Unit 2
Divide and Conquer(D&C)

Prof. Fenil Khatiwala
Department of Computer Engineering
CGPIT, UTU

Outline

- Divide and Conquer Technique
- Multiplying large Integers Problem
- Problem Solving using divide and conquer algorithm –
  - ✓ Binary Search
  - ✓ Sorting (Merge Sort, Quick Sort)
  - ✓ Matrix Multiplication
  - ✓ Exponential

Divide and Conquer Technique

Introduction

- Many useful algorithms are recursive in structure: to solve a given problem, they call themselves recursively one or more times.
- These algorithms typically follow a divide-and-conquer approach:
- The divide-and-conquer approach involves three steps at each level of the recursion:
  1. **Divide:** Break the problem into several sub problems that are similar to the original problem but smaller in size.
  2. **Conquer:** Solve the sub problems recursively. If the sub problem sizes are small enough, just solve the sub problems in a straight forward manner.
  3. **Combine:** Combine these solutions to create a solution to the original problem.

D&C Running Time Analysis

- The **running-time analysis** of such divide-and-conquer (D&C) algorithms is almost automatic.
- Let $g(n)$ be the **time required by D&C** on instances of size n.
- The **total time $\boldsymbol{t(n)}$** taken by this divide-and-conquer algorithm is given by recurrence equation,

$$t(n) = lt(n/b) + g(n)$$

$$\boxed{\mathbf{T(n) \ = \ aT(n/b) \ + \ f(n)}}$$

- The solution of equation is given as,

$$t(n) = \begin{cases} \theta(n^k) & if\ l < b^k \\ \theta(n^k logn) & if\ l = b^k \\ \theta(n^{log_b l}) & if\ l > b^k \end{cases}$$

where k is the power of n in $g(n)$

Multiplying Large Integers Problem

Multiplying Large Integer - Introduction

- Multiplying two n digit large integers using divide and conquer method.
- Example: Multiplication of $\mathbf{981}$ by $\mathbf{1234}$.
  1. Convert both the numbers into same length nos. and split each operand into two parts:

$$\mathbf{09|81} \qquad \mathbf{12|34}$$

$$\boldsymbol{w = 09} \quad \boldsymbol{x = 81} \quad \boldsymbol{y = 12} \quad \boldsymbol{z = 34}$$

  2. We can write as,

$$10^2 w + x$$
$$= 10^2(09) + 81$$
$$= 900 + 81$$
$$=981$$

$$\boxed{\mathbf{0981} = \mathbf{10^2}\boldsymbol{w} + \boldsymbol{x}} \qquad \boxed{\mathbf{1234} = \mathbf{10^2}\boldsymbol{y} + \boldsymbol{z}}$$

Multiplying Large Integer – Example 1

- Now, the required product can be computed as,

$$\begin{aligned} 0981 \times 1234 &= (10^2 w + x) \times (10^2 y + z) \\ &= 10^4 \underline{w \cdot y} + 10^2 \boxed{(\underline{w \cdot z} + \underline{x \cdot y})} + \underline{x \cdot z} \\ &= 1080000 + 127800 + 2754 \\ &= 1210554 \end{aligned}$$

$$\boxed{\begin{aligned} w &= 09 \\ x &= 81 \\ y &= 12 \\ z &= 34 \end{aligned}}$$

- The above procedure still needs **four half-size multiplications**:

$$(i) w \cdot y \ (ii) w \cdot z \ \ (iii) x \cdot y \ (iv) x \cdot z$$

- The computation of $(\underline{w \cdot z} + \underline{x \cdot y})$ can be done as,

$$\boldsymbol{r = (w + x) \otimes (y + z) = \boxed{w \cdot y} + \boxed{(w \cdot z + x \cdot y)} + \boxed{x \cdot z}}$$

- Only **one** multiplication is required instead of two.

Additional terms

Multiplying Large Integer – Example 1

$$\boxed{\mathbf{10^4 w \cdot y + 10^2(w \cdot z + x \cdot y) + x \cdot z}}$$

- Now we can compute the required product as follows:

$w = 09$
$x = 81$
$y = 12$
$z = 34$

$$\boldsymbol{p = w \cdot y = 09 \cdot 12 = 108}$$

$$\boldsymbol{q = x \cdot z = 81 \cdot 34 = 2754}$$

$$\boldsymbol{r = (w + x) \times (y + z) = 90 \cdot 46 = 4140}$$

$$\boldsymbol{r = (w + x) \times (y + z) = w \cdot y + \boxed{(w \cdot z + x \cdot y)} + x \cdot z}$$

$$981 \times 1234 = 10^4 p + 10^2 (r - p - q) + q$$

$$= 1080000 + 127800 + 2754$$

$$= 1210554.$$

Multiplying Large Integer – Analysis

- 981 × 1234 can be reduced to **three multiplications** of two-figure numbers $(09 \cdot 12,\ 81 \cdot 34\ and\ 90 \cdot 46)$ together with a certain number of shifts, additions and subtractions.
- Reducing four multiplications to three will enable us to cut 25% of the computing time required for large multiplications.
- We obtain an algorithm that can multiply two n-figure numbers in a time,

$$\boldsymbol{T(n) = 3t\,(n/2) + g(n)},$$

$$\boxed{\boldsymbol{T(n) = aT(n/b) + f(n)}}$$

- Solving it gives,

$$\boxed{\boldsymbol{T(n) \in \theta\left(n^{lg3} \mid n\ is\ a\ power\ of\ 2\right)}}$$

Multiplying Large Integer – Example 2

- Example: Multiply **8114** with **7622** using divide & conquer method.
- Solution using D&C

