Problem solving methods

Lect. PhD. Arthur Molnar

Babes-Bolyai University

Overview

Problem solving methods

- Strategies for solving more difficult problems, or 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
- In order to apply one of the methods described here, the problem needs to satisfy certain criteria

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

Typical problems for Divide & Conquer

Divide and conquer - general


```
def divide_conquer(data):
    if size(data) < a:
        # solve the problem directly
        # base case
        return rez
    # decompose data into d1, d2, ..., dk
    rez_1 = divide_conquer(d1)
    rez_1 = divide_conquer(d1)
    ...
    rez_k = divide_conquer(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, $d_1, d_2, ..., d_k$, of a size smaller than the size of D.

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

$$\mathrm{T(n)} = \begin{cases} \text{solve trivial problem, } n \text{ small enough} \\ k * T(\frac{n}{k}) + \text{divide time} + \text{combine time, otherwise} \end{cases}$$

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 find_max(data):
    '''
    Find the greatest element in the list
    input: data - list of elements
    output: maximum value
    '''
    if len(data) == 1:
        return data[0]
    # Divide into subproblems of sizes 1 and (n-1)
    max_val = find_max(data[1:])
    # Combine
    if max_val > data[0]:
        return max_val
    return data[0]
```

Step 1 - Divide

- Calculating time complexity
- The recurrence is: $T(n) = \begin{cases} 1, n = 1 \\ T(n-1) + 1, \text{otherwise} \end{cases}$

T(n)=T(n-1)+1,
T(n-1)=T(n-2)+1,
T(n-2)=T(n-3)+1, so
$T(n) = 1 + 1 + 1 + ... + 1 = n, T(n) \in \theta(n)$

Step 1 - Divide

- Divide into k subproblems of size n/k

```
def find_max(data):
    '''
    Find the greatest element in the list
    input: data - list of elements
    output: maximum value
    '''
    if len(data) == 1:
        return data[0]
    # Divide into two subproblems of size n/2
    mid = len(data) // 2
    max_left = find_max(data[:mid])
    max_right = find_max(data[mid:])
    # Combine
    return max(max_left, max_right)
```

Step 1 - Divide

- The recurrence is: $T(n) = \begin{cases} 1, n = 1 \\ 2 * T(\frac{n}{2}) + 1, \text{otherwise} \end{cases}$

$T(2^k) = 2 * T(2^{k-1}) + 1$
$2 * T(2^{k-1}) = 2^2 * T(2^{k-2}) + 2$
$2^2 * T(2^{k-2}) = 2^3 * T(2^{k-3}) + 2^2$
As we can divide the data into halves a number of **k** times, $n = 2^k$, then $k = log_2 n$ and $T(n) =$
$1 + 2^1 + 2^2 + 2^3 + ... + 2^k = \frac{2^{k+1}-1}{2-1} = 2^{k+1} - 1 = 2n \in \Theta(n)$

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

$$x^k = \begin{cases} x^{\frac{k}{2}} * x^{\frac{k}{2}}, k \text{ is even} \\ x^{\frac{k}{2}} * x^{\frac{k}{2}} * x, k \text{ is odd} \end{cases}$$

