UNIT II
Disjoint Sets, Spanning Trees Divide and Conquer

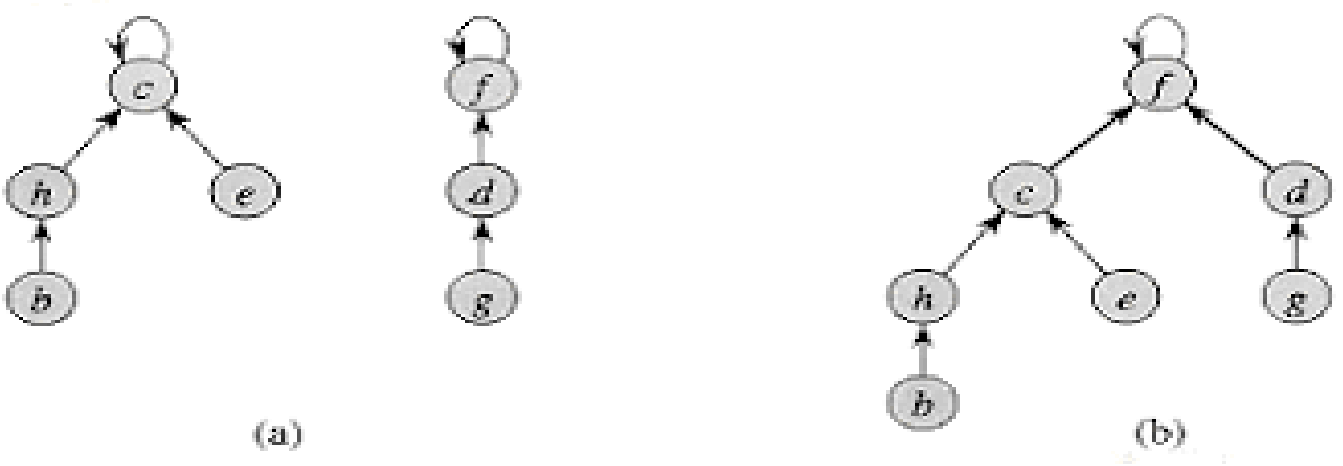
Disjoint Sets

- In computer science, a **set** is an abstract **data type** that can store unique values, without any particular order. It is a computer implementation of the mathematical concept of a finite **set**. ... Some **set data structures** are designed for static or frozen **sets** that do not change after they are constructed.
- Assume that the sets being represented are pair wise disjoint i.e: ,if S_a and S_b , a!=b, are two sets, then there is no element that is in both S_a and $S_{b.}$ [$S_a \cap S_b = \emptyset$]
- The operations we wish to perform on these sets are:

1. Disjoint set union: If S_a and S_b are two disjoint sets, then their union $S_a \cup S_b$ = all elements x such that x is in S_a or S_b.
2. Find : Find the existence of the element in the set, otherwise returns null

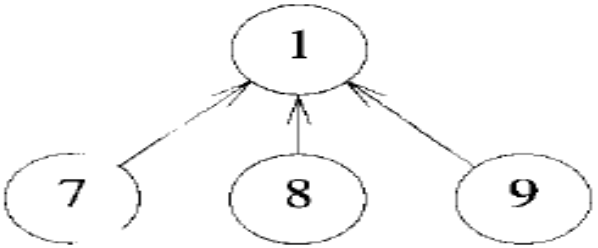
Set Union operation using tree representation

- Consider two Sets Set1 S_1 = { c, h, e, b} , Set2 S_2 = { f, d, g}.
- Above two sets are said to be disjoint sets, since there is no common element between these two sets.
- We perform Set Union operation between these two sets.
- Then S_1 **U** S_2 = {c, h, e, b, f, d, g}.
- Tree Representation of Sets are shown in following diagram with union operation between these sets.
- In Tree representation relation between two elements represented by upward arrow (**↑),** means traversing of an element is done from leaf to root (bottom to top).

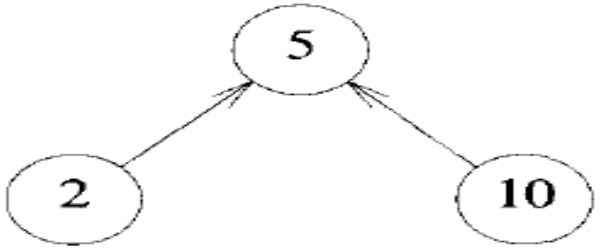


Set Union operation using tree representation

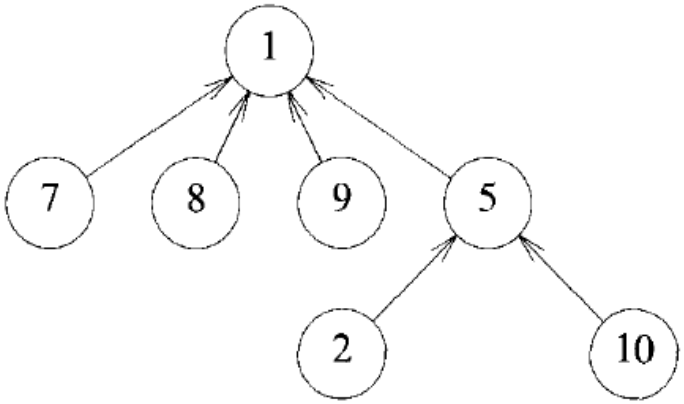
Example : Consider the following 2 sets



S_1

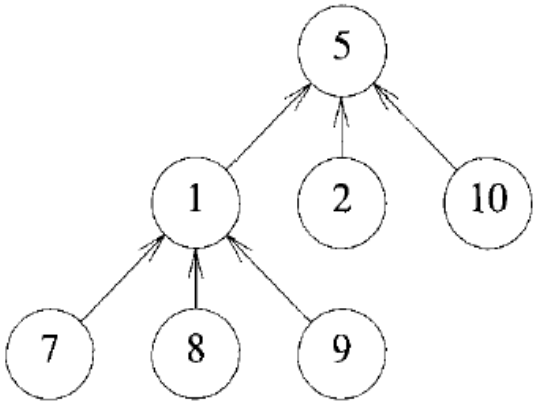


S_2



$S_1 \cup S_2$

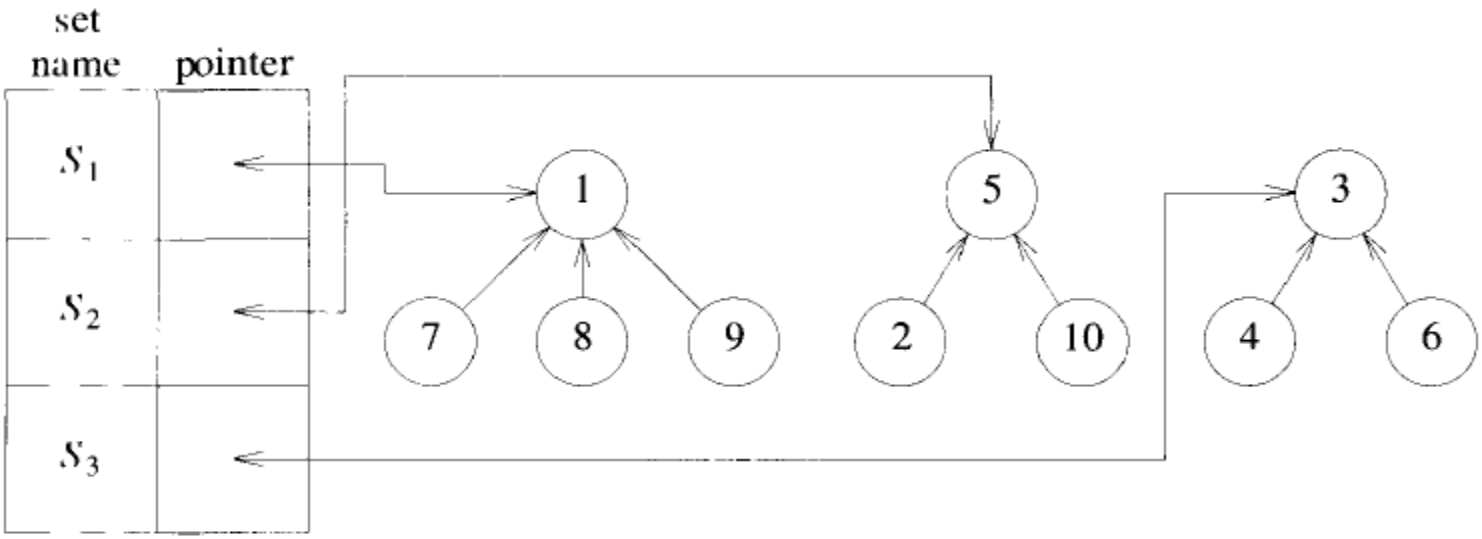
or



$S_1 \cup S_2$

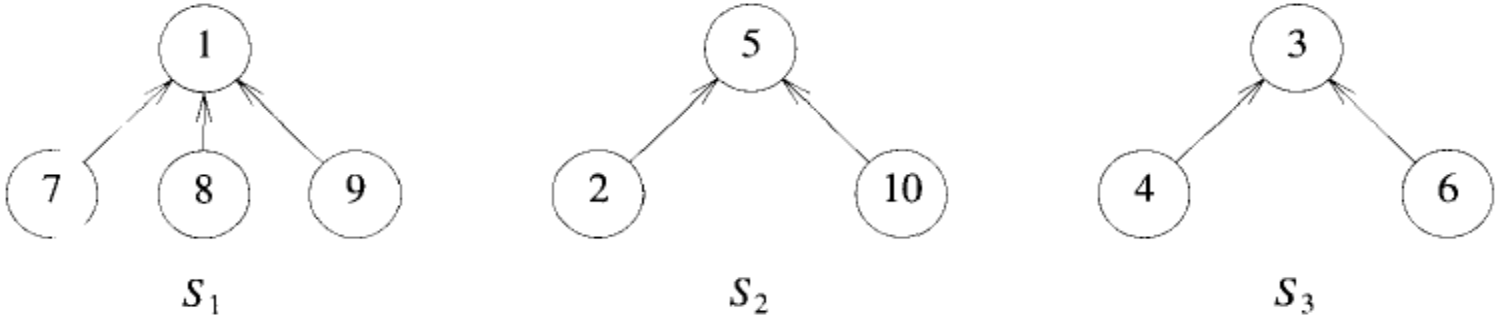
Data Representation for Sets S_1, S_2 and S_3

- Consider 3 sets S1 = {1, 7, 8,9} S2 = { 5, 2, 10} S3 = {3, 4, 6} their representation using set pointers



- The transition to set names is easy. If we determine that element **i** is in a tree with root **j**, and **j** has a pointer to entry **k** in the set name table, then the set name is just **name[k] (Example: S1 [2000])**.
- If we wish to unite sets S_i and S_j, then we wish to unite the trees with roots FindPointer(S_i) and FindPointer(S_j)

Array Representation of Sets S_1, S_2 and S_3



i	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]
p	−1	5	−1	3	−1	3	1	1	1	5

•Since the set elements are numbered 1 through n, we represent the tree nodes using an array p{1:n], where n is the maximum number of elements.

