Design and Analysis of Algorithms 18CS42

Module 2: Divide and Conquer

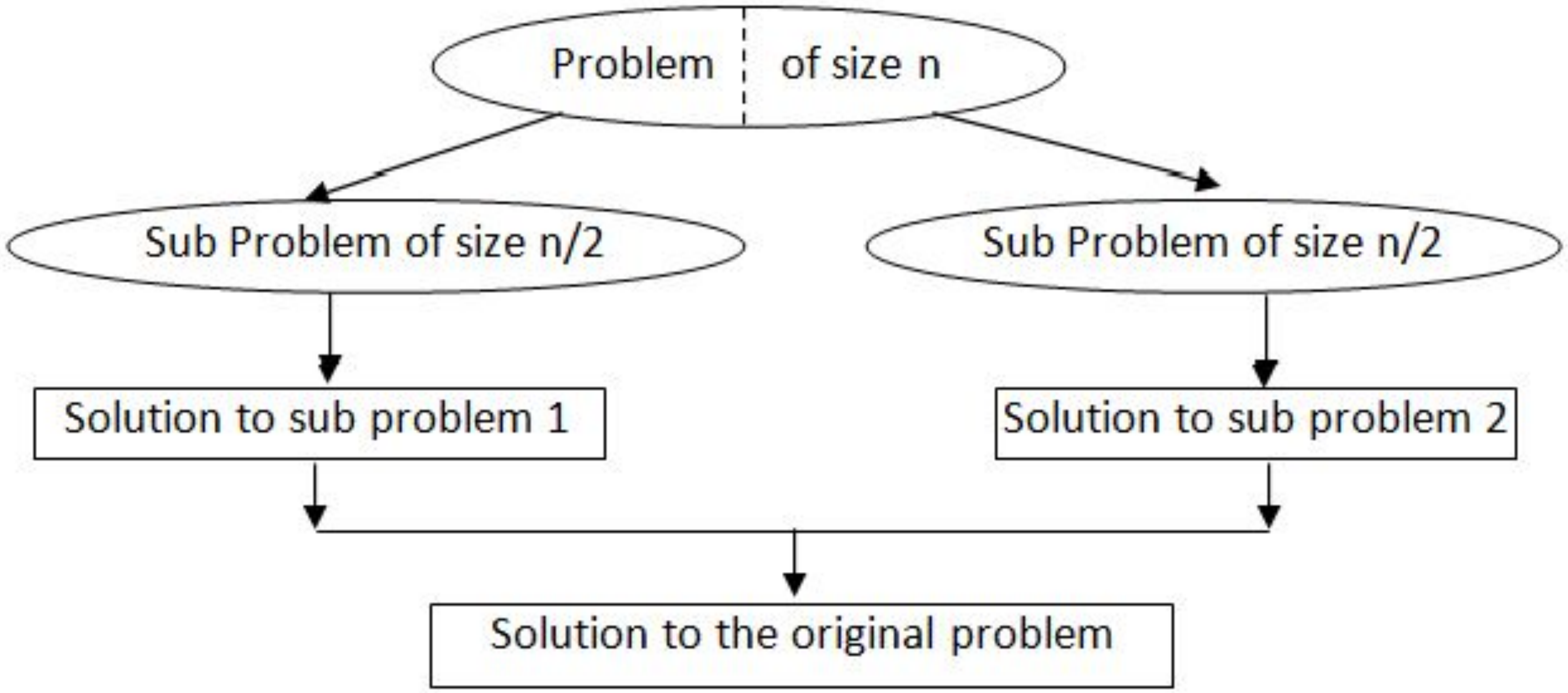
Module 2 – Outline
Divide and Conquer

1. General method
2. Recurrence equation
3. Algorithm: Binary search
4. Algorithm: Finding the maximum and minimum
5. Algorithm: Merge sort
6. Algorithm: Quick sort
7. Algorithm: Strassen’s matrix multiplication
8. Advantages and Disadvantages
9. Decrease and Conquer Approach
10. Algorithm: Topological Sort

Module 2 – Outline
Divide and Conquer

1. **General method**
2. Recurrence equation
3. Algorithm: Binary search
4. Algorithm: Finding the maximum and minimum
5. Algorithm: Merge sort
6. Algorithm: Quick sort
7. Algorithm: Strassen's matrix multiplication
8. Advantages and Disadvantages
9. Decrease and Conquer Approach
10. Algorithm: Topological Sort

Divide and Conquer


Problem
of size n
Sub Problem of size n/2
Sub Problem of size n/2
Solution to sub problem 1
Solution to sub problem 2
Solution to the original problem

Control Abstraction for Divide &Conquer

```
Algorithm DAndC(P)
{
    if Small(P) then return S(P);
    else
    {
        divide P into smaller instances $P_1, P_2, \ldots, P_k$, $k \geq 1$;
        Apply DAndC to each of these subproblems;
        return Combine(DAndC($P_1$),DAndC($P_2$),...,DAndC($P_k$));
    }
}
```

Module 2 – Outline
Divide and Conquer

1. General method
2. **Recurrence equation**
3. Algorithm: Binary search
4. Algorithm: Finding the maximum and minimum
5. Algorithm: Merge sort
6. Algorithm: Quick sort
7. Algorithm: Strassen’s matrix multiplication
8. Advantages and Disadvantages
9. Decrease and Conquer Approach
10. Algorithm: Topological Sort

Recurrence equation for Divide and Conquer

If the size of problem ‘p’ is n and the sizes of the ‘k’ sub problems are n_1, n_2 ….n_k, respectively, then

$$T(n) = \begin{cases} g(n) & n \text{ small} \\ T(n_1) + T(n_2) + \cdots + T(n_k) \quad + \quad f(n) & \text{otherwise} \end{cases}$$

Where,

- T(n) is the time for divide and conquer method on any input of size n and
- g(n) is the time to compute answer directly for small inputs.
- The function f(n) is the time for dividing the problem ‘p’ and combining the solutions to sub problems.

Recurrence equation for Divide and Conquer

- Generally, an instance of size **n** can be divided into **b** instances of size **n/b**,
- Assuming n = b^k ,

$$T(n) = \begin{cases} T(1) & n = 1 \\ aT(n/b) + f(n) & n > 1 \end{cases}$$

where f(n) is a function that accounts for the time spent on dividing the problem into smaller ones and on combining their solutions.

Example 3.1 Consider the case in which $a = 2$ and $b = 2$. Let $T(1) = 2$ and $f(n) = n$. We have

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

Example 3.2 Look at the following recurrence when n is a power of 2:

$$T(n) = \begin{cases} T(1) & n - 1 \\ T(n/2) + c & n > 1 \end{cases}$$

Example 3.3 Next consider the case in which $a = 2$, $b = 2$, and $f(n) = cn$.

Example 3.4 As another example, consider the recurrence $T(n) = 7T(n/2) + 18n^2$, $n \geq 2$ and a power of 2. We obtain $a = 7$, $b = 2$, and $f(n) = 18n^2$. So, $\log_b a = \log_2 7 \approx 2.81$ and $h(n) = 18n^2/n^{\log_2 7} = 18n^{2-\log_2 7} = O(n^r)$, where $r = 2 - \log_2 7 < 0$. So, $u(n) = O(1)$. The expression for $T(n)$ is

$$\begin{aligned} T(n) &= n^{\log_2 7}[T(1) + O(1)] \\ &= \Theta(n^{\log_2 7}) \end{aligned}$$

as $T(1)$ is assumed to be a constant. □

Will be solved under topic **Stressens Matrix Multiplication**

Extra!

Solving recurrence relation using **Master theorem**

It states that, in recurrence equation $T(n) = aT(n/b) + f(n)$, If $f(n) \in \Theta(n^d)$ where $d \geq 0$ then

$$T(n) \in \begin{cases} \Theta(n^d) & \text{if } a < b^d, \\ \Theta(n^d \log_b n) & \text{if } a = b^d, \\ \Theta(n^{\log_b a}) & \text{if } a > b^d. \end{cases}$$

Analogous results hold for the *O* and Ω notations, too. Example: $A(n) = 2A(n/2) + 1.$

Here a = 2, b = 2, and d = 0; hence, since a >b^d,

$$A(n) \in \Theta(n^{\log_b a}) = \Theta(n^{\log_2 2}) = \Theta(n).$$

Module 2 – Outline
Divide and Conquer

1. General method
2. Recurrence equation
3. **Algorithm: Binary search**
4. Algorithm: Finding the maximum and minimum
5. Algorithm: Merge sort
6. Algorithm: Quick sort
7. Algorithm: Strassen’s matrix multiplication
8. Advantages and Disadvantages
9. Decrease and Conquer Approach
10. Algorithm: Topological Sort

Binary Search

Problem definition:

- Let a_i, $1 \le i \le n$ be a list of elements that are sorted in non-decreasing order.
- The problem is to find whether a given element x is present in the list or not.
  - If x is present we have to determine a value j (element's position) such that a_j=x.
  - If x is not in the list, then j is set to zero.

