Лабораторна робота № 3

Оцінка часової складності алгоритмів

Мета роботи є набуття навичок дослідження часової складності алгоритмів і визначення її асимптотичних оцінок.

Методичні вказівки

При розробці програми дуже важливо провести **аналіз алгоритмів**. Аналіз полягає в тому, щоб передбачити необхідні для його виконання ресурси: час роботи, об‘єм пам'яті, пропускна здатність мережі тощо. Однак частіше за все визначаються перші два параметри. Часто вирішити одну і ту ж проблему можна за допомогою декількох алгоритмів і потрібно **вибрати найбільш ефективний** з них.

Час роботи будь-якого алгоритму залежить від набору вхідних значень, наприклад, для сортування ста чисел потрібно більше часу, ніж для сортування десяти чисел. Таким чином, в загальному випадку час роботи алгоритму збільшується зі збільшенням розміру вхідних даних, тому загальноприйнята практика – представляти час роботи програми як функцію, залежну від **кількості вхідних елементів**.

Поняття **розмір вхідних даних** залежить від задачі, що розглядається. У багатьох задачах, таких як сортування – це розмір сортованого масиву, а для перемножування двох цілих чисел – це загальна кількість розрядів, яка необхідна для представлення вхідних чисел тощо.

Час роботи алгоритму для тих чи інших вхідних даних вимірюється в кількості елементарних операцій ("кроків", які необхідно виконати). Елементарні операції, в загальному випадку, є машинно-залежними. Однак будемо виходити з точки зору, згідно з якою час виконання різних рядків алгоритму може відрізнятися, але один той самий рядок виконується за фіксований час.

Наприклад:

```
for(int i=0; i< n; ++i) {     //t1
    // певні дії              //t2
}
```

Час виконання можна розрахувати наступним чином:

$$\mathbf{T(n) = t1 \cdot (n+1) + t2 \cdot n = (t1 + t2) \cdot n + t1.}$$

Слід також звернути увагу на такий параметр як **швидкість росту (порядок росту)** часу роботи алгоритму, тому що цей параметр показує наскільки сильно збільшується час роботи алгоритму при збільшенні кількості вхідних даних.

Для аналізу ефективності алгоритмів використовується наступні критерії:

- **часова ефективність** – час, який витрачається на алгоритм для вирішення поставленого завдання;
- **просторова ефективність** – додатковий обсяг пам'яті даних, який необхідний алгоритму при вирішенні поставленого завдання.

Для аналізу часової ефективності застосовують наступні підходи (спрощення):

- час роботи алгоритму розглядається як $T(n) = Ct \cdot t(n)$, де Ct – константа, яка є машинно-залежною величиною і відповідає часу виконання елементарної операції; $t(n)$ – функціональна залежність між розміром вхідних даних та кількістю елементарних операцій;
- для дослідження функціональної залежності $t(n)$ – використовують оцінку асимптотичної складності алгоритму (асимптотичний порядок росту алгоритму).

Для того щоб можна було порівнювати між собою швидкості зростання і класифікувати їх, були введені умовні позначення:

- $O(n)$ (вимовляється як “О велике“) – асимптотична верхня межа. О - нотація;

- $\Omega(n)$ (“Омега велике“) – асимптотична нижня межа. Ω - нотація;
- $\Theta(n)$ (“Тета велике”) – асимптотична нижня та верхня межа. Θ - нотація.

Розглянемо тільки О – нотацію. Нехай t(n) – це функція (час виконання в гіршому випадку для деякого алгоритму) з вхідними даними розміру n. Функція t(n) має порядок O(g(n)) (належить до класу функцій g(n)), якщо існують такі константи $c>0$ і $n0 \geq 0$, що для всіх $n \geq n0$ виконується умова $t(n) \leq c \cdot g(n)$ (рисунок 3.1). У такому випадку говорять, що t(n) має асимптотичну верхню межу g(n). Важливо підкреслити, що це визначення вимагає існування константи c, що працює для всіх n!

