larger errors accumulate in Strassen’s algorithm than in Naive Method.

Home Work # 5

7.1-1

Using Figure 7.1 as a model, illustrate the operation of PARTITION on the array $A = \langle 13, 19, 9, 5, 12, 8, 7, 4, 21, 2, 6, 11 \rangle$.

7.1-3

Give a brief argument that the running time of PARTITION on a subarray of size n is $\Theta(n)$.

7.2-3

Show that the running time of QUICKSORT is $\Theta(n^2)$ when the array A contains distinct elements and is sorted in decreasing order.

4.2-2

Write pseudocode for Strassen's algorithm.

4.2-3

How would you modify Strassen's algorithm to multiply $n \times n$ matrices in which n is not an exact power of 2? Show that the resulting algorithm runs in time $\Theta(n^{\lg 7})$.