Binary Search

Solution:

Let **P = (n, a_i…a_l , x)** denote an arbitrary instance of search problem

- where ***n*** is the number of elements in the list,

- **a_i…a_l** is the list of elements and

- x is the key element to be searched

Binary Search

Pseudocode

```
Step 1: Pick an index $q$ in the middle range [i, $l$] i.e. $q = \lfloor (n+1)/2 \rfloor$
    and compare x with $a_q$.
Step 2: if $x = a_q$ i.e key element is equal to mid element, the
    problem is immediately solved.
Step 3: if $x < a_q$ in this case x has to be searched for only in
    the sub-list $a_i, a_{i+1}, \ldots, a_{q-1}$. Therefore problem reduces to
    **($q-i, a_i \ldots a_{q-1}, x$).**
Step 4: if $x > a_q$ , x has to be searched for only in the sub-list
    $a_{q+1}, \ldots, a_l$. Therefore problem reduces to **($l-i, a_{q+1} \ldots a_l, x$).**
```

Recursive Binary search algorithm

```
int BinSrch(Type a[], int i, int l, Type x)
// Given an array a[i:l] of elements in nondecreasing
// order, 1<=i<=l, determine whether x is present, and
// if so, return j such that x == a[j]; else return 0.
{
  if (l==i) { // If Small(P)
     if (x==a[i]) return i;
     else return 0;
  }
  else { // Reduce P into a smaller subproblem.
     int mid = (i+l)/2;
     if (x == a[mid]) return mid;
```

Iterative binary search

```
int BinSearch(Type a[], int n, Type x)
// Given an array a[1:n] of elements in nondecreasing
// order, n>=0, determine whether x is present, and
// if so, return j such that x == a[j]; else return 0.
{
    int low = 1, high = n;
    while (low <= high){
        int mid = (low + high)/2;
        if (x < a[mid]) high = mid - 1;
        else if (x > a[mid]) low = mid + 1;
        else return(mid);
    }
    return(0);
}
```

Example 3.6 Let us select the 14 entries

$$-15, -6, 0, 7, 9, 23, 54, 82, 101, 112, 125, 131, 142, 151$$

$x = 151$	*low*	*high*	*mid*
	1	14	7
	8	14	11
	12	14	13
	14	14	14
			found

```
int BinSearch(Type a[], int n, Type x)
{
   int low = 1, high = n;
   while (low <= high){
      int mid = (low + high)/2;
      if (x < a[mid]) high = mid - 1;
      else if (x > a[mid]) low = mid + 1;
      else return(mid);
   }
   return(0);
}
```

Example 3.6 Let us select the 14 entries

$$-15, -6, 0, 7, 9, 23, 54, 82, 101, 112, 125, 131, 142, 151$$

$x = -14$	*low*	*high*	*mid*
	1	14	7
	1	6	3
	1	2	1
	2	2	2
	2	1	not found

```
int BinSearch(Type a[], int n, Type x)
{
   int low = 1, high = n;
   while (low <= high){
      int mid = (low + high)/2;
      if (x < a[mid]) high = mid - 1;
      else if (x > a[mid]) low = mid + 1;
      else return(mid);
   }
   return(0);
}
```

Example 3.6 Let us select the 14 entries

$$-15, -6, 0, 7, 9, 23, 54, 82, 101, 112, 125, 131, 142, 151$$

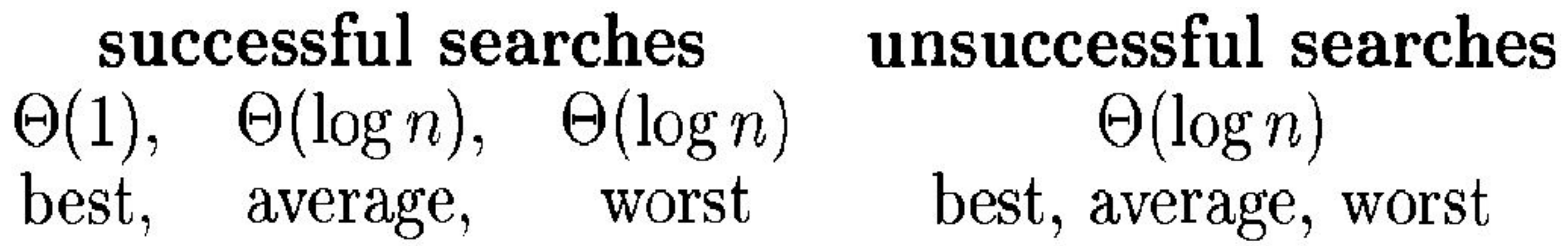
$x = 9$	*low*	*high*	*mid*
	1	14	7
	4	6	5
			found

```
int BinSearch(Type a[], int n, Type x)
{
   int low = 1, high = n;
   while (low <= high){
      int mid = (low + high)/2;
      if (x < a[mid]) high = mid - 1;
      else if (x > a[mid]) low = mid + 1;
      else return(mid);
   }
   return(0);
}
```

Analysis

- Time Complexity
  Recurrence relation (for worst case) $T(n) = T(n/2) + c$

Proof: Not available in the notes !



successful searches			unsuccessful searches
$\Theta(1)$,	$\Theta(\log n)$,	$\Theta(\log n)$	$\Theta(\log n)$
best,	average,	worst	best, average, worst

Analysis

Space Complexity

- Iterative Binary search: Constant memory space
- Recursive: proportional to recursion stack.

Pros

- Efficient on very big list,
- Can be implemented iteratively/recursively.

Cons

- Interacts poorly with the memory hierarchy
- Requires sorted list as an input
- Due to random access of list element, needs arrays instead of linked list.

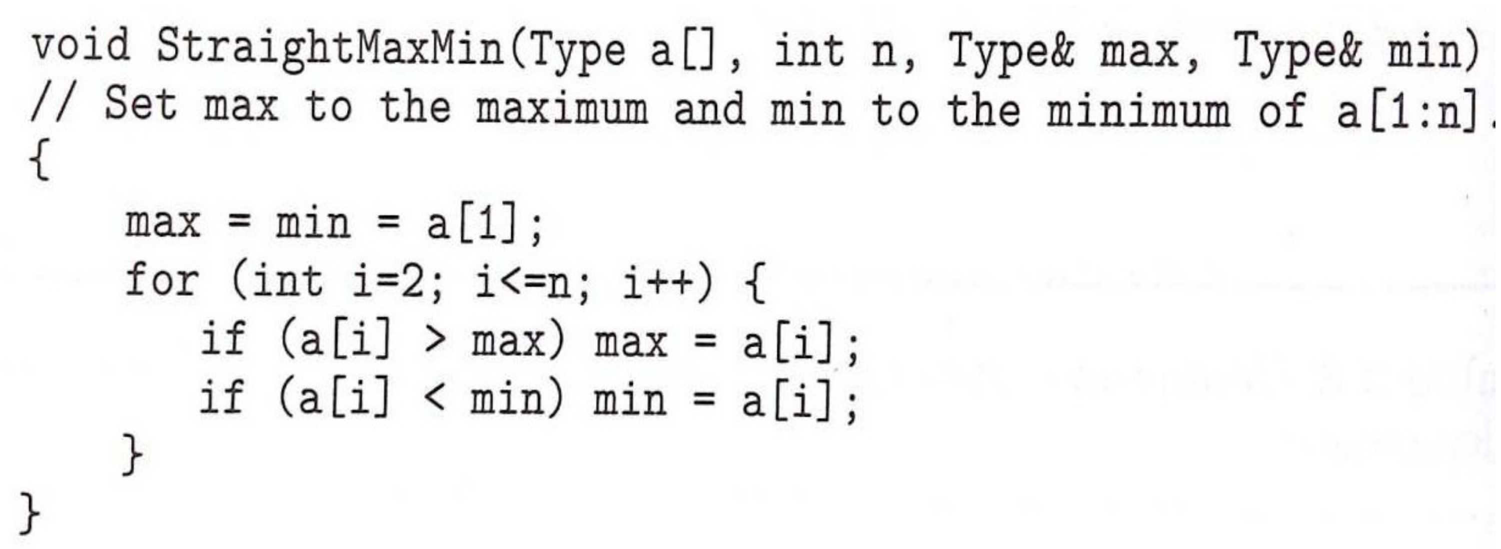
Module 2 – Outline
Divide and Conquer

1. General method
2. Recurrence equation
3. Algorithm: Binary search
4. **Algorithm: Finding the maximum and minimum**
5. Algorithm: Merge sort
6. Algorithm: Quick sort
7. Algorithm: Strassen’s matrix multiplication
8. Advantages and Disadvantages
9. Decrease and Conquer Approach
10. Algorithm: Topological Sort

Max Min

Problem statement

- Given a list of n elements, the problem is to find the maximum and minimum items.
  A simple and straight forward algorithm to achieve this is given below.



```
void StraightMaxMin(Type a[], int n, Type& max, Type& min)
// Set max to the maximum and min to the minimum of a[1:n].
{
    max = min = a[1];
    for (int i=2; i<=n; i++) {
        if (a[i] > max) max = a[i];
        if (a[i] < min) min = a[i];
    }
}
```

