Problem solving methods

Arthur Molnar

Babes-Bolyai University

arthur@cs.ubbcluj.ro

January 12, 2016

Overview

1 Divide and conquer

2 Backtracking
- Introduction
- Generate and test
- Backtracking
- Recursive and iterative

3 Greedy

4 Dynamic programming

5 Dynamic programming vs. Greedy

Problem solving methods

- Strategy for solving more difficult problems
- General algorithms for solving certain types of problems
- A problem may be solved using more than one method - you have to select the most efficient one
- Problem need satisfies the required criteria for using the method
- We apply a general algorithm

Divide and conquer - steps

- **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
  - In many cases, there are more than one way of doing this
- **Conquer** resolve the sub problems recursively
- **Combine** combine the problems results

Remember! - Examples of Divide & Conquer

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)
```

Divide and conquer - general

When can divide&conquer be applied?

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

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

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

Step 1 - Divide

Simplest way: divide the data into 2 parts *(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]
```

Step 1 - Divide

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

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

Step 1 - Divide

- 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
```

Step 1 - Divide

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

Denote: $n=2^k \Rightarrow k=\log_2 n$

$$\begin{aligned} T(2^k) &= 2T(2^{(k-1)})+1 \\ 2T(2^{(k-1)}) &= 2^2T(2^{(k-2)})+2 \\ 2^2T(2^{(k-2)}) &= 2^3T(2^{(k-3)})+2^2 \\ \ldots &= \ldots \\ 2^{(k-1)}T(2) &= 2^kT(1)+2^{(k-1)} \end{aligned} \Rightarrow$$

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

Divide and conquer - Example

- Compute x^k, where $k \geq 1$ is an integer
- Simple approach: $x^k = x * x * ... * x$, k-1 multiplications. Time complexity?
- Divide and conquer approach

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

Divide and conquer - Example

```
def power(x, k):
    """
       compute x^k
       x real number
       k integer number
       return x^k
    """
    if k==1:
        #base case
        return x
    #divide
    half = k/2
    aux =  power(x, half)
    #conquer
    if k%2==0:
        return aux*aux
    else:
        return aux*aux*x
```

Divide and conquer - applications

- 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 ($T(n) \in \theta(n * \log_2 n)$)
  - Divide - partition the array into 2 subarrays
  - Conquer - sort the subarrays
  - Combine - nothing
- Merge-Sort ($T(n) \in \theta(n * \log_2 n)$)
  - 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
- 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

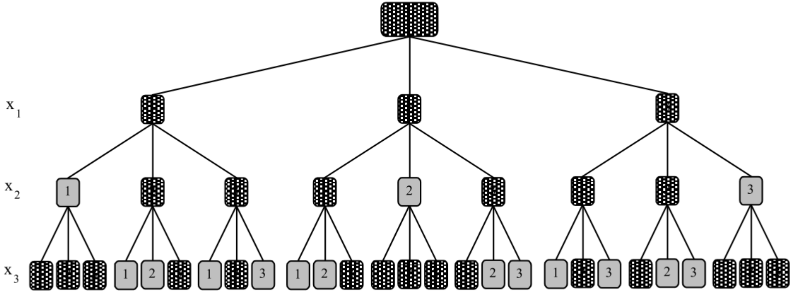
- Problem - Let n be a natural number. Print all permutations of numbers 1, 2, ..., n.
- First solution - **Generate & Test** - generate all possible solutions and verify if they represent a solution
- **This is NOT backtracking!**

```
def perm3():
    for i in range(0,3):
        for j in range(0,3):
            for k in range(0,3):
                #a possible solution
                possibleSol = [i,j,k]
                if i!=j and j!=k and i!=k:
                    #is a solution
                    print possibleSol
```

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

Generate and test

- Generate and test - all possible combinations



Generate and test

- The total number of checked arrays is 3^3 , and in the general case n^n
- First assigns values to all components of the array possible, and afterwards checks whether the array is a permutation
- Implementation above is not general. Only works for n=3

Generate and test

- 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 - $O(k^n)$

Generate and test

Possible improvements

- Avoid constructing 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 other components the value 1
- Work with a potential array (a partial solution)
- When we expand the partial solution verify some conditions (conditions to continue) - so the array does not contains duplicates

Generate and test

- Generate and test recursive - using 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)
```

Backtracking

Test candidates - print only solutions

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

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

- We are still generating all the possible lists (e.g. lists starting with 0,0)
- We should not explore lists that already contains duplicates (certainly not result in a valid permutation)

Backtracking

Test candidates - print only solutions

- Reduce the search space - do not explore all possible candidates
- A candidate is valid (and worth further exploration) if there are no duplicates
- Better than Generate and Test, but the running time complexity is still exponential.