Step 1: $w = 81$ $x = 14$ $y = 76$ $z = 22$

Step 2: Calculate p, q and r

$$p = w \cdot y = 81 \cdot 76 = 6156$$
$$q = x \cdot z = 14 \cdot 22 = 308$$
$$r = (w + x) \cdot (y + z) = 95 \cdot 98 = 9310$$
$$8114 \times 7622 = \mathbf{10^4 p + 10^2\,(r - p - q) + q}$$
$$= \; 61560000 + 284600 + 308$$
$$= 61844908$$

Binary Search

Introduction

- Binary Search is an extremely well-known instance of **divide-and-conquer** approach.
- Let $T[1 \dots n]$ be an array of **increasing sorted order**; that is $T[i] \leq T[j]$ whenever $1 \leq i \leq j \leq n$.
- Let x be some number. The problem consists of **finding $\boldsymbol{x}$** in the array T if it is there.
- If x is not in the array, then we want to find **the position** where it might be inserted.

Binary Search Example

Input: sorted array of integer values. $x = 7$

1	3	7	9	11	32	52	74	90

Step 1:

1	2	3	4	5	6	7	8	9
1	3	7	9	11	32	52	74	90

Find approximate midpoint

Binary Search Example

Step 2:

$x = 7$

1	2	3	4	5	6	7	8	9
1	3	7	9	11	32	52	74	90

Is 7 = midpoint value?

Step 3:

1	2	3	4	5	6	7	8	9
1	3	7	9	11	32	52	74	90

Search for the target in the area before midpoint.

Binary Search Example

$x = 7$

Step 4:

1	2	3	4	5	6	7	8	9
1	3	7	9	11	32	52	74	90

Find approximate midpoint

Step 5:

1	2	3	4	5	6	7	8	9
1	3	7	9	11	32	52	74	90

7 > value of midpoint?

Binary Search Example

$x = 7$

Step 6:

1	2	3	4	5	6	7	8	9
1	3	7	9	11	32	52	74	90

Search for the x in the area after midpoint.

Step 7:

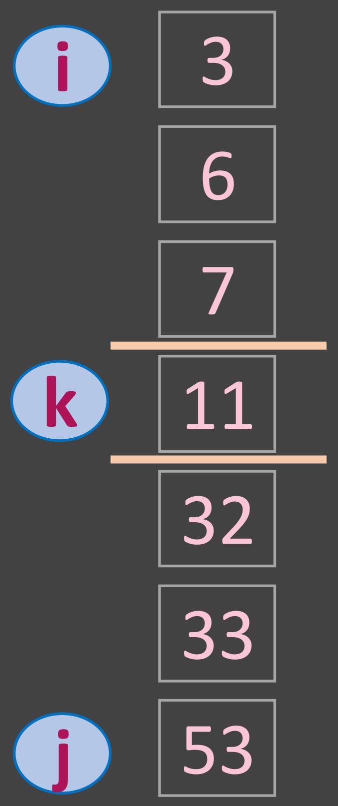
1	2	3	4	5	6	7	8	9
1	3	7	9	11	32	52	74	90

Find approximate midpoint.
Is x = midpoint value?

Binary Search – Iterative Algorithm

$n = 7$ $x = 33$

```
Algorithm: Function biniter(T[1,…,n], x)
    if x > T[n] then return n+1
    i ← 1;
    j ← n;
    while i < j do
        k ← (i + j ) ÷ 2
        if  x ≤ T [k]  then  j ← k
        else i ← k + 1
    return i
```



Binary Search – Recursive Algorithm

```
Algorithm: Function binsearch(T[1,…,n], x)
     if n = 0 or x > T[n] then return n + 1
          else return binrec(T[1,…,n], x)
     Function binrec(T[i,…,j], x)
          if i = j then return i
          k ← (i + j) ÷ 2
          if x ≤ T[k] then
               return binrec(T[i,…,k],x)
          else return binrec(T[k + 1,…,j], x)
```

Binary Search – Analysis

- Let $t(n)$ be the time required for a call on binrec($T[i, \ldots, j], x$), where $n = j - i + 1$ is the number of elements **still under consideration** in the search.
- The recurrence equation is given as,

$$\boldsymbol{t(n) = t(n/2) + \theta(1)}$$

$$\boxed{\boldsymbol{T(n) = aT(n/b) + f(n)}}$$

- Comparing this to the general template for divide and conquer algorithm, $a = 1, b = 2$ and $f(n) = \theta(1)$.

$$\therefore \boldsymbol{t(n) \in \theta(\log n)}$$

- The complexity of binary search is $\boxed{\boldsymbol{\theta(\log n)}}$

Binary Search – Examples

1. Demonstrate binary search algorithm and find the element $\boldsymbol{x = 12}$ in the following array. [3 / 4]

$$2, 5, 8, 12, 16, 23, 38, 56, 72, 91$$

2. Explain binary search algorithm and find the element $\boldsymbol{x = 31}$ in the following array. [7]

$$10, 15, 18, 26, 27, 31, 38, 45, 59$$

3. Let $\boldsymbol{T}[\mathbf{1..}\boldsymbol{n}]$ be a sorted array of distinct integers. Give an algorithm that can find an index $\boldsymbol{i}$ such that $\mathbf{1} \leq \boldsymbol{i} \leq \boldsymbol{n}$ and $\boldsymbol{T}[\boldsymbol{i}] = \boldsymbol{i}$, provided such an index exists. Prove that your algorithm takes time in $O(logn)$ in the worst case.