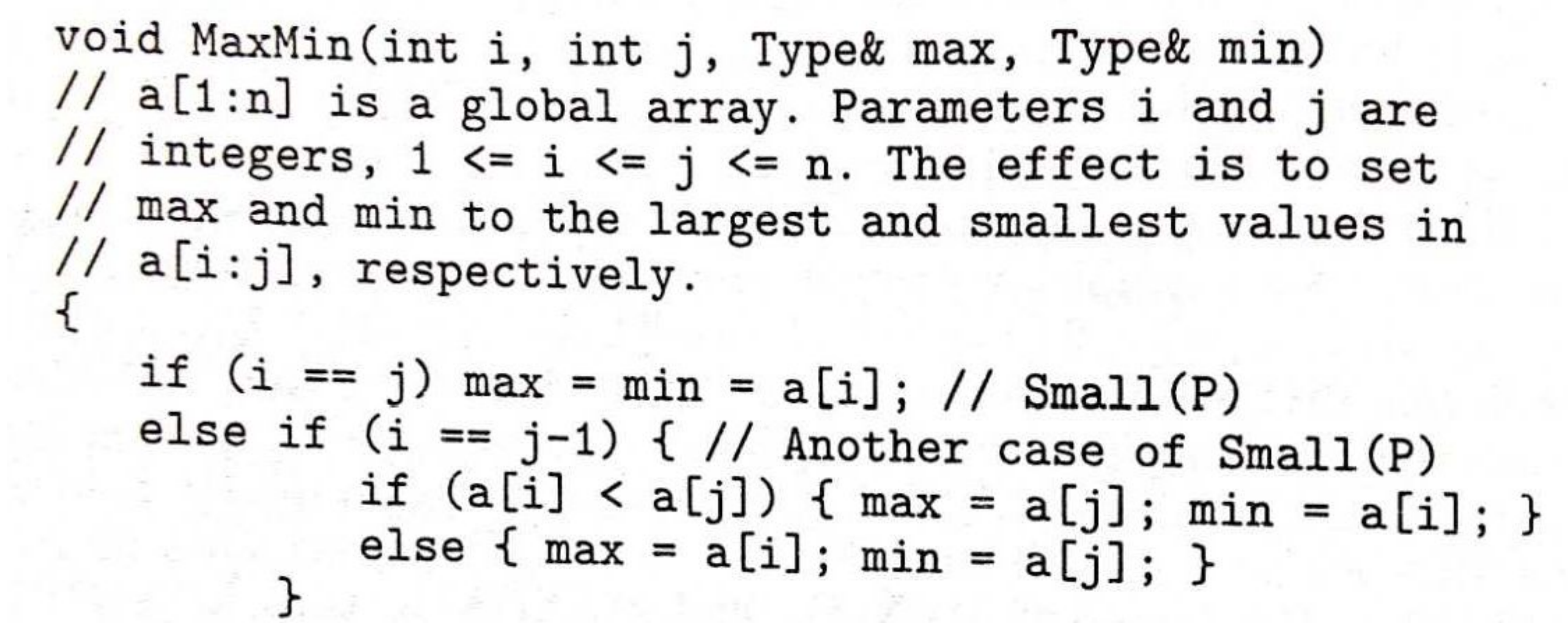
Straight Max Min (Brute Force MaxMin)

- 2(n-1) comparisons in the best, average & worst cases.
- By realizing the comparison of a[i]>max is false, improvement in a algorithm can be done.
  - Hence we can replace the contents of the for loop by,

    If(a[i]>Max) then Max = a[i];

    Else if (a[i]< min) min=a[i]
  - On the average a[i] is > max half the time.
  - So, the avg. no. of comparison is **3n/2-1**.

Algorithm based on D & C strategy



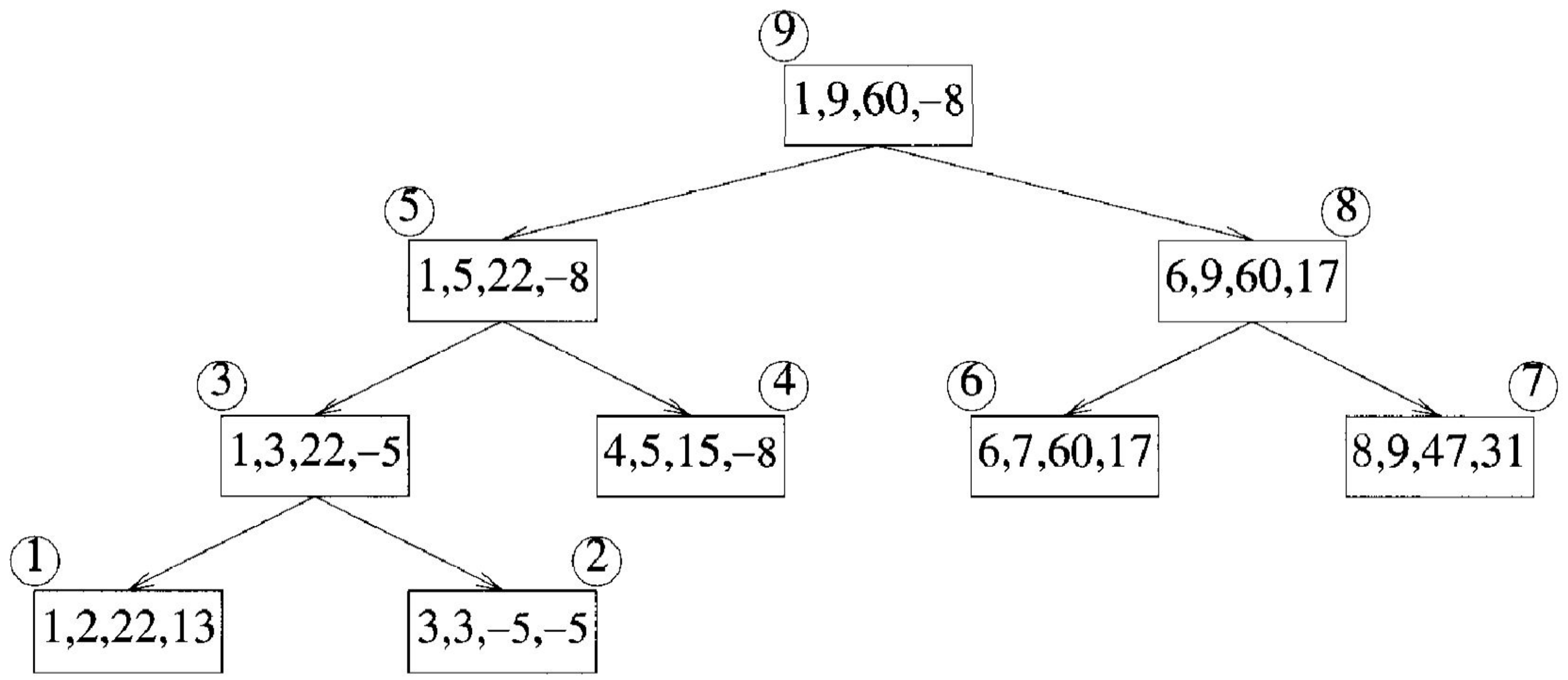
```
void MaxMin(int i, int j, Type& max, Type& min)
// a[1:n] is a global array. Parameters i and j are
// integers, 1 <= i <= j <= n. The effect is to set
// max and min to the largest and smallest values in
// a[i:j], respectively.
{
    if (i == j) max = min = a[i]; // Small(P)
    else if (i == j-1) { // Another case of Small(P)
            if (a[i] < a[j]) { max = a[j]; min = a[i]; }
            else { max = a[i]; min = a[j]; }
        }
```

```
void MaxMin(int i, int j, Type& max, Type& min)
// a[1:n] is a global array. Parameters i and j are
// integers, 1 <= i <= j <= n. The effect is to set
// max and min to the largest and smallest values in
// a[i:j], respectively.
{
    if (i == j) max = min = a[i]; // Small(P)
    else if (i == j-1) { // Another case of Small(P)
            if (a[i] < a[j]) { max = a[j]; min = a[i]; }
            else { max = a[i]; min = a[j]; }
         }
         else { // If P is not small
                 // divide P into subproblems.
             // Find where to split the set.
                 int mid=(i+j)/2; Type max1, min1;
             // Solve the subproblems.
               MaxMin(i, mid, max, min);
               MaxMin(mid+1, j, max1, min1);
             // Combine the solutions.
               if (max < max1) max = max1;
               if (min > min1) min = min1;
         }
     }
```

Example

Suppose we simulate MaxMin on the following nine elements:

a:	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
	22	13	−5	−8	15	60	17	31	47



Analysis - Time Complexity

Now what is the number of element comparisons needed for MaxMin? If $T(n)$ represents this number, then the resulting recurrence relation is

$$T(n) = \begin{cases} T(\lceil n/2 \rceil) + T(\lceil n/2 \rceil) + 2 & n > 2 \\ 1 & n = 2 \\ 0 & n = 1 \end{cases}$$

When n is a power of two, $n = 2^k$ for some positive integer k, then

$$\begin{aligned} T(n) &= 2T(n/2) + 2 \\ &= 2(2T(n/4) + 2) + 2 \\ &= 4T(n/4) + 4 + 2 \\ &\vdots \\ &= 2^{k-1}T(2) + \sum_{1 \le i \le k-1} 2^i \\ &= 2^{k-1} + 2^k - 2 = 3n/2 - 2 \end{aligned} \tag{3.3}$$

Compared with the straight forward method (2n-2)this method saves 25% in comparisons.

