How to construct good algorithms?

Raimondas Čiegis

Matematinio modeliavimo katedra, e-paštas: rc@vgtu.lt

Rugsėjo 11 d., 2023

The dynamic programming method

Let us discuss why variant reselection algorithms are often ineffective. This happens not only because the number of cases is very high, but also because a number of different cases is much less than a number of suproblems generated by direct solution algorithms. The same cases are generated and checked/recalculated many times.

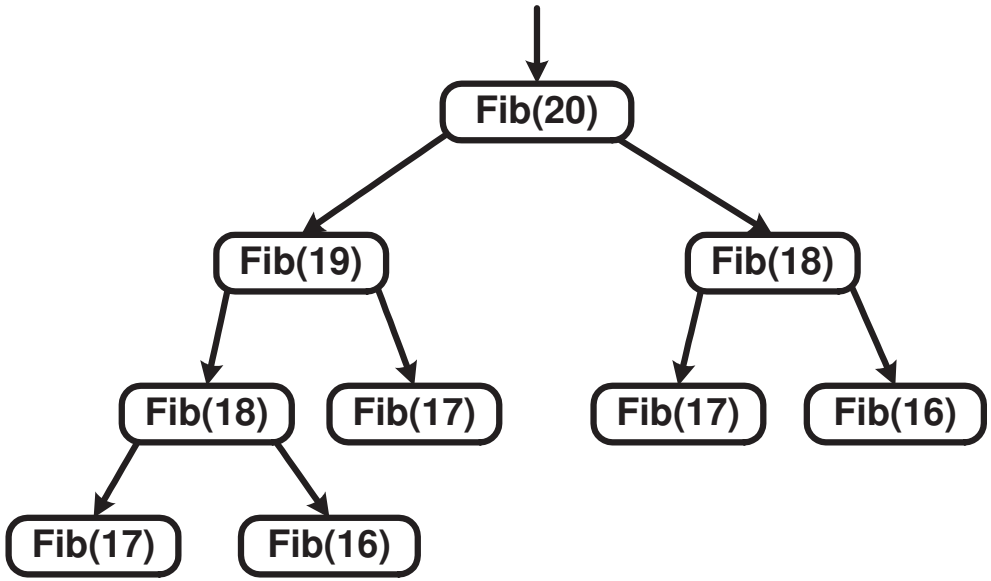
The dynamic programming method

Let us discuss why variant reselection algorithms are often ineffective. This happens not only because the number of cases is very high, but also because a number of different cases is much less than a number of suproblems generated by direct solution algorithms. The same cases are generated and checked/recalculated many times.

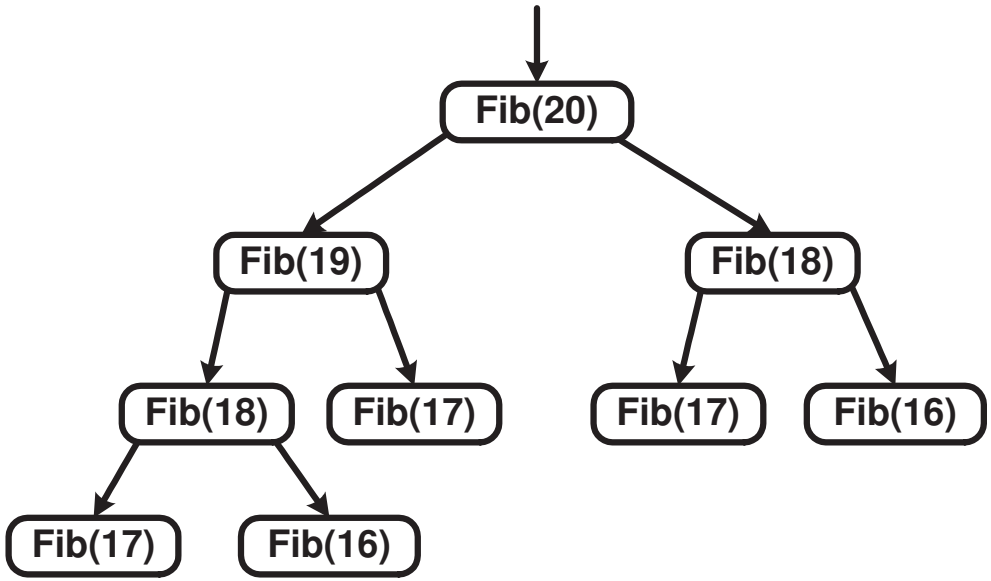
For example recall the recursive algorithm for computation of Fibonacci numbers

$$F(n) = F(n-1) + F(n-2).$$

The figure shows a progress of computations according recursive algorithm when $n = 20$.



