STANDARD ALGORITHM DESIGN TECHNIQUES

By: Tamal Chakraborty

Analysis & Design of Algorithms

- There are a few standard techniques for designing algorithms and analyzing their performance.
- Here we will look at a few such techniques, such as:

1. Divide & Conquer
2. Dynamic Programming
3. Greedy Method
4. Backtracking
5. Branch & Bound

Divide & Conquer

Divide & Conquer

- The Divide & Conquer approach breaks down the problem into multiple smaller sub-problems, solves the sub-problems recursively, then combines the solutions of the sub-problems to create a solution for the original problem.
- The steps involved are:
  1. Divide the problem into number of sub-problems
  2. Conquer the sub-problems by solving them recursively
  3. Combine the solution of the sub-problems to solve the original problem

Divide & Conquer : Merge Sort

- We need to sort a list of n numbers.
- The merge sort algorithm uses divide & conquer strategy to solve this problem in the following way:
- The steps involved are:
  1. **Divide** the n element sequence into 2 sub-sequences of n/2 elements each
  2. **Conquer** sort the sub-sequences recursively
  3. **Combine** merge the two sorted sub-sequences to form the end result

Divide & Conquer: Merge Sort

```
void mergesort(int a[ ], int l, int r)
{
        if (l < r)
        {
                int m = (l + r) / 2;          // divide
                mergesort(a, l, m);           // conquer 1st sub-sequence
                mergesort(a, m+1, r);         // conquer 2nd sub-sequence
                merge(a, l, m, r);            // combine
        }
}
```

Divide & Conquer: Merge Sort

```
void merge(int a[ ], int l, int m, int r)
{
        int n1 = m - l + 1;
        int n2 = r - m;
        int L[n1 + 2];                          // create arrays L & R
        int R[n2 + 2];
        for (int i = 1; i <= n1; i++)
        {
                L[i] = a[l + i - 1];            // initialize L with elements a[l] to a[m]
        }
        for (int j = 1; j <= n2; j++)
        {
                R[j] = a[m + j];                // initialize R with elements a[m+1] to a[r]
        }
        L[n1 + 1] = R[n2 + 1] = MAXINT;                 // #define MAXINT = 65536
        int p = 1;
        int q = 1;
        for (int k = l; k <= r; k++)
                a[k] = (L[p] <= R[q]) ? L[p++] : R[q++];
}
```

Divide & Conquer : Merge Sort

- If a = {5, 12, 17, 6} Function calls are in the following order
- mergesort(a, 1, 4)
  - mergesort(a, 1, 2)
    - mergesort(a, 1, 1)
    - mergesort(a, 2, 2)
    - merge(a, 1, 1, 2)
    - a = 5 12 17 6
  - mergesort(a, 3, 4)
    - mergesort(a, 3, 3)
    - mergesort(a, 4, 4)
    - merge(a, 3, 3, 4)
    - a = 5 12 6 17
  - merge(a, 1, 2, 4)
  - a = 5 6 12 17
- a = 5 6 12 17

```
void mergesort(int a[ ], int l, int r)
{
        if (l < r)
        {
                int m = (l + r) / 2;
                mergesort(a, l, m);
                mergesort(a, m+1, r);
                merge(a, l, m, r);
        }
}
```

Divide & Conquer : Merge Sort Analysis

- Let T(n) be the running time for input size n
- When n = 1, the problem can be solved in constant time
  - $T(n) = \Theta(1)$, when $n = 1$
- When n > 1 we divide the problem into 2 sub-problems, each of size n/2, which contributes to 2T(n/2) running time
- Merging an n elements takes $\Theta(n)$ time
  - $T(n) = 2T(n/2) + \Theta(n)$, when $n > 1$
- To estimate the running time of merge sort for an n element sequence we have to solve this recurrence

Divide & Conquer : Merge Sort Analysis

- $T(n) = 2T(n/2) + \Theta(n)$
- Is of the form $T(n) = a.T(n/b) + f(n)$, $a = 2$ & $b = 2$
- $\log_b(a) = 1$
- $n^{\log_b(a)} = n^1 = n$
- $f(n) = \Theta(n) = \Theta(n^{\log_b(a)})$

