учащихся втузов. Изд. 13-е, испр.– М.: Наука – 1986. – 531 с.

2. Корн Г., Корн Т. Справочник по математике для научных работников и инженеров. – М.: Наука – 1968. – 720 с.

3. Фильчаков П.Ф. Численные и графические методы прикладной математики. – Киев: Наукова думка – 1970. – 792 с.

4. Каширский И.С., Трохименко Я.К. Обобщенная оптимизация электронных схем. – Киев: Техника – 1979. – 192 с.

Каширский И.С. ***Метод рекуррентных формул для научных исследований.*** *В статье описан новый метод рекуррентных формул для численного расчета производных высоких порядков сложных функций. Сложность функций в том, что их аргументами являются не переменные, а полиномы от переменных. Метод удобен для ПК , имеет высокую точность и не ограничивает порядок вычисляемых производных.*
Ключевые слова: *рекуррентный, производная, полином*

Каширський І.С. ***Метод рекурентних формул для наукових досліджень.*** *В статті описан новий метод рекурентних формул для чисельного розрахунку похідних вищих порядків. Складність функцій полягає в тому, що їх аргументами є не змінні, а поліноми від змінних. Метод просто програмується, має високу точність розрахунку и не обмежує порядок обчислюваних похідних.*
Ключові слова: *рекурентний, похідна, поліном*

Kashirsky I.S. ***Method of recurrent formulas for science research.*** *This paper describes new method, based on recurrent formulas, for numerical calculation of derivatives for complicated functions. Arguments of those complicated functions are not simple variables, but polinoms of variables. Method is easy for programming and has not any limits for order of derivatives.*
Key words: *recurrent, derivative, polynomial*

УДК621.372

ПРОИЗВОДИТЕЛЬНОСТЬ И ОПТИМИЗАЦИЯ ПРОГРАММ. ПОПУЛЯРНЫЕ АЛГОРИТМЫ

Реутская Ю.Ю., Новиченко А.А.

В современном мире программы все чаще становятся неотъемлемой частью различных устройств и систем (УиС), и многие программисты часто сталкиваются с ситуациями, когда вычислительной мощности оборудования недостаточно для успешной работы УиС. Оптимизация программ позволяет экономней использовать ресурсы питания во встраиваемых системах, повысить быстродействие программ обработки в реальном времени, уменьшить экономические затраты на оборудование.

В статье обосновывается роль структур данных для оптимизации алгоритма, рассматриваются несколько методов решения поставленной задачи с целью сравнения их производительности, раскрываются вопросы проектирования и оформления программного кода, что облегчит восприятие программы заказчиками (другими программистами) и предотвратит появление многих ошибок в программировании у самого автора программы.

Сложность алгоритмов будет оцениваться выражениями $O(f(n))$, где n — количество обрабатываемых данных, а $f(n)$ — функция, асимптотичес-

кое поведение которой и выражает сложность. Производительность — величина, обратная сложности.

Структуры данных и алгоритмы

Удачно выбранная структура данных (СД) позволяет упростить алгоритм и тем самым повысить производительность программы. Наибольшее внимание уделим фундаментальным СД, таким как линейный массив, связный список и бинарное дерево.

Линейный массив является одной из простейших СД. Свое название она получила из-за реализации: элементы СД располагаются в памяти один за другим линейно (рис.1).

Рис. 1. Линейный массив

В задаче доступа к элементу производительность не зависит от количества элементов и оценивается как *O(1)*, что является достоинством СД. Задача вставки нового элемента в существующий массив потребует *O(n)* дополнительной памяти и *O(n)* времени на копирование данных.

Связный список - это СД, состоящая из узлов, каждый из которых содержит как собственные данные, так и указатель на следующий узел (связь) (рис.2).

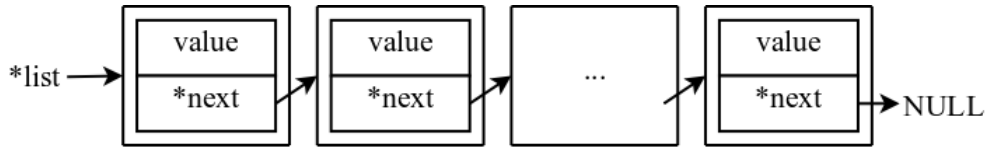

Рис. 2. Связный список

В задаче вставки узла в список потребуется изменить связь в узле, который окажется предыдущим для нового, и связь в новом узле, чтобы не нарушить целостность списка. Сложность вставки элемента имеет вид *O(1)*. Вставка узла в упорядоченный список потребует проведения поиска соответствующей позиции. В связном списке наиболее целесообразно использовать линейный поиск с *O(n)*.