The figure shows a progress of computations according recursive algorithm when $n = 20$.



We see that the same cases are computed many times, and the complexity of this algoritm grows very fast.

The given recursive algorithm for computation of Fibonacci numbers works according the "top-down algorithm design" strategy.

The given recursive algorithm for computation of Fibonacci numbers works according the "top-down algorithm design" strategy.

We start by specifying the largest value of the parameter $F(n)$ and then compute the solution by dividing the problem into successively smaller subproblems (in our case we define two new pieces at each division step).

The given recursive algorithm for computation of Fibonacci numbers works according the "top-down algorithm design" strategy.

We start by specifying the largest value of the parameter $F(n)$ and then compute the solution by dividing the problem into successively smaller subproblems (in our case we define two new pieces at each division step).

We can significantly speed up the execution of this algorithm by using memoization technique. The results of expensive function calls are stored and returned back when the same inputs occure again.
Sure, then we have to use additional memory resources (no such thing as a free lunch).

During practical work, compare the execution time (CPU time) of both variants of the algorithm.

During practical work, compare the execution time (CPU time) of both variants of the algorithm.

Also we ask to implement the iterative algorithm described by the "bottom-up" principle.

Dynamic programming method

The dynamic programming method also refers to simplifying a complicated problem by breaking it down into simpler sub-problems in a recursive manner.
This paradigm is recommended when the following conditions are satisfied:

Dynamic programming method

The dynamic programming method also refers to simplifying a complicated problem by breaking it down into simpler sub-problems in a recursive manner.
This paradigm is recommended when the following conditions are satisfied:

1. For standard recursion methods the generated sub-problems essentially overlap, so we solve the same tasks many times.

Dynamic programming method

The dynamic programming method also refers to simplifying a complicated problem by breaking it down into simpler sub-problems in a recursive manner.
This paradigm is recommended when the following conditions are satisfied:

1. For standard recursion methods the generated sub-problems essentially overlap, so we solve the same tasks many times.

In the dynamic programming method, we store solutions of already solved tasks and solve only new sub-problems (similar to memoization paradigm).

2. The more important is the second condition.

If some problem can be solved optimally by breaking it into sub-problems and then recursively finding the optimal solutions to the sub-problems, then it is said to have optimal substructure.

2. The more important is the second condition.

If some problem can be solved optimally by breaking it into sub-problems and then recursively finding the optimal solutions to the sub-problems, then it is said to have optimal substructure.

We assume that sub-problems can be nested recursively inside larger problems and there is a relation between the optimal value of the larger problem and the values of the sub-problems. This relationship is called the Bellman equation.

2. The more important is the second condition.

If some problem can be solved optimally by breaking it into sub-problems and then recursively finding the optimal solutions to the sub-problems, then it is said to have optimal substructure.

We assume that sub-problems can be nested recursively inside larger problems and there is a relation between the optimal value of the larger problem and the values of the sub-problems. This relationship is called the Bellman equation.

The Bellman equation enables us to reduce the total number of subproblems essentially.

In Divide-and-conquere algorithms all sub-problems are generated in top-down fashion, i.e. the full problem is divided in few smaller subproblems and procedure is repeated recursively.

In Divide-and-conquere algorithms all sub-problems are generated in top-down fashion, i.e. the full problem is divided in few smaller subproblems and procedure is repeated recursively.

In Dynamic programming algorithms first smallest sub-problems are solved, the obtained optimal solutions are used to solve larger sub-problems in a bottom-up fashion. Finally, we find the optimal solution of the full problem.

In Divide-and-conquere algorithms all sub-problems are generated in top-down fashion, i.e. the full problem is divided in few smaller subproblems and procedure is repeated recursively.

In Dynamic programming algorithms first smallest sub-problems are solved, the obtained optimal solutions are used to solve larger sub-problems in a bottom-up fashion. Finally, we find the optimal solution of the full problem.

The dynamic programming algorithm considers only those sub-problems, which may be required to define an optimal strategy (the Bellman condition).

Matrix–chain multiplication

We wish to construct the product of n matrices $A_1 A_2 \cdots A_n$, where A_i is a $p_{i-1} \times p_i$ size matrix.

Matrix–chain multiplication

We wish to construct the product of n matrices $A_1 A_2 \cdots A_n$, where A_i is a $p_{i-1} \times p_i$ size matrix.

For example, engineering, virtual reality, animation applications often have to multiply long chains of large size matrices.

