# Lecture 12 - General problem-solving methods

- **Divide-et-impera (divide and conquer)**
- **Backtracking**

## General problem-solving methods

- strategy for solving more difficult problems
- general algorithms for solving certain types of problems
- a problem may be solved using more methods - select the most efficient one
- problem need to satisfies the required criteria of used the method
- we will apply the general algorithm

## Divide and conquer – steps

- Step1 **Divide -** divide the problem (instance) into smaller problems (of the same structure)
  - divide the problem into two or more disjoint sub problems that can be resolved using the same algorithm
- Step2 **Conquer** – resolve the sub problems recursively
- Step3 **Combine** – combine the problems results

# Divide and conquer – general

```
def divideAndConquer(data):
    if size(data)<a:
        #solve the problem directly
        #base case
        return rez
    #decompose data into d1,d2,..,dk
    rez_1 = divideAndConquer(d1)
    rez_2 = divideAndConquer(d2)
    ...
    rez_k = divideAndConquer(dk)
    #combine the results
    return combine(rez_1,rez_2,...,rez_k)
```

# We can apply divide and conquer if:

A problem *P* on the data set *D* may be solved by solving the same problem P on other data sets, $d_1$, $d_2$, ..., $d_k$, of a size smaller than the size of *D*

The **running time** for solving problems in this manner may be described using recurrences.

$$T(n) = \begin{cases} solving\ trivial\ problem, & if\ \ n\ is\ small\ enough \\ k \cdot T(n/k) + time\ \ for\ dividing + time\ \ for\ combining, & otherwise \end{cases}$$

# Division

## We can divide the data into 2 (chip and conquer): data of size 1 and data of size n-1

## Example: Find the maximum

```
def findMax(l):
    """
      find the greatest element in the list
      l list of elements
      return max
    """
    if len(l)==1:
        #base case
        return l[0]
    #divide into list of 1 elements and a list of n-1 elements
    max = findMax(l[1:])
    #combine the results
    if max>l[0]:
        return max
    return l[0]
```

## Time Complexity

**Recurrence:** $T(n)=\begin{cases} 1 \ for \ n=1 \\ T(n-1)+1 \ otherwise \end{cases}$

$$\begin{aligned} T(n) &= T(n-1)+1 \\ T(n-1) &= T(n-2)+1 \\ T(n-2) &= T(n-3)+1 \\ \ldots &= \ldots \\ T(2) &= T(1)+1 \end{aligned} \quad \Rightarrow \quad T(n)=1+1+\ldots+1=n\in\theta(n)$$

# Divide into k data of size n/k

```
def findMax(l):
    """
      find the greatest element in the list
      l list of elements
      return max
    """
    if len(l)==1:
        #base case
        return l[0]
    #divide into 2 of size n/2
    mid = len(l) /2
    max1 = findMax(l[:mid])
    max2 = findMax(l[mid:])
    #combine the results
    if max1<max2:
        return max2
    return max1
```

# Time complexity:

**Recurrence:** $T(n)=\begin{cases} 1 \; for \; n=1 \\ 2T(n/2)+1 \; otherwise \end{cases}$

**Denote:** $n=2^k$ **=>** $k=\log_2 n$ $\begin{aligned} T(2^k)&=2T(2^{(k-1)})+1 \\ 2T(2^{(k-1)})&=2^2T(2^{(k-2)})+2 \\ 2^2T(2^{(k-2)})&=2^3T(2^{(k-3)})+2^2 \\ \ldots&=\ldots \\ 2^{(k-1)}T(2)&=2^kT(1)+2^{(k-1)} \end{aligned}$ **=>**