•The i^{th} element of this array represents the tree node that contains element i.

•This array element gives the parent pointer of the corresponding tree . Array representation of the sets S_1, S_2 and S_3 is shown above.

•Notice that root nodes have a parent of -1.

Find Algorithm for Sets

- Algorithm SimpleFind(j)

```
{
  while (p[i] >=0)
    do
      {
        i:=p[i];
      }
   return i;
}
```

- We can now implement Find(i) by following the indices, starting at i until we reach a node with parent value -1.
- For example, Find(6) starts at 6 and then moves to 6'sparent,3. Since p[3] is negative, we have reached the root

Union Algorithm for Sets

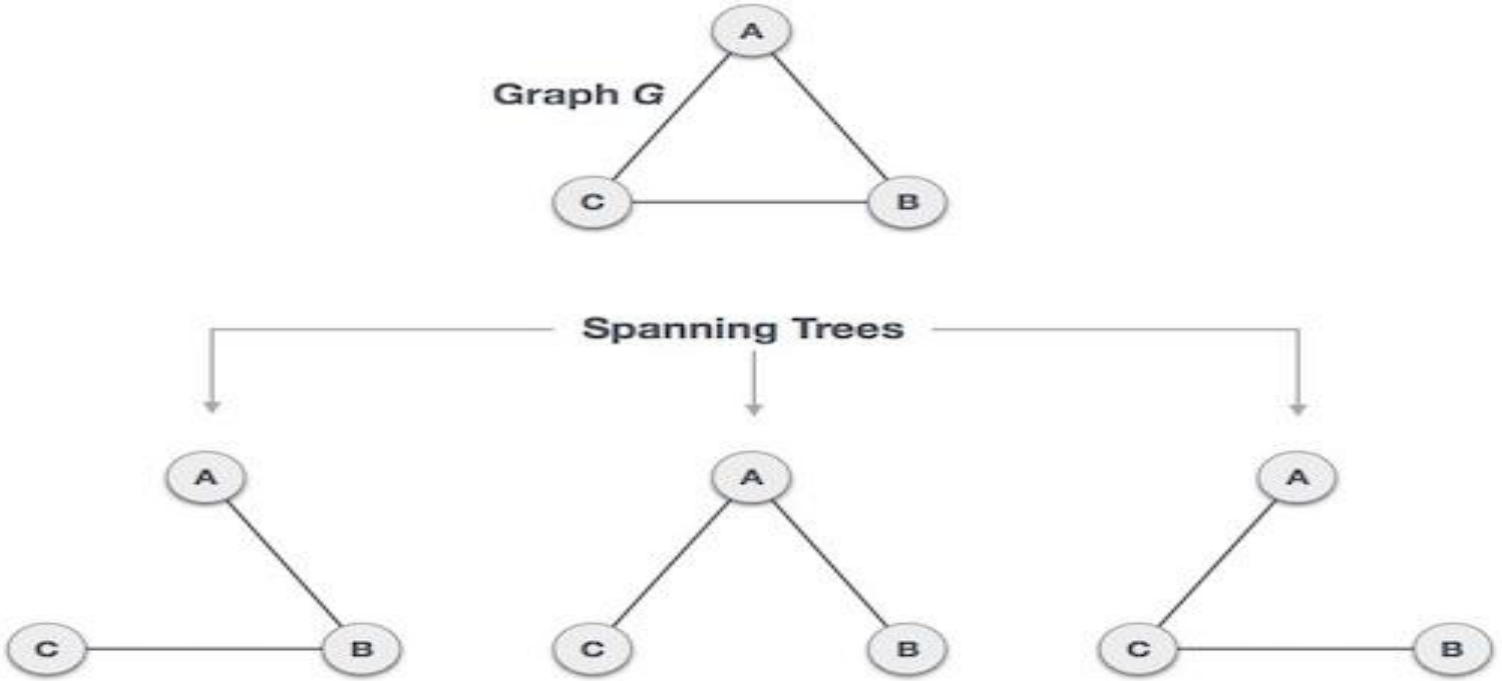
- Algorithm SimpleUnion(i, j)

```
{
  p[i] :=j;
}
```

- The operation Union(i,j) equally simple. We pass in two trees with roots i and j.
- Adopting the convention that the first tree becomes a sub tree of the second tree, the statement p[i] :=j; accomplishes the union.

Spanning Trees

- Definition : Let G = (V, E) be an undirected connected graph. A sub graph t = (V, E')of G is a spanning tree of G if and only if t is a tree. (Avoiding the loops in Graph)



•We found three spanning trees off one complete graph. A complete undirected graph can have maximum $\mathbf{n^{n-2}}$ number of spanning trees, where **n** is the number of nodes.

•In the above addressed example, **n is 3,** hence $\mathbf{3^{3-2} = 3}$ spanning trees are possible.

Spanning tree Properties

- General Properties of Spanning Tree: Following are a few properties of the spanning tree connected to graph G

✓ A connected graph G can have more than one spanning tree.

✓ All possible spanning trees of graph G, have the same number of edges and vertices.

✓ The spanning tree does not have any cycle (loops).

✓ Removing one edge from the spanning tree will make the graph disconnected, i.e. the spanning tree is minimally connected.

✓ Adding one edge to the spanning tree will create a circuit or loop, i.e. the spanning tree is maximally acyclic.

Spanning tree Properties contd..

- Other Properties of Spanning Tree:

✓ Spanning tree has n-1 edges, where n is the number of nodes (vertices).

✓ From a complete graph, by removing maximum e - n + 1 edges, we can construct a spanning tree. (From above ex: We need to remove 3-3+1 edges to construct spanning tree)

✓ A complete graph can have maximum n^{n-2} number of spanning trees. (If n = 4 vertices, then $4^{4-2} = 4^2 = 16$ spanning trees possible)

- Thus, we can conclude that spanning trees are a subset of connected Graph G and disconnected graphs do not have spanning tree.

Applications of Spanning Trees

Application of Spanning Tree are….

- Spanning tree is basically used to find a minimum path to connect all nodes in a graph. Common application of spanning trees are −

✓ Civil Network Planning

✓ Computer Network Routing Protocol

✓ Cluster Analysis

•The minimum cost spanning tree for a given graph is constructed using two algorithms.

1. Prim's Algorithm
2. Kruskal's Algorithm

We will discuss these two algorithms in Unit 3.

Design Methodologies for developing Algorithms

- **Divide and Conquer (Discussed in II Unit)**
- **Greedy method (Discussed in III Unit)**
- **Dynamic Programming (Discussed in IV Unit)**
- **Backtracking (Discussed In V Unit)**
- **Branch and Bound (Discussed In V Unit)**
- **NP-hard and NP-Complete problems (Discussed in VI Unit)**

Divide and Conquer Method

General method:

- ✓ In divide and conquer approach, the problem in hand, is divided into smaller sub-problems and then each problem is solved independently.
- ✓ When we keep on dividing the sub problems into even smaller sub-problems, we may eventually reach a stage where no more division is possible.
- ✓ Those "atomic" smallest possible sub-problem (fractions) are solved. The solution of all sub-problems is finally merged in order to obtain the solution of an original problem.

Divide and Conquer Method contd..

divide-and-conquer approach is given in a three-step process.

- **Divide/Break** : This step involves breaking the problem into smaller sub-problems. Sub-problems should represent a part of the original problem. This step generally takes a recursive approach to divide the problem until no sub-problem is further divisible. At this stage, sub-problems become atomic in nature but still represent some part of the actual problem.
- **Conquer/Solve**: This step receives a lot of smaller sub-problems to be solved. Generally, at this level, the problems are considered 'solved' on their own.
- **Merge/Combine**: When the smaller sub-problems are solved, this stage recursively combines them until they formulate a solution of the original problem. This algorithmic approach works recursively and conquer & merge steps works so close that they appear as one.

Divide and Conquer outline

- Divide and conquer.
  - Break up problem into several parts.
  - Solve each part recursively.
  - Combine solutions to sub-problems into overall solution.

- Most common usage.
  - Break up problem of size n into two equal parts of size ½n.
  - Solve two parts recursively.
  - Combine two solutions into overall solution in linear time.

- Consequence.
  - Brute force: n^2.
  - Divide-and-conquer: n log n.

Applications of Divide and Conquer method:

- ✓ Binary search
- ✓ Quick sort
- ✓ Merge sort
- ✓ Multiplication of large integers
- ✓ Strassen's matrix multiplication.