Matrix–chain multiplication

We wish to construct the product of n matrices $A_1 A_2 \cdots A_n$, where A_i is a $p_{i-1} \times p_i$ size matrix.

For example, engineering, virtual reality, animation applications often have to multiply long chains of large size matrices.

Matrix multiplication is not commutative, but is associative; and we can multiply only two matrices at a time.

Matrix–chain multiplication

We wish to construct the product of n matrices $A_1 A_2 \cdots A_n$, where A_i is a $p_{i-1} \times p_i$ size matrix.

For example, engineering, virtual reality, animation applications often have to multiply long chains of large size matrices.

Matrix multiplication is not commutative, but is associative; and we can multiply only two matrices at a time.

There are numerous ways to multiply this chain of matrices. They will all produce the same final result, however they will take more or less time to compute, based on which particular matrices are multiplied and in what order.

Recall, that multiplication of $n \times m$ and $m \times k$ size matrices AB requires to compute $2nmk$ arithmetic calculations.

Let us assume that we multiple three matrices $A_1 A_2 A_3$ of sizes 10×200, 200×4 and 4×80.

Recall, that multiplication of $n \times m$ and $m \times k$ size matrices AB requires to compute $2nmk$ arithmetic calculations.

Let us assume that we multiple three matrices $A_1 A_2 A_3$ of sizes 10×200, 200×4 and 4×80.

This result can be calculated in two ways shown below.

Recall, that multiplication of $n \times m$ and $m \times k$ size matrices AB requires to compute $2nmk$ arithmetic calculations.

Let us assume that we multiple three matrices $A_1 A_2 A_3$ of sizes 10×200, 200×4 and 4×80.

This result can be calculated in two ways shown below.

1. $(A_1 A_2)A_3$, this way will require

$$2x10x200x4 + 2x10x4x80 = 16000 + 6400 = 22400$$

operations.

Recall, that multiplication of $n \times m$ and $m \times k$ size matrices AB requires to compute $2nmk$ arithmetic calculations.

Let us assume that we multiple three matrices $A_1 A_2 A_3$ of sizes 10×200, 200×4 and 4×80.

This result can be calculated in two ways shown below.

1. $(A_1 A_2)A_3$, this way will require

$$2x10x200x4 + 2x10x4x80 = 16000 + 6400 = 22400$$

operations.

2. $A_1(A_2 A_3)$, the second way will require

$$2x200x4x80 + 2x10x200x80 = 128000 + 320000 = 448000$$

operations. Thus the second way will require 20 times more computations.

Now we will solve this problem by using the dynamic paradigm.

Now we will solve this problem by using the dynamic paradigm.

First, we must find the Bellman optimality equation and show, that the matrix chain multiplication problem has an optimal substructure, i.e. that the optimal solution of the full problem can be computed by using the optimal solutions of embedded smaller subproblems (of the same structure).

Now we will solve this problem by using the dynamic paradigm.

First, we must find the Bellman optimality equation and show, that the matrix chain multiplication problem has an optimal substructure, i.e. that the optimal solution of the full problem can be computed by using the optimal solutions of embedded smaller subproblems (of the same structure).

Let us denote the result of partial matrix chain multiplication by

$$B_{i,j} = A_i A_{i+1} \cdots A_j.$$

$B_{i,j}$ is a matrix of size $p_{i-1} \times p_j$.

The optimal order of parenthesis (computations) can be defined by the equality

$$A_1 A_2 \cdots A_n = B_{1,k} B_{k+1,n}.$$

The optimal order of parenthesis (computations) can be defined by the equality

$$A_1 A_2 \cdots A_n = B_{1,k} B_{k+1,n}.$$

The last step of the optimal multiplication algorithm will require to multiply matrices $B_{1,k}$ and $B_{k+1,n}$.

The complexity of this step is equal to $2p_0 p_k p_n$ arithmetic computations.

It is clear that in order to compute the result of the last stage we must compute the matrices $B_{1,k}$ and $B_{k+1,n}$.

These sub-problems are of the same structure as the full problem and therefore we can compute them in optimal way by using the same algorithm.

It is clear that in order to compute the result of the last stage we must compute the matrices $B_{1,k}$ and $B_{k+1,n}$.

These sub-problems are of the same structure as the full problem and therefore we can compute them in optimal way by using the same algorithm.

Now we define the Bellman equation.

Let us define the number of aritmetic computations $m(i,j)$ required to multiply matrix chain $A_i A_{i+1} \cdots A_j$ in optimal way.

It follows from the analysis above, that there exists the optimal value of k, such that matrix chain $A_i A_{i+1} \cdots A_j$ multiplication can be divided into two sub-chains.

