Algorithm Design

CE2103 - Algorithms and Data Structures II

Disclaimer / Descargo de Responsabilidad

Esta presentación corresponde a una guía usada por el profesor durante las clases. La misma ha sido modificada para ser utilizado en el modelo de cursos asistidos por tecnología. No es una versión final, por lo que la misma podría requerir todavía hacer algunos ajustes. Para aspectos de evaluación esta presentación es solo una guía, por lo que el estudiante debe profundizar con el material de lectura asignado y lo discutido en clases para aspectos de evaluación.

This presentation corresponds to a guide material used by the professor during classes. It has been modified to be used in the model of technology-assisted courses. It is not a final version, so it may still require some adjustments. For evaluation aspects, this presentation is only a guide, so the student should delve with the assigned reading material and what has been discussed in class.

What is design?

➔ A plan or drawing produced to show the look and functions of an object before it is built

➔ Purpose, **planning** or intention that exists behind an action, fact o material object.

Algorithm design process

This process involves a few significant steps:

➔ Formulating the problem with enough mathematical precision. Concrete question and start thinking about algorithms to solve the problem.

➔ Designing an algorithm for the problem.

➔ Analyzing the algorithm to verify if it is correct and evaluate the algorithm’s efficiency.

Algorithm design process

This process involves a few significant st

- ➔ Formulating the problem with enough mathematical precision. Concrete que start thinking about algorithms to solv problem.
- ➔ Designing an algorithm for the proble
- ➔ Analyzing the algorithm to verify if it i and evaluate the algorithm's efficienc

This high-level strategy is carried out in practice with the help of a few fundamental design techniques

Algorithm design techniques

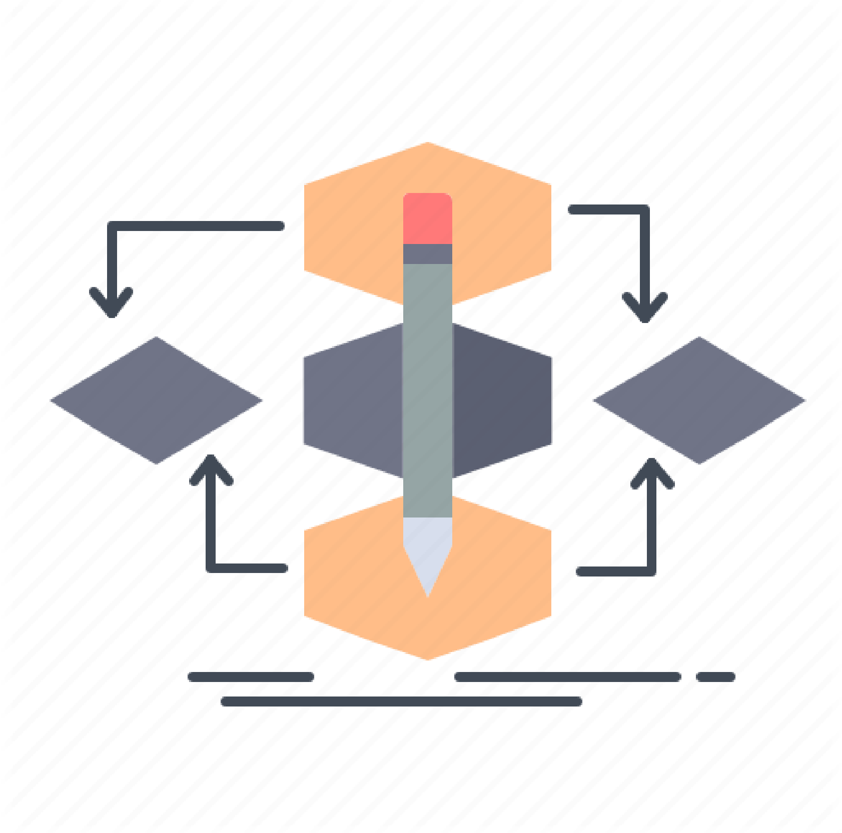
- ➔ Are very useful in assessing the inherent complexity of a problem and in formulating an algorithm to solve it.

- ➔ Becoming familiar with these design techniques is a gradual process.

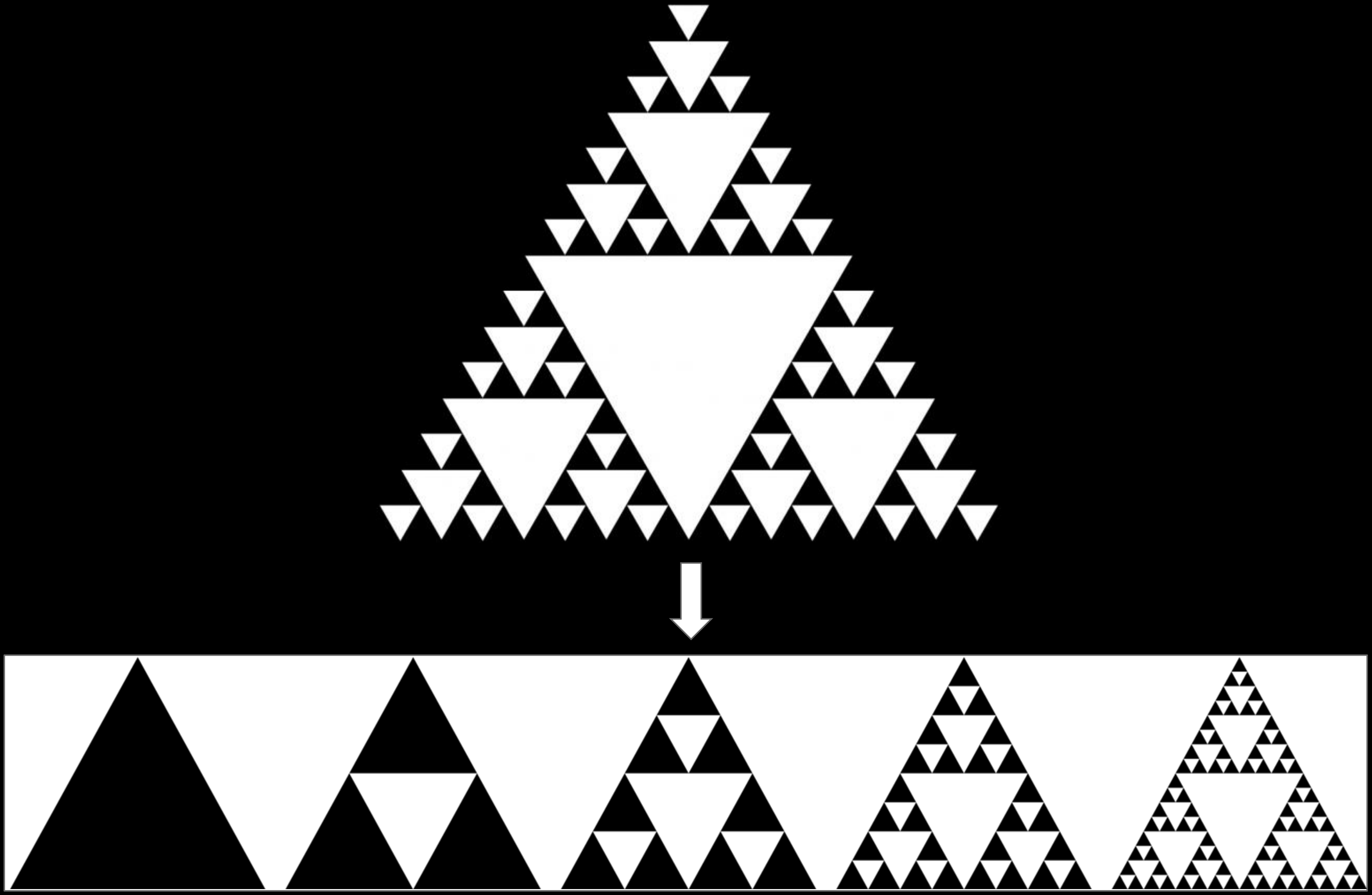
- ➔ With experience it is easy to associate the problems with categories.

Algorithm design techniques (Categories)

- Divide and conquer
- Backtracking
- Dynamic programming
- Heuristic algorithms
  - Genetic algorithms
  - Greedy algorithms
- Probabilistics algorithms



Divide and Conquer



Divide and Conquer

STRATEGY

→ **Divide.** Break the problem into several parts.

→ **Conquer.** Solves the problems in each part **recursively**.

→ **Combine** the solutions of subproblems into an overall solution.

CHARACTERISTICS

→ Once the problem becomes small enough that we no longer recurse, we are at the base case.

→ Independent subproblems.

→ Recursive.

→ Very efficient.

→ Can be naturally parallelized.

Divide and Conquer

+ *Simple, clear, robust and elegant. Ease of debugging and maintenance.*

STRATEGY

- → **Divide.** Break the problem into several parts.
- → **Conquer.** Solves the problems in each part **recursively**.
- → **Combine** the solutions of subproblems into an overall solution.

CHARACTERISTICS

- → Once the problem becomes small enough that we no longer recurse, we are at the base case.

.

- → Independent subproblems.
- → Recursive.
- → Very efficient.
- → Can be naturally parallelized.

Divide and Conquer

🟥 *Longer execution time than iterative ones and complexity associated by the use of stack.*

STRATEGY

- → **Divide.** Break the problem into several parts.
- → **Conquer.** Solves the problems in each part **recursively**.
- → **Combine** the solutions of subproblems into an overall solution.

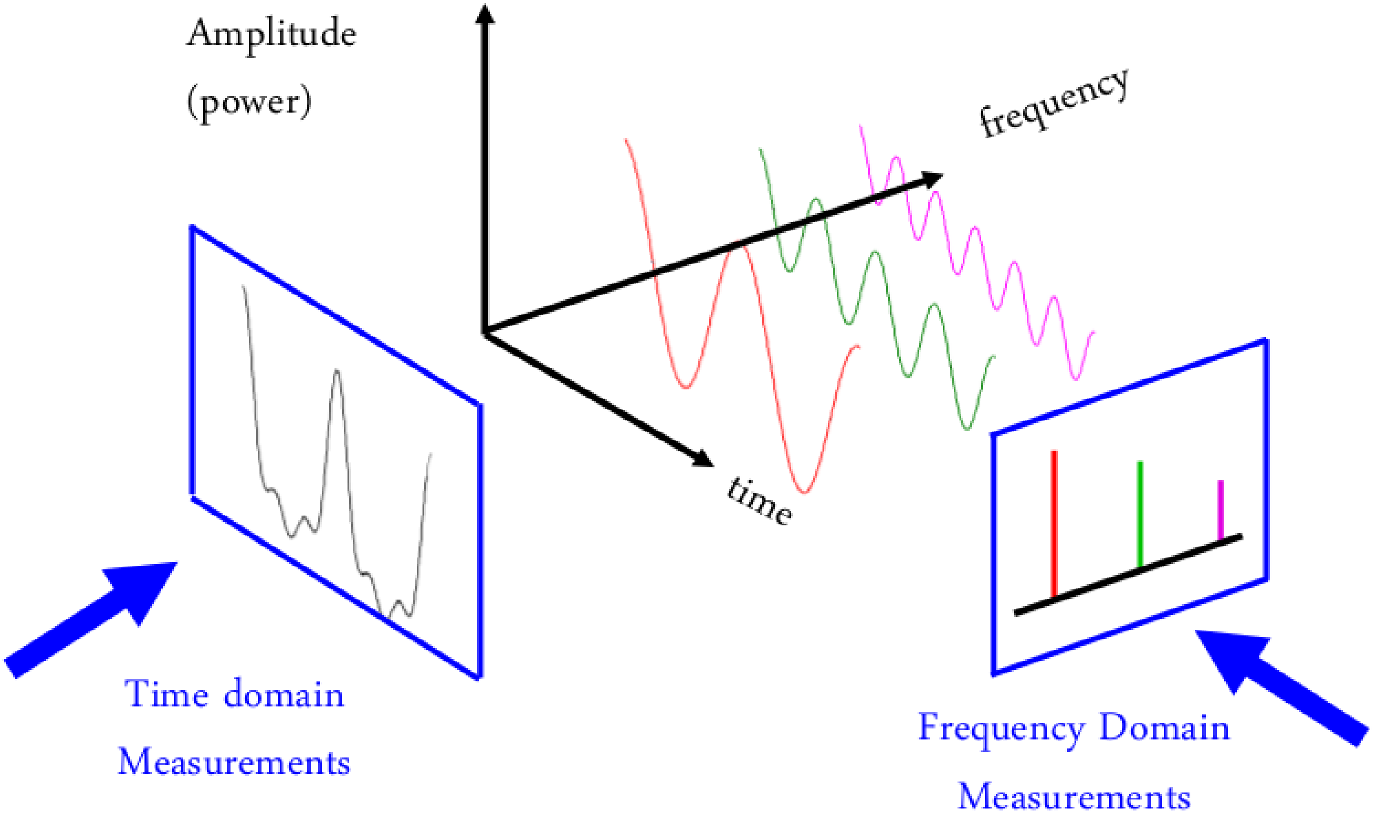
CHARACTERISTICS

- → Once the problem becomes small enough that we no longer recurse, we are at the base case.

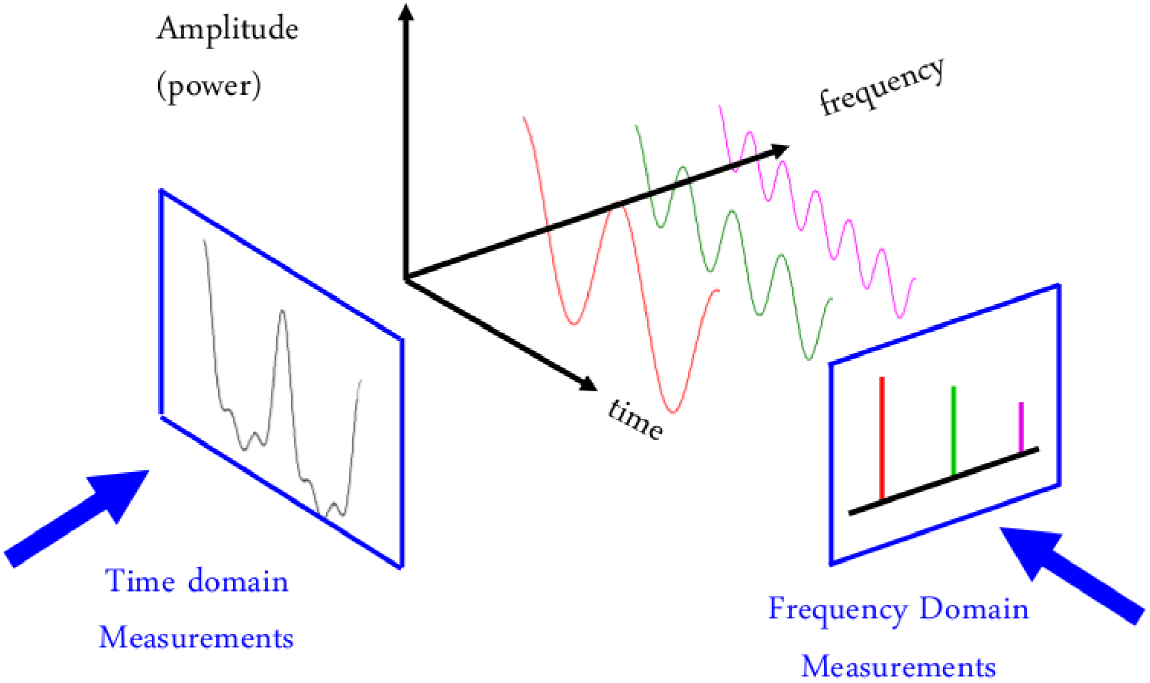
.

- → Independent subproblems.
- → Recursive.
- → Very efficient.
- → Can be naturally parallelized.

Divide and Conquer (Example)

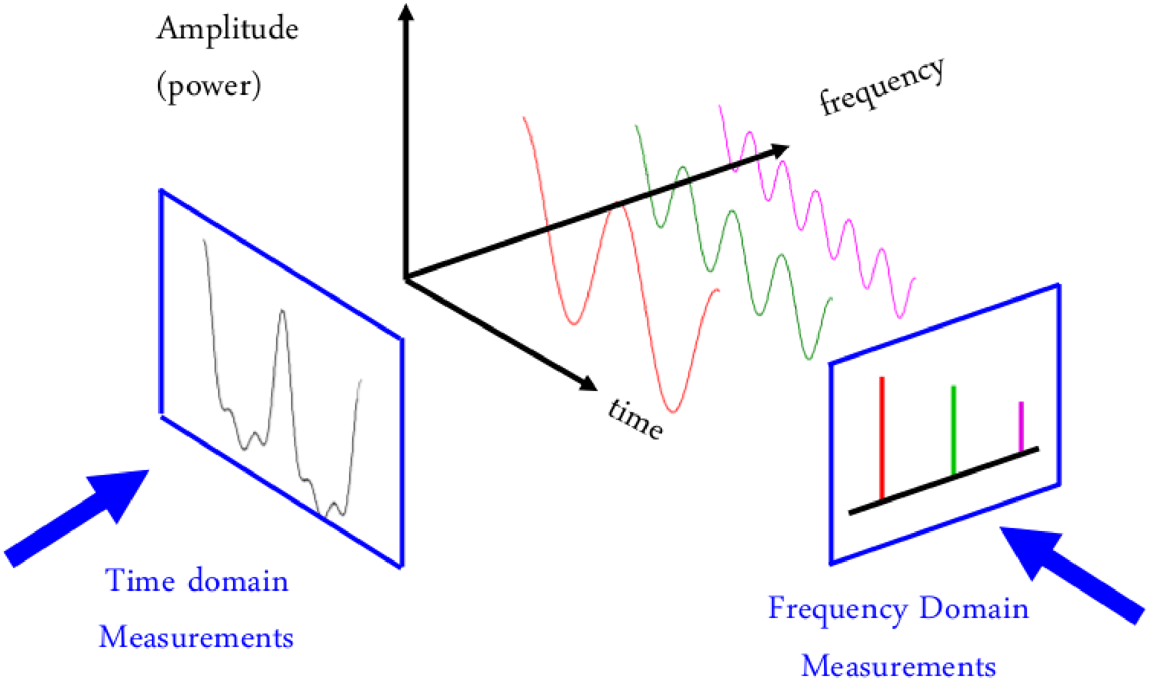


Divide and Conquer (Example)



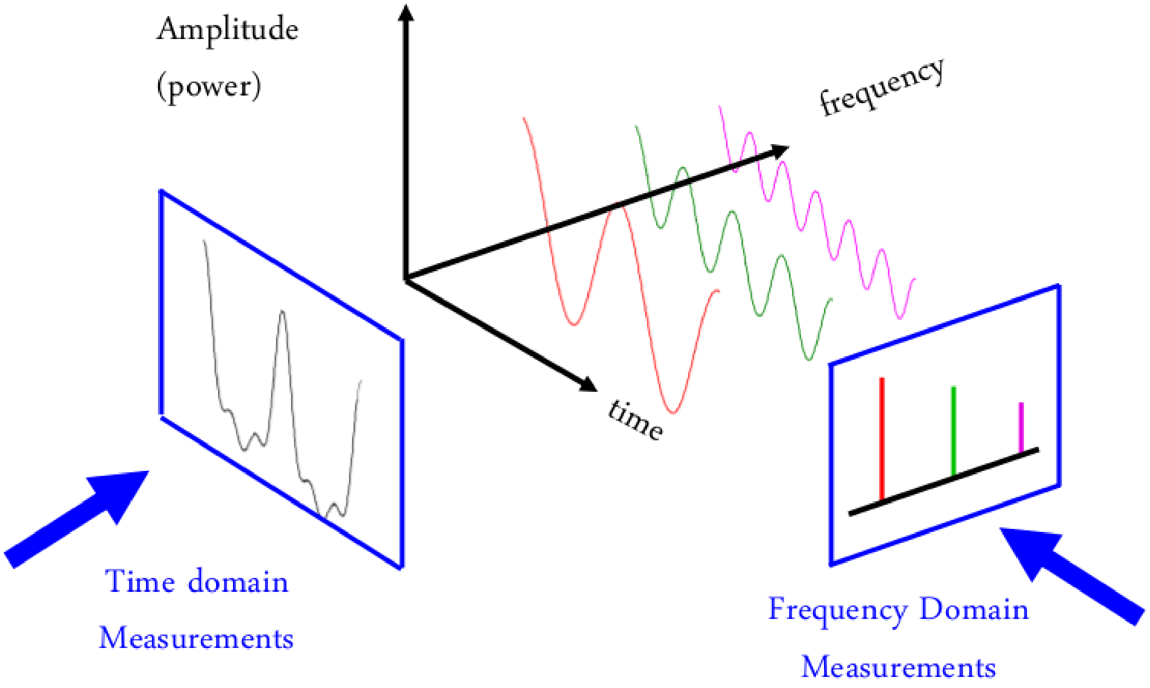
➔ Sound signals are a non discrete and continuous function.

Divide and Conquer (Example)



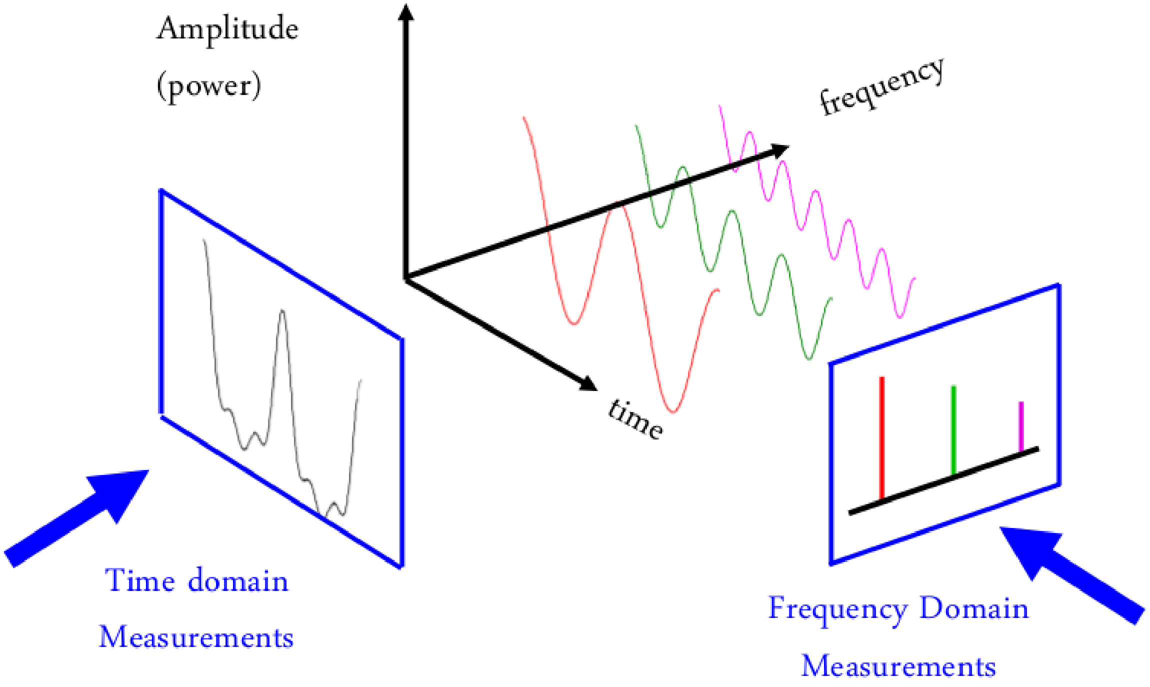
→ This means is very hard to process by a computer.

Divide and Conquer (Example)



➔ Fourier analysis allows to divide this continuous function into a set of frequencies that can be processed by a computer.

Divide and Conquer (Example)

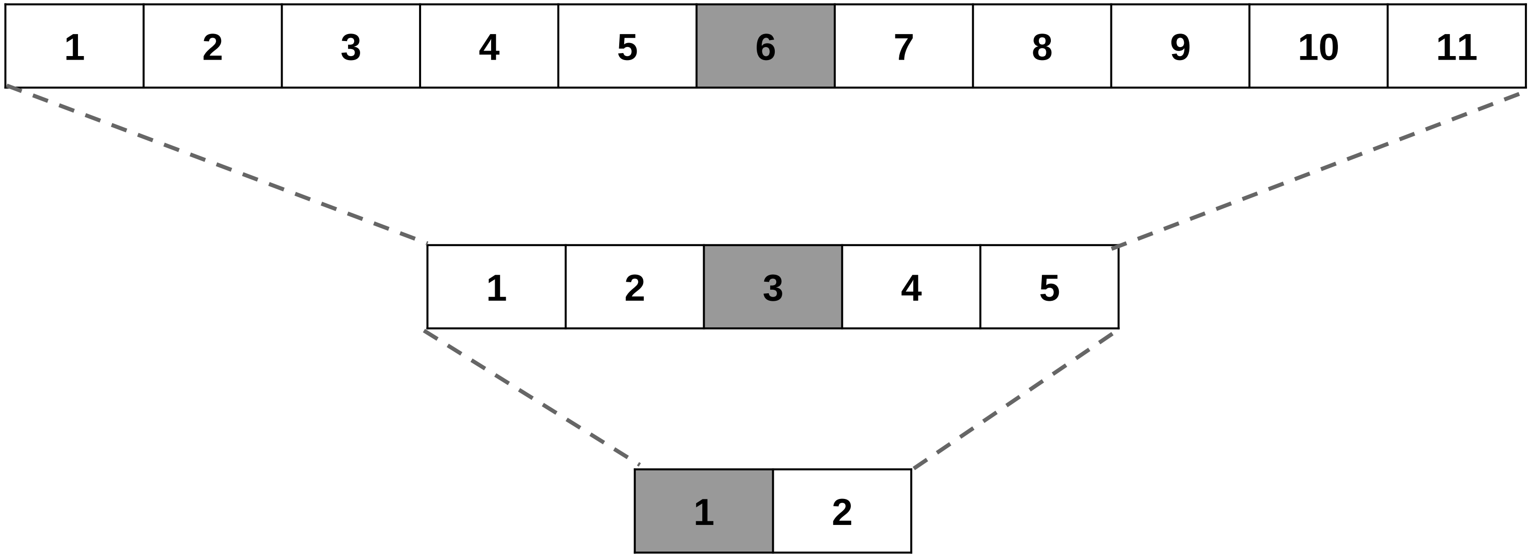


➔ This, in essence is a divide and conquer solution.

Divide and Conquer (Example)

➔ Binary Search

Search value: 1



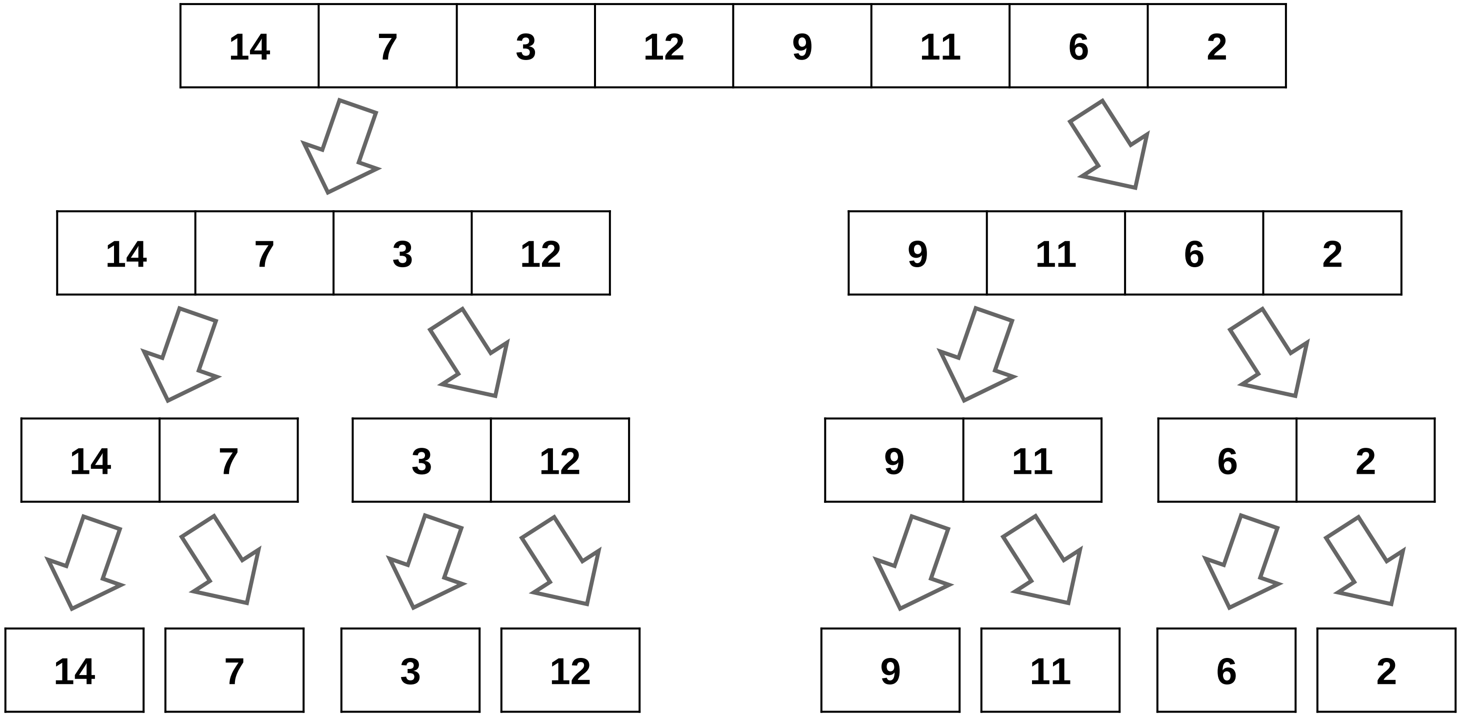
Divide and Conquer (Example)

➔ Binary Search

```
int binarySearch(double a[], double key, int lowEnd, int highEnd) {
    int middle;

    if (lowEnd > highEnd) {
        return -1; //Not found
    } else {
        middle = (lowEnd + highEnd) / 2;
        if (a[middle] == key) {
            return middle;
        } else if (a[middle] < key) {
            return binarySearch(a, key, middle + 1, highEnd);
        } else {
            return binarySearch(a, key, lowEnd, middle - 1);
        }
    }
```

Divide and Conquer (Example)

➔ Mergesort



14 7 3 12 9 11 6 2

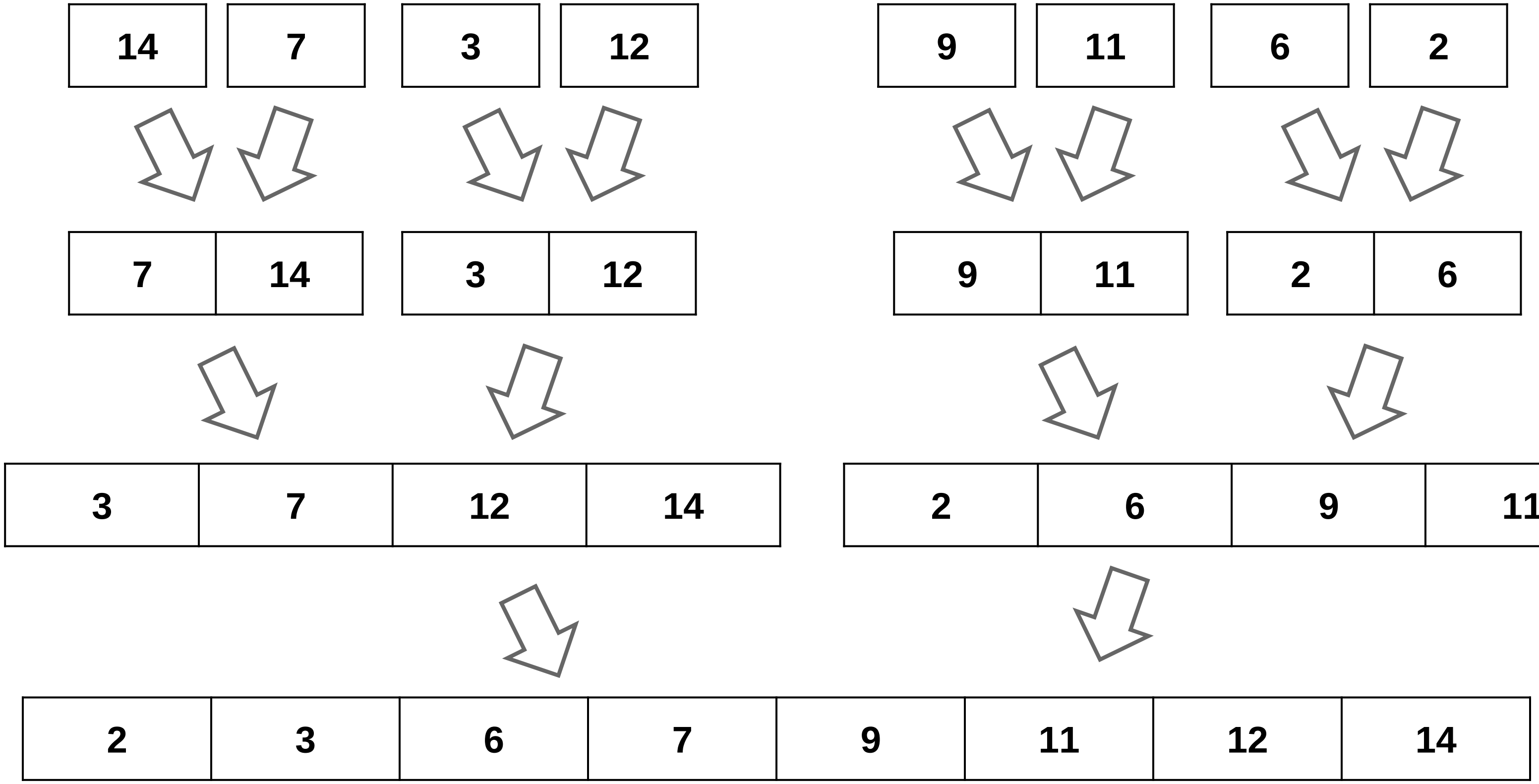
14 7 3 12 | 9 11 6 2

14 7 | 3 12 | 9 11 | 6 2

14 | 7 | 3 | 12 | 9 | 11 | 6 | 2

Divide and Conquer (Example)

➔ Mergesort



14 7 3 12 9 11 6 2

7 14 | 3 12 | 9 11 | 2 6

3 7 12 14 | 2 6 9 11

2 3 6 7 9 11 12 14

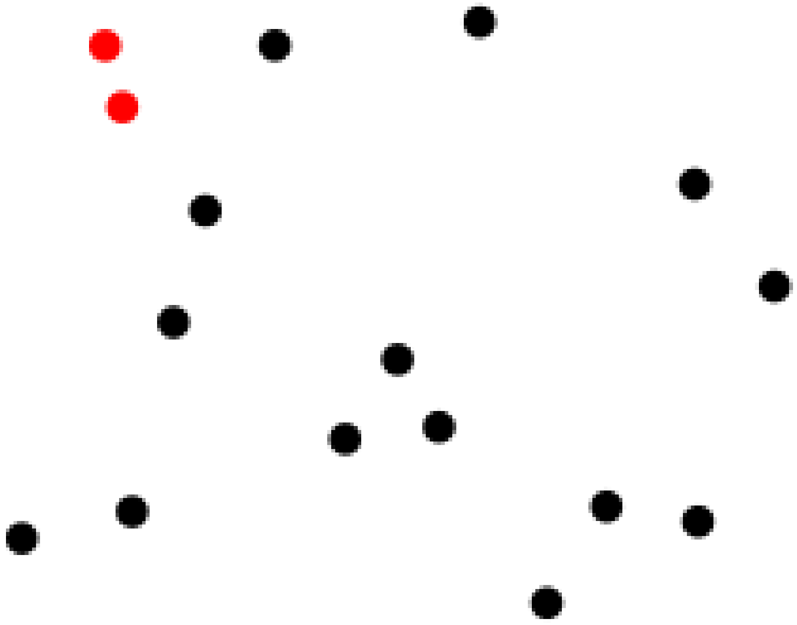
Divide and Conquer (Example)

- ➔ Mergesort

```
void mergeSort(int *a, int low, int high) {
   int mid;
   if (low < high) {
       mid = (low + high) / 2;
       mergesort(a, low, mid);
       mergesort(a, mid + 1, high);
       merge(a, low, high, mid);
   }
   return;
}
```

Divide and Conquer (Example)

➔ Closest pair of points problem



Divide and Conquer (Example)

➔ Closest pair of points problem

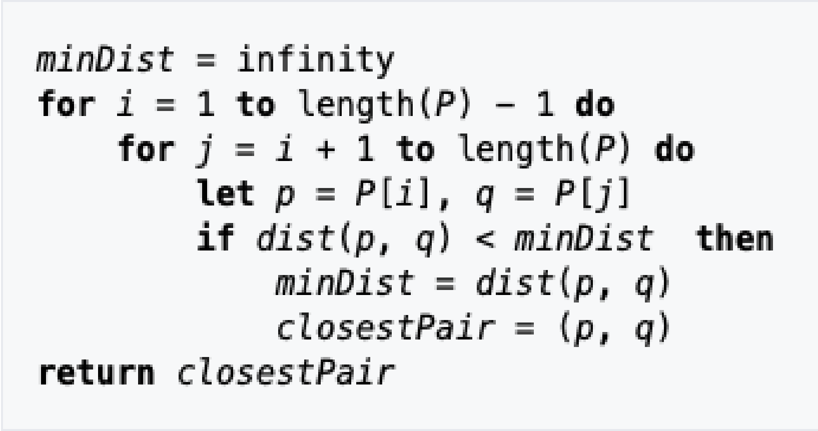
TRADITIONAL APPROACH

?

Divide and Conquer (Example)

➔ Closest pair of points problem

FIRST APPROACH	DIVIDE AND CONQUER APPROACH

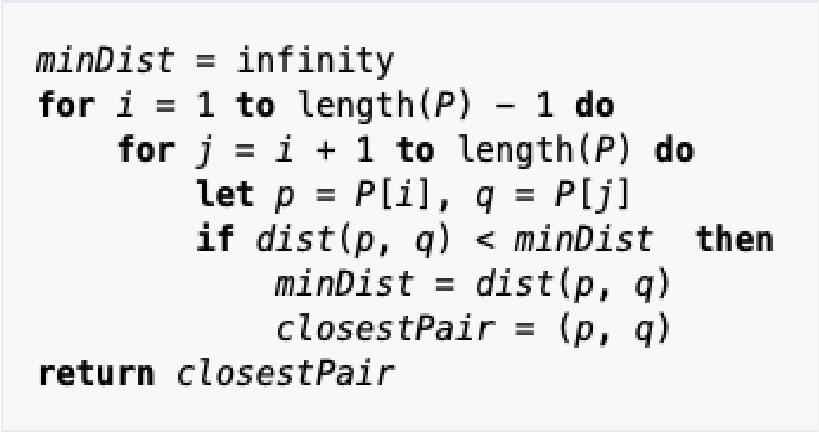


```
minDist = infinity
for i = 1 to length(P) − 1 do
    for j = i + 1 to length(P) do
        let p = P[i], q = P[j]
        if dist(p, q) < minDist  then
            minDist = dist(p, q)
            closestPair = (p, q)
return closestPair
```

Divide and Conquer (Example)

➔ Closest pair of points problem

FIRST APPROACH



```
minDist = infinity
for i = 1 to length(P) - 1 do
    for j = i + 1 to length(P) do
        let p = P[i], q = P[j]
        if dist(p, q) < minDist  then
            minDist = dist(p, q)
            closestPair = (p, q)
return closestPair
```

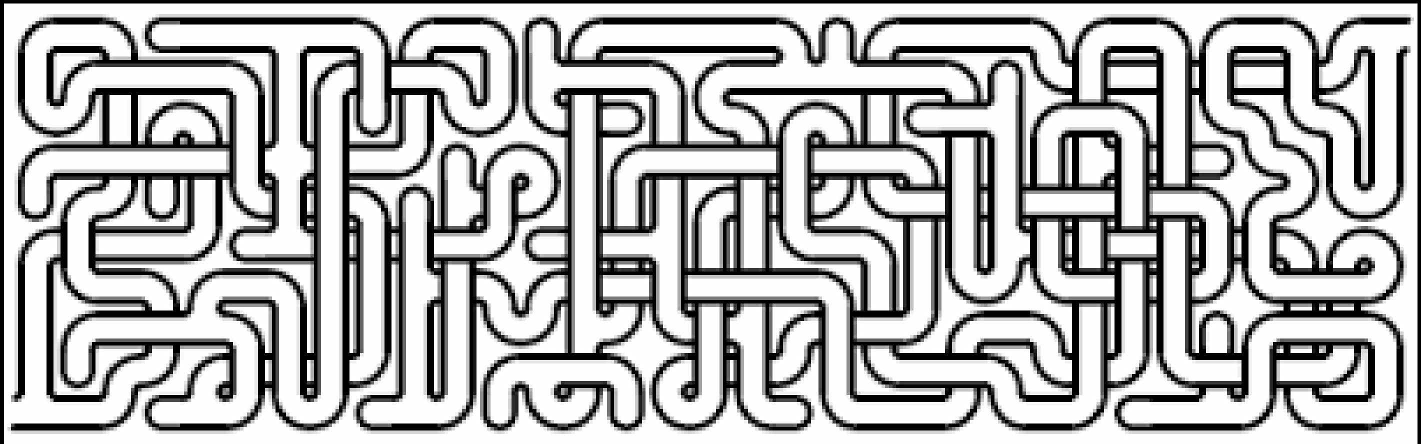
DIVIDE AND CONQUER APPROACH

1. Sort the points according to their X coordinate.
2. If the set size is 2, return the distance between them. If the set has 0 or 1 elements, return ∞.
3. Divide the set of points into two equal parts (of the same number of points). Solve the problem recursively in the left and right parts.