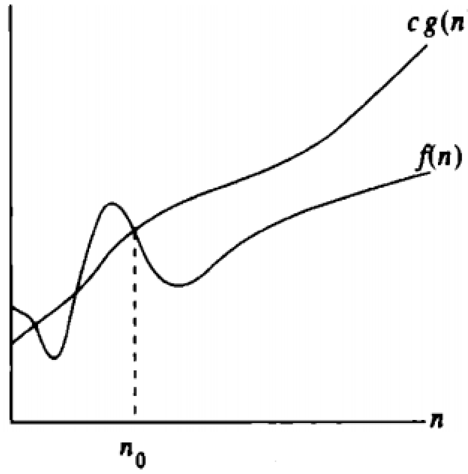

Рисунок 3.1 – Визначення О – нотації

Розглянемо приклад для визначення виразу верхньої межі складності алгоритму. Припустимо, є алгоритм, час виконання якого задається у вигляді: $t(n) = an^2 + bn + c$, де всі константи позитивні.

Слід зауважити, що для всіх $n \geq 1$ істинні умови $bn \leq bn^2$ і $c \leq cn^2$. Отже, можна записати $t(n) = an^2 + bn + c \leq an^2 + bn^2 + cn^2 = (a + b + c) \cdot n^2$. Це нерівність в точності відповідає вимозі визначення О-нотації: $t(n) \leq c_x n^2$, де $c_x = a + b + c$, отже верхня межа складності алгоритму відповідає:

$t(n) = O(n^2)$.

Найбільш часто зустрічаються наступні оцінки складності алгоритмів:

- **O(1)** – обчислювальна складність алгоритму не залежить від розміру вхідних даних (константний порядок зростання);
- **O(n)** – обчислювальна складність алгоритму лінійно зростає зі збільшенням вхідного масиву (лінійний порядок зростання);
- **O(log n)** – обчислювальна складність алгоритму зростає логарифмічно зі збільшенням розміру вхідного масиву (логарифмічний порядок зростання);
- **O(n log n)** – обчислювальна складність алгоритму зростає лінійно-логарифмічно зі збільшенням розміру вхідного масиву (лінеарітмічний порядок зростання);
- **O(n²)** – обчислювальна складність алгоритму зростає квадратично зі збільшенням вхідного масиву (квадратичний порядок зростання);
- **O(n^K)** – обчислювальна складність алгоритму зростає поліноміально зі збільшенням вхідного масиву (поліноміальний порядок зростання);

$O(2^n)$ – обчислювальна складність алгоритму зростає експоненціально зі збільшенням вхідного масиву (експоненціальний порядок зростання);

Як ці оцінки використовуються? Припустимо, що масив з 1000000 об'єктів на заданому ЕОМ сортується 10ms. Потрібно оцінити верхню межу часу виконання, якщо потрібно впорядкувати масив з 5000000 елементів. Вважаємо, що оцінка складності алгоритму визначена (див. попередній приклад) і дорівнює $O(n^2)$.

Проведемо розрахунок. Відомо, що час виконання розраховується за виразом:

$T(n) = C_t \cdot t(n)$, де $t(n) = O(n^2)$.

Отже

$T(n) \leq C_t \cdot C_x g(n) = C_t \cdot C_x \cdot n^2$.

Тоді можна записати:

$t2/t1 \leq C_t \cdot C_x \cdot n^2{}_2 / C_t \cdot C_x \cdot n^2{}_1 = n^2{}_2 / n^2{}_1$.

Звідки слідує, що:

$t2 \leq t1 \cdot n^2{}_2 / n^2{}_1 = 10 \cdot 5000000^2 / 1000000^2 = 10 \cdot 25 = 250$ ms.

Для обчислення часу виконання функції використати бібліотеку C++ <chrono> (у C# використати клас Stopwatch).

Приклад:

```
#include <iostream>
#include <chrono>

#define GETTIME std::chrono::steady_clock::now
#define CALCTIME std::chrono::duration_cast<std::chrono::nanoseconds>

int main() {
  auto begin = GETTIME();
  getchar();
  auto end = GETTIME();

  auto elapsed_ns = CALCTIME(end - begin);
  printf("The time: %lld ns\n", elapsed_ns.count());
}
```

Для того, щоб отримати більш достовірні данні, потрібно зафіксувати частоту процесору. Для цього йдемо у:
“Панель управління->Всі елементи панелі управління->Електроживлення->Зміна параметрів схеми” (рисунок 3.2) та натискаємо на посилання “Змінити додаткові параметри живлення”.