Backtracking

- This is... backtracking!

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

backtracking([], 3)
```

Backtracking

Brief example for exponential runtimes - see **LectureXIIIBacktracking.py**

Backtracking - Typical problem statements

- Permutation problem - Generate all permutations for a given natural number n
  - result: $x = (x_0, x_1, ..., x_n), x_i \in (0, 1, ..., n-1)$
  - is valid: $x_i \neq x_j$, for any $i \neq j$
- n-Queen problem - place n queens on a chess-like board such that no two queens are under reciprocal threat.
  - result: position of the queens on the chess board
  - is valid: no queens attack each other (not on the same row or column)
  - if n=8, number of combinations is over $4.5x10^9$
  - **What is the solution?**

Backtracking - Theoretical support

- Solutions search space: $S = S_1 x S_2 x ... x S_n$
- x is the array which represents the solutions
- $x[1..k]$ in $S_1 x S_2 x ... x S_k$ is the sub-array of solution candidates; it may or may not lead to a solution
- **consistent** function to verify if a candidate can lead to a solution
- **solution** - function to check whether the potential array $x[1..k]$ is a solution of the problem.

Backtracking 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
```

Backtracking recursive

Even more general (components in the solution do not have 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
- Represent the solution as a vector $X = (x_0, x_1, ..., x_n) \in S_0 x S_1 x ... x S_n$
- Define what a valid solution candidate is (conditions filter out candidates that will not conduct to a solution)
- Define condition for a candidate to be a solution

Backtracking - iterative

General algorithm

```
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
```

Greedy

- A strategy to solve optimization problems.
- Applicable where the global optima may be found by successive selections of local optima.
- Allows solving problems without returns to the previous decisions.
- Useful in solving many practical problems that require the selection of a set of elements that satisfies certain conditions (properties) and realizes an optimum.
- Disadvantages: Short-sighted and non-recoverable

Greedy - Sample Problems

The knapsack problem

- A set of objects is given, characterized by usefulness and weight, and a knapsack able to support a total weight of W. We are required to place in the knapsack some of the objects, such that the total weight of the objects is not larger that the given value W, and the objects should be as useful as possible (the sum of the utility values is maximal).

The coins problem

- Let us consider that we have a sum M of money and coins (ex: 1, 5, 25) units (an unlimited number of coins). The problem is to establish a modality to pay the sum M using a minimum number of coins

Greedy - General case

- Let us consider the given set C of candidates to the solution of a given problem P. We are required to provide a subset B,($B \subseteq C$) to fulfil certain conditions (called internal conditions) and to maximize (minimize) a certain objective function.

Greedy - General case

- If a subset X fulfills the internal conditions we will say that the subset X is acceptable (possible).
- Some problems may have more acceptable solutions, and in such a case we are required to provide an as good a solution as we may get, possibly even the better one, i.e. the solution that realizes the maximum (minimum) of a certain objective function.

Greedy - General case

In order for a problem to be solvable using the Greedy method, it should satisfy the following property

- If B is an acceptable solution and $X \subseteq B$ then X is as well an acceptable solution.

Greedy - General case

- The Greedy algorithm finds the solution in an incremental way, by building acceptable solutions, extended continuously. At each step, the solution is extended with the best candidate from $C - B$ at that given moment. For this reason, this method is named greedy.
- The Greedy principle (strategy) is
  - Successively incorporate elements that realize the local optimum
  - No second thoughts are allowed on already made decisions with respect to the past choices.

Greedy - General case

Assuming that θ (the empty set) is an acceptable solution, we will construct set B by initializing B with the empty set and successively adding elements from C.