Divide and Conquer (Example)

- ➔ Other examples
  - ◆ **Quicksort.** The algorithm picks a pivot element, rearranges the array elements. All elements smaller than the picked pivot element move to left side of pivot, and all greater elements move to right side. Finally, the algorithm recursively sorts the subarrays on left and right of pivot element.
  - ◆ **Strassen’s Algorithm.** Algorithm to multiply two matrices.
  - ◆ **Karatsuba Algorithm.** Allows to compute the product of two large numbers using three multiplications of smaller numbers, each with about half as many digits as or, plus some additions and digit shifts.

Backtracking



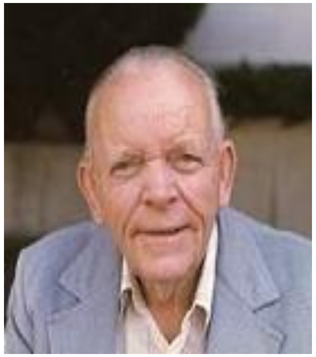
Backtracking (History)

The term “backtrack” was popularized by Derrick Henry Lehmer.

He was an american mathematician.

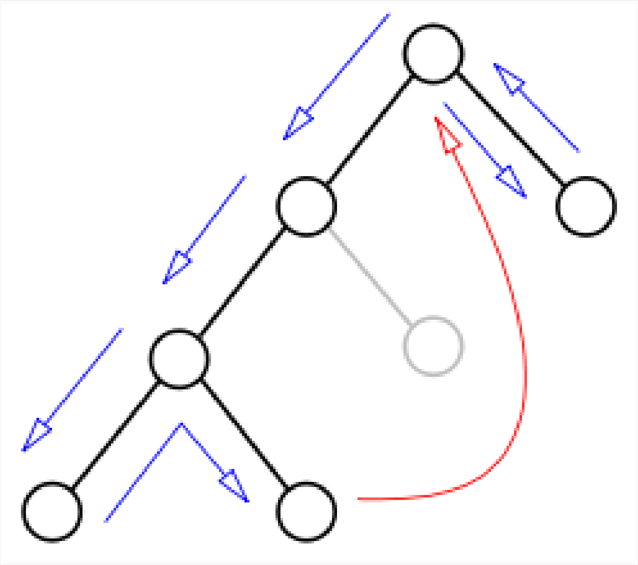
The language SNOBOL may have been the first to provide backtracking facility.

DERRICK HENRY LEHMER



Backtracking

STRATEGY



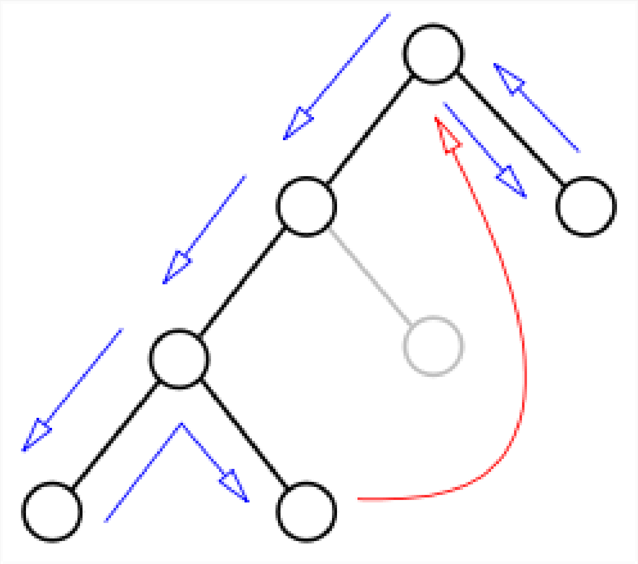
- ➔ Is a methodical way of **trying out various sequences of decisions**, until you find one that works.
- ➔ You can see the possible solutions as a tree.

STRATEGY (CONTINUE)

- ➔ Problem space consists of states (nodes) and actions (paths that lead to new states). When in a node can can only see paths to connected nodes
- ➔ If a node only leads to failure go back to its "parent"node. Try other alternatives. If these all lead to failure then more backtracking may be necessary

Backtracking

STRATEGY



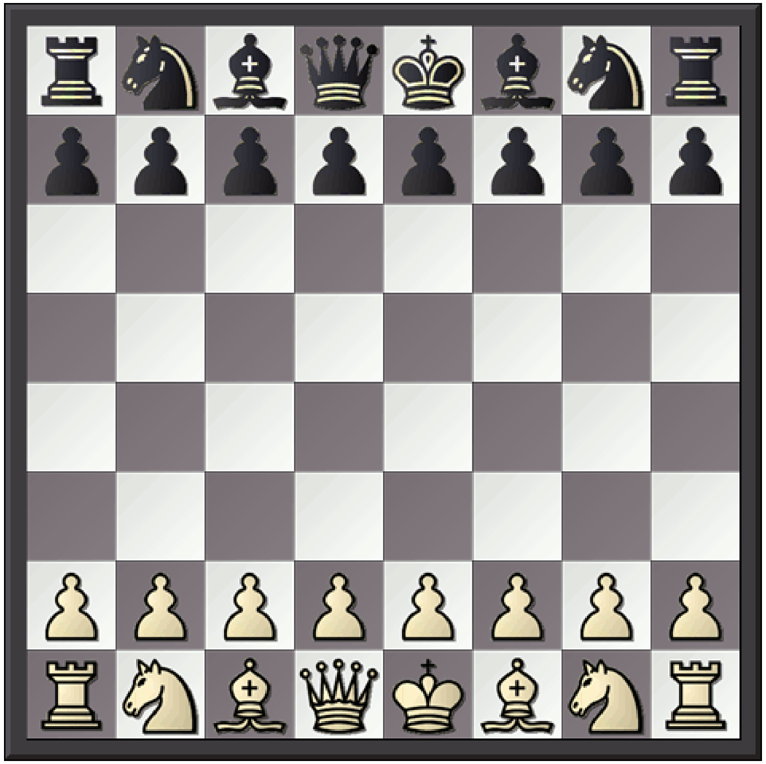
- ➔ Is a methodical way of **trying out various sequences of decisions**, until you find one that works.
- ➔ You can see the possible solutions as a tree.

CHARACTERISTICS

- ➔ Systematically trying and eliminating possibilities.
- ➔ Find all or some solutions.
- ➔ Used extensively by languages like Prolog.
- ➔ Naturally backtracking is a recursive algorithm.

Backtracking (Example)

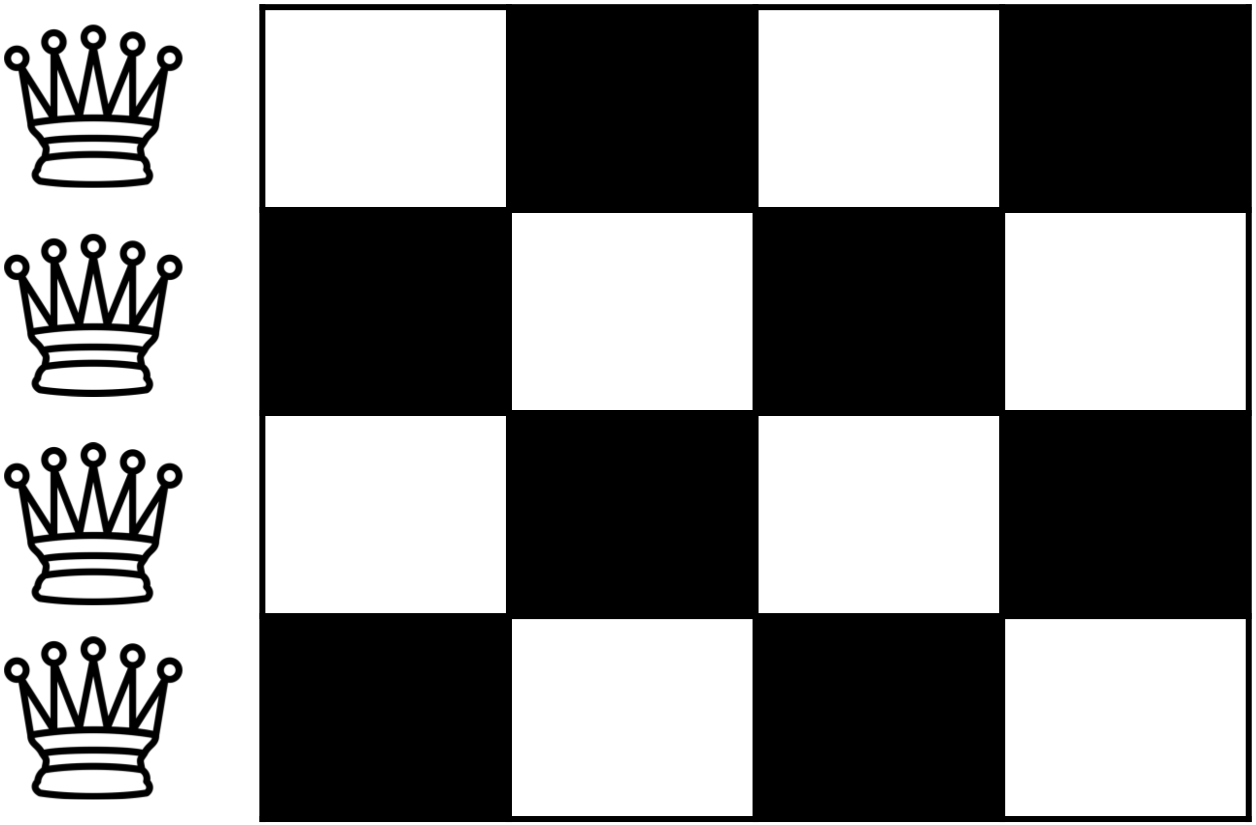
- ➔ Eight Queens Problem



Place eight queens in a board in such a way that no queen interferes with another

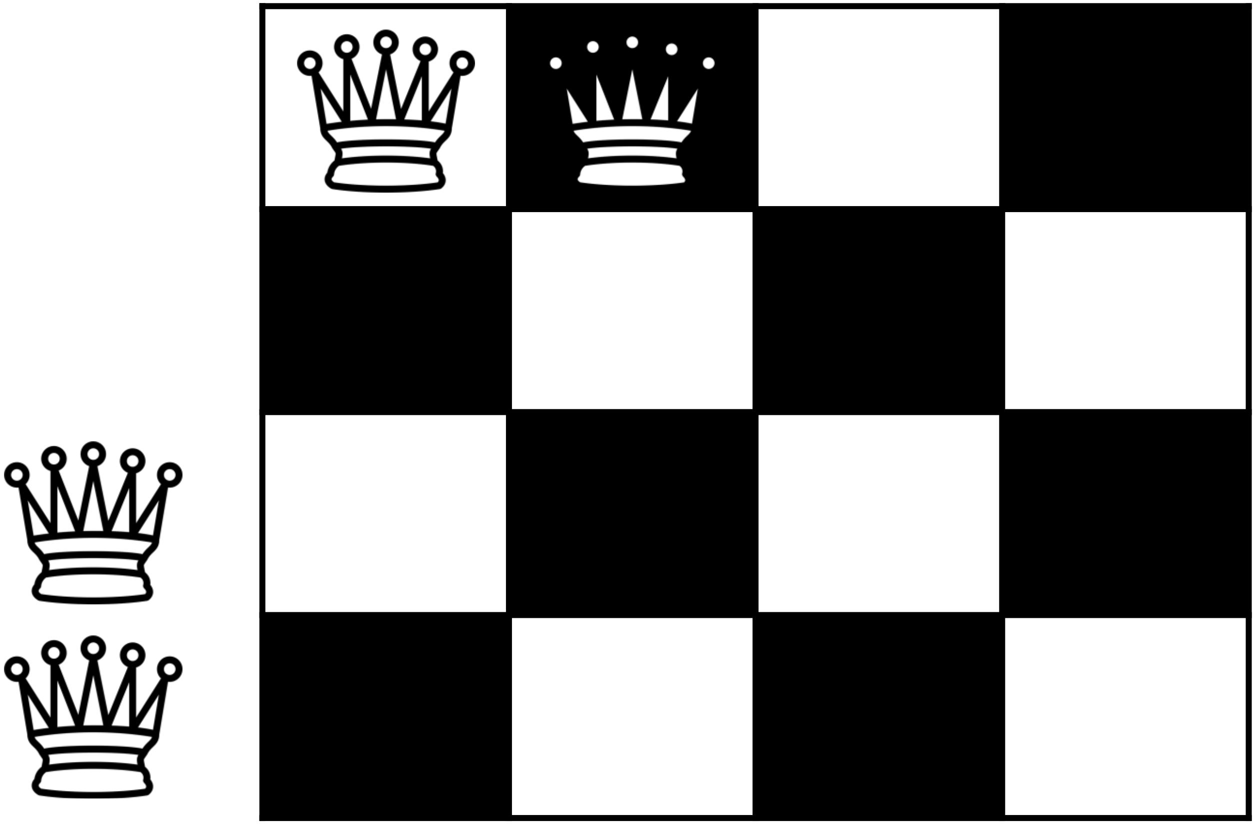
Backtracking (Example)

- Eight Queens Problem (Smaller version)



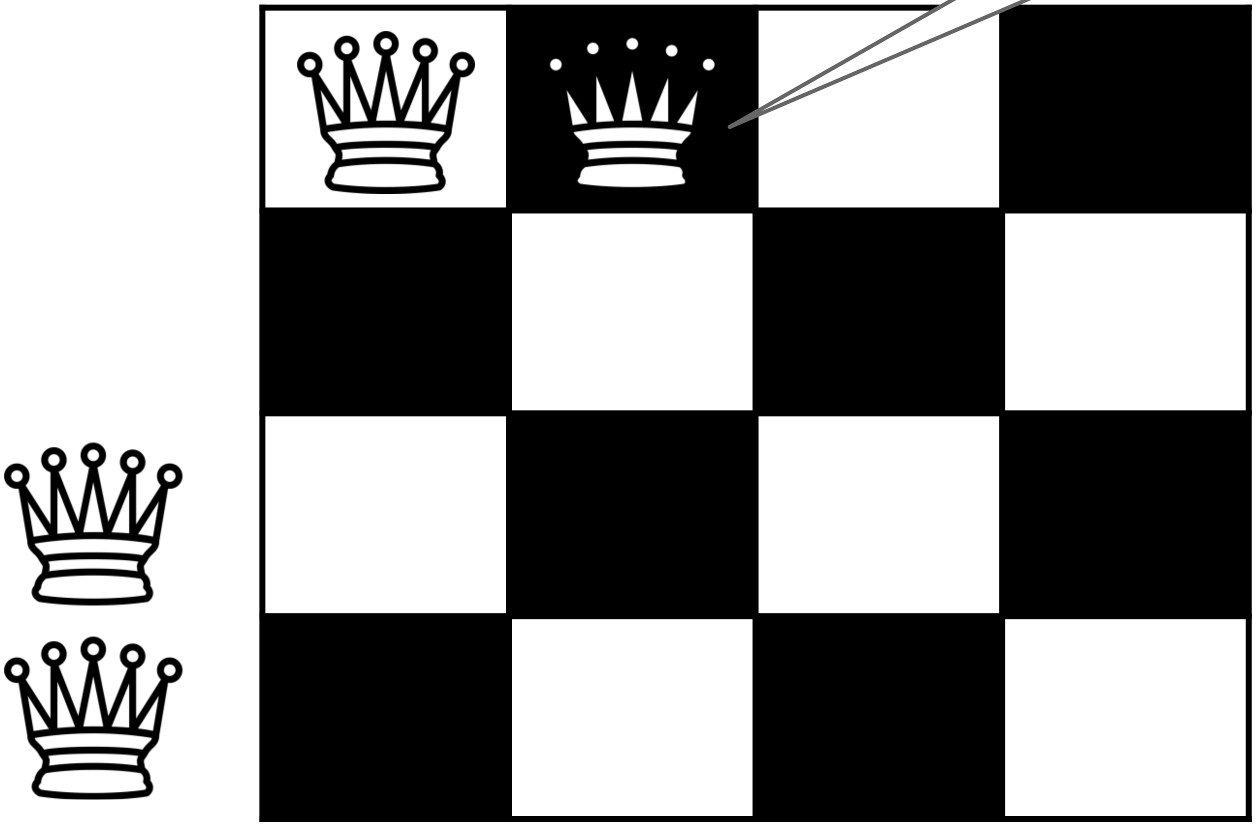
Backtracking (Example)

- ➔ Eight Queens Problem (Smaller version)



Backtracking (Example)

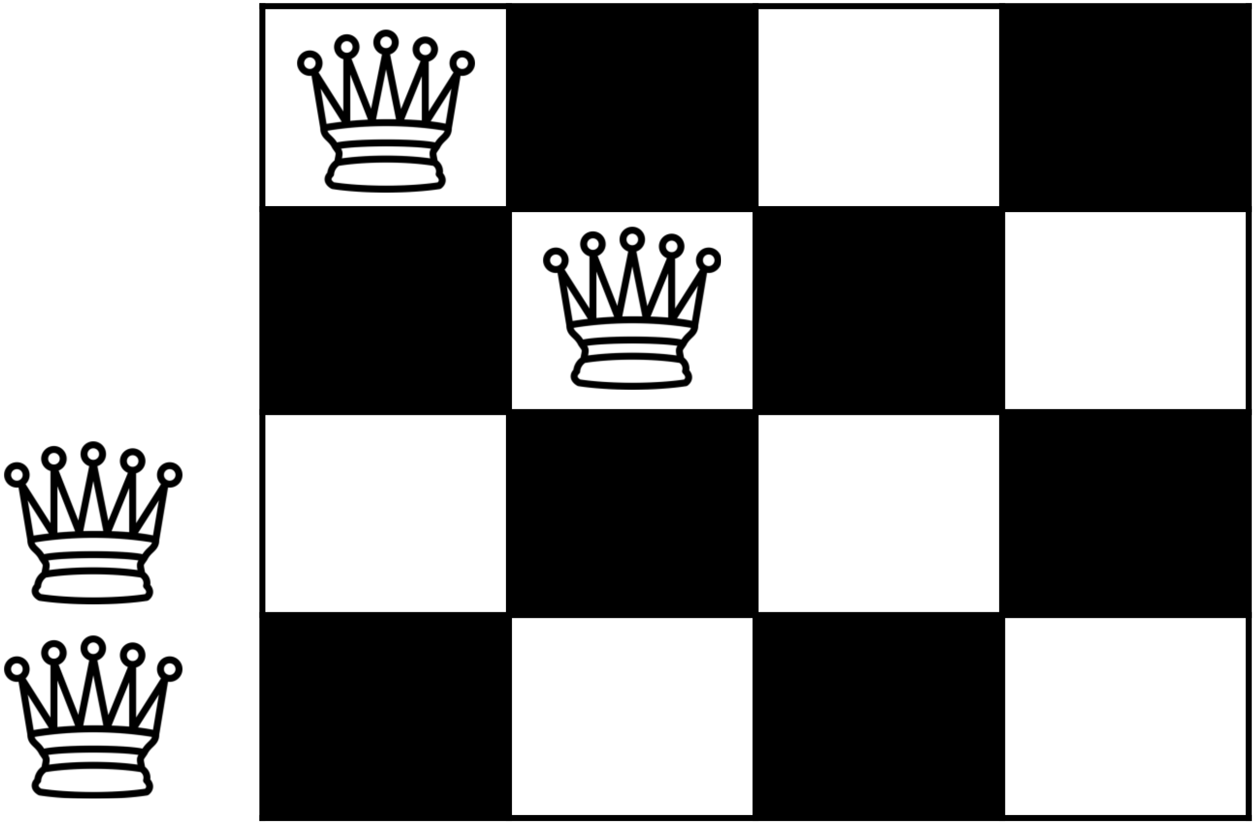
➔ Eight Queens Problem (Smaller version)



That’s no good...backtrack

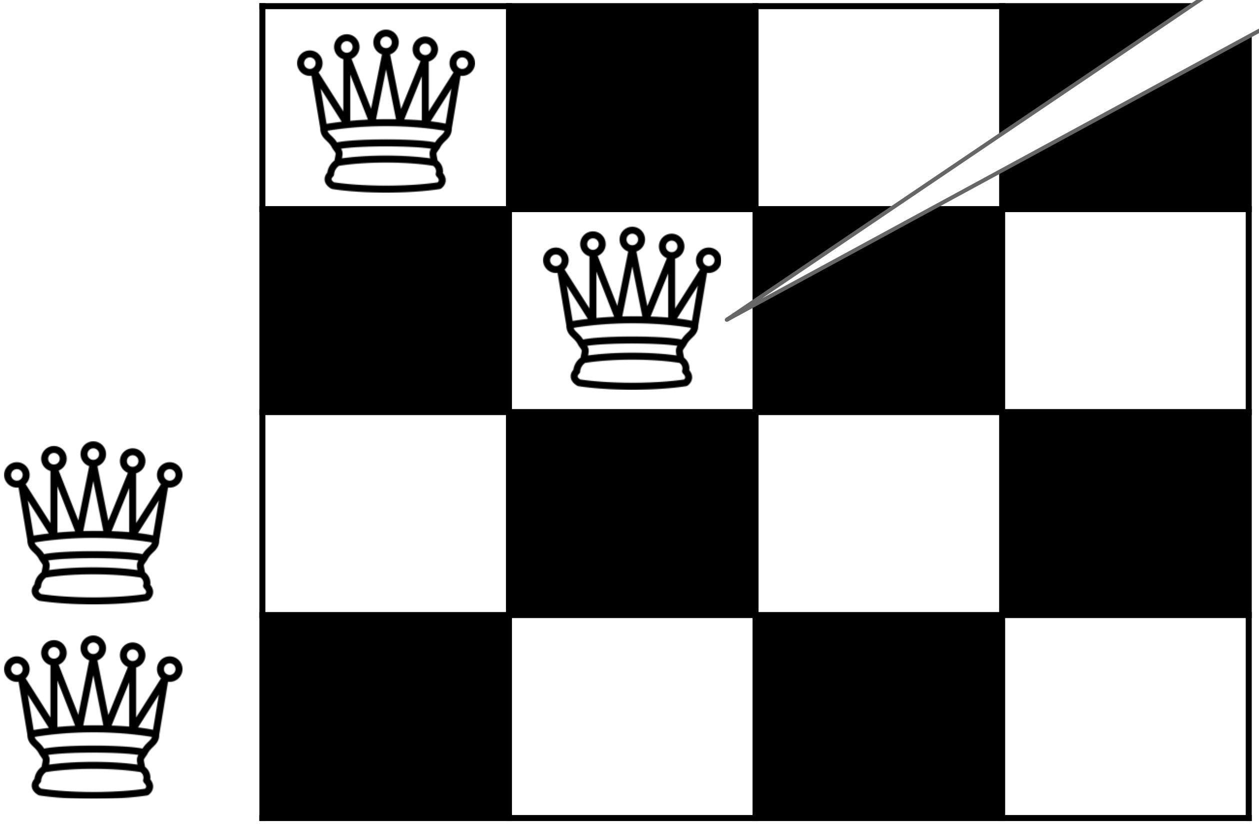
Backtracking (Example)

➔ Eight Queens Problem (Smaller version)



Backtracking (Example)

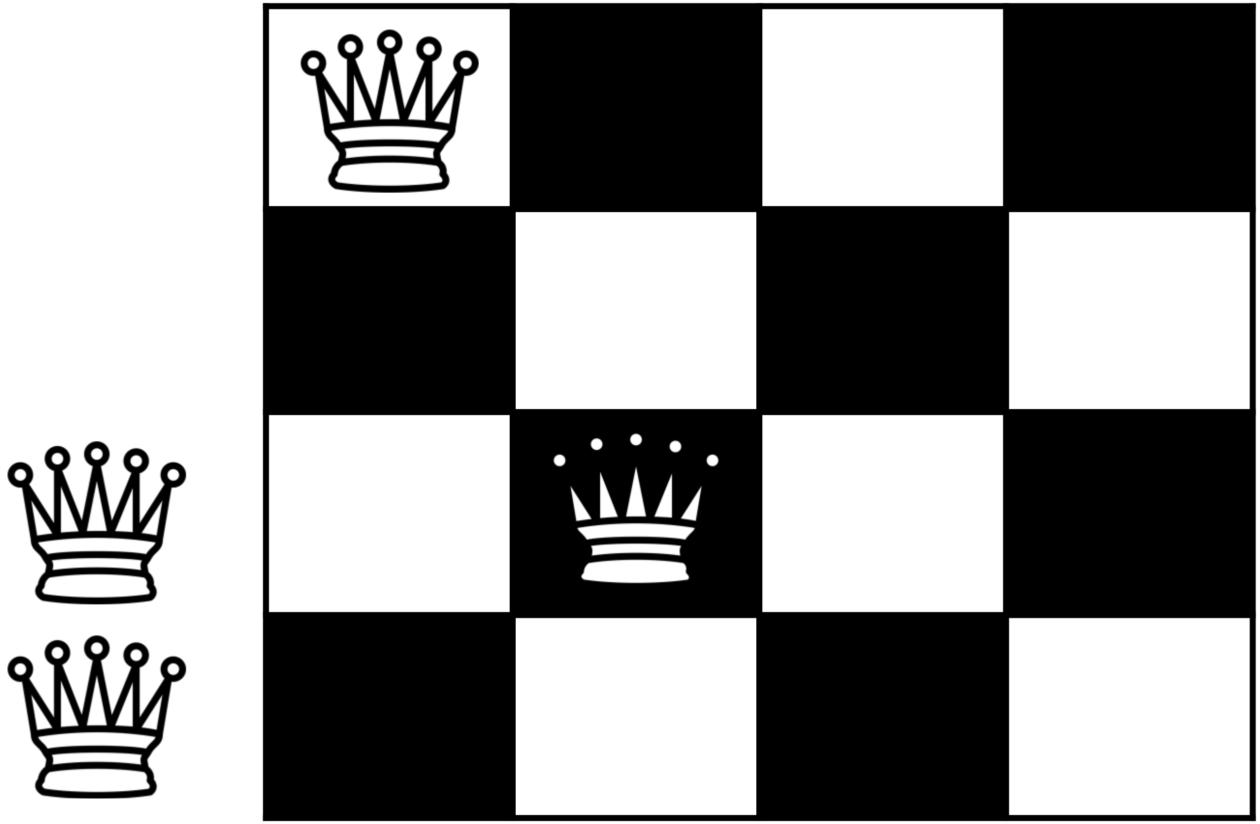
- ➔ Eight Queens Problem (Smaller version)



That's no good...backtrack

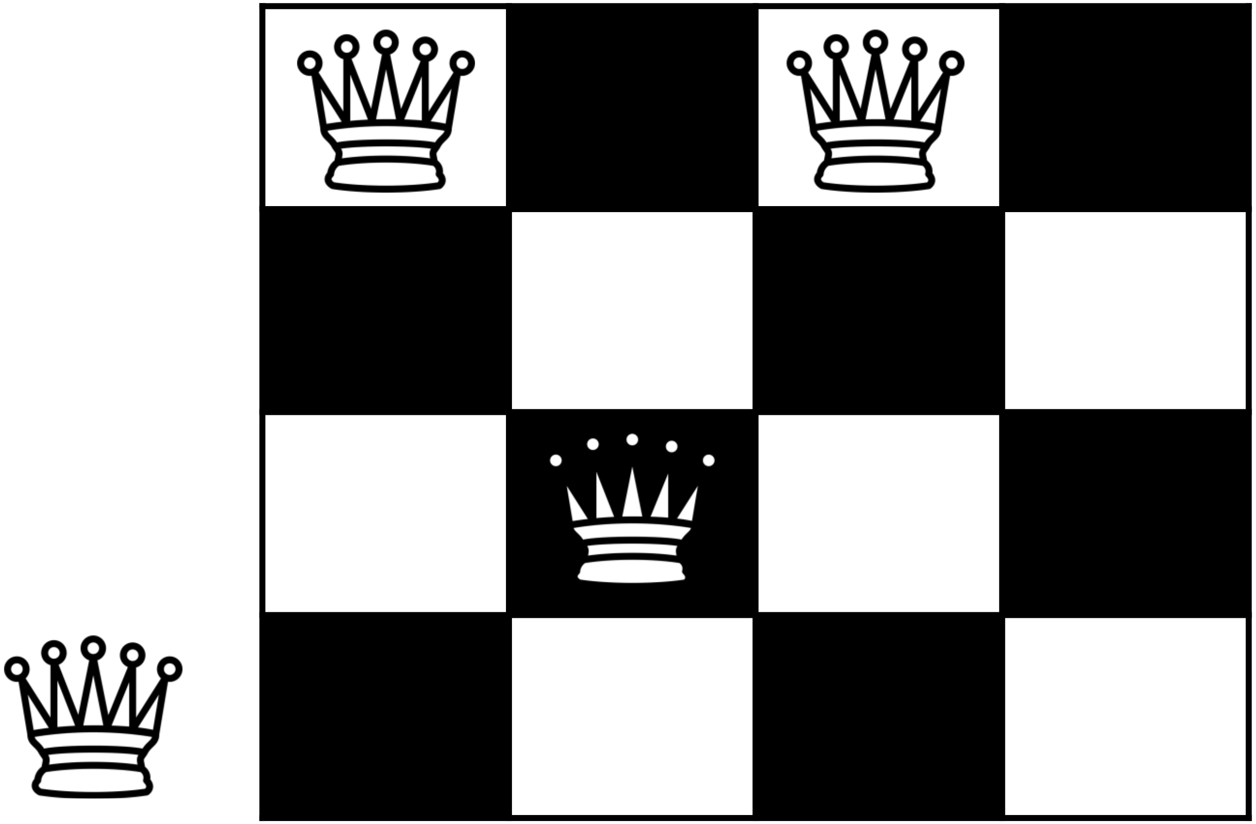
Backtracking (Example)

- ➔ Eight Queens Problem (Smaller version)



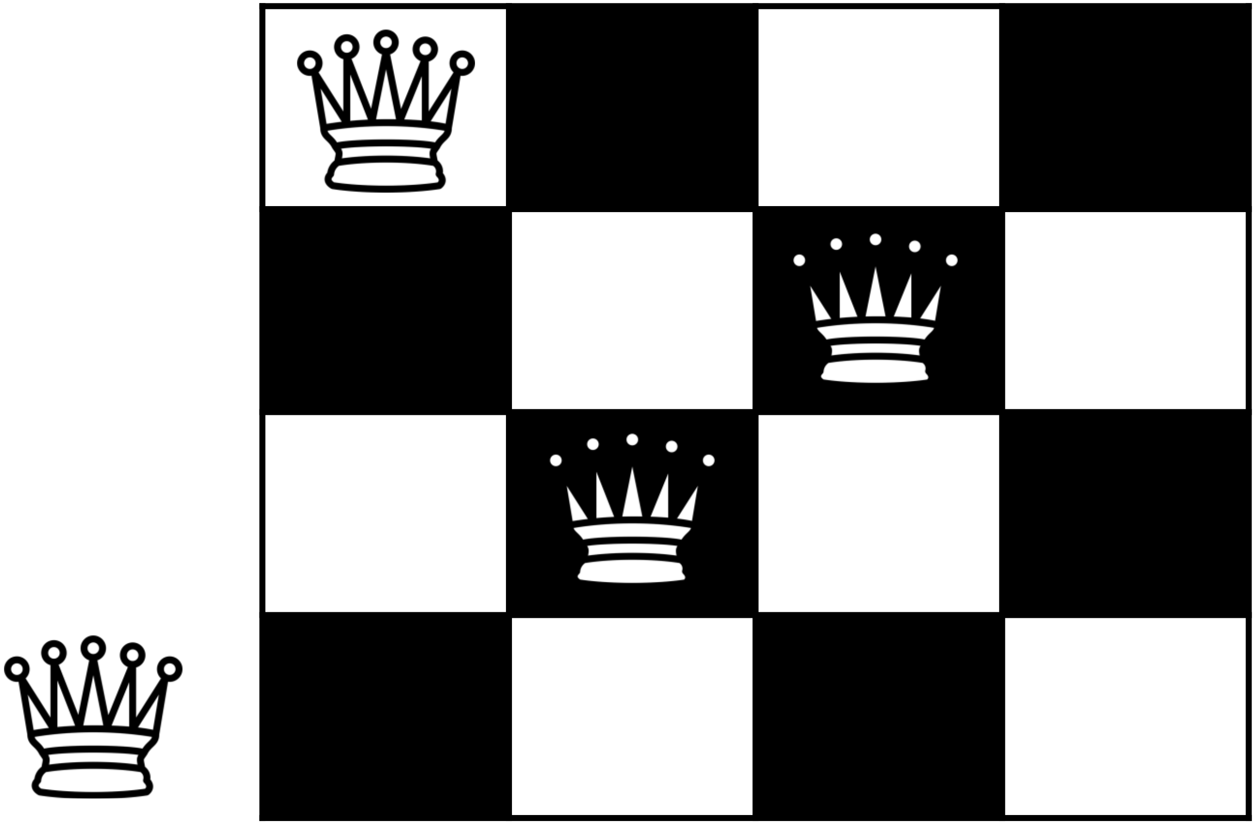
Backtracking (Example)

- ➔ Eight Queens Problem (Smaller version)



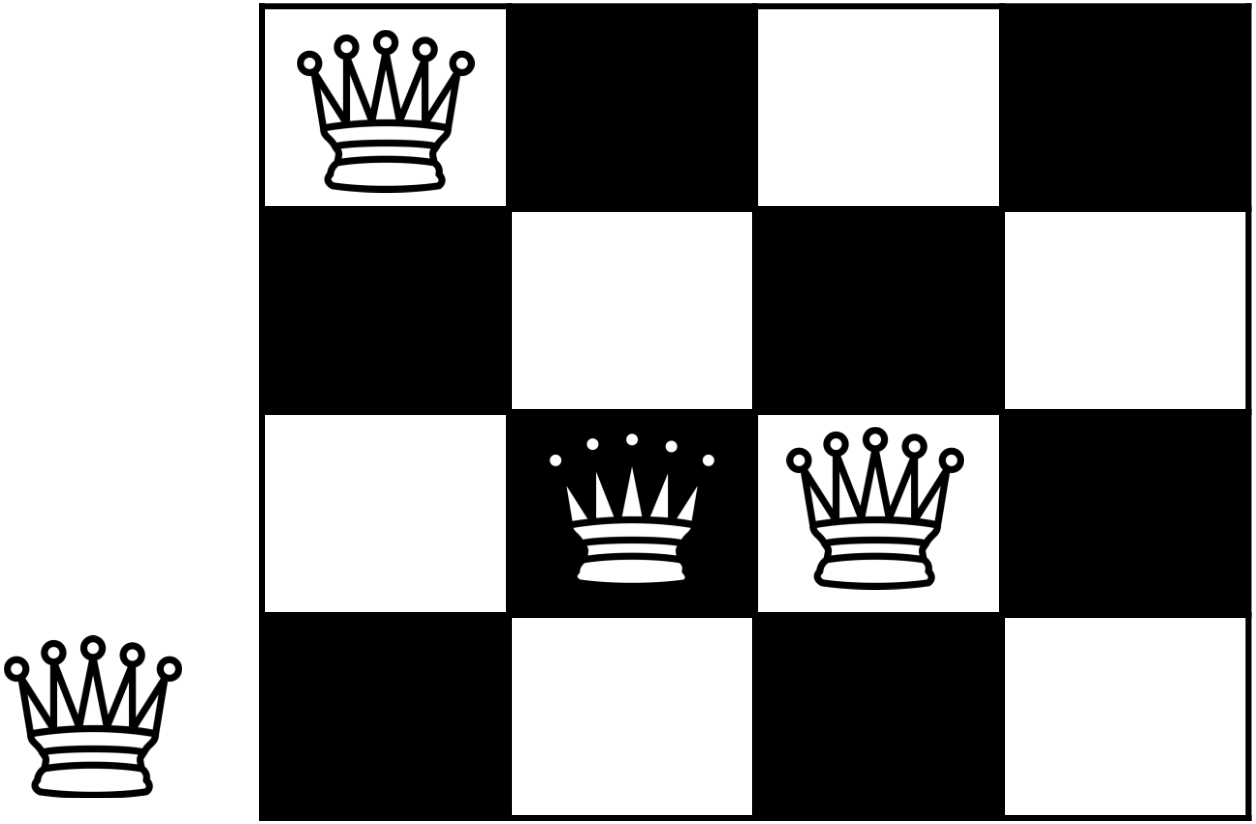
Backtracking (Example)

- ➔ Eight Queens Problem (Smaller version)



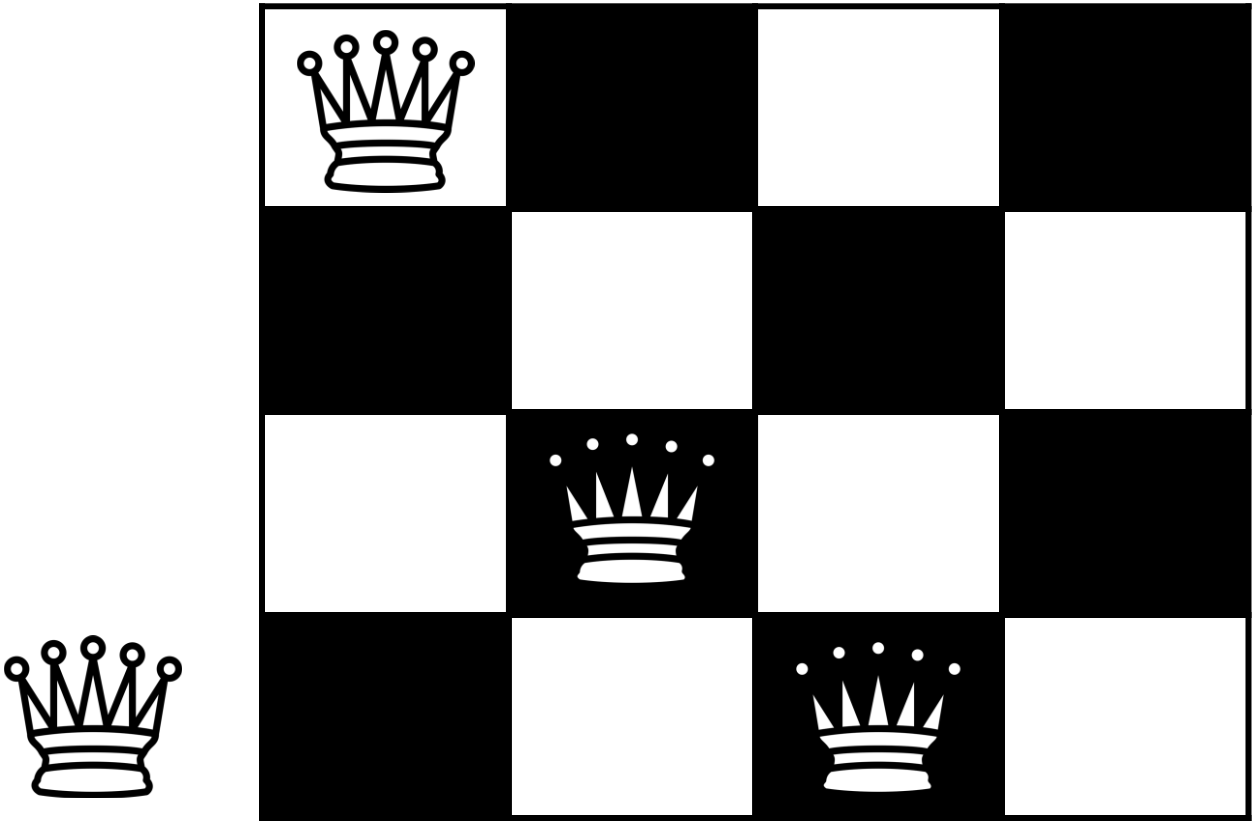
Backtracking (Example)

➔ Eight Queens Problem (Smaller version)