Получение доступа к *i*-му узлу списка потребует перебор *i* узлов от начала списка, следуя связям, что усложняет алгоритм.

Бинарное дерево – древовидная СД, в которой каждый узел имеет не более двух потомков. Узлы, у которых нет потомков, называют листьями. Узел, у которого нет родителя, называют корнем дерева. Связь узла с его потомками называют ветвью. Дерево, у которого длина ветвей от корня до листьев отличается не больше, чем на единицу, называют сбалансированным. Пример упорядоченного бинарного дерева, составленного для фразы «now is the time for all good men to come to the aid of their party», приведен

на рис.3.

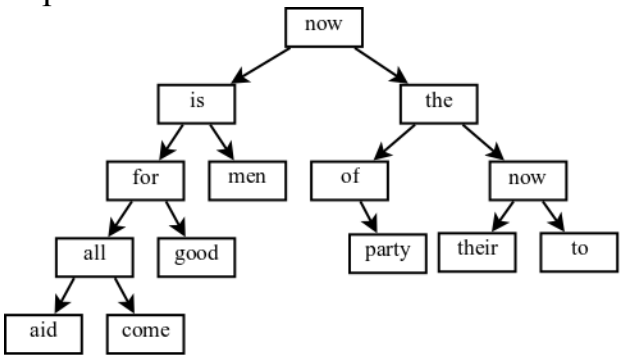

Рис. 3. Бинарное дерево

Аналогично связному списку, вставка нового узла — задача со сложностью *O(1)*. Вставка узла в упорядоченное дерево потребует поиска соответствующего узла. В сбалансированном дереве сложность поиска равна *O(log n)*. Несбалансированное дерево в предельном случае вырождается в связный список, где поиск сводится к линейному массиву с *O(n)*.

Выбор структуры данных следует проводить исходя из поставленной задачи. В случаях, где требуется обработка массивов данных постоянного размера, не требуются операции сортировки и поиска, целесообразно применять линейный массив в силу его простоты. В задачах, где требуется накопление данных, оптимальнее использовать связный список из-за его динамичности. Для поддержания данных в СД упорядоченными, целесообразным будет использование бинарного дерева или более сложных СД (двунаправленные связные списки, красно-черные деревья, ассоциативные массивы и т.д. [2]).

Наиболее востребованной задачей во многих программах является поиск данных. Его сложность в СД зависит от некоторых факторов, основными из которых являются исходные условия проведения поиска и возможности СД.

Очевидным алгоритмом поиска является *последовательный (линейный) поиск*, суть которого заключается в последовательном сравнении элементов СД с искомым. В худшем случае (когда искомый элемент в СД отсутствует) потребуется перебор всех элементов СД, что представляет собой сложность *O(n)*.

Когда данные в СД упорядочены, поиск элемента можно производить методом половинного деления, постепенно приближаясь к искомому элементу. Такой метод называется *бинарным поиском* и в худшем случае имеет сложность *O(log n)*. Однако специфика некоторых СД увеличивает сложность: в связном списке перемещение «вперед» (в сторону следующего элемента) — легкая операция, а перемещение «назад» потребует время на перебор списка с самого начала до предыдущего элемента (этот недостаток устранен в двунаправленном списке [2]).

Производительность бинарного поиска намного выше, чем у линейного, но бинарный поиск требует, чтоб данные в СД были упорядоченными (отсортированными). В дальнейшем сортировка будет рассматриваться, как метод последующего упрощения поиска: данные можно один раз упо-

рядочить, а затем многократно использовать бинарный поиск, что увеличит производительность.

Сложность методов сортировки также зависит от возможностей СД. Те из них, которые обладают не сложными операциями вставки, позволяют упорядочивать данные на этапе формирования СД. Метод *сортировки вставкой*, заключается как раз в формировании СД с упорядоченными данными. Общая сложность метода зависит от сложности поиска подходящего места вставки в выбранной СД.