We compute the multiplication of two matrices $B_{i,k}$ and $B_{k+1,j}$ in optimal way

$$m(i,j) = m(i,k) + m(k+1,j) + 2p_{i-1}p_k p_j.$$

It follows from the analysis above, that there exists the optimal value of k, such that matrix chain $A_i A_{i+1} \cdots A_j$ multiplication can be divided into two sub-chains.

We compute the multiplication of two matrices $B_{i,k}$ and $B_{k+1,j}$ in optimal way

$$m(i,j) = m(i,k) + m(k+1,j) + 2p_{i-1}p_k p_j.$$

Since it is not known in advance what k must be selected, we get the variational Bellman equation

$$m(i,j) = \begin{cases} 0, & i = j, \\ \min\limits_{i \le k < j} \left(m(i,k) + m(k+1,j) + 2p_{i-1}p_k p_j \right), & i < j. \end{cases}$$

For each $m(i,j)$ the optimal value of k is stored in the element $p(i,j)$ of matrix P.

Number $m(1, n)$ defines the complexity of the optimal algorithm for computation of matrix chain $A_1 A_2 \cdots A_n$ multiplication.

Optimal order for multiplication of six matrices

Let us multiply matrices

$$A_1 A_2 A_3 A_4 A_5 A_6,$$

where sizes of matrices are the following: 40×50, 50×20, 20×4, 4×15, 15×25, 25×35.

Optimal order for multiplication of six matrices

Let us multiply matrices

$$A_1 A_2 A_3 A_4 A_5 A_6,$$

where sizes of matrices are the following: 40×50, 50×20, 20×4, 4×15, 15×25, 25×35.

The dynamic programming algorithm computes iteratively the values of diagonal elements of matrices M and P. It starts computations from the main diagonal and moves forward to neighbour diagonals.

In the Figure 1, elements for each diagonal are presented in different colors.

					0
				0	26250
			0	3000	10000
		0	2400	7000	15600
	0	8000	14000	21000	32000
0	80000	24000	28800	35000	45200

a) matrix M

					5
				4	5
			3	3	3
		2	3	3	3
	1	1	3	3	3

b) matrix P

					0
				0	26250
			0	3000	10000
		0	2400	7000	15600
	0	8000	14000	21000	32000
0	80000	24000	28800	35000	45200

a) matrix M

					5
				4	5
			3	3	3
		2	3	3	3
	1	1	3	3	3

b) matrix P

It follows that the optimal order of multiplications is defined as

$$(A_1(A_2A_3))((A_4A_5)A_6).$$

The complexity of the optimal multiplication algorithm is equal to 45200 arithmetic computations.

					0
				0	26250
			0	3000	10000
		0	2400	7000	15600
	0	8000	14000	21000	32000
0	80000	24000	28800	35000	45200

Matrix M

As an example, we compute the value of $m(2,4)$. The sizes of matrices A_2, A_3, A_4 are defined as $50x20$, $20x4$, $4x15$.

					0
				0	26250
			0	3000	10000
		0	2400	7000	15600
	0	8000	14000	21000	32000
0	80000	24000	28800	35000	45200

Matrix M

As an example, we compute the value of $m(2,4)$. The sizes of matrices A_2, A_3, A_4 are defined as $50x20$, $20x4$, $4x15$.
First we consider $k = 2$:

$$m(2,2) + m(3,4) + 2 \cdot 50 \cdot 20 \cdot 15 = 0 + 2400 + 30000 = 32400,$$

the second possibility is $k = 3$:

$$m(2,3) + m(4,4) + 2 \cdot 50 \cdot 4 \cdot 15 = 8000 + 0 + 6000 = 14000.$$

Thus the minimum value is obtained when $k = 3$, and the computation order is the following $(A_2A_3)A_4$.

Greedy algorithms

We often find ourselves in situations where we have to make a decision "here and now". This choice will also affect future results, e.g. the company's profit or the duration of the trip. But to solve this problem by using the other well known algorithms such as divide and conquere, dynamic programming or full selection of options we do not have possibilities (they would take too long).

A greedy algorithm is any algorithm that follows the problem-solving heuristic of making the locally optimal choice at each stage.

A greedy algorithm is any algorithm that follows the problem-solving heuristic of making the locally optimal choice at each stage.

In many problems, a greedy strategy does not produce an optimal solution, but a greedy heuristic can yield locally optimal solutions that approximate a globally optimal solution in a reasonable amount of time.

A greedy algorithm is any algorithm that follows the problem-solving heuristic of making the locally optimal choice at each stage.