Merge Sort

Introduction

- Merge Sort is an example of **divide and conquer algorithm.**
- It is based on the **idea of breaking down a list into several sub-lists** until each sub list consists of a **single element.**
- **Merging those sub lists** in a manner that results into a sorted list.
- **Procedure**
  - Divide the unsorted list into N sub lists, each containing 1 element
  - Take adjacent pairs of two singleton lists and merge them to form a list of 2 elements. N will now convert into N/2 lists of size 2
  - Repeat the process till a single sorted list of all the elements is obtained

Merge Sort - Example

Unsorted Array

724	521	2	98	529	31	189	451
1	2	3	4	5	6	7	8

Step 1: Split the selected array

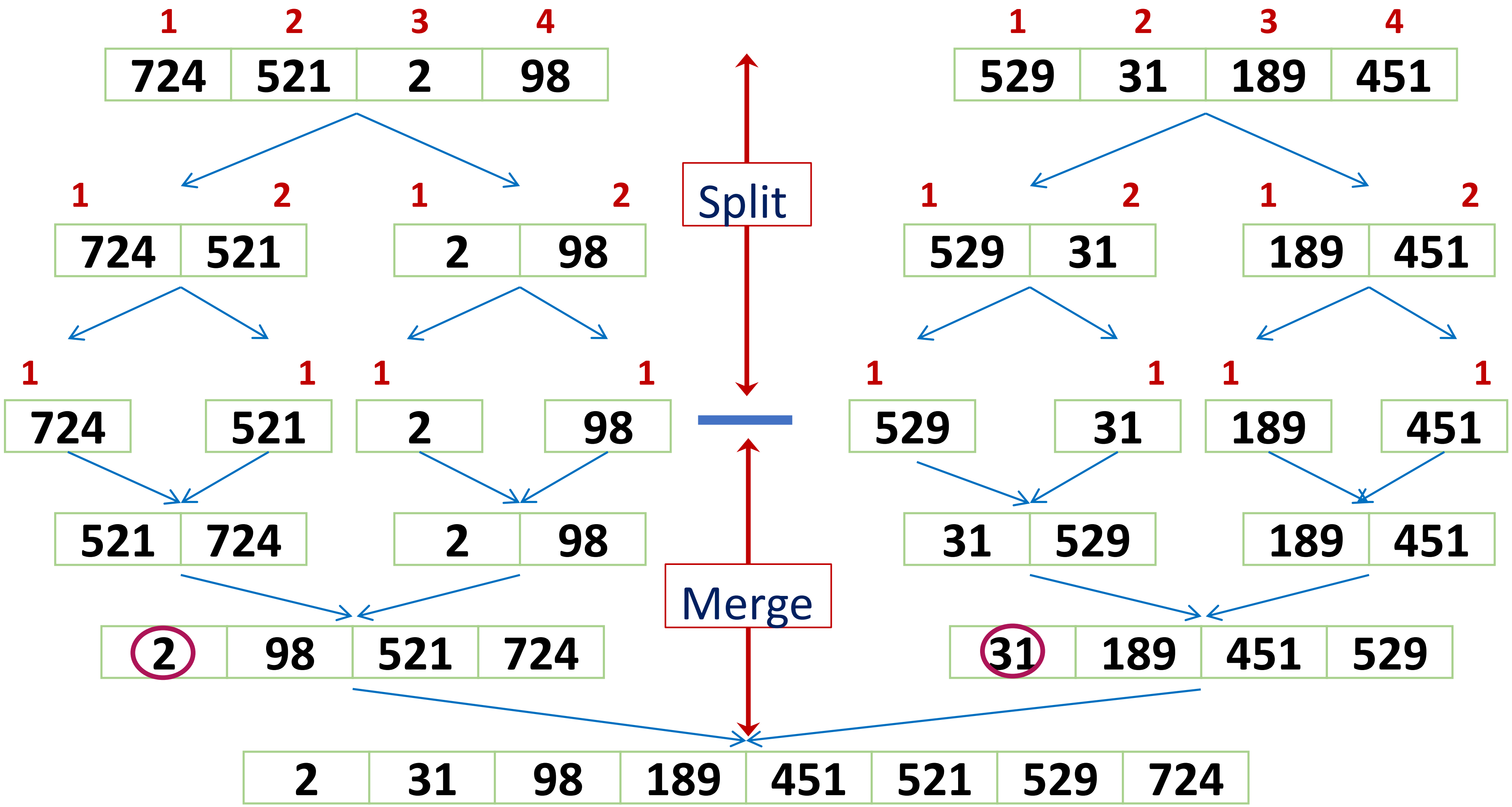
1	2	3	4	5	6	7	8
724	**521**	**2**	**98**	**529**	**31**	**189**	**451**

724	**521**	**2**	**98**
1	2	3	4

529	**31**	**189**	**451**
1	2	3	4

Merge Sort - Example

Select the left subarray and Split | **Select the right subarray and Split**



Merge Sort - Algorithm

```
Procedure: mergesort(T[1,…,n])
if n is sufficiently small
then insert(T)
else
array
U[1,…,1+n/2],V[1,…,1+n/2]
     U[1,…,n/2] ← T[1,…,n/2]
     V[1,…,n/2] ← T[n/2+1,…,n]

     mergesort(U[1,…,n/2])

     mergesort(V[1,…,n/2])
          merge(U, V, T)
```

```
Procedure:
merge(U[1,…,m+1],V[1,…,n+1],T[1,…,m+n])
i ← 1;
j ← 1;
U[m+1], V[n+1] ← ∞;
for k ← 1 to m + n do
     if U[i] < V[j]
          then T[k] ← U[i];
               i ← i + 1;
     else T[k] ← V[j];
          j ← j + 1;
```

Merge Sort - Analysis

- Let $\boldsymbol{T(n)}$ **be the time taken** by this algorithm to sort an array of n elements.
- Separating T into U & V takes **linear time**; $merge(U, V, T)$ also takes **linear time**.

