Ministry of Education and Science of Ukraine

National Technical University

"Kharkiv Polytechnical Institute"

Department of Computer-Aided Management Systems

Guidance
on laboratory works
for course "Algorithms and Data Structures"

for students of directions

"Software Engineering"

Kharkov 2013

Guidance on laboratory works for course "Algorithms and Data Structures" for students of directions "Software Engineering"

Compiler: Stratienko N. K.

TABLE OF CONTENTS

RECOMMENDATIONS FOR WORKS

The objectives of laboratory work are:

– formation of practical skills and use in solving problems of dynamic data structures;

– study of the most common algorithms for solving problems using complex data structures.

Tasks can be perform in a chosen programming language:

– C;

– C++;

– Pascal (rarely used today);

– Java;

– C#;

– Python;

– Ruby.

It is not recommended to use Basic or PHP because they give bad programming style. Also assembly language should be avoided because they add too many details to the algorithm.

The program can be demonstrated on any platform. It is recommended to develop an application so that it could be built and run on a wide class of systems (it is easy to reach it using the recommended languages). It is recommended to use any GNU / Linux distro or FreeBSD.

Choosing a development environment may be dictated by the student's habits. It is advisable to use multiple environments for different works.

Here are a few important free/open editors and development environments:

– Vim;

– Emacs;

– Eclipse;

– NetBeans;

- IntelliJ IDEA;
- KDevelop;
- MonoDevelop;
- Lazarus (a free environment like Delphi).

1 LABORATORY WORK 1. BASIC DATA STRUCTURES (1 PAIR)

Objective: explore basic data structures (list, queue, and stack) and get programming skills of algorithms that process them.

1.1 Guidance

Stacks and queues are dynamic sets in which the element removed from the set by the DELETE operation is prespecified. In a stack, the element deleted from the set is the one most recently inserted: the stack implements a last-in, first-out, or LIFO, policy. Similarly, in a queue, the element deleted is always the one that has been in the set for the longest time: the queue implements a first-in, first-out, or FIFO, policy. There are several efficient ways to implement stacks and queues on a computer.

A linked list is a data structure in which the objects are arranged in a linear order. Unlike an array, however, in which the linear order is determined by the array indices, the order in a linked list is determined by a pointer in each object. Linked lists provide a simple, flexible representation for dynamic sets.

Each element of a doubly linked list L is an object with an attribute key and two other pointer attributes: next and prev. The object may also contain other satellite data. Given an element x in the list, x:next points to its successor in the linked list, and prev[x] points to its predecessor. If prev[x] = NIL, the element x has no predecessor and is therefore the first element, or head, of the list. If next[x] = NIL, the element x has no successor and is therefore the last element, or tail, of the list. An attribute head[L] points to the first element of the list. If head[L] =NIL , the list is empty.

A list may have one of several forms. It may be either singly linked or doubly linked, it may be sorted or not, and it may be circular or not. If a list is singly linked, we omit the pre pointer in each element. If a list is sorted, the linear order of the list corresponds to the linear order of keys stored in elements of the list; the minimum element is then the head of the list, and the maximum element is the tail. If the list is unsorted, the elements can appear in any order. In a circular list, the prev pointer of the head of the list points to the tail, and the next pointer of the tail of the list points to the head. We can think of a circular list as a ring of elements. In the remainder of this section, we assume that the lists with which we are working are unsorted and doubly linked.

1.2 Task

Develop a program that reads from the keyboard sequence of N unique integers N (1 <N <256), saves them to a data structure (according to task) and displays the following characteristics:

- number of elements;
- average of saved items;
- minimum and maximum element;
- fourth element of the sequence;
- element that is before the minimum element.

We emphasize that all the characteristics necessary must be define for a filled data structure. Only those operations that are inherent to given structure are allowed, for example, one must not access an element on an arbitrary position in a queue which is based on an array.

Usage of ready data structures (e.g., STL) is prohibited.

1.3 Variants

Use the following data structures.