- Hence, by case 2 of the master method
- **$T(n) = \Theta(n.\lg(n))$**

Quick Sort

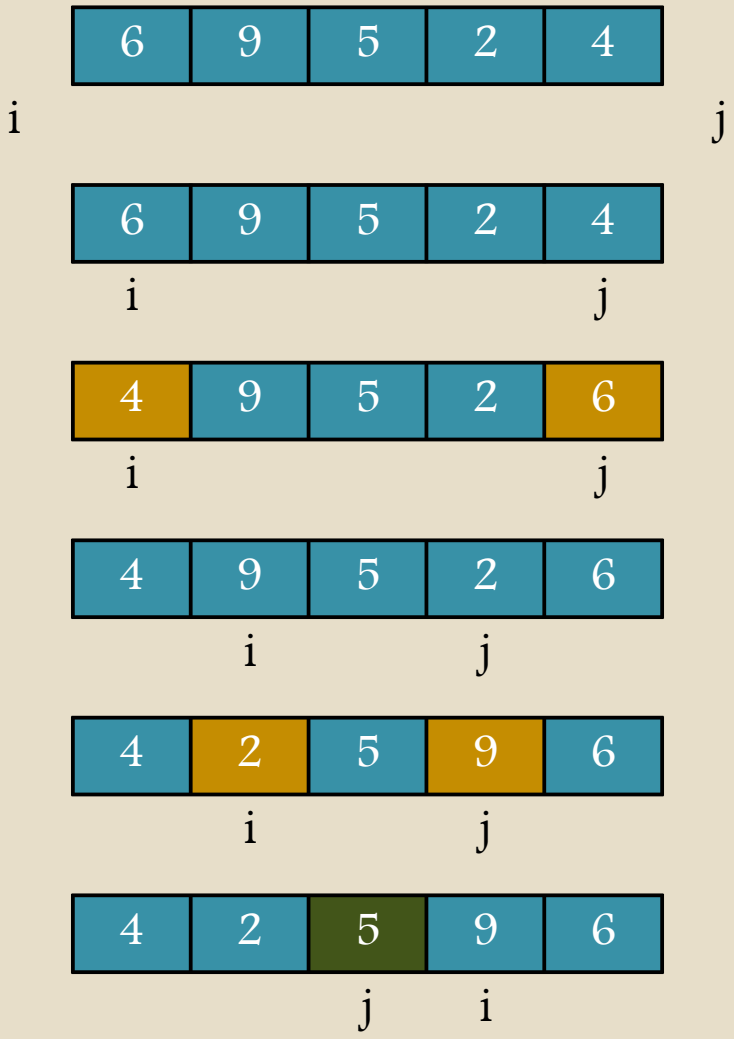
- Quick sort, like merge sort is based on divide and conquer technique for sorting an array a[l … r]
- The steps involved are:
  1. **Divide** the array a[l … r] is partitioned into two sub-components a[l … p] and a[p + 1 … r], such that every element of a[l … p] is less than equal to every element of a[p + 1 … r]
  2. **Conquer** The sub-arrays a[l … p] and a[p + 1 … r] are sorted by recursive calls to quicksort
  3. **Combine** Since the sub-arrays are sorted in place no overhead of combining them is required

Quick Sort

```
void qsort(int a[], int l, int r)
{
   if (l < r)
   {
        int p = partition(a, l, r);
        qsort(a, l, p);
        qsort(a, p + 1, r);
   }
}
```

Quick Sort

```
int partition(int a[], int p, int q)
{
    int x = a[p];
    int i = p - 1;
    int j = q + 1;
    while (1) {
            do {
                    i++;
            } while (a[i] < x);
            do {
                    j--;
            } while (a[j] > x);
            if (i < j) swap(a[i], a[j]);
            else return j;
    }
}
```



Quick Sort: Worst Case Analysis

- The worst case for quick sort is when partition function produces one sub-array with (n – 1) elements and another sub-array with 1 element. If this unbalanced partitioning happens at very step of the algorithm we call it the worst case quick sort behavior.
- Since partitioning takes $\Theta(n)$ time and $T(1) = \Theta(1)$, the recurrence for the worst case running time is
- $T(n) = T(n-1) + \Theta(n)$

$$= T(n-2) + \Theta(n-1) + \Theta(n)$$
$$= \dots$$
$$= T(1) + \dots + \Theta(n-1) + \Theta(n)$$
$$= \Theta(1) + \dots + \Theta(n-1) + \Theta(n)$$
$$= \Theta(1 + 2 + \dots + n)$$
$$= \Theta(n^2)$$

Quick Sort: Best Case Analysis

- The best case for quick sort is when partition function produces two sub-arrays with (n/2) elements each.
- The recurrence is then
- $T(n) = 2T(n/2) + \Theta(n)$
- By case 2 of the master method, the solution is:
- $T(n) = \Theta(n.lg(n))$

Dynamic Programming

Dynamic Programming

- Dynamic programming, like the divide-and-conquer method, solves problems by combining the solutions to sub-problems.
- Dynamic programming is applicable when the sub-problems are not independent, that is, when sub-problems share sub-problems.
- Dynamic programming is typically applied to ***optimization problems***.

Dynamic Programming

- The development of a dynamic-programming algorithm can be broken into a sequence of four steps:

1. Characterize the structure of an optimal solution.
2. Recursively define the value of an optimal solution.
3. Compute the value of an optimal solution in a bottom-up fashion.
4. Construct an optimal solution from computed information.

Traveling Salesperson Problem

- A salesman starts from his hometown and has to visit n different cities to meet his clients.
- He wants to visit each city exactly once, and then come to his hometown (i.e. the place he started from).
- He would like to optimize his cost of travel.
- For example: A postman picks up letters from the post office, delivers them from door to door and then comes back to the post office. This is an example of the Traveling Salesperson Problem.