$$T(n) = T(n/2) + T(n/2) + g(n) \quad \text{where } g(n) \in \theta(n).$$

$$T(n) = 2t(n/2) + \theta(n)$$

$$\boldsymbol{t(n) = lt(n/b) + g(n)}$$

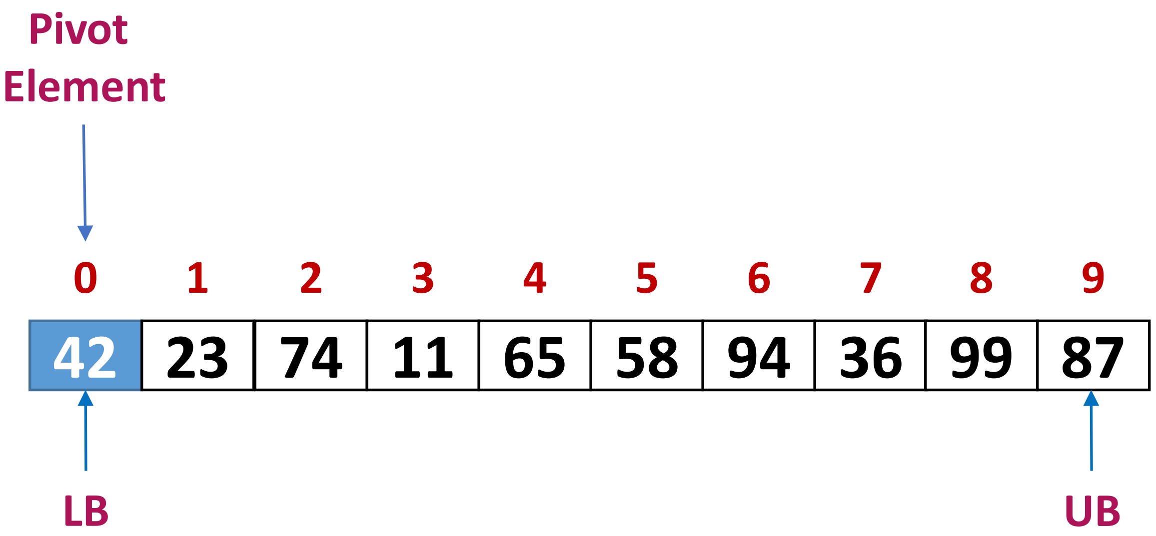
- Applying the general case, $l = 2, b = 2, k = 1$
- Since $l = b^k$ the **second case** applies so, $t(n) \in \theta(nlogn)$.
- Time complexity of merge sort is $\boxed{\boldsymbol{\theta(nlogn)}}$.

$$\boldsymbol{t(n) = \begin{cases} \theta(n^k) & if\ l < b^k \\ \theta(n^k logn) & if\ l = b^k \\ \theta(n^{log_b l}) & if\ l > b^k \end{cases}}$$

Quick Sort

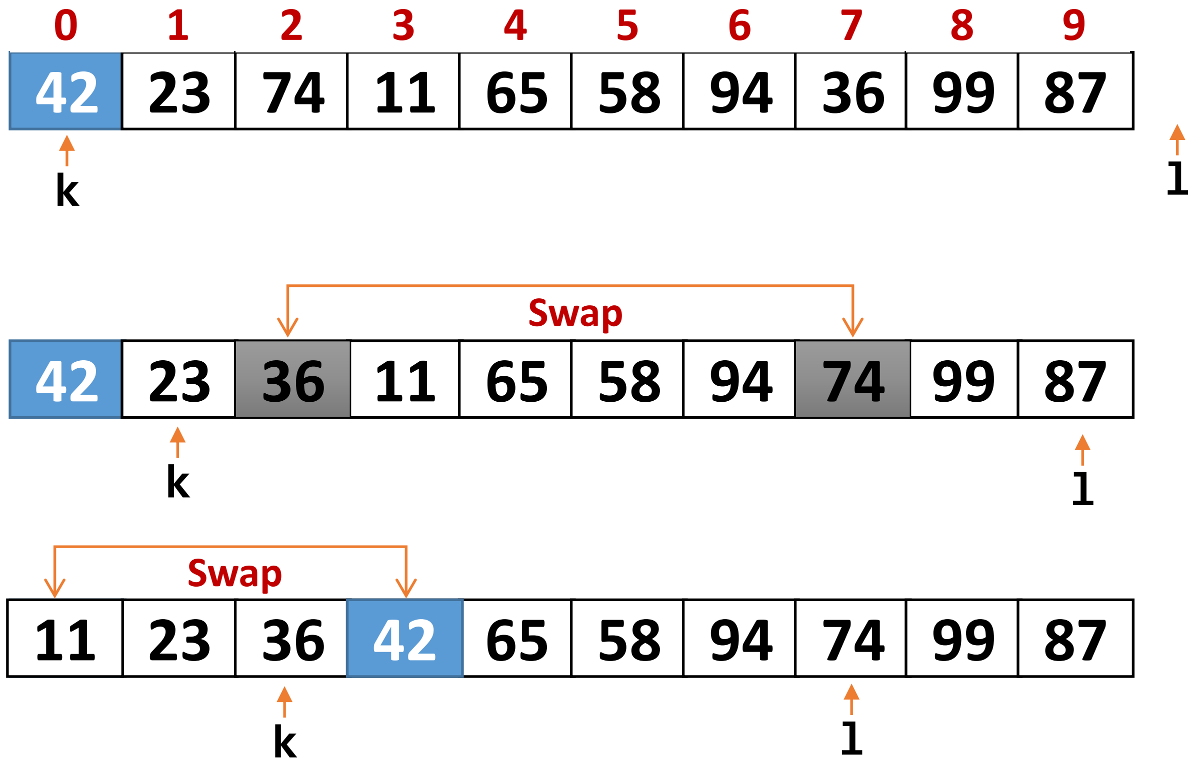
Introduction

- Quick sort chooses the first element as a pivot element, a **lower bound is the first index** and an **upper bound is the last index**.
- The array is then **partitioned** on either side of the pivot.
- Elements are moved so that, those **greater** than the pivot are shifted to its **right** whereas the others are shifted to its **left.**
- Each Partition is **internally sorted recursively**.