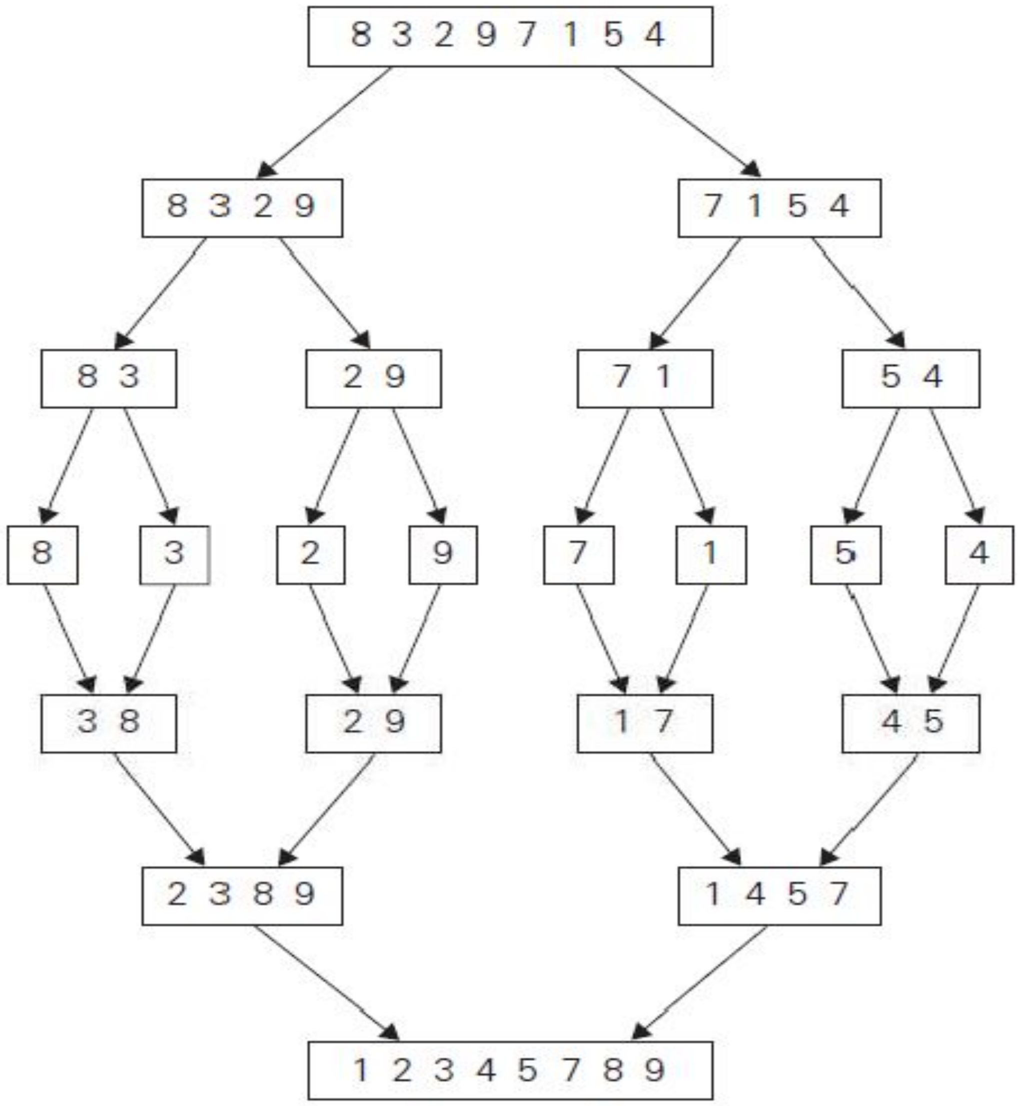
Module 2 – Outline
Divide and Conquer

1. General method
2. Recurrence equation
3. Algorithm: Binary search
4. Algorithm: Finding the maximum and minimum
5. **Algorithm: Merge sort**
6. Algorithm: Quick sort
7. Algorithm: Strassen’s matrix multiplication
8. Advantages and Disadvantages
9. Decrease and Conquer Approach
10. Algorithm: Topological Sort

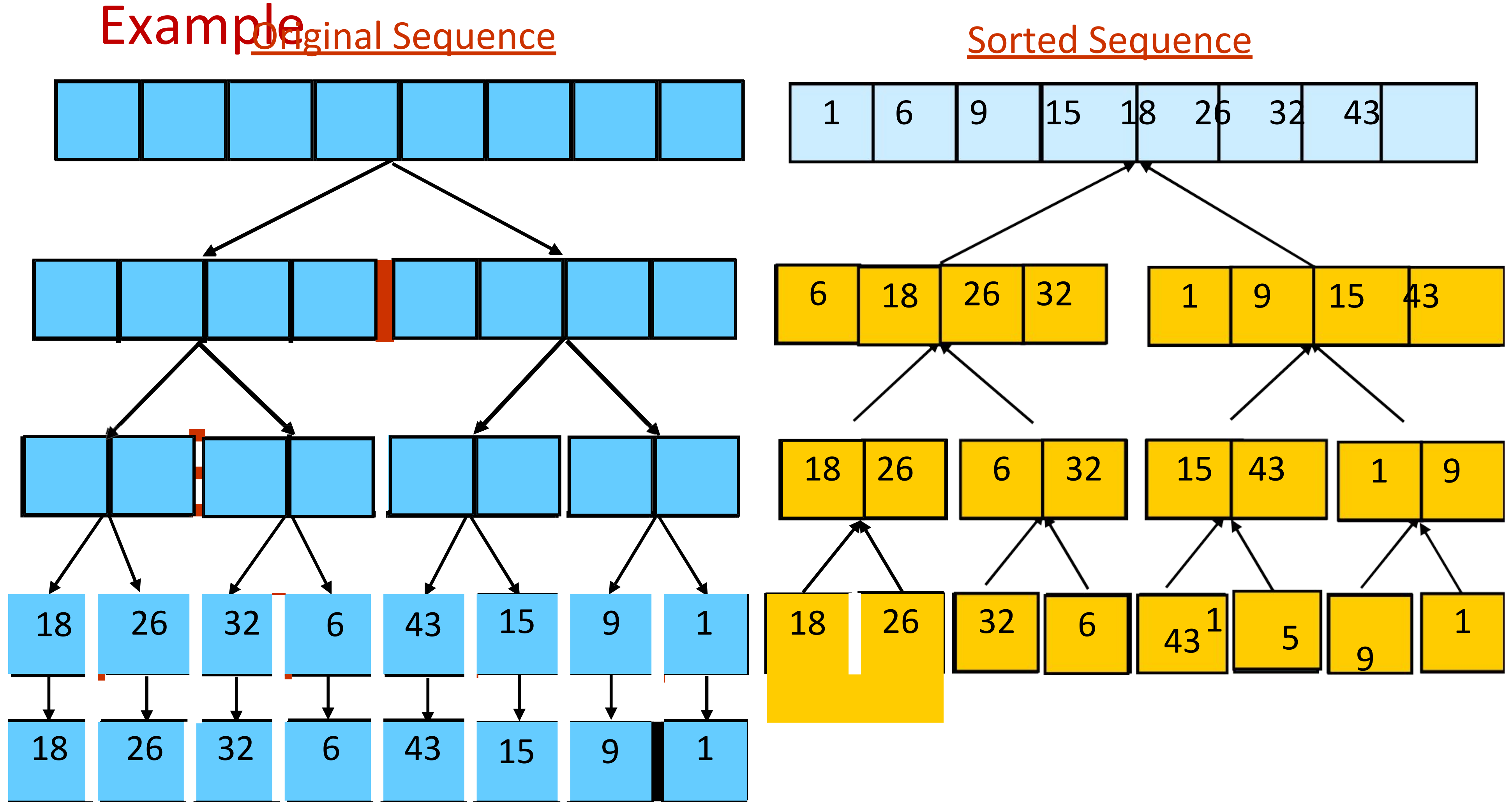
Merge Sort

- Merge sort is a perfect example of divide-and conquer technique.
- It sorts a given array by
  - dividing it into two halves,
  - sorting each of them recursively, and
  - then merging the two smaller sorted arrays into a single sorted one.

