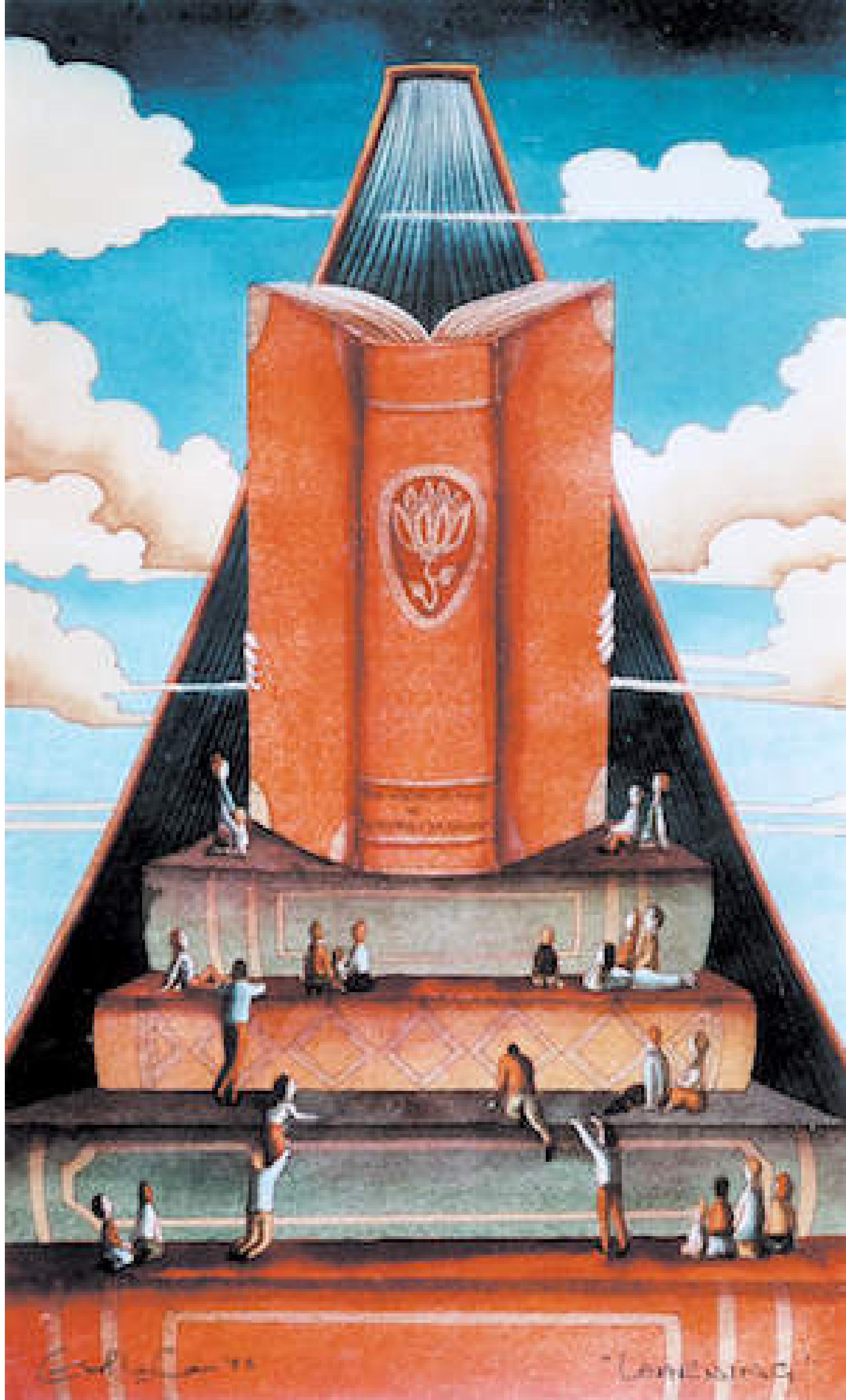
DIVIDE AND CONQUER



Outlines

- Introduction
- Binary Search
- Mergesort
- Quicksort
- Strassen's algorithm for matrix multiplication

Introduction

- **Divide**: Divide the problem to subproblems
- **Conquer**: Solve recursively subproblems
- **Combine**: Use results of subproblems and combine them to obtain result of initial problem
- **Determine Threshold**: for which problem, the algorithm return directly result without dividing to smaller problems

Binary Search

- May only be used on a sorted array
- Eliminates one half of the elements after each comparison
- Locate the middle of the array
- Compare the value at that location with the search key.
- If they are equal - done! Otherwise, decide which half of the array contains the search key.
- Repeat the search on that half of the array and ignore the other half.
- The search continues until the key is matched or no elements remain to be searched.

