Divide and Conquer Strategy

- Divide the problem into a number of smaller subproblems.
- Solve the smaller subproblems.
- Combine the solution of the smaller subproblems to obtain the solution of the original problem.

Problem: Given a set of n numbers, S, develop an algorithm to find the maximum and the minimum numbers in the set. For the sake of simplicity we assume n is a power of 2

Problem: Given a set of n numbers, S, develop an algorithm to find the maximum and the minimum numbers in the set. For the sake of simplicity we assume n is a power of 2

- Algorithm MaxMin1
  - Step1: Using the algorithm FindMax, compute the maximum number, MaxNum, in the set S.
  - Step2: Modify the algorithm FindMax to FindMin and use it to compute the minimum number in the set S.
- Basic Operation: Comparison between numbers
  - How many basic operations are performed?
  - Answer: (n-1) + (n-1) = 2n -2

Second algorithm to find the maximum and minimum number in the set S. For sake of simplicity we assume n is a power of 2

Algorithm FindMaxMin(S)

```
Begin
 1. Split S into two sets S1 and S2 of equal size.
 2. FindMaxMin(S1); Suppose MAX1 and MIN1 are the
    maximum and minimum of S1.
 3. FindMaxMin(S2); Suppose MAX2 and MIN2 are the
    maximum and minimum of S2.
 4. return (MAX= max(MAX1,MAX2), MIN= min(MIN1,
    MIN2)).
End
```

Complexity of the Algorithm FindMaxMin

- Basic Operations: Comparisons between numbers
- Suppose, that the number of basic operations needed to find the maximum and the minimum of the set S is T(n)
- Then, the number of basic operations needed to find the maximum and the minimum of the set S_1 is T(n/2)
- And the number of basic operations needed to find the maximum and the minimum of the set S_2 is T(n/2)
- Once the maximums and the minimums of the sets S_1 and S_2 are computed, two additional comparisons are needed

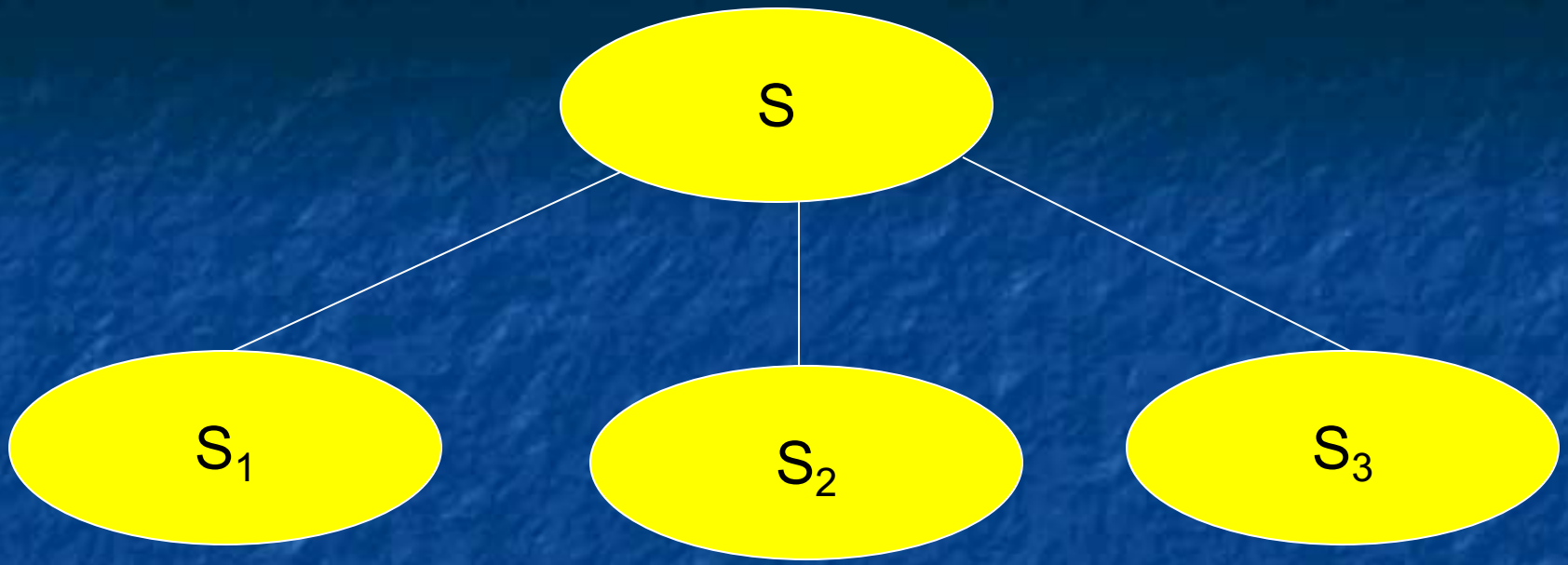
Complexity of the Algorithm FindMaxMin

- Basic Operations: Comparisons between numbers
- Suppose, that the number of basic operations needed to find the maximum and the minimum of the set S is T(n)
- Therefore, $T(n) = 2T(n/2) + 2$ for $n > 2$ and $T(2) = 1$
- $T(n) = ?$