$$T(n)=1+2^1+2^2..+2^k=(2^{(k+1)}-1)/(2-1)=2^k2-1=2n-1\in\theta(n)$$

## Divide and conquer - Example

**Compute** $x^k$ **where** $k \geq 1$ **integer number**

**Simple approach:** $x^k = k * k * \ldots * k$ **- k-1 multiplication (use a for loop)** $T(n) \in \theta(n)$

**Divide and conquer approach**

$$x^k = \begin{cases} x^{(k/2)} x^{(k/2)} \; for\, k\, even \\ x^{(k/2)} x^{(k/2)} x \; for\, k\, odd \end{cases}$$

| | |
|---|---|
| <pre>def power(x, k):<br>    """<br>      compute x^k<br>      x real number<br>      k integer number<br>      return x^k<br>    """<br>    if k==1:<br>        #base case<br>        return x<br>    #divide<br>    half = k/2<br>    aux =  power(x, half)<br>    #conquer<br>    if k%2==0:<br>        return aux*aux<br>    else:<br>        return aux*aux*x</pre> | **Divide: compute k/2**<br>**Conquer: 1 recursive call to compute** $x^{(k/2)}$<br>**Combine: 1 ore 2 multiplications**<br>**Time complexity:** $T(n) \in \theta(\log_2 n)$ |

**Divide and conquer**

- **Binary-Search (** $T(n) \in \theta(\log_2 n)$ **)**
  - **Divide – compute the middle of the list**
  - **Conquer – search on the left or for the right**
  - **Combine - nothing**
- **Quick-Sort (** $T(n) \in \theta(n \log_2 n)$ **average)**
- **Merge-Sort**
  - **Divide – divide the list into 2**
  - **Conquer – sort recursively the 2 list**
  - **Combine – merge the sorted lists**

# Backtracking

- applicable to search problems with more solutions
- generate all the solutions (if there are multiple solutions) for a given problem
- systematically searches for a solution to a problem among all available options
- is a systematic method to iterate through all the possible configurations of a search space
- a general algorithm/technique - must be customized for each individual application.
- disadvantage - it has an exponential running time

# Generate and test

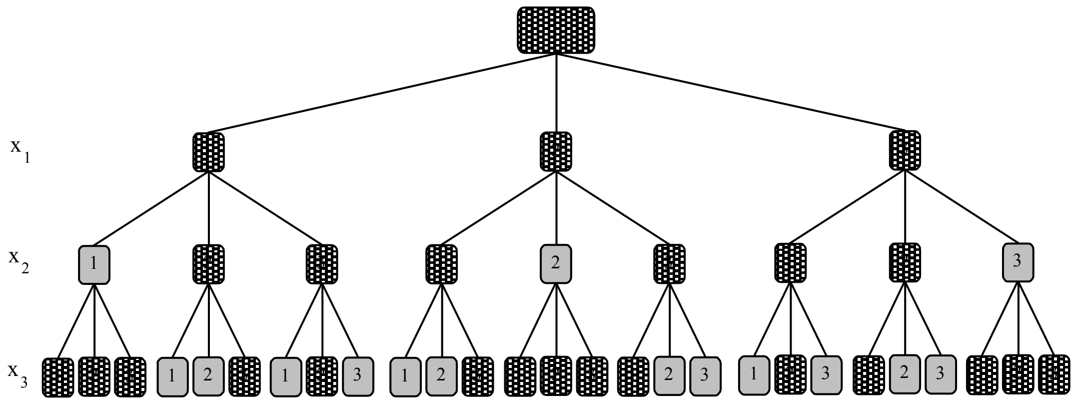
**Problem - Let *n* be a natural number. Print all permutations of numbers 1, 2, ..., *n*.**

**For n=3**

| | |
|---|---|
| <pre>def perm3():<br>    for i in range(0,3):<br>        for j in range(0,3):<br>            for k in range(0,3):<br>                #a possible solution<br>                possibleSol = [i,j,k]<br>                if i!=j and j!=k and i!=k:<br>                    #is a solution<br>                    print possibleSol</pre> | <pre>[0, 1, 2]<br>[0, 2, 1]<br>[1, 0, 2]<br>[1, 2, 0]<br>[2, 0, 1]<br>[2, 1, 0]</pre> |