Binary Search (Cont..)

- Ex. Binary search for 33.

6	**13**	**14**	**25**	**33**	**43**	**51**	**53**	**64**	**72**	**84**	**93**	**95**	**96**	**97**
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
↑ lo														↑ hi

6	**13**	**14**	**25**	**33**	**43**	**51**	**53**	**64**	**72**	**84**	**93**	**95**	**96**	**97**
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
↑ lo							↑ mid							↑ hi

6	**13**	**14**	**25**	**33**	**43**	**51**	53	64	72	84	93	95	96	97
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
↑ lo						↑ hi								

Binary Search (Cont..)

- Ex. Binary search for 33.

6	13	14	25	33	43	51	53	64	72	84	93	95	96	97
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
↑ lo			↑ mid			↑ hi								

6	13	14	25	33	43	51	53	64	72	84	93	95	96	97
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
				↑ lo		↑ hi								

6	13	14	25	33	43	51	53	64	72	84	93	95	96	97
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
				↑ lo	↑ mid	↑ hi								

Binary Search (Cont..)

- Ex. Binary search for 33.

6	13	14	25	33	43	51	53	64	72	84	93	95	96	97
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

↑
lo
hi

6	13	14	25	33	43	51	53	64	72	84	93	95	96	97
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

↑
lo
hi
mid

6	13	14	25	33	43	51	53	64	72	84	93	95	96	97
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

↑
lo
hi
mid

Binary Search (Cont..)

```
BINARY_SEARCH(A, n, key, index)
    low = 1; high = n
    while (low <= high) {
        mid = (low + high) / 2
        if (key < A [mid])
            high = mid -1;      // search low end of array
        else if (key>A[mid])
            low = mid + 1;  // search high end of array
        else { index = mid; return}
End BINARY_SEARCH
```

Merge Sort

- Divide: Divide the n-element sequence to be sorted into two subsequences of n/2 elements each.
- Conquer: Sort the two subsequences recursively using merge sort.
- Combine: Merge the two sorted subsequences to produce the sorted answer.