The Cost Matrix

- Let us represent the cities by numbers 1 to n.
- Then the cost of traveling can be represented by a (n x n) matrix A, which is called the cost matrix.
- The (i, j)th entry of the matrix gives the cost of traveling from city i to city j.
- In general, $A_{ij} \neq A_{ji}$
- For example, in the matrix below, cost of traveling from city 1 to city 3 is 10.

$$A = \begin{pmatrix} 0 & 5 & 10 & 15 \\ 2 & 0 & 5 & 1 \\ 1 & 2 & 0 & 10 \\ 15 & 5 & 1 & 0 \end{pmatrix}$$

The structure of optimal solution

- Let G(i, S) represent the optimal cost, such that salesman starts from city i and visits all the cities in set S and then comes back to city 1.
- For example: if there are 4 cities and the salesman starts from city 1, then cost of optimal tour would be given by G(1, {2, 3, 4}).
- This means that, the salesman starts from city 1, visits all the cities in set {2, 3, 4} and then comes back to city 1 in a way that his travel cost is minimum.

Recursive value of optimal solution

- We have defined G(i, S) as the optimal cost of traveling from city i through all cities in set S and then back to city 1.
- Let us assume that the salesman traveled from city i to city j first. Then the corresponding cost is A_{ij} where A is the cost matrix.
- After traveling to city j, the salesman now has to travel optimally from city j, through all cities in set $S - \{j\}$ and then come back to city 1. This is given by $G(j, S - \{j\})$.
- Thus: $G(i, S) = A_{ij} + G(j, S - \{j\})$
- We can compute G(i, S) by solving the above equation for all possible values of j and selecting the minimum one. i.e.
- $G(i,S) = \min_{j \in S}[A_{ij} + G(j, S - \{j\})]$

Example

Compute the optimal Salesperson's tour from the Cost matrix given below:

$$A = \begin{pmatrix} 0 & 5 & 10 & 15 \\ 2 & 0 & 5 & 1 \\ 1 & 2 & 0 & 10 \\ 15 & 5 & 1 & 0 \end{pmatrix}$$

- We have to compute G(1, {2, 3, 4})
- We know that
- $G(i, S) = \min_{j \in S}[Aij + G(j, S - \{j\})]$
- Then G(1, {2, 3, 4})

$$= \min_{J \in \{2,3,4\}} \begin{bmatrix} A_{12} + G(2, \{3,4\}) \\ A_{13} + G(3, \{2,4\}) \\ A_{14} + G(4, \{2,3\}) \end{bmatrix}$$

- Where G(2, {3, 4})

$$= \min_{J \in \{3,4\}} \begin{bmatrix} A_{23} + G(3, \{4\}) \\ A_{24} + G(4, \{3\}) \end{bmatrix}$$

$$= \min \begin{bmatrix} 5 + 10 + 15 \\ 1 + 1 + 1 \end{bmatrix} = 3$$

- Similarly computing G(3, {2, 4}) & G(4, {2, 3}) we can compute G(1, {2, 3, 4})

Greedy Method

Introduction

- A **greedy algorithm** is a technique for solving an optimization problem.
- Makes a **locally optimal** choice
- With the **hope**
- that it will lead to a **globally optimal** solution

The Greedy Strategy

- A greedy algorithm obtains an optimal solution by making a sequence of choices.
- For each decision point in the algorithm the choice that seems best at the moment is selected.
- This strategy does not always produce an optimal solution.
- Sometimes it does…

The Greedy Choice Property

- A globally optimal solution can be arrived at by a locally optimal (greedy) choice.
- In dynamic programming we make a choice at each step, but the choice depends on the solutions of sub-problems.
- In greedy algorithm whatever choice seems best at the moment and solve the sub-problems arising after the choice has been made.
- Dynamic programming works in bottom-up manner whereas greedy strategy works in top-down fashion.

The Optimal Sub-structure Property

- A problem exhibits optimal sub-structure if an optimal solution to the problem contains within it optimal solution to sub-problems.

- This property is exploited by both greedy and dynamic programming strategies.

Job Sequencing with Deadlines

- We are given a set of n jobs
- Associated with each job is an integer deadline $d_i > 0$ and a profit $p_i > 0$
- For any job the profit is earned if and only if it is completed within deadline

Job Sequencing with Deadlines

- Basic Assumptions:

1. Only one job can be processed at a time
2. Once the job gets started it must be allowed to perform till completed
3. No job can miss the deadline, i.e. it must start within the deadline
4. To complete each job it takes one unit of time

Job Sequencing with Deadlines

- Our problem is to maximize the profit such that maximum jobs are completed within deadline.

- Let n = number of jobs = 3
- Corresponding profits (p1, p2, p3) = (10, 50, 25)
- Corresponding deadlines (d1, d2, d3) = (2, 2, 1)

- We have to sequence these jobs in such a way to gain maximum profit.

Job Sequencing with Deadlines

- The maximum deadline for the given jobs is 2
- So we can't assign more than 2 jobs
- To ensure maximum profit the greedy strategy is to assign maximum profitable jobs within their respective deadlines
- So, to earn maximum profit, we will follow a greedy strategy, which is to assign the highest profitable job in the queue, then the second highest profitable job and so on

Job Sequencing with Deadlines

- All the feasible solutions, sequence of execution and profits of the given problem are as follows:

Feasible Solutions	Sequence of Execution	Profit
(J1, J2)	(J2, J1) or (J1, J2)	60
(J1, J3)	J3, J1	35
(J2, J3)	J3, J2	75
(J1)	J1	10
(J2)	J2	50
(J3)	J3	25

- From the above example the profit is maximum for job sequence (J3, J2) = 75

Job sequencing with deadlines

- The greedy approach works as follows:

1. Find out the maximum number of jobs m, that can be performed, within the given deadlines.
2. Let J be an empty set.
3. Take out the job with he maximum profit, put it in the set J of jobs.
4. Continue step 2 until J has m jobs
5. Arrange the jobs in J in the order of non-decreasing deadlines.

