Data Structures and Algorithms

(CS210A)

Lecture 14:

- Algorithm paradigms
- Algorithm paradigm of Divide and Conquer
- Proof of correctness of the algorithm for 2-Majority element

Algorithm Paradigms

Algorithm Paradigm

Motivation:

- Many problems whose algorithms are based on a <u>common approach</u>.
- A need of a <u>systematic study</u> of such widely used approaches.

Algorithm Paradigms:

- **Divide and Conquer**
- **Greedy Strategy**
- **Dynamic Programming**
- **Local Search**

Divide and Conquer paradigm for Algorithm Design

Divide and Conquer paradigm
An Overview

1. **Divide** the problem instance into two or more instances of the same problem
2. Solve each smaller instances **recursively** (base case suitably defined).
3. **Combine** the solutions of the smaller instances to get the solution of the original instance.

This is usually the main **nontrivial** step in the design of an algorithm using divide and conquer strategy

Example 1

Sorting

A problem in Practice sheet 1

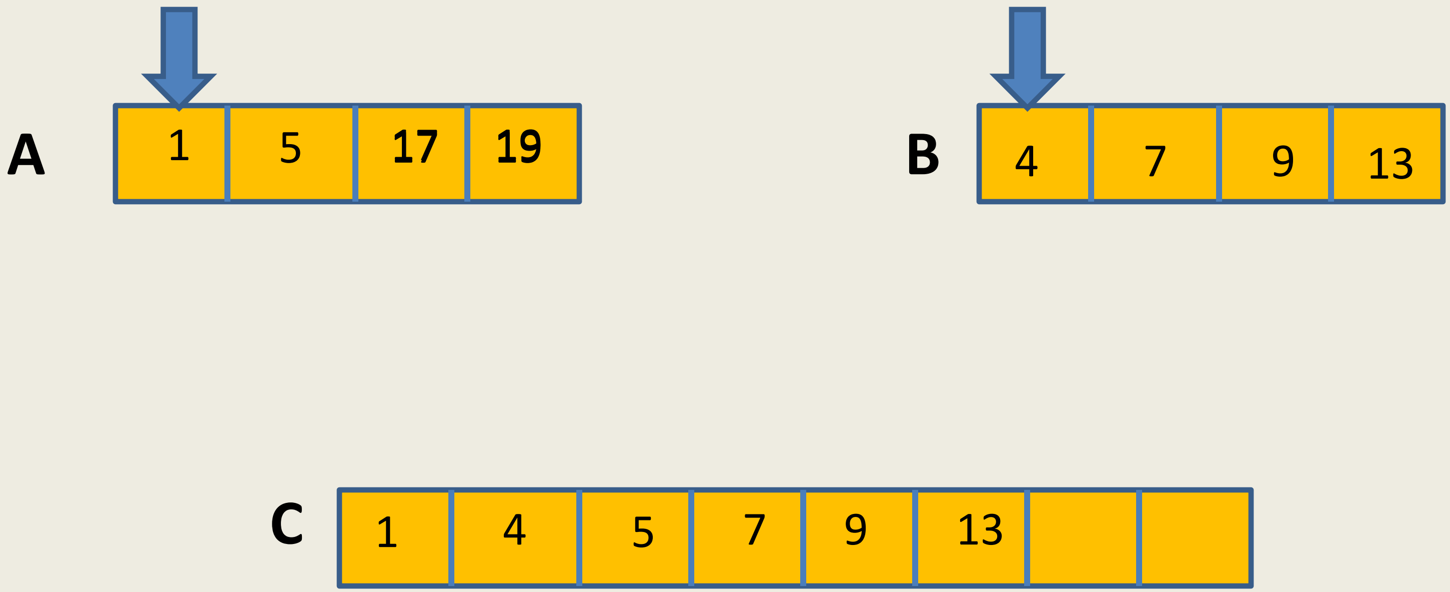
Merging two sorted arrays:

Given two sorted arrays **A** and **B** storing n elements each, Design an **O**(n) time algorithm to output a sorted array **C** containing all elements of **A** and **B**.

Example: If **A**={1,5,17,19} **B=**{4,7,9,13}, then output is

C={1,4,5,7,9,13,17,19}.