1 Queue.

2 Stack.

3 Singly linked list.

4 Doubly linked list.

5 Circular list.

6 Random access array.

2 LABORATORY WORK 2. BASIC DATA STRUCTURES. HASH TABLES (1 PAIR)

Objective: explore hash functions and hash tables and get programming skills of algorithms that process them.

2.1 Guidance

Many applications require a dynamic set that supports only the dictionary opera- tions INSERT, SEARCH, and DELETE. A hash table is an effective data structure for implementing dictionaries. Although searching for an element in a hash table can take as long as searching for an element in a linked list— $O(n)$ time in the worst case—in practice, hashing performs extremely well. Under reasonable assumptions, the average time to search for an element in a hash table is $O(1)$.

With direct addressing, an element with key k is stored in slot k . With hashing, this element is stored in slot $h(k)$; that is, we use a hash function h to compute the slot from the key k . Here, h maps the universe U of keys into the slots of a hash table $T[0..m-1]$: $h:U \rightarrow 0,1,...,m-1$.

A good hash function should (approximately) satisfy the assumptions of uniform hashing: for regular key m all the hash values must have equal expectation.

Trivial hash is to use a hash function that returns its own argument.

Construction of the hash function by division with a remainder (division method) is that the key k is associated with $h(k)=k\,mod\,m$, where m is the number of possible hash values.

For example, if the size of the hash table $m=12$ and the key is 100, then the hash value is 4.

Good results are usually obtained if m is as prime number far from powers of two.

Construction of the hash function by multiplication (multiplication method) is as follows. Let m be the number of hash values. Fix a constant A in interval

$(0,1)$ and let $h(k)=[m(kA \bmod 1)]$, where $kA \bmod 1$ is fractional part of kA .

Advantage of multiplication method is that the quality of the hash function only slightly dependents on the choice of m . Usually m is chosen as a power of two, since for most computers multiplication is implemented as shift.

Pearson hashing is an algorithm proposed by Peter Pearson for processors with 8-bit registers, whose task is to quickly compute a hash code for a string of arbitrary length. Function input is a string W of n characters, each occupies 1 byte. Function returns a value between 0 and 255. The value of the hash code depends on each character of the input string.

The algorithm can be described in Python. T is a table of permutations.

```
def hash_pearson(w):
    h = 0
    for c in w:
        index = h ^ ord(c)
        h = T[index]
    return h
```

2.2 Task

Develop a program that reads integers N, M (1 <N, M <256) from keyboard, N <key, value> pairs (key is an integer, floating point number or a string, depending on the variant, value is a string, all strings up to 255 characters), none of which is repeated, and M keys. All strings separated by a space or a new line. The program stores pairs of strings into a hash table, and displays the values corresponding to listed keys.

Example of key-lines input.

```
3 2
abc x
gh yq
io qw
gh
io
```

Output.

```
yq
qw
```

Usage of ready data structures (e.g., STL) is prohibited, but string implementations can be used (for example, std::string in C++).

2.3 Variants

1 Key is an integer; trivial hashing.

2 Key is an integer, division method.

3 Key is an integer; multiplication method.

4 Key is a real number, multiplication method.

5 Key is a string; hash is a remainder of characters sum.

6 Key is a string; Pearson hashing.

3 LABORATORY WORK 3. BASIC DATA STRUCTURES. RED-BLACK TREES (2 PAIRS)

Objective: explore red-black trees and get programming skills of algorithms that process them.

3.1 Guidance

The search tree data structure supports many dynamic-set operations, including SEARCH, MINIMUM, MAXIMUM, PREDECESSOR, SUCCESSOR, INSERT, and DELETE. Thus, we can use a search tree both as a dictionary and as a priority queue.

Basic operations on a binary search tree take time proportional to the height of the tree. For a complete binary tree with n nodes, such operations run in $\Theta(lg\, n)$ worst-case time. If the tree is a linear chain of n nodes, however, the same operations take $\Theta(n)$ worst-case time.

Thus, the set operations are fast if the height of the search tree is small. If its height is large, however, the set operations may run no faster than with a linked list. Red-black trees are one of many search-tree schemes that are “balanced” in order to guarantee that basic dynamic-set operations take O(lg n) time in the worst case.