С целью сравнения производительности, рассмотрим такие методы сортировки: пузырьковая сортировка, шейкер-соритровка, сортировка выбором, сортировка Шелла и быстрая сортировка. Для упрощения дальнейших изложений предположим, что сортируемые данные находятся в линейном массиве.

Метод *пузырьковой сортировки* — простой для понимания и реализации алгоритм, однако из-за его низкой производительности используется исключительно в учебных целях. Сложность алгоритма оценивают $O(n^2)$.

Алгоритм состоит в повторяющихся проходах по массиву. В каждом проходе попарно сравниваются соседние элементы и в случае, если их порядок неверный — они меняются местами (более легкие элементы «всплывают вверх»). Проходы повторяются то тех пор, пока массив не отсортирован.

Метод *шейкер-соритровки* является производным от метода пузырьковой сортировки. Его суть в том, что при движении от конца массива минимальный элемент «всплывает» на первую позицию, а максимальный сдвигается на одну позицию вправо и в том, что отсортированная часть массива при проходе не изменяется. Таким образом «границы» рабочей (неотсортированной) части массива устанавливаются в месте последнего обмена на каждой итерации, а направление прохода массива в каждой итерации меняется на противоположное. Этот метод является более производительным: в лучшем случае (отсортированный массив) производительность - $O(n)$, в худшем (отсортированный в обратном порядке массив) - $O(n^2)$.

Метод *сортировки выбором* имеет время выполнения в общем случае $O(n^2)$. Идея заключается в следующем: производится поиск наименьшего элемента в массиве, найденный элемент помещается в начало массива, а после этого аналогично обрабатывается оставшаяся часть массива.

Метод *сортировки Шелла* — более гибкий метод. Его идея, аналогично пузырьковой сортировке, заключается в просмотрах массива с целью сравнения элементов и смене их порядка при необходимости. Отличие заключается в том, что сравниваются не соседние элементы, а элементы, отстоящие друг от друга на определенном расстоянии. С каждым просмотром это расстояние уменьшается по некоторому закону. Проход, где происходит сравнение соседних элементов, является последним. Сложность такого алгоритма зависит от закона изменения расстояния между сравниваемыми

элементами. В случае использования закона, где расстояние $d_1=n/2$, $d_i = d_{i-1}/2$, $d_n = 1$, сложность алгоритма будет $O(n^2)$. В случае использования в качестве расстояния последовательности, предложенной Хиббордом: $(2^i-1)<n/2$ — сложность составит $O(n^{1.5})$.

Метод *быстрой сортировки* был предложен Чарльзом Хоаром в 1960 году и является наиболее популярным и эффективным на сегодняшний день. Алгоритм данного метода состоит из следующих шагов:

- в массиве выбирается элемент-разделитель;
- элементы, значение которых ниже разделителя, перемещаются в левую группу, а элементы, которые выше — в правую;
- левая и правая группы сортируются рекурсивно.

В случае удачного выбора разделителя, когда он является медианой и делит массив на две, одинаковые по размеру группы, сложность составляет $O(n \log n)$. Однако, если разделитель будет выбираться неудачно, в предельном случае этот метод вырождается в сортировку выбором, что имеет сложность $O(n^2)$. Поиск медианы может значительно увеличить производительность метода, но в случае недетерминированных входных данных предсказать ее невозможно. В таком случае, часто разделителем выбирают средний, последний или случайный элемент в массиве.

Поскольку при сортировке производится большое количество операций сравнения элементов, сложность вычислений по времени возрастает на временную сложность сравнения элементов. В рассмотренных случаях для простоты изложения, сортировка проводилась в линейном массиве скаляров (чисел), сложность сравнения которых $O(1)$. Сложность сравнения строк текста составляет $O(n)$, что существенно отражается на общей производительности алгоритма. Улучшить производительность в этом случае можно прибегая к различным методам, одним из которых является *поразрядная сортировка*. Для текстовых строк на первом этапе проводится сортировка по первой букве в строке (слове), а затем каждая группа слов, начинающихся с одной буквы, рекурсивно сортируется тем же методом.

Значение узкого места для повышения быстродействия программного кода и улучшения алгоритма

В случае недопустимой производительности программы первым шагом в оптимизации является поиск проблемного места, который можно производить с помощью специальных утилит (программ для выполнения поставленной задачи), в частности профилировщиков (*profilers*) — программ для оценки быстродействия участков программного кода.