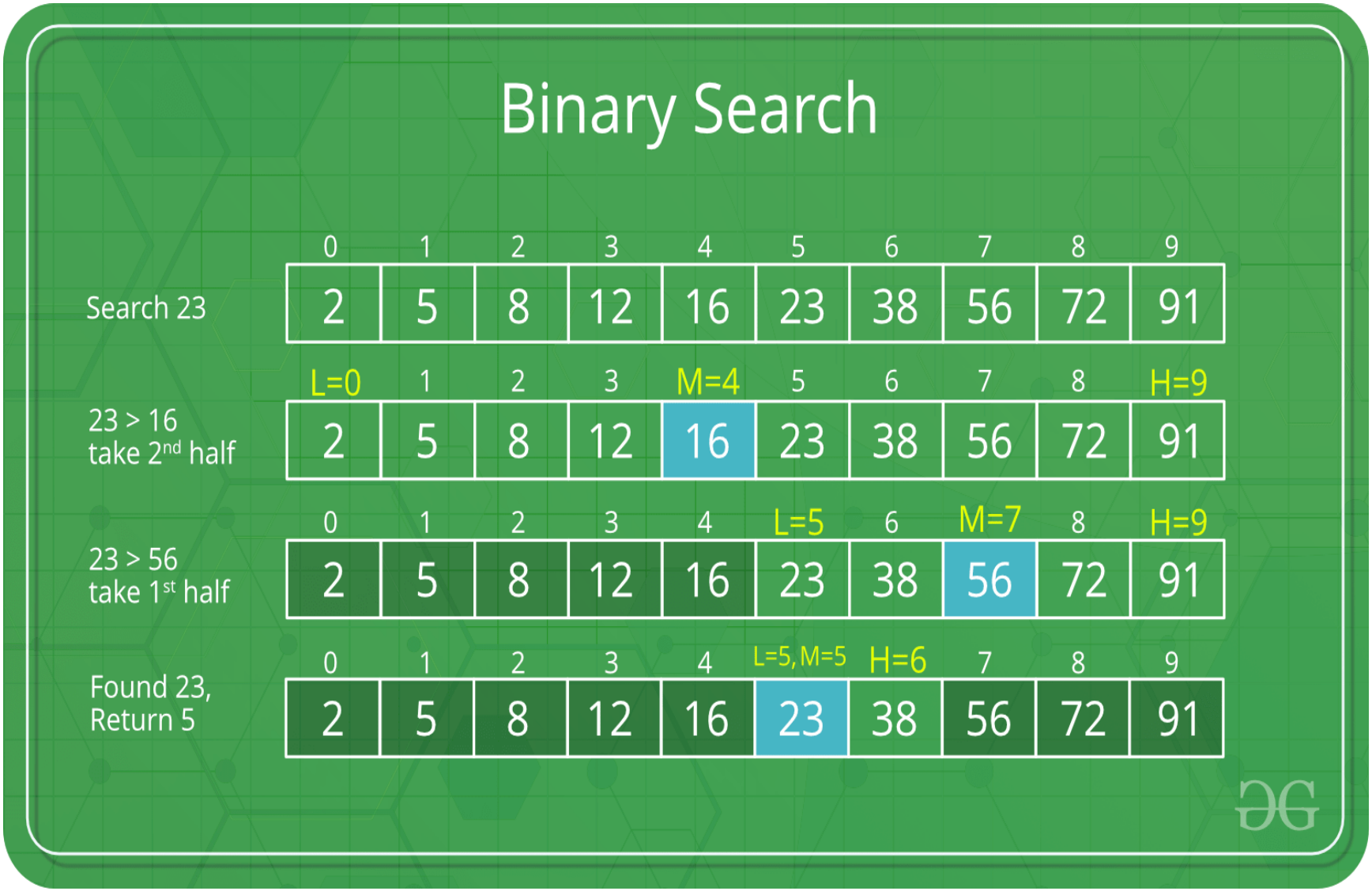
Binary Search

- Binary search can be performed on a sorted array. In this approach, the index of an element **x** is determined if the element belongs to the list of elements.
- In this algorithm, we want to find whether element **x** belongs to a set of numbers stored in an array ***numbers[]***. Where ***l*** and ***r*** represent the left and right index of a sub-array in which searching operation should be performed.
- **Algorithm:**

Binary-Search(numbers[], x, l, r)

```
if l = r then
   return r
else m := ⌊(l + r) / 2⌋
if x ≤ numbers[m] then
   return  Binary-Search(numbers[], x, l, m)
else
   return Binary-Search(numbers[], x, m+1, r)
```

Binary Search Example



Binary Search contd..

Analysis

- Binary search produces the result in ***O(log n)*** time

  Let **T(n)** be the number of comparisons in worst-case in an array of **n** elements.

- Hence,

✓ T(n)= 0 if n=1

✓ T(n)= T(n/2)+1 otherwise

Solving this recurrence relation yields to T(n)=log n

T(n)=log n.

- Therefore, binary search uses O(log n) time.

Sorting applications

- Sorting. Given n elements, rearrange in ascending order.

Obvious sorting applications.

- List files in a directory.
- Organize an MP3 library.
- List names in a phone book.
- Display Google Page Rank results.

Problems become easier once sorted.

- Find the median.
- Find the closest pair.
- Binary search in a database.
- Identify statistical outliers.
- Find duplicates in a mailing list.

Non-obvious sorting applications.

- Data compression.
- Computer graphics.
- Interval scheduling.
- Computational biology.
- Minimum spanning tree.
- Supply chain management.
- Simulate a system of particles.
- Book recommendations on Amazon.
- Load balancing on a parallel computer.
- . . .

Merge sort

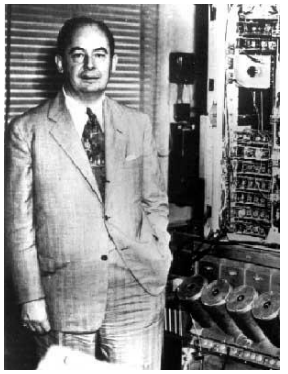
- In Merge sort, sorting a list of numbers implements itself immediately to a divide-and-conquer strategy: split the list into two halves, recursively sort each half, and then merge the two sorted sub-lists.

```
1   Algorithm MergeSort(low, high)
2   // a[low : high] is a global array to be sorted.
3   // Small(P) is true if there is only one element
4   // to sort. In this case the list is already sorted.
5   {
6       if (low < high) then  // If there are more than one element
7       {
8           // Divide P into subproblems.
9               // Find where to split the set.
10                  mid := ⌊(low + high)/2⌋;
11          // Solve the subproblems.
12              MergeSort(low, mid);
13              MergeSort(mid + 1, high);
14          // Combine the solutions.
15              Merge(low, mid, high);
16      }
17  }
```

Merge sort contd..

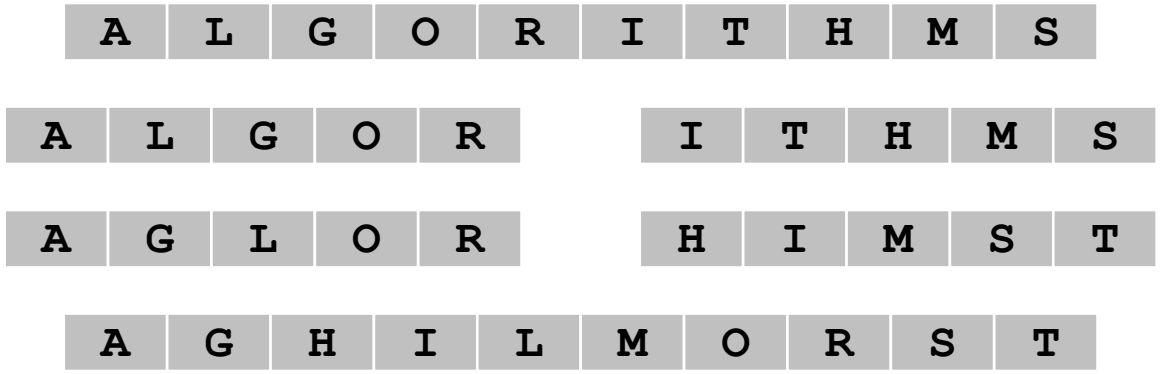
```
1    Algorithm Merge(low, mid, high)
2    // a[low : high] is a global array containing two sorted
3    // subsets in a[low : mid] and in a[mid + 1 : high]. The goal
4    // is to merge these two sets into a single set residing
5    // in a[low : high]. b[ ] is an auxiliary global array.
6    {
7        h := low; i := low; j := mid + 1;
8        while ((h ≤ mid) and (j ≤ high)) do
9        {
10           if (a[h] ≤ a[j]) then
11           {
12               b[i] := a[h];h := h + 1;
13           }
14           else
15           {
16               b[i] := a[j]; j := j + 1;
17           }
18           i := i + 1;
19       }
20       if (h > mid) then
21           for k := j to high do
22           {
23               b[i] := a[k]; i := i + 1;
24           }
25       else
26           for k := h to mid do
27           {
28               b[i] := a[k]; i := i + 1;
29           }
30       for k := low to high do a[k] := b[k];
31   }
```

Mergesort



Jon von Neumann (1945)