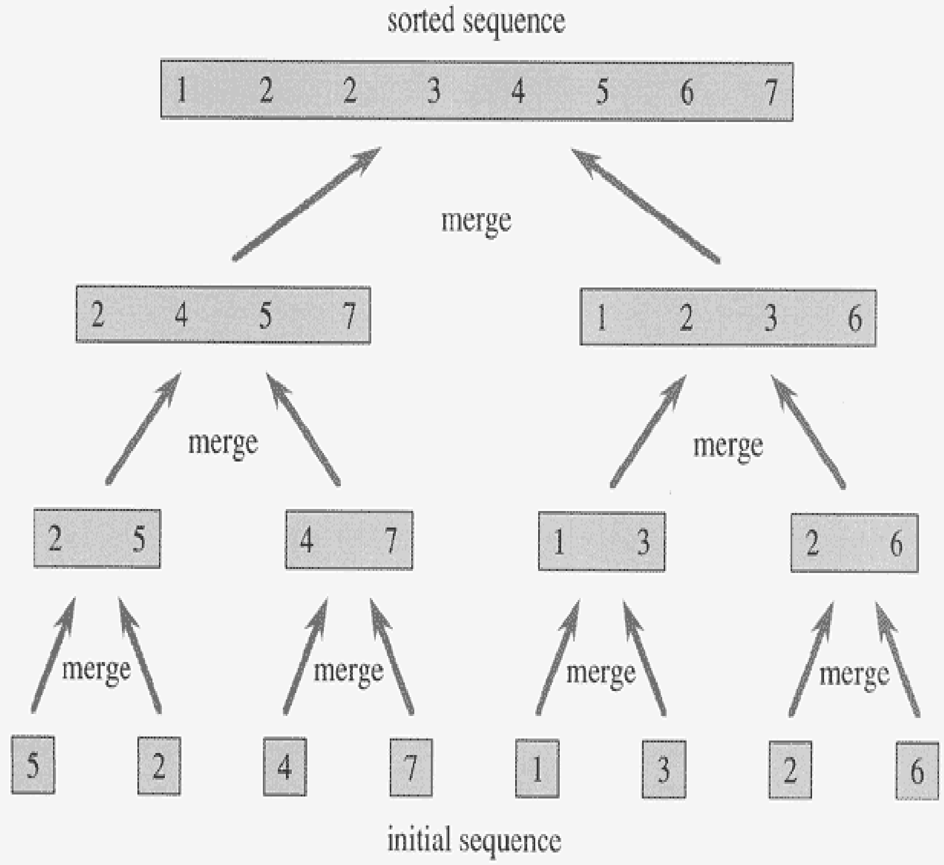


Figure 2.4 The operation of merge sort on the array $A = \langle 5, 2, 4, 7, 1, 3, 2, 6 \rangle$. The lengths of the sorted sequences being merged increase as the algorithm progresses from bottom to top.

Merge Sort (Cont..)

```
MERGE-SORT(A, low, high)
        if low < high then
                mid ← ⌊(low+high)/2⌋
                MERGE-SORT(A, low, mid)
                MERGE-SORT(A, mid + 1, high)
                MERGE(A, low, mid, high)
        end if
End MERGE-SORT
```

```
MERGE (A, low, mid, high)
        h = i = low; j = mid +1
        while h ≤ mid and j ≤ high do
                if A[h] ≤ A[j] then
                        B[i] = A[h]; h = h + 1
                else
                        B[i] = A[j]; j = j + 1
                end if
                i = i + 1
        end while
        if h>mid then
                for k = j to high do //handle any remaining elements
                        B[i] = A[k]; i = i + 1
                end for
        else
                for k = h to mid do
                        B[i] = A[k]; i = i + 1
                end for
        end if
        for k = low to high do
                A[k] = B[k]
        end for
End MERGE-SORT
```

Analyzing Merge Sort

- Divide: D(n) = Θ(1).
- Conquer: solve two subproblems, each of size n/2, which contributes 2T(n/2) to the running time.
- Combine: the MERGE procedure on an n-element subarray takes time Θ(n), so C(n) = Θ(n).
- And then $T(n) = O(n \log_2 n)$

The recurrence for the worst-case running time T(n) of `MERGE-SORT`:

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ 2T(n/2) + \Theta(n) & \text{if } n > 1 \end{cases}$$

equivalently

$$T(n) = \begin{cases} c_1 & \text{if } n = 1 \\ 2T(n/2) + c_2 n & \text{if } n > 1 \end{cases}$$

Quick Sort

- 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 smaller than 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: Since the subarrays are sorted in place, no work is needed to combine them: the entire array A[p.. r] is now sorted.

Quick sort (Cont..)

```
QUICKSORT(A, p, r)
    if p < r then
        q = PARTITION(A, p, r)
        QUICKSORT(A, p, q - 1)
        QUICKSORT(A, q + 1, r)
    end if
end QUICKSORT
```

```
PARTITION(A, p, r)
    x = A[r]
    i = p-1
    for j = p to r-1 do
        if A[j] ≤ x then
            i = i + 1
            exchange (A[i], A[j])
        end if
    end for
    exchange (A[i + 1], A[r])
    return i+1
end PARTITION
```

Quick sort (Cont..)

From $i+1$ to j is a window of elements > x = A[r]. The cursor j moves right one step at a time.

If the cursor j "discovers" an element ≤ x, then this element is swapped with the front element of the window, effectively moving the window right one step; if it discovers an element > x, then the window simply becomes longer one unit.

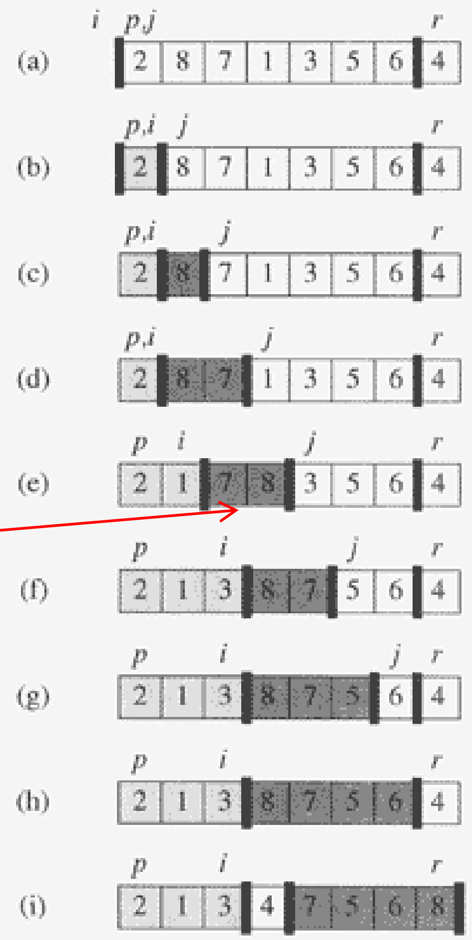


Figure 7.1 The operation of PARTITION on a sample array. Lightly shaded array elements are all in the first partition with values no greater than x. Heavily shaded elements are in the second partition with values greater than x. The unshaded elements have not yet been put in one of the first two partitions, and the final white element is the pivot. **(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 8 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.