Divide and conquer - Example


```
def power(x, k):
    '''
    Calculate x to the power of k
    '''
    if k == 0:
        return 1
    if k == 1:
        return x
    # Divide
    aux = power(x, k // 2)
    # Conquer
    if k % 2 == 0:
        return aux ** 2
    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
- Generates all the solutions, if they exist
- Does a depth-first search through all possibilities that can lead to a solution
- A general algorithm/technique - must be customized for each individual application.

Remember

Backtracking usually has exponential complexity

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

NB!

This is **not** backtracking

Generate and test - iterative


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

NB!

This is **not** backtracking

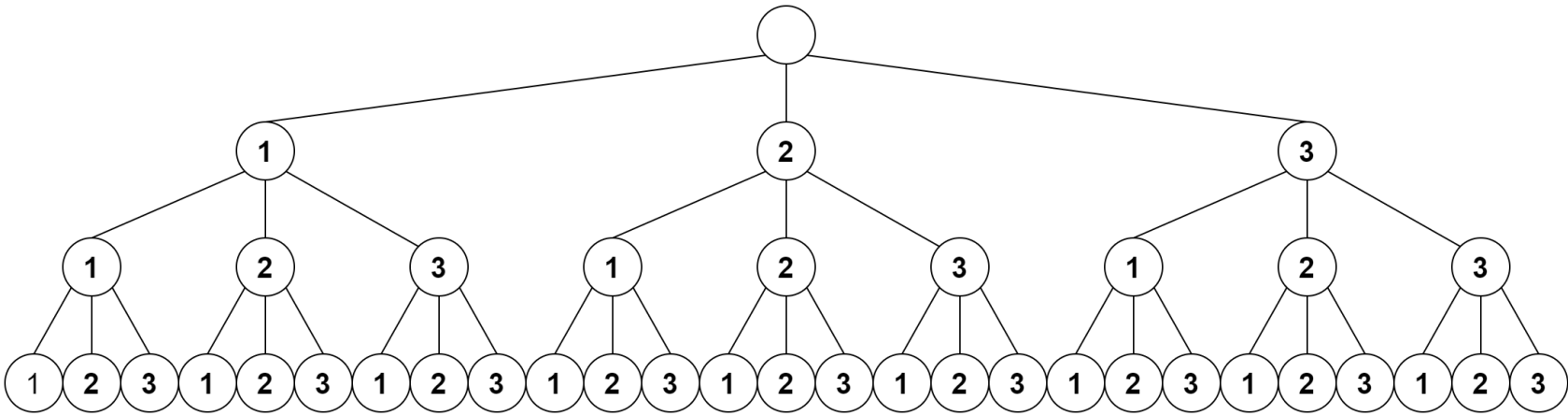
Generate and test - recursive

- 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)
print(generate([], 3))
```

Generate and test

- Generate and test - all possible combinations



Generate and test

What have we learned?

- The total number of checked arrays is 3^3 , and in the general case n^n
- First the algorithm assigns values to all components of the array possible, and only 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

- Do not construct a complete array in case it cannot lead to a correct solution.
- e.g - 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

Test candidates - print only solutions

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

- We still generate all possible lists (e.g. starting with 0,0,...)
- We should not explore lists that contains duplicates (they cannot result in valid permutations)

Backtracking

- Recursive backtracking implementation for determining permutations

```
def backtracking(x, DIM):
    if len(x) == DIM:
        print(x)
    x.append(0)
    for i in range(0, DIM):
        x[-1] = i
        if is_set(x):
            # Continue only if we
            # can reach a solution
            backtracking(x[:], DIM)
print(backtracking([], 3))
```

Demo

Backtracking

Example illustrating exponential runtimes are in **ex34_backtracking.py**

Backtracking - Typical problem statements

- **Permutations** - 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 rank, file or diagonal)

Backtracking - Theoretical support

Backtracking

The key to many problems that can be solved with backtracking lays in correctly and efficiently representing the elements of the search space

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 array $x_{[1..k]}$ represents a solution of the problem.

Backtracking - recursive


```
def backtracking(x):
    # Add component to candidate
    x.append(0)
    for i in range(0, DIM):
        # Set current component
        x[-1] = i
        if consistent(x):
            if solution(x):
                solution_found(x)
            # Deal with next components
            backtracking(x[:])
```