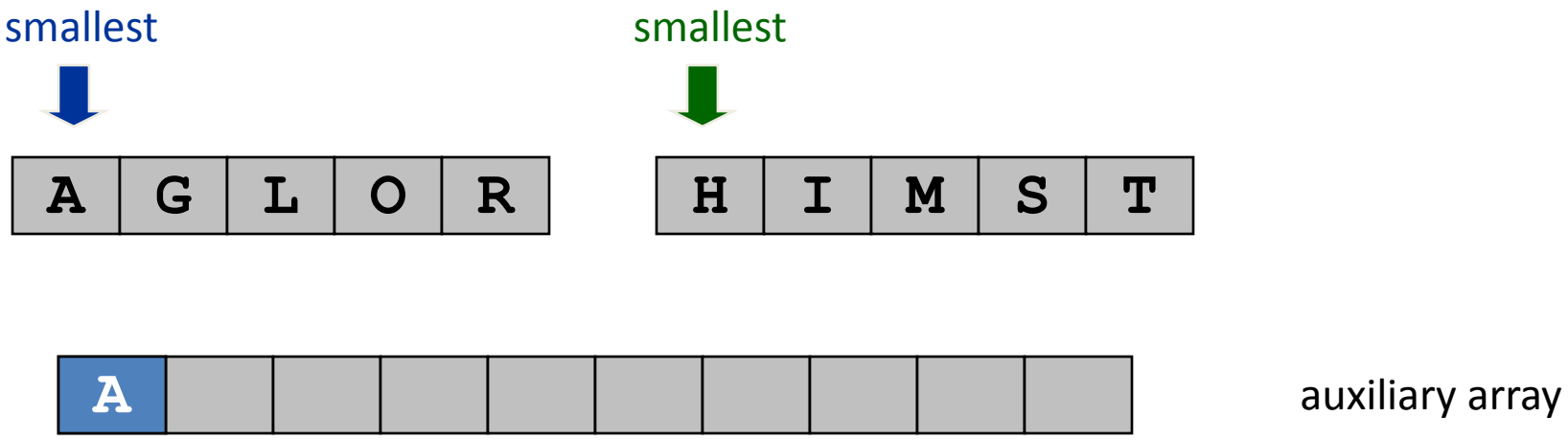
- Mergesort.
  - Divide array into two halves.
  - Recursively sort each half.
  - Merge two halves to make sorted whole.



divide	O(1)
sort	2T(n/2)
merge	O(n)

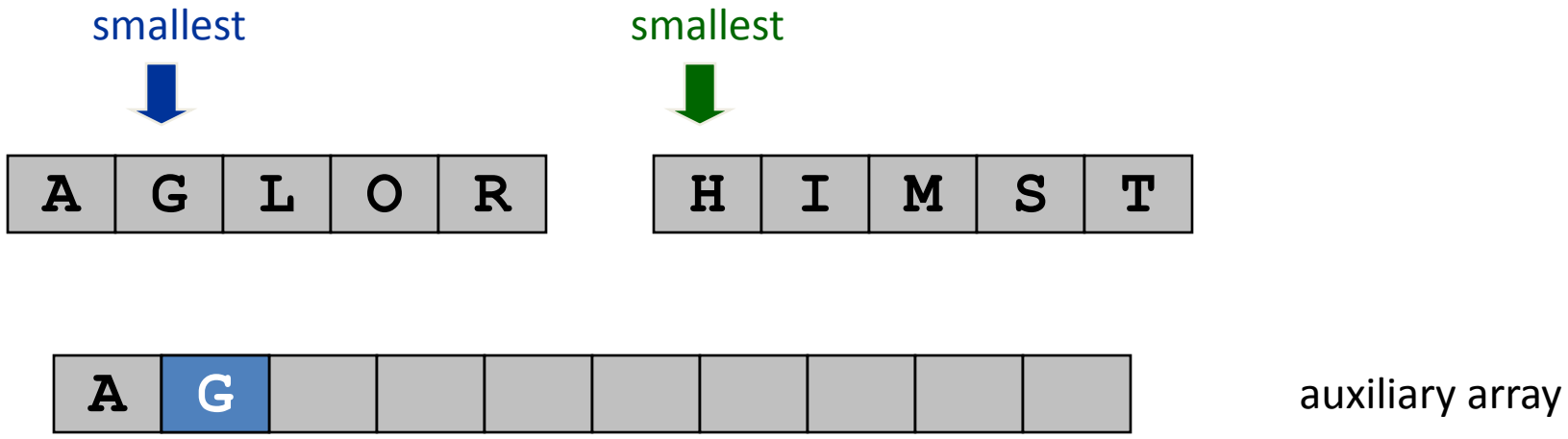
Merging

- Merge.
  - Keep track of smallest element in each sorted half.
  - Insert smallest of two elements into auxiliary array.
  - Repeat until done.



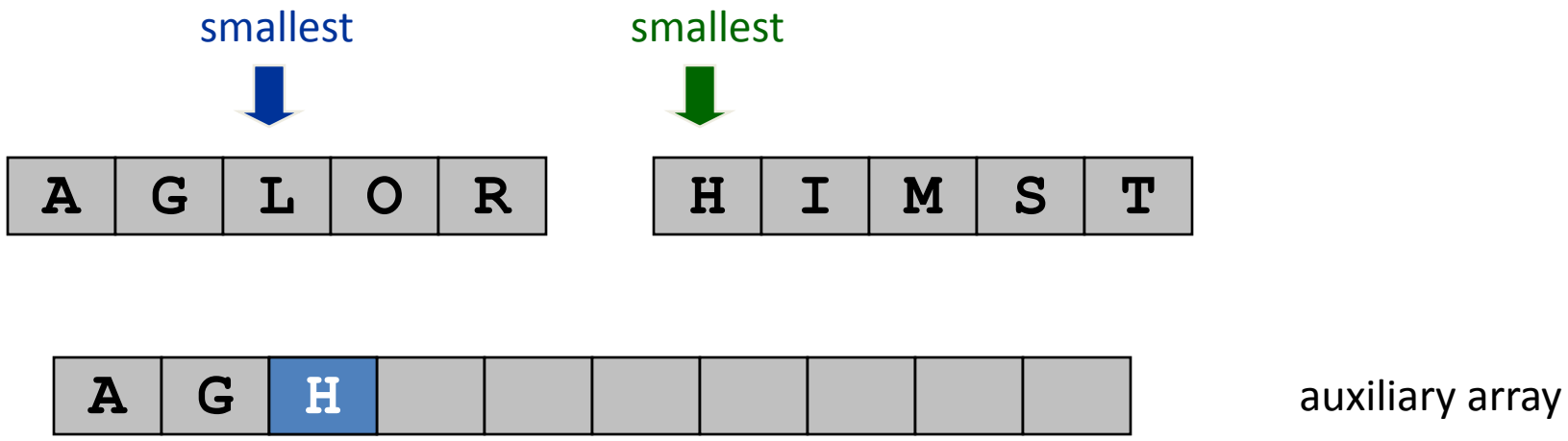
Merging

- Merge.
  - Keep track of smallest element in each sorted half.
  - Insert smallest of two elements into auxiliary array.
  - Repeat until done.



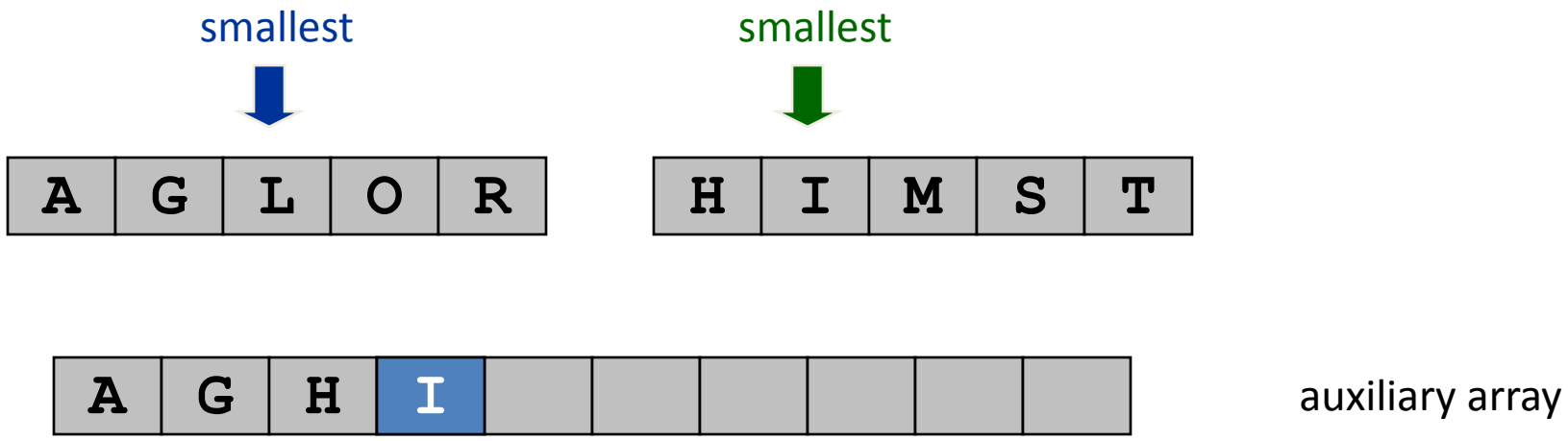
Merging

- Merge.
  - Keep track of smallest element in each sorted half.
  - Insert smallest of two elements into auxiliary array.
  - Repeat until done.