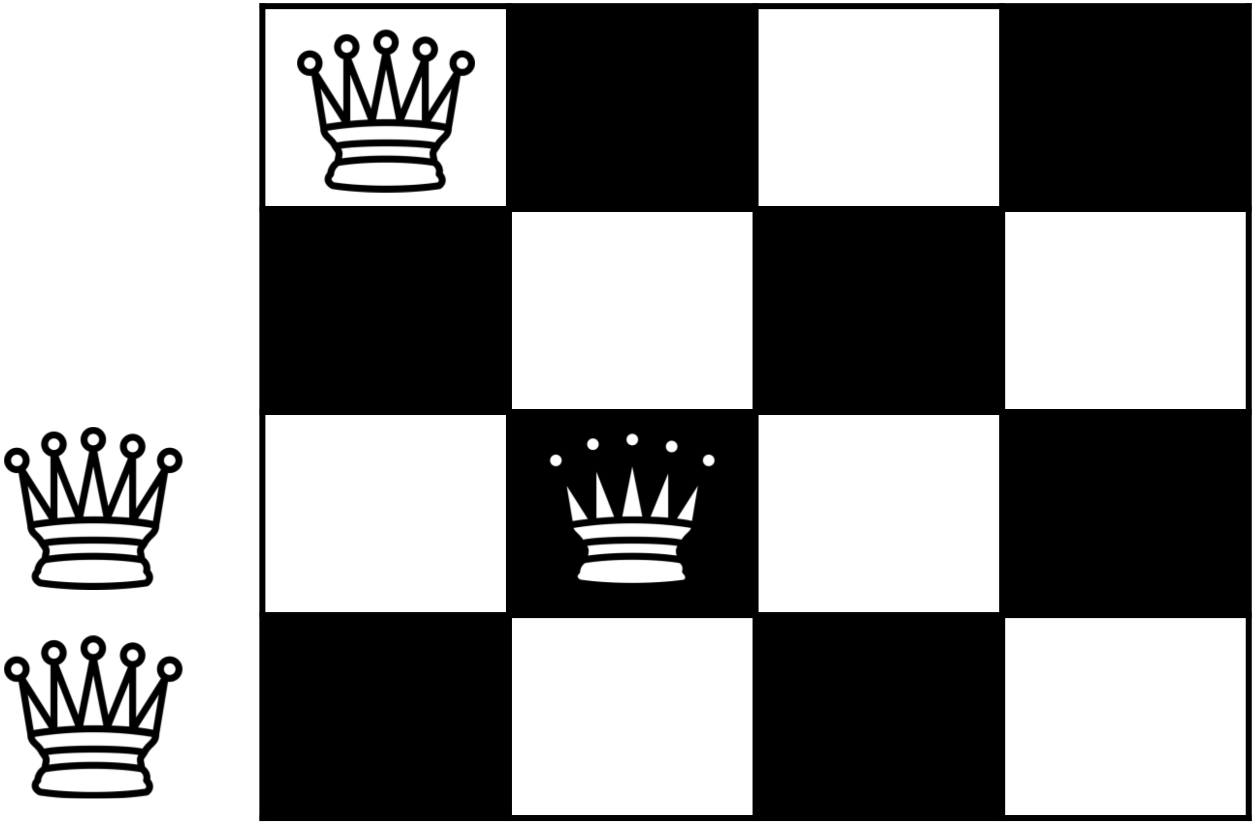
Backtracking (Example)

- ➔ Eight Queens Problem (Smaller version)



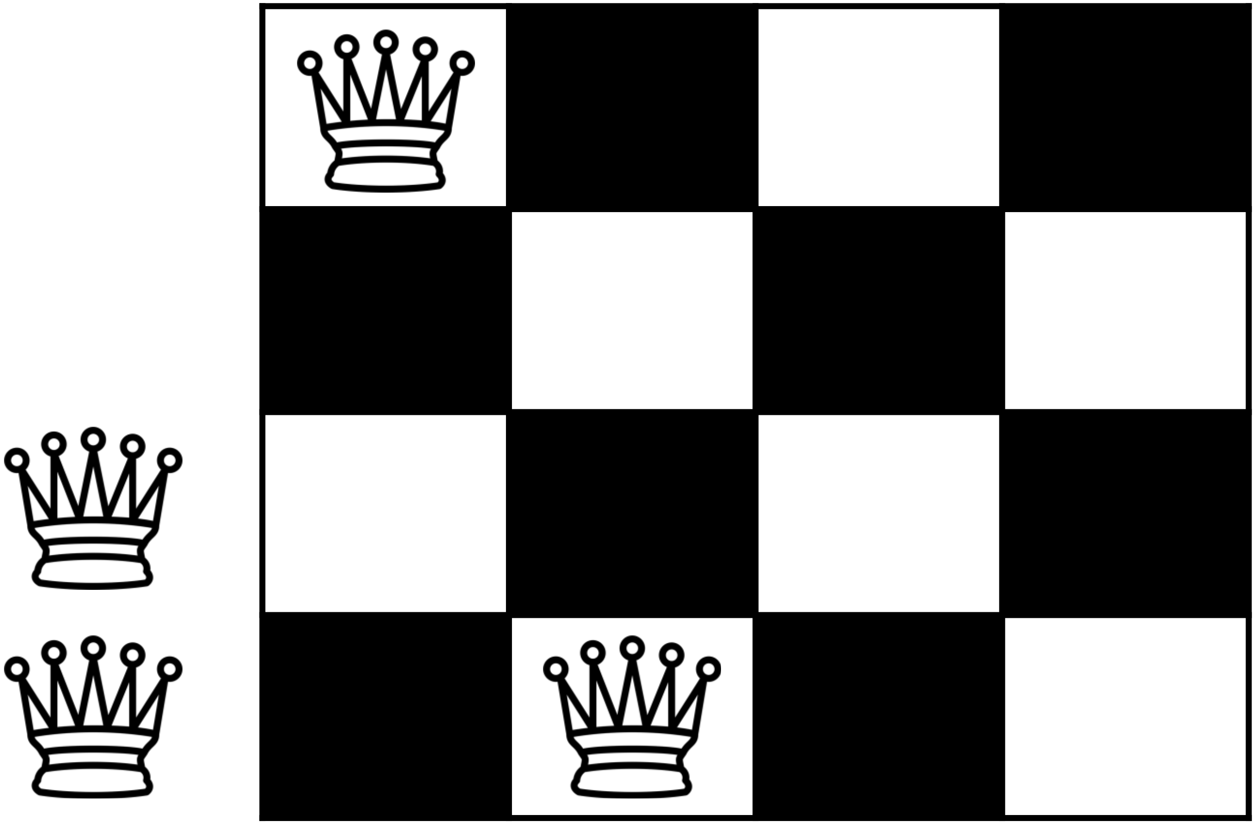
Backtracking (Example)

➔ Eight Queens Problem (Smaller version)



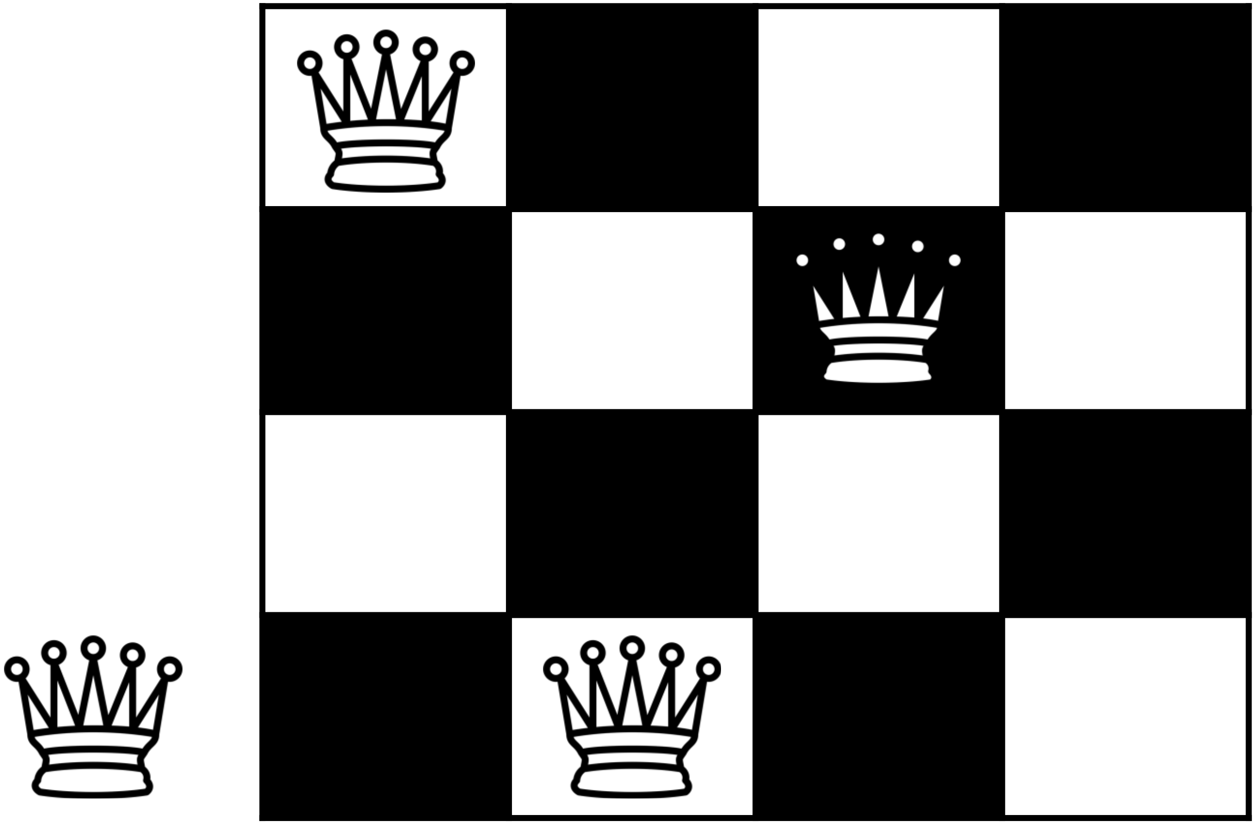
Backtracking (Example)

➔ Eight Queens Problem (Smaller version)



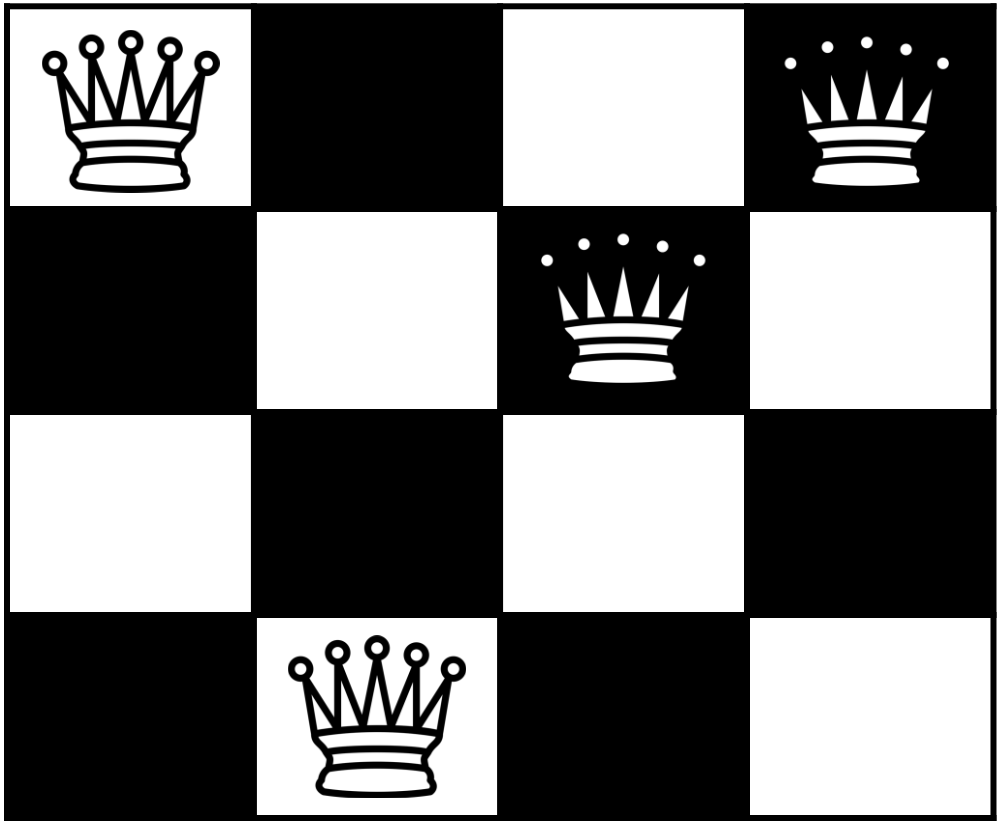
Backtracking (Example)

- ➔ Eight Queens Problem (Smaller version)



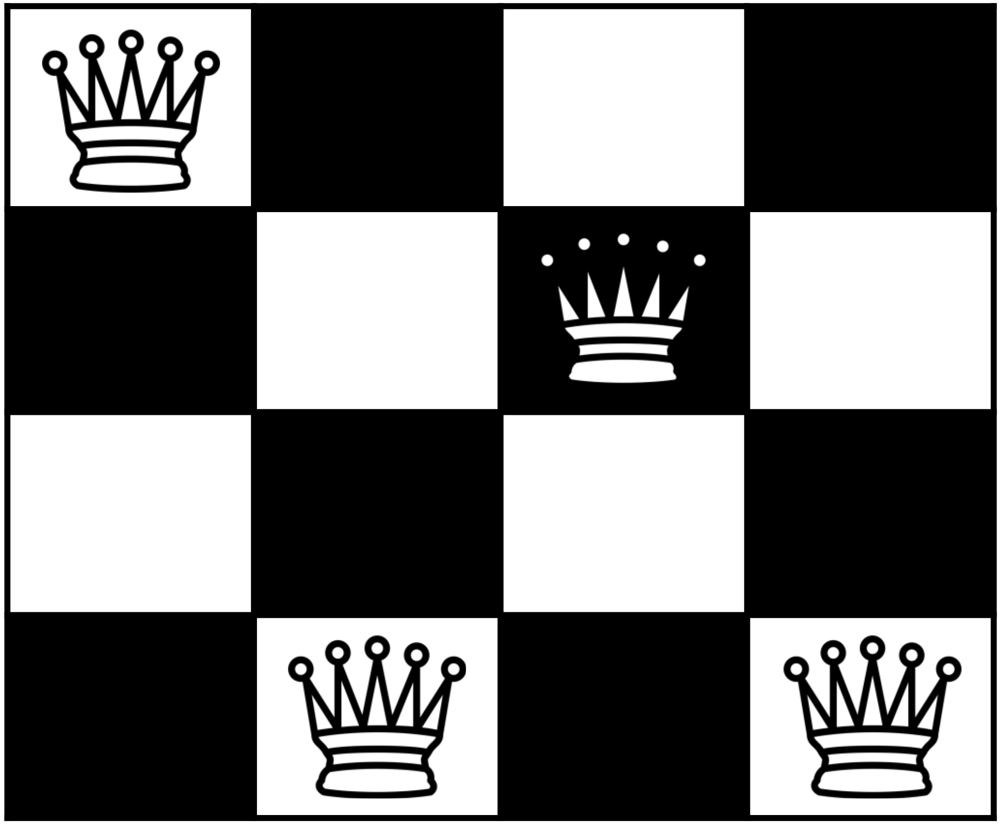
Backtracking (Example)

- ➔ Eight Queens Problem (Smaller version)



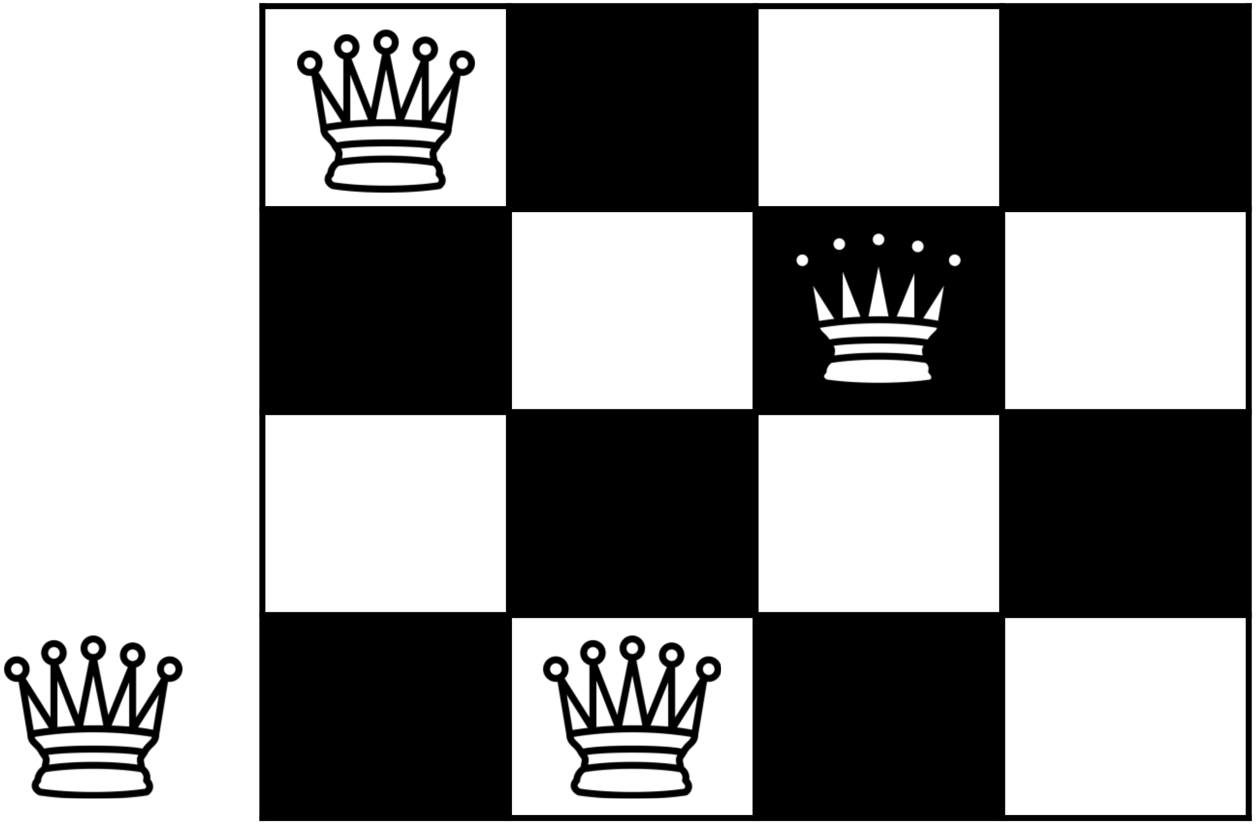
Backtracking (Example)

- ➔ Eight Queens Problem (Smaller version)



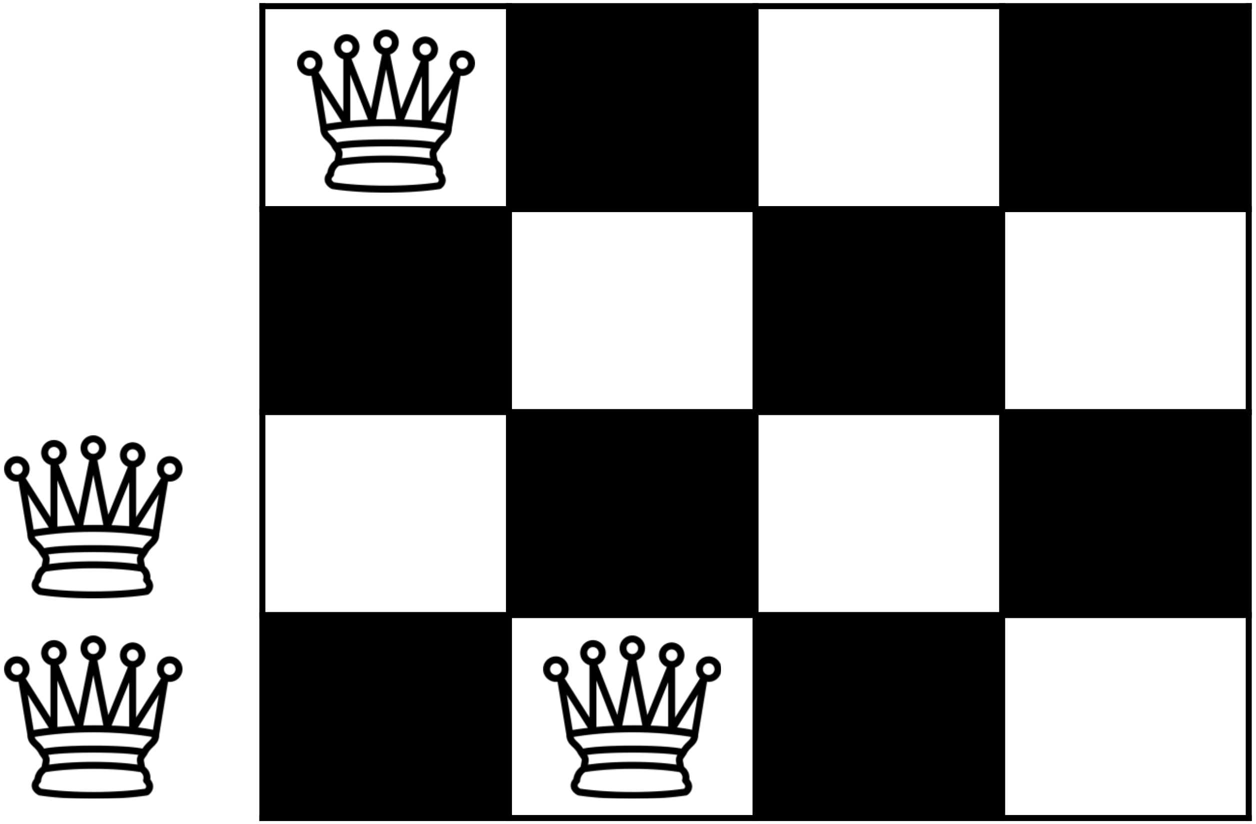
Backtracking (Example)

- ➔ Eight Queens Problem (Smaller version)



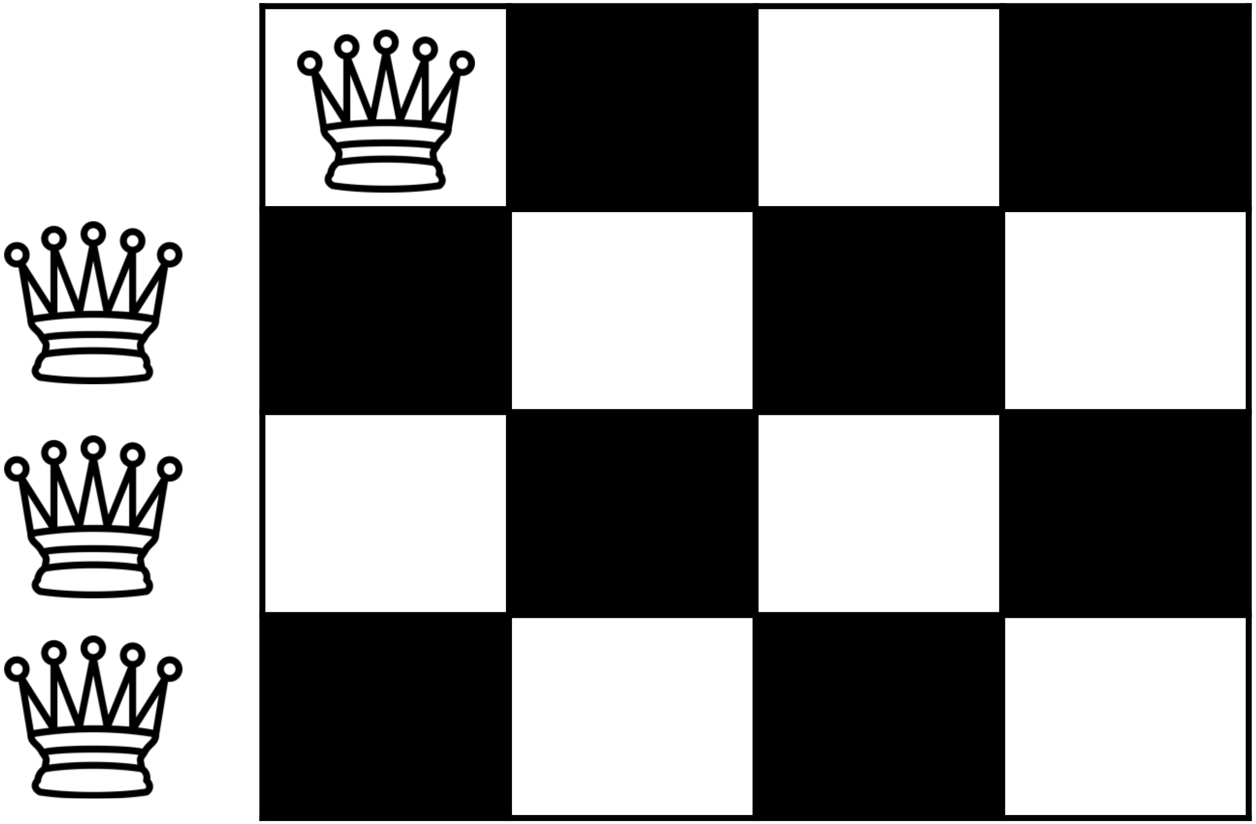
Backtracking (Example)

➔ Eight Queens Problem (Smaller version)



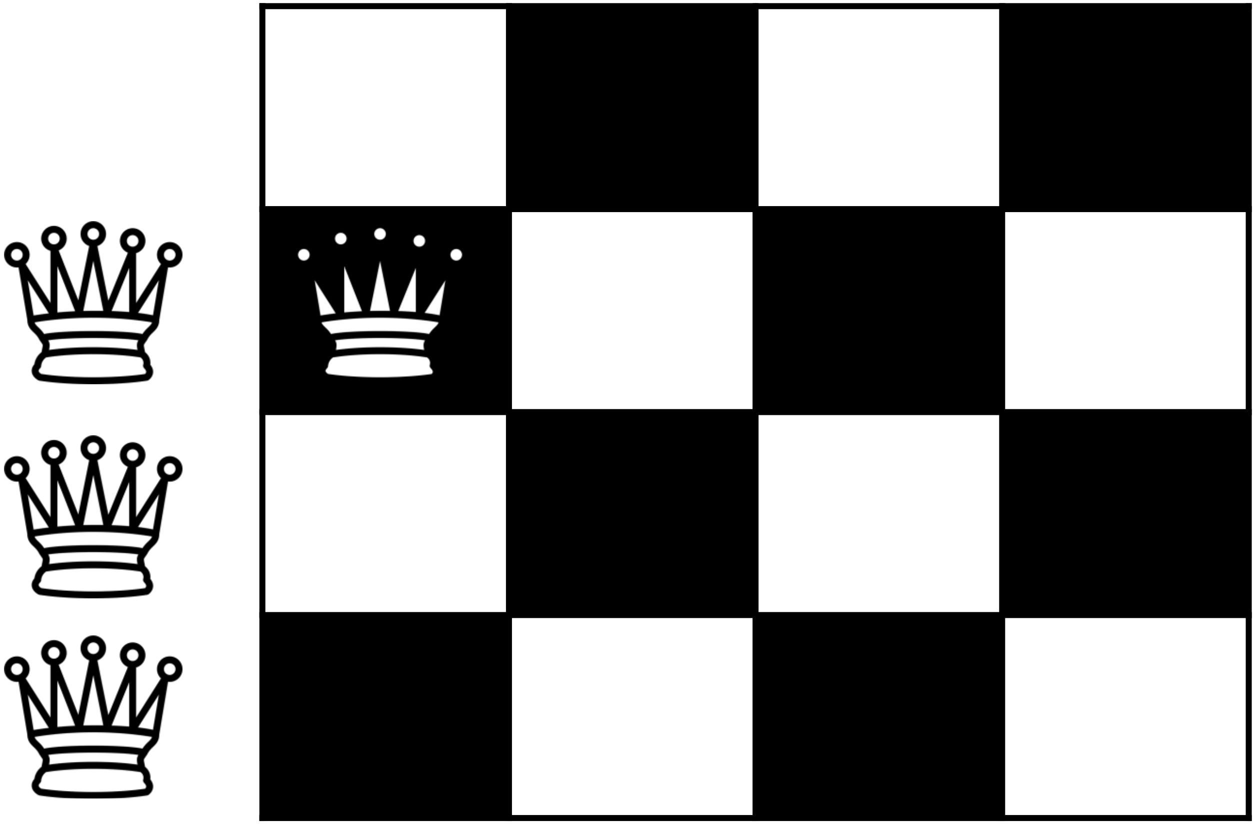
Backtracking (Example)

➔ Eight Queens Problem (Smaller version)



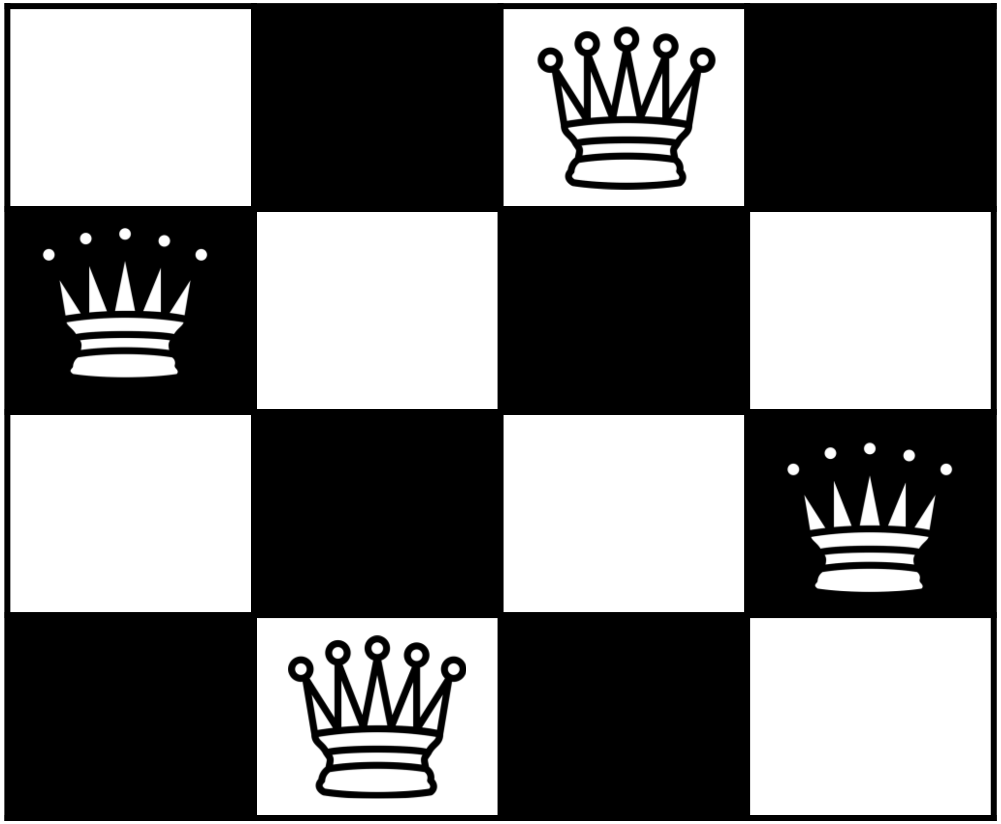
Backtracking (Example)

➔ Eight Queens Problem (Smaller version)



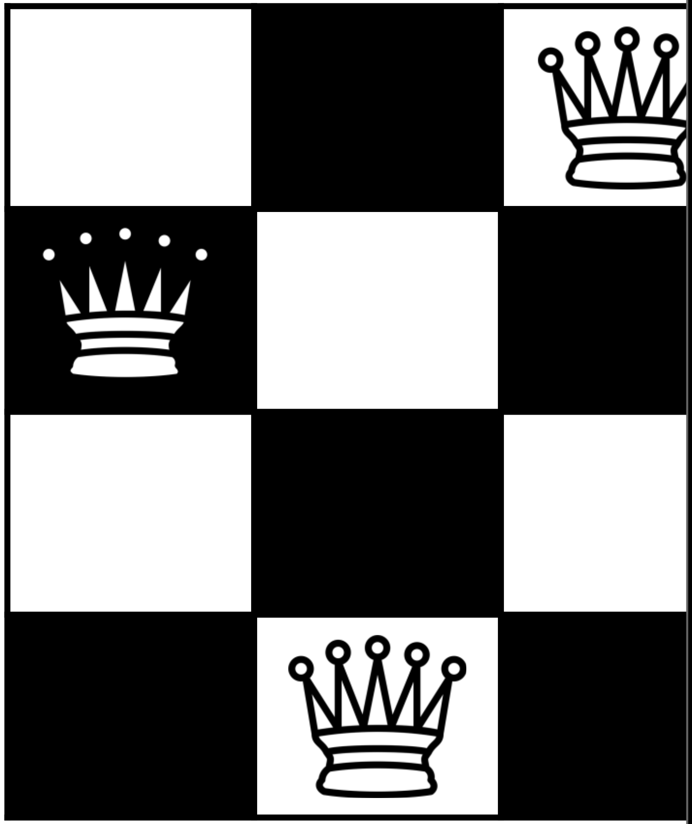
Backtracking (Example)

- ➔ Eight Queens Problem (Smaller version)



Backtracking (Example)

- ➔ Eight Queens Problem (Smaller versio



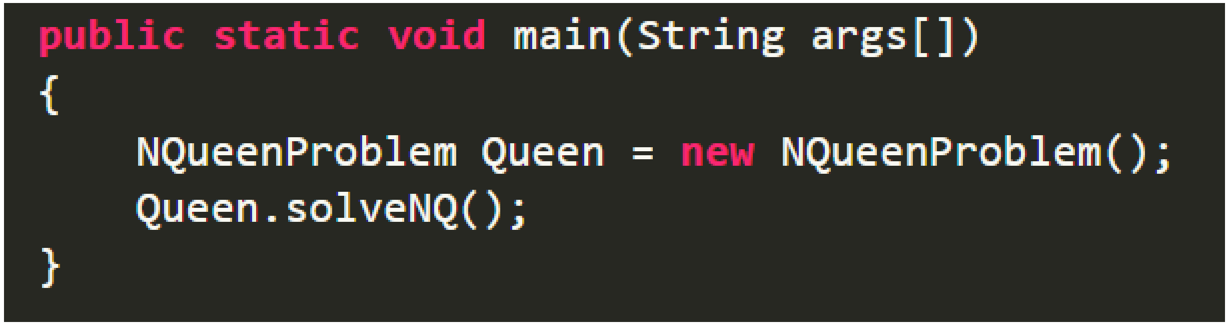
➔ 92 possible solutions.

➔ 12 different solutions.

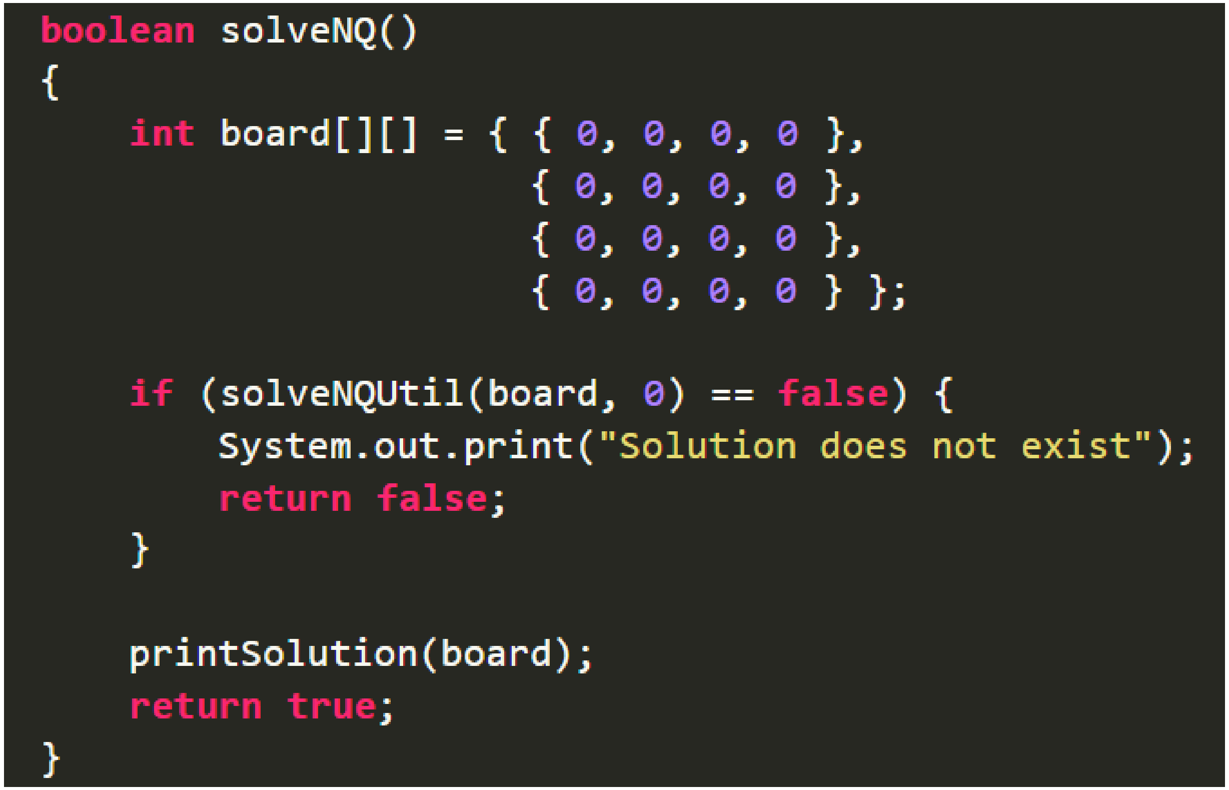
Backtracking (Example)

➔ Eight Queens Problem

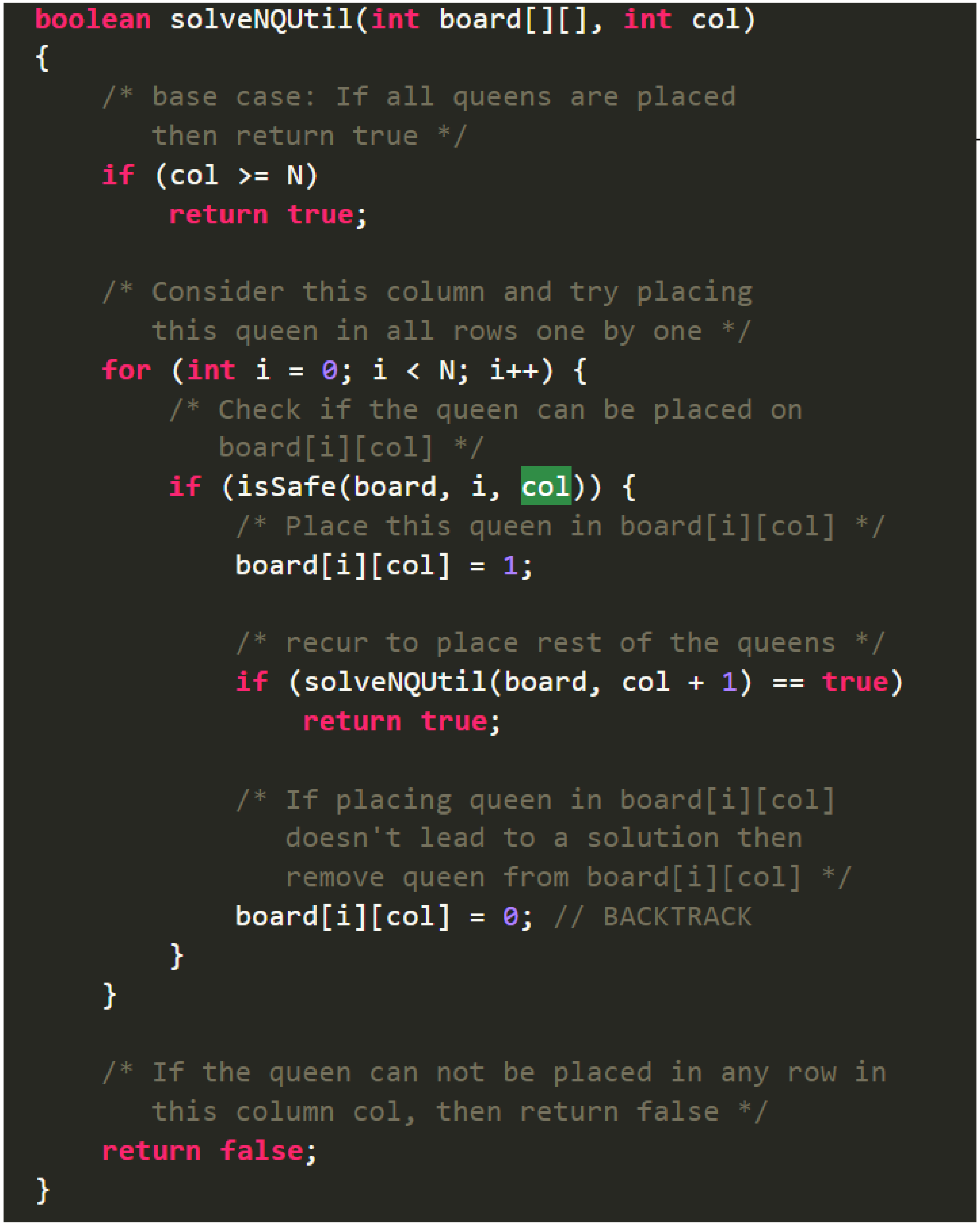
```
static boolean theBoardSolver(int board[][], int col) {
        if (col >= N)
                return true;

        for (int i = 0; i < N; i++) {
                if (toPlaceOrNotToPlace(board, i, col)) {
                        board[i][col] = 1;
                        if (theBoardSolver(board, col +
1))
                                return true;

                        board[i][col] = 0;
                }
        }
        return false;
}
```