- The choice of an element from C, with the purpose of enriching the acceptable solution B, is realized with the purpose of achieving an optimum for that particular moment, and this, by itself, does not generally guarantee the global optimum.

Greedy - General case

- If we have discovered a selection rule to help us reach the global optimum, then we may safely use the Greedy method.
- There are situations in which the completeness requirements (obtaining the optimal solution) are abandoned in order to determine an "almost" optimal solution, but in a shorter time.

Greedy - General case

Greedy technique

- Renounces the backtracking mechanism.
- Offers a single solution (unlike backtracking, that provides all the possible solutions of a problem).
- Provides polynomial running time.

Greedy - General code


```
def greedy(c):
    """
       Greedy algorithm
       c - a list of candidates
       return a list (B) the solution found (if exists) using the greedy
strategy, None if the algorithm
       selectMostPromissing - a function that return the most promising
candidate
       acceptable - a function that returns True if a candidate solution can be
extended to a solution
       solution - verify if a given candidate is a solution
    """
    b = [] #start with an empty set as a candidate solution
    while not solution(b) and c!=[]:
        #select the local optimum (the best candidate)
        candidate = selectMostPromissing(c)
        #remove the current candidate
        c.remove(candidate)
        #if the new extended  candidate solution is acceptable
        if acceptable(b+[candidate]):
            b.append(candidate)

    if solution(b):
        return b
    #there is no solution
    return None
```

Greedy - General code (w/o specification)


```
def greedy(c):
    b = [] #start with an empty set as a candidate solution
    while not solution(b) and c!=[]:
        #select the local optimum (the best candidate)
        candidate = selectMostPromising(c)
        #remove the current candidate
        c.remove(candidate)
        #if the new extended  candidate solution is acceptable
        if acceptable(b+[candidate]):
            b.append(candidate)

    if solution(b):
        return b
    #there is no solution
    return None
```

Greedy - Essential elements

1. A **candidate set**, from which a solution is created.
2. A **selection function**, which chooses the best candidate to be added to the solution.
3. A **feasibility function**, that is used to determine if a candidate can be used to contribute to a solution.
4. An **objective function**, which assigns a value to a solution, or a partial solution.
5. A **solution function**, which will indicate when we have discovered a complete solution.

Greedy - The coins problem

- Let us consider that we have a sum M of money and coins (ex: 1, 5, 25) units (an unlimited number of coins). The problem is to establish a modality to pay the sum M using a minimum number of coins.

Greedy - Coins problem solution

Candidate Set:

The list of coins - COINS = {1, 5, 25, 50}

Candidate Solution:

A list of selected coins - $X=(X_0, X_1, .., X_k)$ **where** $X_i \in COINS$ **– used coin**

Selection Function:

candidate solution: $X=(X_0, X_1, .., X_k)$

choose the biggest coin less than the remaining sum to pay

Acceptable (*feasibility function*):

The sum payed with the current coins are not exceeding M

Candidate solution: $X=(X_0, X_1, .., X_k)$ $S=\sum_{(i=0)}^{k} X_i \leq M$

Solution function:

The sum payed with the coins in X is equal with M

Candidate solution: $X=(X_0, X_1, .., X_k)$ $S=\sum_{(i=0)}^{k} X_i = M$

Greedy - Coins problem code

```
def selectMostPromissing(c):
    """
      select the largest coin from the remaining
      c - candidate coins
      return a coin
    """
    return max(c)

def acceptable(b):
    """
    verify if a candidate solution is valid
    basically verify if we are not over the sum M
    """
    sum = _computeSum(b)
    return sum<=SUM
```

Greedy - Coins problem code

```
def solution(b):
    """
    verify if a candidate solution is an actual solution
    basically verify if the coins conduct to the sum M
    b – candidate solution
    """
    sum = _computeSum(b)
    return sum==SUM
def printSol(b):
  """
    Print the solution: NrCoinns1 * Coin1 +  NrCoinns2 *
Coin2 +...
  """
    solStr = ""
    sum = 0
    for coin in b:
        nrCoins = (SUM-sum) / coin
        solStr+=str(nrCoins)+"*"+str(coin)
        sum += nrCoins*coin
        if SUM-sum>0:solStr+=" + "
    print solStr
```