Merging two sorted arrays A and B



Pesudo-code for Merging two sorted arrays

```
Merge(A[0..n-1],B[0..m-1], C)    // Merging two sorted arrays A and B into array C.
{  i← 0;  j← 0;
   k← 0;
  While(i<n  and j<m)
  {      If(A[i]< B[j]) {    C[k] ← A[i]; k++;  i++      }
         Else            {    C[k] ← B[j]; k++;  j++    }
   }
   While(i<n) {  C[k] ← A[i]; k++;  i++   }
   While(j<m) {  C[k] ← B[j]; k++;  j++   }
   return C;
}
```

Correctness : homework exercise

Time Complexity = $O(n+m)$

Divide and Conquer based sorting algorithm

MSort(**A**,i,j) // Sorting the subarray **A**[$i..j$].

{ **If** (i< j)

{ **mid**←(i+j)/2;

MSort(**A**,i,**mid**);

MSort(**A**,**mid**+1,j);

Divide step

Create temporarily **C**[0..$j-i$]

Merge(**A**[i..**mid**], **A**[**mid**+1..j], **C**);

Copy **C**[0..$j-i$] to **A**[$i..j$]

Combine/conquer step

}

This is Merge Sort algorithm

Divide and Conquer based sorting algorithm

MSort(**A**,i,j) // Sorting the subarray **A**[i..j].

{ **If** (i< j)

{ **mid**←(i+j)/2;

MSort(**A**,i,**mid**); ← **T**(n/**2**)

MSort(**A**,**mid**+1,j); ← **T**(n/**2**)

Create temporarily **C**[0..$j - i$]

Merge(**A**[i..**mid**], **A**[**mid**+1..j], **C**);

Copy **C**[0..$j - i$] to **A**[i..j]

c n

}

Time complexity:

If n = 1,

T(n) = c for some constant c

If n > 1,

T(n) = c n + **2 T**(n/**2**)

= c n + c n + 2^2 **T**($n/2^2$)

= c n + c n + c n + 2^3 **T**(n /2^3)

= c n + …(log n terms)…+ c n

= **O**(n **log** n)

Proof of correctness of Merge-Sort

```
MSort(A,i,j)  // Sorting the subarray A[i..j].
{  If (  i< j  )
   {   mid←(i+j)/2;
       MSort(A,i,mid);
       MSort(A,mid+1,j);
       Create temporarily  C[0..j − i]
       Merge(A[i..mid], A[mid+1..j], C);
       Copy C[0..j − i]  to A[i..j]
}
```

Question: What is to be proved ?

Answer: **MSort(A,i,j)** sorts the subarray **A**[i..j]

Question: How to prove ?

Answer:

- By **induction** on the length ($j - i + 1$) of the subarray.
- Use correctness of the algorithm **Merge.**

Example 2

Faster algorithm for multiplying two integers

Addition is faster than multiplication

Given: any two n-bit numbers **X** and **Y**

Question: how many **bit-operations** are required to compute **X+Y** ?

Answer: $O(n)$

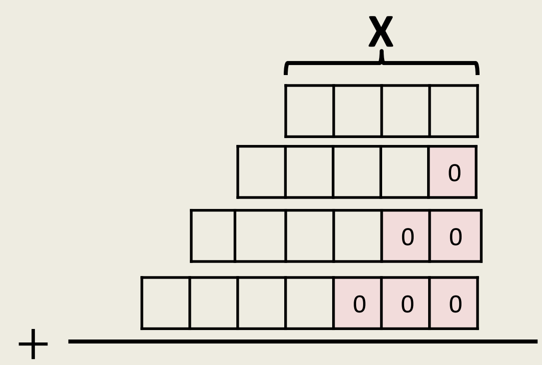
Question: how many **bit-operations** are required to compute **X*** $\mathbf{2^n}$?