A red-black tree is a binary search tree with one extra bit of storage per node: its color, which can be either RED or BLACK. By constraining the node colors on any simple path from the root to a leaf, red-black trees ensure that no such path is more than twice as long as any other, so that the tree is approximately balanced.

Each node of the tree now contains the attributes color, key, left, right, and p. If a child or the parent of a node does not exist, the corresponding pointer attribute of the node contains the value NIL. We shall regard these NILs as being pointers to leaves (external nodes) of the binary search tree and the normal, key-bearing nodes as being internal nodes of the tree.

A red-black tree is a binary tree that satisfies the following red-black properties.

1. Every node is either red or black.
2. The root is black.
3. Every leaf (NIL) is black.
4. If a node is red, then both its children are black.
5. For each node, all simple paths from the node to descendant leaves contain the same number of black nodes

Tree-Insert and Tree-Delete operations take $O(lg\,n)$ on a red-black tree, they do not guarantee that the modified binary search tree will be a red-black tree. A tree can require modification and rotations. This process is described in lectures and recommended sources.

3.2 Task

Develop a program that reads numbers N, M (1 <N, M <256), a sequence of N keys (integers, real numbers or strings (up to 255 characters) depending on the variant), and a sequence of M keys. The program saves the first sequence to red-black trees.

Whenever a new element is added to tree, statistics must be display according to variant.

1 The minimum element and its color;

2 The maximum element and its color.

After building a tree, results of the following operations must be shown for the tree and for every element x of the second sequence (according to variant).

1 Does item x exist in the tree and what is its color.

2 Successor(x) and its color.

3 Predecessor(x) and its color.

Usage of ready data structures (e.g., STL) is prohibited, but string implementations can be used (for example, std::string in C++).

3.3 Variants

1 1.1.

2 1.2.

3 1.3.

4 2.1.

5 2.2.

6 2.3.

4 LABORATORY WORK 4. SORTING ALGORITHMS (2 PAIRS)

Objective: to explore algorithms of sorting and binary search.

4.1 Guidance

Many algorithms use sorting as an intermediate step. There are many different sorting algorithms; the choice of a particular situation depends on the length of the sorted sequence, of degree how much it is already sorted, and the type of available memory.

Bubble sort consists of repeated passes through the sorted array. For each pass, elements are sequentially compared in pairs and if the order in a pair is wrong, they are exchanged. Passes through the array are repeated N - 1 times or until a new pass detects that no exchanges are needed, which means that the array is sorted. With each pass of the inner loop of the algorithm, the next largest element of the array is put into place at the end of the array next to the previous largest element and the smallest element is moved one position to the beginning of the array ("pops up "to the desired position as a bubble in the water – here the name of the algorithm).

Insertion sort is useful for short sequences. In this way, cards are usually sorted: one holds already sorted cards in the left hand and takes the next card from unsorted set in the right hand to insert it in the correct place, comparing it with existing cards in the left hand and going from right to left.

Quicksort is based on “divide and conqueror” rule. Sorting of $A[p..r]$ segment is done this way.

1 Elements of array A are reordered so that any element from $A[p],...,A[q]$ is not more than any element from $A[q+1];...;A[r]$, where q is a number in the interval $p..r$. This operation is called partition.

2 Sorting is applied to $A[p..q]$ and $A[q+1..r]$ recursively.

3 $A[p..r]$ Is sorted now.

Merge sort is based on "divide and conqueror", too. First, we divide the array into two half smaller parts. Then we sort each half separately. Then we can merge the two ordered array into one. Recursive partitioning of the problem into smaller ones occurs until as long as the array size is more that 1 (any array of length 1 can be considered as ordered).

Nontrivial part is to connect the two ordered arrays into one. It is performed by an auxiliary procedure $Merge(A,p,q,r)$. The parameters of this procedure are array A and numbers p,q,r indicating the border of merged parts. The procedure assumes that $p<q<r$ and that areas $A[p..q]$ and $A[q+1..r]$ are sorted and merges them into one section $A[p..r]$.

Heapsort is based on sorting tree (binary heap). Sorting tree is a binary tree that meets the following conditions.