Приведем пример программы сортировки больших массивов одинаковых данных, где сортировка проводится двумя методами: примитивным вариантом пузырьковой и простым вариантом быстрой сортировок.

Содержимое программы *sort.c*:

```
#include <stdio.h>
```

```
#include <stdlib.h>
#define LEN 10000
/* genarr: generate random data array of size len */
int *genarr(int len);
/* bubblesort: sort v[0]...v[n-1] into increasing order */
void bubblesort(int v[], int n);
/* quicksort: sort v[left]...v[right] into increasing order */
void quicksort(int v[], int left, int right);
/* swap: interchange v[i] and v[j] */
void swap(int v[], int i, int j);
int main()
{
    int *a1, *a2;
    int i;
    if ((a1 = genarr(LEN)) == NULL)
            return 1;
    else if ((a2 = genarr(LEN)) == NULL) {
            free(a1);
            return 1;
    }
    bubblesort(a1, LEN);
    quicksort(a2, 0, LEN-1);
    for (i = 0; i < LEN; i++)
            printf("%04d: %05d %05d\n", i, a1[i], a2[i]);
    free(a1);
    free(a2);
    return 0;
}
int *genarr(int len)
{
    int *a;
    int i;
    if ((a = malloc(len * sizeof(int))) == NULL)
            return a;
    srand(0);
    for (i = 0; i < len; i++)
            a[i] = rand() % 100000;
    return a;
}
void bubblesort(int v[], int n)
{
    int i, j;
    for (i = 0; i < n-1; i++)
            for (j = 0; j < n-1; j++)
                    if (v[j+1] < v[j])
                            swap(v, j, j+1);
}
void quicksort(int v[], int left, int right)
{
```

```
    int i, last;
    if (left >= right)
            return;
    swap(v, left, (left + right)/2);
    last = left;
    for (i = left+1; i <= right; i++)
            if (v[i] < v[left])
                    swap(v, ++last, i);
    swap(v, left, last);
    quicksort(v, left, last-1);
    quicksort(v, last+1, right);
}
void swap(int v[], int i, int j)
{
    int tmp = v[i];
    v[i] = v[j];
    v[j] = tmp;
}
```

Выполним программу и проведем ее профилирование. Приведем результаты профилирования этой программы:

Таблица 1

% Time	Cumulative Seconds	Self Seconds	Calls	Self ms/call	Total ms/call	Name
83.49	0.44	0.44	1	442.51	527.72	bubblesort
16.13	0.53	0.09	25007596	0.00	0.00	swap
0.95	0.53	0.01	1	5.03	5.31	quicksort
0.00	0.53	0.00	2	0.00	0.00	genarr

Одним узким местом в этой программе является функция *swap*, она предназначена для смены местами двух элементов в массиве. Во время сортировок *swap* вызывается 25 миллионов раз. Эта функция проста, она состоит из трех операций присвоения, и оптимизировать ее можно прибегая только к платформенно-зависимым решениям. Например, в процессорах платформы *x86* есть инструкция *XCHG*, которая меняет местами операнды и выполняется намного быстрее трех инструкций пересылки (присваивания) *MOV*. Переписав *swap* на Ассемблере можно немного выиграть в производительности, однако, если учесть операции по получению значения аргументов этой функции — выигрыш составит сотые доли процента.

Функция *bubblesort* заслуживает большего внимания, так как она вызывается всего один раз и требует 83% от общего времени работы программы. Эта функция — реализация метода пузырьковой сортировки. Производительность функции *quicksort* (метода быстрой сортировки) почти на два порядка выше *bubblesort*, что является четким доказательством асимптотических оценок сложности вычислений этих алгоритмов. У пузырьковой сортировки сложность равна $O(n^2) = 10000^2 = 10^8$, а у быстрой сложность в среднем равна $O(n \log n) = 10000 \cdot 13.3 = 13300$. Отношение сложностей со-

ставляет *750* раз. Практическая разница (*88* раз) вышла меньше теоретической из-за простоты реализации алгоритма быстрой сортировки.

В приведенном примере видно, что выбор неверного алгоритма приводит к значительным потерям производительности. Решая любую поставленную задачу, сначала следует провести обзор готовых решений и воспользоваться наиболее оптимальным алгоритмом, а при необходимости, его модификациями.