Answer: $O(n)$ [left shift the number **X** by n places, (do it carefully)]

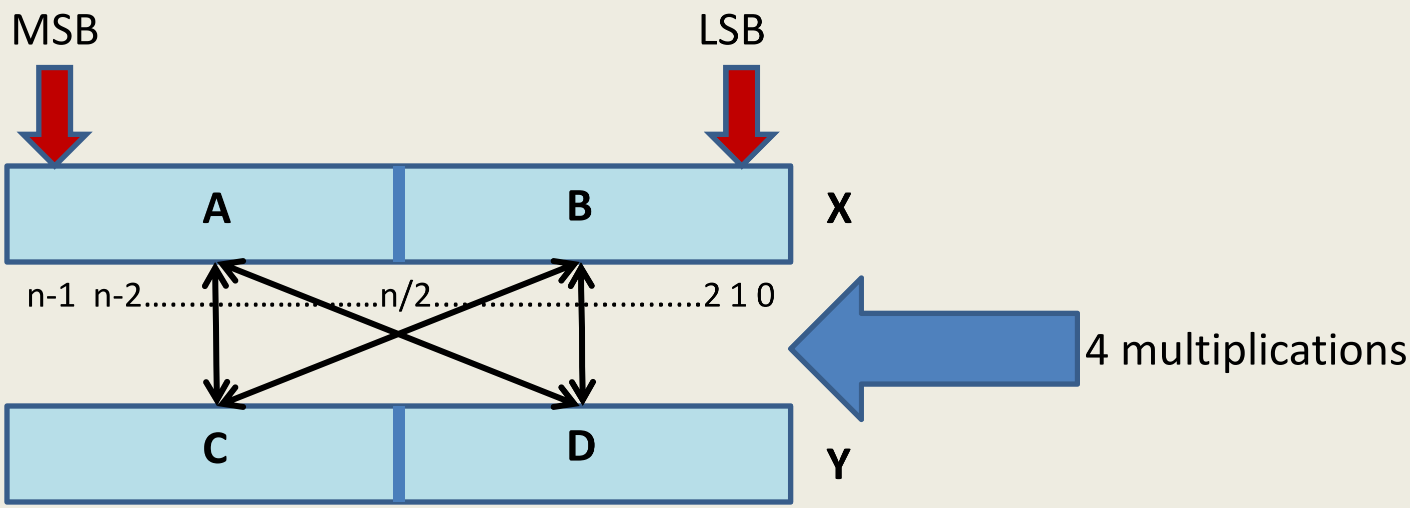
Question: how many **bit-operations** are required to compute **X*Y** ?

Answer: $O(n^2)$

Can we compute
X*Y faster ??



Pursuing Divide and Conquer approach



Question: how to express **X*Y** in terms of **multiplication/addition** of **{A,B,C,D} ?**

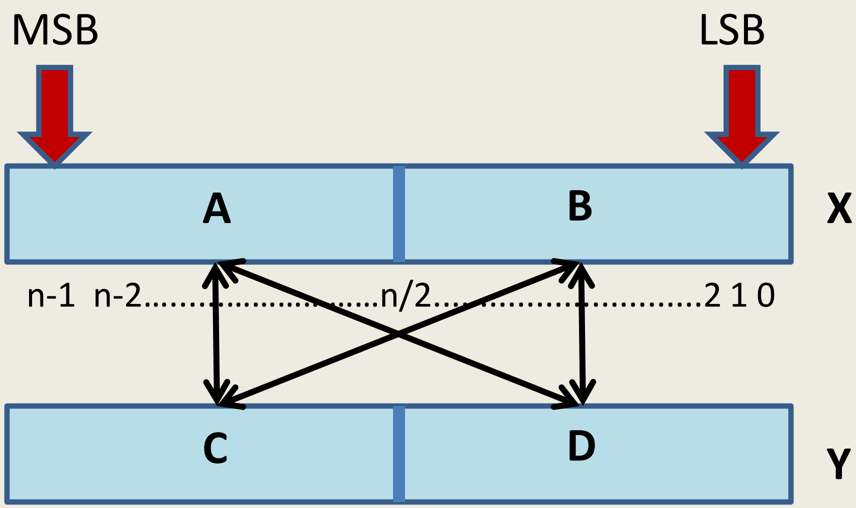
Hint: First Express **X** and **Y** in terms of **{A,B,C,D}.**

X = **A*** $2^{n/2}$ + **B** and **Y** = **C*** $2^{n/2}$ + **D** .

Hence …

$$\mathbf{X*Y = (A*C)*}\boxed{2^{n}} + \mathbf{(A*D + B*C)*}\boxed{2^{n/2}} + \mathbf{B*D}$$

Pursuing Divide and Conquer approach



$$X*Y = (A*C)*\boxed{2^n} + (A*D + B*C)*\boxed{2^{n/2}} + B*D$$

Let $T(n)$: time complexity of multiplying **X** and **Y** using the above equation.