1 Any leaf has depth d or $d-1$, where d is the maximal depth of the tree.

2 Value in any vertex is not less (in the other option “is not more”) than value of its descendants.

A useful structure for sorting tree is an array A that $A[0]$ is the root element and descendants of $A[i]$ are $A[2i+1]$ and $A[2i+2]$.

Sorting algorithm consists of two main steps.

1 Build a sorting tree that each child is smaller than its parent. This step requires $O(n)$ operations.

2 Remove elements from the tree root one by one, place it after the end of the array, and rebuild the rest of the tree. This way, the array will be filled with sorted elements, starting from the end. Elements $A[0]$ are $A[n-1]$ swapped on the first step and $A[0],A[1],\dots,A[n-2]$ is rebuilt a a sorting tree. Then $A[0]$ and $A[n-2]$ are swapped and $A[0],A[1],\dots,A[n-3]$ is rebuilt. The process continues until only one element lefts in the tree. Then $A[0],A[1],\dots,A[n-1]$ is an sorted sequence. This steps require $O(n\log n)$ operations.

Counting sort can be applied if all the elements of the sequence are positive integers in a certain range (which does not exceed the pre-known k). If $k=O(n)$, then sorting algorithm takes $O(n)$.

The idea of this algorithm is to precalculate for each item x how many elements of the input sequence are less then x , and write x directly to the output array in accordance with that number (if, say, 17 elements of the input array are less then x , then x should be written in place number 18). If the sorted sequence can contains equal numbers, this scheme should be slightly modified to avoid placing equals numbers into one place.

4.2 Task

Develop a program that reads from the keyboard numbers N, M (1 < N, M < 256), a sequence of N keys (integers or real numbers, depending on the variant), and a sequence of M keys. The program saves the first sequence into an array and performs sorting. The program then displays the sorted sequence on the screen and performs a binary search for each element of the second sequence x: for every x announce whether he is in the first sequence, and if so, where is it.

4.3 Variants

1 Bubble sort.

2 Insertion sort.

3 Quicksort.

4 Merge sort.

5 Heapsort.

6 Counting sort (for integers).

5 LABORATORY WORK 5. COMBINATORIAL ALGORITHMS (1 PAIR)

Objective: study random number generators and randomness tests.

5.1 Guidance

Linear congruential method is used in simple cases and has no cryptographic resistance.

Linear congruential method is based on calculating the members of linear recurrent sequence by modulo of some natural number m by the following formula: $X_{k+1}=(aX_k+c)\bmod m$, where a and c are some integer coefficients. The resulting sequence depends on the choice of starting number (seed). Different seeds give different sequences of random numbers. At the same time, many of the properties of the sequence depend on chosen coefficients in the formula and do not depend on the choice of starting number.

Distribution of random numbers generated by the linear congruential algorithm makes impossible to use them in statistical algorithms that require high resolution.

One of the widely used Fibonacci generators is based on the following formula: $X_k=\begin{cases} X_{k-a}-X_{k-b}, \text{якщо } X_{k-a}\geq X_{k-b} \\ X_{k-a}-X_{k-b}+1, \text{якщо } X_{k-a}<X_{k-b} \end{cases}$, where X_k are real numbers from [0, 1), a,b are positive integers called lags. With the implementation using integers formula $X_k=X_{k-a}-X_{k-b}$ is sufficient (arithmetic overflow will occur). To use Fibonacci generator, the previous $\max\{a,b\}$ random numbers must be know. They can be generated by simple congruential generator.

Lags a and b are magic. They should not be arbitrarily chosen. Recommended lags are (a, b) = (55,24), (17,5) або (97,33).

Statistical tests give a numerical description of the sequence and can tell if the test is passed. Consider several tests from the NIST package.

Bit-frequency test. The essence of this test is to determine the relationship between zeros and ones in the entire binary sequence. The purpose is to find out whether the number of ones and zeros in a sequence are approximately the same as would be expected in the case of truly random binary sequence. This test checks if part of ones is close to 0.5. Thus, the number of zeros and ones should be about the same. If during the test calculated probability value $p < 0.01$, then this binary sequence is not truly random. Otherwise, the sequence is random. It should be noted that all subsequent tests are conducted under the condition that the test is performed.

