These types of algorithms are primarily used in situations where we already have some algorithm but wish to speed it up. I will refer to an existing algorithm, (for want of an appropriate terminology), as "Brute-Force" algorithm. The idea is to use this only for small sized problems but to use Divide-and-Conquer algorithm for larger size instances. In this process, we use solutions of smaller sized instances (sub-instances, possibly of the original instance) to get to solution of larger size instances. (Hence the name Divide-and-Conquer)

Thus several "decisions" to make:

1. What is "small" size? What do we do for small sized instances?
2. How do we divide (the input) of larger size instance?
   (a) Are these "pieces" disjoint?
   (b) How many "pieces"?
   (c) Are these "consecutive" or some other way?
3) After getting the "Solutions" of smaller pieces, how do we "merge" them to get a Solution of larger pieces?

4) Look upon this "process" top-down, i.e. in a recursive manner. [Most of you are used to looking "bottom-up"—Do Not do this for divide-and-Conquer algorithms. It is much easier the other way.

Now we will do several examples (some from the book and others not)

Example 1.
Problem Statement: MAX-MIN (A[1..n])


Desired output: An index j such that

\[ A[j] \leq A[i] \quad i \neq j \] 

(i.e. find maximum)

And index k such that

\[ A[k] \leq A[i] \quad i \neq k \] 

(i.e. minimum)
Brute Force Algorithm

**Step 1**: Find maximum: Comparing elements in the order 1, 2, ..., n; always retain larger and compare with the next.

W.C. Complexity: \((n-1)\) Comparisons

**Step 2**: Find minimum of the elements other than one found in Step 1.

W.C. Complexity: \((n-2)\) Comparisons

**Total**: \(2n-3\) Comparisons.

Q: Can we do better?

---

Divide-and-Conquer Algorithm I

**Small**: \(n \leq 2\): One comparison will determine both max and min.

(Our "Brute-Force" algorithm)

**Divide**: We break the input and solve the MAX-MIN problem on pieces by calling MAX-MIN Alg (d-and-e type unless \(n\) is small - in which case "Brute-Force" algorithm is used). We need to decide how many pieces and how are they formed.
Please note: "Subproblems" are identical to the main problem, except for the size of the input. We divide into \( n \) parts based on consecutive indices of the original array. This leads to:

\[
\text{MAX-MIN} \left( A \left[ 1, \ldots, \left\lfloor \frac{n}{2} \right\rfloor \right] \right) + \\
\text{MAX-MIN} \left( A \left[ \left\lceil \frac{n}{2} \right\rceil, \ldots, n \right] \right)
\]

Now comes the "merge" part in which from the answers to subproblems, we determine answers to their union.

\[
\text{MAX} = \max \left[ \text{MAX} \left( A \left[ 1, \ldots, \left\lfloor \frac{n}{2} \right\rfloor \right] \right), \text{MAX} \left( A \left[ \left\lceil \frac{n}{2} \right\rceil, \ldots, n \right] \right) \right]
\]

\[
\text{MIN} = \min \left[ \text{MIN} \left( A \left[ 1, \ldots, \left\lfloor \frac{n}{2} \right\rfloor \right] \right), \text{MIN} \left( A \left[ \left\lceil \frac{n}{2} \right\rceil, \ldots, n \right] \right) \right]
\]

Thus, it takes two comparisons to find overall MAX-MIN if we already have computed MAX-MIN for each subproblem.

Now Recurrence Relation for this algorithm:

\[
t(n) = t \left( \left\lfloor \frac{n}{2} \right\rfloor \right) + t \left( \left\lceil \frac{n}{2} \right\rceil \right) + 2
\]

\[
t(2) = 1
\]

\[
t(n) = \# of comparisons for \text{MAX-MIN} \left( A [1, \ldots, n] \right)
\]
If we use Master Theorem to solve this, we get \( t(n) = \Theta(n) \).

But such a result does not tell us whether this algorithm is any better than computing \( \text{MAX} \) first & then \( \text{MIN} \) of the remaining array. We need a better analysis of the recurrence relation. So we do this using Iteration Method on special case, with \( n = 2^k \), \( k \) int first and then using Substitution Method to establish it for all \( n \). [This latter part is left to you as an exercise.]