Performance of Quicksort

- Worst-case partitioning: one subproblem of size n-1, other 0.

  Time: $\Theta(n^2)$. Why?
- Best-case partitioning: each subproblem of size at most n/2.

  Time: $\Theta(n\log n)$. Why?
- Balanced partitioning: even if each subproblem size is at least a constant proportion of the original problem the running time is $\Theta(n\log n)$.

Strassen's Algorithm for Matrix Multiplication

- If A and B are two matrix of n X n size and each element are represented by a_{ij} and b_{ij} respectively, then the product C=A.B.
- We can define each element of C as $c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}$
- We need to compute n^2 entries each is the sum of n values
- This process takes $\Theta(n^3)$ times

```
SQUARE-MATRIX-MULTIPLY(A, B)

n = A.rows
let C be a new n × n matrix
for i = 1 to n
    for j = 1 to n
        c_ij = 0
        for k = 1 to n
            c_ij = c_ij + a_ik · b_kj
return C
```

Strassen's Algorithm for Matrix Multiplication(c.)

- Strassen's method
  - Divide the input matrices A and B and output matrix C into $\frac{n}{2} \times \frac{n}{2}$ submatrices

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

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

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

  - Create 10 matrices S_1, S_2, …., S_{10}, each of which is $\frac{n}{2} \times \frac{n}{2}$ and is the sum or difference of two matrices created in step 1

$$\begin{aligned} S_1 &= B_{12} - B_{22}, & S_6 &= B_{11} + B_{22}, \\ S_2 &= A_{11} + A_{12}, & S_7 &= A_{12} - A_{22}, \\ S_3 &= A_{21} + A_{22}, & S_8 &= B_{21} + B_{22}, \\ S_4 &= B_{21} - B_{11}, & S_9 &= A_{11} - A_{21}, \\ S_5 &= A_{11} + A_{22}, & S_{10} &= B_{11} + B_{12}. \end{aligned}$$

Strassen's Algorithm for Matrix Multiplication(c.)

- Strassen's method (Cont..)
  - Using the submatrices created in step 1 and the 10 matrices created in step 2, recursively compute seven matrix products P_1, P_2,….,P_7. Each matrix P_i is $\frac{n}{2} \times \frac{n}{2}$.

$$
\begin{aligned}
P_1 &= A_{11} \cdot S_1 &&= A_{11} \cdot B_{12} - A_{11} \cdot B_{22} , \\
P_2 &= S_2 \cdot B_{22} &&= A_{11} \cdot B_{22} + A_{12} \cdot B_{22} , \\
P_3 &= S_3 \cdot B_{11} &&= A_{21} \cdot B_{11} + A_{22} \cdot B_{11} , \\
P_4 &= A_{22} \cdot S_4 &&= A_{22} \cdot B_{21} - A_{22} \cdot B_{11} , \\
P_5 &= S_5 \cdot S_6 &&= A_{11} \cdot B_{11} + A_{11} \cdot B_{22} + A_{22} \cdot B_{11} + A_{22} \cdot B_{22} , \\
P_6 &= S_7 \cdot S_8 &&= A_{12} \cdot B_{21} + A_{12} \cdot B_{22} - A_{22} \cdot B_{21} - A_{22} \cdot B_{22} , \\
P_7 &= S_9 \cdot S_{10} &&= A_{11} \cdot B_{11} + A_{11} \cdot B_{12} - A_{21} \cdot B_{11} - A_{21} \cdot B_{12} .
\end{aligned}
$$

  - Compute the desired submatrices C_{11}, C_{12}, C_{21}, C_{22} of the result matrix C by adding and subtracting various combinations of the P_i matrices.

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

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

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

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

- In case of Strassen's method $T(n) = \Theta(n^{\ln 7})$

Thanks for your Attention

Q&A