Test for the same sequence of bits. This test is to count the total number of strings in the original sequence, where the word "string" means a continuous subsequence of equal bits. A string of bits of length k consists of k equal bits, beginning and ending with the opposite bit. The purpose of this test is to conclude whether the number of series of ones and zeros of different lengths corresponds to this number for a random sequence. In particular, it determines how quickly or slowly ones and zeros alternate each other in the original sequence. If during the test calculated probability value $p < 0{,}01$, then this binary sequence is not truly random. Otherwise, it is random.

Test for the longest sequence of ones in a block. This test determines the longest number of ones within a block of m bits. The purpose is to find out whether the length of this block corresponds to the expectations of the longest number of ones for a completely random sequence. If calculated $p < 0.01$, then the original sequence is not random. Otherwise, it is random. Note that the assumption about the same frequency of occurrence of ones and zeros (test number 1) that the same results of this test are obtained by considering the longest series of zeros. Therefore, measurements can be carried out only with ones.

5.2 Task

Develop a program that reads from the keyboard number N (1 <N <256) and a random number generator parameters and displays the generated sequence of N numbers. The program saves a graphic representation of the sequence to file accoring to variant and displays the result of a NIST test (according to variant).

Random number generators.

1 Linear congruential method.

2 Lagged Fibonacci method.

Graphical representations.

1 Histogram of elements distribution.

2 Distribution on plane (pairs of elements are processed as poing coordinates (x, y)).

3 Autocorrellation (user sets shift for copy of the sequence).

NIST tests.

1 Bit-frequency test.

2 Test for the same sequence of bits.

3 Test for the longest sequence of ones in a block.

5.3 Variants

1 1.1.1.

2 1.1.2.

3 1.1.3.

4 1.2.1.

5 1.2.2.

6 1.2.3.

7 1.3.1.

8 1.3.2.

9 1.3.3.

10 2.1.1.

11 2.1.2.

12 2.1.3.

13 2.2.1.

14 2.2.2.

15 2.2.3.

16 2.3.1.

17 2.3.2.

18 2.3.3.

6 LABORATORY WORK 6. FUNDAMENTAL ALGORITHMS ON GRAPHS AND TREES (2 PAIRS)

Objective: explore ways of representing graphs and gain skills of programming algorithms that process them.

6.1 Guidance

Data structures and algorithms to use are discussed in the lectures as well as in the proposed literature.

6.2 Task

Develop a program that reads from the keyboard numbers N, M (1 <N, M <256) being numbers of vertices and edges of the graph, a sequence of M integer pairs being the edges. The program saves the graph and performs on it an algorithm.

Representing graphs.

1 Adjacency matrix.

2 Adjacency lists.

Algorithms.

1 Depth-first search. Display the order of vertex traversal. Show time of arrival and parent in the tree traversal for each one.

2 Depth-first search. Display the order of vertex traversal. Show time of arrival and leaving and parent in the tree traversal for each one.

3 Topological sorting. Display the same data as for DFS and sorting results.

4 Determine whether a given graph is a tree or a forest.

5 Construct a spanning tree with Prim's algorithm.

6 Construct a spanning tree with Kruskal' algorithm.

6.3 Variants

1 1.1.

2 1.2.

3 1.3.

4	1.4.
5	1.5.
6	1.6.
7	2.1.
8	2.2.
9	2.3.
10	2.4.
11	2.5.
12	2.6.

7 LABORATORY WORK 7. GEOMETRICAL ALGORITHMS (1 PAIR)

Objective: get acquainted with the basic geometric algorithms.

7.1 Guidance

The primary means of many geometric algorithms is the vector product concept.