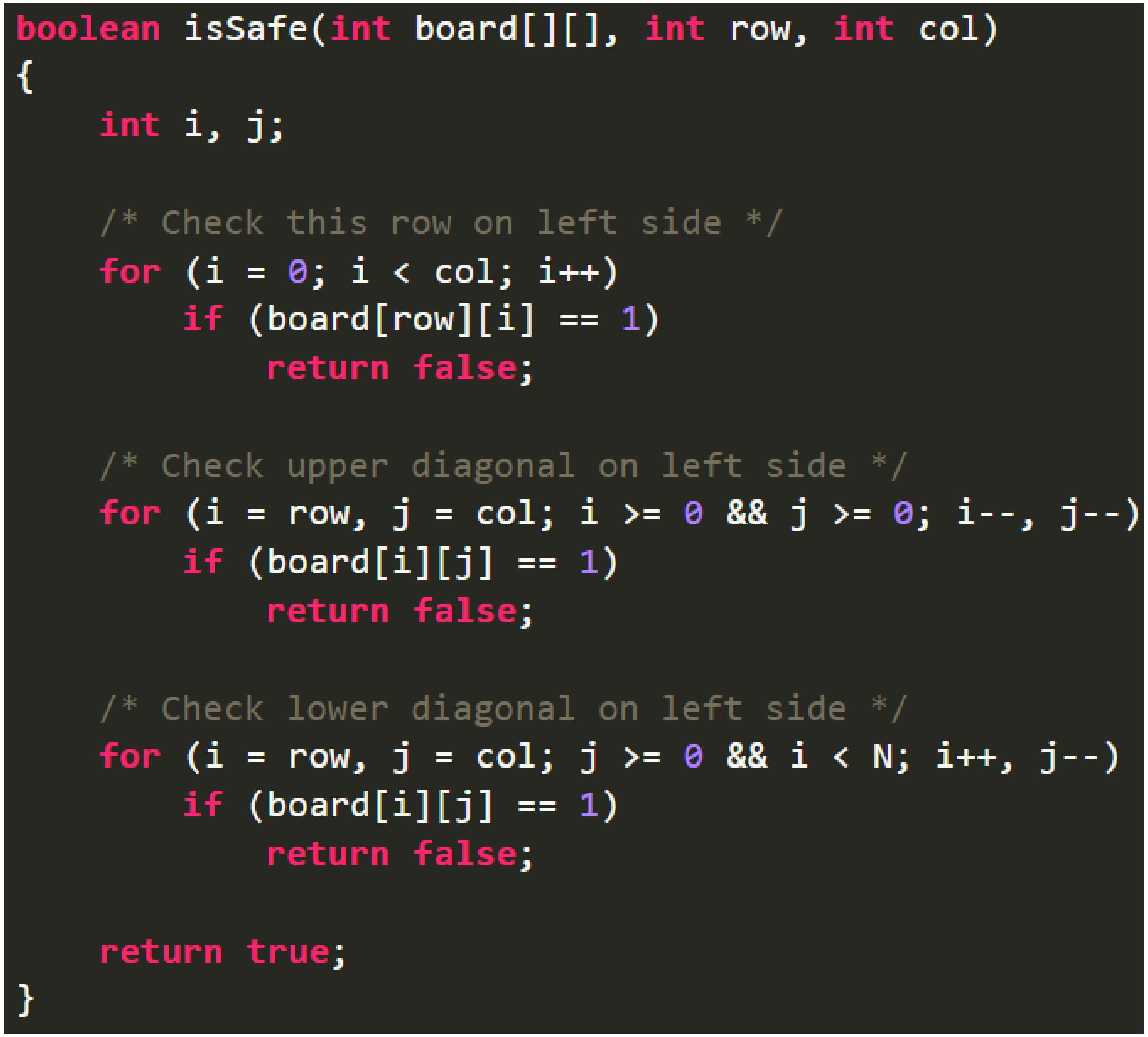
```
public static void main(String args[])
{
    NQueenProblem Queen = new NQueenProblem();
    Queen.solveNQ();
}
```



```
boolean solveNQ()
{
    int board[][] = { { 0, 0, 0, 0 },
                      { 0, 0, 0, 0 },
                      { 0, 0, 0, 0 },
                      { 0, 0, 0, 0 } };

    if (solveNQUtil(board, 0) == false) {
        System.out.print("Solution does not exist");
        return false;
    }

    printSolution(board);
    return true;
}
```



```java
boolean solveNQUtil(int board[][], int col)
{
    /* base case: If all queens are placed
       then return true */
    if (col >= N)
        return true;

    /* Consider this column and try placing
       this queen in all rows one by one */
    for (int i = 0; i < N; i++) {
        /* Check if the queen can be placed on
           board[i][col] */
        if (isSafe(board, i, col)) {
            /* Place this queen in board[i][col] */
            board[i][col] = 1;

            /* recur to place rest of the queens */
            if (solveNQUtil(board, col + 1) == true)
                return true;

            /* If placing queen in board[i][col]
               doesn't lead to a solution then
               remove queen from board[i][col] */
            board[i][col] = 0; // BACKTRACK
        }
    }

    /* If the queen can not be placed in any row in
       this column col, then return false */
    return false;
}
```



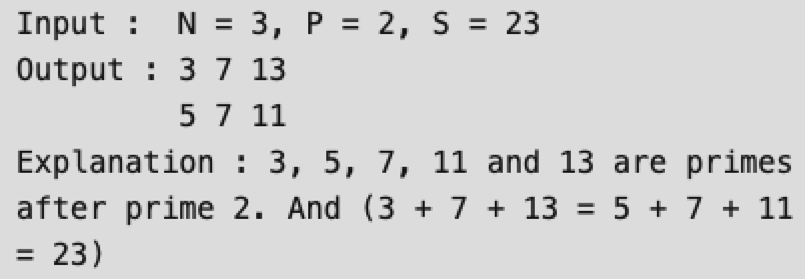
```
boolean isSafe(int board[][], int row, int col)
{
    int i, j;

    /* Check this row on left side */
    for (i = 0; i < col; i++)
        if (board[row][i] == 1)
            return false;

    /* Check upper diagonal on left side */
    for (i = row, j = col; i >= 0 && j >= 0; i--, j--)
        if (board[i][j] == 1)
            return false;

    /* Check lower diagonal on left side */
    for (i = row, j = col; j >= 0 && i < N; i++, j--)
        if (board[i][j] == 1)
            return false;

    return true;
}
```

Backtracking (Example)

➔ Prime numbers after prime P with sum S

DESCRIPTION

Given three numbers: sum S, prime P and N, find all N prime numbers after prime P such that their sum is equal to S



```
Input :  N = 3, P = 2, S = 23
Output : 3 7 13
         5 7 11
Explanation : 3, 5, 7, 11 and 13 are primes
after prime 2. And (3 + 7 + 13 = 5 + 7 + 11
= 23)
```

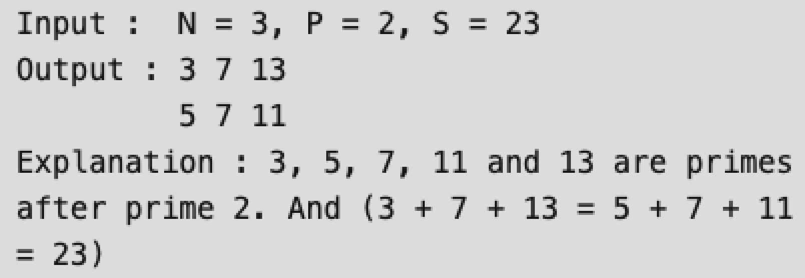
BACKTRACKING APPROACH

Backtracking (Example)

➔ Prime numbers after prime P with sum S

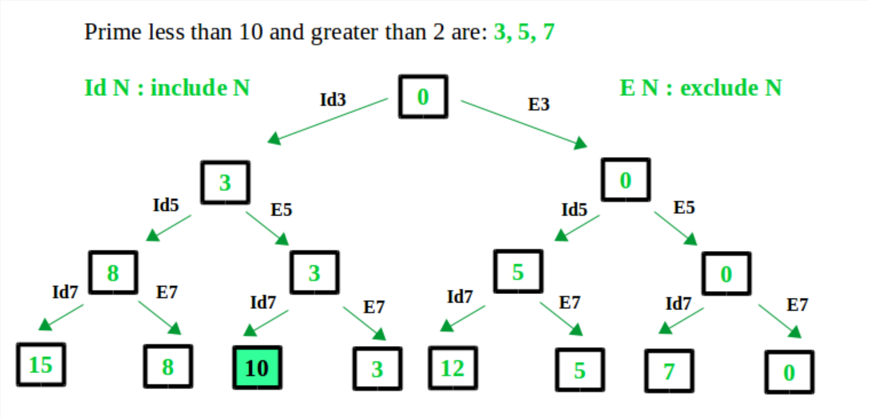
DESCRIPTION

Given three numbers: sum S, prime P and N, find all N prime numbers after prime P such that their sum is equal to S



```
Input :  N = 3, P = 2, S = 23
Output : 3 7 13
         5 7 11
Explanation : 3, 5, 7, 11 and 13 are primes
after prime 2. And (3 + 7 + 13 = 5 + 7 + 11
= 23)
```

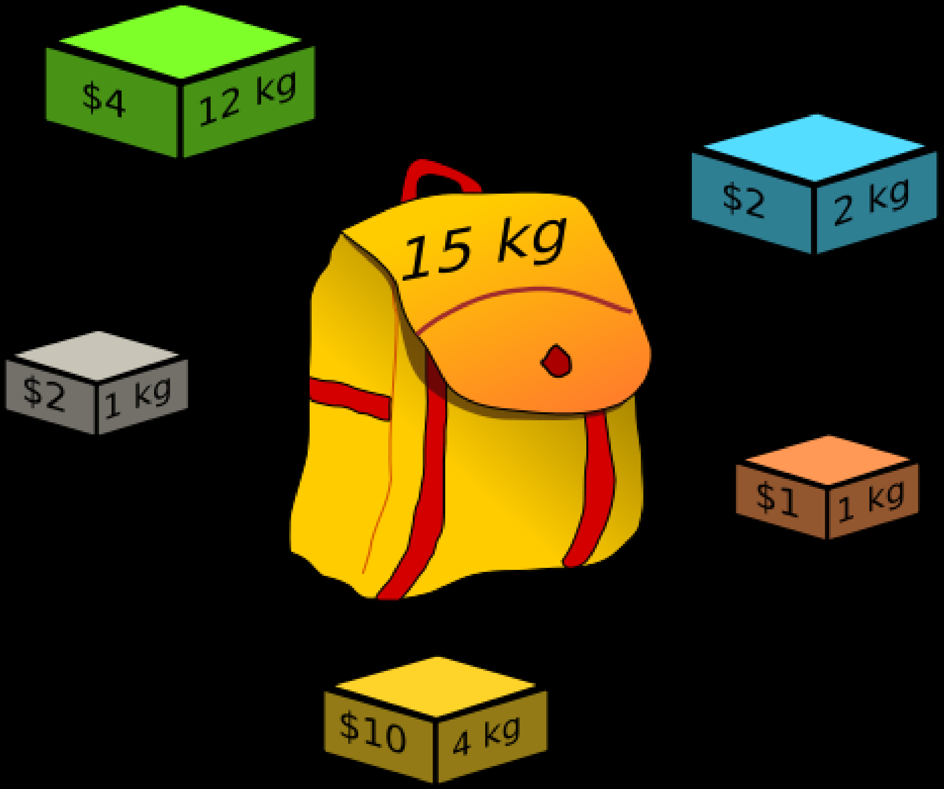
BACKTRACKING APPROACH



Backtracking (Example)

- Other examples
  - **The Knight’s Tour Problem.** The knight is placed on the first block of an empty board and moving according to the rules of chess. Must visit each square exactly once.
  - **Mazes.** Find the way out in a maze.
  - **Coloring Graph Problem.** Given an undirected graph and a number m, determine if the graph can be colored with at most m colors such that no two adjacent vertices of the graph are colored with the same color.

Dynamic Programming



Dynamic Programming (History)

Invented Dynamic Programming in the 1950s.

He was an american applied mathematician.

RICHARD E. BELLMAN

Dynamic Programming

STRATEGY

- → **Divide.** Break the problem into several parts.
- → **Conquer.** Solves the problems in each part **recursively**.
- → **Combine** the solutions of subproblems into an overall solution.

CHARACTERISTICS

- → **Optimal substructure.** Solution of a given optimization problem can be found by combination of optimal solutions to its subproblems.
- → **Overlapping subproblems.** The space of subproblems must be small, you have to **solve the same subproblem over and over**.
- → **The principal difference between DyC and DP is…** later. First, an example.

Dynamic Programming

- → Case of Fibonacci

ral

ne
part

ions
to
.

CHARACTERISTICS

- → **Optimal substructure.** Solution of a given optimization problem can be found by combination of optimal solutions to its subproblems.

- → **Overlapping subproblems.** The space of subproblems must be small, you have to **solve the same subproblem over and over**.

- → **The principal difference between DyC and DP is**… later. First, an example.

Dynamic Programming

- ➔ Case of Fibonacci
- ➔ **Optimal substructure.** Each solution to a sub problem is optimal, we have two base cases that are optimal (exact) fib(0) ad fib(1).

CHARACTERISTICS

- ➔ **Optimal substructure.** Solution of a given optimization problem can be found by combination of optimal solutions to its subproblems.
- ➔ **Overlapping subproblems.** The space of subproblems must be small, you have to **solve the same subproblem over and over**.
- ➔ **The principal difference between DyC and DP is…** later. First, an example.

Dynamic Programming

- ➔ Case of Fibonacci

- ➔ **Overlapping subproblems.** You have to calculate the same problem over and over, for example in the case of fib(10), you have to calculate fib(7) three times.

ral

ne
part

ions
to
.

CHARACTERISTICS

- ➔ **Optimal substructure.** Solution of a given optimization problem can be found by combination of optimal solutions to its subproblems.

- ➔ **Overlapping subproblems.** The space of subproblems must be small, you have to **solve the same subproblem over and over**.

- ➔ **The principal difference between DyC and DP is…** later. First, an example.

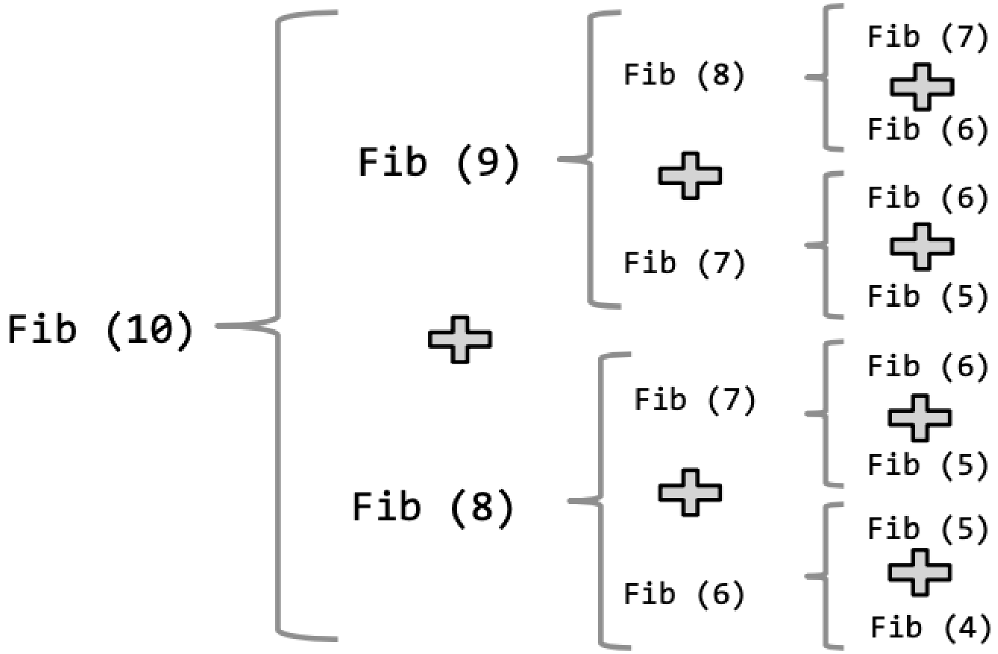
Dynamic Programming (Example)

➔ Fibonacci

$$\mathrm{Fib}(n) \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ \mathrm{Fib}\ (n-1) + \mathrm{Fib}\ (n-2) & \text{if } n > 1 \end{cases}$$

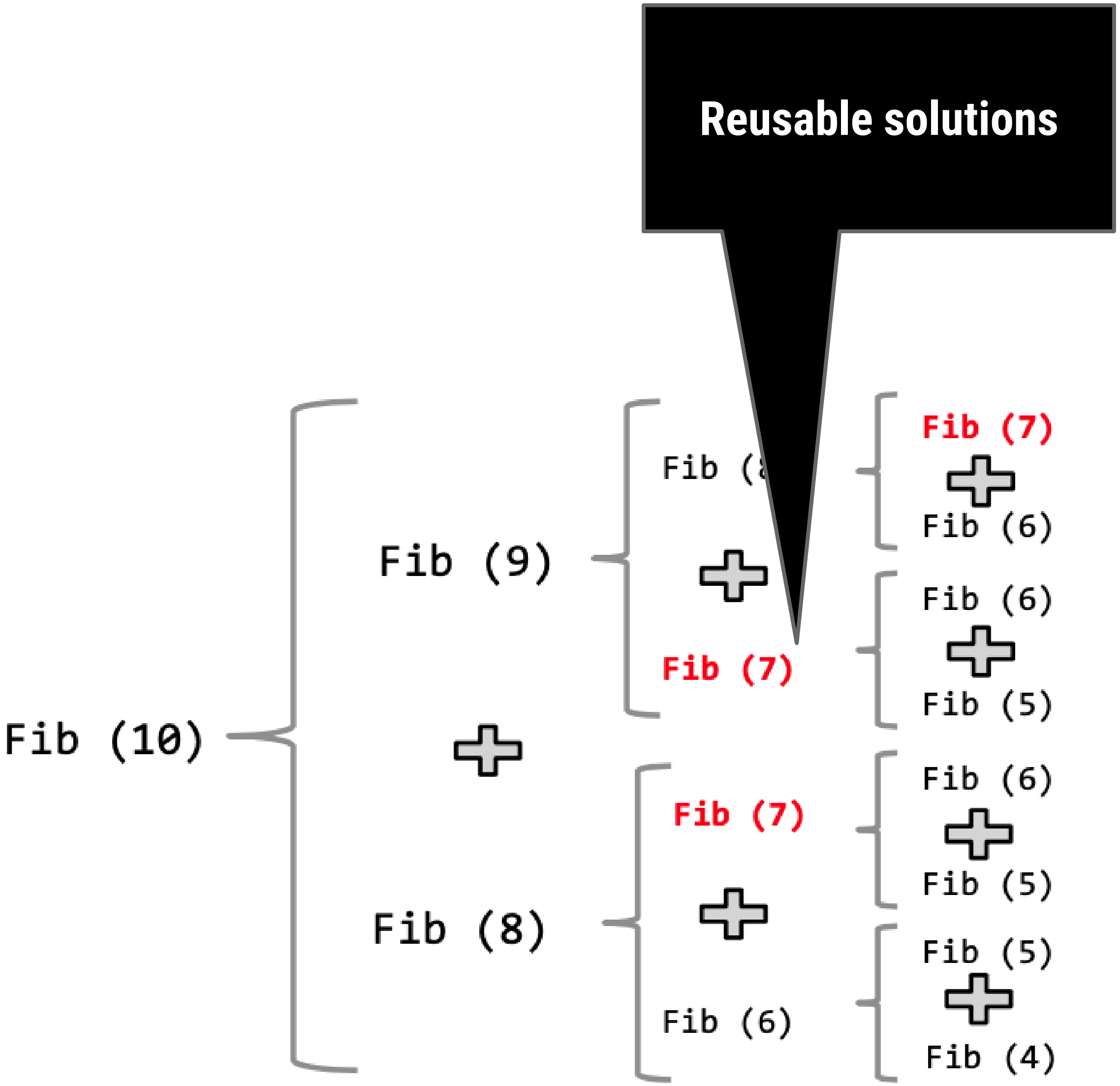
Dynamic Programming (Example)

- ➔ Fibonacci



Dynamic Programming (Example)

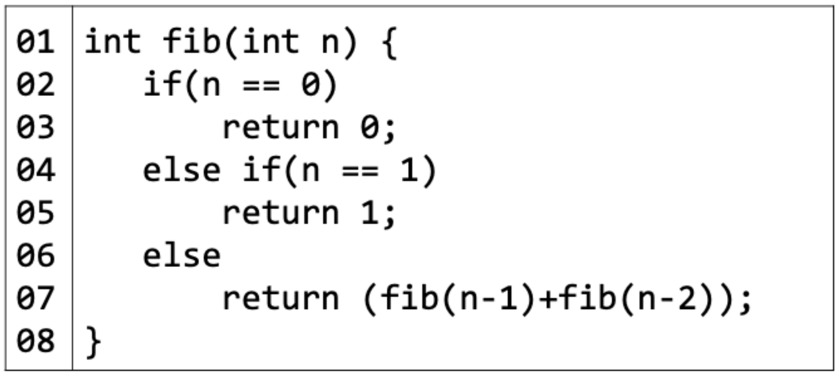
- → Fibonacci



Dynamic Programming (Example)

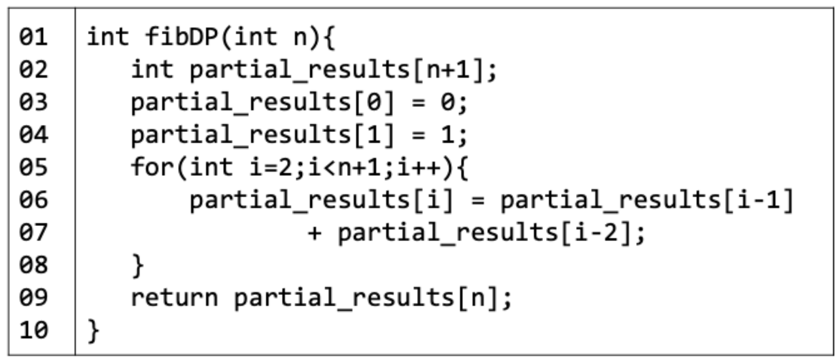
➔ Fibonacci

DIVIDE AND CONQUER APPROACH



```
int fib(int n) {
   if(n == 0)
       return 0;
   else if(n == 1)
       return 1;
   else
       return (fib(n-1)+fib(n-2));
}
```

DYNAMIC PROGRAMMING APPROACH

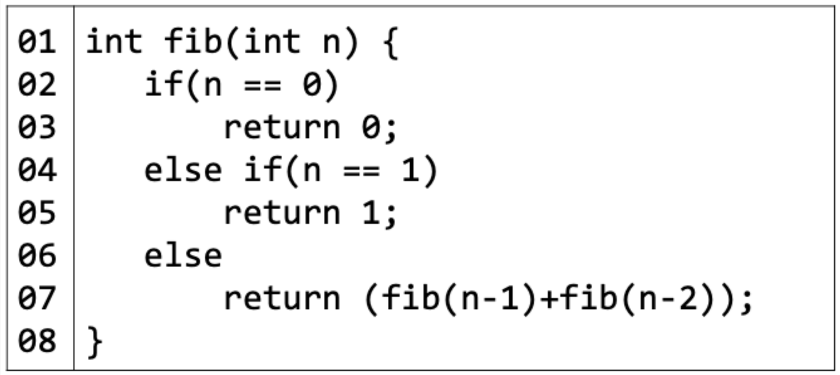


```
int fibDP(int n){
   int partial_results[n+1];
   partial_results[0] = 0;
   partial_results[1] = 1;
   for(int i=2;i<n+1;i++){
       partial_results[i] = partial_results[i-1]
               + partial_results[i-2];
   }
   return partial_results[n];
}
```

Dynamic Programming (Example)

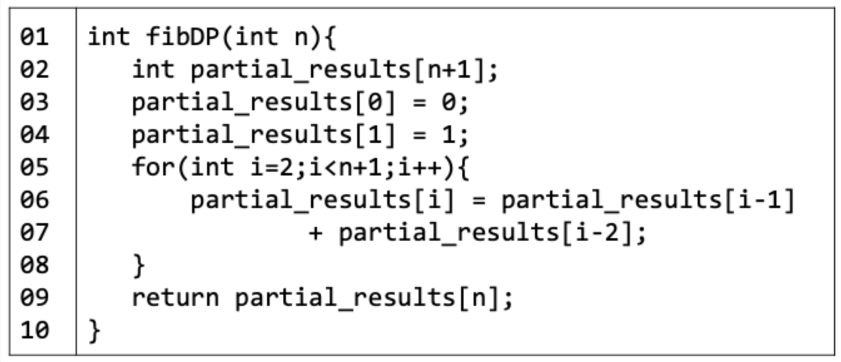
➔ Fibonacci

DIVIDE AND CONQUER APPROACH



```
int fib(int n) {
   if(n == 0)
       return 0;
   else if(n == 1)
       return 1;
   else
       return (fib(n-1)+fib(n-2));
}
```

DYNAMIC PROGRAMMING APPROACH



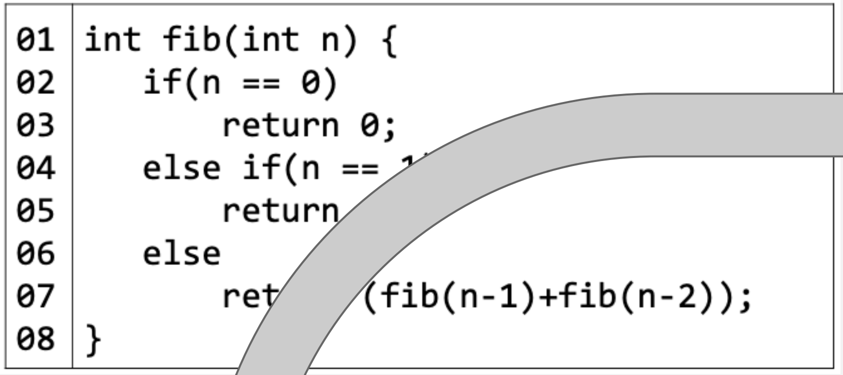
```
int fibDP(int n){
   int partial_results[n+1];
   partial_results[0] = 0;
   partial_results[1] = 1;
   for(int i=2;i<n+1;i++){
       partial_results[i] = partial_results[i-1]
               + partial_results[i-2];
   }
   return partial_results[n];
}
```

Difference?

Dynamic Programming (Example)

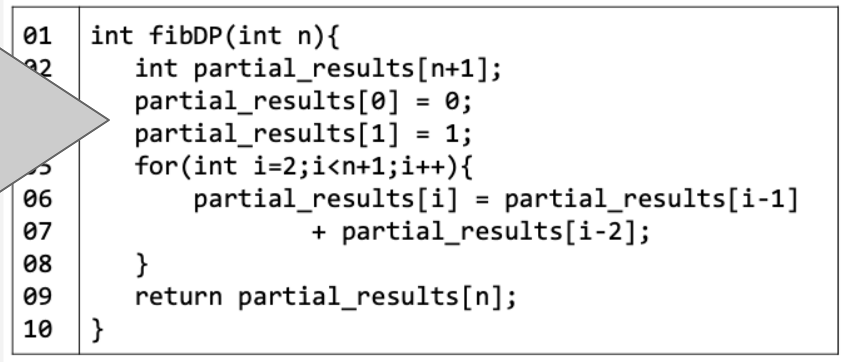
➔ Fibonacci

DIVIDE AND CONQUER APPROACH



```
int fib(int n) {
   if(n == 0)
       return 0;
   else if(n == 1
       return
   else
       ret    (fib(n-1)+fib(n-2));
}
```

DYNAMIC PROGRAMMING APPROACH



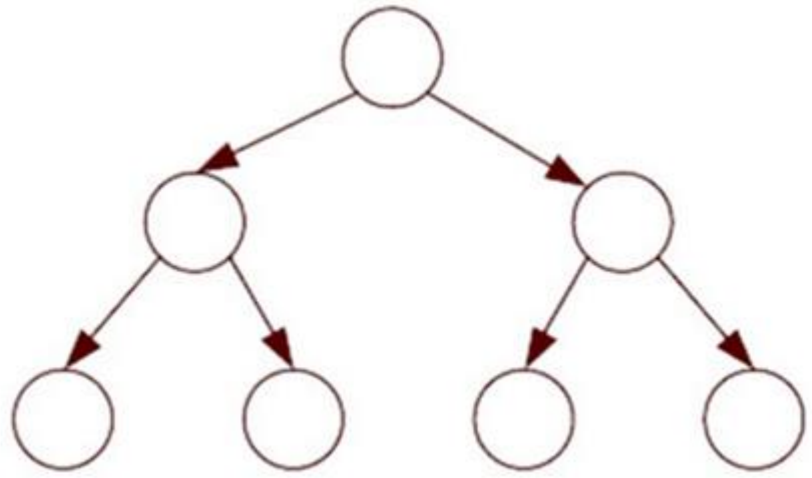
```
int fibDP(int n){
   int partial_results[n+1];
   partial_results[0] = 0;
   partial_results[1] = 1;
   for(int i=2;i<n+1;i++){
       partial_results[i] = partial_results[i-1]
               + partial_results[i-2];
   }
   return partial_results[n];
}
```

Memoization. After computing a solution to a subproblem, store it in a table. Subsequent calls check the table to avoid redoing work

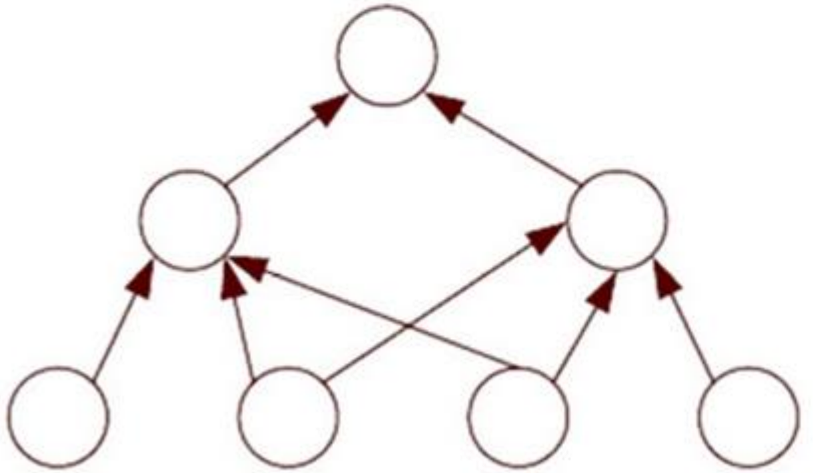
Dynamic Programming

- ➔ Difference with Divide and Conquer

Divide and Conquer



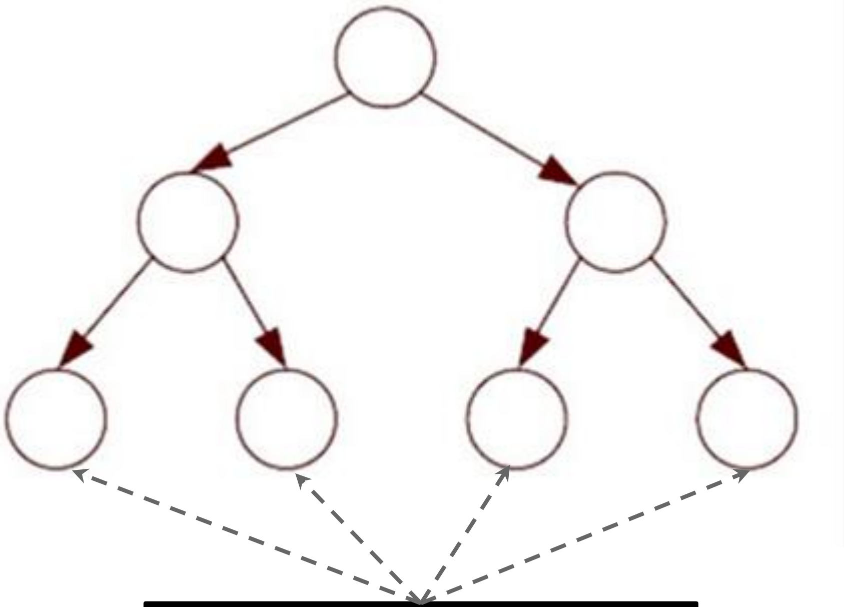
Dynamic Programming



Dynamic Programming

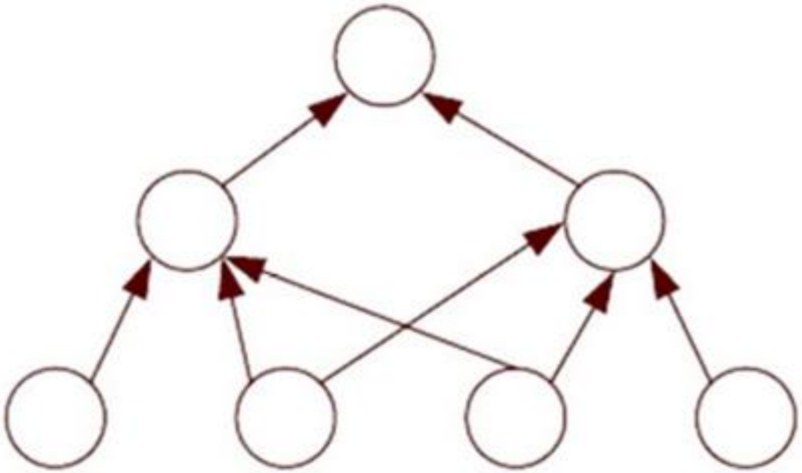
➔ Difference with Divide and Conquer

Divide and Conquer



For example, in mergesort, each subproblem is different and doesn't help solving other subproblems.

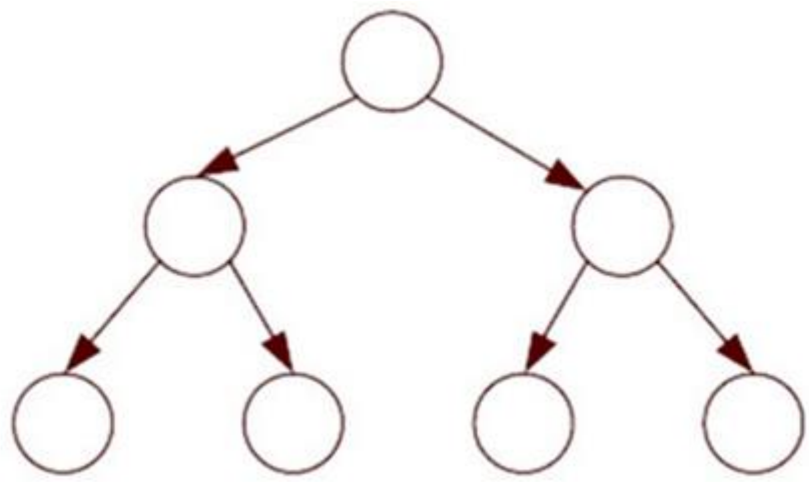
Dynamic Programming



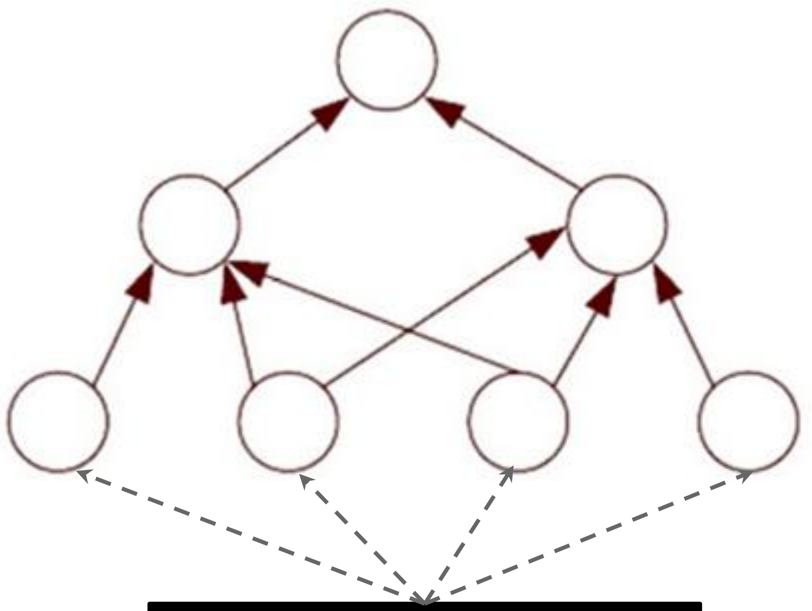
Dynamic Programming

➔ Difference with Divide and Conquer

Divide and Conquer



Dynamic Programming

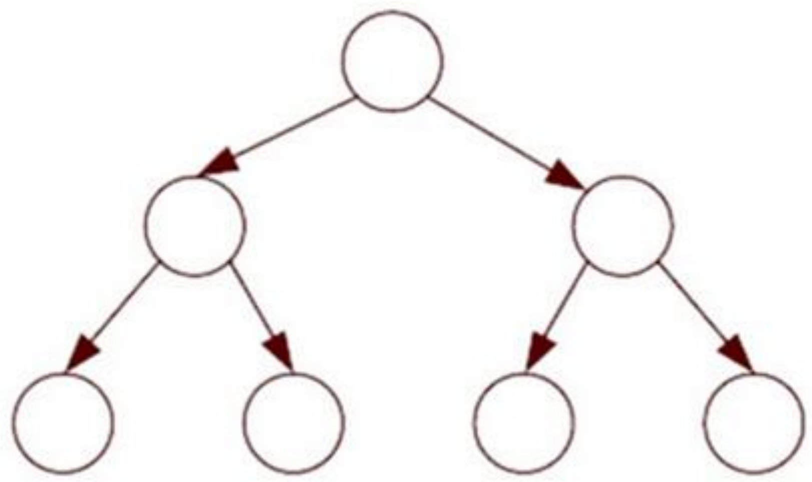


Each of these subproblems are **similar** between each other.

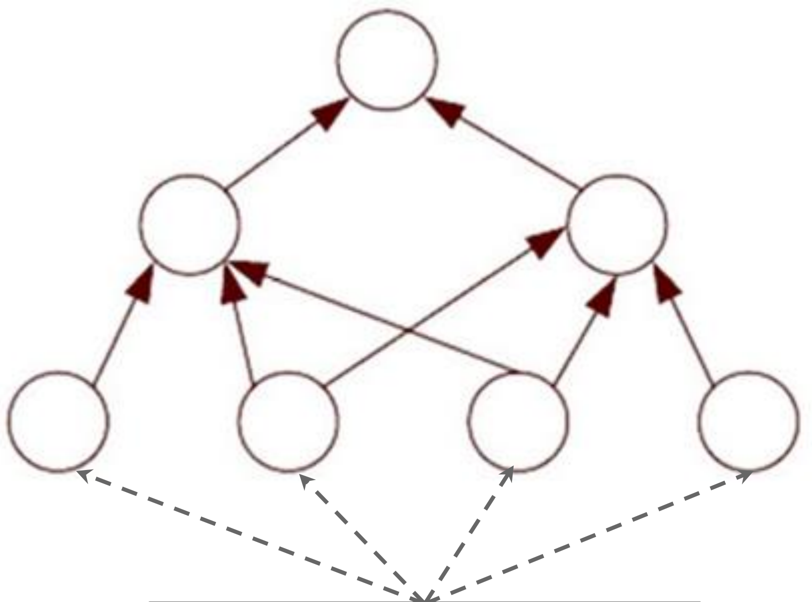
Dynamic Programming

➔ Difference with Divide and Conquer

Divide and Conquer



Dynamic Programming



Solve a subproblem helps to solve other subproblems.