Job sequencing with deadlines

- For our example:
- Maximum two jobs can be performed, since the highest possible deadline is 2.
- We start with an empty set J of jobs.
- We pick J2, since it has the most profit (50), and put it in J.
- Then out of the remaining jobs we pick J3 with the most profit (25) and put it in J.
- Now J has two jobs {J2, J3}
- We arrange them in the order of non-decreasing deadlines, i.e. J3 first then J2.
- That yields the result {J3, J2}, with profit 75.

Knapsack Problem

- A Thief enters a store in the night, with a knapsack in his hand.
- The knapsack can carry only a certain amount of weight, say W at most.
- The store has n items, each having a value V and weight W.
- The thief has to select items, such that he fills his knapsack with items, giving him the maximum value.

0/1 and Fraction Knapsack problems

- There are two variations of this problem.
- In the 0/1 knapsack problem the thief has to either take the entire item or leave it altogether.
- In the fraction knapsack problem the thief is allowed to take fraction of items.
- For example, if the store contained gold biscuits it would constitute a 0/1 knapsack problem. Whereas if the store contained gold dust, it would constitute a fraction knapsack problem, since the thief can take a portion of the gold dust.

Example

- Let us suppose the store has three items as shown below:

Item	Value (V)	Weight (W)	V/W
1	60	10	6
2	100	20	5
3	120	30	4

- Let the Knapsack capacity be 50.
- The thief would like to make a greedy choice, hence he will pick up items with higher V/W values.

Example: 0/1 Knapsack Problem

- Let us assume that it is a 0/1 Knapsack Problem, i.e. the thief has to either take the whole item or leave it.
- By making a greedy choice the thief would pick up items 1 & 2 first, thereby filling the knapsack by W = 30 and earning V = 160
- He can't take the item 3 anymore since his knapsack can only take 20 units of weight, whereas item 3 weighs 30.
- But as we can see, if the thief picked up items 2 & 3 he would have earned V = 220.
- Thus greedy choice did not lead to an optimal solution

Example: Fraction Knapsack Problem

- Let us assume that it is a Fraction Knapsack Problem, i.e. the thief can take a part of an item.
- By making a greedy choice the thief would pick up items 1 & 2 first, thereby filling the knapsack by W = 30 and earning V = 160
- He will now take 20 units of weight of item 3, thereby gaining another 80 units of money.
- His total profit is 160 + 80 = 240
- Thus greedy choice leads to an optimal solution for a fraction knapsack problem.

Backtracking

Steps

- Define the solution space of the problem.
- Organize the solution space so that it can be searched easily, typically as a tree.
- Search the solution space in a depth-first manner beginning at the start node.
- The start node is both a **Live** node as well as an **E** node (Expansion Node). From this E node we try to move to the next node.
- The new node becomes the next E node and Live node, the old node remains a Live node

Strategy

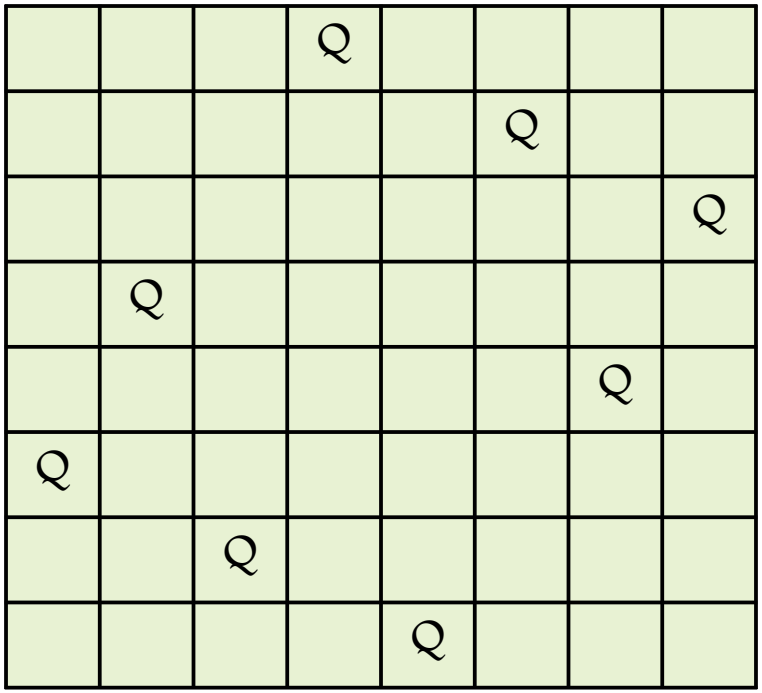
- If we can't move to a new E node then the current E node dies and we backtrack to the most recently seen Live node.
- The search continues in this manner until we reach the goal node or there are no more Live nodes.