Greedy - Coins problem code

```
def _computeSum(b):
    """
   compute the payed amount with the current candidate
    return int, the payment
   b - candidate solution
    """
    sum = 0
    for coin in b:
        nrCoins = (SUM-sum) / coin
        #if this is in a candidate solution we need to
use at least 1 coin
        if nrCoins==0: nrCoins =1
        sum += nrCoins*coin
    return sum
```

Greedy - Remarks

- Before applying Greedy, is is required to prove that it provides the optimal solution. Often, the proof of applicability is non-trivial.
- Greedy leads to a polynomial running time. Usually, if the cardinality of the set C of candidates is n, Greedy algorithms have $O(n^2)$ time complexity.

Greedy - Remarks

- There are a lot of problems that can be solved using Greedy: determining the minimum spanning tree in a graph (Kruskal's algorithm), determining the shortest path between two nodes in an undirected or directed graph (Dijkstra's and Bellman-Kalaba's algorithms).
- There are problems for which Greedy algorithms do not provide optimal solution. In some cases, it is preferable to obtain a very close to the optimal solution in polynomial time, instead of the optimal solution in exponential time - heuristic algorithms.

Greedy example - Interval scheduling

- You want to schedule a number of jobs on a computer
- Jobs have the same value, and are characterized by their start and end times, namely (s_i, f_i) (the start and finish times for job "i")
- Run as many jobs as possible, making sure no two jobs overlap

Greedy example - Interval scheduling

- A Greedy implementation will directly select the next job to schedule, using some criteria
- The question to be answered - **how to determine the criteria?**

Source

Example is from "Algorithm Design", by Kleinberg & Tardos

Greedy example - Interval scheduling

Some ideas for selecting the next job

- **The job that starts earliest** - the idea being that you start using the computer as soon as possible
- **The shortest job** - the idea is to fit in as many as possible
- **The job that overlaps the smallest number of jobs remaining** - we keep our options open
- **The job that finishes earliest** - we get to use the computer as soon as possible

Greedy example - Interval scheduling

Lecture 13
Arthur Molnar
Divide and conquer
Backtracking
Introduction
Generate and test
Backtracking
Recursive and iterative
Greedy
Dynamic programming
Dynamic programming vs. Greedy

Ideas that **don't** work:

- The job that starts earliest
- The shortest job
- The job that overlaps the smallest number of jobs remaining

Greedy example - Interval scheduling

Ideas that **don't** work:

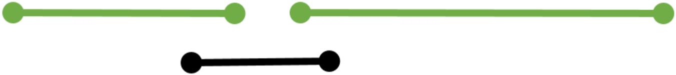
- The job that starts earliest

- The shortest job
- The job that overlaps the smallest number of jobs remaining

Greedy example - Interval scheduling

Ideas that **don't** work:

- The job that starts earliest
- The shortest job

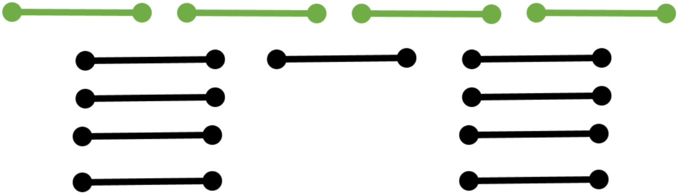


- The job that overlaps the smallest number of jobs remaining

Greedy example - Interval scheduling

Ideas that **don't** work:

- The job that starts earliest
- The shortest job
- The job that overlaps the smallest number of jobs remaining



Greedy example - Interval scheduling

An idea that **works**:

- The job that finishes earliest
- Proving this is done using mathematical induction, but it is beyond our scope.

```
S = set of jobs
while S is not empty:
    next_job = the job that has the
    soonest finishing time
    add next_job to solution
    remove from S jobs that overlap q
```

Further learning resources

Further resources

http://www.cs.princeton.edu/ wayne/kleinberg-tardos/

Dynamic programming

Applicable in solving optimality problems where

- The solution is the result of a sequence of decisions, $d_1, d_2, ..., d_n$,
- The principle of optimality holds.
- Usually leads to a polynomial running time
- Always provides the optimal solution (unlike Greedy).
- Like the data division method solves problems by combining the sub solutions of their sub problems, but, unlike it, calculates only once a sub solution, by storing the intermediate results.