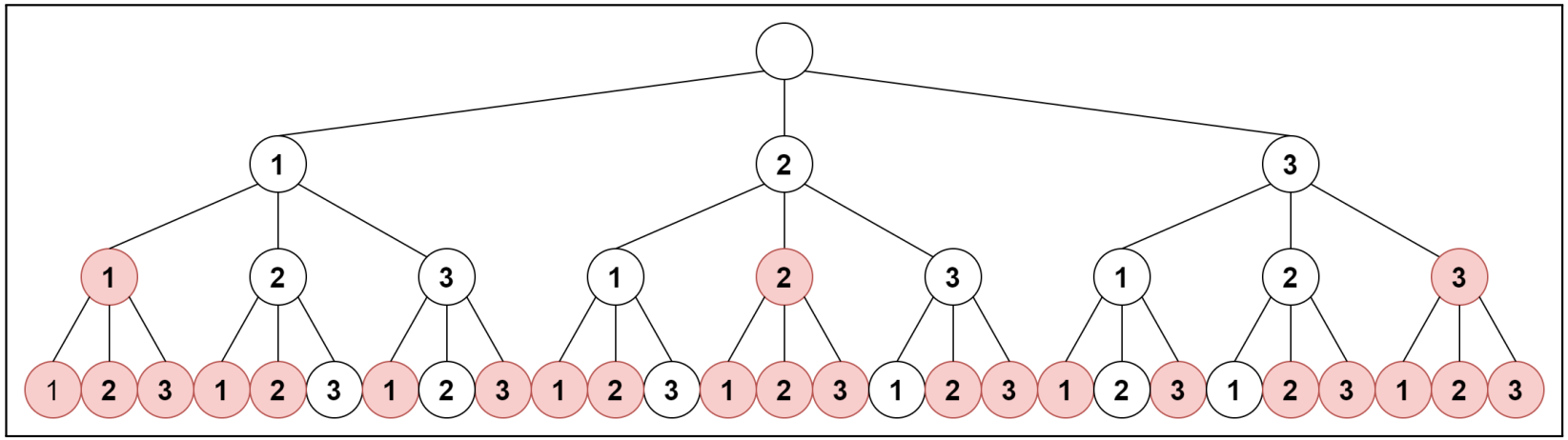
Backtracking - recursive

More generally, when solution components do not have the same domain:

```
def backtracking(x):
    # Add component to candidate
    el = first(x)
    x.append(el)
    while el != None:
        # Set current component
        x[-1] = el
        if consistent(x):
            if solution(x):
                solutionFound(x)
            # Deal with next components
            backtracking(x[:])
        el = next(x)
```

Backtracking

The nodes explored with backtracking for the permutations of 3



Backtracking

- How to use 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

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 returning to 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 (fractional) 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 fulfill 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 as good a solution as we may get, possibly even the best 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 an acceptable solution as well.