Since \( n = 2^k \), \( \lceil \frac{n}{2} \rceil = \lceil \frac{n}{2^2} \rceil = \frac{n}{2} \) for all subproblem. Hence, our recurrence is

\[
t(n) = 2t\left(\frac{n}{2}\right) + 2
\]

\[
= 2 + 2 \left[2 + 2t\left(\frac{n}{2^2}\right)\right]
\]

\[
= 2 + 2^2 + 2^2 \cdot t\left(\frac{n}{2^2}\right)
\]

\[
= 2 + 2^2 + \cdots + 2^{k-1} + 2^{k-1} \cdot t(2)
\]

\[
= 2 \left[1 + 2 + \cdots + 2^{k-2}\right] + 2^{k-1}
\]

\[
= 2 \cdot \frac{2^{k-1} - 1}{2-1} + 2^{k-1} = 2 \cdot \frac{2^{k-1} - 1}{2-1} + 2^{k-1}
\]

\[
= n + \frac{n}{2} - 2 = \frac{3n}{2} - 2
\]

Ex:
Show \( t(n) = \left\lceil \frac{3}{2} n - 2 \right\rceil \) for all \( n \) by Substitution Method.
\[ \frac{3}{2}n - 2 < 2n - 3 \text{ by about } 25\% \]

\[ \therefore \text{ Hence this is a better algorithm.} \]

Here is another algorithm (easier to show complete):

Divide into pairs & compare each pair.

\[
\begin{align*}
\lceil \frac{n}{2} \rceil \begin{cases} 
\end{cases}
\end{align*}
\]

if \( n \) is odd \( \times \)

Overall MAX is in the green set \( \to \lceil \frac{n}{2} \rceil - 1 \)

\( \therefore \text{ MIN is in the red set } \to \lceil \frac{n}{2} \rceil - 1 \)

Total work:

\[
\begin{align*}
\lceil \frac{n}{2} \rceil + \lceil \frac{n}{2} \rceil - 1 + \lceil \frac{n}{2} \rceil - 1 \\
= \lceil \frac{3}{2}n - 2 \rceil
\end{align*}
\]

Are these two algorithm "really" different?

Now we will consider a slightly different problem. Finding MIN - SMIN (A[1]...n])
(Second Min)
**MIN-SMIN**

Small \( n \leq 2 \): One comparison is needed
("Brute force" algorithm)

Finding MIN: \( n-1 \) comparisons

S.MIN: \( n-2 \)

Total: \( 2n-3 \)

Not using D-and-C alg

---

**D-and-C alg.**

**MIN-SMIN** \((A[1 \ldots n])\)

Small: \( n \leq 2 \): One comparison.

**Divide:** Two roughly equal parts

\[ \text{MIN-SMIN} (A[1 \ldots \lceil \frac{n}{2} \rceil]) \rightarrow \text{MIN1, SMIN1} \]

\[ \text{MIN-SMIN} (A(\lceil \frac{n}{2} \rceil+1, \ldots n]) \rightarrow \text{MIN2, SMIN2} \]

Getting solution to the larger problem from answers to two smaller problems.

We have four quantities: Hence there are almost \( 2 \) comparisons among them.

How many of these are needed?

Are these sufficient?
"Merge" pseudo-code:

\[
\begin{align*}
&\text{if } \text{MIN}_1 \leq \text{MIN}_2 \\
&\quad \text{then } \text{MIN} \leftarrow \text{MIN}_1 \\
&\quad \text{if } \text{SMIN}_1 \leq \text{MIN}_2 \\
&\quad \quad \text{then } \text{SMIN} \leftarrow \text{SMIN}_1 \\
&\quad \quad \text{else } \text{SMIN} \leftarrow \text{MIN}_2 \\
&\quad \text{else } \text{MIN} \leftarrow \text{MIN}_2 \\
&\quad \text{if } \text{SMIN}_2 \leq \text{MIN}_1 \\
&\quad \quad \text{then } \text{SMIN} \leftarrow \text{SMIN}_2 \\
&\quad \quad \text{else } \text{SMIN} \leftarrow \text{MIN}_1 \\
&\end{align*}
\]

W.C.: # of comparisons = 2.

\[
\therefore t(n) = t\left(\lceil \frac{n}{2} \rceil \right) + t\left(\lfloor \frac{n}{2} \rfloor \right) + 2
\]

\[t(2) = 1\]

Same equation as in MIN-MAX and hence

Same solution:

\[t(n) = \lceil \frac{3}{2} n - 2 \rceil\]

Q: Is the following statement true or false?

Either both MAX-MIN, MIN-SMIN have better algorithms or neither has better algorithms.