После реализации выбранных алгоритмов может потребоваться дополнительное увеличение производительности. В данном случае, обычно удается поднять производительность не больше, чем на 5%. Рассмотрим принципы высокоуровневой (не прибегающей к зависимости от платформы) оптимизации:

1. Выносите емкие операции за тело цикла, например длина строки *password* в коде:

```
len = strlen(password);
for (i = 0; i < len; i++) ...
```

будет рассчитана единожды, а в коде:

```
for (i = 0; i < strlen(password); i++) ...
```

на каждой итерации, что в случае, когда эта строка остается неизменной — является излишней тратой времени.

2. Учитывайте временные затраты на основные конструкции языка, вложенные циклы:

```
for (i = 0; i < 50; i++)
for (j = 0; j < 65536; j++) ...
```

отработают быстрее, нежели:

```
for (j = 0; j < 65536; j++)
for (i = 0; i < 50; i++) ...
```

за счет того, что во втором случае небольшой цикл вызывается большое количество раз, на что расходуется больше времени.

3. Разворачивайте небольшие циклы. Используйте циклы, когда это оправдано количеством итераций или динамичностью алгоритма.

4. Буферизация ввода/вывода повышает производительность за счет того, что данные поступают медленнее, чем выполняется программа. Структура программ, которые получают данные асинхронно, позволяет проводить другие полезные вычисления во время ожидания данных.

Оптимизацию следует проводить с осторожностью, для ее проведения необходимо иметь четкий алгоритм и работающий прототип программы. Оптимизация программы сводится к детальному анализу кода (текста) программы. Очень важным фактором успеха оптимизации является читабельность, естественность, простота и структурность программного кода.

Оформление кода и стиль программирования

«Принципы хорошего стиля программирования состоят вовсе не в наборе каких-то обязательных правил, а в здравом смысле, исходящем из

опыта. Код должен быть прост и понятен: очевидная логика, естественные выражения, использование соглашений, принятых в языке разработки, осмысленные имена, аккуратное форматирование, развернутые комментарии, а также отсутствие хитрых трюков и необычных конструкций. Логичность и связность необходимы, потому что другим будет проще читать код, написанный вами, а вам, соответственно, — их код, если все будут использовать один и тот же стиль.»[1].

Сопровождение программы, коллективное программирование, отладка и оптимизация станут намного проще и потребуют намного меньше времени для программ, код (текст) которых оформлен соответственно установленным правилам. Ниже будут рассмотрены общепринятые правила оформления программного кода на языке Си [3] и его диалектах.

Переменные следует называть информативно. Глобальные переменные нужно именовать настолько информативно, чтоб их назначение было понятно в пределах одного модуля или программы вцелом. Локальные переменные следует именовать кратко. Переменным-счетчикам подходят названия *i*, *j*, *k*, переменным-размерам подходят *m*, *n*, *len*. Пример раздражающей избыточности в именах переменных покажем на примере:

```
for (stepOfPower = 0;
 stepOfPower < powerTo;
stepOfPower++)
    result *= powerValue;
```

конструкция с правильным именованием имеет вид:

```
for (i = 0; i < n; i++)
r *= base;
```

Давайте функциям осмысленные имена, повелительные имена позволят не прибегать к чтению документации, а понимать код интуитивно, например конструкция: *now = date.getTime();*
понимается, как получение времени суток.

Избегайте неоднозначности в названиях функций, например стандартная проверка на принадлежность символа к классу цифр имеет вид:

if (isdigit(c)) ...,

где название функции подсказывает, что в случае принадлежности будет возвращена истина, в противном случае ложь. Название этой же функции:

if (checkdigit(c)) ...

даст читателю понять только то, что выполняет функция, но о возвращаемом значении имя функции ничего не сообщает. Программисту потребуется посмотреть в документацию или исходный код этой функции, что затруднит работу с кодом.

Поддерживайте структуру кода. Отсутствие структуры кода затрудняет чтение и понимание кода, а также влечет за собой ошибки в логике программ. Например, код аналога *strncpy*:

```
while (++i < n && *to++ = *from++); *to = 0; return i;
```

может привести к неверному его пониманию из-за малозаметного пустого оператора *;*. Код будет понятнее в следующем случае:

```
while (++i < n && *to++ = *from++)
    ;
*to = '\0';
return i;
```

Более сложную структуру кода следует разделять на блоки, например при помощи одной пустой строки.