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 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):
    # The empty set is the candidate solution
    b = []
    while not solution(b) and c != []:
        # Select best candidate (local optimum)
        candidate = select_most_promising(c)
        c.remove(candidate)
        # If the candidate is acceptable, add it
        if acceptable(b + [candidate]):
            b.append(candidate)
        if solution(b):
            solution_found(b)
        # In case no solution
        return None
```

Greedy - General code (explanation)

- **c** - list of candidates
- **select_most_promising()** - returns the most promising candidate
- **acceptable()** - decides if the candidate solution can be extended
- **solution()** - verifies if a candidate represents a solution

Greedy - Essential elements

More generally, the required elements are:

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 coin denominations (e.g. 1, 5, 10 bani).
- **Candidate solution** - a list of selected coins
- **Selection function** - choose the coin with the biggest value less than the remainder sum
- **Acceptable** - the total sum paid does not exceed the required sum
- **Solution function** - the paid sum is exactly the required sum

Demo

Greedy

The source code for the coins problem can be found in **ex35_coins.py**

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 polynomial run 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, such as 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 close to optimal solution in polynomial time, instead of the optimal solution in exponential time - these are known as *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 finish 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 crucial question for solving the problem - **how to determine the correct criteria?**

Source

This example, like many others can be found in "Algorithm Design", by Kleinberg & Tardos[1]

[1] https://www.cs.princeton.edu/~wayne/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 jobs as possible
- **The job that overlaps the smallest number of jobs remaining** - we keep our options open
- **The job that finishes earliest** - we free up the computer as soon as possible

Greedy example - Interval scheduling

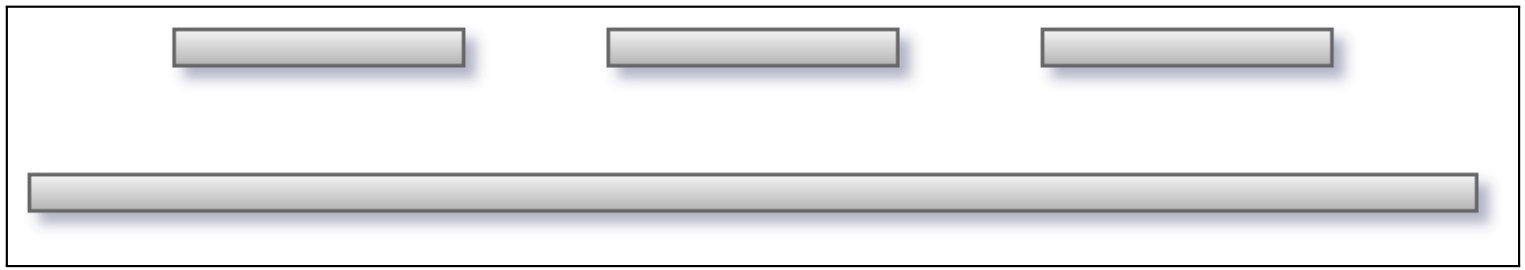
Ideas that **don't** work:

- 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

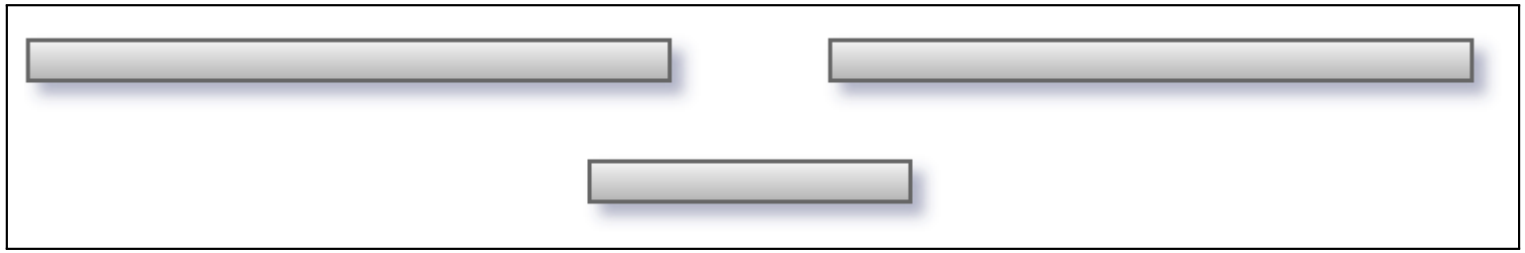


- The shortest job
- The job that overlaps the smallest number of jobs remaining

Greedy example - Interval scheduling

Ideas that **don't** work:

- The job that starts earliest
- The shortest job

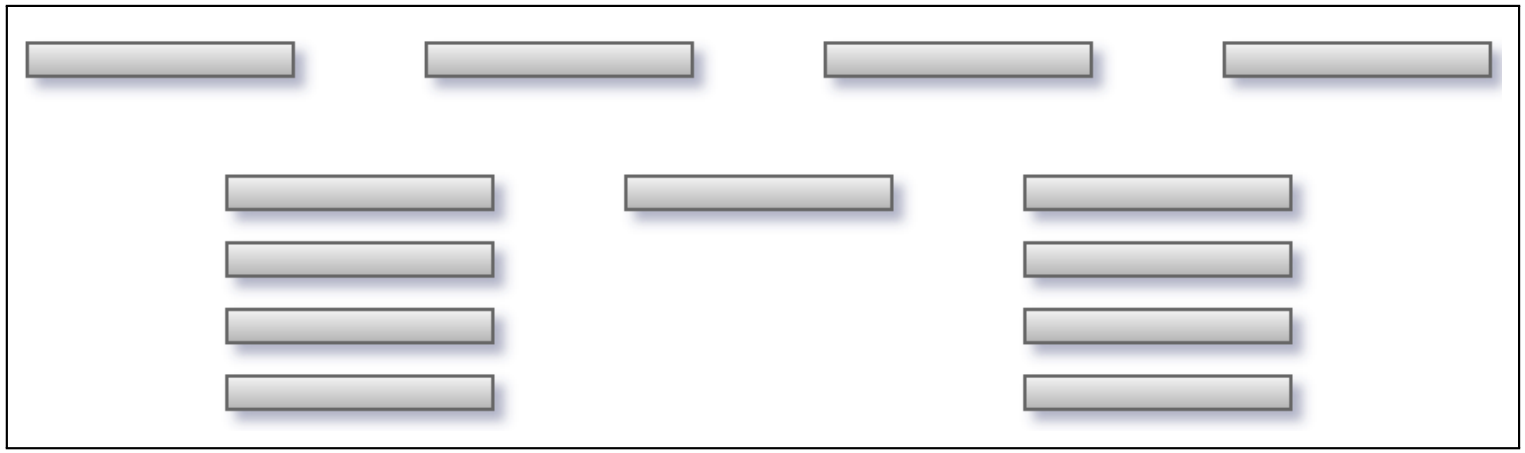


- The job that overlaps the smallest number of jobs remaining

Greedy example - Interval scheduling

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** - Start the job that finishes earliest

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**

NB!

Proving this is done using mathematical induction, but it is beyond our scope.

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.

Dynamic programming:

- Usually leads to polynomial run time.
- Always provides the optimal solution (unlike Greedy).
- Like divide & conquer, it combines the solutions from sub-problems, but also stores intermediate results that are reused multiple times.
- Visualize it as an optimization to applying recursion.

Dynamic programming

- We consider 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):

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

Dynamic programming

The method makes use of three main concepts:

- Optimality principle
- Overlapping sub problems
- Memoization.

Dynamic programming - Optimality principle

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 state s_{k-1} to 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 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 state s_n and, respectively, from state s_0 to state s_k, $\forall k, 1 \leq k \leq n$ **(mixed variant)**

Dynamic programming - Overlapping sub-problems, memoization

- **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
- **Memoization** - Store the solutions to the sub-problems for later use

Dynamic programming - Requirements

- The principle of optimality (in one of the forward, backward or mixed forms) is proven.
- 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 subsequence

Problem statement

Let us consider the list $a_1, a_2, ..., a_n$. Determine the longest increasing subsequence $a_{i_1}, a_{i_2}, ..., a_{i_s}$ of list a.