Summary

- The backtracking algorithm enumerates a set of partial candidates that, in principle, could be *completed* in various ways to give all the possible solutions to the given problem. The completion is done incrementally, by a sequence of *candidate extension steps.*
- Conceptually, the partial candidates are the nodes of a tree structure, the *potential search tree.* Each partial candidate is the parent of the candidates that differ from it by a single extension step; the leaves of the tree are the partial candidates that cannot be extended any further.
- The backtracking algorithm traverses this search tree recursively, from the root down, in depth-first order.

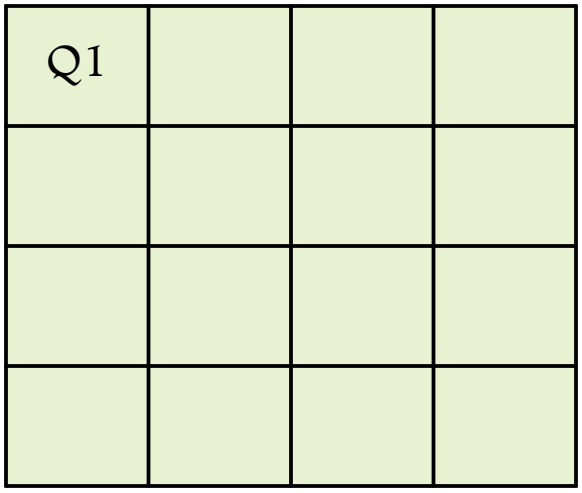
The 8 Queens Problem

We have an 8x8 chessboard, our job is to place eight queens on the chessboard, so that no two of them can attack. That is, no two of them are in the same row, column or diagonal

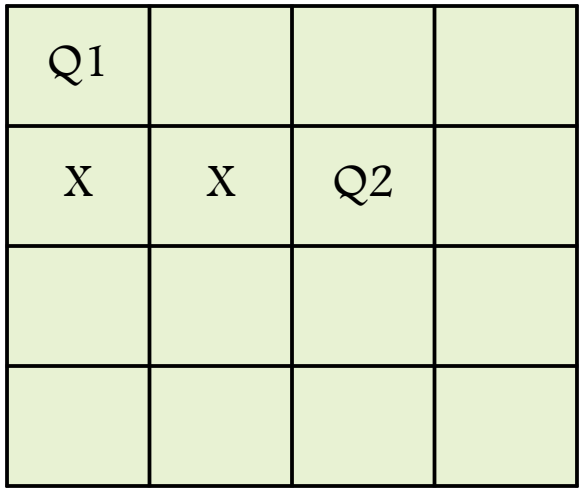


The n Queens Problem

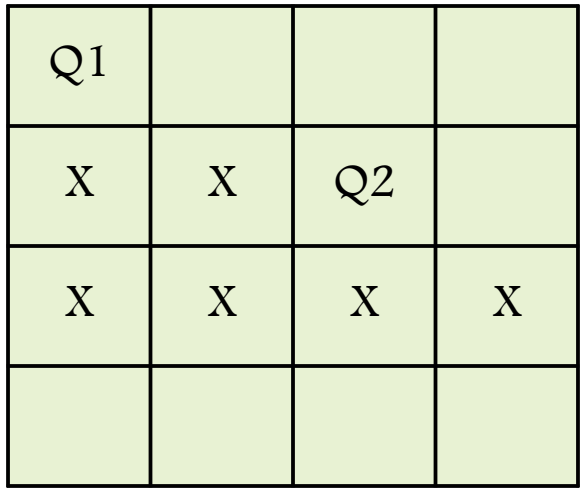
The generalized version of the q Queens problem is the n Queens problem, where n is a positive integer. Let us illustrate a backtracking approach to this problem with a 4 Queens example.