$T(n) = c\,n + 4\,T(n/2)$ for some constant c

$= c\,n + 2c\,n + 4^2\,T(n/2^2)$

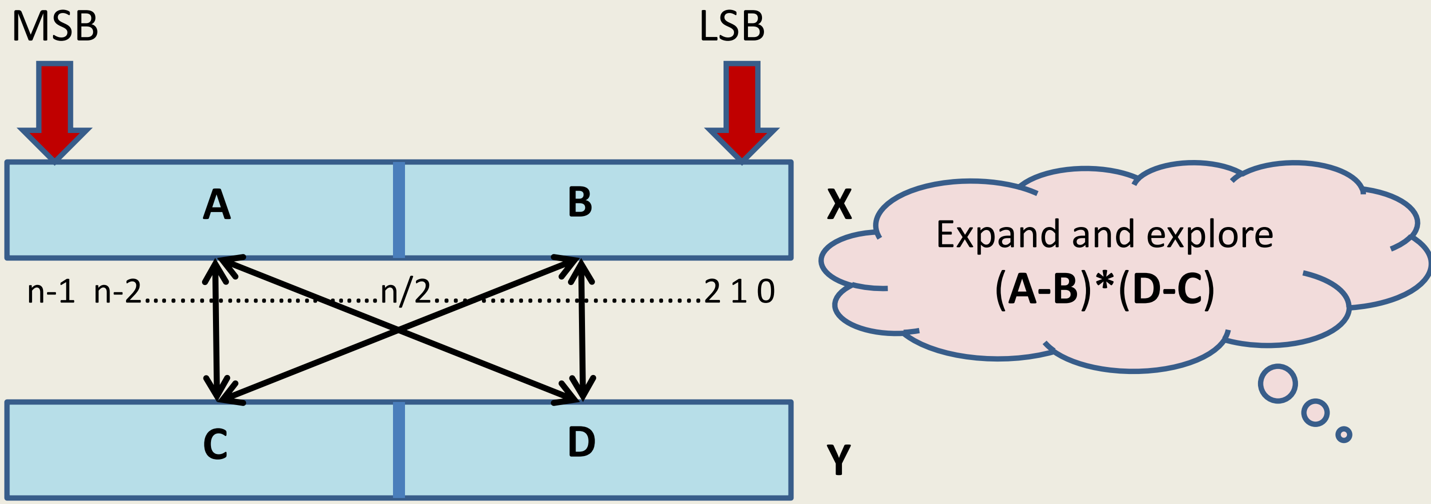
$= c\,n + 2c\,n + 4c\,n + 4^3\,T(n/2^3)$

$= c\,n + 2c\,n + 4c\,n + 8c\,n + \ldots + 4^{\log_2 n} T(1)$

$= c\,n + 2c\,n + 4c\,n + 8c\,n + \ldots + c\,n^2$

$O(n^2)$ time algo

Pursuing Divide and Conquer approach



X*Y = (A*C)* $\boxed{2^{n}}$ **+ $\boxed{\text{(A*D + B*C)}}$*** $\boxed{2^{n/2}}$ **+ B*D**

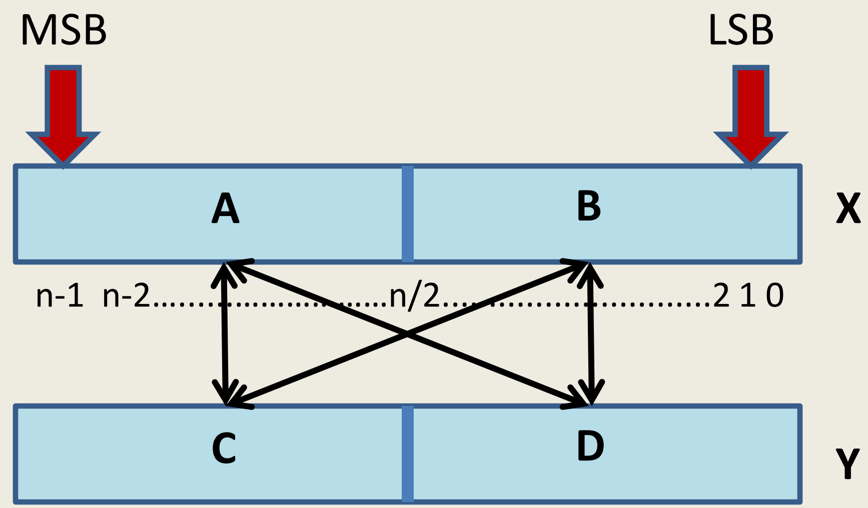
Observation: A*D + B*C = (A-B)*(D-C) + A*C + B*D

Question: How many multiplications do we need **now** to compute **X*Y** ?

