BABEŞ-BOLYAI UNIVERSITY
Faculty of Mathematics and Computer Science

Algorithms and Programming

Lecture 11 – Problem solving methods (II)

Camelia Chira

Course content

Programming in the large

- Introduction in the software development process
- Procedural programming
- Modular programming
- Abstract data types
- Software development principles
- Testing and debugging

Programming in the small

- Recursion
- Complexity of algorithms
- Search and sorting algorithms
- **Problem solving methods**
  - Generate and test, Backtracking, Divide et impera
  - **Dynamic programming, Greedy**
- Recap

Problem solving by search

- Solving problems by search using standard methods
  - Exact methods
    - **Generate and test**
    - **Backtracking**
    - **Divide and conquer**
    - **Dynamic programming**
  - Heuristic methods
    - **Greedy method**

Dynamic Programming (DP)

- Basic idea:
  - Break the problem in overlapping sub-problems which are similar to the initial problem but are smaller in size
  - Solve the sub-problems
  - Compute the final solution by combining the sub-solutions
- Applicable in solving problems where:
  - Problems where one needs to find the best decisions one after another
  - The solution is the result of a sequence of decisions dec_1; dec_2; ...; dec_n.
  - The principle of optimality holds (whatever the initial state is remaining decisions must be optimal with regard the state following from the first decision)

DP: Mechanism

- Break the problem in nested sub-problems P(P1(P2(P3(...(Pn))...)

- Solve the most inner sub-problem P_n and store the partial result
- Solve the sub-problem P_{n-1} based on the solution found for sub-problem P_n and store the partial result
- Solve the sub-problem P_{n-2} based on the solution found for sub-problem P_{n-1} and store the partial result
- ....
- Solve the sub-problem P_1 based on the solution found for sub-problem P_2 and store the partial result
- Solve the problem P based on the solution found for sub-problem P_1 and store the **final** result

Dynamic Programming

- When DP can be used?
  - Problem P (optimization problem) with input data D can be solved by solving the same problem P but with input data d, where d < D
  - Solution is the result of a sequence of decisions dec1, dec2, ...
  - The problem can be divided in overlapping problems
  - The solutions of the sub-problems can be stored for future uses
  - The principle of optimality

- Features
  - Always gives the optimal solution
  - Polynomial run time

Dynamic Programming

- Notations
- We consider states $s_0, s_1, \ldots s_n$
  - s_0 is the initial state
  - s_n is the final state
  - States are obtained by successively applying the sequence of decisions dec_1, $dec_2, \ldots,$ dec_n (using the decision d_i we pass from state s_{i-1} to state s_i)

$$s_0 \xrightarrow{dec_1} s_1 \xrightarrow{dec_2} s_2 \longrightarrow \cdots \longrightarrow s_{n-1} \xrightarrow{dec_n} s_n$$

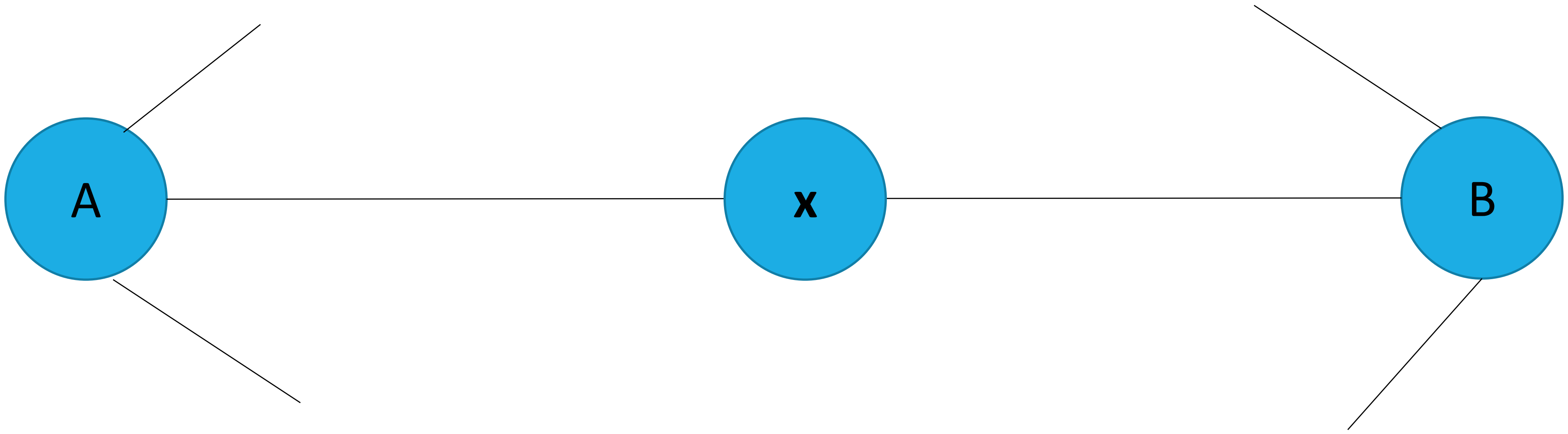
DP: Principle of optimality

- Principle of optimality
  - The general optimum implies the local / partial optimum
  - In an optimal sequence of decisions, each decision is optimum
  - The principle does not hold true for any problem