Dynamic Programming (Example)

➔ Find the longest common substring

DESCRIPTION

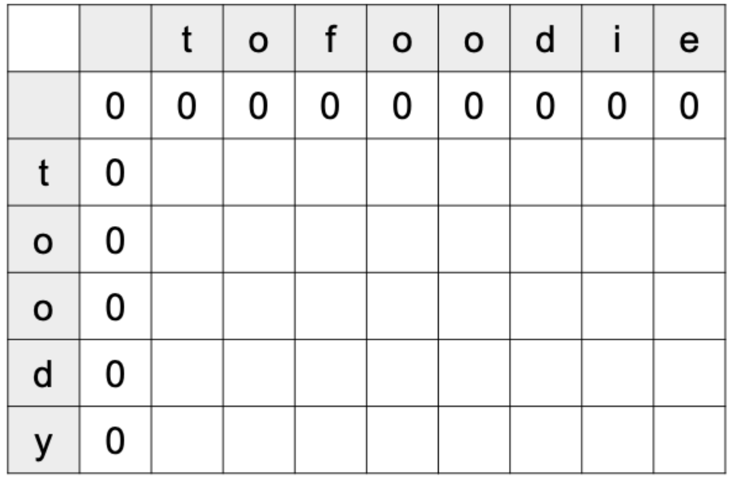
➔ Let s = “tofoodie” and t = “toody”

➔ Let D[i,j] be the length of the longest matching string between substrings $s_1 ... s_i$ and the substring of t between $t_1 ... t_j$

D[i,j] = 0 if s[i] <> t[j]

D[i-1,j-1] + 1 if s[i] == t[j]

DYNAMIC PROGRAMMING APPROACH



		t	o	f	o	o	d	i	e
	0	0	0	0	0	0	0	0	0
t	0								
o	0								
o	0								
d	0								
y	0								

Dynamic Programming (Example)

➔ Find the longest common substring

No matches on empty strings

DESCRIPTION

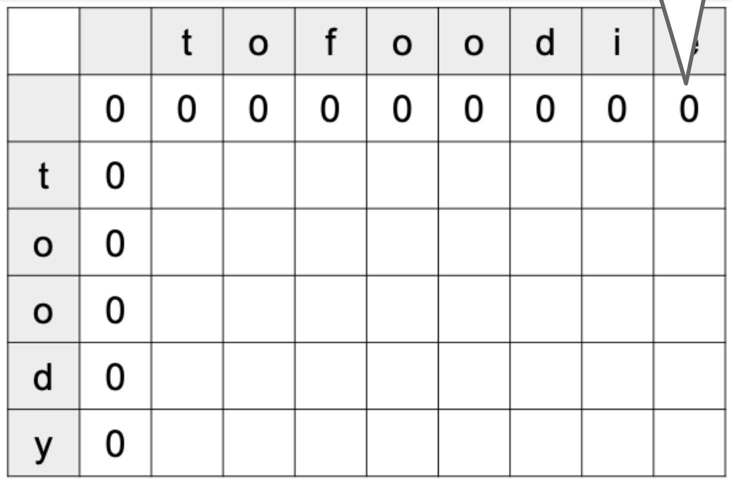
➔ Let s = “tofoodie” and t = “toody”

➔ Let D[i,j] be the length of the longest matching string between substrings $s_1...s_i$ and the substring of t between $t_1...t_j$

D[i,j] = 0 if s[i] <> t[j]

D[i-1,j-1] + 1 if s[i] == t[j]

DYNAMIC PROGRA



		t	o	f	o	o	d	i	e
	0	0	0	0	0	0	0	0	0
t	0								
o	0								
o	0								
d	0								
y	0								

Dynamic Programming (Example)

➔ Find the longest common substring

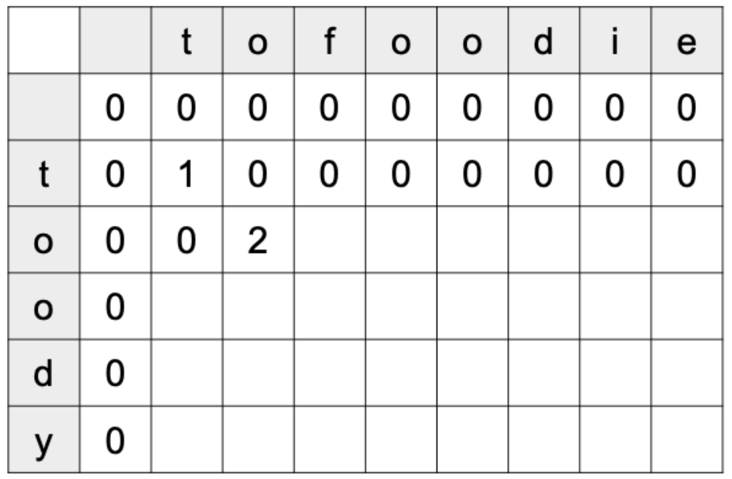
DESCRIPTION

- ➔ Let s = “tofoodie” and t = “toody”
- ➔ Let D[i,j] be the length of the longest matching string between substrings $s_1 ... s_i$ and the substring of t between $t_1 ... t_j$

D[i,j] = 0 if s[i] <> t[j]

D[i-1,j-1] + 1 if s[i] == t[j]

DYNAMIC PROGRAMMING APPROACH



		t	o	f	o	o	d	i	e
	0	0	0	0	0	0	0	0	0
t	0	1	0	0	0	0	0	0	0
o	0	0	2						
o	0								
d	0								
y	0								

Dynamic Programming (Example)

➔ Find the longest common substring

DESCRIPTION

➔ Let s = “tofoodie” and t = “toody”

➔ Let D[i,j] be the length of the longest matching string between substrings $s_1 \ldots s_i$ and the substring of t between $t_1 \ldots t_j$

D[i,j] = 0	if s[i] <> t[j]
D[i-1,j-1] + 1	if s[i] == t[j]

DYNAMIC PROGRAMMING APPROACH

		t	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
	0	0	0	[illegible]	0	0	0	0	0
t	0	1	0	0	0	0	0	0	0
o	0	0	2						
o	0								
d	0								
y	0								

D [i - 1, j - 1] + 1

Dynamic Programming (Example)

➔ Find the longest common substring

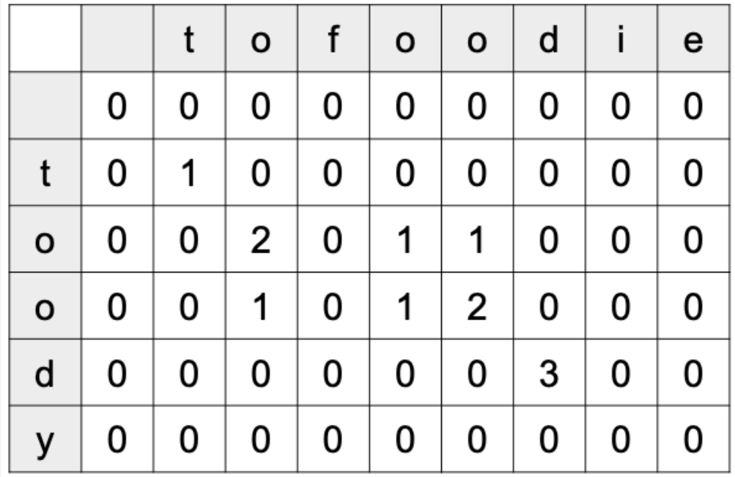
DESCRIPTION

- ➔ Let s = “tofoodie” and t = “toody”
- ➔ Let D[i,j] be the length of the longest matching string between substrings $s_1...s_i$ and the substring of t between $t_1...t_j$

D[i,j] = 0 if s[i] <> t[j]

D[i-1,j-1] + 1 if s[i] == t[j]

DYNAMIC PROGRAMMING APPROACH



		t	o	f	o	o	d	i	e
	0	0	0	0	0	0	0	0	0
t	0	1	0	0	0	0	0	0	0
o	0	0	2	0	1	1	0	0	0
o	0	0	1	0	1	2	0	0	0
d	0	0	0	0	0	0	3	0	0
y	0	0	0	0	0	0	0	0	0

Dynamic Programming (Example)

➔ Find the longest common substring

DESCRIPTION

➔ Let s = “tofoodie” and t = “toody”

➔ Let D[i,j] be the length of the longest matching string between substrings $s_1...s_i$ and the substring of t between $t_1...t_j$

D[i,j] = 0 if s[i] <> t[j]

D[i-1,j-1] + 1 if s[i] == t[j]

DYNAMIC PRO

LCS

		t	o	f	o	o	d	i	e
	0	0	0	0	0	0	0	0	0
t	0	1	0	0	0	0	0	0	0
o	0	0	2	0	1	1	0	0	0
o	0	0	1	0	1	2	0	0	0
d	0	0	0	0	0	0	3	0	0
y	0	0	0	0	0	0	0	0	0

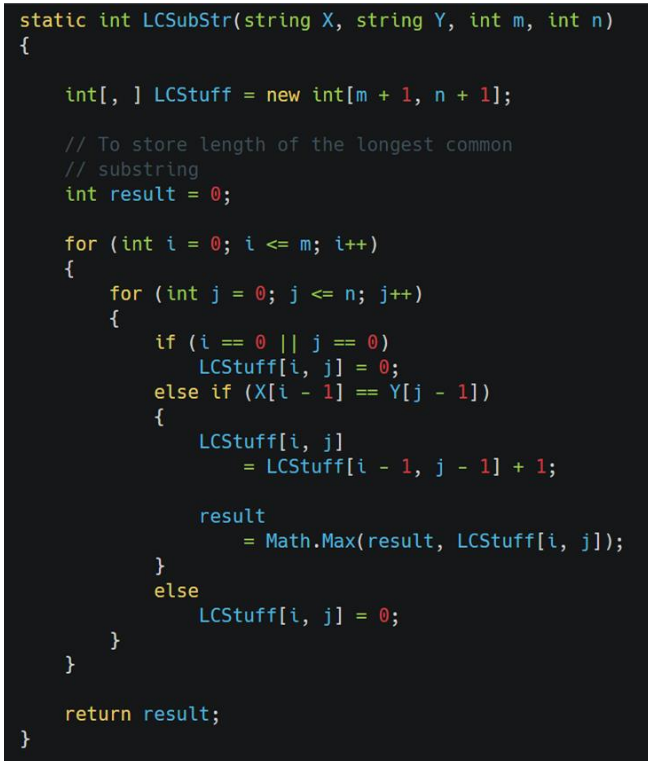
Largest value found

Dynamic Programming (Example)

➔ Find the longest common substring. For example, for “abcdxyz” and “xyzabcd”, the result would be “abcd”

COMMON APPROACH

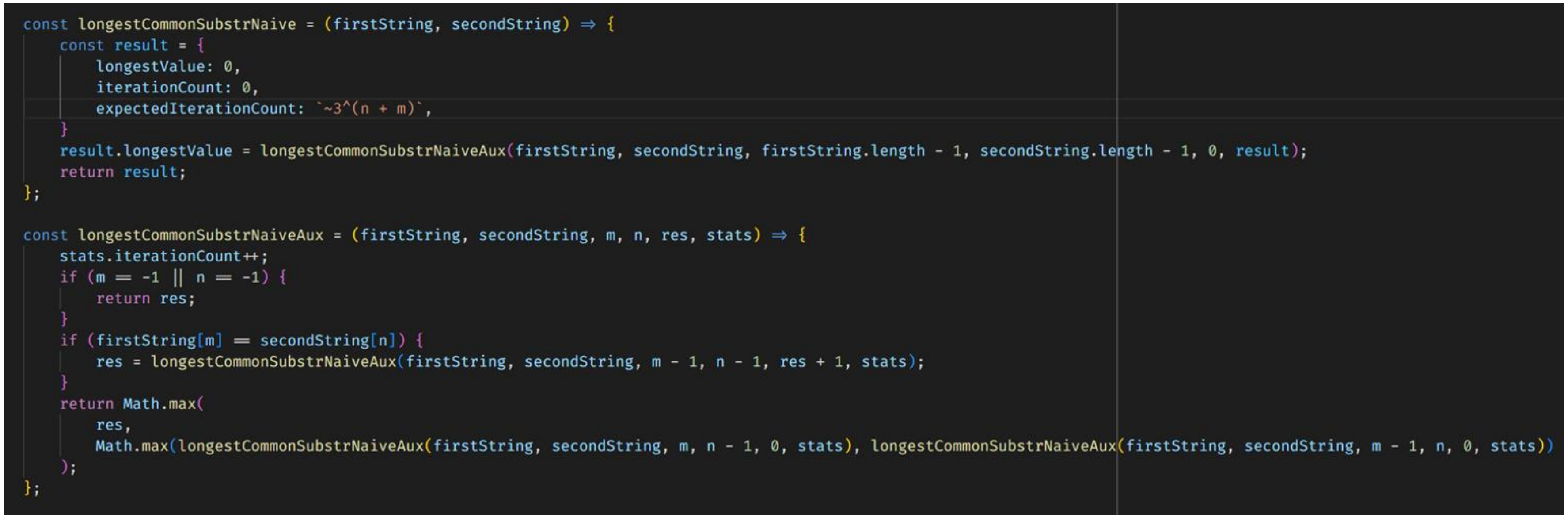
DYNAMIC PROGRAMMING APPROACH



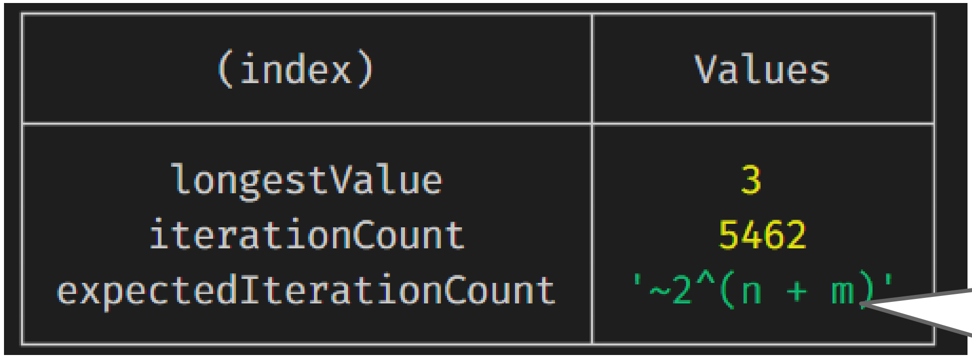
```
static int LCSubStr(string X, string Y, int m, int n)
{

    int[, ] LCStuff = new int[m + 1, n + 1];

    // To store length of the longest common
    // substring
    int result = 0;

    for (int i = 0; i <= m; i++)
    {
        for (int j = 0; j <= n; j++)
        {
            if (i == 0 || j == 0)
                LCStuff[i, j] = 0;
            else if (X[i - 1] == Y[j - 1])
            {
                LCStuff[i, j]
                    = LCStuff[i - 1, j - 1] + 1;

                result
                    = Math.Max(result, LCStuff[i, j]);
            }
            else
                LCStuff[i, j] = 0;
        }
    }

    return result;
}
```

Dynamic Programming (Example)

COMMON APPROACH



```
const longestCommonSubstrNaive = (firstString, secondString) => {
    const result = {
        longestValue: 0,
        iterationCount: 0,
        expectedIterationCount: `~3^(n + m)`,
    }
    result.longestValue = longestCommonSubstrNaiveAux(firstString, secondString, firstString.length - 1, secondString.length - 1, 0, result);
    return result;
};

const longestCommonSubstrNaiveAux = (firstString, secondString, m, n, res, stats) => {
    stats.iterationCount++;
    if (m == -1 || n == -1) {
        return res;
    }
    if (firstString[m] == secondString[n]) {
        res = longestCommonSubstrNaiveAux(firstString, secondString, m - 1, n - 1, res + 1, stats);
    }
    return Math.max(
        res,
        Math.max(longestCommonSubstrNaiveAux(firstString, secondString, m, n - 1, 0, stats), longestCommonSubstrNaiveAux(firstString, secondString, m - 1, n, 0, stats))
    );
};
```

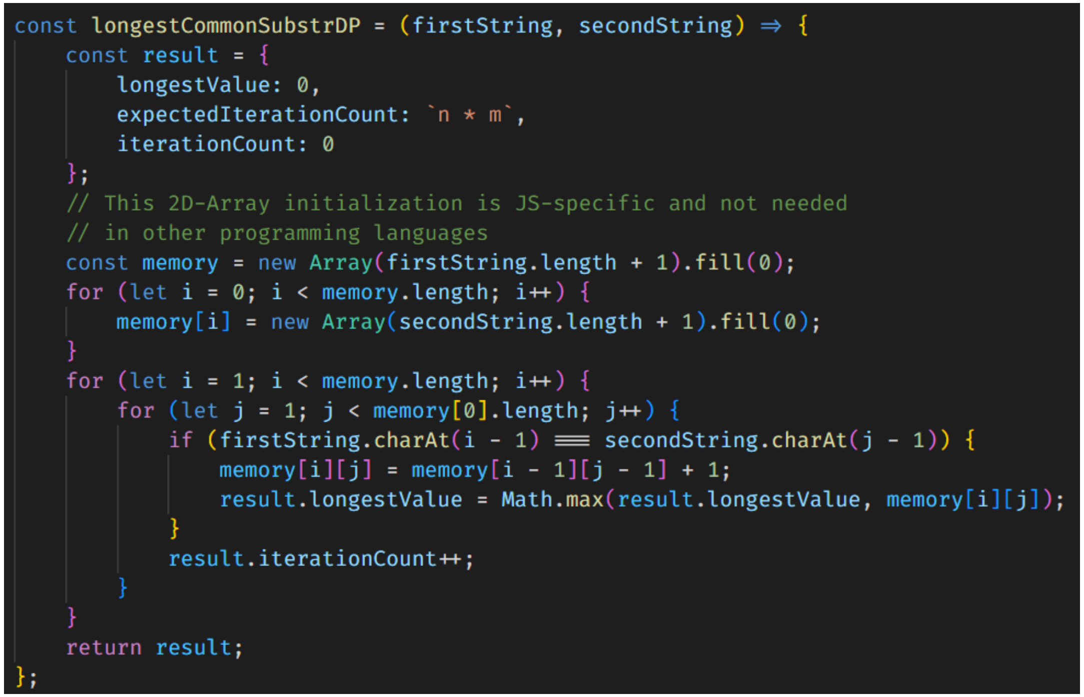


(index)	Values
longestValue	3
iterationCount	5462
expectedIterationCount	'~2^(n + m)'

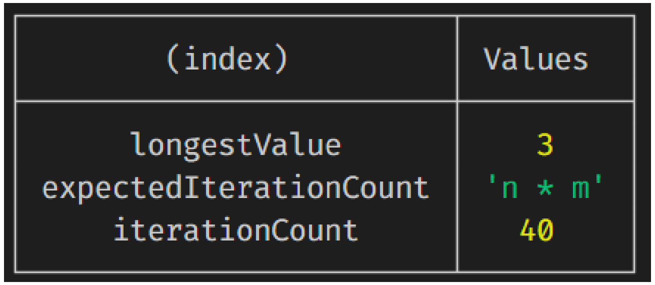
To calculate this, we need to generate the recursion tree

Dynamic Programming (Example)

DYNAMIC PROGRAMMING APPROACH



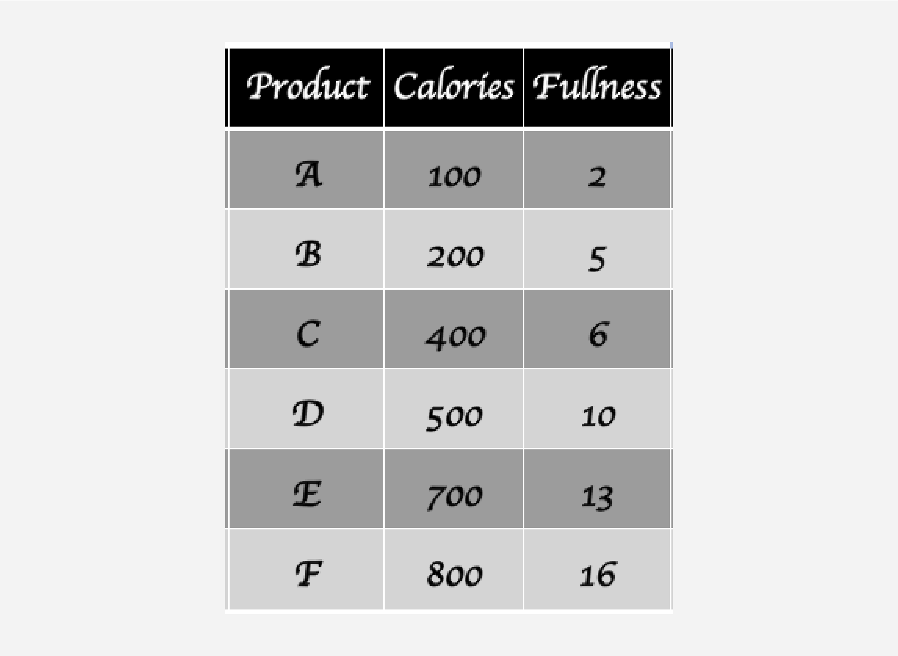
```javascript
const longestCommonSubstrDP = (firstString, secondString) => {
    const result = {
        longestValue: 0,
        expectedIterationCount: `n * m`,
        iterationCount: 0
    };
    // This 2D-Array initialization is JS-specific and not needed
    // in other programming languages
    const memory = new Array(firstString.length + 1).fill(0);
    for (let i = 0; i < memory.length; i++) {
        memory[i] = new Array(secondString.length + 1).fill(0);
    }
    for (let i = 1; i < memory.length; i++) {
        for (let j = 1; j < memory[0].length; j++) {
            if (firstString.charAt(i - 1) === secondString.charAt(j - 1)) {
                memory[i][j] = memory[i - 1][j - 1] + 1;
                result.longestValue = Math.max(result.longestValue, memory[i][j]);
            }
            result.iterationCount++;
        }
    }
    return result;
};
```



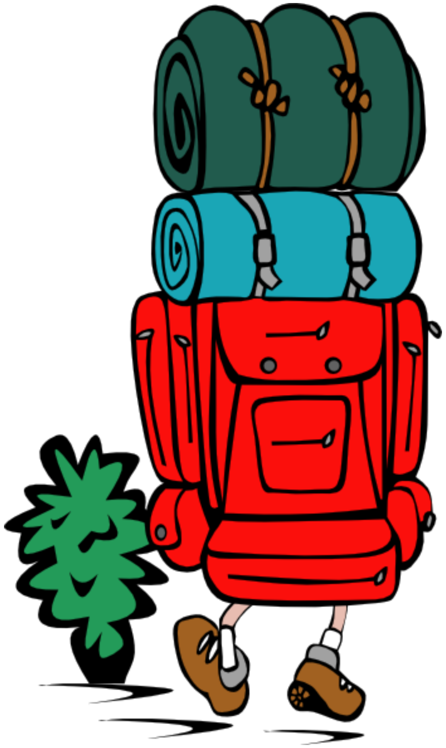
(index)	Values
longestValue	3
expectedIterationCount	'n * m'
iterationCount	40

Dynamic Programming (Example)

➔ Knapsack Problem



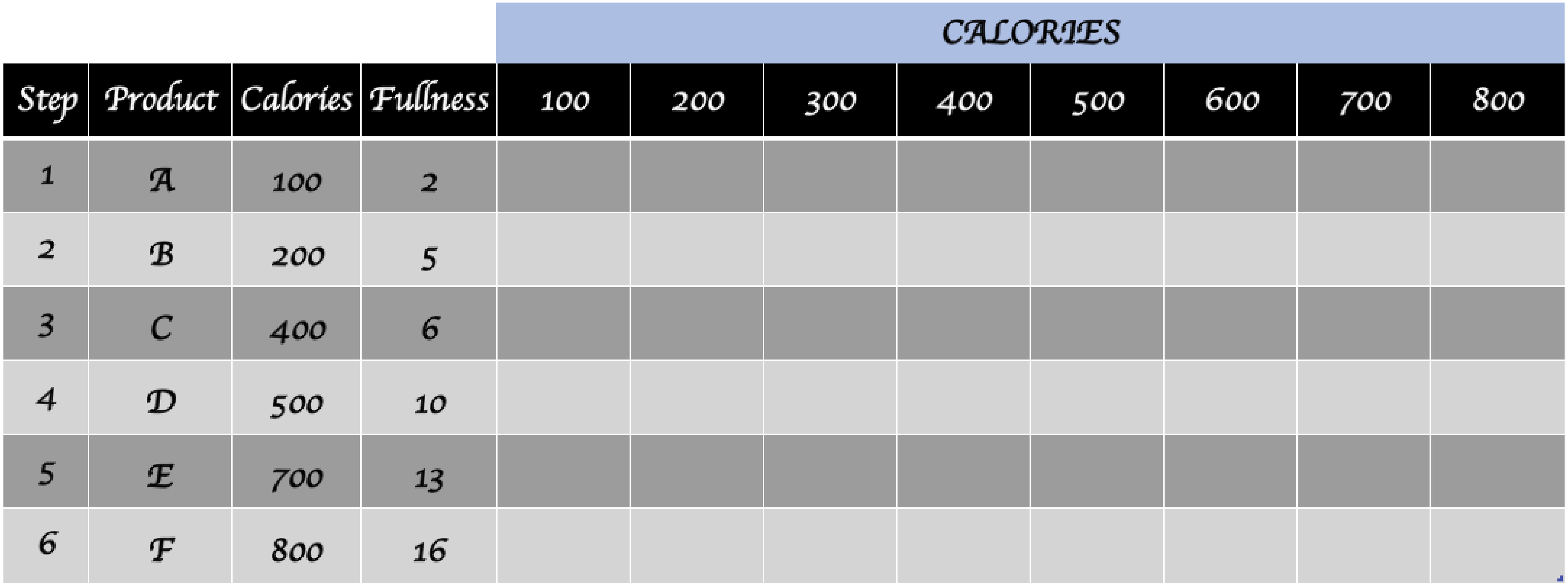
Product	Calories	Fullness
A	100	2
B	200	5
C	400	6
D	500	10
E	700	13
F	800	16



Have to choose the right food for you. Only can eat 800 calories by day and have to maximize the fullness sensation

Dynamic Programming (Example)

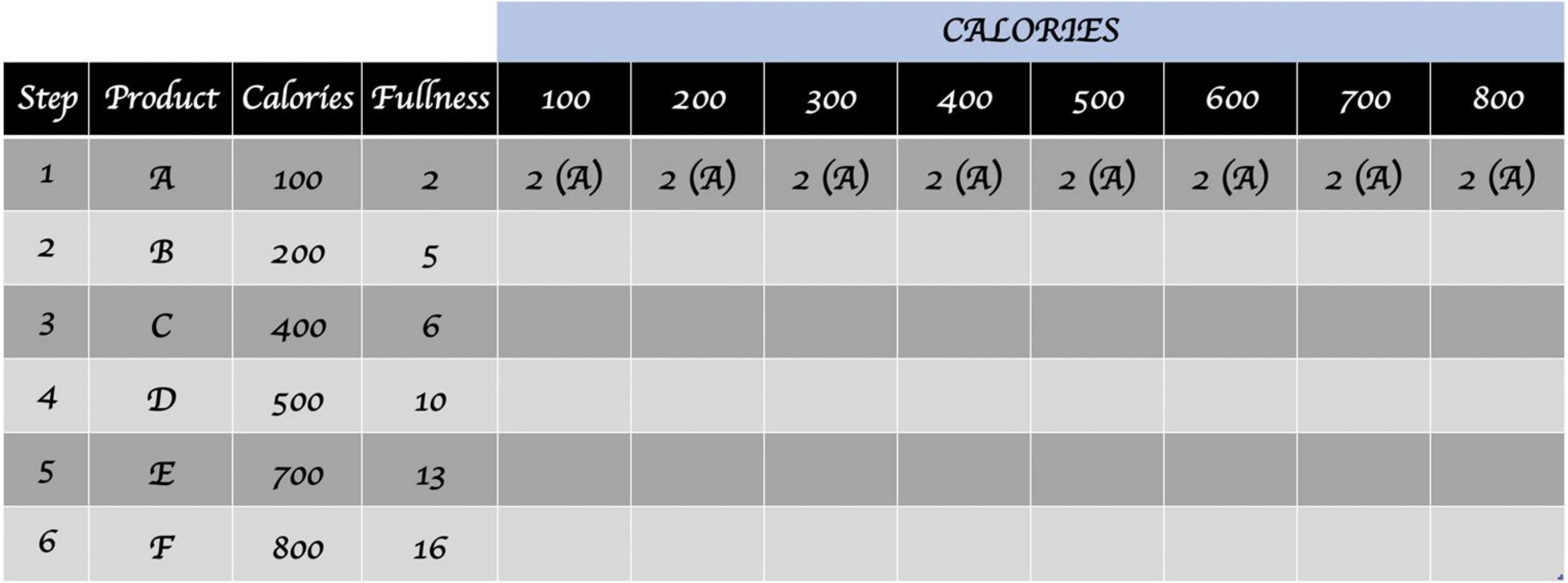
➔ Knapsack Problem



				CALORIES							
Step	Product	Calories	Fullness	100	200	300	400	500	600	700	800
1	A	100	2								
2	B	200	5								
3	C	400	6								
4	D	500	10								
5	E	700	13								
6	F	800	16								

Dynamic Programming (Example)

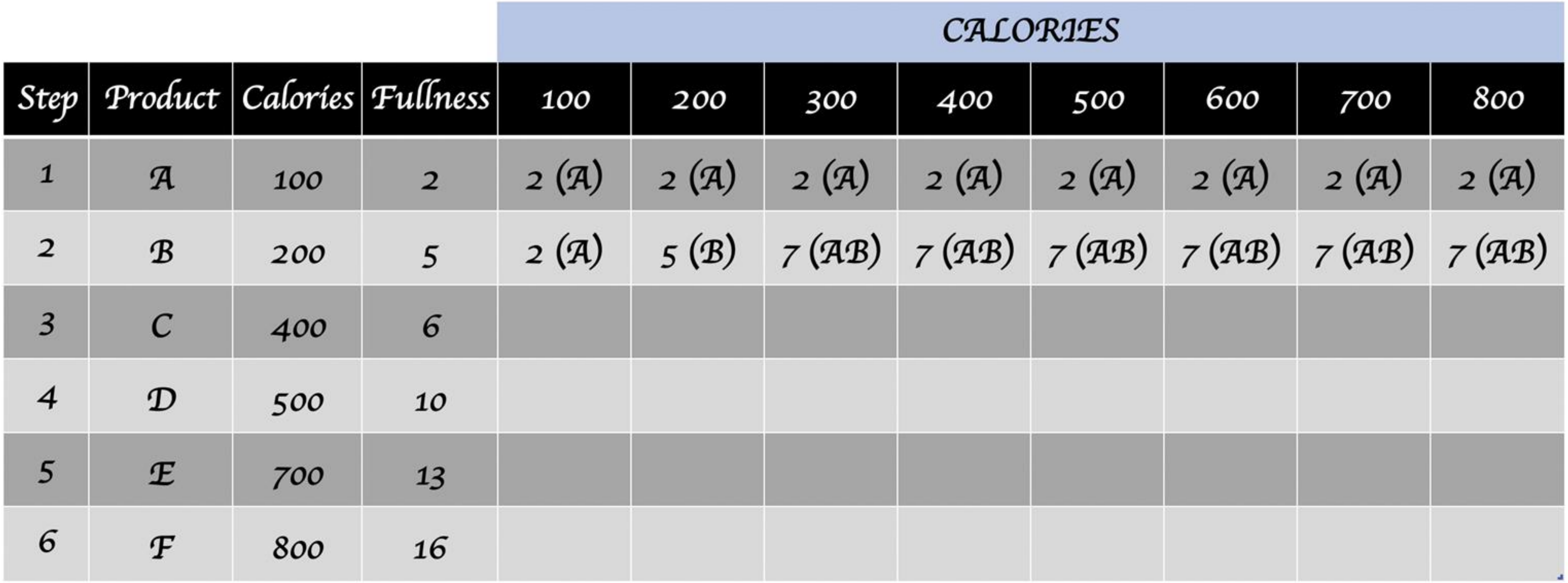
- ➔ Knapsack Problem



				CALORIES							
Step	Product	Calories	Fullness	100	200	300	400	500	600	700	800
1	A	100	2	2 (A)	2 (A)	2 (A)	2 (A)	2 (A)	2 (A)	2 (A)	2 (A)
2	B	200	5								
3	C	400	6								
4	D	500	10								
5	E	700	13								
6	F	800	16								

Dynamic Programming (Example)

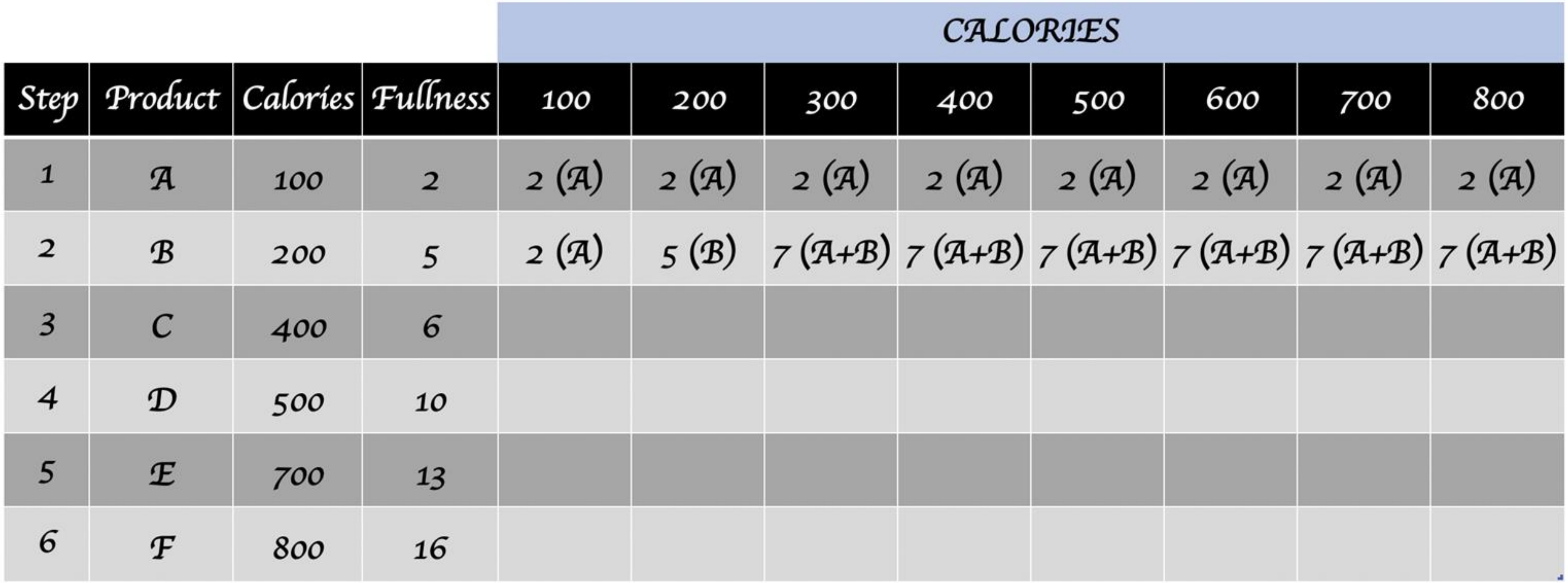
➔ Knapsack Problem



				CALORIES							
Step	Product	Calories	Fullness	100	200	300	400	500	600	700	800
1	A	100	2	2 (A)	2 (A)	2 (A)	2 (A)	2 (A)	2 (A)	2 (A)	2 (A)
2	B	200	5	2 (A)	5 (B)	7 (AB)	7 (AB)	7 (AB)	7 (AB)	7 (AB)	7 (AB)
3	C	400	6								
4	D	500	10								
5	E	700	13								
6	F	800	16								

Dynamic Programming (Example)

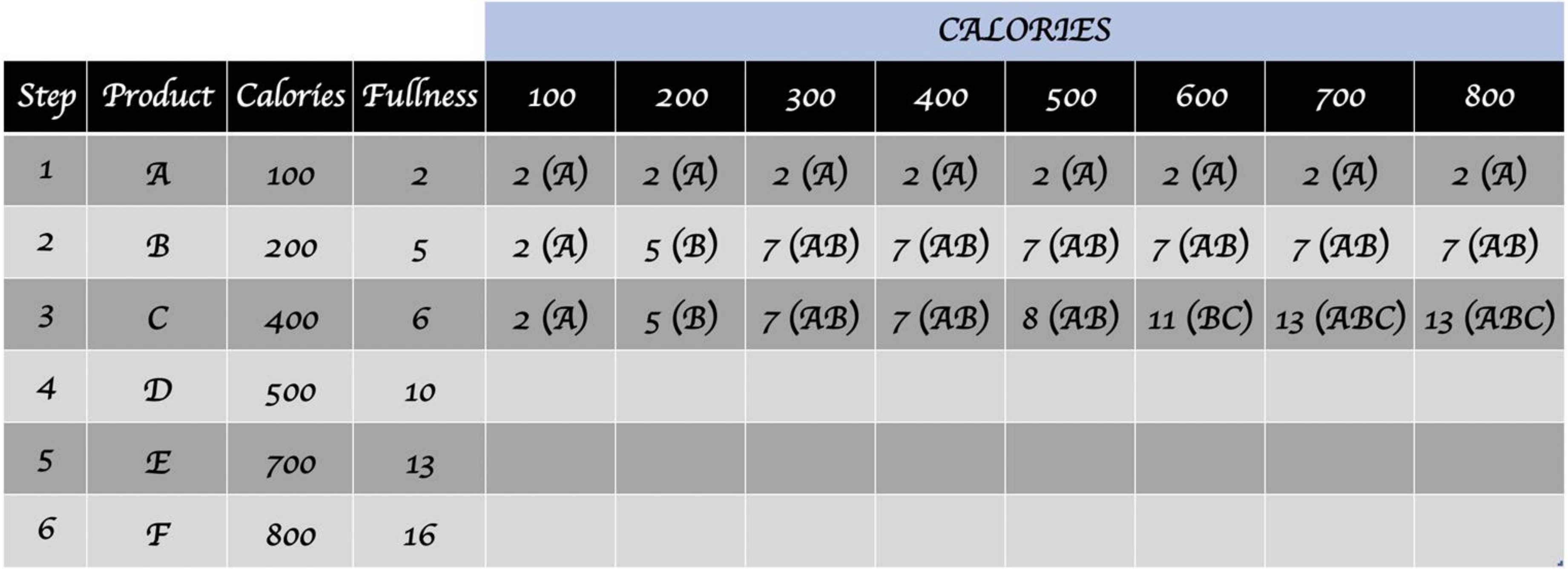
➔ Knapsack Problem



				CALORIES							
Step	Product	Calories	Fullness	100	200	300	400	500	600	700	800
1	A	100	2	2 (A)	2 (A)	2 (A)	2 (A)	2 (A)	2 (A)	2 (A)	2 (A)
2	B	200	5	2 (A)	5 (B)	7 (A+B)	7 (A+B)	7 (A+B)	7 (A+B)	7 (A+B)	7 (A+B)
3	C	400	6								
4	D	500	10								
5	E	700	13								
6	F	800	16								

Dynamic Programming (Example)

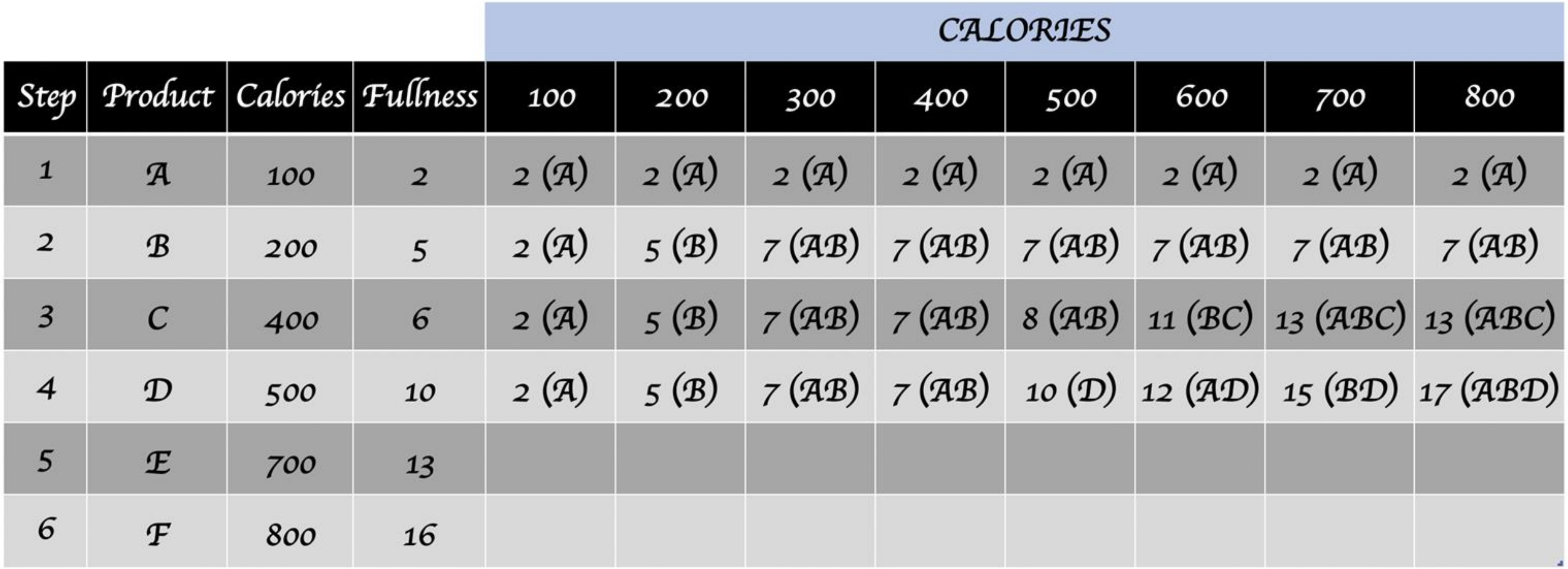
➔ Knapsack Problem



				CALORIES							
Step	Product	Calories	Fullness	100	200	300	400	500	600	700	800
1	A	100	2	2 (A)	2 (A)	2 (A)	2 (A)	2 (A)	2 (A)	2 (A)	2 (A)
2	B	200	5	2 (A)	5 (B)	7 (AB)	7 (AB)	7 (AB)	7 (AB)	7 (AB)	7 (AB)
3	C	400	6	2 (A)	5 (B)	7 (AB)	7 (AB)	8 (AB)	11 (BC)	13 (ABC)	13 (ABC)
4	D	500	10								
5	E	700	13								
6	F	800	16								