Given a sequence of n numbers, the following algorithm sorts the sequence in ascending order. For sake of simplicity we assume that n is a power of 2.

Algorithm Sort(S)

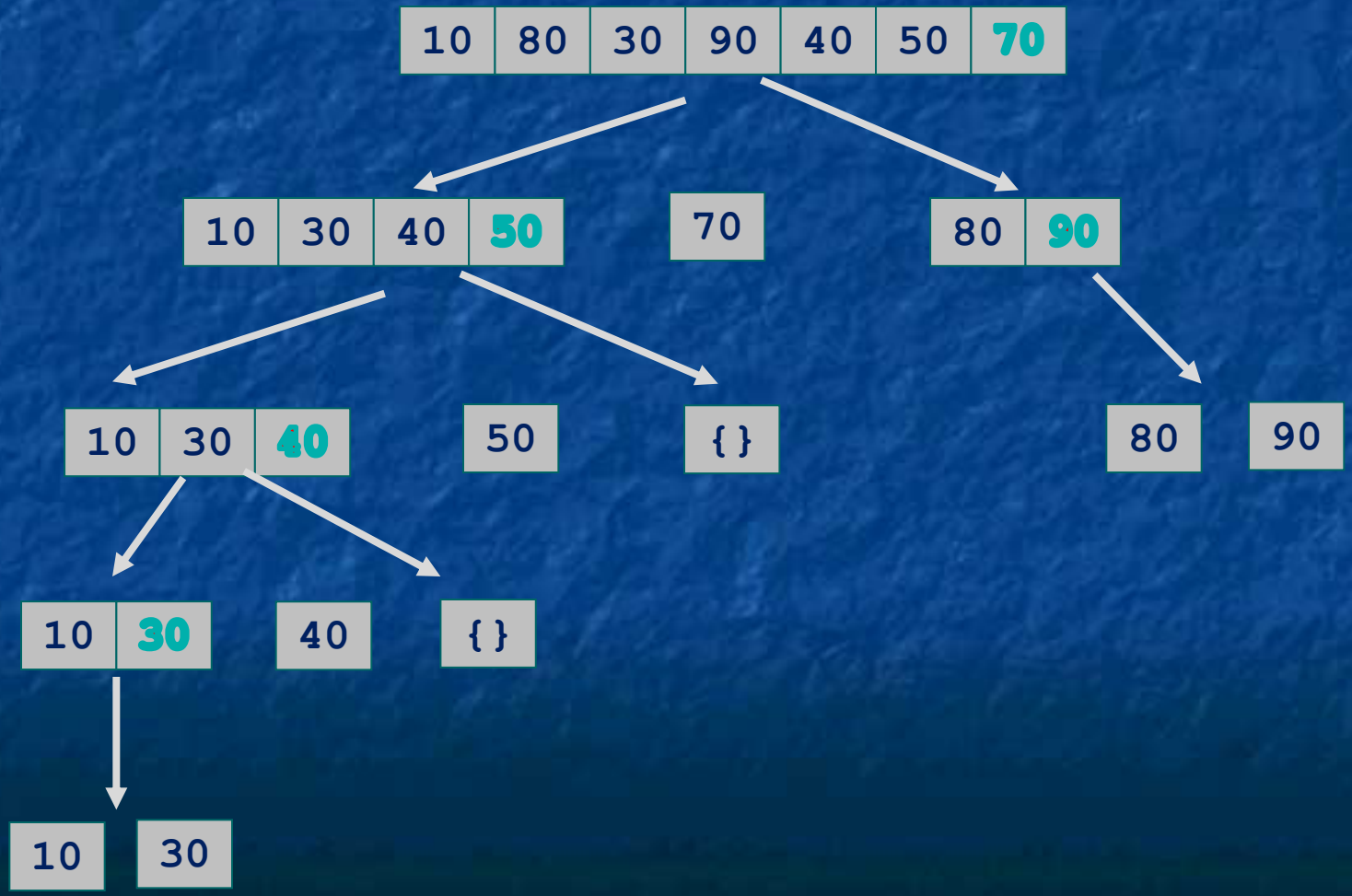
```
if |S| = 2 then compare the two numbers and return
   (min,max)
else
 begin
   1. Pick an arbitrary element s_k of the    sequence
  S.
   2. Divide S into parts S_1, S_2, S_3 such that the
  elements of S_1, S_2,and S_3 are less than, equal to
  and greater than s_k respectively.
   3. return (Sort(S_1),S_2,Sort(S_3))
 end
```