Step1: Place Q1 on Col 1



Step2: Place Q2



No legal place for Q3

The n Queens Problem

Q1			
X	X	X	Q2

Backtrack and alter Q2

Q1			
X	X	X	Q2
X	Q3		

Place Q3

Q1			
X	X	X	Q2
X	Q3		
X	X	X	X

No legal place for Q4

	Q1		

Backtrack and alter Q1

	Q1		
X	X	X	Q2

Place Q2

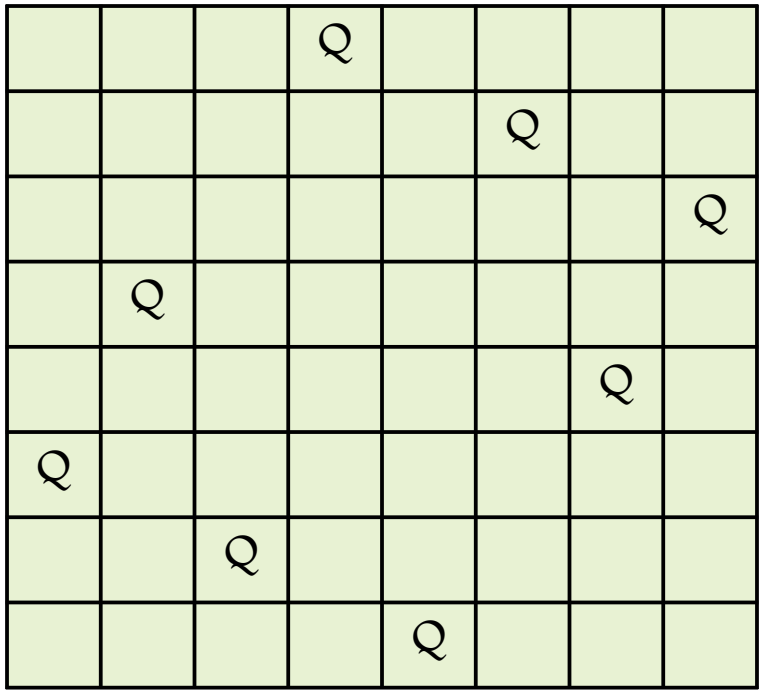
	Q1		
X	X	X	Q2
Q3			
X	X	Q4	

Place Q3 and then Q4

The n Queens Problem

Let $(x_1, x_2, \ldots, x_n)$ represent a solution in which x_i is the column of the i^{th} row, where the i^{th} queen has been placed. The x_is will all be distinct, since no two queens can be in the same column.

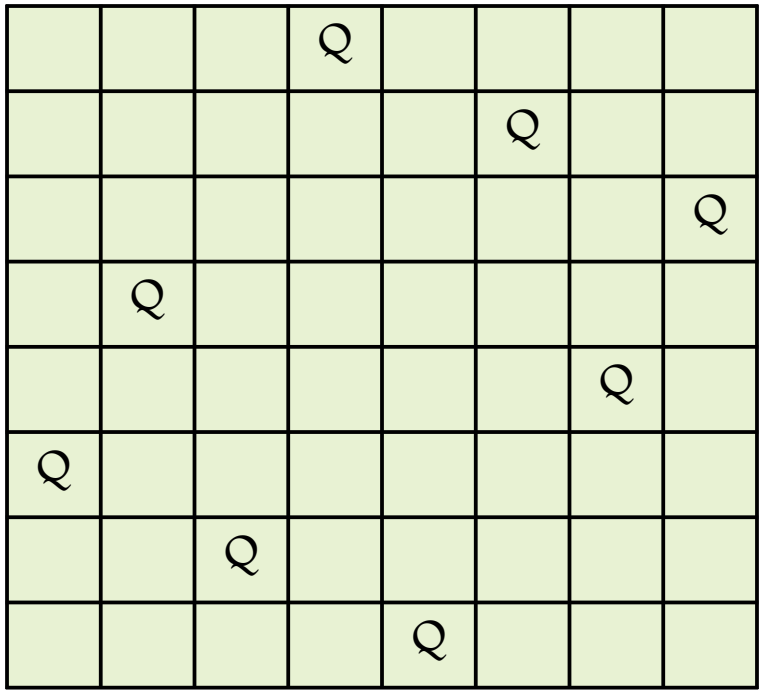
Now, how to check that they are not in the same diagonal?



The n Queens Problem

Let the chessboard square be represented by the 2D array a[1…8, 1…8].

Every element on the same diagonal that runs from top-left to right-bottom has the same (row – column) value. For example, a Queen at a[4,2] has attacking Queens, diagonal to it at a[3,1], a[5,3], a[6,4] … (up-left to low–right).



The n Queens Problem

Also all elements on the same diagonal from top-left to bottom-right have the same (row + column) value.

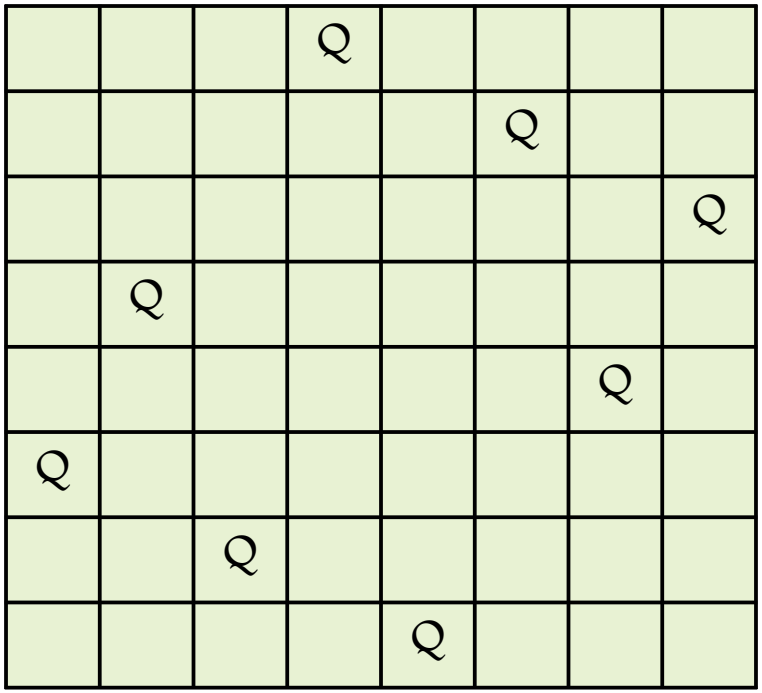
If the Queens are placed at (i,j) and (k,l) cells, then they are in the same diagonal only if

$i - j = k - l$ or

$i + j = k + l$

i.e.