- Formally, a sequence of decisions $dec_1, dec_2, \ldots, dec_n$ optimally leads from state s_0 to state s_n if at least one of the following conditions is satisfied:
  - $dec_k, dec_{k+1}, \ldots, dec_n$ is a sequence of decisions that optimally leads from state s_{k-1} to state s_n for any k, $1 \leq k \leq n$ (**forward** method)
  - $dec_1, dec_2, \ldots, dec_k$ is a sequence of decisions that optimally leads from state s_0 to state s_k for any k, $1 \leq k \leq n$ (**backward** method)
  - $dec_{k+1}, \ldots, dec_n$ and $dec_1, dec_2, \ldots, dec_k$ are two sequences of decisions that optimally lead from state s_{k-1} to state s_n and from state s_0 to state s_k for any k, $1 \leq k \leq n$ (**mixed** method)

Principle of optimality

- In solving a problem, we have to make a sequence of n decisions
- If this sequence is optimal then the last k decisions (1<k<n) must be optimal



DP: Algorithm

- Verify the principle of optimality
- Establish the structure of the solution
  - Break the problem in sub-problems
  - Overlapping sub-problems – break down the problem into sub-problems which are reused multiple times
- Memoization
  - Store the solutions to the sub-problems for later use
- Based on the principle of optimality, the value of the optimal solution is recursively 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

DP: Example

- **Problem**: find the longest increasing subsequence from a list of integer numbers.

i	**1**	**2**	**3**	**4**	**5**
list	2	1	9	6	12

- Solution
  - For each i, calculate the length of the longest increasing subsequence that can be formed
  - In the end, select the element where the longest subsequence is formed

i	**1**	**2**	**3**	**4**	**5**
list	2	1	9	6	12
L	**3** 2,9,12 sau 2,6,12	**3** 1,9,12 sau 1,6,12	**2** 9,12	**2** 6,12	**1** 12

DP: Example

- **Problem**: find the longest increasing subsequence from a list of integer numbers.

i	**1**	**2**	**3**	**4**	**5**
list	2	1	9	6	12

- Step 1: The principle of optimality
  - The principle of optimality is verified in its forward variant
  - The longest subsequence that starts at position i has k elements => the subsequences that can be formed from it (with k-1, k-2,...elements) are increasing subsequences and have maximal length

DP: Example

- **Problem**: find the longest increasing subsequence from a list of integer numbers.

i	1	2	3	4	5
list	2	1	9	6	12

- Step 2: The structure of the optimal solution
  - Break the problem in sub-problems
  - Problem: determine the longest increasing subsequence
  - Sub-problem: determine the longest increasing subsequences that starts with list[i] for i =n, n-1, n-2,..., 1. These subsequences have the length at most 1,2,...,n.
  - Solution: the longest subsequence from the n subsequences.

DP: Example

i	1	2	3	4	5
list	2	1	9	6	12

- **Problem**: longest increasing subsequence

- Step 3: Determine the global optimum based on the partial optimas
  - Let L_i be the length of the longest subsequence that starts with list[i]
  - The increasing sub-sequences that start with list[i] are obtained by adding the element list[i] in front of an increasing subsequence that starts with list[j] if list[i] ≤ list[j]

$$L_i = 1 + \max_{j=i+1,n} \{L_j, list[i] \leq list[j]\}, for\ i = n-1, \ldots, 1$$

  - $L_n = 1$

  - Optimal solution: $L_{max} = \max_{i=i,n}\{L_i\}$

```python
def long_seq(s):
    L = [0] * len(s)
    ind = [0] * len(s) #index of the succesor of list[i] in the long seq
    #compute vector L
    L[len(s) - 1] = 1
    ind[len(s) - 1] = -1
    for i in range(len(s) - 2, -1, -1):
        ind[i] = -1
        L[i] = 1
        for j in range(i+1, len(s)):
            if (s[i] <= s[j]):
                if (L[i] <= L[j] + 1):
                    L[i] = L[j] + 1
                    ind[i] = j

    #determine position max elem from L
    max_pos = 0
    for i in range(1, len(s)):
        if (L[i] > L[max_pos]):
            max_pos = i

    #construct the solution
    sol = []
    i = max_pos
    while (i != -1):
        sol.append(s[i])
        i = ind[i]
    return sol
```

```python
def test_long_seq():
    assert long_seq([2,1,9,6,12]) == [2, 6, 12]
    assert long_seq([0,-2,3,1,0,-1,2,5,-5,5,-8.10,7,-3,1]) == [0,0,2,5,5,7]
```

Greedy method

- Basic idea
  - Break the problem in successive sub-problems similar to the initial problem but of smaller dimensions
  - Solve the sub-problems and determine the final solution by successively selecting the best sub-solutions
  - Global optimum = a sequence of local optimas