- Size of S = n
- Size of S_1
  - Smallest = 0
  - Largest = n-1
- Size of S_3
  - Smallest = 0
  - Largest = n-1

Quicksort Example

- Assume the last element is always the pivot.



- Given a sequence of n numbers, the following algorithm sorts the sequence in ascending order. For the sake of simplicity we assume that n is a power of 2.

Algorithm MergeSort(S)

```
if |S| =1 then
   return (S);
else
   Divide S into two sequences S1 and
  S2 of equal size;
   return (merge (MergeSort(S1),
                  MergeSort(S2));
```

Mergesort

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

Jon von Neumann (1945)

A	L	G	O	R	I	T	H	M	S

A	L	G	O	R

I	T	H	M	S

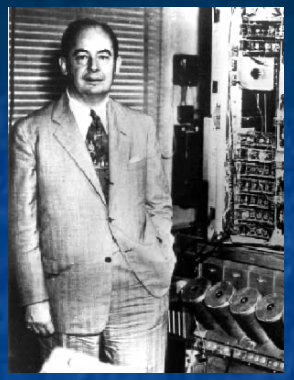
A	G	L	O	R

H	I	M	S	T

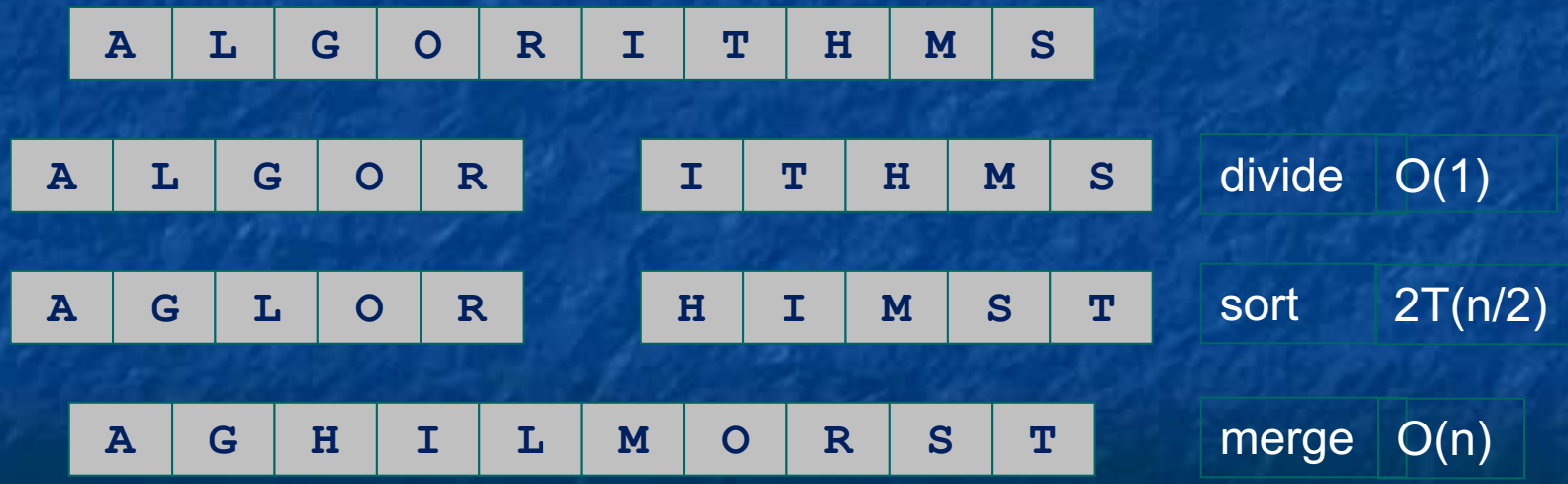
A	G	H	I	L	M	O	R	S	T

Mergesort

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



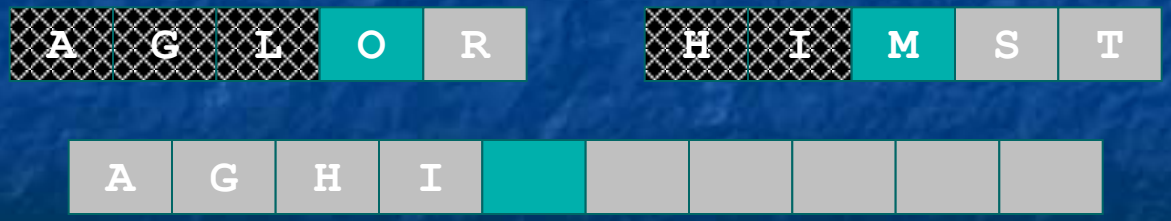
Jon von Neumann (1945)



Merging

- Merging: Combine two pre-sorted lists into a sorted whole.