We will come back to this later.
Permutation Networks:
(Divide-and-Conquer in Device Construction)

Basic Part

Switch \( \leq \text{on} \)

Two position Switch-box

Inputs

\( \begin{array}{c}
1 \\
2 \\
\end{array} \) \( \begin{array}{c}
\text{off} \\
\text{off} \\
\end{array} \)

Outputs

\( \begin{array}{c}
1 \\
2 \\
\end{array} \) \( \begin{array}{c}
\text{on} \\
\text{on} \\
\end{array} \)

We want to "create" \( n \times n \) boxes using (possibly many) \( 2 \times 2 \) boxes. We do this for \( n = 2^k \), \( k \) integer \( \geq 0 \).

\( n = 4, n = 8 \) Cases [These will illustrate general case.]

\( n = 4 \)

\[ 4 \times 4 \text{ box} \]

8x8 on next page
\[ t(n) = \# \text{ of } 2 \times 2 \text{ boxes used in } n \times n \text{ box} \]

\[ = 2t\left(\frac{n}{2}\right) + n \]

\[ \therefore t(n) = \Theta(n \log n) \text{ by Master Theorem Case 2}. \]

Now we need to show that any desired output permutation can be obtained from given input permutation (conveniently taken to be \(1, 2, \ldots, n\)) by appropriately setting switches.

Such a network is called a valid network.

We now show that above network is valid.
We need a bit of Graph Theory.

\[ \text{Graph: } G = [V, E] \]

- \( V \): Set of vertices (nodes, points)
- \( E \): Set of edges (arcs, links) \( \subseteq V \times V \).

For this example, this graph is undirected.

- We do not allow structures with edges of the form \([u, u]\)
- \( \bigcirc \): Self loops, prohibited
- Parallel edges are allowed.

Special graphs: Bipartite graphs

- \( V = V_1 \cup V_2 \), \( V_1 \cap V_2 = \emptyset \)
- \( (u, v) \in E \), \( u \in V_1 \Rightarrow v \in V_2 \)
- \( u \in V_2 \Rightarrow u \in V_1 \)

Permitted edges in green
Prohibited ones in red
An example of a Bipartite graph.

Tree

(V_1, V_2, V_3, V_4, V_5)

(The process may be started at any node.)
(There may be more than one way of doing this)

Given an undirected graph, how to check if it is bipartite?

Without loss, assume that G is connected—i.e. for every pair of nodes there is a path between the two. \[ G \text{ is in One piece}. \]

Step 1: Find a spanning tree in G (includes all nodes, if G) Since G is connected, it exists; can be found by BFS or DFS, for example.

Step 2: Label vertices using spanning tree.

As shown above.

Step 3: For each edge \((u,v)\) in \( T \), check if it is of the form \( u \in V_1, v \in V_2 \) then \( G \) is not bipartite.
What does all this got to do with our problem?

We are going to create a graph called Forbidden pairs Diagram as follows.

**Input:** Input permutation and output permutation.

\[ V: \text{Inputs: } \{1, 2, \ldots, n\} \]

\[ E: \text{if } (i, j) \in \text{Same in put box,} \]

\[ \begin{array}{c}
  (i) \quad (j) \\
  \text{Input forbidden pairs}
\end{array} \]

(Since they can not go into the same middle box) So if \( i \in N_1 \), \( j \in N_2 \).

\[ \text{if } (i, j) \in \text{Same out put box,} \]

\[ \begin{array}{c}
  i \quad j \\
  \end{array} \]

For our example

Forbiden Pairs Diagram $G$.

$G$ is bipartite. Hence we can partition the nodes.
We have set all the "outer" switches. Setting switches for \( N_1 \) and \( N_2 \) is done similarly. Since all these forbidden graphs are bipartite, there will be no difficulty.

Hence this network is valid. It also happens to be easily scalable.

Now to show any other feasible system will also use \( \Theta(n \log n) \) 2x2 boxes, i.e. our design is optimal.
Suppose some (other) feasible system uses $k$ $2 \times 2$ boxes. Hence, in such a system, there are $k$ switches (one per $2 \times 2$ box) that can be set on or off. Hence, the system has a maximum possible $2^k$ configurations.

Given a configuration of the system, we can get at most one permutation as output. [Maybe there is some conflict in some and no output is obtained; several configurations might result in same output] hence the max # of output from such a system $\leq 2^k$.

But we need the possibility of generating all $(n!)$ permutations if the system is valid, for a given input permutation. $\therefore 2^k \geq n!$

$\therefore k \geq \lg (n!) = \Omega(n \lg n)$

Since $\lg (n!) = \Theta(n \lg n)$ (shown in Order Notation Section)

$\therefore$ Hence our system is optimal.