Merge sort - example



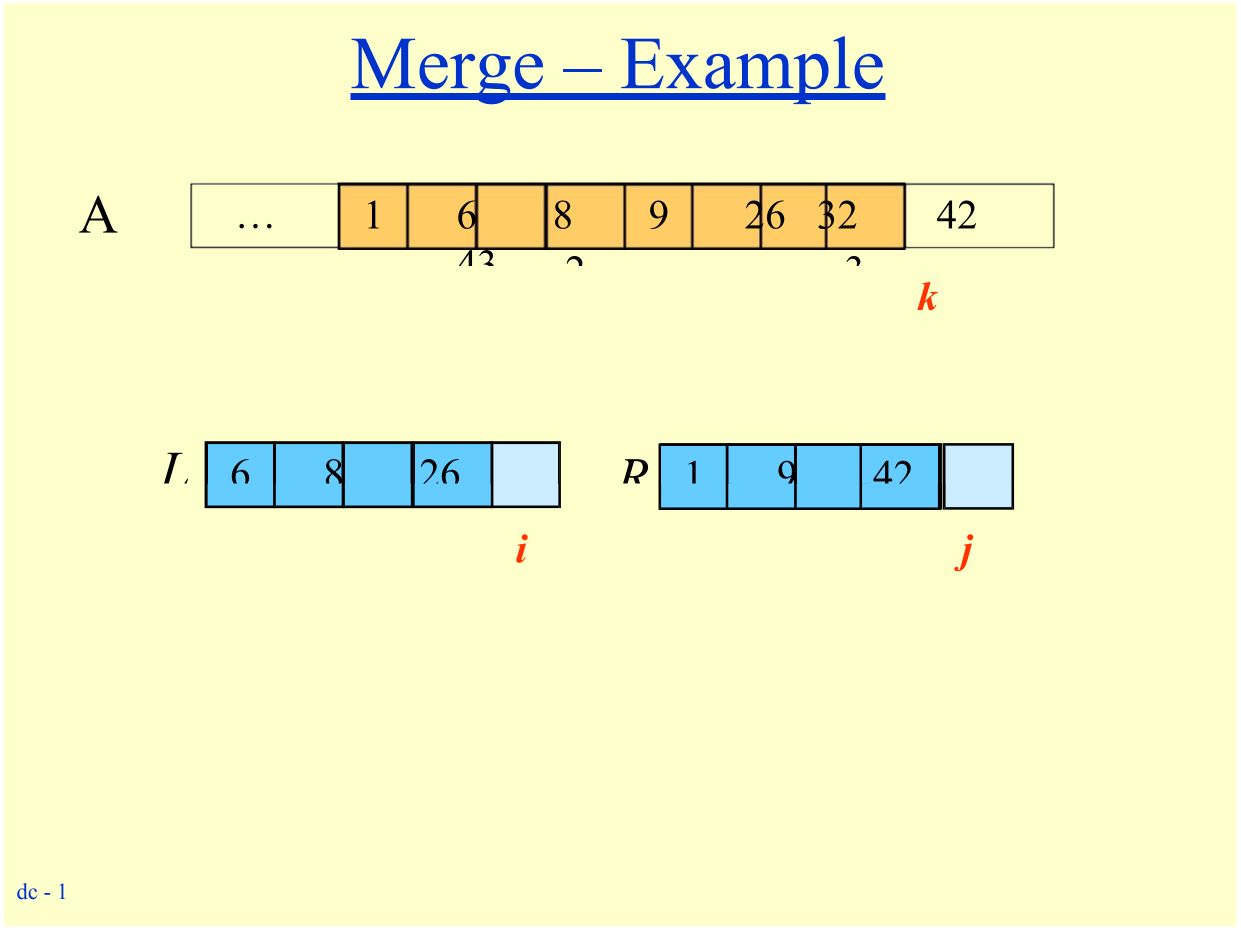
Merge Sort - Example



Merge Sort

```
ALGORITHM  $Mergesort(A[0..n-1])$
    //Sorts array $A[0..n-1]$ by recursive mergesort
    //Input: An array $A[0..n-1]$ of orderable elements
    //Output: Array $A[0..n-1]$ sorted in nondecreasing order
    if $n > 1$
        copy $A[0..\lfloor n/2 \rfloor - 1]$ to $B[0..\lfloor n/2 \rfloor - 1]$
        copy $A[\lfloor n/2 \rfloor..n-1]$ to $C[0..\lceil n/2 \rceil - 1]$
        $Mergesort(B[0..\lfloor n/2 \rfloor - 1])$
        $Mergesort(C[0..\lceil n/2 \rceil - 1])$
        $Merge(B, C, A)$    //see below
```

How to implement merge? (Link to Animated slides)



Merge

```
**ALGORITHM** $Merge(B[0..p-1], C[0..q-1], A[0..p+q-1])$
    //Merges two sorted arrays into one sorted array
    //Input: Arrays $B[0..p-1]$ and $C[0..q-1]$ both sorted
    //Output: Sorted array $A[0..p+q-1]$ of the elements of $B$ and $C$
    $i \leftarrow 0$;  $j \leftarrow 0$;  $k \leftarrow 0$
    while $i < p$ and $j < q$ do
        if $B[i] \leq C[j]$
            $A[k] \leftarrow B[i]$;  $i \leftarrow i + 1$
        else $A[k] \leftarrow C[j]$;  $j \leftarrow j + 1$
        $k \leftarrow k + 1$
    if $i = p$
        copy $C[j..q-1]$ to $A[k..p+q-1]$
    else copy $B[i..p-1]$ to $A[k..p+q-1]$
```

Analysis

- Basic operation - key comparison.
- Best Case, Worst Case, Average Case exists?
  - Execution does not depend on the order of the data
  - Best case and average case runtime are the same as worst case runtime.
- Worst case:
  - During key comparison, neither of the two arrays becomes empty before the other one contains just one element

Analysis – Worst Case

- Assuming for simplicity that total number of elements **n** is a power of 2, the recurrence relation for the number of key comparisons *C*(*n*) is

$$C(n) = 2C(n/2) + C_{merge}(n) \quad \text{for } n > 1, \quad C(1) = 0.$$

- $C_{merge}(n)$ - the number of key comparisons performed during the merging stage.
- At each step, exactly one comparison is made, total comparisons are (n-1)

$$C_{worst}(n) = 2C_{worst}(n/2) + n - 1 \quad \text{for } n > 1, \quad C_{worst}(1) = 0.$$

Analysis

$$C_{worst}(n) = 2C_{worst}(n/2) + n - 1 \quad \text{for } n > 1, \quad C_{worst}(1) = 0.$$

- Here a = 2, b = 2, f (n) = n-1 = Θ (n) => d = 1.
- Therefore 2 = 2^1, case 2 holds in the master theorem
- C_{worst} (n) = Θ (n^d log n) = Θ (n^1 log n) = Θ (n log n)
- Therefore **C_{worst}(n) = Θ (n log n)**

Master theorem

It states that, in recurrence equation T(n) = aT(n/b) + f(n),
If f(n)∈ Θ (n^d) where d ≥ 0 then

$$T(n) \in \begin{cases} \Theta(n^d) & \text{if } a < b^d, \\ \Theta(n^d \log_b n) & \text{if } a = b^d, \\ \Theta(n^{\log_b a}) & \text{if } a > b^d. \end{cases}$$

Advantage
s

- Number of comparisons performed is nearly optimal.
- For large n, the number of comparisons made by this algorithm in the average case turns out to be about 0.25n less and hence is also in *Θ(n* log *n).*
- Mergesort will never degrade to $O(n^2)$
- Another advantage of mergesort over quicksort is its stability.

  (A sorting algorithm is said to be stable if two objects with equal keys appear in the same order in sorted output as they appear in the input array to be sorted.)

Limitations

- The principal shortcoming of mergesort is the linear amount O(n) of extra storage the algorithm requires.
- Though merging can be done in-place, the resulting algorithm is quite complicated and of theoretical interest only.

Variation
s

- The algorithm can be **implemented bottom up** by merging pairs of the array's elements, then merging the sorted pairs, and so on
  - This avoids the time and space overhead of using a stack to handle recursive calls.
- We can divide a list to be sorted in **more than two parts**, sort each recursively, and then merge them together.
  - This scheme, which is particularly useful for sorting files residing on secondary memory devices, is called **multiway mergesort**.

Module 2 – Outline
Divide and Conquer

1. General method
2. Recurrence equation
3. Algorithm: Binary search
4. Algorithm: Finding the maximum and minimum
5. Algorithm: Merge sort
6. **Algorithm: Quick sort**
7. Algorithm: Strassen’s matrix multiplication
8. Advantages and Disadvantages
9. Decrease and Conquer Approach
10. Algorithm: Topological Sort

Quick Sort

Extra!

- It is a Divide and Conquer method
- Sorting happens in Divide stage itself.
- C.A.R. Hoare (also known as Tony Hore), prominent British computer scientist invented quicksort.

Quick Sort

- Quicksort divides (or partitions) array according to the value of some pivot element A[s]
- Divide-and-Conquer:
  - If n=1 terminate (every one-element list is already sorted)
  - If n>1, partition elements into two; based on pivot element

$$\underbrace{A[0]\dots A[s-1]}_{\text{all are } \le A[s]} \quad A[s] \quad \underbrace{A[s+1]\dots A[n-1]}_{\text{all are } \ge A[s]}$$

Quick Sort

$$\underbrace{A[0] \dots A[s-1]}_{\text{all are } \le A[s]} \quad A[s] \quad \underbrace{A[s+1] \dots A[n-1]}_{\text{all are } \ge A[s]}$$

```
ALGORITHM  $Quicksort(A[l..r])$
    //Sorts a subarray by quicksort
    //Input: Subarray of array $A[0..n-1]$, defined by its left and right
    //          indices $l$ and $r$
    //Output: Subarray $A[l..r]$ sorted in nondecreasing order
    if $l < r$
        $s \leftarrow Partition(A[l..r])$  //$s$ is a split position
        $Quicksort(A[l..s-1])$
        $Quicksort(A[s+1..r])$
```

How do we partition?

- There are several different strategies for selecting a pivot and partitioning.
- We use the sophisticated method suggested by C.A.R. Hoare, the inventor of quicksort.

- Select the subarray’s first element: **p = A[l].**
- Now scan the subarray from both ends, comparing the subarray’s elements to the pivot.

How do we partition? (Link to animated slides)

Example

We are given array of n integers to sort:

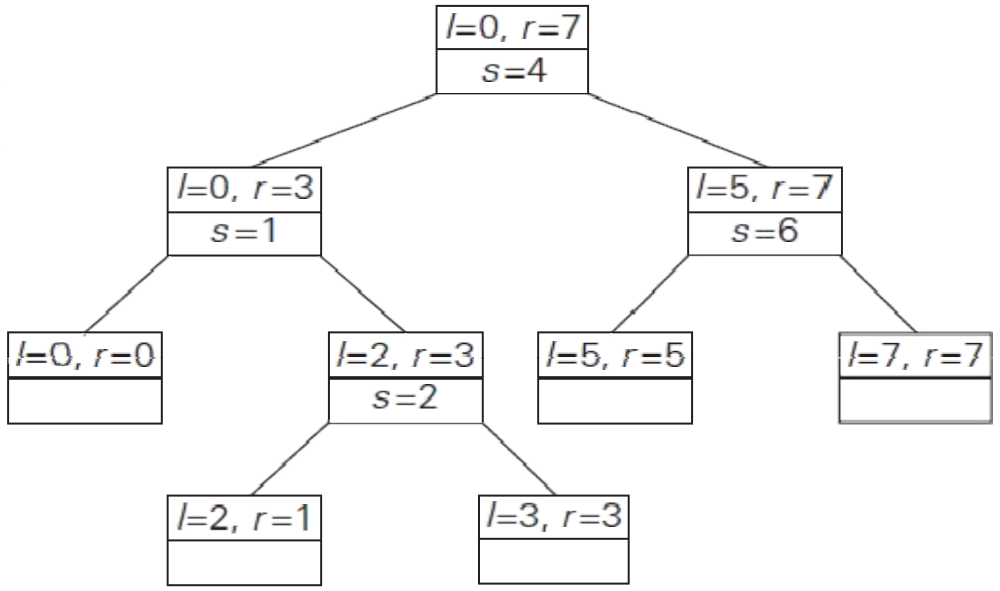
40	20		10	80		60	50	

7 30 100

How do we partition?