- **called *Generate and Test -***
  - **Generate*:* all possible combinations of variables are first generated**
  - **Test: test in order to verify if they represents a solution.**

## Generate and test – all possible combinations

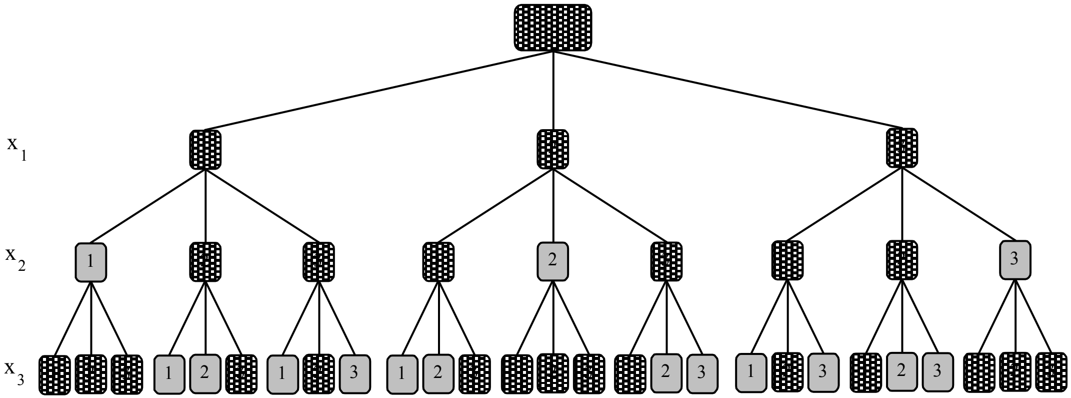



- The total number of **checked arrays is $3^3$**, and in the general case $\boldsymbol{n^n}$
- **first** assigns values to all components of the array possible, and **afterward** checks whether the array is a permutation
- It is not general. Only works for n=3

In general: if $n$ is the depth of the tree (the number of variables in a solution) and assuming that each variable has $k$ possible values, the number of nodes in the tree is $k^n$. This means that searching the entire tree leads to an exponential **time complexity**, $O(k^n)$.

# Possible improvements

- **avoiding the construction of a complete array in the case we are certain it does not lead to a correct solution.**
  - **if the first component of the array is 1, then it is useless to assign the second component the value 1**




- **work with a potential array (a partial solution)**
- **when we expand the partial solution verify some conditions (*conditions to continue)***
  - ***if the array not contains duplicates***

# Generate and test recursive

## use recursion to generate all the possible list (candidate solutions)

```
def generate(x,DIM):
    if len(x)==DIM:
        print x
    if len(x)>DIM:
        return
    x.append(0)
    for i in range(0,DIM):
        x[-1] = i
        generate(x[:],DIM)

generate([],3)
```

```
[0, 0, 0]
[0, 0, 1]
[0, 0, 2]
[0, 1, 0]
[0, 1, 1]
[0, 1, 2]
[0, 2, 0]
[0, 2, 1]
[0, 2, 2]
[1, 0, 0]
...
```

# Test candidates – print only solutions

| | |
|---|---|
| <pre>def generateAndTest(x,DIM):<br>    if len(x)==DIM and isSet(x):<br>        print x<br>    if len(x)>DIM:<br>        return<br>    x.append(0)<br>    for i in range(0,DIM):<br>        x[-1] = i<br>        generateAndTest(x[:],DIM)<br>generateAndTest([],3)</pre> | <pre>[0, 1, 2]<br>[0, 2, 1]<br>[1, 0, 2]<br>[1, 2, 0]<br>[2, 0, 1]<br>[2, 1, 0]</pre> |

- **we are still generating all the possible lists ex: lists starting with 0,0**
- **we should not explore lists that already contains duplicates (certainly not result in a valid permutation)**