In many problems, a greedy strategy does not produce an optimal solution, but a greedy heuristic can yield locally optimal solutions that approximate a globally optimal solution in a reasonable amount of time.

We usually divide the search for the solution into n stages and at each step we choose a solution from a small finite number m of options. Thus we only test small part of the all possible options.

Greedy algorithms find a locally optimal solution in nm basic operations, thus these algorithms are very fast.

Greedy algorithms find a locally optimal solution in nm basic operations, thus these algorithms are very fast.

We should repeat once more, that in most cases the locally optimal solutions approximate a globally optimal solution with a reasonable accuracy. Thus such greedy algorithms are only heuristcs.

Greedy algorithms find a locally optimal solution in nm basic operations, thus these algorithms are very fast.

We should repeat once more, that in most cases the locally optimal solutions approximate a globally optimal solution with a reasonable accuracy. Thus such greedy algorithms are only heuristcs.

However, there are important applications when greedy algorithms define the exact solution. We will study such applications later.

How to calculate the optimal coin change?

Given an integer array of coins representing different types of denominations

$$V_1 > V_2 > \ldots > V_m$$

and an integer sum G, the task is to find a way to make this sum by using the minimum number of coins.

How to calculate the optimal coin change?

Given an integer array of coins representing different types of denominations

$$V_1 > V_2 > \ldots > V_m$$

and an integer sum G, the task is to find a way to make this sum by using the minimum number of coins.

Thus we solve the following task:

$$\min_{(n_1,\ldots,n_m)\in D}(n_1 + n_2 + \ldots + n_m),$$

$$D = \{n_1 V_1 + n_2 V_2 + \ldots + n_m V_m = G, \quad n_j \geq 0\}.$$

Let us assume that $V_m = 1$, thus we allways can find at least one combination to make the required sum G.

Let us assume that $V_m = 1$, thus we allways can find at least one combination to make the required sum G.

Assume that you have an infinite supply of each type of coin.

Let us assume that $V_m = 1$, thus we allways can find at least one combination to make the required sum G.

Assume that you have an infinite supply of each type of coin.

The greedy algorithm. The idea is very simple:

First, we try to collect a change by using coins with largest denominations, the number of such coins is equal to $n_1 = \lfloor G/V_1 \rfloor$, then a bigger part of the remaining sum $G_1 = G - n_1 V_1$ is returned by using coins with nomination V_2 $n_2 = \lfloor G_1/V_2 \rfloor$ and so on.

Let us assume that $V_m = 1$, thus we allways can find at least one combination to make the required sum G.

Assume that you have an infinite supply of each type of coin.

The greedy algorithm. The idea is very simple:

First, we try to collect a change by using coins with largest denominations, the number of such coins is equal to $n_1 = \lfloor G/V_1 \rfloor$, then a bigger part of the remaining sum $G_1 = G - n_1 V_1$ is returned by using coins with nomination V_2 $n_2 = \lfloor G_1/V_2 \rfloor$ and so on.

If at some stage of the algorithm $G_j < V_{j+1}$, then coins with nomination V_{j+1} are not used.

Let us assume that we have a collection of coins

$$V_1 = 25, \quad V_2 = 11, \quad V_3 = 5, \quad V_4 = 1.$$

Applying the greedy algorithm we compute that a change of 63 cents can be calculated as

$$63 = 2 \times 25 + 1 \times 11 + 2 \times 1,$$

thus 5 coins are used. It is easy to check that this solution is optimal.

Let us assume that we have a collection of coins

$$V_1 = 25, \quad V_2 = 11, \quad V_3 = 5, \quad V_4 = 1.$$

Applying the greedy algorithm we compute that a change of 63 cents can be calculated as

$$63 = 2 \times 25 + 1 \times 11 + 2 \times 1,$$

thus 5 coins are used. It is easy to check that this solution is optimal.

If the sum is equal to $G = 15$, then the greedy algorithm gives the answer

$$15 = 1 \times 11 + 4 \times 1,$$

thus again we use 5 coins.

Let us assume that we have a collection of coins

$$V_1 = 25, \quad V_2 = 11, \quad V_3 = 5, \quad V_4 = 1.$$

Applying the greedy algorithm we compute that a change of 63 cents can be calculated as

$$63 = 2 \times 25 + 1 \times 11 + 2 \times 1,$$

thus 5 coins are used. It is easy to check that this solution is optimal.

If the sum is equal to $G = 15$, then the greedy algorithm gives the answer

$$15 = 1 \times 11 + 4 \times 1,$$

thus again we use 5 coins.