$Abs(j - l) = Abs(i - k)$



The n Queens Problem

```
bool place(int k, int i) // returns true if a queen can be placed at k row and i column
{
        for (int j = 1; j <= k; j++)
        {
                if ((x[j] == i) || (abs(x[j] - i) == abs(j - k)))
                // in the same column or same diagonal?
                {
                        return false;
                }
        }
        return true;
}
```

The n Queens Problem

```
void nQueens(int k, int n)
{
        for (int i = 1; i <= n; i++)
        {
                if (place(k, i))
                {
                        x[k] = i;
                        if (k == n)
                        {
                                for (int j = 1; j <= n; j++)
                                        cout<<"Queen "<<j<<" is at "<<x[j]<<": ";
                        }
                        else
                                nQueens(k + 1, n);
                }
        }
}
```

Branch & Bound

Steps

- Define the solution space of the problem.
- Organize the solution space so that it can be searched easily, typically as a tree.
- Search the solution space in a breadth-first manner beginning at the start node.
- The start node is both a **Live** node as well as an **E** node (Expansion Node). From this E node we try to move to the next node.

Strategy

- Each Live node becomes an E node exactly once.
- When a node becomes a E node all new nodes that can be reached using a single move are generated.
- Generated nodes that can't possibly lead to an optimal solution are discarded.
- Remaining nodes are added to the list of live nodes, one node from this list is selected to become the next E node.
- The selected node is extracted from the list of live nodes and expanded. The expansion process is continued until the answer is found or the list of live nodes is empty.

Two ways of selecting the next E node

- FIFO
  - Extract from the list of live nodes in the same order as they are put into it
- Least Cost/Max Profit
  - Associate a cost with each node
  - The next E node is the live node with least cost/max profit

Summary

- B&B is composed of two main actions, but there's an additional step. The first step is, as its name suggests, the *branching*. This is where we define the tree structure from the set of candidates in a recursive manner.
- The second action is called *bounding* since this procedure calculates the upper and lower bounds of each node from the tree.
- Furthermore, there's the additional *pruning* step that we can add. In a nutshell, it means that if the lower bound for some node of the tree is greater than the upper bound of some other node of the tree, then the first node of the tree can be "discarded".

The 15 Puzzle Problem

Consists of 15 numbered tiles in a square frame of capacity 16 tiles, with an initial arrangement

1	3	4	15
2		5	12
7	6	11	14
8	9	10	13

Objective is to transform the initial arrangement into the goal arrangement

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	

The 15 Puzzle Problem

•The only legal moves are the ones in which the tiles adjacent to the empty slot (ES) move to ES. For example only four tiles in this puzzle can legally move; 2,3,5,6.

•A state is reachable from the initial state if there are a series of legal moves from the initial state to that state.

1	3	4	15
2		5	12
7	6	11	14
8	9	10	13

The 15 Puzzle Problem

- A straight forward way of solving the puzzle is to search the state space of the initial state for the goal state and use the path from the initial state to the goal state as the answer.
- There are 16! different arrangements of the tiles in a frame.

1	3	4	15
2		5	12
7	6	11	14
8	9	10	13

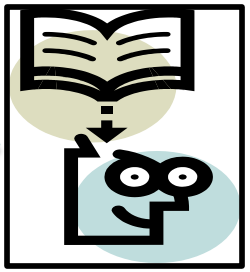
The 15 Puzzle Problem

1	2	3	4
5	6	7	8
9	10	11	12
13	15	14	

Can you solve this?

The 15 Puzzle Problem: Solvability

How to check if a given game is solvable?



12	1	10	2
7	11	4	14
5		9	15
8	13	6	3

The 15 Puzzle Problem: Solvability

1. Find the total number of inversions

12	1	10	2
7	11	4	14
5		9	15
8	13	6	3

An inversion is when a tile precedes another tile with a lower number on it. The solution state has zero inversions. For example, if, in a 4 x 4 grid, number 12 is top left, then there will be 11 inversions from this tile, as numbers 1-11 come after it

The 15 Puzzle Problem: Solvability

1. Find the total number of inversions

12	1	10	2
7	11	4	14
5		9	15
8	13	6	3

the 12 gives us 11 inversions
the 1 gives us none
the 10 gives us 8 inversions
the 2 gives us none
the 7 gives us 4 inversions
the 11 gives us 6 inversions
the 4 gives us one inversion
the 14 gives us 6
the 5 gives us one
the 9 gives us 3
the 15 gives us 4
the 8 gives us 2
2 from the 13
one from the 6
none from 3

Total
49
Inversions

The 15 Puzzle Problem: Solvability

2. Find the row number of empty slot from bottom

12	1	10	2
7	11	4	14
5		9	15
8	13	6	3

ES in
Row 2

The 15 Puzzle Problem: Solvability

The 15 puzzle problem is solvable if

12	1	10	2
7	11	4	14
5		9	15
8	13	6	3

It has an even number of inversions and ES is in an odd row, counting from the bottom.

OR

It has an odd number of inversions and ES is in an even row, counting from the bottom.