Quick Sort - Example

```
Procedure pivot(T[i,…,j]; var l)
p ← T[i]
k ← i; l ← j+1
Repeat
k ← k+1 until T[k] > p or k ≥ j
Repeat
l ← l-1 until T[l] ≤ p
While k < l do
    Swap T[k] and T[l]
    Repeat k ← k+1 until T[k] > p
    Repeat l ← l-1 until T[l] ≤ p
Swap T[i] and T[l]
```



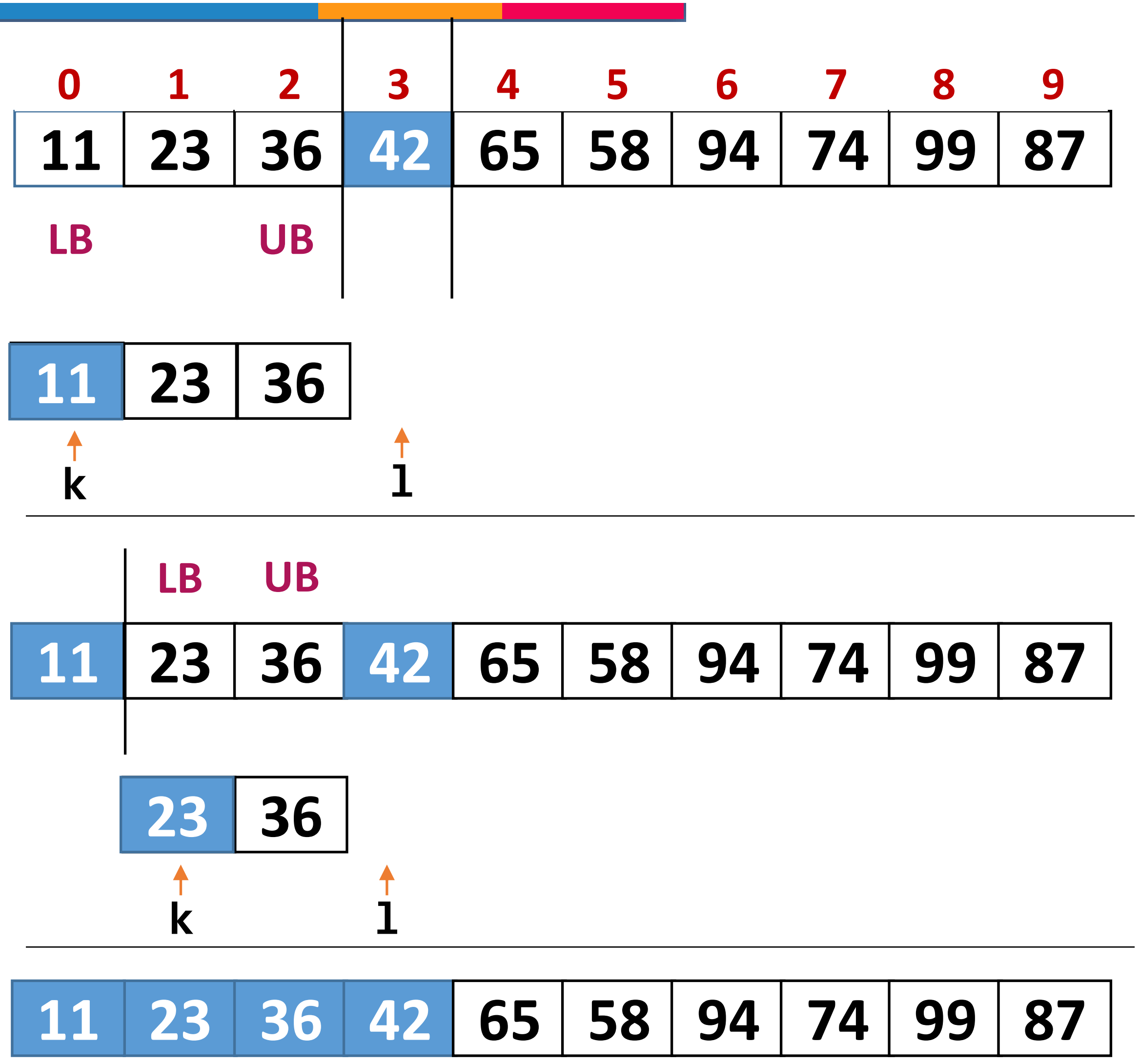
LB = 0, UB = 9

p = 42

k = 0, l = 10

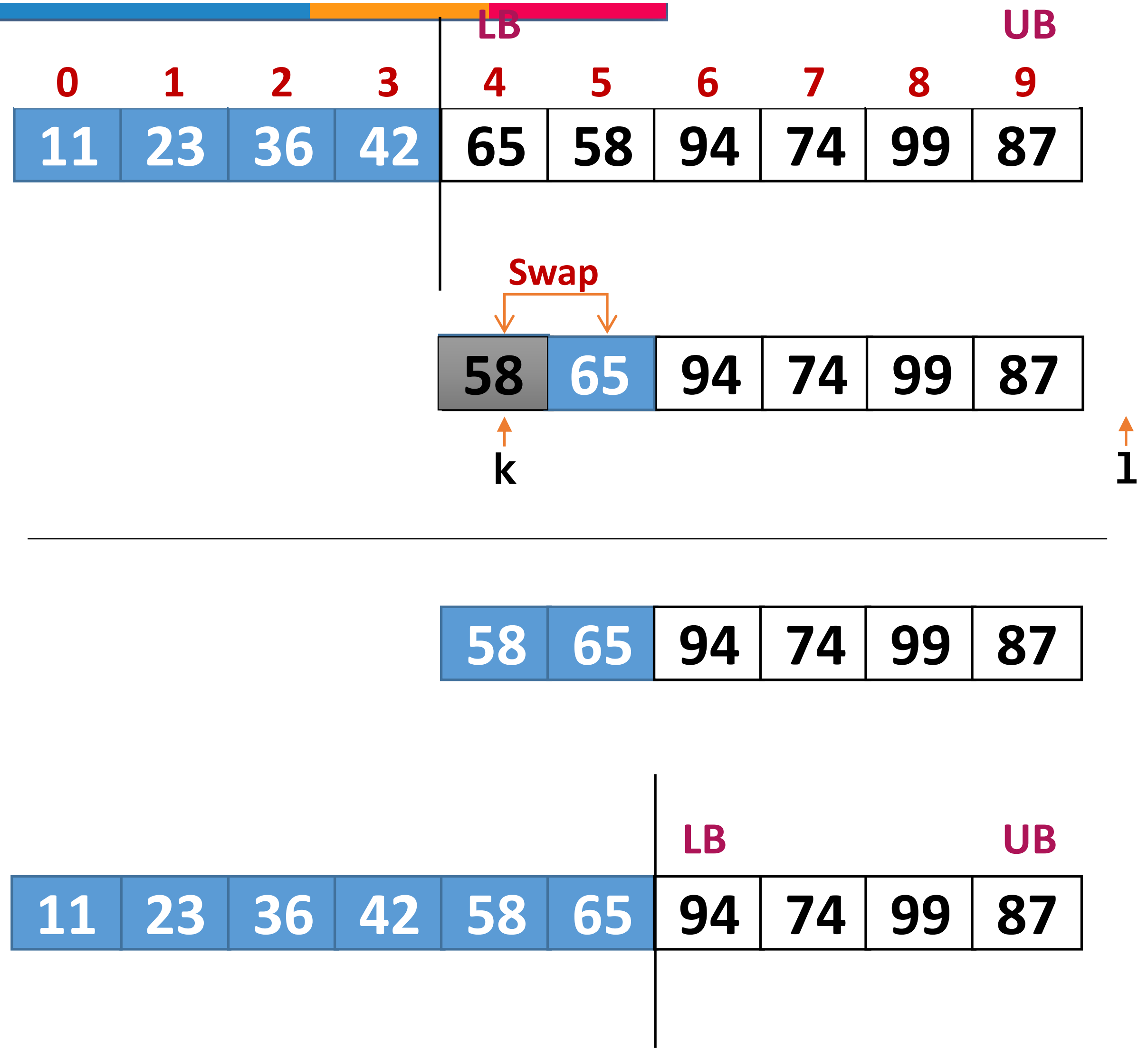
Quick Sort - Example

```
Procedure pivot(T[i,…,j]; var l)
p ← T[i]
k ← i; l ← j+1
Repeat