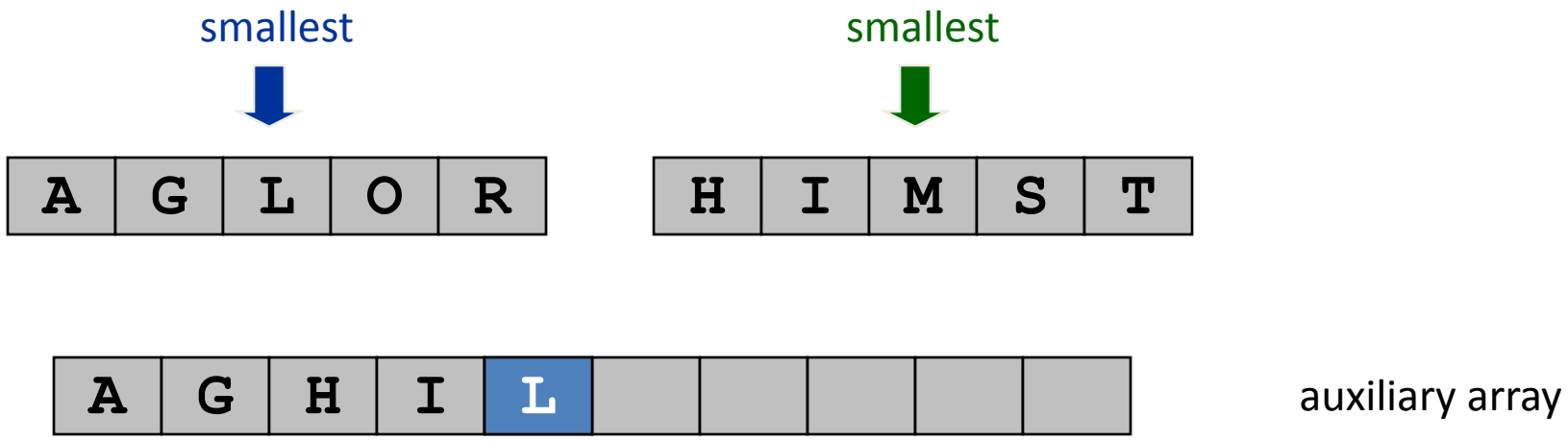
Merging

- Merge.
  - Keep track of smallest element in each sorted half.
  - Insert smallest of two elements into auxiliary array.
  - Repeat until done.



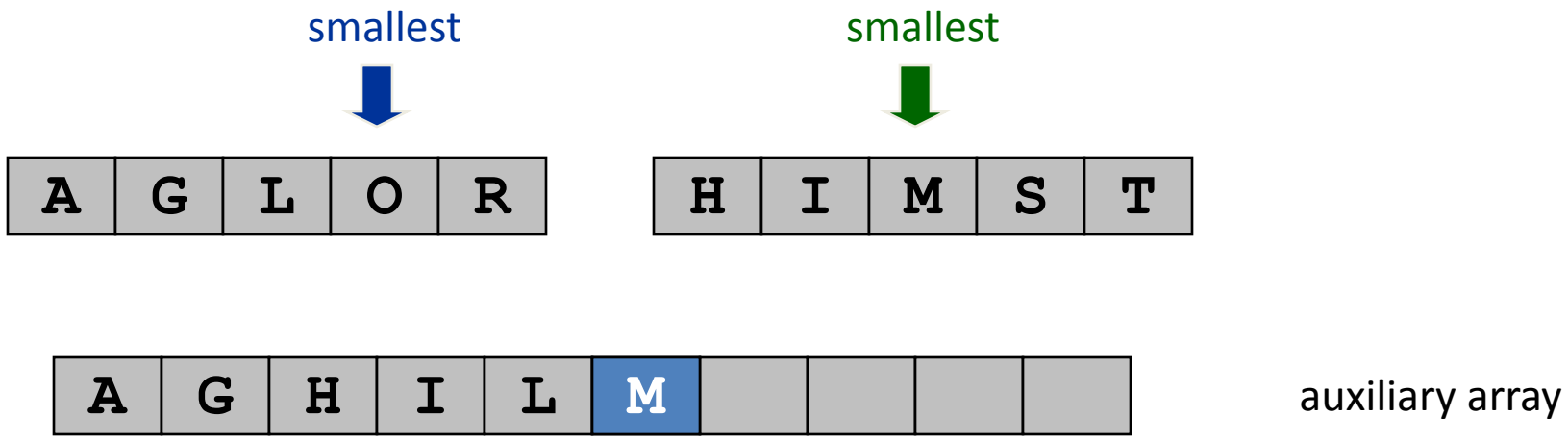
Merging

- Merge.
  - Keep track of smallest element in each sorted half.
  - Insert smallest of two elements into auxiliary array.
  - Repeat until done.



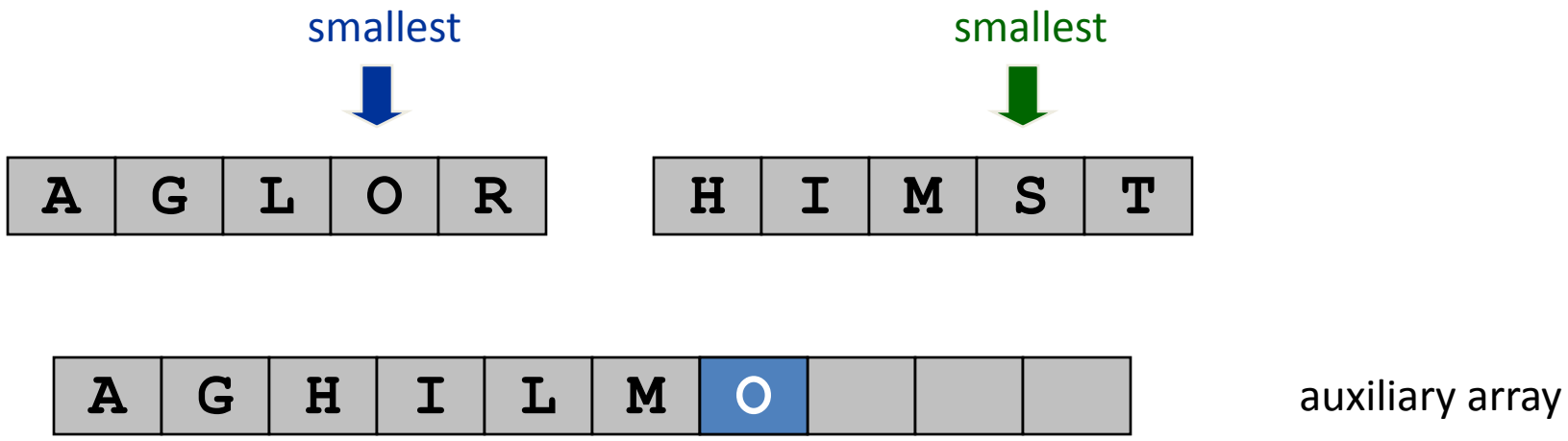
Merging

- Merge.
  - Keep track of smallest element in each sorted half.
  - Insert smallest of two elements into auxiliary array.
  - Repeat until done.



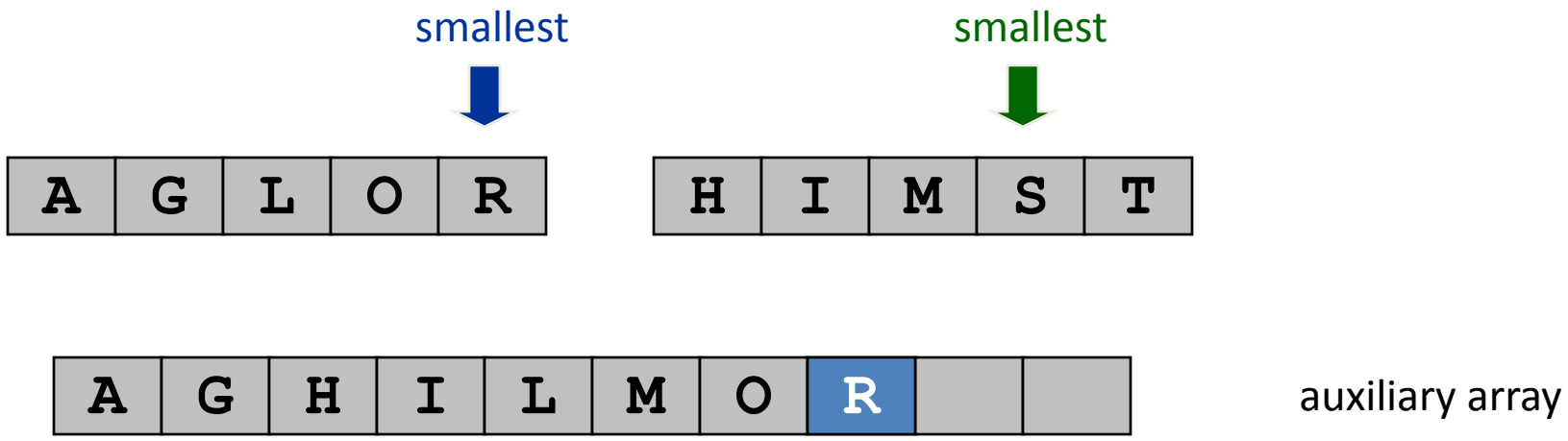
Merging

- Merge.
  - Keep track of smallest element in each sorted half.
  - Insert smallest of two elements into auxiliary array.
  - Repeat until done.



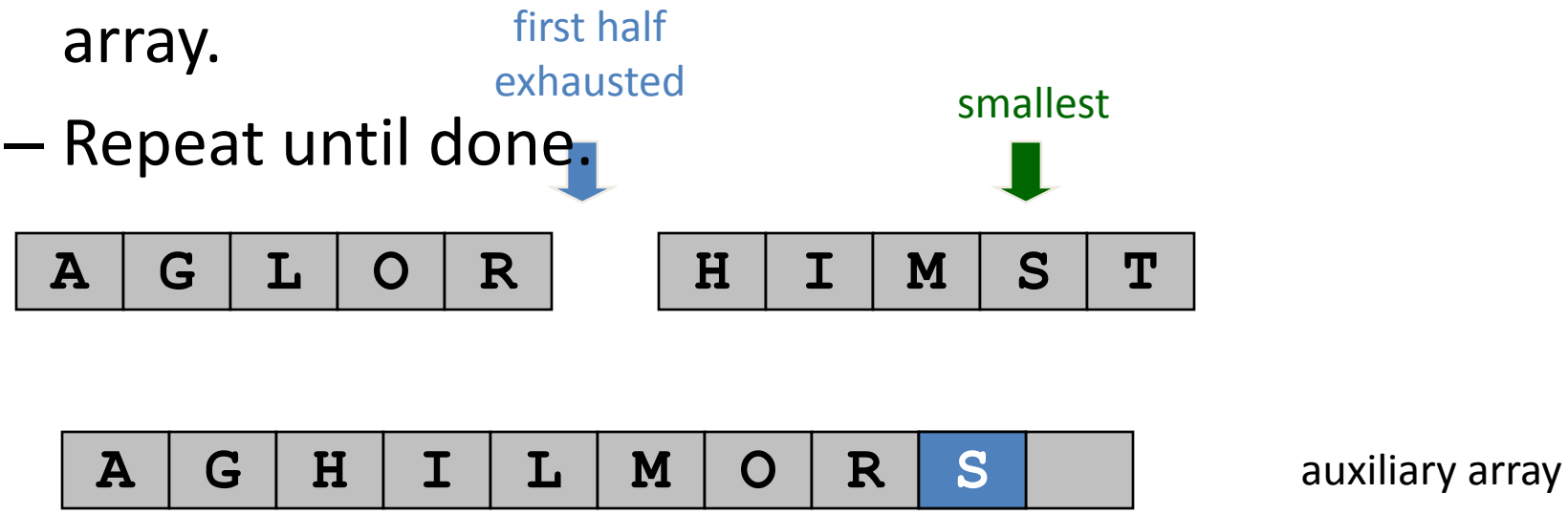
Merging

- Merge.
  - Keep track of smallest element in each sorted half.
  - Insert smallest of two elements into auxiliary array.
  - Repeat until done.