But it is easy to check that there exists a better solution with only three coins: $15 = 3 \times 5$.

Discrete knapsack problem

Given a set of n items, each with a volume v_j and a value p_j, $j = 1, \ldots, n$, determine which items to include in the collection so that the total volume is less than or equal to a given limit V and the total value is as large as possible.

Discrete knapsack problem

Given a set of n items, each with a volume v_j and a value p_j, $j = 1, \ldots, n$, determine which items to include in the collection so that the total volume is less than or equal to a given limit V and the total value is as large as possible.

The most common problem being solved is the 0-1 knapsack problem, which restricts the number x_i of copies of each kind of item to zero or one.

Discrete knapsack problem

Given a set of n items, each with a volume v_j and a value p_j, $j = 1, \ldots, n$, determine which items to include in the collection so that the total volume is less than or equal to a given limit V and the total value is as large as possible.

The most common problem being solved is the 0-1 knapsack problem, which restricts the number x_i of copies of each kind of item to zero or one.

Given a set of n items numbered from 1 up to n, each with a volume v_i and a value p_i, along with a maximum volume capacity V. We solve the following problem

$$\max_{(x_1,\ldots,x_n)\in D} (x_1 p_1 + x_2 p_2 + \ldots + x_n p_n),$$

$$D = \{x_1 v_1 + x_2 v_2 + \ldots + x_n v_n \leq V, \quad x_j \in (0, 1)\}.$$

First, we define a relative value of each item $s_j = p_j / v_j$.

Next we sort items according the obtained relative values

$$s_1 \geq s_2 \geq \ldots \geq s_n.$$

First, we define a relative value of each item $s_j = p_j/v_j$.

Next we sort items according the obtained relative values

$$s_1 \geq s_2 \geq \ldots \geq s_n.$$

A greedy strategy. At each step, we try to put into the bag the most valuable item.

If this item is too big to fit into the remaing empty place of the bag, then we take the next item and repeat this procedure.

Assume that we have eight items (v_j, p_j):

$$(25, 50), \quad (20, 80), \quad (20, 50), \quad (15, 45),$$
$$(30, 105), \quad (35, 35), \quad (20, 10), \quad (10, 45).$$

Assume that we have eight items (v_j, p_j):

$$(25, 50), \quad (20, 80), \quad (20, 50), \quad (15, 45),$$
$$(30, 105), \quad (35, 35), \quad (20, 10), \quad (10, 45).$$

First we compute the relative value of each item

$$S = \{\, 2, \quad 4, \quad 2.5, \quad 3, \quad 3.5, \quad 1, \quad 0.5, \quad 4.5 \,\}$$

and sort them

$$(10, 45), \quad (20, 80), \quad (30, 105), \quad (15, 45),$$
$$(20, 50), \quad (25, 50), \quad (35, 35), \quad (20, 10).$$

Assume that we have eight items (v_j, p_j):

$$(25, 50), \quad (20, 80), \quad (20, 50), \quad (15, 45),$$
$$(30, 105), \quad (35, 35), \quad (20, 10), \quad (10, 45).$$

First we compute the relative value of each item

$$S = \{ 2, \quad 4, \quad 2.5, \quad 3, \quad 3.5, \quad 1, \quad 0.5, \quad 4.5 \}$$

and sort them

$$(10, 45), \quad (20, 80), \quad (30, 105), \quad (15, 45),$$
$$(20, 50), \quad (25, 50), \quad (35, 35), \quad (20, 10).$$

Let us assume that a volume of our bag is equal to $V = 80$.

Applyting the greedy algorithm we put four most valuable items into it, the total volume of all selected items is 75, and their value is equal to 275.

$$(10, 45),\ (20, 80),\ (30, 105),\ (15, 45),$$
$$(20, 50),\ (25, 50),\ (35, 35),\ (20, 10).$$

This combination is not optimal, since if we take the first three items and then the fifth one, then the bag is fully filled and the value of all selected items is equal to 280.

$$(10, 45),\ (20, 80),\ (30, 105),\ (15, 45),$$
$$(20, 50),\ (25, 50),\ (35, 35),\ (20, 10).$$

This combination is not optimal, since if we take the first three items and then the fifth one, then the bag is fully filled and the value of all selected items is equal to 280.

Thus for this problem the greedy strategy defines only an heuristic.

In conclusion we can state that an introduction is given for the following five methods

testing of all cases,

recursion,

divide-and-conquer,

dinamic programming,

greedy search.

In conclusion we can state that an introduction is given for the following five methods

testing of all cases,

recursion,

divide-and-conquer,

dinamic programming,

greedy search.