k ← k+1 until T[k] > p or k ≥ j
Repeat
l ← l-1 until T[l] ≤ p
While k < l do
   Swap T[k] and T[l]
   Repeat k ← k+1 until T[k] > p
   Repeat l ← l-1 until T[l] ≤ p
Swap T[i] and T[l]
```



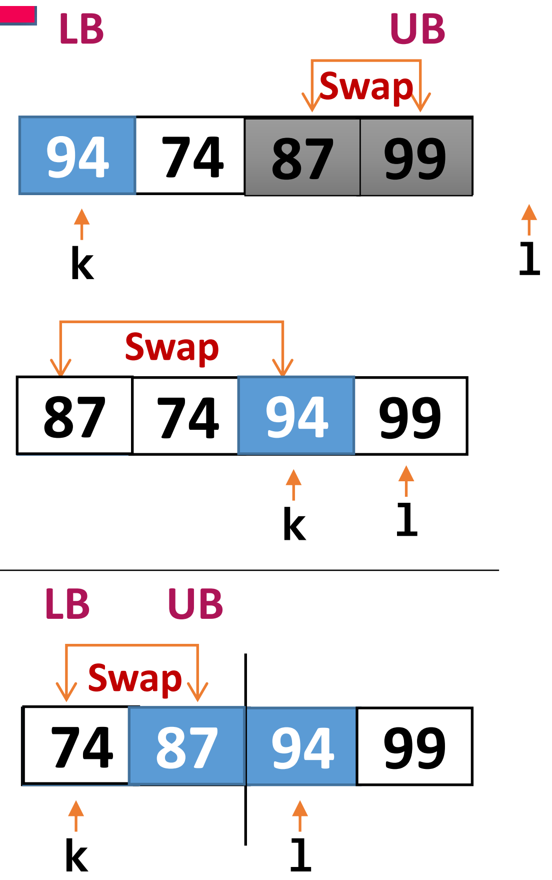
Quick Sort - Example

```
Procedure pivot(T[i,…,j]; var l)
p ← T[i]
k ← i; l ← j+1
Repeat
k ← k+1 until T[k] > p or k ≥ j
Repeat
l ← l-1 until T[l] ≤ p
While k < l do
    Swap T[k] and T[l]
    Repeat k ← k+1 until T[k] > p
    Repeat l ← l-1 until T[l] ≤ p
Swap T[i] and T[l]
```