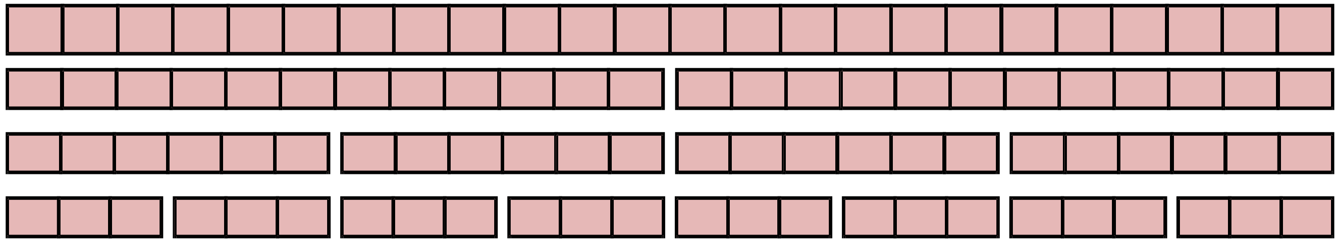
```
ALGORITHM  HoarePartition(A[l..r])
    //Partitions a subarray by Hoare's algorithm, using the first element
    //        as a pivot
    //Input: Subarray of array A[0..n − 1], defined by its left and right
    //        indices l and r (l < r)
    //Output: Partition of A[l..r], with the split position returned as
    //        this function's value
    p ← A[l]
    i ← l; j ← r + 1
    repeat
        repeat i ← i + 1 until A[i] ≥ p
        repeat j ← j − 1 until A[j] ≤ p
        swap(A[i], A[j])
    until i ≥ j
    swap(A[i], A[j])    //undo last swap when i ≥ j
    swap(A[l], A[j])
    return j
```

0	1	2	3	4	5	6	7
5	$\overset{i}{3}$	1	9	8	2	**4**	$\overset{j}{7}$
5	3	1	$\overset{i}{9}$	8	2	$\overset{j}{\mathbf{4}}$	7
		3	$\overset{ij}{\mathbf{4}}$				
		$\overset{j}{\mathbf{3}}$	$\overset{i}{\mathbf{4}}$				
			4				
					8	$\overset{i}{9}$	$\overset{j}{7}$
					8	$\overset{i}{7}$	$\overset{j}{9}$



Analysis

- Basic Operation : Key Comparison
- Best case exists
  - all the splits happen in the middle of subarrays,
  - So the depth of the recursion in $\log_2 n$



$$C_{best}(n) = 2C_{best}(n/2) + n \quad \text{for } n > 1, \quad C_{best}(1) = 0.$$

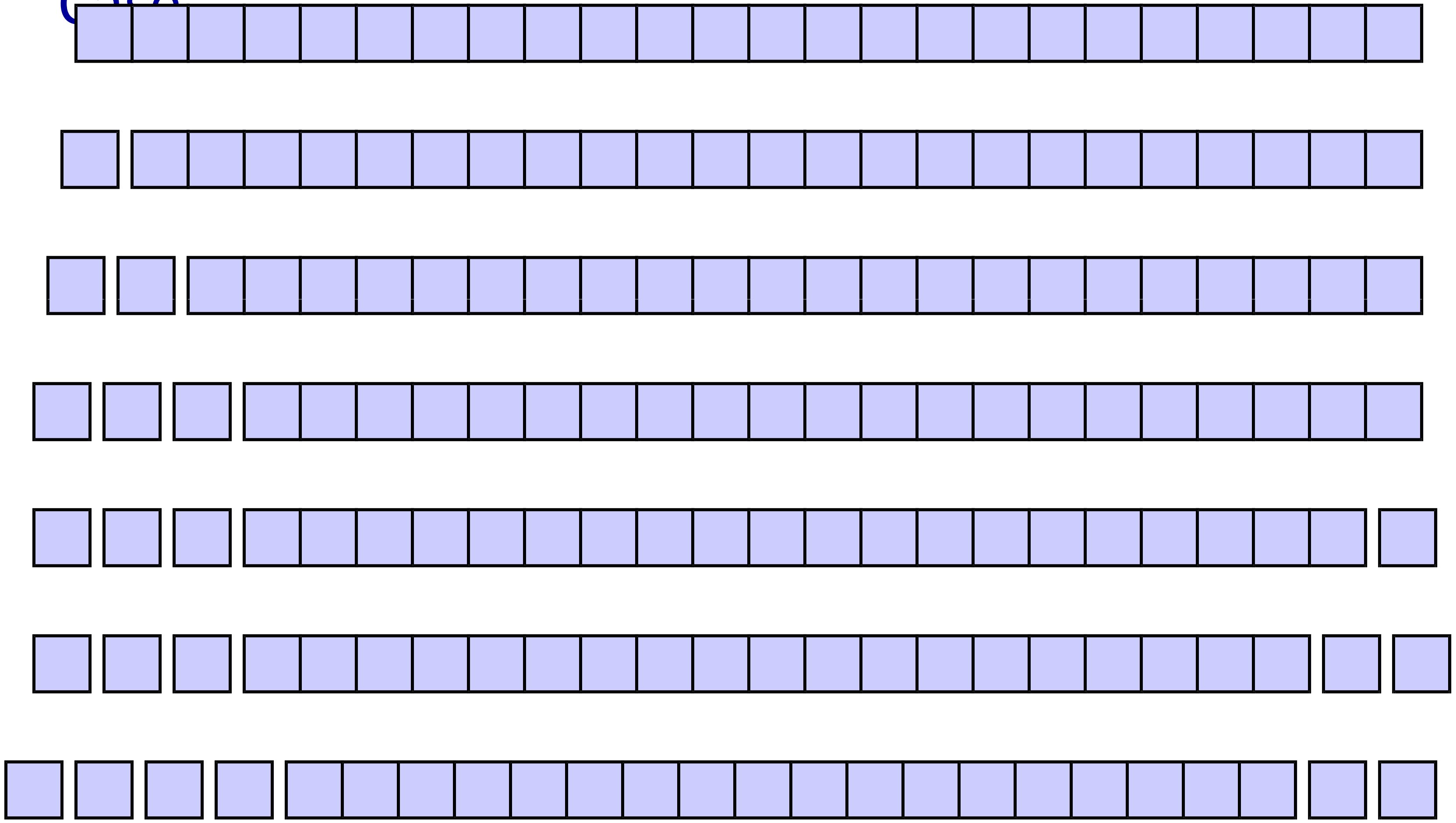
– As per Master Theorem, $C_{best}(n) \in \Theta(n \log_2 n)$;

Analysis

- Worst Case
  - Splits will be skewed to the extreme
  - This happens if the input is already sorted
- In the worst case, partitioning always divides the size n array into these three parts:
  - A length one part, containing the pivot itself
  - A length zero part, and
  - A length n-1 part, containing everything else
- Recurring on the length n-1 part requires (in the worst case) recurring to depth n-1

Analysis

- Worst Case



Analysis

Worst Case

- if A[0..n − 1] is a strictly increasing array and we use A[0] as the pivot,
  - the left-to-right scan will stop on A[1] while the right-to- left scan will go all the way to reach A[0], indicating the split at position 0
  - n + 1 comparisons required

Total comparisons

$$C_{worst}(n) = (n+1) + n + \cdots + 3 = \frac{(n+1)(n+2)}{2} - 3 \in \Theta(n^2).$$

Analysis

Average Case

- Let $C_{avg}(n)$ be the average number of key comparisons made by quicksort on a randomly ordered array of size n.
- A partition can happen in any position s $(0 \le s \le n-1)$
- n+1 comparisons are required for partition.
- After the partition, the left and right subarrays will have s and $n - 1 - s$ elements, respectively.

Analysis

Average Case

- Assuming that the partition split can happen in each position s with the same probability 1/n, we get

$$C_{avg}(n) = \frac{1}{n}\sum_{s=0}^{n-1}[(n+1) + C_{avg}(s) + C_{avg}(n-1-s)] \quad \text{for } n > 1,$$

$$C_{avg}(0) = 0, \quad C_{avg}(1) = 0.$$

$$C_{avg}(n) \approx 2n \ln n \approx 1.39n \log_2 n.$$

Pros and Cons

- Pros
  - Good average case time complexity
- Cons
  - It is not stable.
  - It requires a stack to store parameters of subarrays that are yet to be sorted.