Dynamic programming

- We consider the states $s_0, s_1, ..., s_n - 1, s_n$, where s_0 is the initial state, and s_n is the final state, obtained by successively applying the sequence of decisions $d_1, d_2, ..., d_n$ (using the decision d_i we pass from state $s_i - 1$ to state s_i , for i=1,n):

$$s_0 \xrightarrow{d_1} s_1 \xrightarrow{d_2} s_2 \xrightarrow{} \overset{...}{...} \xrightarrow{} s_{n\text{-}1} \xrightarrow{d_n} s_n$$

Dynamic programming

Dynamic programming method makes use of three main issues

- The principle of optimality
- Overlapping sub problems
- Memoization.

Dynamic programming - The principle of optimality

- The general optimum implies partials optima
- In an optimal sequence of decisions, each decision is optimal.
- Is not always satisfied, especially in cases when sub-sequences are not independent and optimization of one of them is in conflict with the optimization of other.

Dynamic programming - The principle of optimality

If $d_1, d_2, ..., d_n$ is a sequence of decisions that optimally leads a system from the state s_0 to s_n, then one of the following conditions has to be satisfied:

- $d_k, d_k + 1, ..., d_n$ is a sequence of decisions that optimally leads the system from the state $s_k - 1$ to the state s_n, $\forall k, 1 \leq k \leq n$ **(forward variant)**
- $d_1, d_2, ..., d_k$ is a sequence of decisions that optimally leads the system from the state s_0 to the state s_k, $\forall k, 1 \leq k \leq n$ **(backward variant)**
- $d_k + 1, d_k + 2, ..., d_n$ and $d_1, d_2, ..., d_k$ are sequences of decisions that optimally lead the system from state s_k to the state s_n and, respectively, from state s_0 to the state s_k, $\forall k, 1 \leq k \leq n$ **(mixed variant)**

Dynamic programming - Overlapping sub-problems and memorization

- **Overlapping sub-problems** - A problem is said to have overlapping sub-problems if it can be broken down into sub-problems which are reused multiple times
- **Memorization** - Store the solutions to the sub-problems for later reuse

Dynamic programming - Requirements

- The principle of optimality (in one of its form: forward, backward or mixed) is proved.
- The structure of the optimal solution is defined.
- Based on the principle of optimality, the value of the optimal solution is recursively defined. This means that recurrent relations, indicating the way to obtain the general optimum from partial optima, are defined.
- The value of the optimal solution is computed in a bottom-up manner, starting from the smallest cases for which the value of the solution is known.

Dynamic programming - Longest increasing sublist

- Problem statement - Let us consider the list $a_1, a_2, ..., a_n$. Determine the longest increasing sublist $a_{i_1}, a_{i_2}, ..., a_{i_s}$ of list a.

Dynamic programming - Longest increasing sublist

- Principle of optimality - verified in its forward variant.
- Structure of the optimal solution - we will construct a sequences: $L =< L_1, L_2, ..., L_n >$ so that for each $1 \leq i \leq n$ we have that L_i is the length of the longest increasing sublist ending at i.
- The recursive definition for the value of the optimal solution:
  - $L_i = max\{1 + L_j | A_j$, so that $A_j \leq A_i\}, \forall j = i-1, n-2, ..., 1$

Dynamic programming - Code for longest increasing sublist (from Seminar 14)


```python
for i in range(1, len(A)):
    '''
    The maximum length of the increasing subsequence ending at index i
    '''
    indices_array.append(1)
    previous_indices.append(-1)

    '''
    The maximum length is increased by 1 if 'j' exists, so that:
        A[j] < A[i] and the length of the subsequence would increase
    '''
    for j in range(i - 1, -1, -1):
        if (indices_array[j] + 1 > indices_array[i]) and (A[j] < A[i]):
            indices_array[i] = indices_array[j] + 1
            previous_indices[i] = j
```

Dynamic programming - Egg dropping puzzle

Suppose you have a number of **n** eggs and a building having **k** floors. Using a minimum amount of drops, determine the lowest floor from which dropping an egg breaks it.