Dynamic Programming (Example)

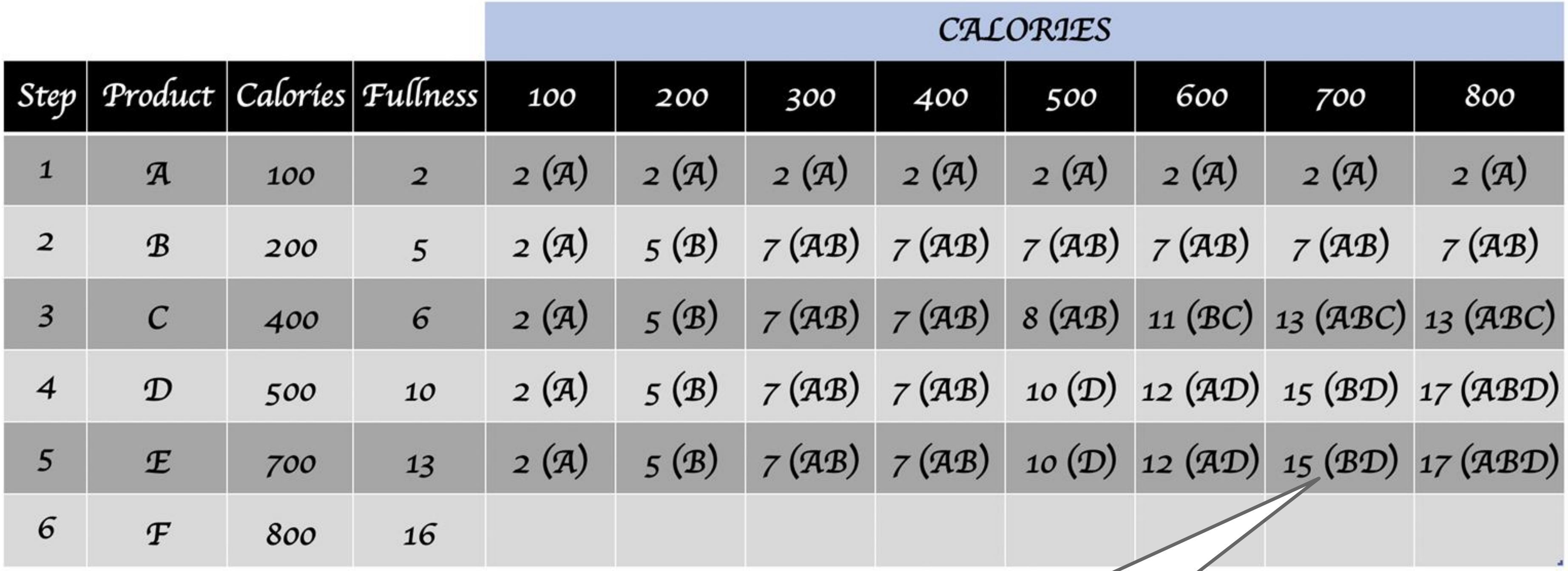
➔ Knapsack Problem



				CALORIES							
Step	Product	Calories	Fullness	100	200	300	400	500	600	700	800
1	A	100	2	2 (A)	2 (A)	2 (A)	2 (A)	2 (A)	2 (A)	2 (A)	2 (A)
2	B	200	5	2 (A)	5 (B)	7 (AB)	7 (AB)	7 (AB)	7 (AB)	7 (AB)	7 (AB)
3	C	400	6	2 (A)	5 (B)	7 (AB)	7 (AB)	8 (AB)	11 (BC)	13 (ABC)	13 (ABC)
4	D	500	10	2 (A)	5 (B)	7 (AB)	7 (AB)	10 (D)	12 (AD)	15 (BD)	17 (ABD)
5	E	700	13								
6	F	800	16								

Dynamic Programming (Example)

➔ Knapsack Problem

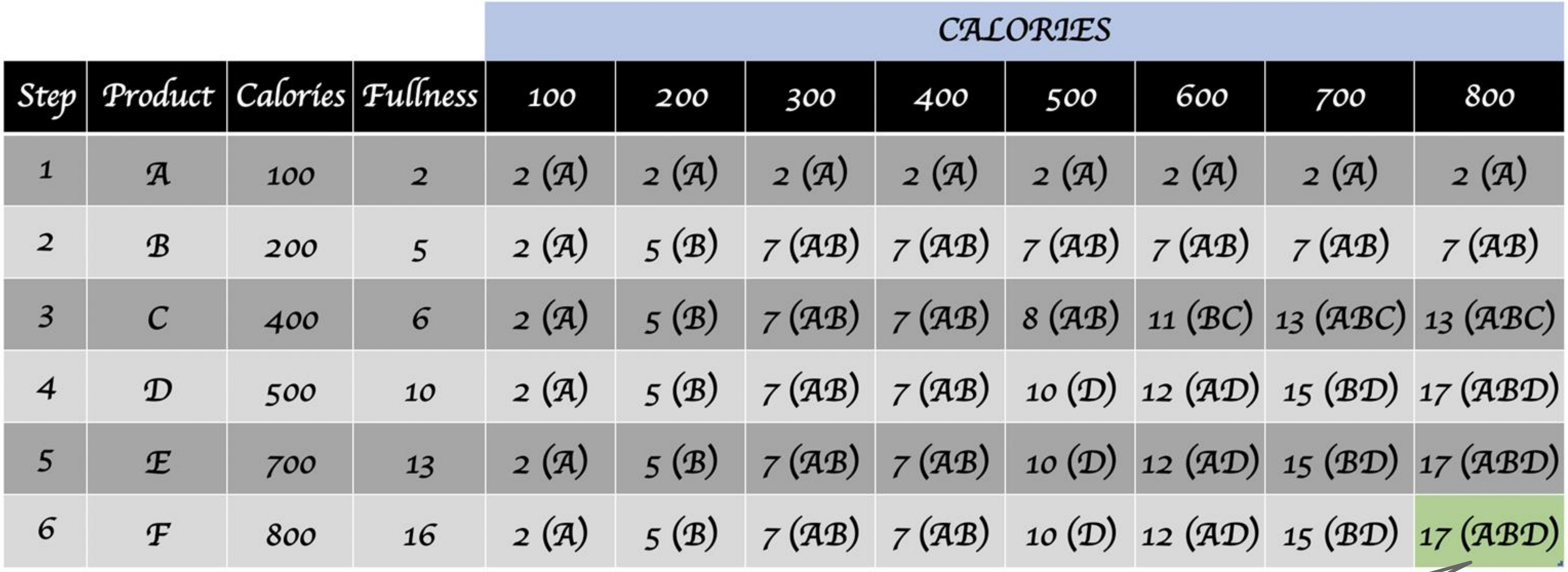


				CALORIES							
Step	Product	Calories	Fullness	100	200	300	400	500	600	700	800
1	A	100	2	2 (A)	2 (A)	2 (A)	2 (A)	2 (A)	2 (A)	2 (A)	2 (A)
2	B	200	5	2 (A)	5 (B)	7 (AB)	7 (AB)	7 (AB)	7 (AB)	7 (AB)	7 (AB)
3	C	400	6	2 (A)	5 (B)	7 (AB)	7 (AB)	8 (AB)	11 (BC)	13 (ABC)	13 (ABC)
4	D	500	10	2 (A)	5 (B)	7 (AB)	7 (AB)	10 (D)	12 (AD)	15 (BD)	17 (ABD)
5	E	700	13	2 (A)	5 (B)	7 (AB)	7 (AB)	10 (D)	12 (AD)	15 (BD)	17 (ABD)
6	F	800	16								

Why no changes on this step?

Dynamic Programming (Example)

➔ Knapsack Problem



				CALORIES							
Step	Product	Calories	Fullness	100	200	300	400	500	600	700	800
1	A	100	2	2 (A)	2 (A)	2 (A)	2 (A)	2 (A)	2 (A)	2 (A)	2 (A)
2	B	200	5	2 (A)	5 (B)	7 (AB)	7 (AB)	7 (AB)	7 (AB)	7 (AB)	7 (AB)
3	C	400	6	2 (A)	5 (B)	7 (AB)	7 (AB)	8 (AB)	11 (BC)	13 (ABC)	13 (ABC)
4	D	500	10	2 (A)	5 (B)	7 (AB)	7 (AB)	10 (D)	12 (AD)	15 (BD)	17 (ABD)
5	E	700	13	2 (A)	5 (B)	7 (AB)	7 (AB)	10 (D)	12 (AD)	15 (BD)	17 (ABD)
6	F	800	16	2 (A)	5 (B)	7 (AB)	7 (AB)	10 (D)	12 (AD)	15 (BD)	17 (ABD)

This is the best option to get 800 calories and maximice fullness sensation

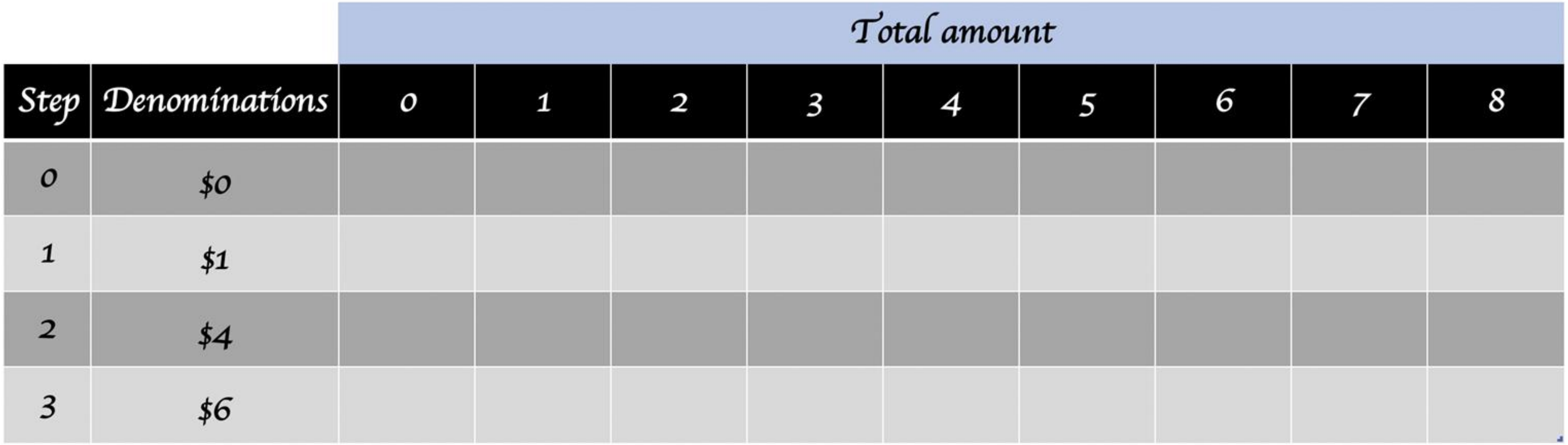
Dynamic Programming (Example)

- Cashier Problem

For a specific amount give the least amount of units of the different denominations.

Dynamic Programming (Example)

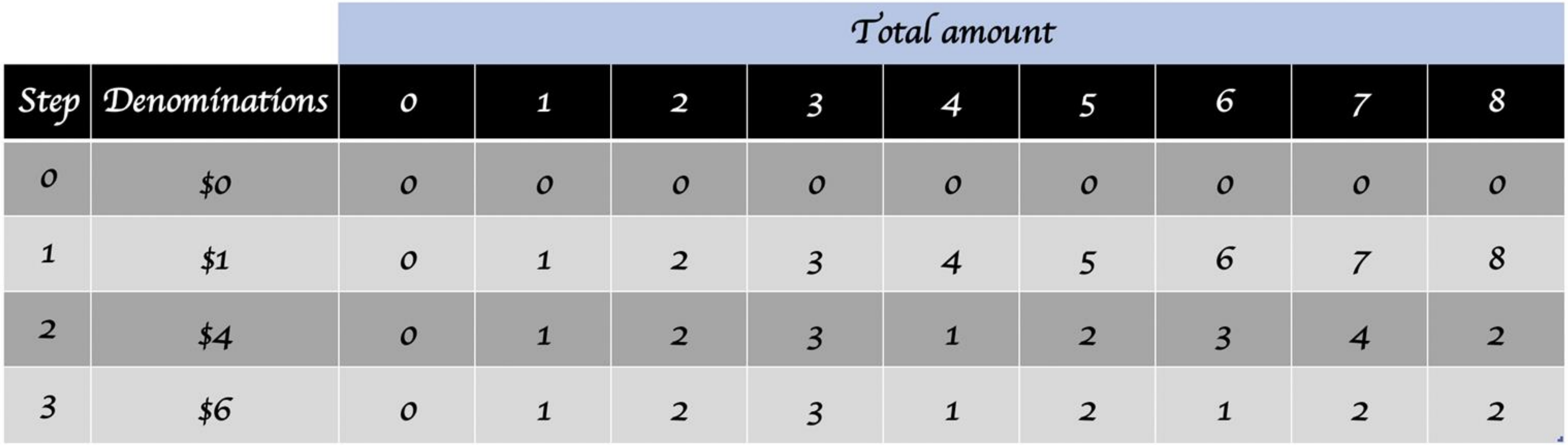
- ➔ Cashier Problem



		Total amount								
Step	Denominations	0	1	2	3	4	5	6	7	8
0	$0									
1	$1									
2	$4									
3	$6									

Dynamic Programming (Example)

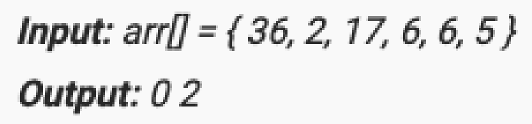
- ➔ Cashier Problem



		Total amount								
Step	Denominations	0	1	2	3	4	5	6	7	8
0	$0	0	0	0	0	0	0	0	0	0
1	$1	0	1	2	3	4	5	6	7	8
2	$4	0	1	2	3	1	2	3	4	2
3	$6	0	1	2	3	1	2	1	2	2

Dynamic Programming (Example)

- Other examples
    - **Ugly numbers.** Given a number n, find n Ugly numbers. Ugly numbers are numbers whose only prime factors are 2, 3 or 5. The number 1 is included by convention. The sequence 1, 2, 3, 4, 5, 6, 8, 9, 10, 12, 15 shows the first 11 ugly numbers.
    - **Min Cost Path.** Given a cost matrix and a position, calculate the cost of minimum cost path form position (0,0) to the input position. Each cell of the matrix represents a cost to traverse through that cell.
    - **Longest Common Subsequence of three strings.**
    - **Index of elements which are equal to the sum of all succeeding elements**



Input: *arr[] = { 36, 2, 17, 6, 6, 5 }*

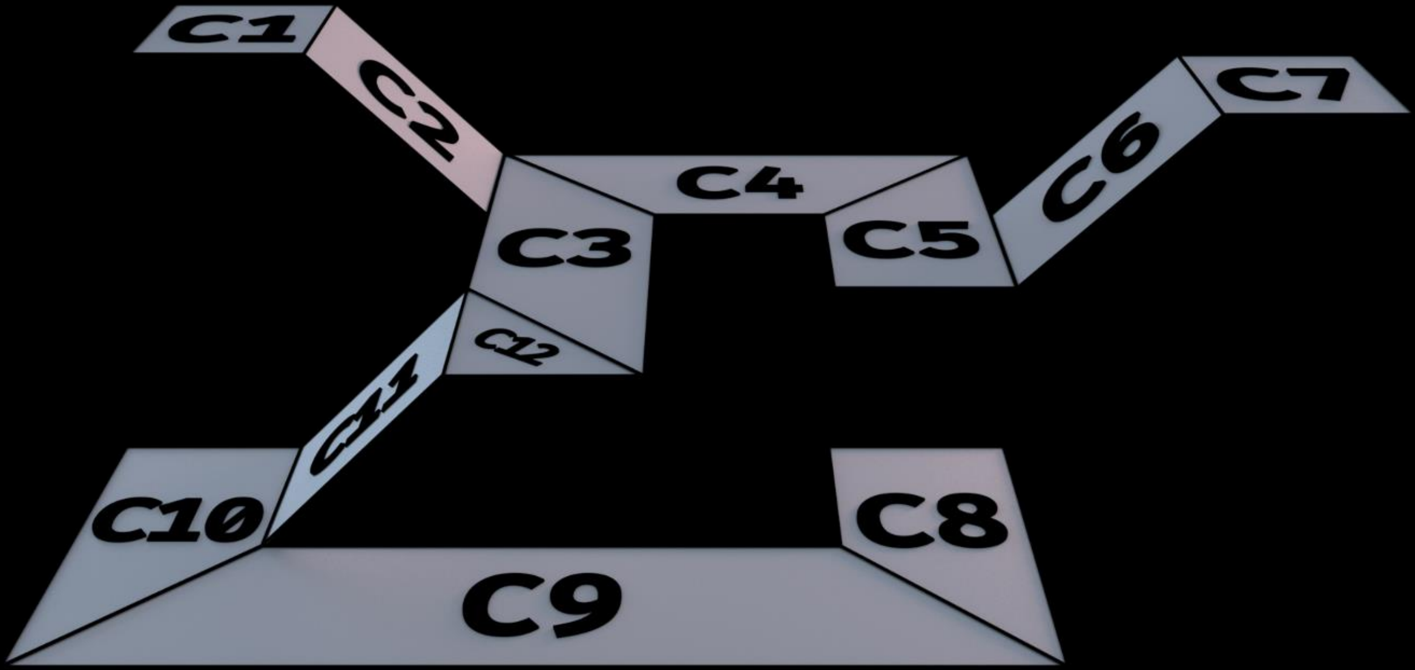
Output: *0 2*

Different approaches to the same problem

- There are problems that can be addressed with different approaches.

- For example, knapsack problem can be solved using:
  - Dynamic Programming (recently seen).
  - Backtracking (your responsibility).
  - Genetic algorithms (more on this later).

Heuristic Algorithms



Heuristic Algorithms

- ➔ Algorithms that find solutions that are guaranteed to be close to optimal.

- ➔ Scenarios where precision can be sacrificed for performance.

- ➔ Also called approximate algorithms.

- ➔ Examples of heuristic algorithms:
  - ◆ Genetic algorithms.
  - ◆ Greedy algorithms.

Genetic Algorithms

Genetic Algorithms

➔ In the 50's and 60's computer scientists studied evolutionary systems with the idea of use them as a tool to optimize engineering problems.

➔ Inspired by **biological evolution process** and based on concepts like: **natural selection, genetic inheritance and mutations**.

Developed by

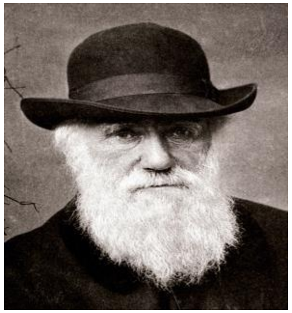
(1970s)

JOHN HENRY HOLLAND

Based on theory of

(1859)

CHARLES DARWIN



Genetic Algorithms

- ➔ Genetic Algorithms (GAs) are adaptive heuristic search algorithm based on the evolutionary ideas of natural selection and genetics.
- ➔ They represent an intelligent exploitation of a **random search** used to solve optimization problems.
- ➔ Not entirely random. They exploit **historical information** to direct the search
- ➔ This simulate the natural selection process. Use techniques inspired by evolutionary biology such as inheritance, mutation, selection, and recombination.

Genetic Algorithms

HAVE TO BE DEFINED

➔ Genetic representation of the solution domain.

➔ Fitness function to evaluate the solution domain.

DEFINITIONS

➔ **Cell.** Contains chromosomes (string DNA).

.

➔ **Chromosomes.** Set of genes (blocks DNA). Scheme or blueprint for an individual.

➔ **Genotype.** Collection of genes responsible for a particular trait.

➔ **Trait.** Aspect of an individual.

Genetic Algorithms

HAVE TO BE DEFINED

➔ Genetic representation of the solution domain.

➔ Fitness function to evaluate the solution domain.

DEFINITIONS

➔ **Reproduction.** Combination of parental genes.

➔ **Mutation.** Errors during reproduction.

➔ **Individual.** Any possible solution.

.

➔ **Fitness.** Function used to select the best individuals. How many times can an individual reproduce before dying.

Genetic Algorithms

HAVE TO BE DEFINED

- ➔ Genetic representation of the solution domain.
- ➔ Fitness function to evaluate the solution domain.

DEFINITIONS

- ➔ **Population.** Group of all individuals.
- ➔ **Search space.** All the possible solutions for a problem.
.
- ➔ **Genome.** Collection of chromosomes for an individual.

Genetic Algorithms (Representation of solutions)

- ➔ Typical representation of a solution is an array of bits (Bit Vectors).
- ➔ Try to use a fixed length representation, will facilitate the crossover.
- ➔ Initially we solve the knapsack problem using Dynamic Programming, but... What happens if we try to use genetic algorithms?

Genetic Algorithms (Representation of solutions)

➔ Knapsack Problem (Using Genetic Algorithms)

1	1	1	0	0	1	0	1	1	0	0	0	0	1	0	1
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

- ➔ Sweet Cookies
- ➔ Apple
- ➔ Integral Cookies
- ➔ Hamburguer
- ➔ Salad + Chicken
- ➔ Chocolat Bar
- ➔ Chicken and Rice
- ➔ Rice and Beans + Eggs

0 = don't choose this product
1 = choose this product

Genetic Algorithms (Representation of solutions)

- ➔ Bit Vectors
  - ◆ Specialized type to work with bit arrays (boolean values).
  - ◆ No waste of space.
  - ◆ There is not type “bit” in the programming languages, we work with bigger types to represent a bit array (bytes).
  - ◆ Save space when we are going to transfer data over the network. Efficient use of resources.
  - ◆ Used to compress data and encryption algorithms.

Genetic Algorithms (Representation of solutions)

- Bit Vectors
  - Specialized type to work with bit arrays (boolean values).
  - No waste of space.
  - There is not type “bit” in the programming languages, we work with bigger types to represent a bit array (bytes).
  - Save space when we are going to transfer data over the network. Efficient use of resources.

You can represent a Bit Vector as follows:

```
int bit_vector = 0;
```

Genetic Algorithms (Representation of solutions)

- ➔ Bit Vectors
  - ◆ Specialized type to work with bit arrays (boolean values).
  - ◆ No waste of space.
  - ◆ There is not type “bit” in the programming languages, we work with bigger types to represent a bit array (bytes).
  - ◆ Save space when we are going to tran network. Efficient use of resources

This give you a bit vector of 32 positions (4 bytes). Use bitwise operations to perform operations.

You can represent a Bit Vector as follows:

```
int bit_vector = 0;
```

Genetic Algorithms (Fitness)

- ➔ Defined over the genetic representation.
- ➔ Measures the quality of a given solution.
- ➔ Defines which solutions have more probabilities of survive and reproduce.

Genetic Algorithms (Fitness)

- ➔ Knapsack Problem (Using Genetic Algorithms)

fitness(solution) = fullness(solution) + calories(solution)

fullness(solution) = sum(fullness of each item in the
solution)

-----

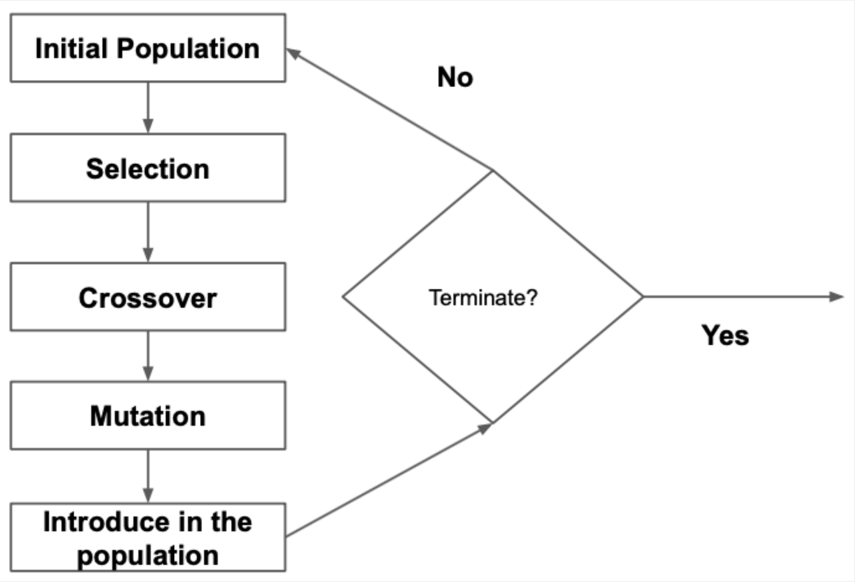
sum(fullness all the possible items)

calories(solution) =

```
        x = abs(sum(calories of each item in the solution) - 2000)

        if (x == 0)
                x = 1
        x = 2000/x
```

Genetic Algorithms

STRATEGY

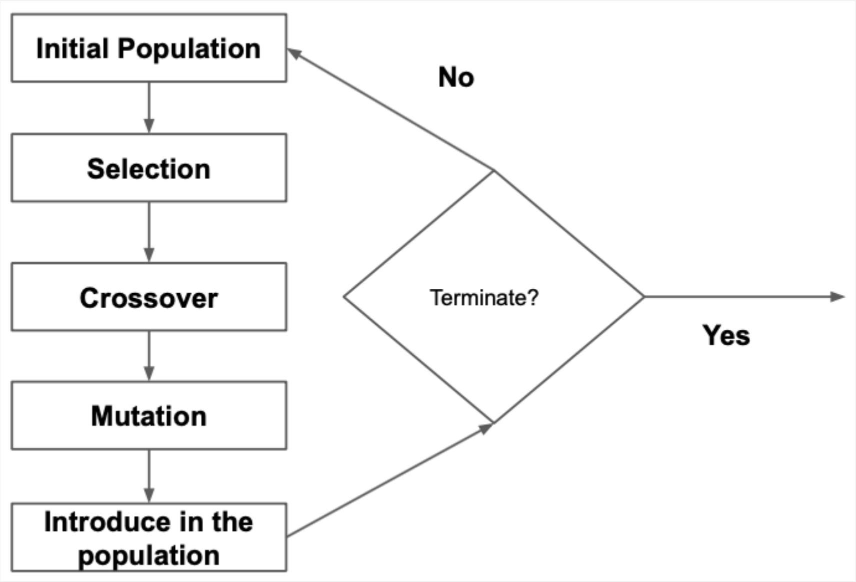


GENERAL DESCRIPTION

- ➔ Evolution starts with a **random** population, this is called **first generation**.
- ➔ In each generation, **fitness** of every individual is calculated.
- ➔ Several individuals are **selected** from the current population based on their fitness.
- ➔ These individuals are **combined** to obtain new individuals.
- ➔ Some individuals are **discarded** from the new population (lowest fitness).
- ➔ At the end of each iteration we have a new population (new generation) that will be used in the next iteration.

Genetic Algorithms

STRATEGY



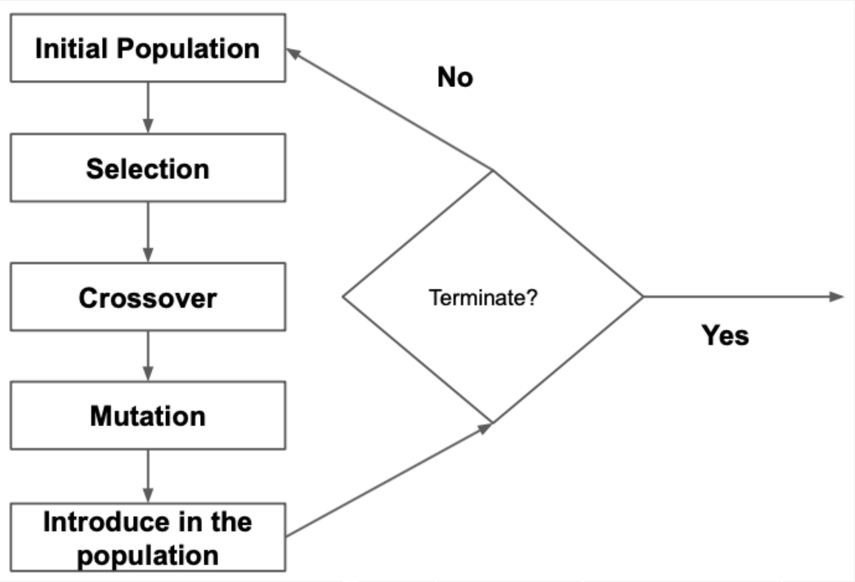
GENERAL DESCRIPTION

- ➔ Evolution starts with a **random** population, this is called **first generation**.
- ➔ In each generation, **fitness** of every individual is calculated.
- ➔ Several individuals are **selected** from the current population based on their fitness.
- ➔ These individu new individual
- ➔ Some individuals are **rded** from the new population (lowest fit s).
- ➔ At the end of each iteration we have a new population (new generation) that will be used in the next iteration.

When does a genetic algorithm end?

Genetic Algorithms

STRATEGY



GENERAL DESCRIPTION

- ➔ Evolution starts with a **random** population, this is called **first generation**.
- ➔ In each generation, **fitness** of every individual is calculated.
- ➔ Several individuals are **selected** from the current population based on their fitness.
- ➔ These individu new individual
- ➔ Some individuals are **rded** from the new population (lowest fit s).

When does a genetic algorithm end?

- ➔ Reach a maximum number of generations.
- ➔ There is no change in the genetic material of the population

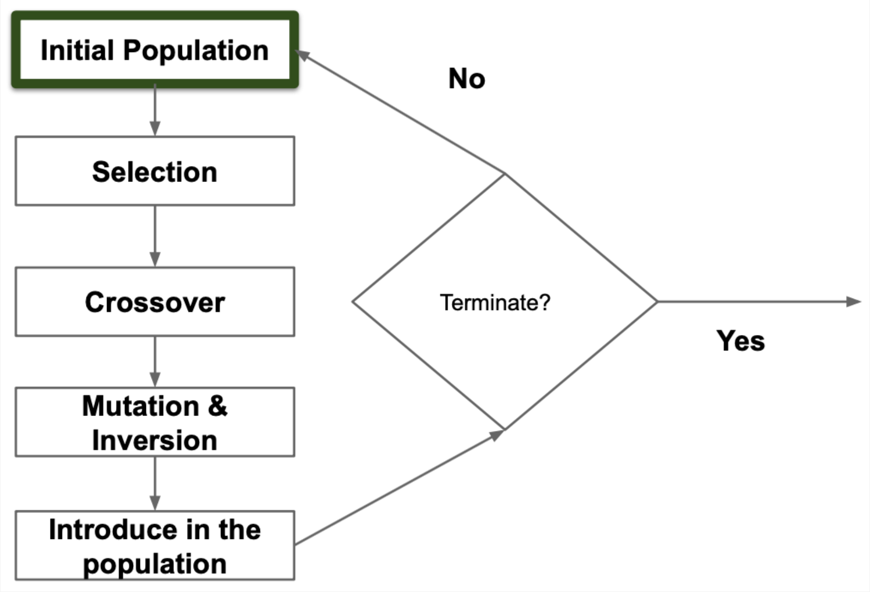
Both cases: *A suitable solution may or may not have reached*

Genetic Algorithms

INITIAL POPULATION

- ➔ Randomly generated, covering the entire search space.
- ➔ Population size depends on the problem (usually has several hundreds or thousands).
- ➔ Solution may be seeded in areas where optimal solutions can be found.
- ➔ An small population can give you a local maximum. A huge population requires too much computational resources.

STRATEGY

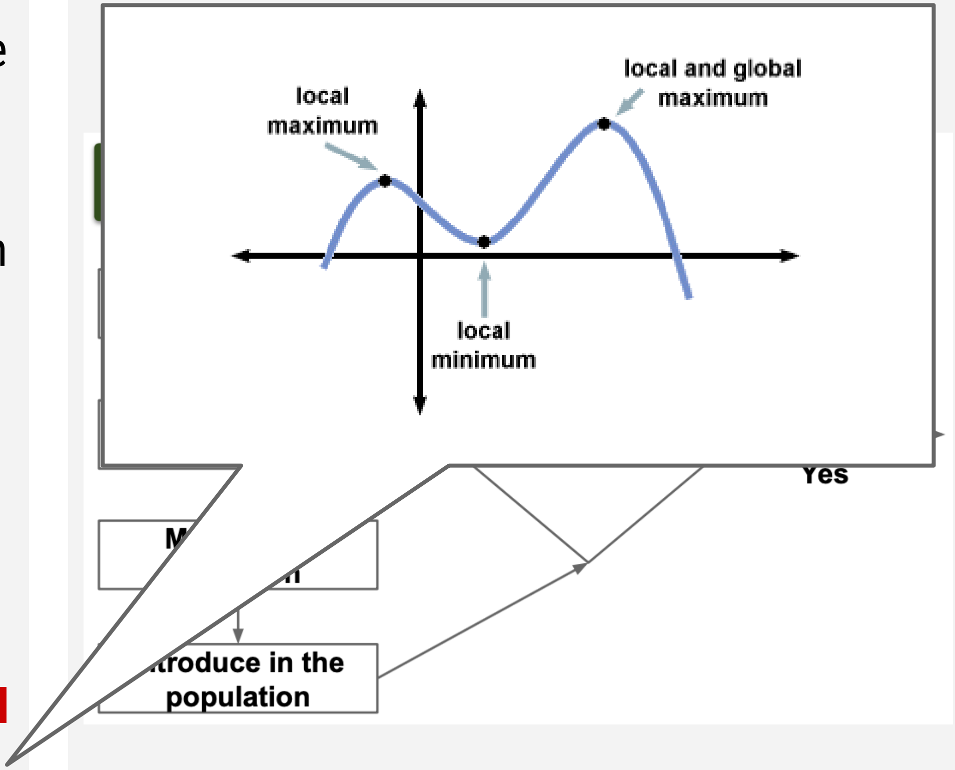


Genetic Algorithms

INITIAL POPULATION

- ➔ Randomly generated, covering the entire search space.
- ➔ Population size depends on the problem (usually has several hundreds or thousands).
- ➔ Solution may be seeded in areas where optimal solutions can be found.
- ➔ An small population can give you a **local maximum**. A huge population requires too much computational resources.

STRATEGY

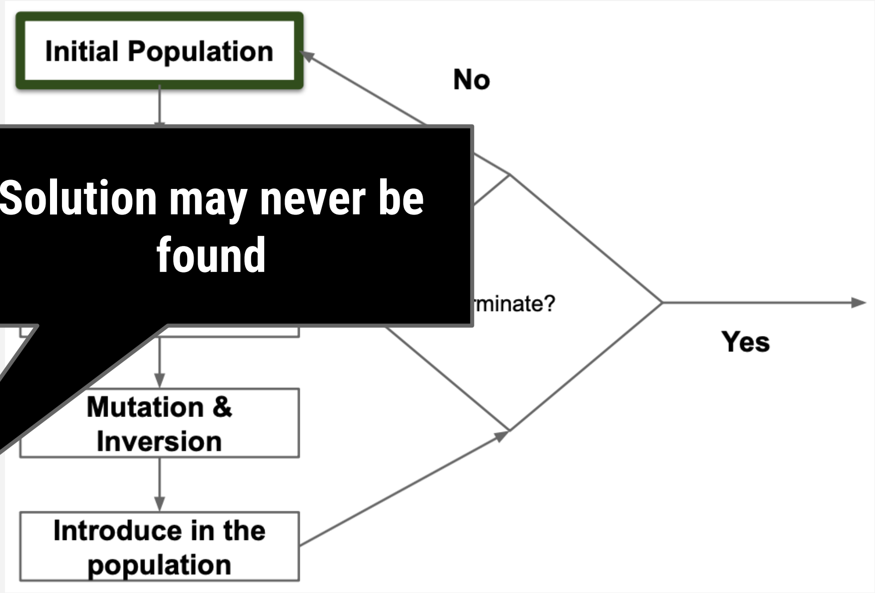


Genetic Algorithms

INITIAL POPULATION

- ➔ Randomly generated, covering the entire search space.
- ➔ Population size depends on the problem (usually has several hundreds or thousands).
- ➔ Solution may be seeded in areas where optimal solutions can be found.
- ➔ An small population can give you a **local maximum**. A huge population requires too much computational resources.

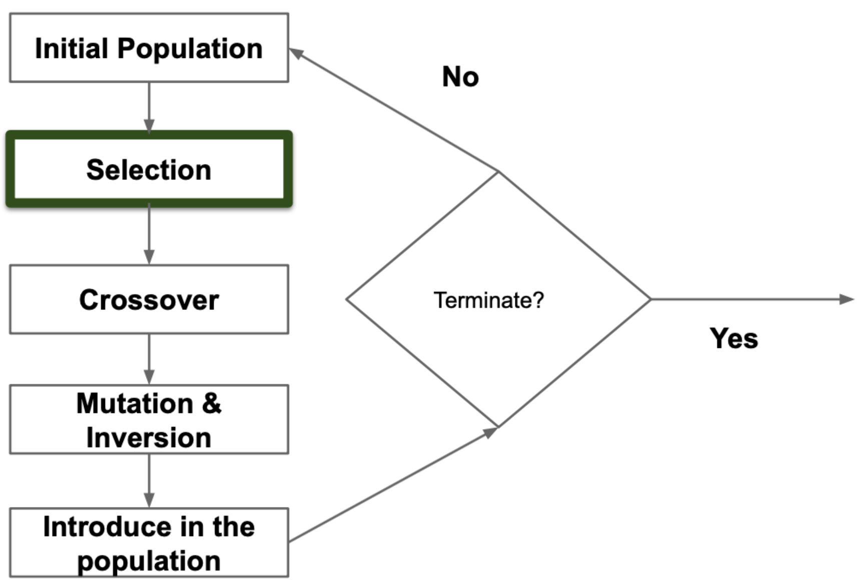
STRATEGY



Solution may never be found

Genetic Algorithms

STRATEGY



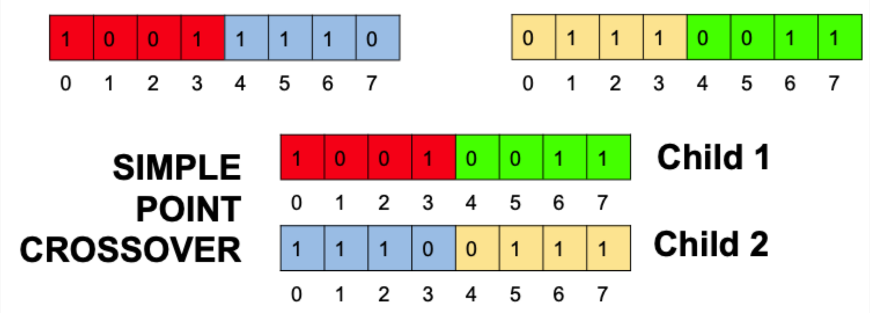
SELECTION

- ➔ During each generation you select a part of the population to create a new generation.
- ➔ Individuals are selected based on their fitness (proportional to the fitness).
- ➔ There are other methods to select individuals, for example, random.

Genetic Algorithms

REPRODUCTION

- ➔ Crossover, mutation and inversion.
- ➔ For each new solution, select a pair of parents (proportional to their fitness).
- ➔ Crossover:



1 0 0 1 1 1 1 0

0 1 2 3 4 5 6 7

0 1 1 1 0 0 1 1

0 1 2 3 4 5 6 7

SIMPLE POINT CROSSOVER

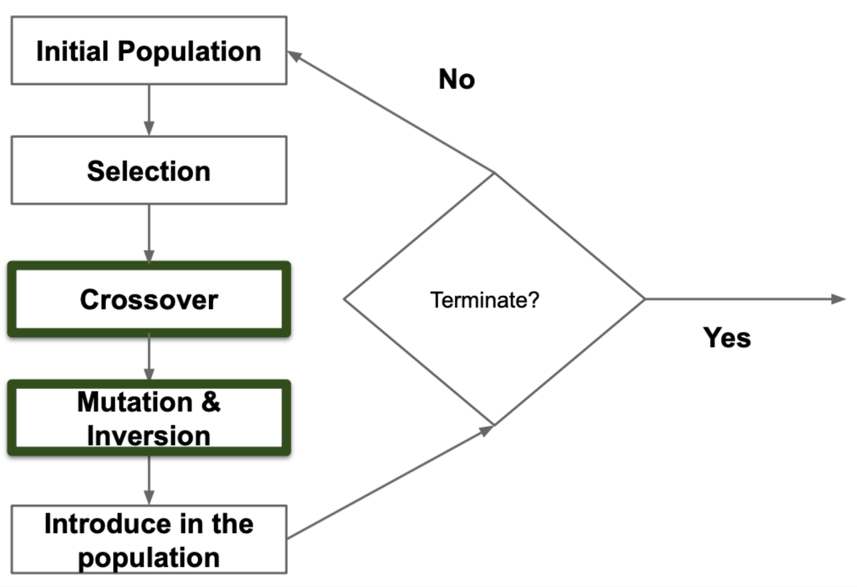
1 0 0 1 0 0 1 1 Child 1

0 1 2 3 4 5 6 7

1 1 1 0 0 1 1 1 Child 2

0 1 2 3 4 5 6 7

STRATEGY



Initial Population

Selection

Crossover

Mutation & Inversion

Introduce in the population

Terminate?