e.g. given sequence [0, 8, 4, 12, 2, 10, 6, 14, 1, 9, 5, 13, 3, 11, 7, 15], a longest increasing subsequence is [0, 2, 6, 9, 11, 15].

Dynamic programming - Longest increasing subsequence

- **Optimality principle** - verified in its forward variant.
- **Structure** of the optimal solution - we construct 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 subsequence ending at i.
- The recursive definition for the value of the optimal solution:
  - $L_i = max\{1 + L_j | A_j$, so that $A_j \leq A_i\}, \forall j = i - 1, n - 2, ..., 1$

Demo

Longest Increasing Subsequence

Examine the source code in **ex36_longestIncreasingSubsequence.py**

Maximum subarray sum

Problem statement

Calculate the maximum sum of a subarray consisting of consecutive elements. e.g. for subarray **[-2, -5, 6, -2, -3, 1, 5, -6]** the maximum sum is 7 (6-2-3+1+5, as the numbers must be consecutive)

Maximum subarray sum

This is a great problem, because there are several implementations.

1. Naive implementation(s)
2. Divide & conquer
3. Dynamic programming

Maximum subarray sum

Naive implementation(s)

1. Uses 3 loops; one for interval start, one for interval end, one to calculate partial sum. At every step, compare obtained sum with previous maximum.
2. Uses 2 loops; final loop of previous implementation can be eliminated by calculating partial sums using the second loop

Maximum subarray sum

Divide & Conquer implementation

1. **Divide** - The subarray we are searching for can be in one of only 3 places:
   1. Contained in left side of array.
   2. Contained in right side of array.
   3. Subarray includes middle element of array.
2. **Conquer** - Calculate alternatives using 2 recursions and one $O(n)$ algorithm. Final complexity is $O(n * log_2(n))$
3. **Combine** - Return maximum value.

Maximum subarray sum

Dynamic programming implementation

- We iterate over the array once (so $O(n)$ complexity).
- For each index, we calculate the maximum array ending at that position. If the sum is a new maximum, we record it.

Implementation

The source code for all implementations can be found in **ex37_maximumSubarraySum.py**

Dynamic programming - 0-1 knapsack problem

Problem statement

You have a collection of items, each having its own weight and utility. You have a knapsack with capacity W. Which of the items can you pack in order to maximize their utility. You cannot break up items (0-1 property), and you cannot pack the same item more than once (bounded version)

Demo

0-1 knapsack problem

Source code in **ex38_01knapsack.py**

Dynamic programming - Egg dropping puzzle

Problem statement

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

Dynamic programming - Egg dropping puzzle

Rules:

- All eggs are equivalent.
- An egg that survives a drop is unharmed and reusable.
- A broken egg cannot be reused.
- 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 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

Let's start with the simplest case...

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

Dynamic programming - Egg dropping puzzle

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.
- In this case, the maximum number of drops is equal to k, the number of floors the building has.

Dynamic programming - Egg dropping puzzle

So, how about if we have more eggs?

- Building has **k** floors, but now we have **n=2** eggs.

Discussion

How do we keep the maximum number of drops to a minimum?

Dynamic programming - Egg dropping puzzle

Possible strategies for the n=2 case...

1. The **n=1** case was basically linear search, so we can try binary search with the first egg (drop it from the mid-level floor).
2. How about dropping from every 20th floor, starting from ground level?

Dynamic programming - Egg dropping puzzle

Let's try to describe the situation in a structured way...

- Imagine we drop the first egg at a given floor **m**.
- If it breaks, we have a maximum of **(m-1)** drops, starting from ground.
- If it does 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
  $m + (m-1) + (m-2) + (m-3) + (m-4) + ... + 1 \geq 100$
- 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

Now for 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 one fewer egg.
  2. **Egg is ok** - Problem is reduced to one with **k-x** floors and same number of 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 $1 \leq x \leq k$}

Demo

Egg dropping puzzle

Full source code available in **ex39_eggDroppingPuzzle.py**

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.