In fact exactly these methods will be used during all our lectures. A more detailed analysis of each of them will be presented.

Now let us consider a simple but very useful test problem. It can be solved by using different algorithms.

The construction of such algorithms is done by applying general methods proposed above.

We start by considering 1D problem

Assume that one-dimensional array A is given.

All elements $A[i]$, $i = 1, \ldots, n$ are positive integer numbers (natural numbers).

We start by considering 1D problem

Assume that one-dimensional array A is given.

All elements $A[i]$, $i = 1, \ldots, n$ are positive integer numbers (natural numbers).

We will say that $A[j]$ defines a peak, if

$$A[j-1] \leq A[j] \leq A[j+1], \quad 1 < j < n.$$

We start by considering 1D problem

Assume that one-dimensional array A is given.

All elements $A[i]$, $i = 1, \ldots, n$ are positive integer numbers (natural numbers).

We will say that $A[j]$ defines a peak, if

$$A[j-1] \leq A[j] \leq A[j+1], \quad 1 < j < n.$$

This definition is modified at the end points, e.g. $A[1]$ is a peak, if $A[1] \geq A[2]$.

Thus some element is a peak, if its neighbours are not larger than it's value.

It is sufficient to check a local information in order to test the existence of a peak at $A[j]$.

Thus some element is a peak, if its neighbours are not larger than it's value.

It is sufficient to check a local information in order to test the existence of a peak at $A[j]$.

It is easy to prove that at least one peak always exists (do it Yourself).

Algorithm 1. Let us test all cases. We start a search from element $A[1]$, if $A[2] > A[1]$, then $A[1]$ is not a peak.

Algorithm 1. Let us test all cases. We start a search from element $A[1]$, if $A[2] > A[1]$, then $A[1]$ is not a peak.

Then we continue the search procedure and test the condition $A[2] \geq A[3]$. If this condition is also not satisfied we move forward and test the next element $A[3]$.

This process is continued till we find a peak.

Next we estimate the complexity of the proposed algorithm. Let us assume that with equal probability any element can define the first peak.

Then in the worst case all elements of A should be tested and $T_B(n) = n$.

The average case complexity is similar $T_V(n) = \frac{1}{2}n$ (i.e. the complexity depends linearly on the size of A).

Algoritmas 2. It is based on the divide-and-conquer method.

We start a search from the middle element $A[n/2]$. If

$$A[n/2-1] \leq A[n/2] \leq A[n/2+1],$$

then $A[n/2]$ is a peak. It was sufficient to make a comparison only twice.

Algoritmas 2. It is based on the divide-and-conquer method.

We start a search from the middle element $A[n/2]$. If

$$A[n/2-1] \leq A[n/2] \leq A[n/2+1],$$

then $A[n/2]$ is a peak. It was sufficient to make a comparison only twice.

Otherwise we continue our search and select that side of A in the direction of which the inequality was not satisfied.

If both directions are possible, the selection is done in random.

Algoritmas 2. It is based on the divide-and-conquer method.

We start a search from the middle element $A[n/2]$. If

$$A[n/2-1] \leq A[n/2] \leq A[n/2+1],$$

then $A[n/2]$ is a peak. It was sufficient to make a comparison only twice.

Otherwise we continue our search and select that side of A in the direction of which the inequality was not satisfied.

If both directions are possible, the selection is done in random.

We note that after the first step a set of active elements is reduced twice.

For example, we need to test only elements $A[i]$, $i = 1, \ldots, n/2-1$.

The basic step is repeated while a peak is computed.

It is important to remenber that if A has only one element, then this element defines a peak.

Next we will estimate the complexity if this new algorithm.

Next we will estimate the complexity if this new algorithm.

Since after each iteration the remaining number of elements is reduced twice, it is sufficient to make no more than $\log n$ iterations.

We get that the complexity of the second algorithm even in the worst case is equal to

$$T_B(n) = 2 \log n.$$

Next we will estimate the complexity if this new algorithm.

Since after each iteration the remaining number of elements is reduced twice, it is sufficient to make no more than $\log n$ iterations.

We get that the complexity of the second algorithm even in the worst case is equal to

$$T_B(n) = 2 \log n.$$

Assume $n = 1000000$, then $\log n = 20$, thus this algorithm is much better than the first one.

Next, let us consider a 2D version of this problem, when matrix A has dimension $m \times n$.

Next, let us consider a 2D version of this problem, when matrix A has dimension $m \times n$.