No

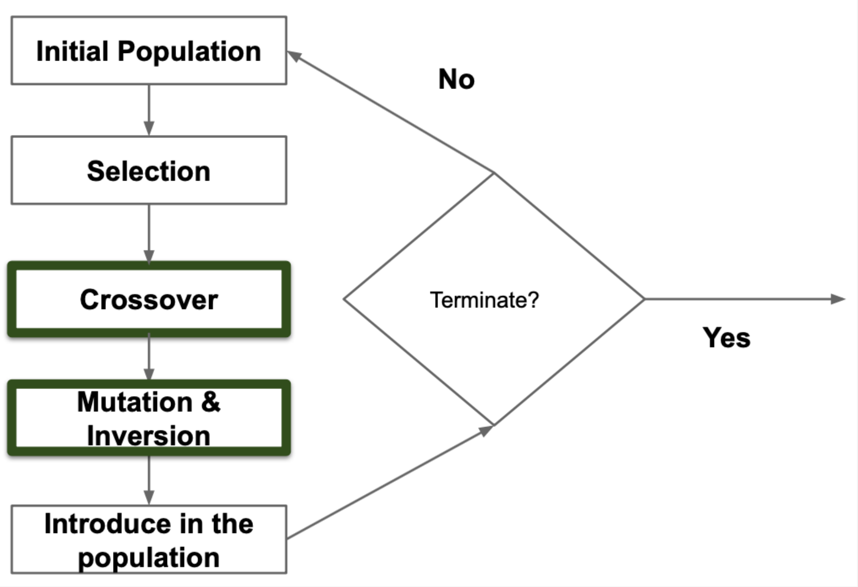
Yes

Genetic Algorithms

REPRODUCTION

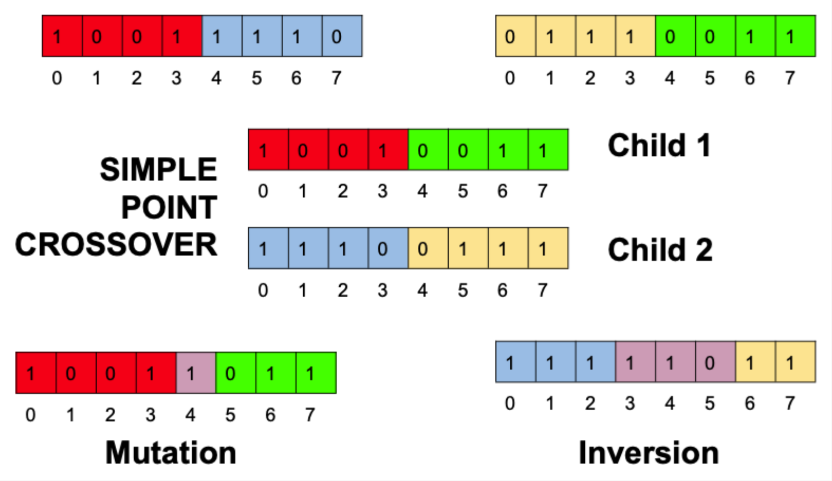
- ➔ Mutation (low probability) and Inversion (very low probability).
- ➔ In the mutation we select a random bit and add 1, discard the overflow.
- ➔ In the inversion select a random chain of bits and apply complement to this chain.

STRATEGY



Genetic Algorithms

REPRODUCTION



1	0	0	1	1	1	1	0
0	1	2	3	4	5	6	7

0	1	1	1	0	0	1	1
0	1	2	3	4	5	6	7

SIMPLE POINT CROSSOVER

Child 1

1	0	0	1	0	0	1	1
0	1	2	3	4	5	6	7

Child 2

1	1	1	0	0	1	1	1
0	1	2	3	4	5	6	7

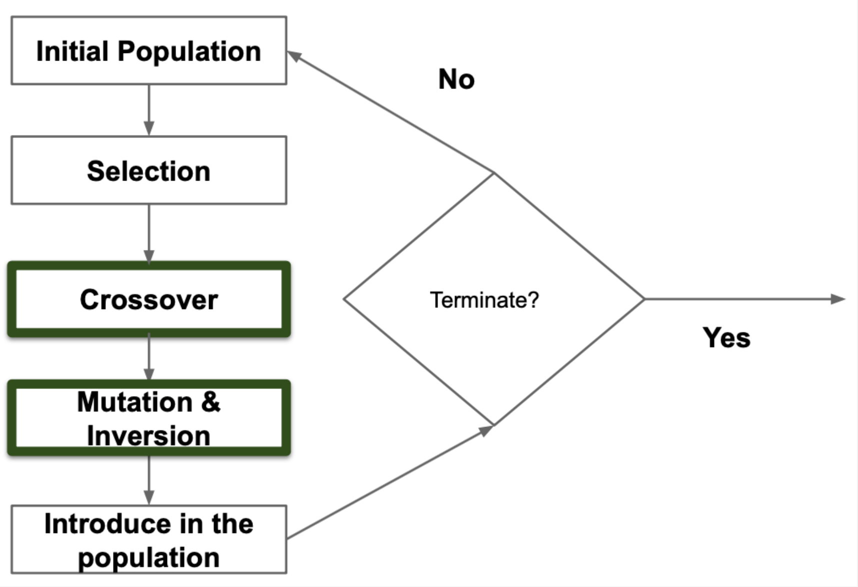
Mutation

1	0	0	1	1	0	1	1
0	1	2	3	4	5	6	7

Inversion

1	1	1	1	1	0	1	1
0	1	2	3	4	5	6	7

STRATEGY



Initial Population

Selection

Crossover

Mutation & Inversion

Introduce in the population

Terminate?

No

Yes

Genetic Algorithms (Example)

- Other examples
  - **Travelling Salesman Problem.** Given a collection of cities, determine the minimum cost route, visiting each city exactly once and returning to the starting point.
  - **Cashier Problem.** The same used in Dynamic Programming.

Greedy Algorithms

Greedy Algorithms

STRATEGY

- ➔ Builds up a solution in small steps, **choosing a decision at each step**.
- ➔ A greedy algorithm always makes the **choice that looks better at the moment**. Makes a locally optimal choice in the hope that this choice will lead to a global optimal solution.

CHARACTERISTICS

- ➔ In some cases it doesn’t provide an optimal solution.
- ➔ It can approximate an optimal solution.
- ➔ Optimal solution depends on an heuristic function.
- ➔ It is fast.

Greedy Algorithms

OPTIMAL SOLUTION

GREEDY

Greedy Algorithms

STRATEGY

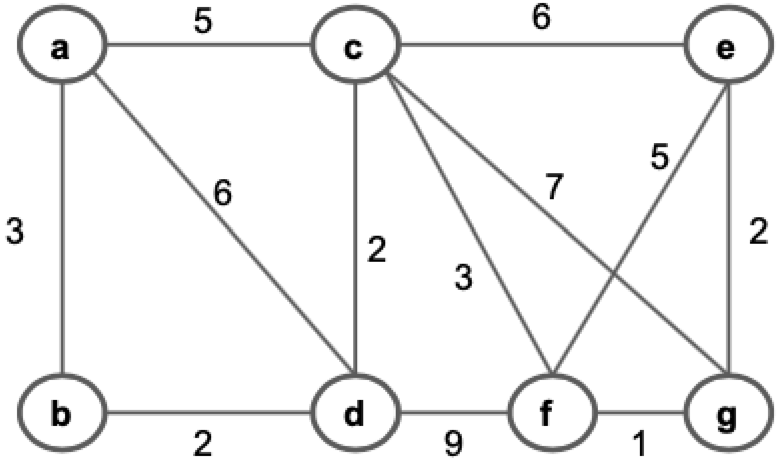
- ➔ A **candidate set** from which a solution is created.
- ➔ A **selection function**, which chooses the best candidate to be added to the solution.
- ➔ A **feasibility function**, that is used to determine if a candidate can be used to contribute to a solution.

STRATEGY

- ➔ An **objective function** which assigns a value to a solution or a partial solution.
- ➔ A **solution function** which will indicate when we have discovered a complete solution.

Greedy Algorithms (Examples)

➔ Dijkstra Algorithm

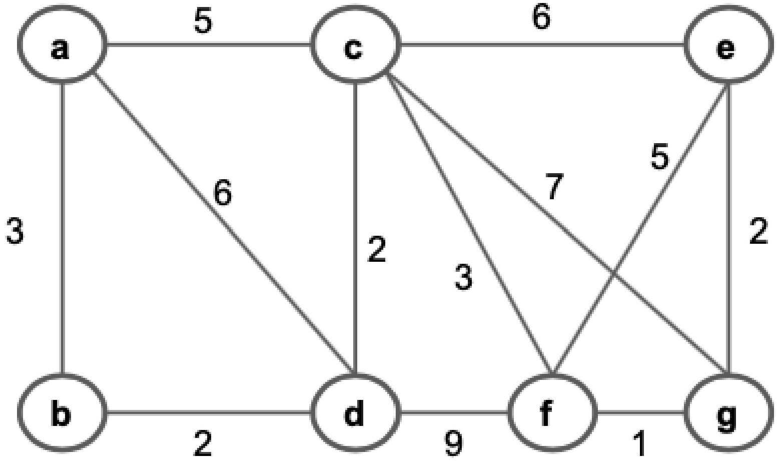


	Step 1	Step 2	Step 3	Step 4	Step 5	Step 6	Step 7
a	(0,a)	*	*	*	*	*	*
b							
c							
d							
e							
f							
g							

Find the shortest path from A to every other vertex

Greedy Algorithms (Examples)

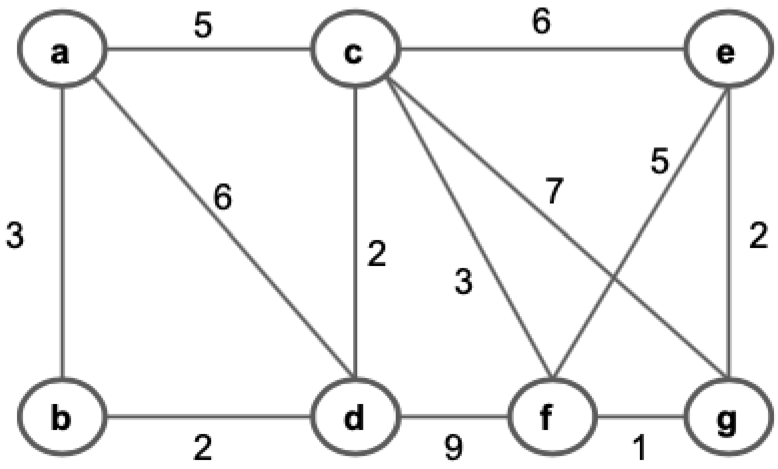
➔ Dijkstra Algorithm



	Step 1	Step 2	Step 3	Step 4	Step 5	Step 6	Step 7
a	(0,a)	*	*	*	*	*	*
b	(3,a)						
c	(5,a)						
d	(6,a)						
e	∞						
f	∞						
g	∞						

Greedy Algorithms (Examples)

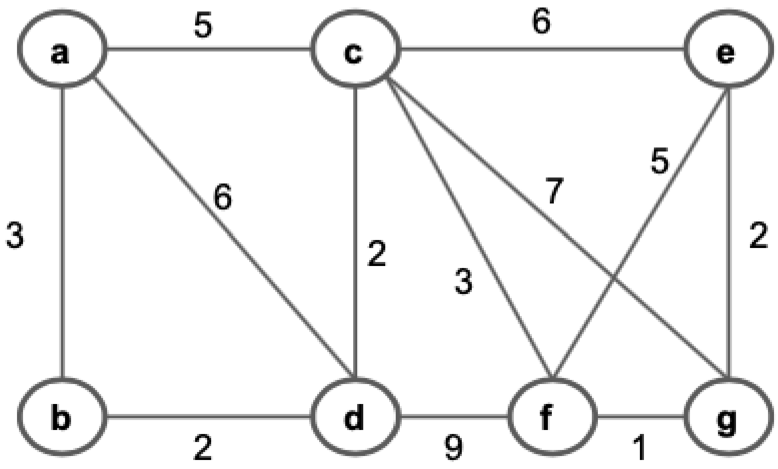
➔ Dijkstra Algorithm



	Step 1	Step 2	Step 3	Step 4	Step 5	Step 6	Step 7
a	(0,a)	*	*	*	*	*	*
b	(3,a)	(3,a)	*	*	*	*	*
c	(5,a)						
d	(6,a)						
e	∞						
f	∞						
g	∞						

Greedy Algorithms (Examples)

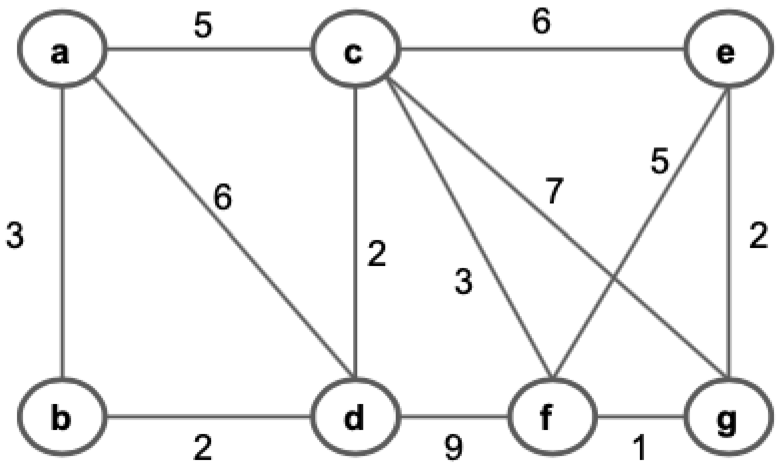
➔ Dijkstra Algorithm



	Step 1	Step 2	Step 3	Step 4	Step 5	Step 6	Step 7
a	(0,a)	*	*	*	*	*	*
b	(3,a)	(3,a)	*	*	*	*	*
c	(5,a)	(5,a)					
d	(6,a)	(5,b)					
e	∞	∞					
f	∞	∞					
g	∞	∞					

Greedy Algorithms (Examples)

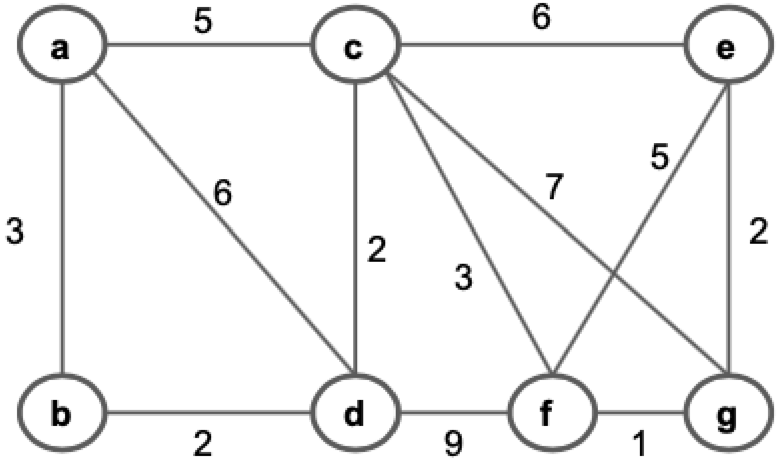
➔ Dijkstra Algorithm

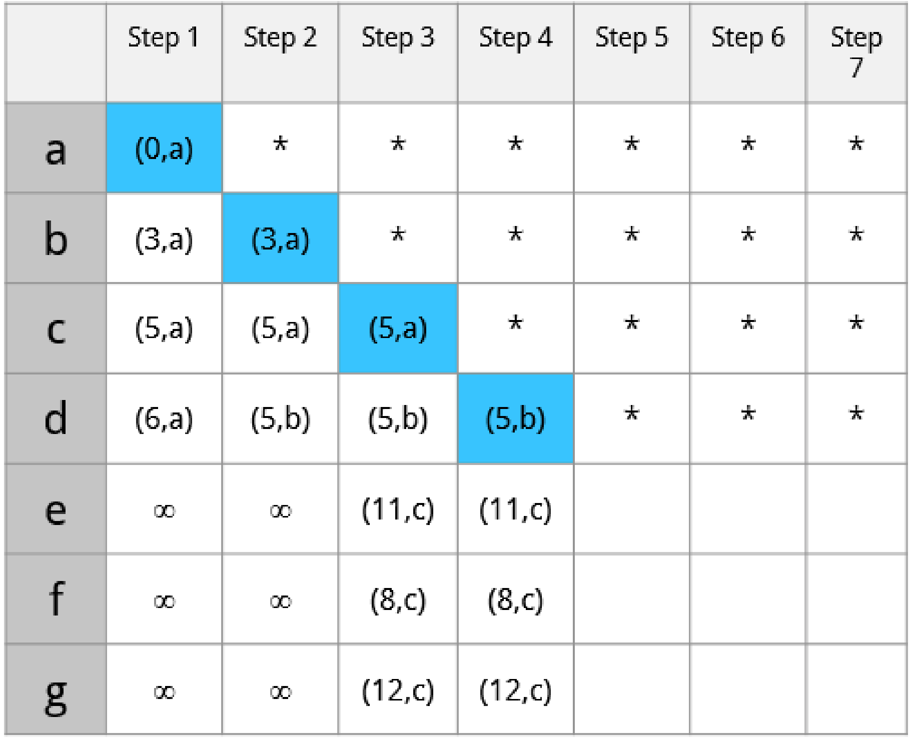


	Step 1	Step 2	Step 3	Step 4	Step 5	Step 6	Step 7
a	(0,a)	*	*	*	*	*	*
b	(3,a)	(3,a)	*	*	*	*	*
c	(5,a)	(5,a)	(5,a)	*	*	*	*
d	(6,a)	(5,b)	(5,b)				
e	∞	∞	(11,c)				
f	∞	∞	(8,c)				
g	∞	∞	(12,c)				

Greedy Algorithms (Examples)

➔ Dijkstra Algorithm

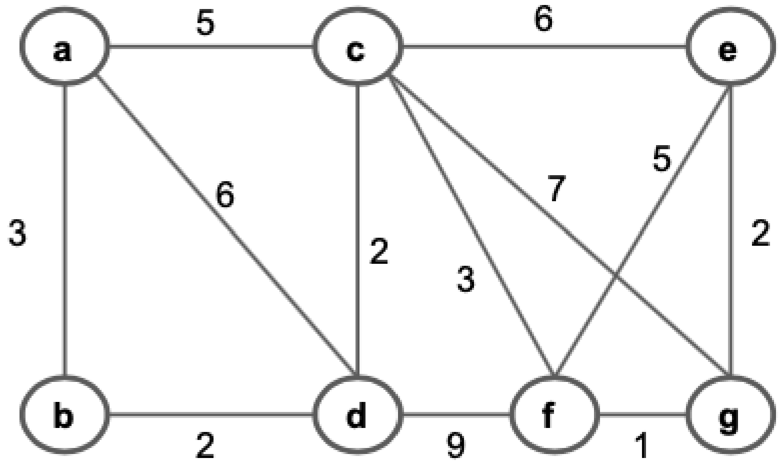




	Step 1	Step 2	Step 3	Step 4	Step 5	Step 6	Step 7
a	(0,a)	*	*	*	*	*	*
b	(3,a)	(3,a)	*	*	*	*	*
c	(5,a)	(5,a)	(5,a)	*	*	*	*
d	(6,a)	(5,b)	(5,b)	(5,b)	*	*	*
e	∞	∞	(11,c)	(11,c)			
f	∞	∞	(8,c)	(8,c)			
g	∞	∞	(12,c)	(12,c)			

Greedy Algorithms (Examples)

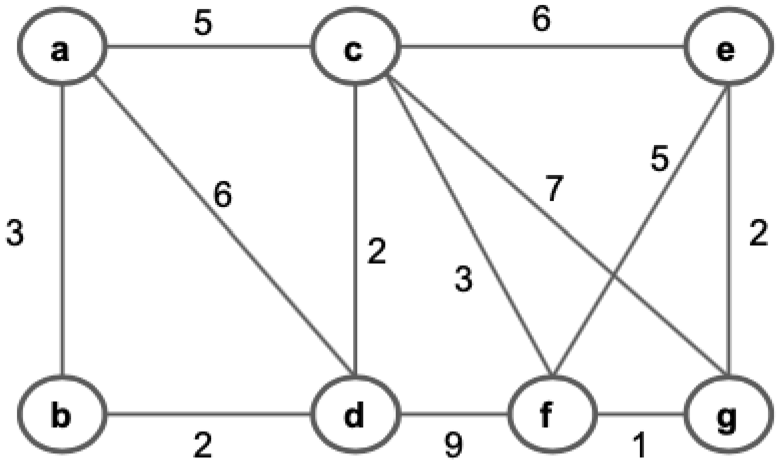
➔ Dijkstra Algorithm



	Step 1	Step 2	Step 3	Step 4	Step 5	Step 6	Step 7
a	(0,a)	*	*	*	*	*	*
b	(3,a)	(3,a)	*	*	*	*	*
c	(5,a)	(5,a)	(5,a)	*	*	*	*
d	(6,a)	(5,b)	(5,b)	(5,b)	*	*	*
e	∞	∞	(11,c)	(11,c)	(11,c)		
f	∞	∞	(8,c)	(8,c)	(8,c)	*	*
g	∞	∞	(12,c)	(12,c)	(9,f)		

Greedy Algorithms (Examples)

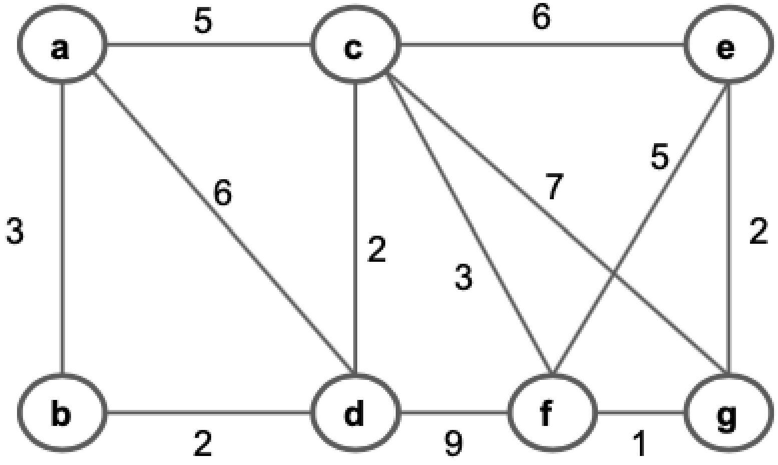
➔ Dijkstra Algorithm

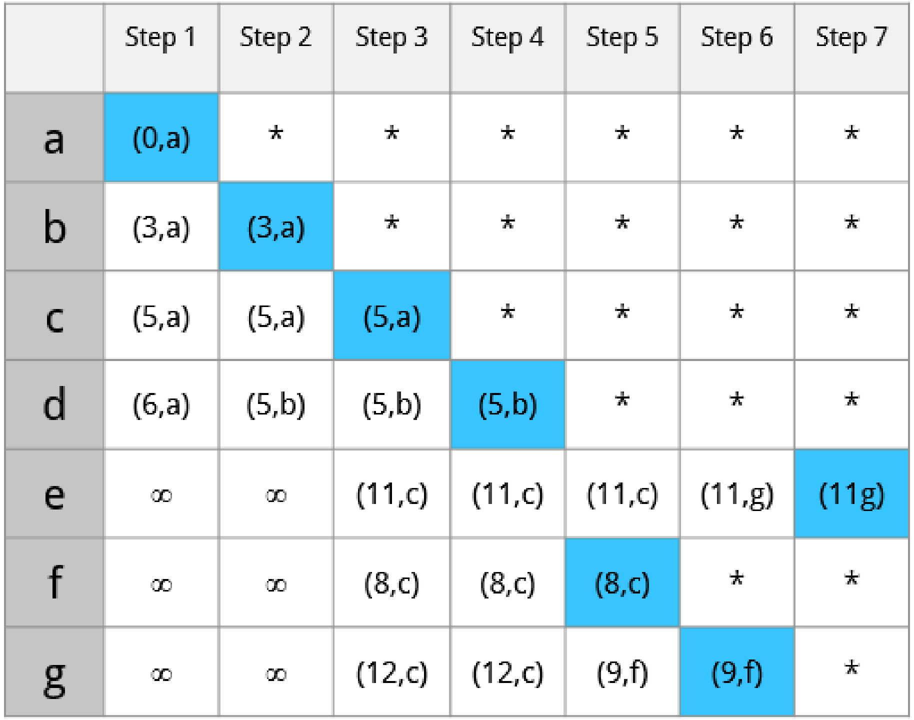


	Step 1	Step 2	Step 3	Step 4	Step 5	Step 6	Step 7
a	(0,a)	*	*	*	*	*	*
b	(3,a)	(3,a)	*	*	*	*	*
c	(5,a)	(5,a)	(5,a)	*	*	*	*
d	(6,a)	(5,b)	(5,b)	(5,b)	*	*	*
e	∞	∞	(11,c)	(11,c)	(11,c)	(11,g)	(11g)
f	∞	∞	(8,c)	(8,c)	(8,c)	*	*
g	∞	∞	(12,c)	(12,c)	(9,f)	(9,f)	*

Greedy Algorithms (Examples)

- → Dijkstra Algorithm

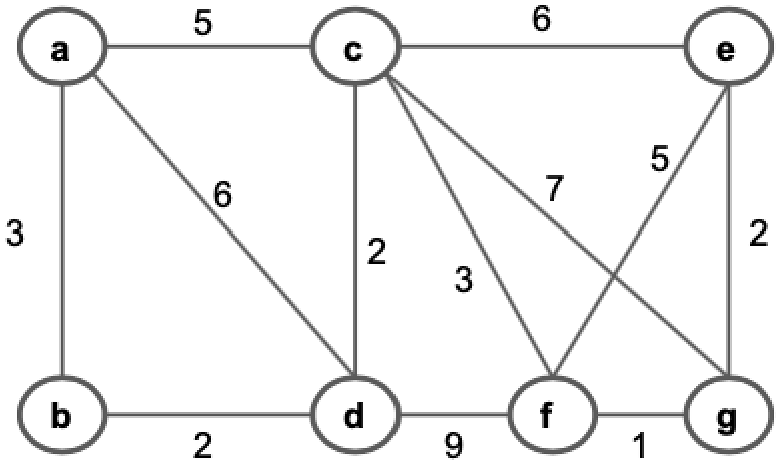




	Step 1	Step 2	Step 3	Step 4	Step 5	Step 6	Step 7
a	(0,a)	*	*	*	*	*	*
b	(3,a)	(3,a)	*	*	*	*	*
c	(5,a)	(5,a)	(5,a)	*	*	*	*
d	(6,a)	(5,b)	(5,b)	(5,b)	*	*	*
e	∞	∞	(11,c)	(11,c)	(11,c)	(11,g)	(11g)
f	∞	∞	(8,c)	(8,c)	(8,c)	*	*
g	∞	∞	(12,c)	(12,c)	(9,f)	(9,f)	*

Greedy Algorithms (Examples)

→ Dijkstra Algorithm



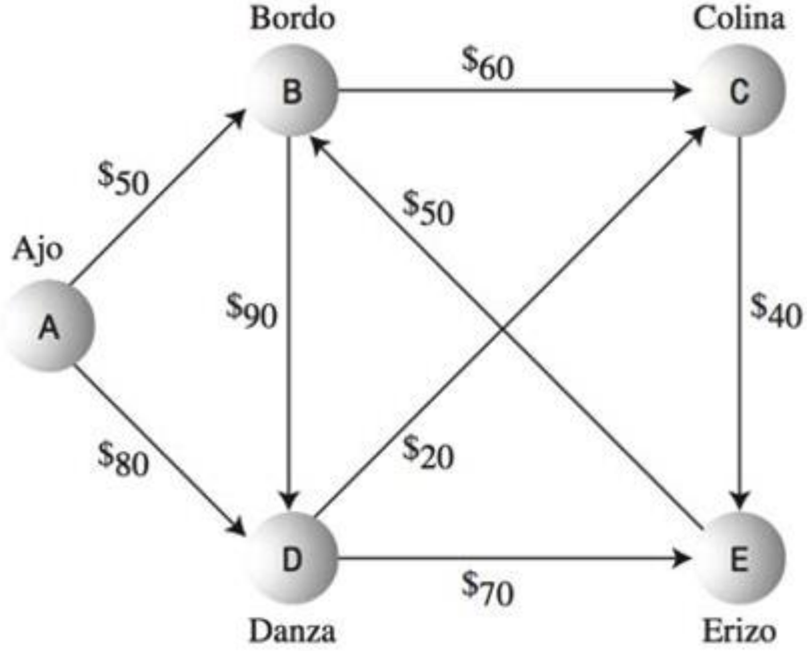
	Step 1	Step 2	Step 3	Step 4	Step 5	Step 6	Step 7
a	(0,a)	*	*	*	*	*	*
b	(3,a)	(3,a)	*	*	*	*	*
c	(5,a)	(5,a)	(5,a)	*	*	*	*
d	(6,a)	(5,b)	(5,b)	(5,b)	*	*	*
e	∞	∞	(11,c)	(11,c)	(11,c)	(11,g)	(11g)
f	∞	∞	(8,c)	(8,c)	(8,c)	*	*
g	∞	∞	(12,c)	(12,c)	(9,f)	(9,f)	*

The shortest path from A to G is:

A -> C -> F -> G

Greedy Algorithms (Examples)

➔ Dijkstra Algorithm

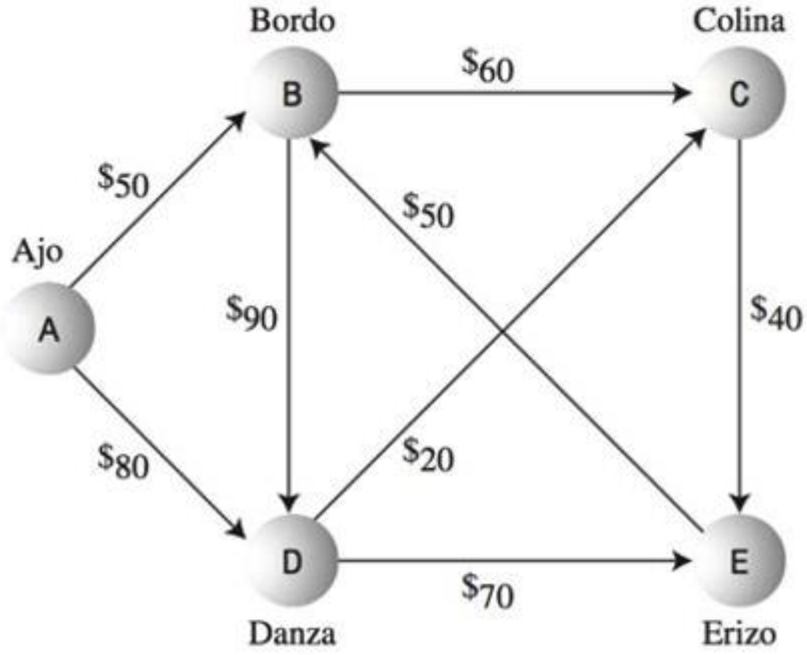


	Step 1	Step 2	Step 3	Step 4	Step 5	Step 6	Step 7
a	(0,a)	*	*	*	*	*	*
b							
c							
d							
e							
f							
g							

Find the shortest path from A to every other vertex

Greedy Algorithms (Examples)

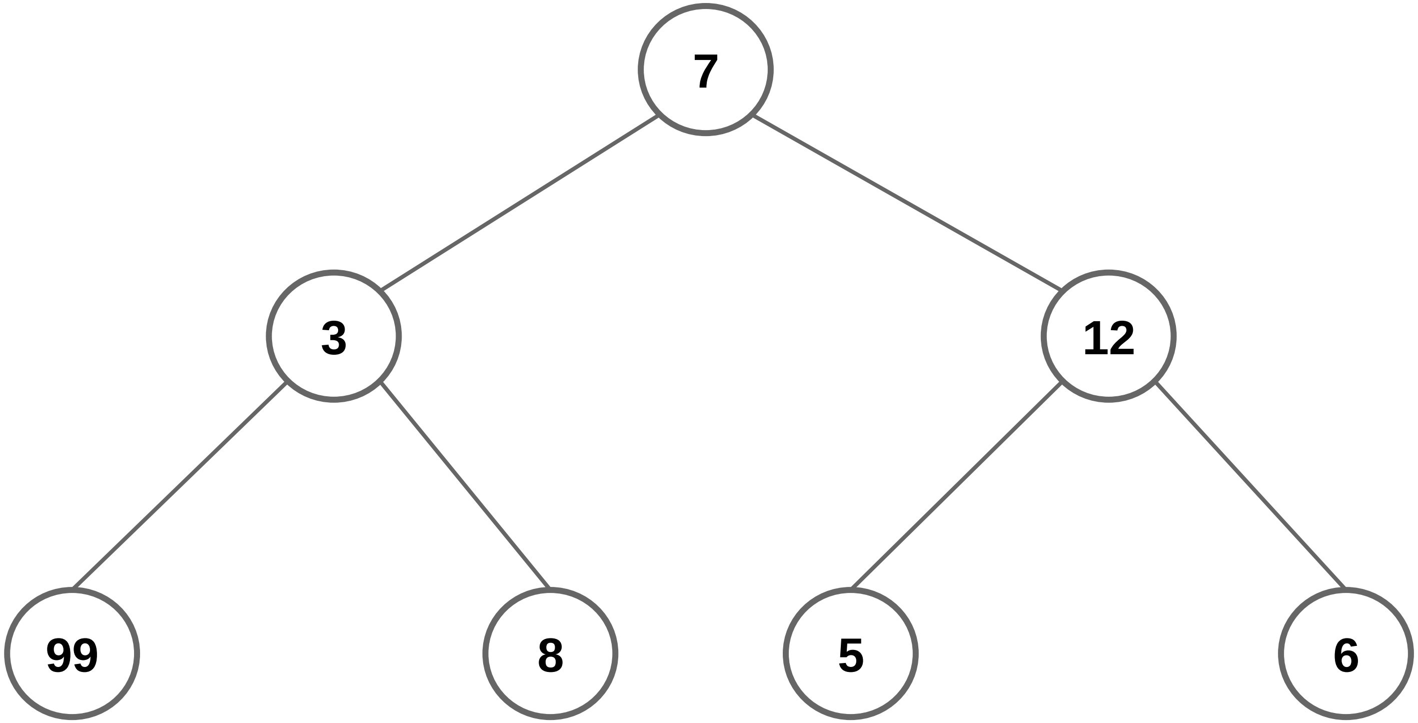
➔ Dijkstra Algorithm



	Step 1	Step 2	Step 3	Step 4	Step 5
A	(0, A)	*	*	*	*
B	(50, A)	(50, A)	*	*	*
C	∞	(110, B)	(100, D)	(100, D)	*
D	(80, A)	(80, A)	(80, A)	*	*
E	∞	∞	(150, D)	(140, C)	(140, C)

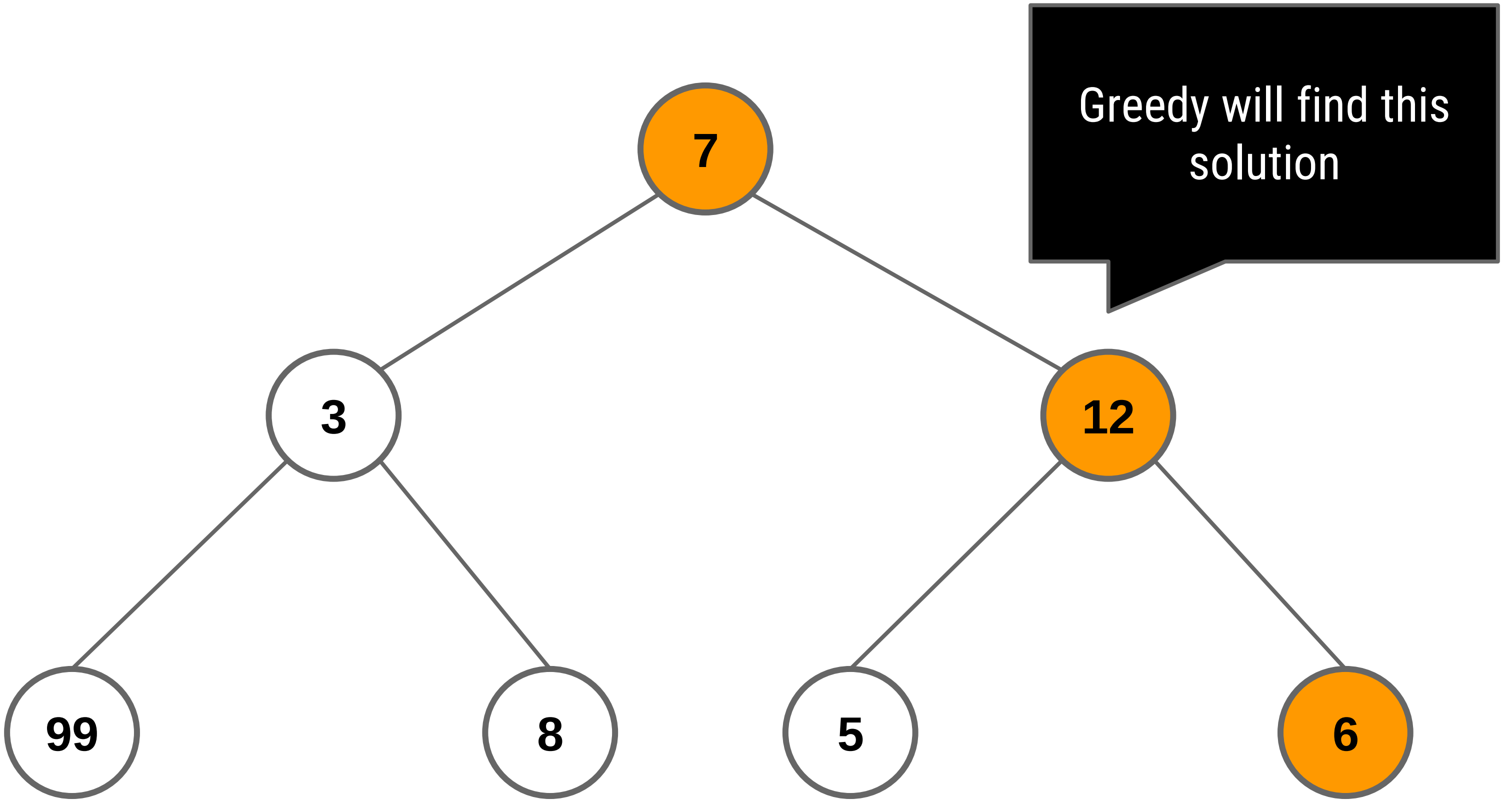
Greedy Algorithms (Examples)

- ➔ Find largest sum



Greedy Algorithms (Examples)

- ➔ Find largest sum

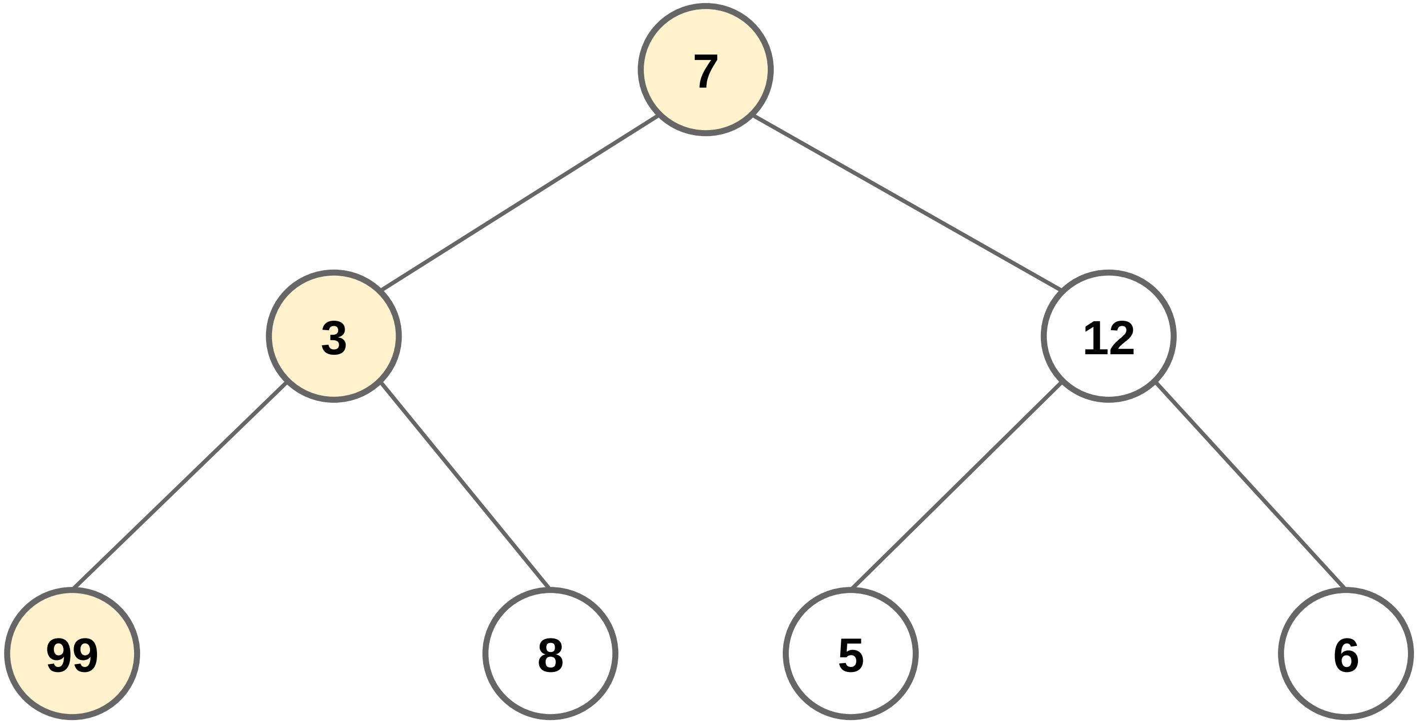


Greedy will find this solution

Greedy Algorithms (Examples)

But this is the best solution...

