Design Technique - Divide and Conquer

The divide –and- conquer approach involves three steps at each level of the recursion

- **Divide** the problem into a number of subproblems
- **Conquer** the subproblems by solving them recursively.
- **Combine** the solutions to the subproblems into the solution for the original problem

Design Technique - Divide and Conquer

- The number of smaller instances into which the input is divided is k.
- For an input of size n, Let D(n) be the number of steps done by divide, and let C(n) be the number of steps done by combine.
- Then the general form of the recurrence equation that describes the amount of workdone by the algorithm is

  $T(n)=D(n) + \sum_{i=1}^{k} T(size(I_i)) + C(n)$, for n>smallsize

Design Technique - Divide and Conquer

Divide and conquer skeleton

```
Solve(I)
    n=size(I);
    if (n<=smallsize)
    solution=directlysolve(I);
    else
    divide I into $I_1,I_2,\ldots\ldots.I_k$;
    for each i ε {1,2….k}
    Si=solve($I_i$);
    solution=combine($S_1,S_2,\ldots.S_k$);
return solution;
```

Design Technique – Greedy Method

- A greedy algorithm always makes the choice that looks best at the moment
- That is, it makes a locally optimal choice in the hope that this choice will lead to a globally optimal solution
- Greedy algorithms do not always yield optimal solutions, but for many problems they do.

Design Technique – Greedy Method

- The greedy method suggests that one can devise an algorithm which works in stages, considering one input at a time.
- At each stage, a decision is made regarding whether or not a particular input is in an optimal solution
- If the inclusion of the next input into the partially constructed optimal solution will result in an infeasible solution, then this input is not added to the partial solution.
- The selection procedure itself is based on some optimization measure

Design Technique – Greedy Method

Greedy method control abstraction

```
Procedure Greedy(A, n) //A(1:n) contains the n inputs//
Solution=  ø //initialize the solution to empty//
for i=1 to n do
    x=SELECT(A)
    if FEASIBLE (solution, x)
    then solution= UNION(solution, x)
    endif
repeat
return(solution)
end
```

Design Technique – Greedy Method

Greedy method control abstraction

- The function SELECT selects an input from A, removes it and assigns its value to x.
- FEASIBLE is a boolean valued function which determines if x can be included into the solution vector.
- UNION actually combines x with solution

Design Technique – Dynamic Programming

- Dynamic programming is applicable when the subproblems are not independent, ie when subproblems share subsubproblems.
- Dynamic programming algorithm solves every subsubproblem just once and then saves its answer in a table, thereby avoiding the work of recomputing the answer every time the subsubproblem is encountered.
- Dynamic programming is typically applied to optimization problems
- Example: Optimal matrix multiplication

Design Technique – Dynamic Programming

- The development of a dynamic programming algorithm can be broken into a sequence of four steps
  - Characterize the structure of an optimal solution
  - Recursively define the value of an optimal solution
  - Compute the value of an optimal solution in a bottom-up fashion
  - Construct an optimal solution from computed information

Design Technique – Dynamic Programming

- The essential difference between the greedy method and dynamic programming is that in the greedy method only one decision sequence is ever generated. In dynamic programming, many decision sequences may be generated.

Design Technique – Backtracking

- The principal idea in backtracking is to construct solutions one component at a time and evaluate such partially constructed candidates as follows
  - If a partially constructed solution can be developed further without violating the problem’s constraints, it is done by taking the first remaining legitimate option for the next component.
  - If there is no legitimate option for the next component, no alternatives for any remaining component need to be considered, in this case, the algorithm backtracks to replace the last component of the partially constructed solution with its next option

Design Technique – Backtracking

- It is convenient to implement this kind of processing by constructing a tree of choices being made, called **the state-space tree**.
- Its root represents an initial state before the search for a solution begins.
- The nodes of the first level in the tree represent the choices made for the first component of a solution, the nodes of the second level represent the choices for the second component, and so on.