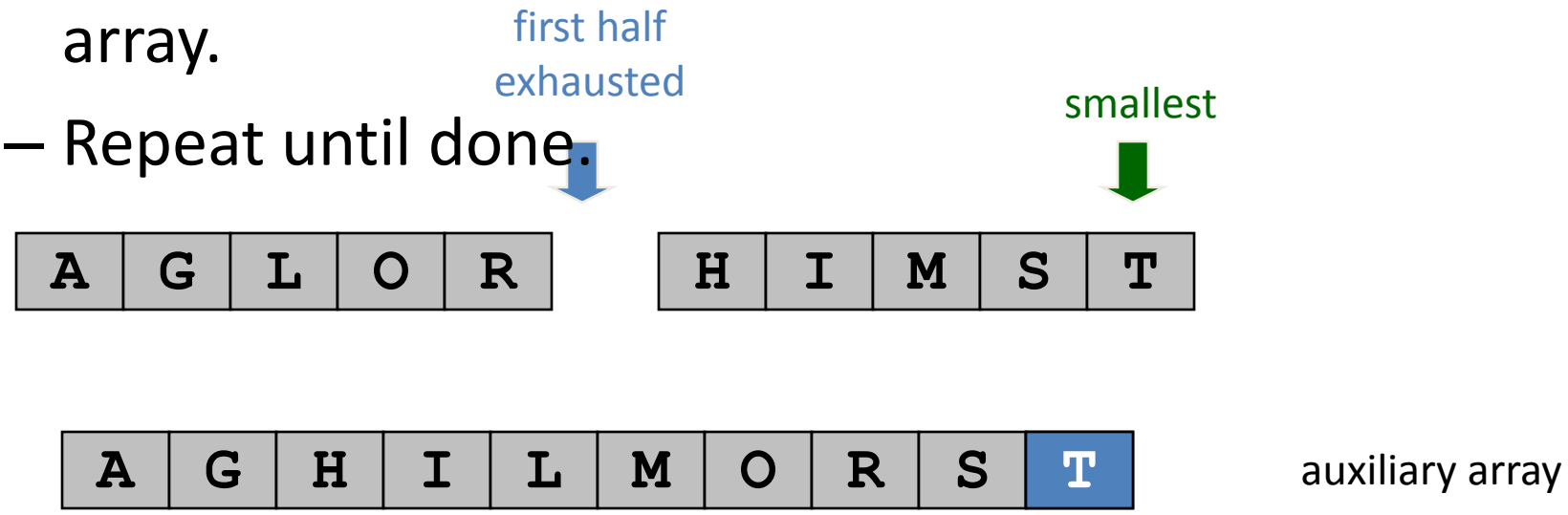
Merging

- Merge.
  - Keep track of smallest element in each sorted half.
  - Insert smallest of two elements into auxiliary array.
  - Repeat until done.



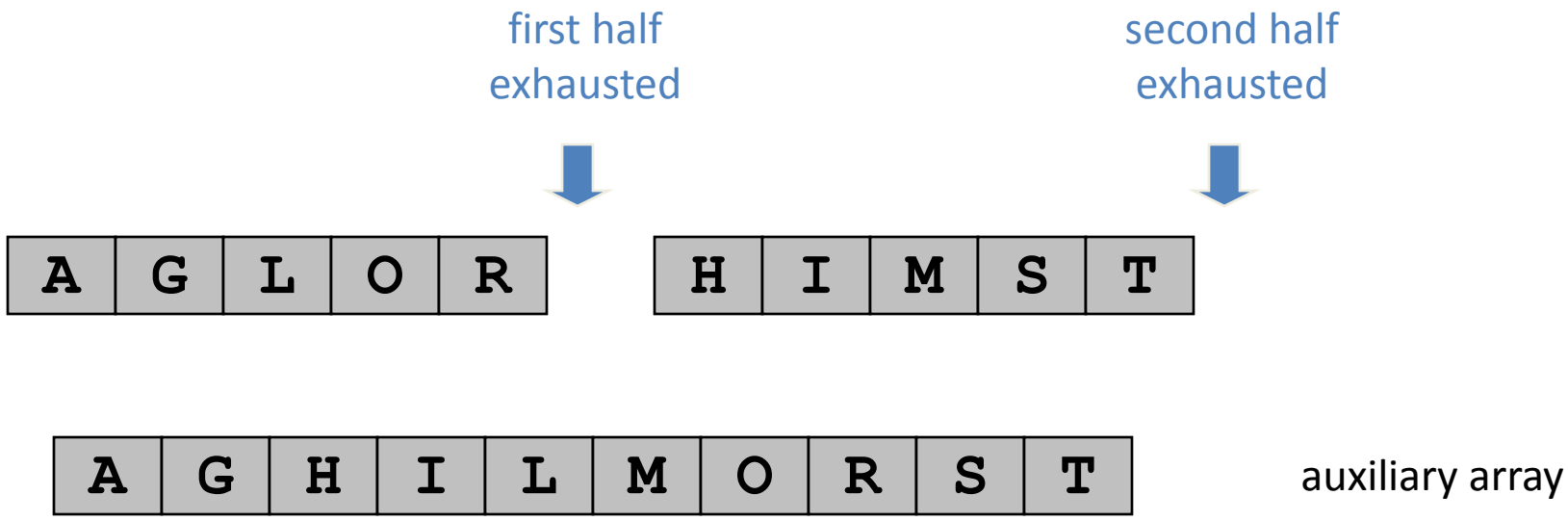
Merging

- Merge.
  - Keep track of smallest element in each sorted half.
  - Insert smallest of two elements into auxiliary array.
  - Repeat until done.

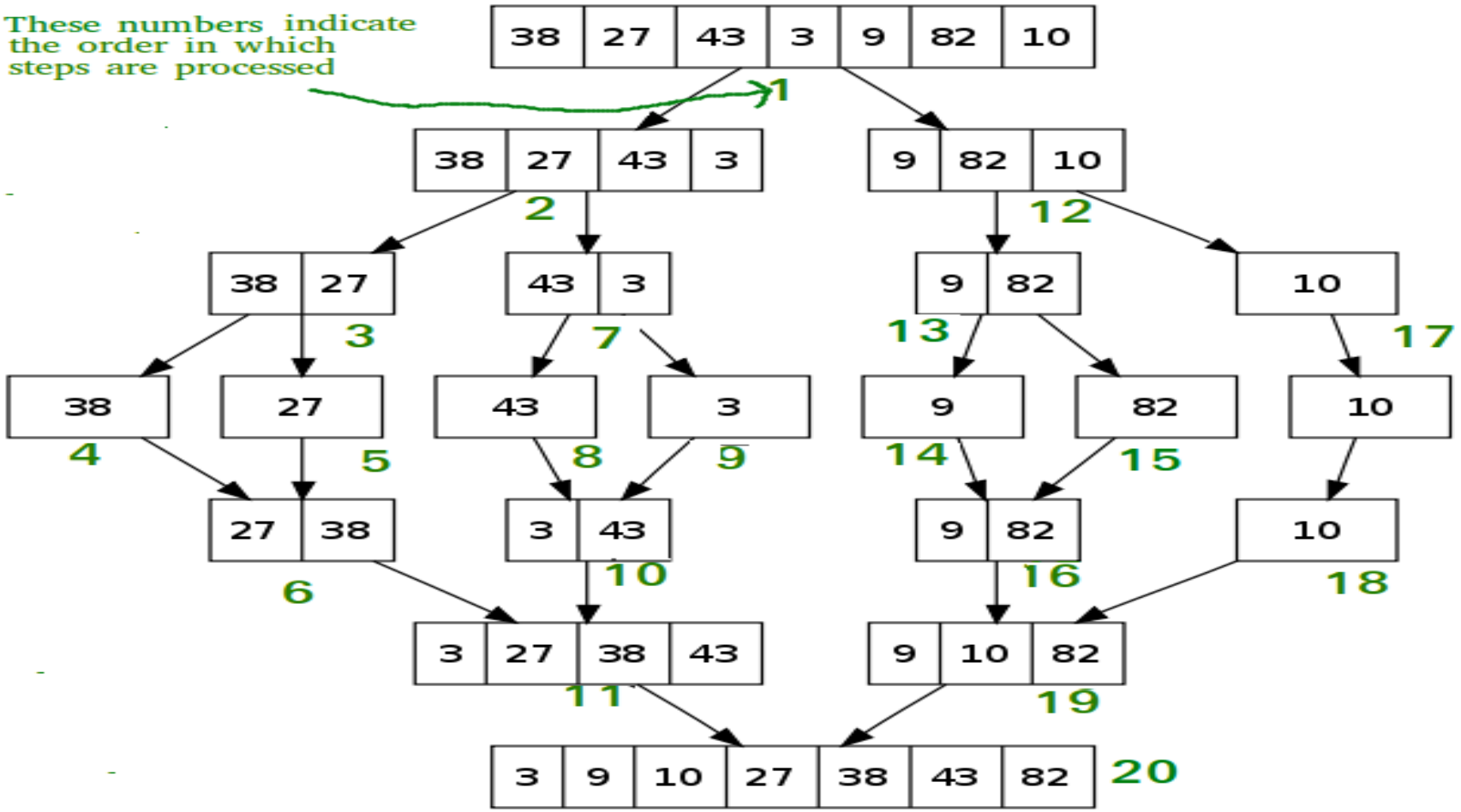


Merging

- Merge.
  - Keep track of smallest element in each sorted half.
  - Insert smallest of two elements into auxiliary array.
  - Repeat until done.