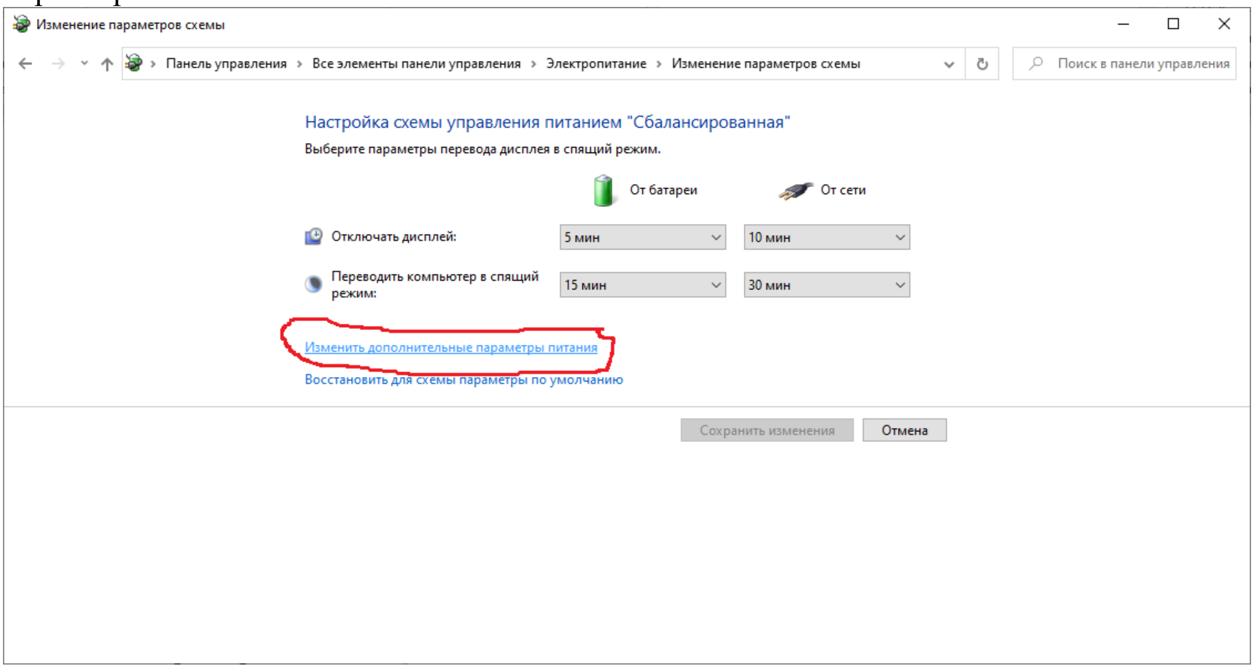

Рисунок 3.2 – Схема управління живленням

У вікні, що з'явилось необхідно змінити стан “Управління живленням процесора” (рисунок 3.3). Щоб зафіксувати частоту процесора, потрібно величини “Мінімальний стан процесора” та “Максимальний стан процесора” зробити однаковими, наприклад, 100% (максимальна продуктивність).

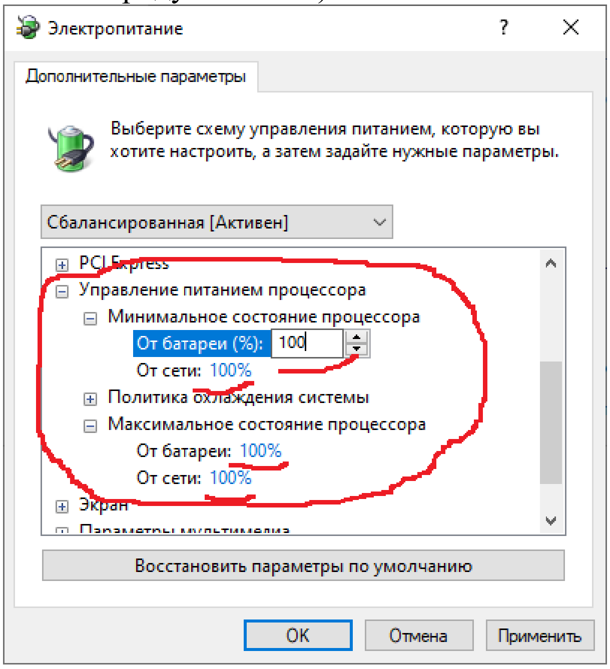

Рисунок 3.3 – Управління живленням процесора

Порядок виконання роботи

1. Написати програму для табулювання наступних функцій: $f(n)=n$; $f(n)=\log(n)$; $f(n)=n\cdot\log(n)$; $f(n)=n^2$; $f(n)=2^n$; $f(n)=n!$. Табулювання виконати на відрізку [0, 50] з кроком 1. Побудувати графіки функцій (за допомогою Excel) в одній декартовій системі координат. Значення осі ординат обмежити величиною 500.
2. Напишіть програму згідно індивідуального завдання (таблиця 3.1 та таблиця 3.2). Виміряти час виконання функцій та побудувати графіки за допомогою Excel. Провести аналіз і оцінку часової складності алгоритмів. Порівняти практично отримані результати з оцінкою часової складності алгоритмів.