Consider vectors p_1 and p_2 . We are interested only in vectors lying in the same plane, so cross product $p_1 \times p_2$ can be understood as the signed area of a parallelogram formed by points $(0;0), p_{1,} p_{2,} p_1 + p_2 = (x_1 + x_2; y_1 + y_2)$. A definition of cross product that is convenient for computing is matrix determinant

$$p_1 \times p_2 = \det \begin{pmatrix} x_1 & x_2 \\ y_1 & y_2 \end{pmatrix} = x_1 y_2 - x_2 y_1 = -p_2 \times p_1 \quad .$$

If $p_1 \times p_2$ is positive, then the shortest turn from p_1 to p_2 is counterclockwise. If is is negative, it is clockwise.

Algorithms for constructing the convex hull are covered in lectures as well as in the proposed literature.

7.2 Task

Develop a program that reads from the keyboard number N (1 <N <256) and N pairs of real numbers - the coordinates of points on plane. It executes an algorithm on them.

7.3 Variants

1 The points belong to a polygonal chain. Show for each turn whether it is right or left.

2 The points belong to a polygonal chain. Show for each segment how many times it intersects any of the previous ones.

3 Points are the coordinates of a polygon in the order of traversal. Print the area of the polygon and report whether vertex order is counterclockwise or clockwise.

4 Construct a convex hull using Graham's algorithm.

5 Construct a convex hull using Jarvis' algorithm.

8 LABORATORY WORK 8. MATHEMATICAL FOUNDATIONS OF ALGORITHM ANALYSIS (2 PAIRS)

Objective: learn how to identify the complexity of algorithms.

8.1 Guidance

Analyzing the algorithm, we can try to find the exact number of performed operations. But in most cases it is enough to estimate the asymptotic growth time of the algorithm when its input size tends to infinity.

$$\Theta(g(n))=\{f(n):\exists \text{ positive constants } c_1, c_2 \text{ and } n_0 \text{ so that } 0\le c_1 g(n)\le f(n)\le c_2 g(n)\,\forall\, n\ge n_0\}$$

8.2 Task

Determine complexity for each of the implemented algorithm from previous works. To do this, specify the number of execution times for each line of the algorithm, depending on input data dimension. Write result using Θ .

For algorithms from works "Basic data structures" and "Fundamental algorithms on graphs and trees", determine whether the chosen data structure is optimal, and if not, then recommend those that increase the algorithm speed.

9 LABORATORY WORK 9. RECURSION (1 PAIR)

Objective: learn how to assess the complexity of recursive algorithms.

9.1 Guidance

The methods to be used are discussed in lectures as well as in the proposed literature.

Here is the master theorem of recurrence relations.

Let $a \geq 1$ and $b > 1$ be constants, $f(n)$ function, and $T(n)$ recurrently defined on nonnegative integers as

$$T(n) = aT(n/b) + f(n)$$

where we interpret n/b as $\lfloor n/b \rfloor$ or $\lceil n/b \rceil$. Then $T(n)$ has the following asymptotic estimations.

1 If $f(n) = O(n^{\log_b a - \epsilon})$ for a constant $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$.

2 If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \lg n)$.

3 If $f(n) = \Omega(n^{\log_b a + \epsilon})$ for a constant $\epsilon > 0$, and if $a f(n/b) \leq c f(n)$ for a constant $c < 1$ and all big enough n, then $T(n) = \Theta(f(n))$.

9.2 Task

For an algorithm whose complexity is described by recurrent formula $T(n) = a T(n/b) + f(n)$ (according to variant), define the complexity using the following methods:

- substitution method;
- converting to a sum;
- master method for solving recurrences;
- modeling using a program (get T(n) for different n and find a formula).

Compare obtained results.

9.3 Variants

1 $T(n) = 2T(n/2) + n$

2 $T(n) = 2T(n/2 + 35) + n$

3 $T(n)=4T(n/5)+2\mathrm{n}$

4 $T(n)=4T(n/5)+n^3$

5 $T(n)=2T(n/2)+\lg n$

6 $T(n)=3T(n/3)+n\lg^2 n$

10 LABORATORY WORK 10. DYNAMIC PROGRAMMING (2 PAIRS)

Objective: learn how to use dynamic programming and assess its complexity.

10.1 Guidance

Data structures and algorithms to use are discussed in the lectures as well as in the proposed literature.

10.2 Task

Develop a program that reads the input and solves the problem using dynamic programming. Determine the complexity of the algorithm.