Ex2: Merge sort contd.. (Example)



Merge Sort is analysed using a useful Recurrence Relation

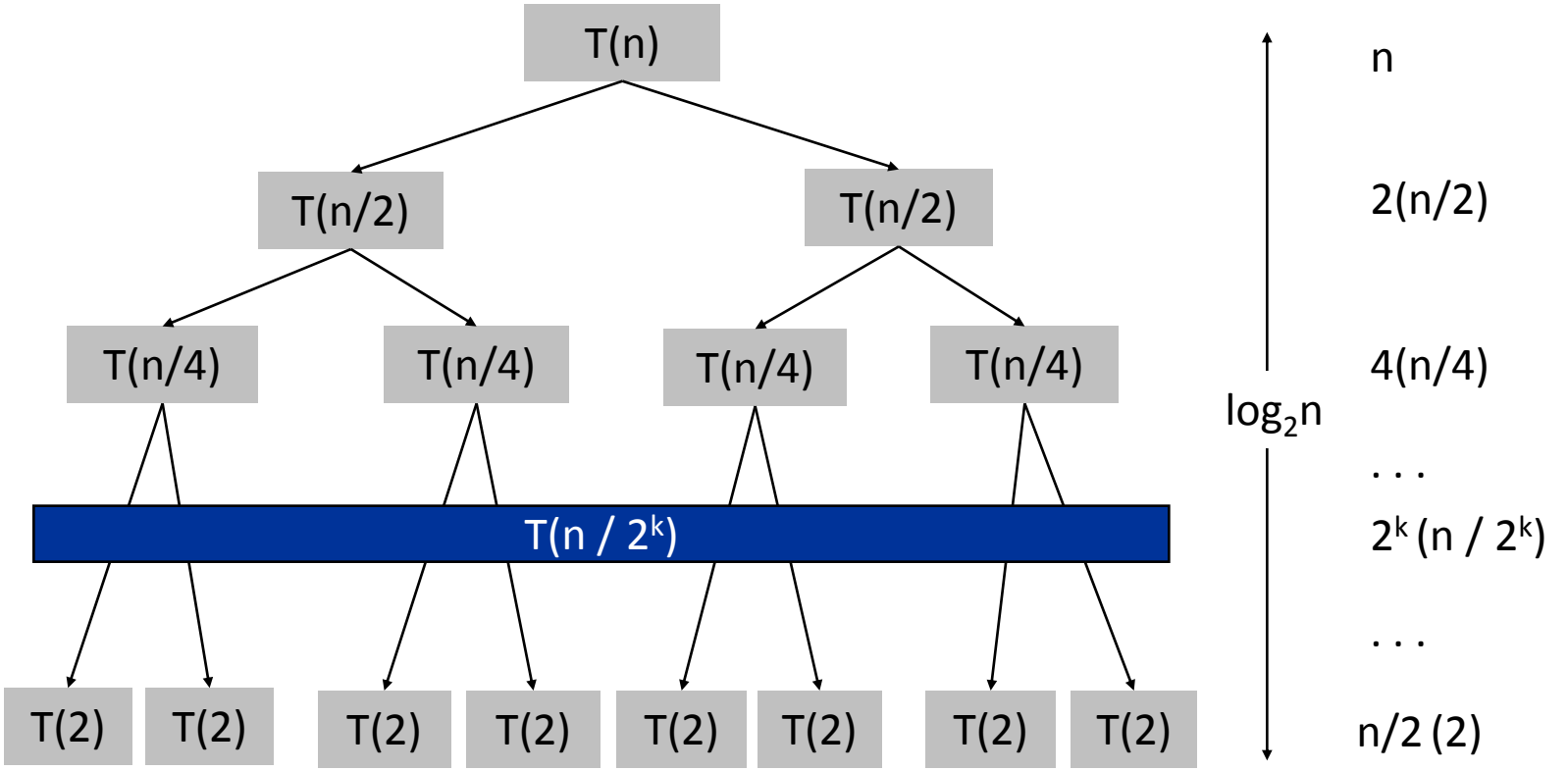
- Def. T(n) = number of comparisons to merge sort an input of size n.
- Merge sort recurrence.

$$T(n) \leq \begin{cases} 0 & \text{if } n = 1 \\ \underbrace{T(\lceil n/2 \rceil)}_{\text{solve left half}} + \underbrace{T(\lfloor n/2 \rfloor)}_{\text{solve right half}} + \underbrace{n}_{\text{merging}} & \text{otherwise} \end{cases}$$

- Solution. $T(n) = O(n \log_2 n)$.
- Assorted proofs. We describe several ways to prove this recurrence. Initially we assume n is a power of 2 and replace $\leq$ with =.

Merge Sort Analysis: Proof by Recursion Tree

$$T(n) = \begin{cases} 0 & \text{if } n = 1 \\ 2T(n/2) + n & \text{otherwise} \\ \text{sorting both halves} \quad \text{merging} \end{cases}$$



Analysis of Merge sort Recurrence

- Claim. If T(n) satisfies the following recurrence, then $T(n) \le n \lceil \lg n \rceil$.

$\lg n$ = $\log_2 n$

$$T(n) \le \begin{cases} 0 & \text{if } n = 1 \\ T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + n & \text{otherwise} \\ \text{solve left half} \quad \text{solve right half} \quad \text{merging} & \end{cases}$$

- Pf. (by induction on n)
  - Base case: n = 1.
  - Define n1 = $\lfloor n / 2 \rfloor$, n2 = $\lceil n / 2 \rceil$.
  - Induction step: assume true for 1, 2, ... , n–1.

$$\begin{aligned} T(n) &\le T(n_1) + T(n_2) + n \\ &\le n_1 \lceil \lg n_1 \rceil + n_2 \lceil \lg n_2 \rceil + n \\ &\le n_1 \lceil \lg n_2 \rceil + n_2 \lceil \lg n_2 \rceil + n \\ &= n \lceil \lg n_2 \rceil + n \\ &\le n(\lceil \lg n \rceil - 1) + n \\ &= n \lceil \lg n \rceil \end{aligned}$$

$$\begin{aligned} n_2 &= \lfloor n/2 \rfloor \\ &\le \lceil 2^{\lceil \lg n \rceil} / 2 \rceil \\ &= 2^{\lceil \lg n \rceil} / 2 \\ \Rightarrow \lg n_2 &\le \lceil \lg n \rceil - 1 \end{aligned}$$

Merge sort contd.. (Analysis)

- The running time of Merge-Sort as *T(n)*. Hence,

$$T(n) = \begin{cases} a & n = 1, a \text{ a constant} \\ 2T(n/2) + cn & n > 1, c \text{ a constant} \end{cases}$$

When n is a power of 2, $n = 2^k$,
we can solve above equation by successive substitutions as follows:

T{n) = 2(2T(n/4)+cn/2) + cn
= 4T(n/4) + 2cn
= 4(2T(n/8)+en/4) + 2cn

$= 2^kT(1) + kcn$
= an + cnlogn

It is easy to see that if $2^k < n < 2^{k+1,}$ then T{n) <= $T(2^{k+1})$.
Therefore T(n) = 0(nlogn)

Quick sort

- The principle of divide-and-conquer used in Quick sort for sorting numbers.
- Quick sort works by partitioning a given array using pivot element in the array ***A[p ... r]*** into two non-empty sub array

  A[p ...q] and ***A[q+1 ... r]*** such that every key in ***A[p ... q]*** is less than or equal to every key in ***A[q+1 ... r]***.
- Then, the two sub-arrays are sorted by recursive calls to Quick sort. The exact position of the partition depends on the given array and index ***q*** is computed as a part of the partitioning procedure.
- **Algorithm: Quick-Sort (A, p, r)**

```
if p < r then
    q Partition (A, p, r)
    Quick-Sort (A, p, q)
    Quick-Sort (A, q + r, r)
```

Quick sort contd..

- Note that to sort the entire array, the initial call should be ***Quick-Sort (A, 1, length[A])***
- As a first step, Quick Sort chooses one of the items in the array to be sorted as pivot. Then, the array is partitioned on either side of the pivot. Elements that are less than or equal to pivot will move towards the left, while the elements that are greater than or equal to pivot will move towards the right.
- Partitioning the Array

Partition (A, p, r)

```
x ← A[p] ;
i ← p-1 ;
j ← r+1 ;
while TRUE do
        Repeat j ← j - 1
            until A[j] ≤ x ;
        Repeat i← i+1
            until A[i] ≥ x ;
         if i < j then
            exchange A[i] ↔ A[j] ;
        else
            return  j ;
```

Quick sort contd..

- Advantages

✓ It is in-place since it uses only a small auxiliary stack.

✓ It requires only ***n (log n)*** time to sort **n** items.

✓ It has an extremely short inner loop.

✓ This algorithm has been subjected to a thorough mathematical analysis, a very precise statement can be made about performance issues.

- Disadvantages

✓ It is recursive. Especially, if recursion is not available, the implementation is extremely complicated.