Module 2 – Outline

Divide and Conquer

1. General method
2. Recurrence equation
3. Algorithm: Binary search
4. Algorithm: Finding the maximum and minimum
5. Algorithm: Merge sort
6. Algorithm: Quick sort
7. **Algorithm: Strassen’s matrix multiplication**
8. Advantages and Disadvantages
9. Decrease and Conquer Approach
10. Algorithm: Topological Sort

Matrix Multiplication

Direct Method:

- Suppose we want to multiply two n x n matrices, A and B.
- Their product, C=AB, will be an n by n matrix and will therefore have $\mathbf{n^2}$ elements.
- The number of multiplications involved in producing the product in this way is $\mathbf{\Theta(n^3)}$

$$C(i,j) = \sum_{1 \le k \le n} A(i,k)B(k,j)$$

Matrix Multiplication

- Divide and Conquer method for Matrix multiplication

$$\left[\begin{array}{c|c} C_{00} & C_{01} \\ \hline C_{10} & C_{11} \end{array}\right] = \left[\begin{array}{c|c} A_{00} & A_{01} \\ \hline A_{10} & A_{11} \end{array}\right] * \left[\begin{array}{c|c} B_{00} & B_{01} \\ \hline B_{10} & B_{11} \end{array}\right]$$

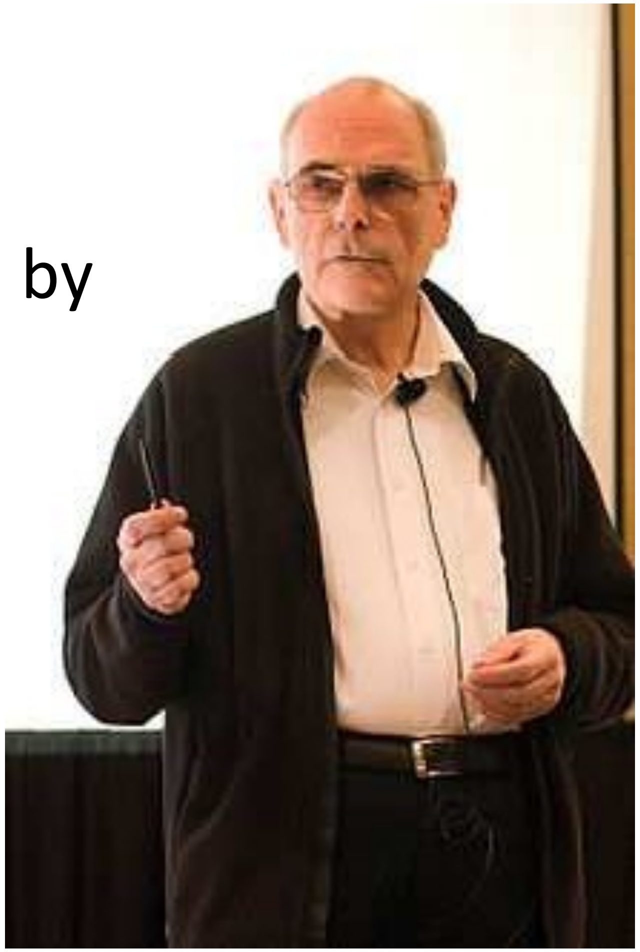
$$= \left[\begin{array}{c|c} A_{00}B_{00} + A_{01}B_{10} & A_{00}B_{01} + A_{01}B_{11} \\ \hline A_{10}B_{00} + A_{11}B_{10} & A_{10}B_{01} + A_{11}B_{11} \end{array}\right]$$

- How many Multiplications?
- 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 $T(n) = 8T(n/2) + O(n^2)$ which happen to be $O(n^3)$; same as the direct method

Stressen’s matrix multiplication

Multiplication of 2 × 2 matrices:

- By using divide-and-conquer approach we can reduce the number of multiplications.
- Such an algorithm was published by V. Strassen in 1969.



Strassen's matrix multiplication

- The principal insight of the algorithm
  - product C of two 2 × 2 matrices A and B
  - with **just seven multiplications**
- This is accomplished by

$$\begin{bmatrix} c_{00} & c_{01} \\ c_{10} & c_{11} \end{bmatrix} = \begin{bmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{bmatrix} * \begin{bmatrix} b_{00} & b_{01} \\ b_{10} & b_{11} \end{bmatrix}$$

$$= \begin{bmatrix} m_1 + m_4 - m_5 + m_7 & m_3 + m_5 \\ m_2 + m_4 & m_1 + m_3 - m_2 + m_6 \end{bmatrix}$$

$$m_1 = (a_{00} + a_{11}) * (b_{00} + b_{11}),$$
$$m_2 = (a_{10} + a_{11}) * b_{00},$$
$$m_3 = a_{00} * (b_{01} - b_{11}),$$
$$m_4 = a_{11} * (b_{10} - b_{00}),$$
$$m_5 = (a_{00} + a_{01}) * b_{11},$$
$$m_6 = (a_{10} - a_{00}) * (b_{00} + b_{01}),$$
$$m_7 = (a_{01} - a_{11}) * (b_{10} + b_{11}).$$

Strassen's matrix multiplication

$$\begin{bmatrix} c_{00} & c_{01} \\ c_{10} & c_{11} \end{bmatrix} = \begin{bmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{bmatrix} * \begin{bmatrix} b_{00} & b_{01} \\ b_{10} & b_{11} \end{bmatrix}$$

$$= \begin{bmatrix} m_1 + m_4 - m_5 + m_7 & m_3 + m_5 \\ m_2 + m_4 & m_1 + m_3 - m_2 + m_6 \end{bmatrix}$$

$$m_1 = (a_{00} + a_{11}) * (b_{00} + b_{11}),$$
$$m_2 = (a_{10} + a_{11}) * b_{00},$$
$$m_3 = a_{00} * (b_{01} - b_{11}),$$
$$m_4 = a_{11} * (b_{10} - b_{00}),$$
$$m_5 = (a_{00} + a_{01}) * b_{11},$$
$$m_6 = (a_{10} - a_{00}) * (b_{00} + b_{01}),$$
$$m_7 = (a_{01} - a_{11}) * (b_{10} + b_{11}).$$

Analysis

- Recurrence relation (considering only multiplication)

$$M(n) = 7M(n/2) \quad \text{for } n > 1, \quad M(1) = 1.$$

Since $n = 2^k$,

$$M(2^k) = 7M(2^{k-1}) = 7[7M(2^{k-2})] = 7^2 M(2^{k-2}) = \cdots$$
$$= 7^i M(2^{k-i}) \cdots = 7^k M(2^{k-k}) = 7^k.$$

Since $k = \log_2 n$,

$$M(n) = 7^{\log_2 n} = n^{\log_2 7} \approx n^{2.807},$$

Analysis(From T2: Horowitz et al)

Suppose if we consider both **multiplication and addition.** The resulting recurrence ration T(n) is

$$T(n) = \begin{cases} b & n \leq 2 \\ 7T(n/2) + an^2 & n > 2 \end{cases}$$

Note: No. of addition/ subtraction Operations $18(n/2)^2 = an^2$

where a and b are constants. Working with this formula, we get

$$\begin{aligned} T(n) &= an^2[1 + 7/4 + (7/4)^2 + \cdots + (7/4)^{k-1}] + 7^k T(1) \\ &\leq cn^2(7/4)^{\log_2 n} + 7^{\log_2 n}, \ c \text{ a constant} \\ &= cn^{\log_2 4 + \log_2 7 - \log_2 4} + n^{\log_2 7} \\ &= O(n^{\log_2 7}) \approx O(n^{2.81}) \end{aligned}$$

Module 2 – Outline
Divide and Conquer

1. General method
2. Recurrence equation
3. Algorithm: Binary search
4. Algorithm: Finding the maximum and minimum
5. Algorithm: Merge sort
6. Algorithm: Quick sort
7. Algorithm: Strassen’s matrix multiplication
8. **Advantages and Disadvantages**
9. Decrease and Conquer Approach
10. Algorithm: Topological Sort

Advantages and Disadvantages of Divide & Conquer

- ✔ **Parallelism:** Divide and conquer algorithms tend to have a lot of inherent parallelism.
- ✔ **Cache Performance:** Once a sub-problem fits in the cache, the standard recursive solution reuses the cached data until the sub-problem has been completely solved.
- ✔ It allows solving **difficult** and often impossible looking problems like the Tower of Hanoi
- X Recursion is slow
  - – sometimes it can become more **complicated than a basic iterative approach**, the same sub problem can occur many times. It is solved again.