- Mechanism
  - Divide the problem in successive sub-problems P1, P2, ...Pn
  - Progress to the final solution by selecting at each step the best decision

Greedy method

- When to use Greedy?
  - Problem P (optimization)
  - Solution is the result of a successive selections of local optima
  - Problems with solution represented by subsets or chartesian products that achieve a certain optimum (minum or maximum) of an objective function
- Features
  - Can reach the optimal solution
  - Builds the solution step by step
  - Offers a single solution (unlike backtracking)
  - Polynomial run time
- Disadvantages: Short-sighted and non-recoverable

Greedy Algorithm

- Let *S* be a solution to the problem and *C* the set of local optima for each sub-problem (candidate elements of the solution)

```
def greedy(C):
    S = Φ
    while (not isSolution(S)) and (C≠Φ):
        el = selectMostPromissing(C)
        C.remove(el)
        if acceptable(el, S):
            S.append(el)
    if isSolution(S):
        return S
    else:
        return None
```

Greedy - Example of Problems

- Coins Problem
  - Consider a sum of money and a set of coins units. The problem is to establish a modality to pay the sum of money using a minimum number of coins.

- Knapsack Problem
  - Consider a set of objects, each having a value and weight, and a knapsack able to support a total weight of W. Place in the knapsack some of the objects, such that the total weight of the objects is not larger than W and the objects have max value.

- General Problem
  - Let us consider the given set C of candidates to the solution of a given problem P.
  - The objective is to provide a subset B to fulll certain conditions (called internal conditions) and to maximize (minimize) a certain objective function.

Greedy strategy

- Greedy algorithm finds the solution in an incremental way
- Greedy strategy
  - Successively incorporate elements that realize the local optimum
  - No second thoughts are allowed on already made decisions
- Generally, the required elements of a greedy strategy are:
  - A **candidate set** (from which a solution is created)
  - A **selection function** (selects the best candidate to be added to the solution)
  - A **feasibility function** (determines if a candidate can be used in a solution)
  - An **objective function** (assigns a value to a solution, or a partial solution)
  - A **solution function** (checks if a complete solution has been found)

Greedy: Coins Problem

- Problem: Find a way to pay a sum of money using a minimum number of coins (different values of coins are available).

  - Data: Sum = 80, Coins = [1, 5, 10, 25, 50]
  - Results: 80 = 50 + 25 + 5

  - Data: Sum = 10, Coins = [1, 2, 3, 4]
  - Results : 10 = 4 + 3 + 2 + 1

  - Data : Sum = 10, Coins = [2, 3, 4, 5]
  - Results : 10 = 5 + 3 + 2

Greedy: Example

- Solution
  - C – list of available coins
  - *isSolution(sol)*
    - If the sum of coins selected in sol is equal to the desired sum
  - *selectMostPromissing(C)*
    - Select the highest value coin in C
  - *acceptable(el,sol)*
    - If the sum of coins in sol + el is not over the desired sum

```python
def sum(l):
    s = 0
    for el in l:
        s = s + el
    return s

def isSolution(solution, limit):
    return sum(solution) == limit

def selectMostPromissing(candidates):
    return max(candidates)

def acceptable(element, solution, limit):
    return sum(solution) + element <= limit

def greedy_coins(coins, sumOfMoney):
    sol = []
    while (not isSolution(sol, sumOfMoney)) and (coins != []):
        el = selectMostPromissing(coins)
        coins.remove(el)
        if acceptable(el, sol, sumOfMoney):
            sol.append(el)
    if isSolution(sol, sumOfMoney):
        return sol
    else:
        return None
```

```python
def test_greedy_coins():
    assert greedy_coins([1, 5, 10, 25, 50], 80) == [50, 25, 5]
    assert greedy_coins([1, 2, 3, 4], 10) == [4, 3, 2, 1]
    #assert greedy_coins([1, 2, 3, 4, 5], 10) == [5, 3, 2]
    assert greedy_coins([2, 3, 4, 5], 10) == None

test_greedy_coins()
```

Dynamic Programming vs Greedy

- Both techniques are applied in optimization problems

- DP is applicable to problems in which the general optimum implies partial optima
- Greedy is applicable to problems for which the general optimum is obtained from partial (local) optima

- DP always provides the optimal solution
- Greedy does not guarantee finding the optimal solution

Reading materials and useful links

1. The Python Programming Language - https://www.python.org/
2. The Python Standard Library - https://docs.python.org/3/library/index.html
3. The Python Tutorial - https://docs.python.org/3/tutorial/
4. M. Frentiu, H.F. Pop, Fundamentals of Programming, Cluj University Press, 2006.
5. MIT OpenCourseWare, Introduction to Computer Science and Programming in Python, https://ocw.mit.edu, 2016.
6. K. Beck, Test Driven Development: By Example. Addison-Wesley Longman, 2002. http://en.wikipedia.org/wiki/Test-driven_development
7. M. Fowler, Refactoring. Improving the Design of Existing Code, Addison-Wesley, 1999. http://refactoring.com/catalog/index.html