Design Technique – Backtracking

- A node in a state-space tree is said to be **promising** if it corresponds to a partially constructed solution that may still lead to a complete solution; otherwise, it is called **nonpromising**.

- Leaves represent either nonpromising dead ends or complete solutions found by the algorithm.

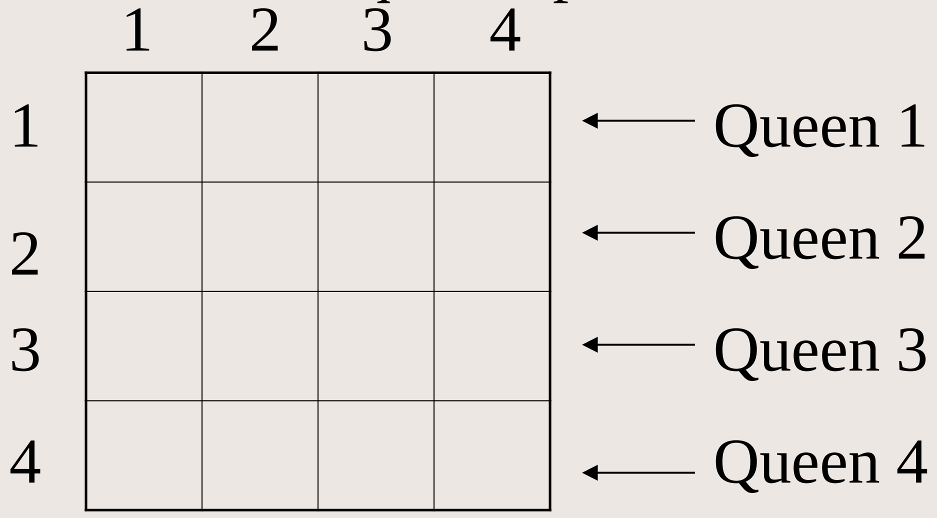
Design Technique – Backtracking

- If the current node is promising, its child is generated by adding the first remaining legitimate option for the next component of a solution, and the processing moves to this child.
- If the current node turns out to be nonpromising, the algorithm backtracks to the node’s parent to consider the next possible option for its last component; if there is no such option, it backtracks one more level up the tree and so on.
- Finally, if the algorithm reaches a complete solution to the problem, it either stops or backtracks to continue searching for other possible solutions

Design Technique – Backtracking

Example: n-Queens Problem

- The problem is to place n queens on an n-by-n chessboard so that no two queens attack each other by being in the same row or in the same column or on the same diagonal. Let us consider the four-queens problems.