✓ It requires quadratic (i.e., $O(n^2)$) time in the worst-case.

✓ It is fragile, i.e. a simple mistake in the implementation can go unnoticed and cause it to perform badly.

- Analysis

✓ The worst case complexity of Quick-Sort algorithm is $O(n^2)$. However using this technique, in average cases generally we get the output in $O(n \log n)$ time.

Quick Sort Analysis

- For an array, in which **partitioning** leads to unbalanced subarrays, to an extent where on the left side there are no elements, with all the elements greater than the **pivot**, hence on the right side. (Pivot element may be first element, or last element or middle element)
- And if keep on getting unbalanced subarrays, then the running time is the worst case, which is $O(n^2)$.
- Where as if **partitioning** leads to almost equal subarrays, then the running time is the best, with time complexity as **O(n*log n)**.

✓ Worst Case Time Complexity [Big-O]: **$O(n^2)$**

✓ Best Case Time Complexity [Big-omega]: Ω (n*log **n)**

✓ Average Time Complexity [Big-theta]: **Θ (n*log n)**

✓ Space Complexity: **O(n*log n)**

Quick Sort Analysis contd..

- As we know now, that if subarrays **partitioning** produced after partitioning are unbalanced, quick sort will take more time to finish. If someone knows that you pick the last index as **pivot** all the time, they can intentionally provide you with array which will result in worst-case running time for quick sort.
- To avoid this, you can pick random **pivot** element too. It won't make any difference in the algorithm, as all you need to do is, pick a random element from the array, swap it with element at the last index, make it the **pivot** and carry on with quick sort.

✓ Space required by quick sort is very less, only O(n*log n) additional space is required.

✓ Quick sort is not a stable sorting technique, so it might change the occurence of two similar elements in the list while sorting.

Karatsuba multiplication algorithm for multiplying large integers

- The **Karatsuba algorithm** is a fast multiplication algorithm that uses a **divide and conquer approach** to multiply two numbers. It was discovered by Anatoly Karatsuba in 1960 and published in 1962.
- This happens to be the first algorithm to demonstrate that multiplication can be performed at a lower complexity than O(N^2) which is by following the classical multiplication technique. Using this algorithm, multiplication of two n-digit numbers is reduced from O(N^2) to O(N^(log 3) that is O(N^1.585).

Multiplication of large integers using Karatsuba Algorithm Example 1

- Consider the following multiplication: 47*78

✓ x = 47 = 4*10+7

✓ Then $x_1 = 4$ and $x_2 = 7$

✓ y = 78 = 7*10+8

✓ Then $y_1 = 7$ and $y_2 = 8$

✓ $a = x_1 * y_1 = 4*7 = 28$

✓ $c = x_2 * y_2 = 7*8 = 56$

✓ $b = (x_1 + x_2)(y_1 + y_2) - a - c = 11*15 - 28 - 56 = 81$

- Substituting above values in
- $xy = a * B^{(2m)} + b * B^m + c$
- $= 28*10^2 + 81*10^1 + 56$

  $= 3666$

Multiplication of large integers using Karatsuba Algorithm Example 2

- Consider the following multiplication: 2345*5678

✓ $x = 2345 = 23*10^2+45$

✓ Then $x_1 = 23$ and $x_2 = 45$

✓ $y = 5678 = 56*10^2+78$

✓ Then $y_1 = 56$ and $y_2 = 78$

✓ $a = x_1 * y_1 = 23*56 = 1288$ (Apply Recursive approach using D&C on 23*56)

✓ $c = x_2 * y_2 = 45*78 = 3510$ (Apply Recursive approach using D&C on 45*78)

✓ $b = (x_1 + x_2)(y_1 + y_2) - a - c = (23+45)*(56+78) - 1288 - 3510$

$= 68 * 134 - 1288 - 3510$

(Apply Recursive approach on using D&C 68 * 134)

$= 4314$

- Substituting above values in
- $xy = a * B^{(2m)} + b * B^m + c$
- $= 1288 *10^4+(4314) *10^2 +3510$
- $= 13314910$

Multiplication of large integers using Karatsuba Algorithm

- Basically Karatsuba stated that if we have to multiply two n-digit numbers x and y, this can be done with the following operations, assuming that B is the base of m and m < n (for instance: m = n/2)
- First both numbers **x** and **y** can be represented as **x1,x2** and **y1,y2** with the following formula.

✓ $x = x1 * B^m + x2$

✓ $y = y1 * B^m + y2$

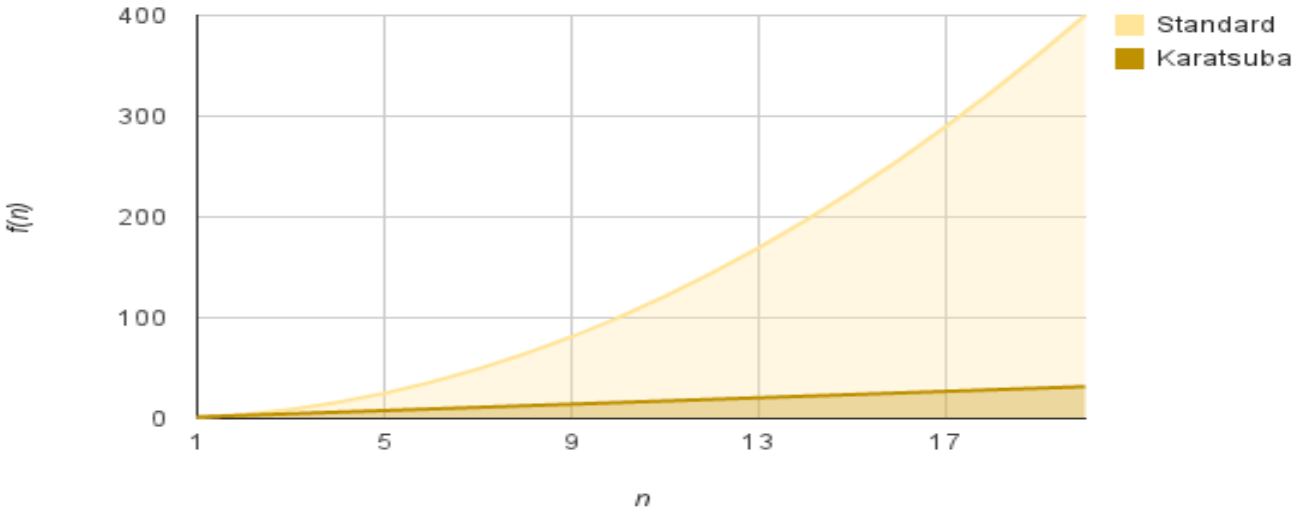
Multiplication of large integers using Karatsuba Algorithm contd…

- The product x X y becomes the following product:
- ✓ $xy = (x1 * B^m + x2) * (y1 * B^m + y2)$
- ✓ $=> xy = x1 * y1 * B^{(2m)} + x1 * y2 * B^m + x2 * y1 * B^m + x2 * y2$
- Observe that there are 4 sub-problems: X1 * Y1, X1 * Y2, X2 * Y1 and X2 * Y2
- With a clever insight, we can reduce this to 3 sub-problems and hence, the acceleration.
- ✓ Let $a = x1 * y1$,
- ✓ $b = x1 * y2 + x2 * y1$
- ✓ $c = x2 * y2$
- Finally, **x * y** becomes
- ✓ $xy = a * B^{(2m)} + b * B^m + c$
- That is why Karatsuba came up with the brilliant idea to calculate b with the following formula:
- ✓ $b = (x1 + x2)(y1 + y2) - a - c$

Karatsuba algorithm Time Complexity

- Assuming that we replace two of the multiplications with only one makes the program faster. The question is how fast. Karatsuba improves the multiplication process by replacing the initial complexity of $O(n^2)$ by $O(n^{(\log 3)})$, which as you can see on the diagram below is much faster for big n.

✓ $$T(n)=3*T(n/2)+O(n) = O(n^{(\log 3)}) = O(n^{1.58})$$



Application: The Karatsuba algorithm is very efficient in tasks that involve integer multiplication. It can also be useful for polynomial multiplications.

It is to be noted that faster multiplication algorithms do exist.

Strasson's Matrix multiplication

- Given two square matrices A and B of size n x n each, find their multiplication matrix.
- ***Traditional 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 $O(N^3)$.

Strasson's Matrix multiplication contd..

- ***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.

Strasson's Matrix multiplication contd..

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

A B 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

- In the above method, we do 8 multiplications for matrices of size N/2 x N/2 and 4 additions.

Strasson's Matrix multiplication contd..

- 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, time complexity of above method is $O(N^3)$ which is unfortunately same as the above Traditional method.
- In the above divide and conquer method, the main component for high time complexity is 8 recursive calls.

Strasson's Matrix multiplication contd..

- The idea of **Strassen's method** is to reduce the number of recursive calls to 7.
- Strassen's method is similar to above 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 above diagram.
- But in Strassen's method, the four sub-matrices of result are calculated using following formulae.

Strasson's Matrix multiplication contd..

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

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

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

A B 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

Strasson's Matrix multiplication contd..

- **Time Complexity Analysis 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 $O(N^{2.8074})$

Strasson's Matrix multiplication contd..

- 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 Traditional method works better.

✓ For Sparse matrices, there are better methods especially designed for them.

✓ The submatrices in recursion take extra space.

✓ Because of the limited precision of computer arithmetic on noninteger values, larger errors accumulate in Strassen's algorithm than in Naive Method

Recall of Unit II

- Disjoint sets and operations:
- ✓ Union and find algorithms
- Spanning trees.
- Divide and Conquer methodology: General method
- Applications of Divide and Conquer methodology:
- ✓ Binary search
- ✓ Quick sort
- ✓ Merge sort
- ✓ Multiplication of large integers using Karatsuba algorithm
- ✓ Strassen's matrix multiplication.