Answer: **3** multiplications :

- **A*C**
- **B*D**
- **(A-B)*(D-C) .**

Pursuing Divide and Conquer approach



$$X*Y = (A*C)*\boxed{2^{n}} + \underline{((A-B)*(D-C) + A*C + B*D)}\boxed{2^{n/2}} + B*D$$

Let **T(n)** : time complexity of the **new algo** for multiplying two n-bit numbers

$T(n) = c\,n + 3\,T(n/2)$ for some constant c

$= c\,n + 3\,c\frac{n}{2} + 3^2\,T(n/2^2)$

$= c\,n + 3c\frac{n}{2} + 9c\frac{n}{4} + + \ldots + 3^{\log_2 n}\,T(1)$

$= O(n^{\log_2 3}) = O(n^{1.58})$

Conclusion

Theorem: There is a **divide and conquer** based algorithm for multiplying any two n-bit numbers in **O**($n^{1.58}$) time **(bit operations).**

Note:

The fastest algorithm for this problem runs in almost **O**(n **log** n) time.

One such algorithm was designed in **2008** at CSE, IIT Kanpur.

By (Dey, Kurur, Saha, and Saptharishi).

Example 3

Counting the number of *"inversions"* in an array

Counting Inversions in an array
Problem description

Definition (Inversion): Given an array **A** of size n,

a pair (i,j), $0 \le i < j < n$ is called an inversion if $\mathbf{A}[i] > \mathbf{A}[j]$.

Example:

	0	1	2	3	4	5	6	7
A	3	15	8	19	9	67	11	27

Inversions are :

(1,2), (1,4), (1,6),

(3,4),(3,6),

(5,6), (5,7)

AIM: An efficient algorithm to count the number of inversions in an array **A**.

Counting Inversions in an array
Problem familiarization

```
Trivial-algo(A[0..n-1])
{ count ← 0;
  For(j=1  to n-1) do
  {      For( i=0 to j-1 )
         {      If (A[i]>A[j])  count ← count + 1;
         }
  }
}
```

Ponder over the **divide and conquer algorithm** for this problem. We shall discuss it in the next class.

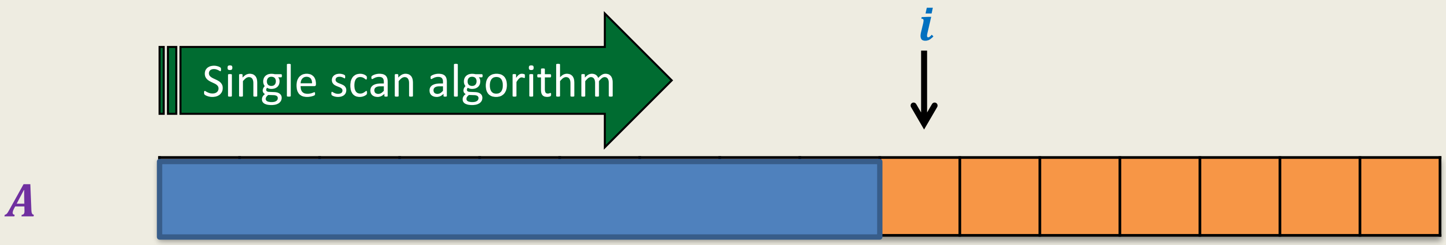
Time complexity: O(n^2)

Question: What can be the max. no. of inversions in an array **A** ?

Answer: $\binom{n}{2}$, which is O(n^2).

Question: Is the algorithm given above optimal ?

Answer: No, our aim is **not** to report all inversions but to report the count.



Question: What assertion holds at the end of ith iteration ?

Answer:

P(i) : α is a majority element of {**x**,…count times…,**x**, **A**[i],…,**A**[n-1]}

Question: What is **P(n) ?**

Answer: α is a majority element of {**x**,…count times…,**x**}

➔ **x = α**

As a homework exercise, prove assertion **P(i)** by induction on i.