- How to merge efficiently?
  - Linear number of comparisons.
  - Use temporary array.



A Useful Recurrence Relation

- Def. T(n) = number of comparisons to mergesort an input of size n.

- Mergesort recurrence.

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

A Useful Recurrence Relation

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

- Solution. T(n) = O(n $\log_2$ n).

Proof

- There are several ways to prove this recurrence. Initially we assume n is a power of 2 and let $T(n)=2T(n/2)+n$.
- By master theorem, it is easy to see that $n^{\log 2}=n$, therefore, $T(n)= O(f(n)\log n)=O(n\log n)$

Proof by iterative method

- We can expand the equation as follow:
- $T(n)=2T(n/2)+n$

  $=2(2T(n/4)+n/2)+n=2^2T(n/4)+2n$

  $=2^2(2T(n/8)+n/4)+2n=2^3T(n/8)+3n$

  $=\ldots$

  $= 2^iT(n/2^i)+ in$

  $=\ldots$

  $=2^(\log n)T(n/n)+\log n *n=O(n\log n)$

  $=n+n+\ldots+n= n \log n$

Recurrence Relations and Divide and Conquer Technique

- There are three components of the Divide and Conquer technique
  - Divide the problem into sub-problems
  - Solve the sub-problems
  - Combine the solutions of the sub-problems to obtain the solution of the original problem
- Total time taken by a Divide and Conquer algorithm will have contribution from each of the three components
  - Time to divide the problem into sub-problem
  - Time to solve the sub-problems
    - How many sub-problems?
    - What is a size of the sub-problems?
  - Time to combine the solutions of the sub-problems

Recurrence Relations and Divide and Conquer Technique

- Total time taken by a Divide and Conquer algorithm will have contribution from each of the three components
  - Time to divide the problem into sub-problem
    - It will be some function of the problem size n, Let's say $d_1(n)$
  - Time to solve the sub-problems
    - How many sub-problems? Let's say 'a' sub-problems
    - What is a size of the sub-problems? Let's say 'n/b', where b is a constant
  - Time to combine the solutions of the sub-problems
    - It will be some function of the problem size n, Let's say $d_2(n)$
- If T(n) is the time taken by the algorithm to solve a problem of size n then
  - $T(n) = aT(n/b) + d_1(n) + d_2(n) = aT(n/b) + d(n)$,

    where $d(n) = d_1(n) + d_2(n)$

Recurrence Relations and Divide and Conquer Technique

- If T(n) is the time taken by the algorithm to solve a problem of size n then
  - $T(n) = aT(n/b) + d_1(n) + d_2(n) = aT(n/b) + d(n)$,
  - where
    - a: the number of sub-problems
    - b: n/b is the size of the sub-problem
    - d(n) = amount of computational time needed to divide the problem to sub-problems and to re-combine the solution of the sub-problems to obtain the solution of the original problem

Algorithm ***Binary Search:*** Given a table of records whose keys are in increasing order $K_1<K_2<\ldots<K_n$, this algorithm searches for a given key K

S1. [Initialize] Set l=1, u=n.

S2. [Get midpoint] if u<l, the algorithm terminates unsuccessfully. Otherwise, i=(l+u)/2, the approximate midpoint of the relevant table area.

S3. [Compare] If $K<K_i$, go to S4; if $K>K_i$, go to S5; and if $K=K_i$, the algorithm terminates successfully.

S4. [Adjust u] Set u=i-1, and return to S2.

S5. [Adjust l] Set l=i+1, and return to S2.

Binary search analysis

- If T(n) is the time taken by the binary search algorithm to solve the given problem of size n then
- T(n)=?

Binary search analysis

- If T(n) is the time taken by the binary search algorithm to solve the given problem of size n then
- $T(n)=T(n/2)+O(1) =O(?)$

Binary search analysis

- If T(n) is the time taken by the binary search algorithm to solve the given problem of size n then

- $T(n)=T(n/2)+O(1) = O(\log n)$