Таблиця 3.1 – Варіанти завдань

№ варіанту	**Номери задач**
1	1, 3
2	2, 5
3	4, 8
4	9, 6
5	7, 10
6	1, 5
7	2, 8
8	4, 10
9	9, 3
10	7, 6
11	1, 8
12	2, 10
13	4, 3
14	9, 5
15	7, 6

Таблиця 3.2 – Задачі для виконання

№ задачі	Опис завдання
1	Дано два вхідних цілих числа **a** і **b**, де $0 \le a \le 10$, $0 \le b < 20$. Реалізувати функцію піднесення заданого числа **a** у ступінь **b**.
2	Дано вхідне ціле число **a**, де $0 \le a \le 20$. Реалізувати функцію за допомогою рекурсії знаходження факторіалу числа **a**.
3	Дан масив десяткових цифр. Обсяг масиву **m**<20. Реалізувати функцію, яка повертає найбільше можливе число з даних цифр. Цифри згенерувати генератором випадкових чисел.
4	Реалізувати функцію обчислення **n**-го числа Фібоначчі, де $n \le 90$, n – ціле число. Обраховуються числа Фібоначчі за виразом: $F_0=0$, $F_1=1$, $F_n=F_{n-1}+F_{n-2}$, де $n \ge 2$.
5	Дан масив цифр вісімкової системи числення. Обсяг масиву **m**≤20. Реалізувати функцію, яка повертає найбільше можливе число з даних цифр. Цифри згенерувати генератором випадкових чисел.
6	Реалізувати функцію обчислення **n**-го числа Фібоначчі за допомогою рекурсії, де $n \le 40$, n – ціле число. Обраховуються числа Фібоначчі за виразом: $F_0=0$, $F_1=1$, $F_n=F_{n-1}+F_{n-2}$, де **n**≥2.
7	Дан масив цифр двійкової системи числення. Обсяг масиву **m**≤40. Реалізувати функцію, яка повертає найбільше можливе число з даних цифр. Цифри згенерувати генератором випадкових чисел.
8	Дан масив чисел типу float. Обсяг масиву **m**≤1000. Реалізувати функцію бульбашкового сортування масиву в порядку спадання чисел.
9	Дано ціле число **a** з розрядністю **b**, де $8 \le b \le 64$. Реалізувати функцію, яка повертає двійкове представлення числа **a** у вигляді строки.
10	Дан масив чисел типу int. Обсяг масиву **m**<1024. Реалізувати функцію бульбашкового сортування масиву в порядку спадання чисел.

Список літератури

1. Вирт Н. Алгоритмы и структуры данных. 2001
2. Майкл Мейн, Уолтер Савитч. Структуры данных и другие объекты в C++. — 2-е изд. — М.: Вильямс, 2002
3. Седжвик Р. Фундаментальные алгоритмы на C++. Части 1-4. Анализ. Структуры данных. Сортировка. Поиск. 2001
4. Седжвик Р. Фундаментальные алгоритмы на C++. Часть 5. Алгоритмы на графах.
5. Топп У., Форд У. Структуры данных в C++. 1999
6. Ахо Альфред В., Хопкрофт Джон Э., Ульман Джеффри Д. Структуры данных и алгоритмы. 2000
7. Хэзфилд Р., Кирби Л. Искусство программирования на C. Фундаментальные алгоритмы, структуры данных и примеры приложений. 2001
8. Сибуя М., Ямамото Т. Алгоритмы обработки данных. – М: Мир, 1986
9. Лэгсам Й, Огенстайн М. Структуры данных для персональных ЭВМ – М: Мир, 1989
10. Кнут Д. Искусство программирования для ЭВМ. Том 1: Основные алгоритмы. : Пер. с англ. -М.: Мир,1976.
11. Кнут Д. Искусство программирования для ЭВМ. Том 2 : Получисленные алгоритмы : Пер. с англ. -М.: Мир,1978.