The 15 Puzzle Problem: Solvability

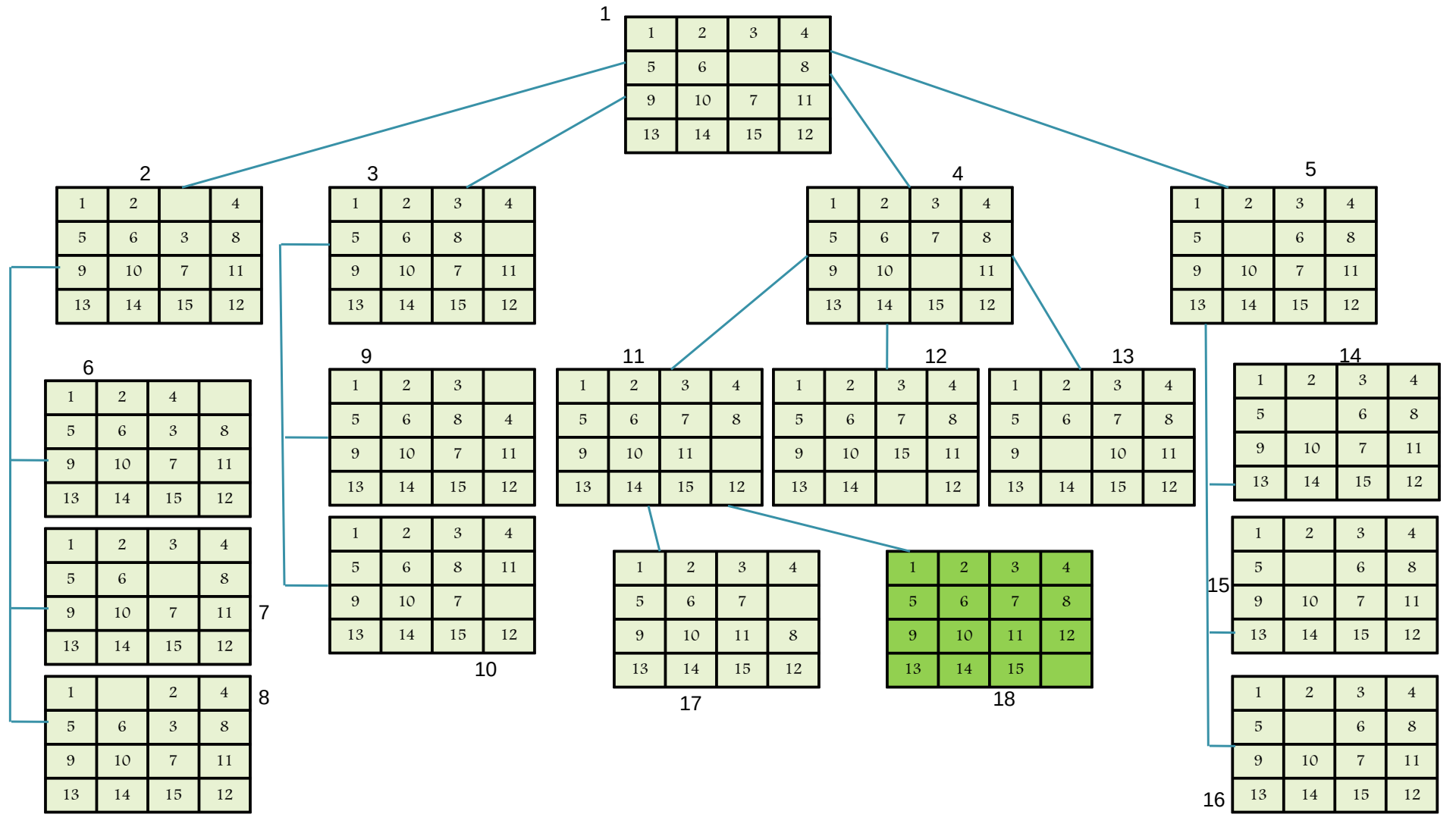
12	1	10	2
7	11	4	14
5		9	15
8	13	6	3

- Note that the goal state satisfies the rule of solvability
- Those properties are satisfied by every legal move
- The goal state can be reached from the initial state only through a series of legal moves.

Solving the 15 Puzzle Problem

- If a 15 puzzle is solvable then we can proceed to determine a series of legal moves from initial to goal state.
- To carry out this search the state space can be organized as a tree. The children of each node x represent the number of nodes reachable from x by one legal move.
- It is convenient to think that the empty slot (ES) moves left, right, up or down rather than a tile.

Solving the 15 Puzzle Problem



Solving the 15 Puzzle Problem

- A more intelligent search method would be to associate a cost with each node of the tree and choose the node with the least cost at each step.
- Let c(x) be the corresponding cost for node x. Where

  $$c(x) = f(x) + g(x)$$

- f(x) = length of the path from root to node x
- g(x) = number of non-blank tiles not in their goal position
- g(x) is an estimate of the shortest path from x to goal node

Solving the 15 Puzzle Problem

- At least g(x) moves have to be made to transform state x to a goal state. So g(x) gives a lower bound for the cost.
- A least cost search will begin with node 1 as the E node. All its children 2,3,4,5 are generated.
- The next node to become an E node is a live node with least c(x).
- Now, c(2) = 1 + 4 = 5, c(3) = 1 + 4 = 5, c(4) = 1 + 2 = 3, c(5) = 1 + 4 = 5. So 4 becomes the next E node.
- This process continues till we reach the goal node.


Thank
You!