АиСД ДЗ-3

Потякин Арсений

TODO:

1. Оценить и обосновать асимптотическую верхнюю границу сложности для этой функции
2. Как можно улучшить/оптимизировать этот алгоритм? Разработайте оптимизированный алгоритм и обоснуйте его сложность
3. Упражнение 6
4. Упражнение 5.1

```
#include <iostream>
#include <vector>
#include <climits>

int long_find_max_sum(const std::vector<int>& arr,
const int k) {
 const unsigned long n = arr.size();
 int max_sum = INT_MIN;

 for (int i = 0; i <= n - k; ++i) {
  int current_sum = 0;
  for (int j = i; j < i + k; ++j) {
   current_sum += arr[j];
  }

  max_sum = std::max(max_sum, current_sum);
 }

 return max_sum;
}
```

Task 1: (Оценка сложности)

```
int long_find_max_sum(const std::vector<int>& arr,
                      const int k) {
    const unsigned long n = arr.size();
    int max_sum = INT_MIN;                          (k * c_1)(n-k+1)*c_2 + 3

    for (int i = 0; i <= n - k; ++i) {  (k * c_1)(n-k+1)*c_2
        int current_sum = 0;            c_2
        for (int j = i; j < i + k; ++j) {       k * c_1
            current_sum += arr[j];      c_1
        }

        max_sum = std::max(a: max_sum, b: current_sum);  c_2
    }
    return max_sum;
}
```

Может казаться, что циклы зависимы, но это не так. Внутренний цикл бежит от i до i + k, то есть делает ровно k операций. Пусть внутреняя операция занимает c_1 времени, тогда внутренний цикл занимает $k \cdot c_1$ времени. Рассмотрим внешний цикл. В нем есть две операции, пусть они занимают c_2 времени; сам цикл "бежит"от 0 до $n-k+1$, а значит совершает $(n-k+1) \cdot c_2$ операций. Собрав все вместе, получим $(k \cdot c_1)(n-k+1) \cdot c_2 + 3$ операций (3 операции из функции). Переводя сложность в Big O нотацию получим $O(k(n-k+1)) = O(n \cdot k - k^2 + k)$. Для больших n значение k и константы несущественны, поэтому итоговая сложность — $O(n \cdot k)$.

Task 2.1: (Оптимизация алгоритма)

Для данного алгоритма основная проблема — это вложенный цикл, из-за которого его временная сложность составляет $O(n \cdot k)$. Что предлагается сделать: вместо вложенного цикла, который совершает k операций, сделать два цикла: один делает k операций (ищет сумму первых k элементов), а потом двигать "блок"размера k на один элемент вправо, добавляя новое число и вычитая число, вышедшее за левую часть "блока". Вот реализация средствами языка C++:

```
#include <iostream>
#include <vector>

int long_find_max_sum(const std::vector<int>& arr,
const int k) {
 const unsigned long n = arr.size();

 // Инициализация блок длиной k элементов
 int current_sum = 0;
 for (int i = 0; i < k; ++i) {
  current_sum += arr[i];
 }

 int max_sum = current_sum;

 // Двигаем блок вправо
 for (int i = k; i < n; ++i) {
  current_sum += arr[i] - arr[i - k];
  max_sum = std::max(max_sum, current_sum);
 }

 return max_sum;
}
```

Task 2.2: (Обоснование сложности оптимизированного алгоритма)

```
int long_find_max_sum(const std::vector<int>& arr,
                      const int k) {
    const unsigned long n = arr.size();               c_1

    // Инициализация блок длиной k элементов
    int current_sum = 0;
    for (int i = 0; i < k; ++i) {
        current_sum += arr[i];                      k * c_2
    }

    int max_sum = current_sum;                        c_3

    // Двигаем блок вправо
    for (int i = k; i < n; ++i) {
        current_sum += arr[i] - arr[i - k];                n * c_4
        max_sum = std::max(a: max_sum, b: current_sum);
    }

    return max_sum;                 c_5
}
```

Количество выполненных операций: $c_1 + k \cdot c_2 + c_3 + n \cdot c_4 + c_5$, что представимо в Big O нотации как $O(n)$, так как k не играет роли при больших n.

Task 3: (Упражнение 6)

```
void func(int n) {
 int i = 1, s = 1;
 while (s <= n) {
  ++i;
  s += i;
 }
}
```

В каждой итерации цикла переменная s увеличивается на текущее значение i. На первой итерации s = 1 + 2 = 3, на второй s = 3 + 3 = 6, на третьей s = 6 + 4 = 10 и так далее. То есть, значение s накапливается по формуле суммы первых i натуральных чисел:

$$s = 1 + 2 + 3 + \ldots + i = \frac{i(i+1)}{2}$$

Цикл while работает до тех пор, пока s <=n $\Longrightarrow \frac{i(i+1)}{2} \approx n \Longrightarrow \frac{i^2}{2} \approx n \Longrightarrow i^2 \approx 2n \Longrightarrow i \approx \sqrt{2n} \Longrightarrow i$ растет как $\sqrt{n}$
Следовательно, количество итераций цикла while пропорционально $\sqrt{n} \Longrightarrow$ асимптотическая верхняя граница временной сложности этой функции это $O(\sqrt{n})$

Ответ: $O(\sqrt{n})$

Task 4: (Упражнение 5.1)

Алгоритм A: $T_A(n) = 0,1 \cdot n^2 \cdot \log_{10} n$
$\log_{10} n = \frac{\log_2 n}{\log_2 10} \Longrightarrow T_A(n) = 0,1 \cdot n^2 \cdot \frac{\log_2 n}{\log_2 10}$
Откидываем константы и получаем:

$$T_A(n) = O(n^2 \cdot \log n)$$

Алгоритм B: $T_B(n) = 2,5 \cdot n^2$
Откидываем константы и получаем:

$$T_B(n) = O(n^2)$$

Вывод: Алгоритм B более эффективен с точки зрения асимптотической сложности.

Начиная с какой сложности B эффективнее A?
Найдем для каких значений $T_A(n)$ эффективнее $T_B(n)$
$0,1 \cdot n^2 \cdot \log_{10} n > 2,5 \cdot n^2 \Longrightarrow 0,1 \cdot \log_{10} n > 2,5 \Longrightarrow \log_{10} n > 25 \Longrightarrow n > 10^{25}$
Таким образом, для значений n > 10^{25} алгоритм A будет менее эффективен, чем B.

Исходя из вышепроизведенных вычислений можно сделать вывод, что стоит отдать предпочтение алгоритму A. Убедимся в этом, подставив значение n в формулу $T_A(n)$ и $T_B(n)$.

$T_A(10^9) = 9 \times 10^{17}$ мс
$T_B(10^9) = 2,5 \times 10^{18}$ мс
Очевидно, что при таких входных данных алгоритм A эффективнее алгоритма B.