➔ Find largest sum



Greedy Algorithms (Example)

➔ Cashier Algorithm

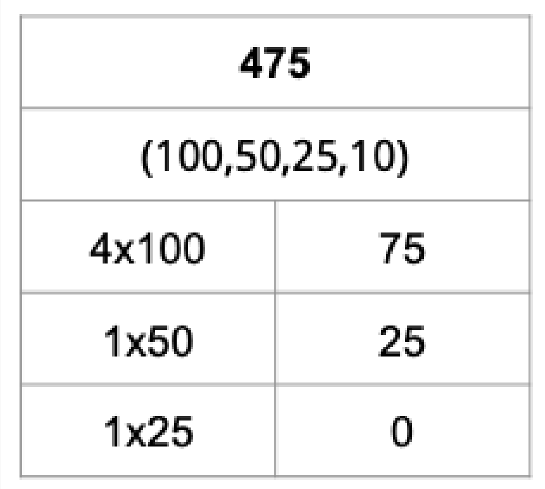
CASE

1. **475 (100,50,25,10)** ⬅

1. 279 (100,50,25,10,1)

1. 7 (1,3,4,5)

SOLUTION



475	
(100,50,25,10)	
4x100	75
1x50	25
1x25	0

Greedy Algorithms (Example)

➔ Cashier Algorithm

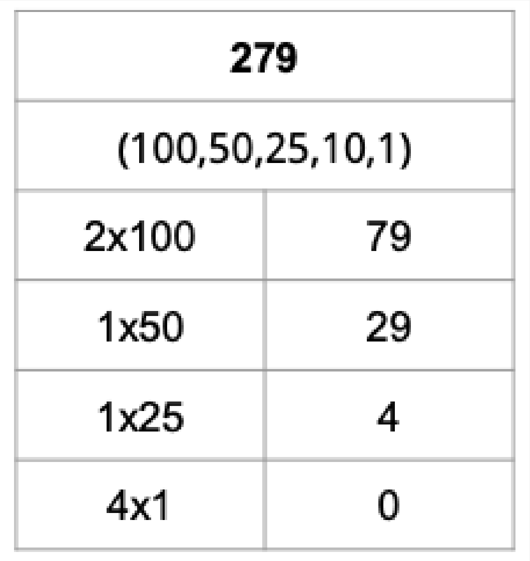
CASE

1. 475 (100,50,25,10)

1. **279 (100,50,25,10,1)**

1. 7 (1,3,4,5)

SOLUTION



279	
(100,50,25,10,1)	
2x100	79
1x50	29
1x25	4
4x1	0

Greedy Algorithms (Example)

➔ Cashier Algorithm

CASE

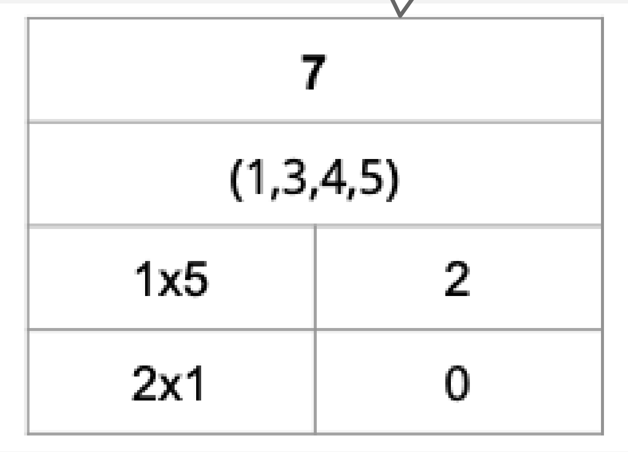
1. 475 (100,50,25,10)

1. 279 (100,50,25,10,1)

1. **7 (1,3,4,5)**

SO

But this is the best solution...

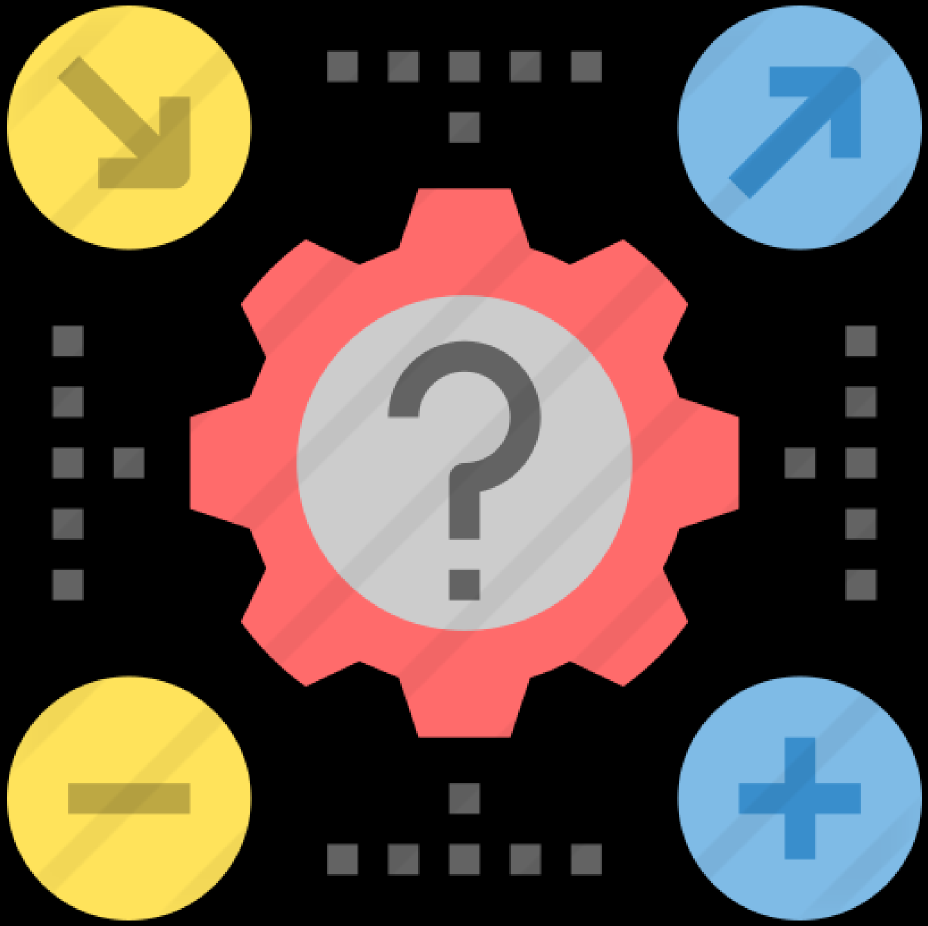


7	
(1,3,4,5)	
1x5	2
2x1	0

Greedy Algorithms (Example)

- Other examples
  - **Kruskal and Prim Minimum Spanning Tree.** Both algorithms creates a Minimum Spanning Tree by picking edges one by one. The difference is the criterion for selecting the edge.
  - **Huffman Coding.** Is a loss-less compression technique (more on this later).
  - **Traveling Salesman Problem.** Given a collection of cities, determine the minimum cost route, visiting each city exactly once and returning to the starting point.

Probabilistic Algorithms



Probabilistic Algorithms

- ➔ Algorithms that uses random numbers to decide what to do next.

- ➔ Finding the optimal solution requires too much time or resources.

- ➔ Even with the same input, the algorithm behavior will vary on each execution. Running time and result may vary too.

Probabilistic Algorithms

- Probabilistic vs Deterministic
  - Could fail or never end running as long as the probability for those events is too low. 😠
  - Can find different solutions for the same input 🙂
- Even if an algorithm is deterministic, that does not guarantee that the solution is accurate, why?

Probabilistic Algorithms

➔ Probabilistic vs Deterministic

◆ Could fail or never end running as long as the probability for those events is too low. 😠

◆ Can find different solutions for the same input 🙂

➔ Even if an algorithm is deterministic, that does not guarantee that the solution is accurate, why?

Hardware failure

Probabilistic Algorithms

- ➔ Categories

LAS VEGAS	MONTE CARLO
➔ These algorithms always produce correct or optimum results. ➔ If not find the correct answer report the failure.	➔ Produce correct or optimum result with some probability. ➔ May return an answer that is not correct.

Probabilistic Algorithms (Example)

- ➔ Other examples
  - ◆ **Hashing.**
  - ◆ **Sorting.**
  - ◆ **Searching.**
  - ◆ **Load Balancing.**
  - ◆ **Minimum spanning trees.**
  - ◆ **Shortest paths.**

Geek Corner

Some thoughts on web development

- ➔ What is a web page?
- ➔ What is a web application?
- ➔ What are the components of a web application?
- ➔ What is client-side?
- ➔ What is server-side?

Geek Corner

Some thoughts on web development

➔ Ajax?

➔ Web frameworks? Toolkits?

➔ Web server?

➔ Web application server?

Geek Corner

Some thoughts on web development

- ➔ A **web page** is any document that is suitable for the world wide web and a web browser

- ➔ A web page is written in some kind of code (HTML, XML, JS) that the browser can parse and render to the user

- ➔ You can talk of **static pages** which the content is static and cannot change unless you edit the actual page

- ➔ **Dynamic pages** can change the content depending of some logic. These are generated by the server

Geek Corner

Some thoughts on web development

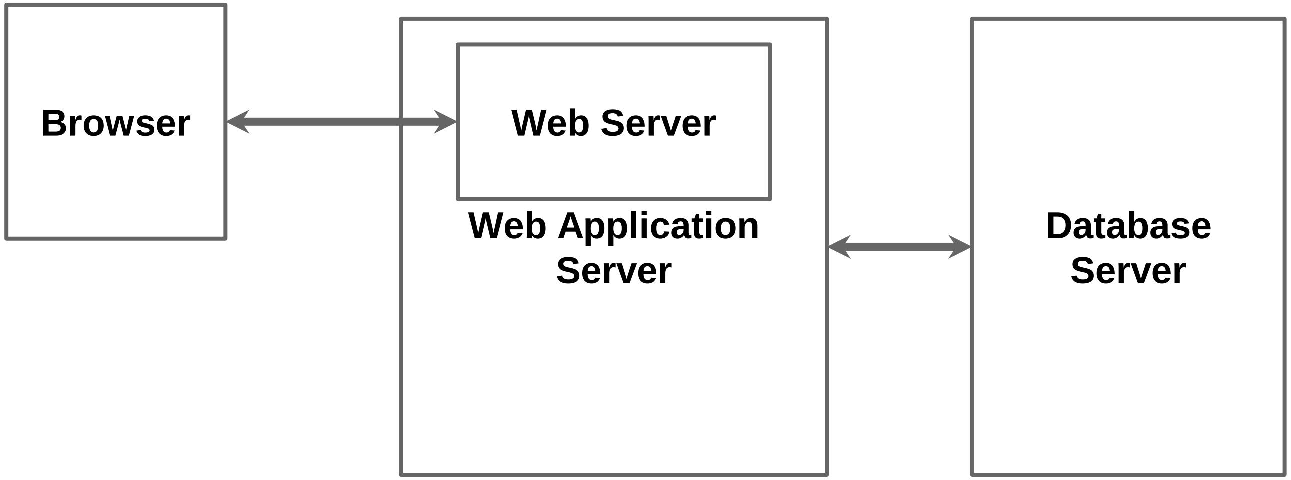
- A **web application** is software. Software that can run on a web browser.

- As any other well-written software application, a web application is composed of layers:
  - Presentation or UI layer.
  - Business or logic layer.
  - Data layer.

- To run the complete application, many components are needed:
  - Browser
  - DB server
  - Application server and optionally web server.

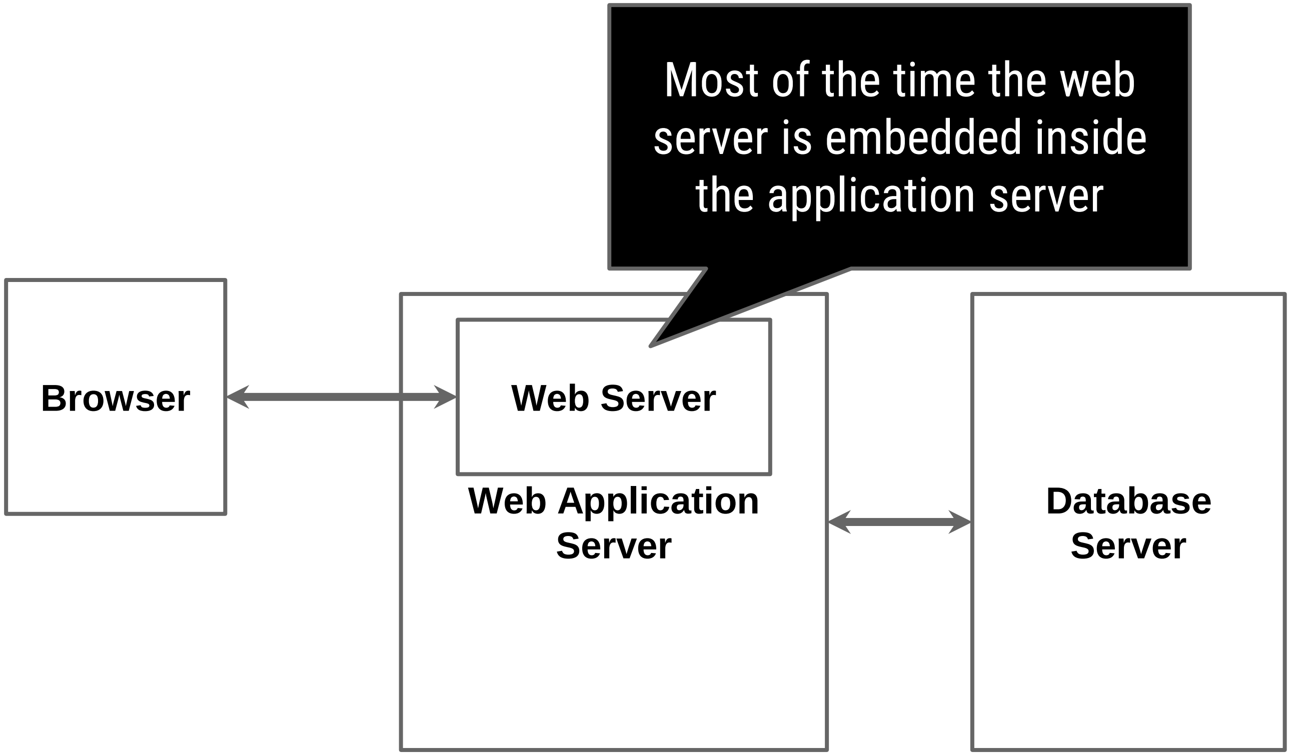
Geek Corner

Some thoughts on web development



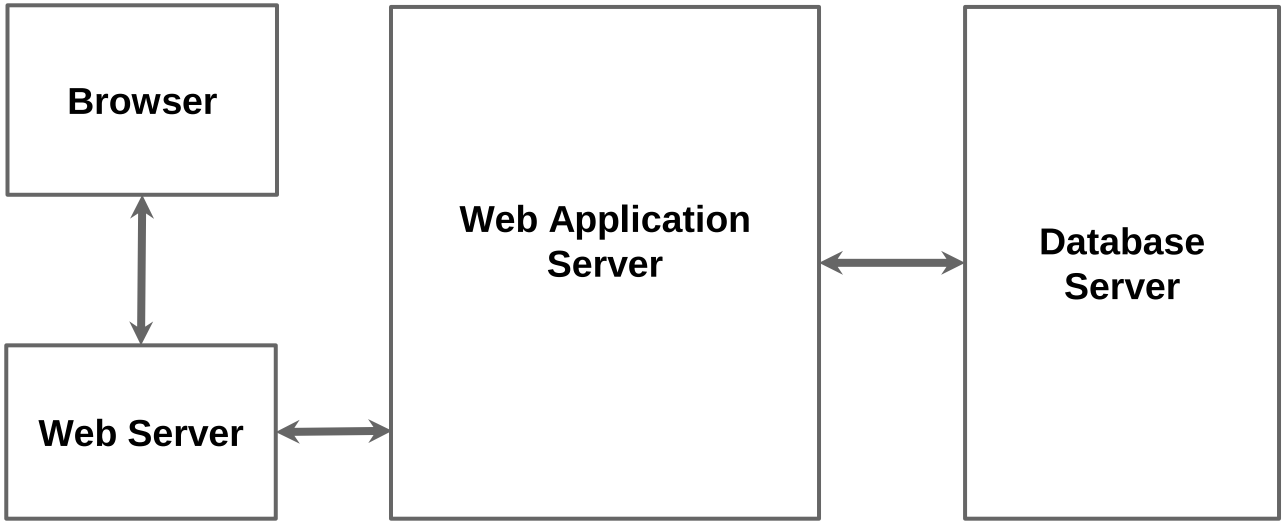
Geek Corner

Some thoughts on web development


Most of the time the web server is embedded inside the application server

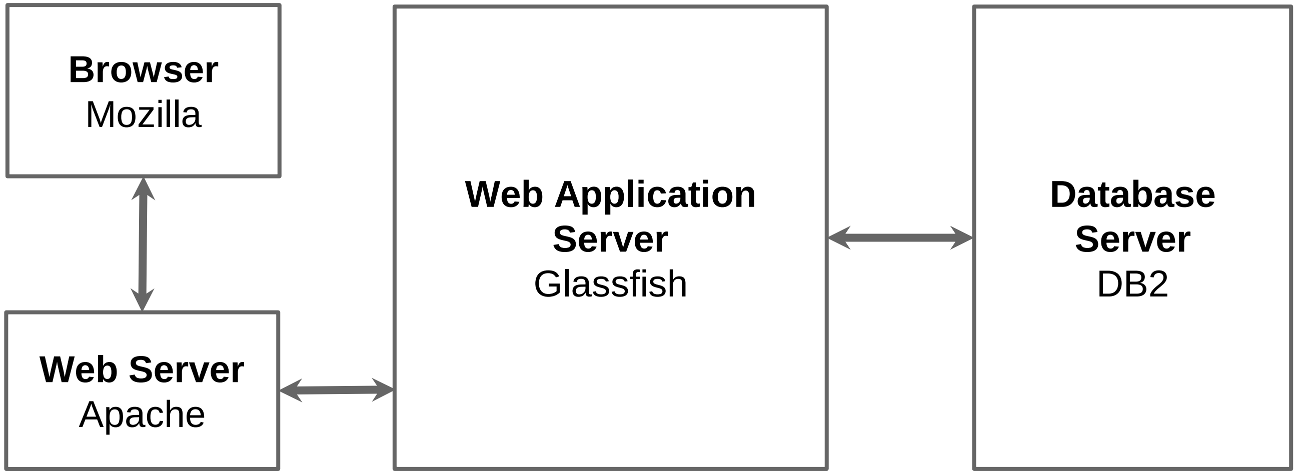
Geek Corner

Some thoughts on web development



Geek Corner

Some thoughts on web development



Browser
Mozilla

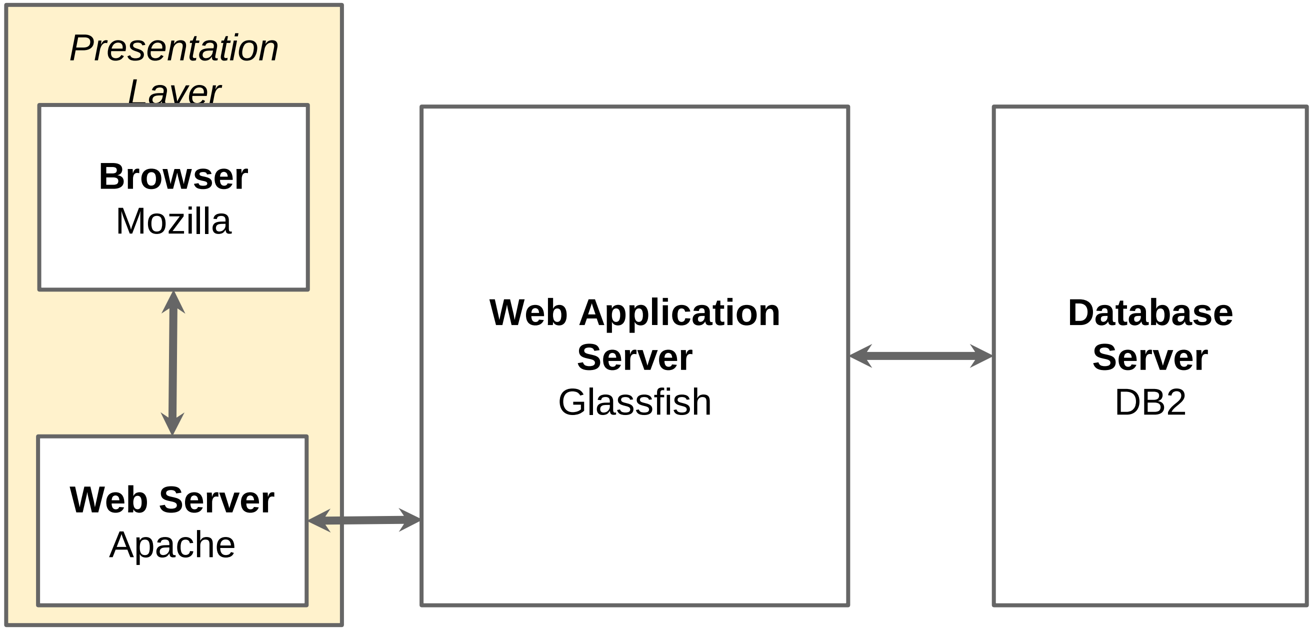
Web Server
Apache

Web Application Server
Glassfish

Database Server
DB2

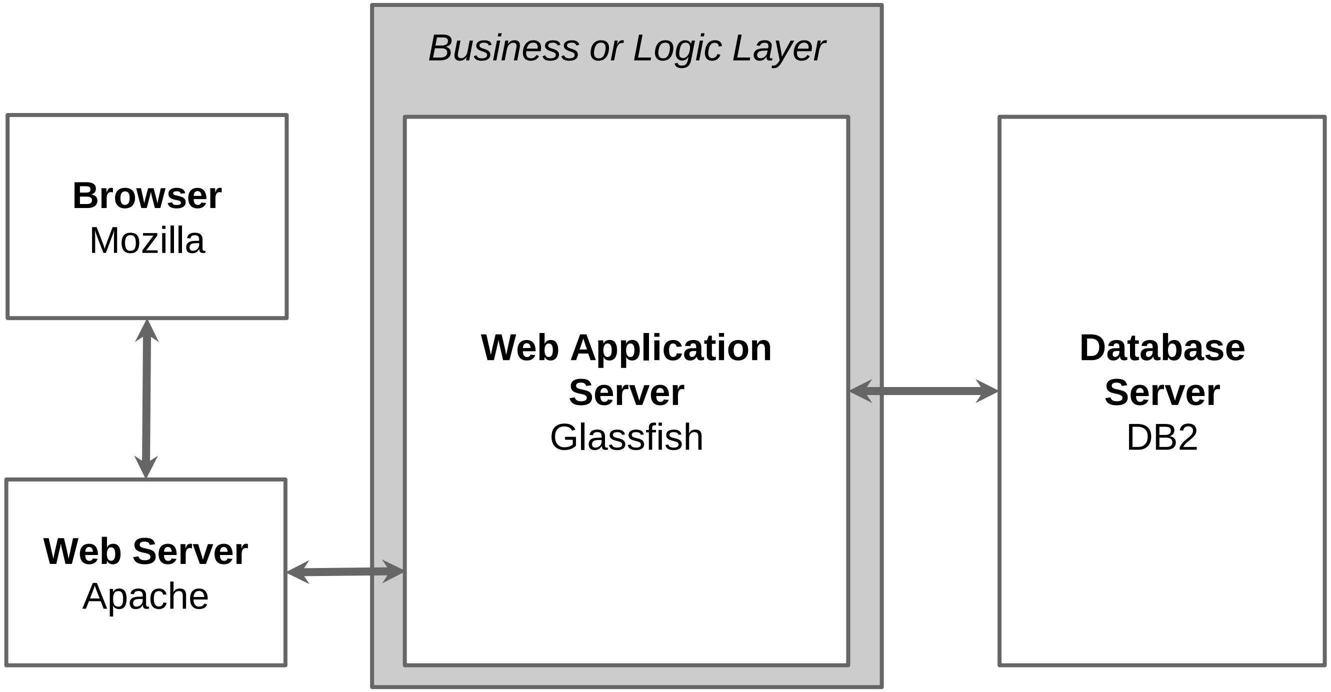
Geek Corner

Some thoughts on web development



Geek Corner

Some thoughts on web development



Business or Logic Layer

Browser
Mozilla

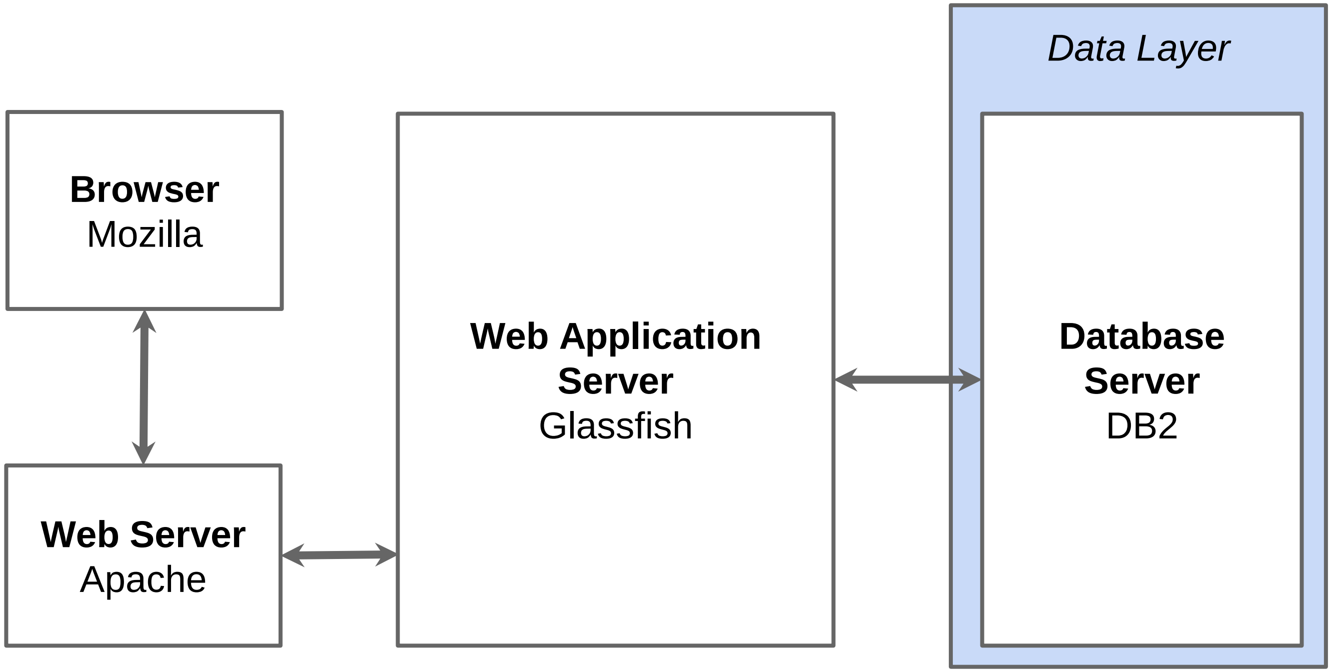
Web Server
Apache

Web Application Server
Glassfish

Database Server
DB2

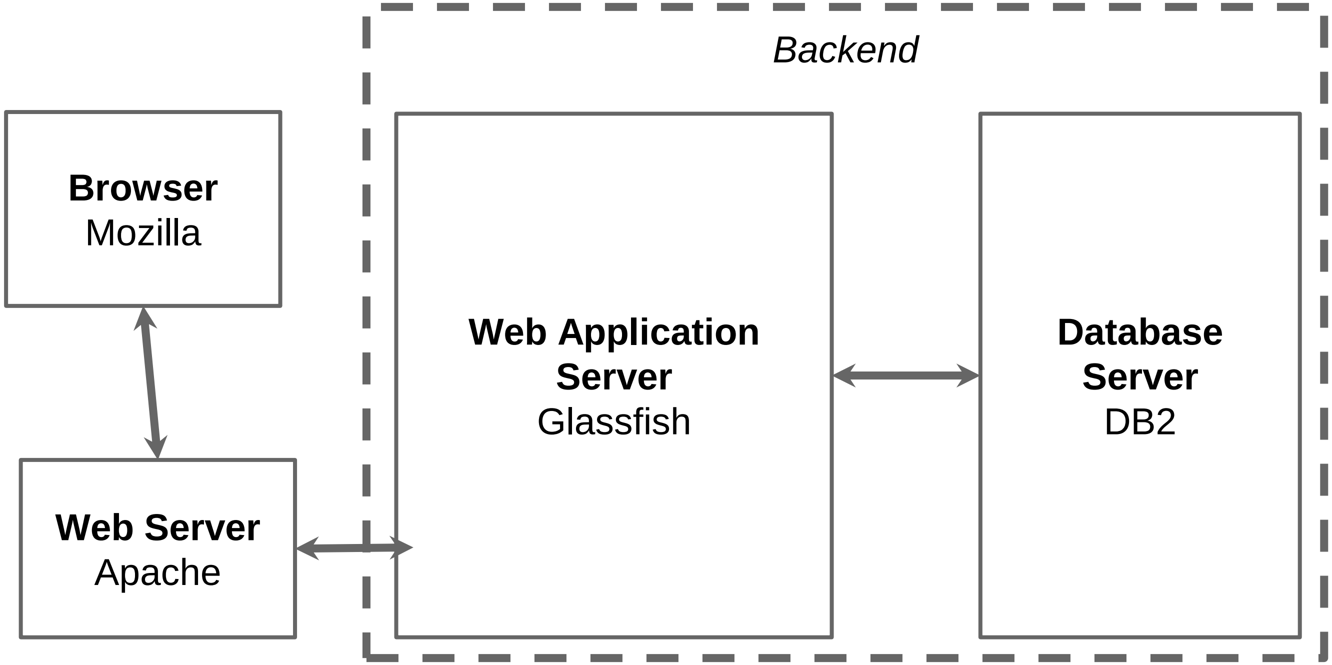
Geek Corner

Some thoughts on web development



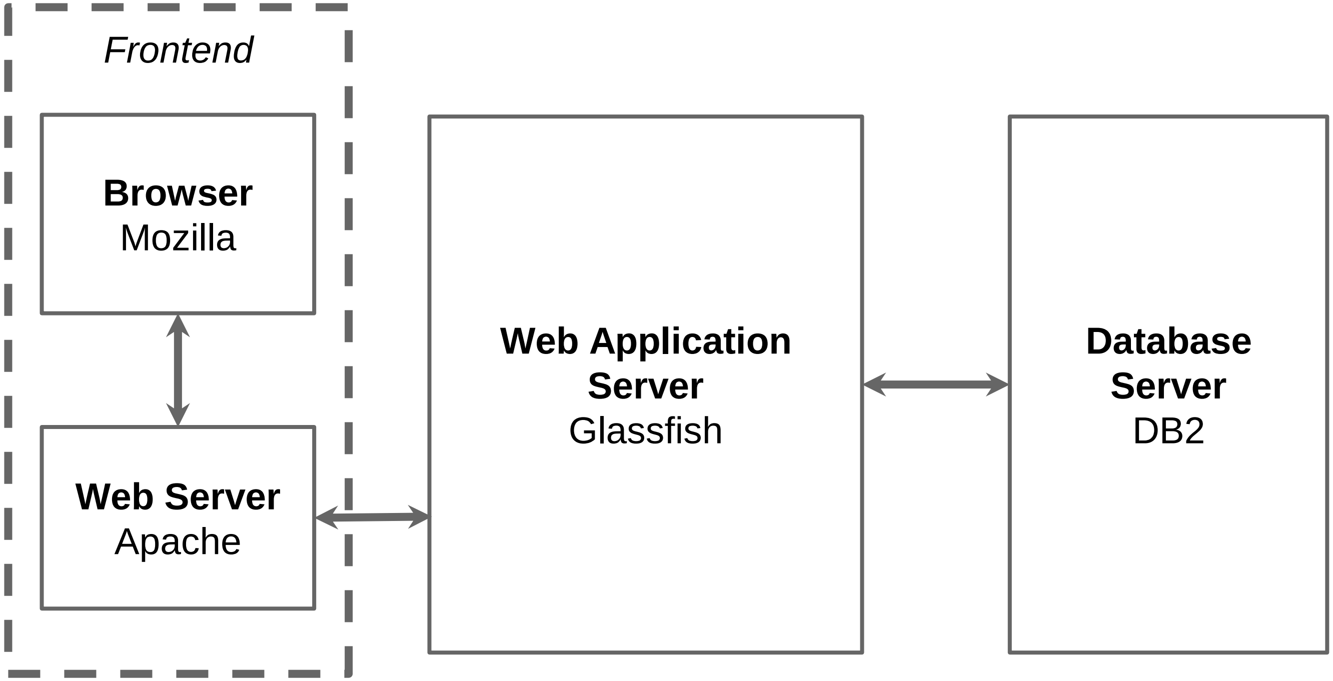
Geek Corner

Some thoughts on web development



Geek Corner

Some thoughts on web development



Frontend

Browser
Mozilla

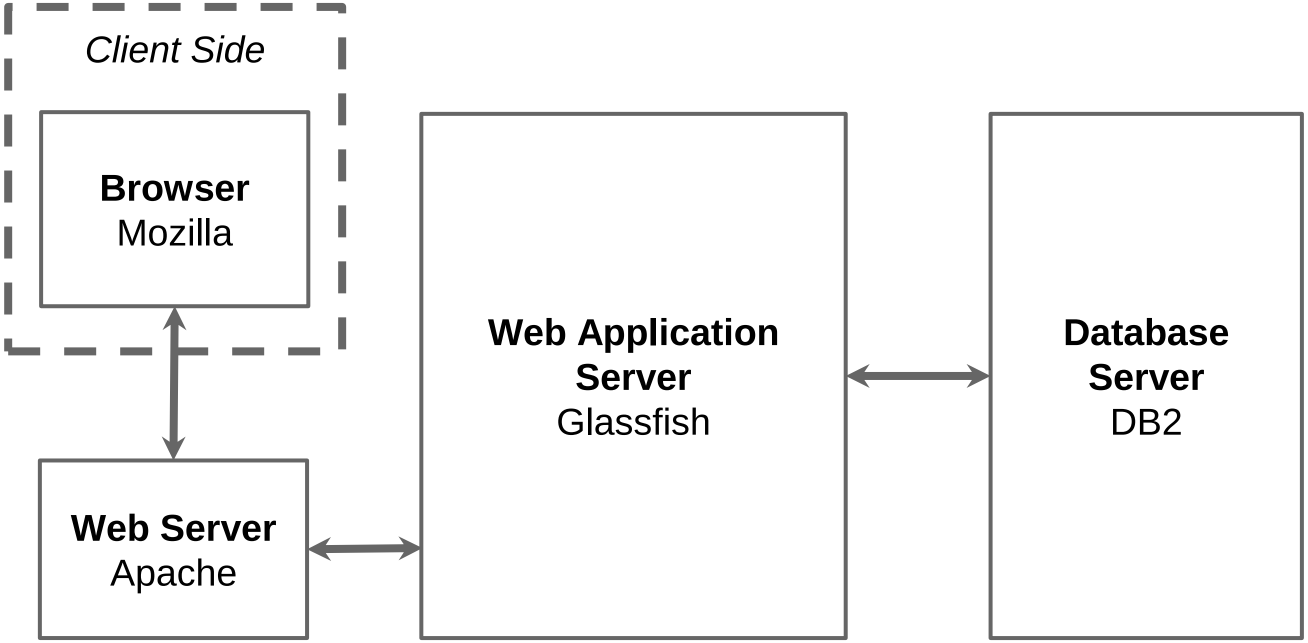
Web Server
Apache

Web Application Server
Glassfish

Database Server
DB2

Geek Corner

Some thoughts on web development


Client Side

Browser
Mozilla

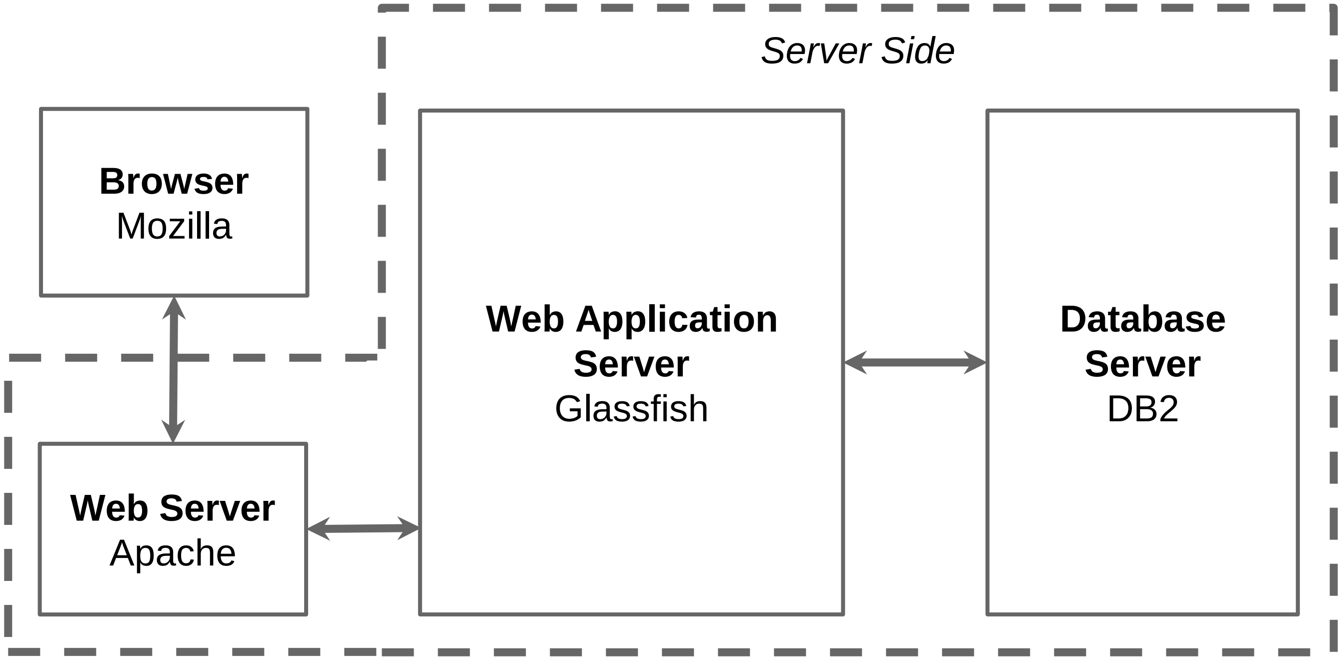
Web Server
Apache

Web Application Server
Glassfish

Database Server
DB2

Geek Corner

Some thoughts on web development



Server Side

Browser
Mozilla

Web Server
Apache

Web Application Server
Glassfish

Database Server
DB2

Geek Corner

Some thoughts on web development

- ➔ If you are going to build a web application you have to choose different technologies for each component
- ➔ In **Web 1.0**, full pages were generated on the server side and the sent to the browser which will render them to the user
- ➔ The need for a richer user experience result in the creation of **Web 2.0** which gives more power to the browser.

Geek Corner

Some thoughts on web development

➔ In **Web 2.0** more functionality is on the client side. Using Javascript, the UI is more rich and interactive.

➔ Web applications use AJAX more intensively to improve the UX

➔ Server exposes its functionality through Web APIs (fancy name for REST WebServices)

➔ **AJAX** stands for Asynchronous Javascript and XML. Allows to bring sections of pages in the background.

Geek Corner

Some thoughts on web development

- There are many frameworks/toolkits/languages to build web applications. When you are about to create a new application is recommended to use these tools

- Some of these tools are server side and others are client side. Keep that in mind.

- For example, AngularJS is client-side. Node.js is server-side.

Algorithm Design