## Reduce the search space – do not explore all possible candidates

## A candidate is valid (and worth further exploration) if there are no duplicates

| | |
|---|---|
| <pre>def backtracking(x,DIM):<br>    if len(x)==DIM:<br>        print x<br>    if len(x)>DIM:<br>        return  #stop recursion<br>    x.append(0)<br>    for i in range(0,DIM):<br>        x[-1] = i<br>        if isSet(x):<br>            #continue only if x can conduct to a solution<br>            backtracking(x[:],DIM)<br><br>backtracking([], 3)</pre> | <pre>[0, 1, 2]<br>[0, 2, 1]<br>[1, 0, 2]<br>[1, 2, 0]<br>[2, 0, 1]<br>[2, 1, 0]</pre> |

**is better than *Generate and Test*, but the running time complexity is still exponential.**

# Permutation problem

- **the result:** $x=(x_{0,}x_{1,..},\ldots,x_n),\ x_i\in(0,1,\ldots,n-1)$
- **is valid:** $x_i\neq x_j\ for\ any\ i\neq j$

## 8 Queens problem:

**Place 8 queens on a chess board such that no two queens are under reciprocal threat.**

- **Result: position of 8 queens on the chess board**
- **Is valid: if no queens are reciprocal threat**
  - **not on the same row, column or diagonal**
- **The total number of possible placement (both valid and invalid):**
  - **combinations of 64 taken 8, C(64, 8) $\approx 4.5\times 10^9$)**
- **Generate and test will not work in reasonable time**

**We should generate placements that can conduct to a solution (reduce the search space)**

- **if the first 2 queens are under reciprocal threat there is no reason to continue and try to place other queens on the table**
- **we need all the solutions**

# Backtracking

- the solutions **search space**: $S = S_1 \times S_2 \times \ldots \times S_n$;
- ***x*** is the array to represent the solutions;
- $x[1..k]$ in $S_1 \times S_2 \times \ldots \times S_k$ is the sub-array of **solution candidates**; it may or may not lead to a solution, i.e. it may or may not be extended to form a complete solution; the index $k$ is the number of already constructed solution elements;
- **consistent** – function to verify if a candidate can lead to a solution
- ***solution*** is a function to check whether the potential array $x[1..k]$ is a solution of the problem.

## Backtracking algorithm – recursive

```
def backRec(x):
    x.append(0) #add a new component to the candidate solution
    for i in range(0,DIM):
        x[-1] = i  #set current component
        if consistent(x):
            if solution(x):
                solutionFound(x)
            backRec(x[:]) #recursive invocation to deal with next components
```

## Even more general  (the components in the solution are not having the same domain)

```
def backRec(x):
    el = first(x)
    x.append(el)
    while el!=None:
        x[-1] = el
        if consistent(x):
            if solution(x):
                outputSolution(x)
            backRec(x[:])
        el = next(x)
```

**Backtracking**

**When we solve a problem using backtracking :**

- **need to represent the solution as a vector** $X=(x_{0,} x_{1,} \ldots x_n) \in S_0 x S_1 x \ldots x S_n$
- **define what a valid solution candidate is (conditions to filter out candidates that will not conduct to a solution)**
- **define the condition for a candidate to be an actual solution**

```
def consistent(x):
    """
      The candidate can lead to an actual
      permutation only if there are no duplicate elements
    """
    return isSet(x)



def solution(x):
    """
      The candidate x is a solution if
      we have all the elements in the permutation
    """
    return len(x)==DIM
```

**Backtracking – iterative**

```
def backIter(dim):
    x=[-1]   #candidate solution
    while len(x)>0:
        choosed = False
        while not choosed and x[-1]<dim-1:
            x[-1] = x[-1]+1  #increase the last component
            choosed = consistent(x, dim)
        if choosed:
            if solution(x, dim):
                solutionFound(x, dim)
            x.append(-1) # expand candidate solution
        else:
            x = x[:-1]  #go back one component
```