Блоки, вложенные в конструкции языка (циклы, переключатели, условные операторы и т.п.) следует выделять отступами. Это подчеркнет принадлежность каждой строки кода к той или иной конструкции языка. Программный код с блоком, заключенным внутри фигурных скобок *{}* с дополнительным отступом имеет вид:

```
nsecs = tfnd = flags = 0;
while ((opt = getopt(argc, argv, "nt:")) != -1) {
    switch (opt) {
    case 'n':
            flags = 1;
            break;
    case 't':
            nsecs = atoi(optarg);
            tfnd = 1;
            break;
    default: /* '?' */
            fprintf(stderr, "Usage: %s [-t nsecs] [-n] name\n", argv[0]);
            exit(EXIT_FAILURE);
} ...
```

Избегайте загадочных чисел. Средства языка позволяют создавать именованные константы. Аналогично именованию переменных, константам следует давать имена, облегчающие чтение кода: `#define FRAME_SIZE 128`
`#define SAMPLE_RATE 8600  for (i = 0; i < FRAME_SIZE; i++) ...`

В случае отсутствия константы *FRAME_SIZE*, запись цикла оказалась бы не очевидной и потребовала бы обратиться к документации, где описан алгоритм.

Избегайте сложных выражений, разбивайте их на простые. Сложные конструкции могут быть неверно восприняты читателем. Например, сложная конструкция инициализации элемента массива: `s[i] = ++i;`
является непонятной. Если разбить это выражение на простые конструкции: `s[i] = i + 1;  i++;`,
получим понятную последовательность действий и переносимость.

Используйте естественные выражения. Двойные отрицания и выражения «от противного» способны сбить с толку читателя. В результате это заставит остановить внимание на маловажных конструкциях:

```
if (!(c < 'A') || !(c > 'Z')) ...
```

лучше записать так: `if (c >= 'A' && c <= 'Z') ...`

Рассмотренные решения позволяют более эффективно программиро-

вать для различных УиС. Оценка быстродействия основных алгоритмов и структур данных для выполнения сходных задач показывает важность выбора наиболее оптимального алгоритма для поставленной цели. А поиск критического места и оценка работы программы профилировщиком помогают программисту найти необходимый алгоритм решения задачи увеличения производительности. Высокоуровневая оптимизация помогает увеличить производительность программы после разработки и реализации алгоритмов. Следование правилам оформления программного кода поможет эффективно работать с ним и концентрироваться на идеях и алгоритмах.

Литература

1. Керниган Б., Пайк Р. Практика программирования //Пер. с англ. - СПб.:Невский Диалект, 2001. - 381 с. ISBN 5-7940-0058-9

2. Вирт Н. Алгоритмы и структуры данных //Пер. с англ. - СПб.: Невский Диалект, 2008. 352 с.. ISBN 5-7940-0065-8

3. Керниган Б., Ритчи Д. Язык программирования Си //Пер. с англ., 3-е изд. испр. - СПб.:Невский Диалект, 2001. 352 с. ISBN 5-7940-0045-7

Реутская Ю.Ю., Новиченко А.А. ***Производительность и оптимизация программ. Популярные алгоритмы.*** *Обосновывается роль структур данных для оптимизации алгоритма, рассматриваются несколько методов решения поставленной задачи с целью сравнения их производительности, а также раскрываются вопросы проектирования и оформления программного кода.*

Ключевые слова: *Структура данных, алгоритм, программный код, производительность*

Реутська Ю.Ю. Новіченко О.О. ***Продуктивність та оптимізація програм. Популярні алгоритми.*** *Обґрунтовується роль структур даних для оптимізації алгоритму, розглядаються декілька методів рішення поставленої задачі з метою порівняння їх продуктивності, а також розкриваються питання проектування та оформлення програмного коду.*

Ключові слова: *Структура даних, алгоритм, програмний код, продуктивність*

Reutskaya Y.Y., Novichenko A.A. ***The Productivity and optimization of the programs. Popular algorithms.*** *The role of structures of data is grounded for optimization of algorithm, a few methods of decision of the put task are examined with the purpose of comparison of their productivity, and also the questions of planning and registration of programmatic code open up.*

Keywords: *Structure of data, algorithm, programmatic code, productivity*