A peak element is at (i, j), if its neighbours in raw and column directions are not larger than the element itself

$$A[i][j] \geq A[i-1][j], \quad A[i][j] \geq A[i+1][j],$$
$$A[i][j] \geq A[i][j-1], \quad A[i][j] \geq A[i][j+1].$$

This definition should be modified at boundaries.

Algoritmas 1. The greedy algorithm.

We select an element (i, j) as a starting point in our search.

If this element is not a peak, then we select the largest neighbour.

This procedure is continued till a peak is selected.

Algoritmas 1. The greedy algorithm.

We select an element (i, j) as a starting point in our search.

If this element is not a peak, then we select the largest neighbour.

This procedure is continued till a peak is selected.

It is easy to prove that this greedy algorithm always leads to a peak element.

Let's consider an example. We start our search from element 12:

$$\begin{pmatrix} 13 & 11 & 9 & 22 \\ 14 & 13 & 12 & 10 \\ 15 & 9 & 11 & 17 \\ 16 & 17 & 19 & 21 \end{pmatrix}.$$

Let's consider an example. We start our search from element 12:

$$\begin{pmatrix} 13 & 11 & 9 & 22 \\ 14 & 13 & 12 & 10 \\ 15 & 9 & 11 & 17 \\ 16 & 17 & 19 & 21 \end{pmatrix}.$$

The algorithm moves forward in the following way:

$$\begin{pmatrix} 13 & 11 & 9 & 22 \\ 14 & 13 & 12 & 10 \\ 15 & 9 & 11 & 17 \\ 16 & 17 & 19 & 21 \end{pmatrix}.$$

The logic of this greedy algorithm is simple, but the complexity of the worst and average cases is proportional to $\Theta(n\,m)$.

Thus we should test a bigger part of all elements.

Algoritmas 2. Divide-and-conquer strategy.

1. We select a middle column of the matrix j= m/2.

Algoritmas 2. Divide-and-conquer strategy.

1. We select a middle column of the matrix j= m/2.
2. We find the largest element $A(i, j)$ in this column.

Algoritmas 2. Divide-and-conquer strategy.

1. We select a middle column of the matrix j= m/2.

2. We find the largest element $A(i, j)$ in this column.

3. We compare $A(i, j)$ with neighbours at the same raw $A(i, j-1)$ and $A(i, j+1)$.
If both neighbours are not larger, then (i, j) is a peak.

Algoritmas 2. Divide-and-conquer strategy.

1. We select a middle column of the matrix j= m/2.

2. We find the largest element $A(i, j)$ in this column.

3. We compare $A(i, j)$ with neighbours at the same raw $A(i, j-1)$ and $A(i, j+1)$.
If both neighbours are not larger, then (i, j) is a peak.

4. Otherwise, we select that half of the matrix, which contain the largest element.

Thus the amount of remaining columns is reduced twice.

Algoritmas 2. Divide-and-conquer strategy.

1. We select a middle column of the matrix j= m/2.

2. We find the largest element $A(i, j)$ in this column.

3. We compare $A(i, j)$ with neighbours at the same raw $A(i, j-1)$ and $A(i, j+1)$.
If both neighbours are not larger, then (i, j) is a peak.

4. Otherwise, we select that half of the matrix, which contain the largest element.

Thus the amount of remaining columns is reduced twice.

5. The basic search step is repeated in the new matrix.

Algoritmas 2. Divide-and-conquer strategy.

1. We select a middle column of the matrix j= m/2.

2. We find the largest element $A(i, j)$ in this column.

3. We compare $A(i, j)$ with neighbours at the same raw $A(i, j-1)$ and $A(i, j+1)$.
If both neighbours are not larger, then (i, j) is a peak.

4. Otherwise, we select that half of the matrix, which contain the largest element.

Thus the amount of remaining columns is reduced twice.

5. The basic search step is repeated in the new matrix.

If only one column is remained in the matrix, then its largest element is a peak.

Let's esimate the complexity of the algorithm in the worst case

$$\begin{aligned} T(n,m) &= T(n,m/2) + \Theta(n) \\ &= T(n,m/4) + 2\Theta(n) \\ &\dots \\ &= T(n,1) + \log(m)\,\Theta(n) \\ &= \log(m)\,\Theta(n). \end{aligned}$$

Let's esimate the complexity of the algorithm in the worst case

$$\begin{aligned} T(n,m) &= T(n,m/2) + \Theta(n) \\ &= T(n,m/4) + 2\Theta(n) \\ &\dots \\ &= T(n,1) + \log(m)\,\Theta(n) \\ &= \log(m)\,\Theta(n). \end{aligned}$$

Explain how to modify the algorithm if $m \ll n$?