Board for the four queens problem

Design Technique – Backtracking

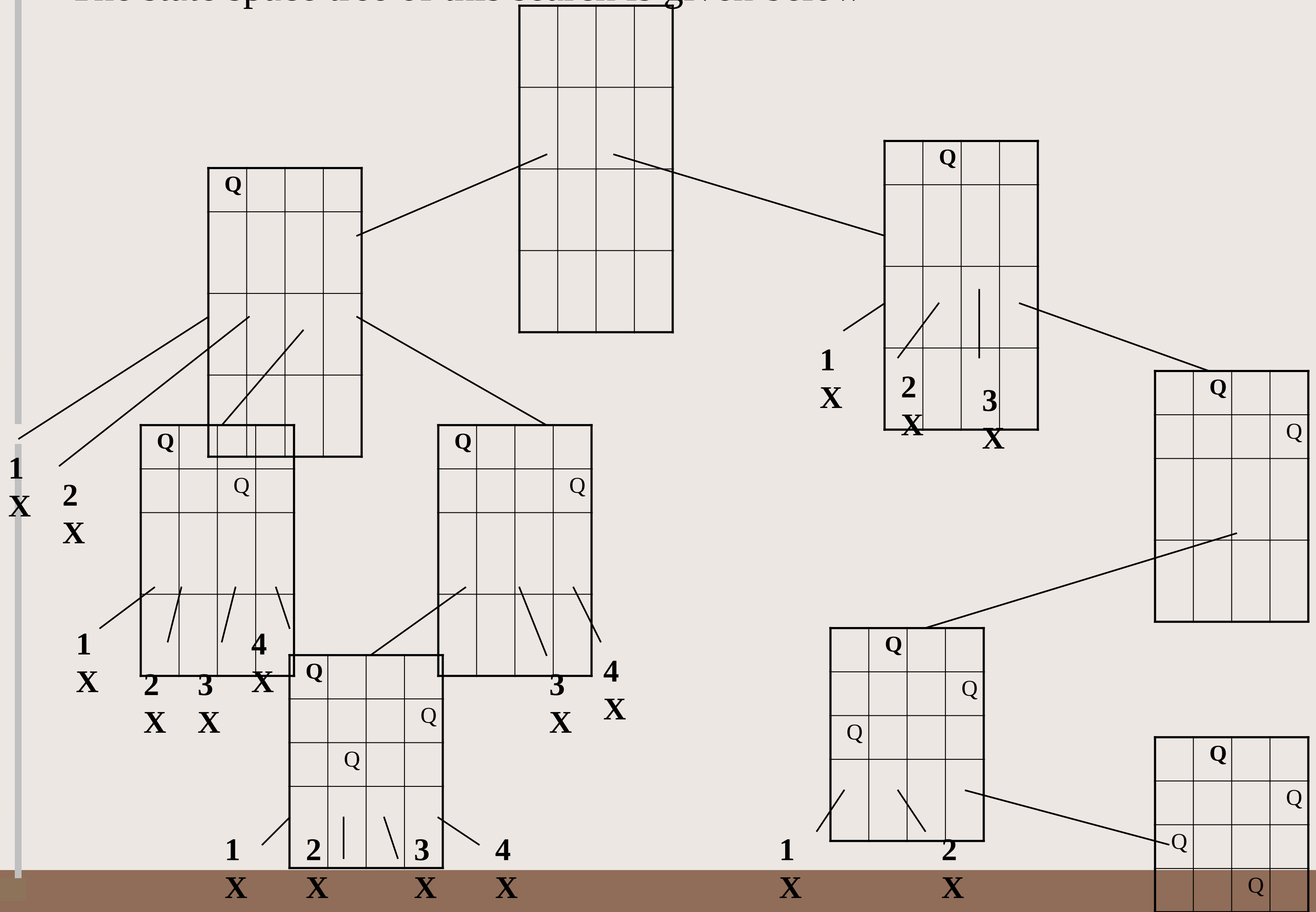
- We start with the empty board and then place queen 1 in the first possible position of its row, which is in column 1 of row 1.
- Then we place queen 2, after trying unsuccessfully columns 1 and 2, in the first acceptable position for it, which is square (2,3), the square in row 2 and column 3.
- This proves to be a dead end because there is no acceptable position for queen 3.

Design Technique – Backtracking

- So, the algorithm backtracks and puts queen 2 in the next possible position at (2,4).
- Then queen 3 is placed at (3,2), which proves to be another dead end.
- The algorithm then backtracks all the way to queen 1 and moves it to (1,2). Queen 2 then goes to (2,4), queen 3 to (3,1)and queen 4 to (4,3), which is a solution to the problem

Design Technique – Backtracking

- The state space tree of this search is given below



Design Technique – Backtracking

- An output of a backtracking algorithm can be thought of as an n-tuple(x_1,x_2..x_n) where each coordinate x_i is an element of some finite linearly ordered set S_i

```
Algorithm **Backtrack(X[1..i])**
//Gives a template of a generic backtracking algorithm
//Input:X[1..i] specifies first i promising components of a solution
//Output: All the tuples representing the problem's solutions
If X[1..i] is a solution writeX[1..i]
else
  for each element $x \in S_{i+1}$ consistent with X[1..i] and the constraints do
     X[i+1]=x
     Backtrack(X[1..i+1])
```

Thank You