10.3 Variants

1 Search route on a rectangular field, so that the sum of the numbers in the visited cells was the highest. The movement begins from the upper left cells and ends in the lower right. Each step shift to one position right or down. Input: positive integers N, M (1 <N, M <256) and a sequence of N * M of natural numbers - a rectangular box line by line. Output: the table of dynamic programming ($A(i, j)$ = maximum route to the cell (i, j)); rectangular field of route that is marked by asterisks, and the sum of the numbers in the cells of the route.

2 Find the longest common subsequence. Input: positive integers N, M (1 <N, M <256) and two sequences X and Y of positive integers of length N and M respectively. Output: the dynamic table ($A(i, j)$ = length for prefixes X_i and Y_i) and the LCS for X and Y.

3 Finding the optimal method of matrix multiplication. Input: a positive integer N (1 <N <256) being the number of matrices, integers $x_i, i=\overline{0,N}$ being the dimensions of matrices (matrix $B_i, i=\overline{1,N}$ has dimension $x_{i-1} \times x_i$). Output: the table of dynamic programming ($A(i, j)$ = smallest number of multiplications to compute the product of matrices $B_i B_{i+1} \dots B_{j-1} B_j$) and optimal placement of parentheses in the expression $A_1 A_2 \dots A_N$.

11 LABORATORY WORK 11. GREEDY ALGORITHMS (1 PAIR)

Objective: learn how to use greedy algorithms and evaluate their complexity.

11.1 Guidance

Data structures and algorithms to use are discussed in the lectures as well as in the proposed literature.

11.2 Task

Develop a program that reads the input and solves the problem using a greedy algorithm. Determine the complexity of the algorithm.

11.3 Variants

1 Solve the problem of choosing the largest number of activities for a room. Input: number of activities N (1 <N <256), N pairs of integers being the beginning and end s_i, f_i . Output: activities ordered by increasing of end time, and the number of chosen activities.

2 Compress a sequence of zeros and ones using Huffman algorithm. Input: positive integers N, M (1 <N <256, 1 <M <9) and a sequence of N groups of M zeros and ones. Output: list of codes to be used for each group and the compressed sequence.

3 Solve the fractional knapsack problem. Input: positive integers N, W (1 <N <256, 1 <W <1024) being the number of products and knapsack capacity, the sequence of N pairs p[i], v[i] being the cost and the weight of product number i. Output: a set of real numbers that indicate how many of each product must be put into a knapsack so that the total cost was maximized.

RECOMMENDED SOURCES

1 T. H. Cormen; C. E. Leiserson, R. L. Rivest, C. Stein. Introduction to Algorithms (3rd ed.). MIT Press and McGraw-Hill, 2009.

2 N. Wirth. Algorithms + Data Structures = Programs. Prentice-Hall, 1976.

3 A. V. Aho, J. E. Hopcroft, J. D. Ullman, Data Structures and Algorithms. Addison-Wesley, 1983

4 Н. Вирт. Алгоритмы и структуры данных. Новая версия для Оберона – М.: ДМК Пресс, 2010. –272 с.

5 А.В. Ахо, Д.Э.Хопкрофт, Д.Д.Ульман: Структуры данных и алгоритмы. — М. : Вильямс, 2003 – 384 с.

6 Т. Кормен, Ч. Лейзерсон, Р. Ривест, К. Штайн. Алгоритмы: построение и анализ.2-е изд./ Пер. с англ. под ред.А.Шеня. – М.: МЦНМО, 2005. – 1293 с.

7 Игошин В.И. Математическая логика и теория алгоритмов : учеб. пособие для студ. высш. учеб. заведений / В. И. Игошин. — 2-е изд., стер. — М. : Издательский центр «Академия», 2008. — 448 с.

8 Игошин В.И. Задачи и упражнения по математической логике и теории алгоритмов / В. И. Игошин. — 3-е изд., стер. — М. : Издательский центр «Академия», 2007. — 304 с.

9 Романовский И.В. Вычислительная математика и структура алгоритмов. — М.: МГУ. 2006. —112 с.