Quick Sort - Example

```
Procedure pivot(T[i,…,j]; var l)
p ← T[i]
k ← i; l ← j+1
Repeat
k ← k+1 until T[k] > p or k ≥ j
Repeat
l ← l-1 until T[l] ≤ p
While k < l do
    Swap T[k] and T[l]
    Repeat k ← k+1 until T[k] > p
    Repeat l ← l-1 until T[l] ≤ p
Swap T[i] and T[l]
```



11	23	36	42	58	65	74	87	94	99

Quick Sort - Algorithm

```
Procedure: quicksort(T[i,…,j])
{Sorts subarray T[i,…,j] into
ascending order}
if j – i is sufficiently small
then insert (T[i,…,j])
else
      pivot(T[i,…,j],l)
      quicksort(T[i,…,l - 1])
      quicksort(T[l+1,…,j]
```

```
Procedure: pivot(T[i,…,j]; var l)
p ← T[i]
k ← i
l ← j + 1
repeat k ← k+1 until T[k] > p or k ≥ j
repeat l ← l-1 until T[l] ≤ p
while k < l do
      Swap T[k] and T[l]
       Repeat k ← k+1 until T[k] > p
       Repeat l ← l-1 until T[l] ≤ p
Swap T[i] and T[l]
```

Quick Sort - Algorithm

1. Worst Case
   - Running time depends on **which element is chosen as key or pivot** element.
   - The worst case behavior for quick sort occurs when the array is partitioned into one sub-array with $\boldsymbol{n-1}$ **elements and the other with $\mathbf{0}$ element**.
   - In this case, the recurrence will be,

$$T(n) = T(n-1) + T(0) + \theta(n)$$

$$T(n) = T(n-1) + \theta(n)$$

$$\boxed{\boldsymbol{T(n) = \theta(n^2)}}$$

2. Best Case
   - **Occurs when partition produces sub-problems each of size n/2.**
   - Recurrence equation:

$$T(n) = 2T(n/2) + \theta(n)$$

$$l = 2, b = 2, k = 1, so\ l = b^k$$

$$\boxed{\boldsymbol{T(n) = \theta(nlogn)}}$$

Quick Sort - Algorithm

3. Average Case
   - Average case running time is much closer to the best case.
   - If suppose the partitioning algorithm produces a **9:1 proportional** split the recurrence will be,

$$T(n) = T(9n/10) + T(n/10) + \theta(n)$$

$$\boxed{\boldsymbol{T(n) = \theta(nlogn)}}$$

Quick Sort - Examples

- Sort the following array in ascending order using quick sort algorithm.
  1. 5, 3, 8, 9, 1, 7, 0, 2, 6, 4
  2. 3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5, 8, 9
  3. 9, 7, 5, 11, 12, 2, 14, 3, 10, 6

Strassen's Algorithm for Matrix Multiplication

Matrix Multiplication

- Multiply following two matrices. Count how many scalar multiplications are required.

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

$$answer = \begin{bmatrix} 1 \underline{\times} 6 + 3 \underline{\times} 4 & 1 \underline{\times} 8 + 3 \underline{\times} 2 \\ 7 \underline{\times} 6 + 5 \underline{\times} 4 & 7 \underline{\times} 8 + 5 \underline{\times} 2 \end{bmatrix}$$

- To multiply 2×2 matrices, total 8 (2^3) scalar multiplications are required.

Matrix Multiplication

- In general, A and B are two 2×2 matrices to be multiplied.

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

$$C = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \cdot \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

$$\begin{aligned} C_{11} &= A_{11} \cdot B_{11} + A_{12} \cdot B_{21} \\ C_{12} &= A_{11} \cdot B_{12} + A_{12} \cdot B_{22} \\ C_{21} &= A_{21} \cdot B_{11} + A_{22} \cdot B_{21} \\ C_{22} &= A_{21} \cdot B_{12} + A_{22} \cdot B_{22} \end{aligned}$$

- Computing each entry in the product takes $\boldsymbol{n}$ **multiplications** and there are $\boldsymbol{n^2}$ **entries** for a total of $\boldsymbol{O(n^3)}$.

Matrix Multiplication

- In general, A and B are two $\mathrm{n} \times \mathrm{n}$ matrices to be multiplied.

Algorithm: Matrix_Multiplication(A, B, C)

```
for i = 1 to n do
    for j = 1 to n do
        C[i, j] = 0
         for k = 1 to n do
            C[i, j] = C[i, j] + A[i, k] × B[k, j]
```

- Computing each entry in the product takes $\boldsymbol{n}$ **multiplications** and there are $\boldsymbol{n^2}$ **entries** for a total of $\boldsymbol{O(n^3)}$.

Strassen's Algorithm for Matrix Multiplication

- Consider the problem of **multiplying** two $n \times n$ matrices.
- Strassen's devised a better method which has the **same basic method** as the multiplication of long integers.
- The main idea is **to save one multiplication** on a small problem and then use recursion.

Strassen's Algorithm for Matrix Multiplication

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

Step 1

$$S_1 = B_{12} - B_{22}$$
$$S_2 = A_{11} + A_{12}$$
$$S_3 = A_{21} + A_{22}$$
$$S_4 = B_{21} - B_{11}$$
$$S_5 = A_{11} + A_{22}$$
$$S_6 = B_{11} + B_{22}$$
$$S_7 = A_{12} - A_{22}$$
$$S_8 = B_{21} + B_{22}$$
$$S_9 = A_{11} - A_{21}$$
$$S_{10} = B_{11} + B_{12}$$

Step 2

$$P_1 = A_{11} \odot S_1$$
$$P_2 = S_2 \odot B_{22}$$
$$P_3 = S_3 \odot B_{11}$$
$$P_4 = A_{22} \odot S_4$$
$$P_5 = S_5 \odot S_6$$
$$P_6 = S_7 \odot S_8$$
$$P_7 = S_9 \odot S_{10}$$

All above operations involve only **one multiplication**.