Dynamic programming - Egg dropping puzzle

Rules:

- An egg that survives the fall is unharmed and reusable.
- A broken egg cannot be reused.
- All eggs are equivalent.
- If an egg breaks when dropped from a given floor, it will also break when dropped from a higher floor.
- If an egg does not break when released from a given floor, it can be safely dropped from a lower floor.
- You cannot assume that dropping eggs from the first floor is safe, nor that dropping from the last floor is not.

Dynamic programming - Egg dropping puzzle

The problem is actually about finding the correct strategy to improve the worst case outcome - make sure that the **maximum** number of drops is kept to a minimum (a.k.a minimization of maximum regret).

Dynamic programming - Egg dropping puzzle

So, what do we do in the simplest case?

- Building has **k** floors, and we have **n=1** egg.

Dynamic programming - Egg dropping puzzle

So, what do we do in the **n=1** case?

- Drop the egg at each floor until it breaks or you've reached the top, starting from first (ground) floor.

Dynamic programming - Egg dropping puzzle

So, how about if we have more eggs?

- Building has **k** floors, and we have **n=2** eggs.
- **How do we improve the maximum number of throws scenario, a.k.a the worst case?**

Dynamic programming - Egg dropping puzzle

So, what do we do in the **n=2** case?

1. The **n=1** case was basically linear search, so we can try binary.
2. How about dropping from every 20th floor, starting from ground level?

Which of the strategies above is better? Which one is optimal, if any?

Dynamic programming - Egg dropping puzzle

So, what do we do in the **n=2** case?

- Imagine we drop the first egg at a floor **m**.
- If it breaks, we have a maximum of **(m-1)** drops, starting from ground.
- If it did not break, we increase by **(m-1)** floors, as we have one less drop.
- Following the same logic, at each step we decrease the number of floors by 1.

Dynamic programming - Egg dropping puzzle

So, if we have **n=2** eggs and **k=100** floors?

- We solve $n + (n-1) + (n-2) + (n-3) + (n-4) + \ + 1 \geq 100$
- Solution is between 13 and 14, which we round to **14**. First drop is from floor 14

Dynamic programming - Egg dropping puzzle

Egg drops for **n=2** eggs and **k=100** floors...

Drop	#1	#2	#3	#4	#5	#6	#7	#8	#9	#10	#11	#12
Floor	14	27	39	50	60	69	77	84	90	95	99	100

Dynamic programming - Egg dropping puzzle

How about the general case - Building has **k** floors, and we have **n** eggs.

- **Optimal substructure** - Dropping an egg from floor **x** might result in two cases (subproblems):
  1. **Egg breaks** - Problem is reduced to one with **x-1** floors and **n-1** eggs.
  2. **Egg is ok** - Problem is reduced to one with **k-x** floors and **n** eggs.

Dynamic programming - Egg dropping puzzle

- **Overlapping subproblems** - let's create function eggDrop(n, k), where **n** is the number of eggs and **k** the number of floors.
- **eggDrop(n, k)** $=1+$ min$\{$max(eggDrop(n - 1, x - 1), eggDrop(n, k - x)), with $1 \leq x \leq k\}$

Dynamic programming - Egg dropping puzzle

Full solution available in the **LectureXIVDynamicProgramming.py** file.

Dynamic programming vs. Greedy - Remarks

- Both techniques are applied in optimization problems
- Greedy is applicable to problems for which the general optimum is obtained from partial (local) optima
- Dynamic programming is applicable to problems in which the general optimum implies partial optima.

Dynamic programming vs. Greedy - An example

- Problem statement - Let us consider the problem to determine the optimal path between two vertices i and j of a graph.
- What we can notice:
  - The principle of optimality is verified: if the path from i to j is optimal and it passes through node x, then the paths from i to x, and from x to j are also optimal.
  - Can we apply Greedy? - if the paths from i to x, and from x to j are optimal, there is no guarantee that the path from i to j that passes through x is also optimal

Dynamic programming vs. Greedy - Remarks

- The fact that the general optimum implies the partial optimum, does not mean that partial optima also implies the general optimum (the example before is relevant).
- The fact that the general optimum implies partial optima is very useful, because we will search for the general optimum among the partial optima, which are stored at each moment. Anyway, the search is considerably reduced.