Module 2 – Outline
Divide and Conquer

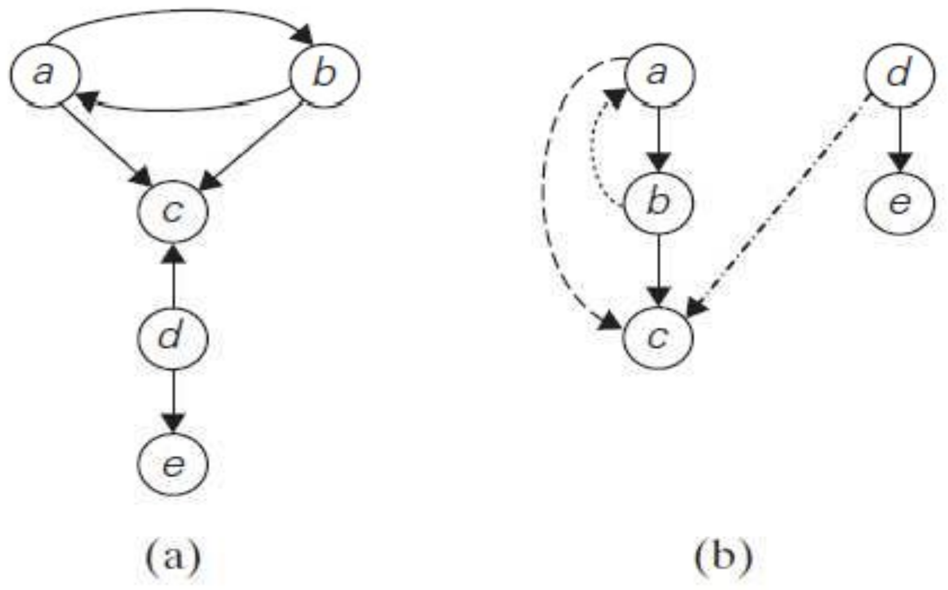
1. General method
2. Recurrence equation
3. Algorithm: Binary search
4. Algorithm: Finding the maximum and minimum
5. Algorithm: Merge sort
6. Algorithm: Quick sort
7. Algorithm: Strassen’s matrix multiplication
8. Advantages and Disadvantages
9. **Decrease and Conquer Approach**
10. **Algorithm: Topological Sort**

Decrease and Conquer Approach

- There are three major variations of decrease-and- conquer:
  - decrease-by-a-constant, most often by one (e.g., insertion sort)
  - decrease-by-a-constant-factor, most often by the factor of two (e.g., binary search)
  - variable-size-decrease (e.g., Euclid’s algorithm)

Topologocal Sorting

- Graph, Digraph
- Adjacency matrix and adjacency list
- DFS, BFS



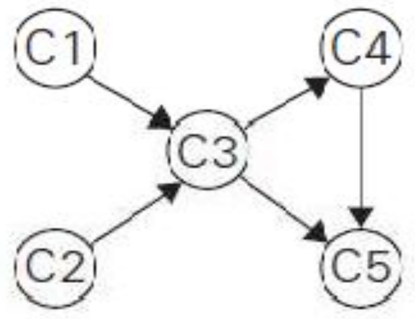
(a) Digraph. (b) DFS forest of the digraph for the DFS traversal started at a.

Digraph

- A ***directed cycle*** in a digraph is a sequence of three or more of its vertices that starts and ends with the same vertex
- For example, *a, b, a* is a directed cycle in the digraph in Figure given above.
- Conversely, if a DFS forest of a digraph has no back edges, the digraph is a ***dag***, an acronym for ***directed acyclic graph***.

Motivation for topological sorting

- Consider a set of five required courses {C1, C2, C3, C4, C5} a part-time student has to take in some degree program.
- The courses can be taken in any order as long as the following course prerequisites are met:
  - C1 and C2 have no prerequisites,
  - C3 requires C1 and C2,
  - C4 requires C3, and
  - C5 requires C3 and C4.
- The student can take only one course per term.
- In which or er should the udent ta he courses?



This problem is called topological sorting.

Topological Sort

- For topological sorting to be possible, a digraph in question must be a DAG.
- There are two efficient algorithms that both verify whether a digraph is a dag and, if it is, produce an ordering of vertices that solves the topological sorting problem.
  - The first one is based on depth-first search
  - the second is based on a direct application of the decrease-by-one technique.

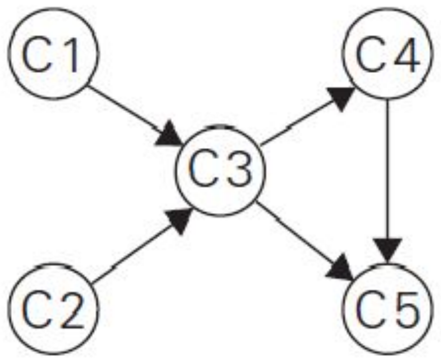
Topological Sorting based on DFS

Method

1. Perform a DFS traversal and note the order in which vertices become dead-ends
2. Reversing this order yields a solution to the topological sorting problem,

   provided, no back edge has been encountered during the traversal.

   If a back edge has been encountered, the digraph is not a dag, and topological sorting of its vertices is impossible.



(a)

$C5_1$

$C4_2$

$C3_3$

$C1_4$ $C2_5$

(b)

The popping-off order:

C5, C4, C3, C1, C2

The topologically sorted list:

$$\begin{tikzcd}
C2 \arrow[rr, bend right] & C1 \arrow[r] & C3 \arrow[r] \arrow[rr, bend right] & C4 \arrow[r] & C5
\end{tikzcd}$$

(c)

Topological Sorting using decrease-and-conquer technique:

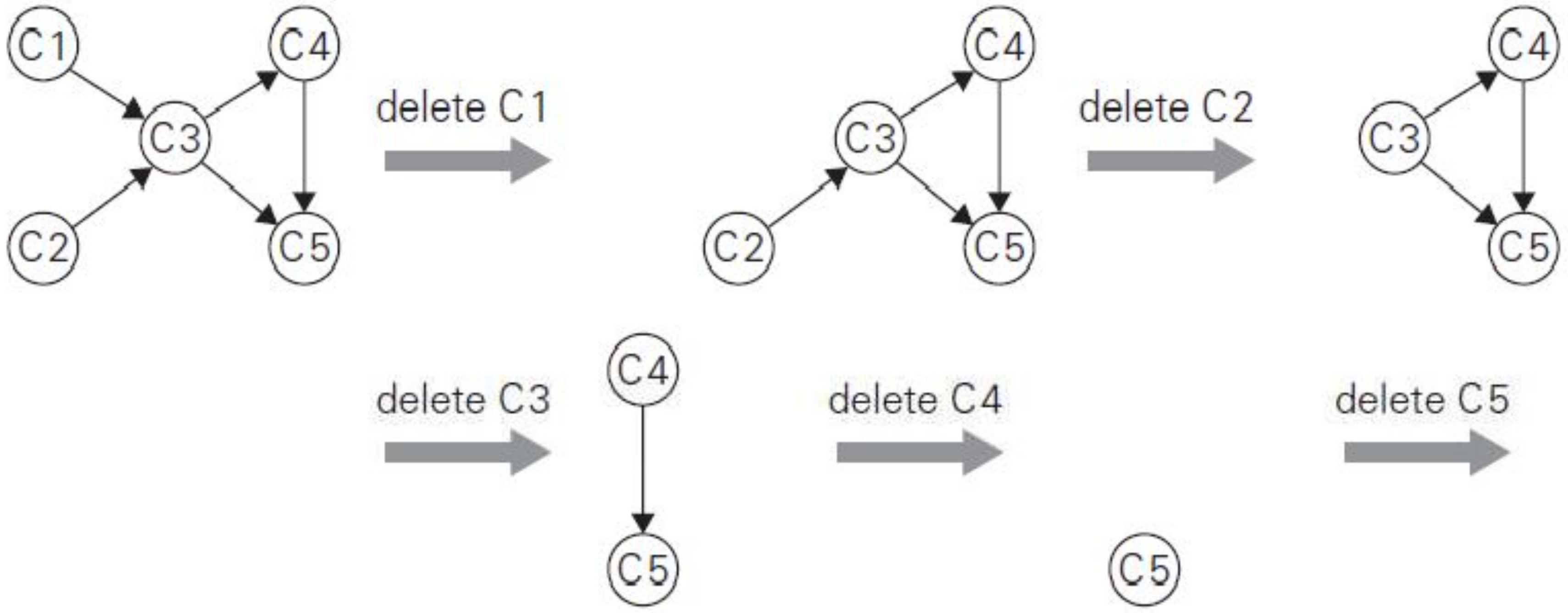
Method: The algorithm is based on a direct implementation of the decrease-(by one)-and-conquer technique:

1. Repeatedly, identify in a remaining digraph a source, which is a vertex with no incoming edges, and delete it along with all the edges outgoing from it.

   (If there are several sources, break the tie arbitrarily. If there are none, stop because the problem cannot be solved.)

2. The order in which the vertices are deleted yields a solution to the topological sorting problem.

Illustration



The solution obtained is C1, C2, C3, C4, C5

Applications of Topological Sorting

- Observation: Topological sorting problem may have several **alternative solutions**.

- Instruction scheduling in program compilation
- Cell evaluation ordering in spreadsheet formulas,
- Resolving symbol dependencies in linkers.

Summary

- Divide and Conquer
  - Recurrence equation
  - Binary search
  - Finding the maximum and minimum
  - Merge sort
  - Quick sort
  - Strassen’s matrix multiplication

- Advantages and Disadvantages of D & C

- Decrease and Conquer Approach
  - Algorithm: Topological Sort

Assignment-2 Due: Within 5 days

1. Solve the following recurrence relation by substitution method. $T(n) = 9T(n/3)+4n^6$, $n \geq 3$ and n is a power of 3

2. Discuss how quick-sort works to sort an array and trace for the following dataset. Draw the tree of recursive calls made.

   65, 70, 75, 80, 85, 60, 55, 50, 45

3. What are the three major variations of decrease and conquer technique? Explain with an example for each.

4. Apply Strassen's matrix multiplication to multiply following matrices. Discuss method is better than direct matrix multiplication method

$$\begin{bmatrix} 4 & 3 \\ 2 & 1 \end{bmatrix} \times \begin{bmatrix} 2 & 5 \\ 1 & 6 \end{bmatrix}$$

Extra Byte

1. **Arrange +ve and –ve:** Design an algorithm (using divide and conquer) to rearrange elements of a given array of *n* real numbers so that all its negative elements precede all its positive elements. Your algorithm should be both time efficient and space efficient.

2.: The ***Dutch national flag problem*** is to rearrange an array of characters *R*, *W*, and *B* (red, white, and blue are the colors of the Dutch national flag) so that all the *R*'s come first, the *W*'s come next, and the *B*'s come last. Design a linear in-place algorithm for this problem.