Step 3

Final Answer:

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

Where,

$$C_{11} = P_5 + P_4 - P_2 + P_6$$
$$C_{12} = P_1 + P_2$$
$$C_{21} = P_3 + P_4$$
$$C_{22} = P_5 + P_1 - P_3 - P_7$$

No multiplication is required here.

Strassen's Algorithm - Analysis

- It is therefore possible to multiply two 2×2 matrices using only **seven scalar multiplications**.
- Let $t(n)$ be the time needed to multiply two $n \times n$ matrices by **recursive use of equations**.

$$\boldsymbol{t(n) = 7t(n/2) + g(n)}$$

$$\boldsymbol{t(n) = lt(n/b) + g(n)}$$

Where $g(n) \in O(n^2)$.

- The general equation applies with $l = 7, b = 2$ and $k = 2$.
- Since $l > b^k$, the **third case** applies and $t(n) \in O\left(n^{lg7}\right)$.
- Since $lg7 \approx 2.81$, it is possible to multiply two $n \times n$ matrices in a time $\boxed{\boldsymbol{O(n^{2.81})}}$.

$$\boldsymbol{t(n) = \begin{cases} \theta(n^k) & if\ l < b^k \\ \theta(n^k logn) & if\ l = b^k \\ \theta(n^{log_b l}) & if\ l > b^k \end{cases}}$$

Exponentiation

Exponentiation – Sequential

- Let a and n be two integers. We wish to compute the **exponentiation** $x = a^n$.
- Algorithm using **Sequential Approach:**

```
function exposeq(a, n)
   r ← a
   for i ← 1 to n - 1 do
      r ← a * r
   return r
```

- This algorithm takes a time in $\boldsymbol{\theta(n)}$ since the instruction $\boldsymbol{r = a * r}$ is executed exactly $\boldsymbol{n - 1}$ **times**, provided the multiplications are counted as elementary operations.

Exponentiation – Sequential

- But **to handle larger operands**, we must consider the time required for each multiplication.
- Let $\boldsymbol{m}$ is the size of operand $\boldsymbol{a}$.
- Therefore, the multiplication performed the $\boldsymbol{i^{th}}$ **time** round the loop concerns **an integer of size** $\boldsymbol{m}$ and **an integer whose size is between** $\boldsymbol{im - i + 1}$ **and** $\boldsymbol{im}$, which takes a time between

$$M(m, im - i + 1) \text{ and } M(m, im)$$

$a = 5\ \ so\ m = 1\ and\ n = 25$ and suppose $i = 10$

The body of loop executes 10^{th} time as,

$$\boldsymbol{r = a * r}$$

here 9 times multiplication is already done so $\boldsymbol{r} = \boldsymbol{5^9} = 1953125$

The size of r in the 10th iteration will be between $im - i + 1\ to\ im$, i.e., between $\boldsymbol{1\ to\ 10}$

10-10+1

10

Exponentiation – Sequential

- The total time $T(m,n)$ spent multiplying when computing an with ***exposeq*** is therefore,

$$\sum_{i=1}^{n-1} M(m, im-1+1) \le T(m,n) \le \sum_{i=1}^{n-1} M(m, im)$$

$$T(m,n) \le \sum_{i=1}^{n-1} M(m, im) \le \sum_{i=1}^{n-1} cm\ im$$

$$cm^2 \sum_{i=1}^{n-1} i \le cm^2n^2 = \boldsymbol{\theta\left(m^2n^2\right)}$$

- If we use the **divide-and-conquer** multiplication algorithm,

$$\boxed{\boldsymbol{T(m,n) \in \theta(m^{lg3}n^2)}}$$

Exponentiation – D & C

- Suppose, we want to compute $\boldsymbol{a^{10}}$
- We can write as,

$$a^{10} = (a^5)^2 = (a.a^4)^2 = (a.(a^2)^2)^2$$

- In general,

$$a^n = \begin{cases} a & if\ n = 1 \\ \left(a^{n/2}\right)^2 & if\ n\ is\ even \\ a \times a^{n-1} & otherwise \end{cases}$$

- Algorithm using **Divide & Conquer Approach:**

```
function expoDC(a, n)
   if n = 1 then return a
   if n is even then return [expoDC(a, n/2)]²
   return a * expoDC(a, n - 1)
```

Exponentiation – D & C

Number of operations performed by the algorithm is given by,

$$N(n) = \begin{cases} 0 & if\ n = 1 \\ N(n/2) + 1 & if\ n\ is\ even \\ N(n-1) + 1 & otherwise \end{cases}$$

Time taken by the algorithm is given by,

$$T(m,n) = \begin{cases} 0 & if\ n = 1 \\ T(m, n/2) + M(m\,n/2, m\,n/2) & if\ n\ is\ even \\ T(m, n-1) + M(m, (n-1)m) & otherwise \end{cases}$$

Solving it gives, $\boxed{\boldsymbol{T(m,n)\ \in\ \theta\ (m^{lg3} n^{lg3})}}$

```
function expoDC(a, n)
   if n = 1 then return a
   if n is even then return [expoDC(a, n/2)]²
   return a * expoDC(a, n - 1)
```

Exponentiation – Summary

	Multiplication	
	Classic	D&C
exposeq	$\theta(m^2n^2)$	$\theta(m^{lg3}n^2)$
expoDC	$\theta(m^2n^2)$	$\theta(m^{lg3}n^{lg3})$

Thank You!