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. O - нотація;

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

Розглянемо тільки O – 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 ≥ 0, що для всіх n ≥ n0 виконується умова t(n) ≤ c·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!

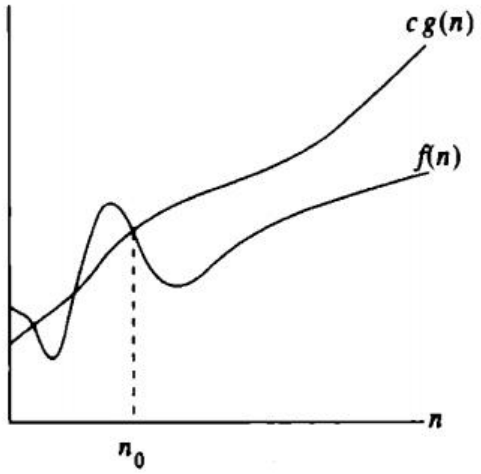

Рисунок 3.1 – Визначення O – нотації

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 O-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);
- **$O(n^2)$** – 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”.

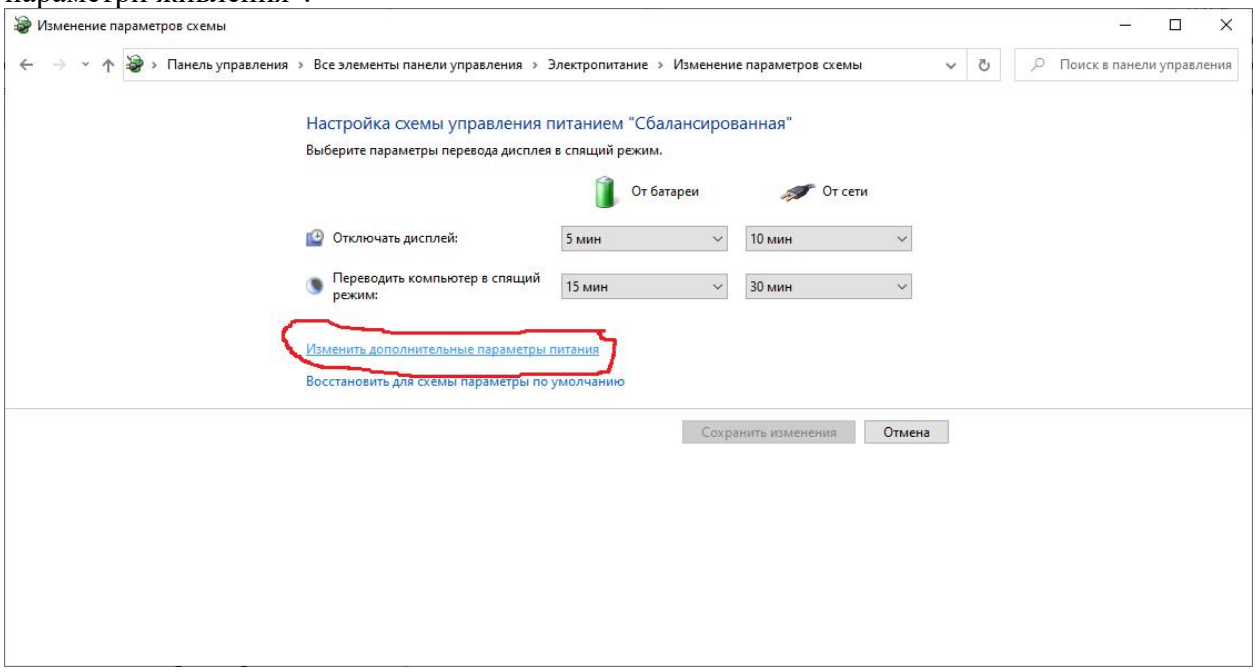

Рисунок 3.2 – Схема управління живленням

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).

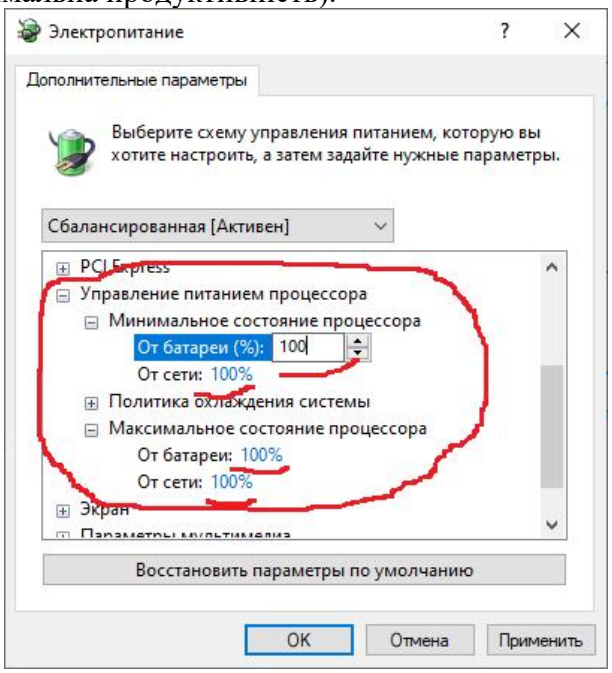

Рисунок 3.3 – Управління живленням процесора

Порядок виконання роботи

1. Написати програму для табулювання наступних функцій: 1) f(n)=n; 2) f(n)=log(n); 3) f(n)=$\sqrt{n}$; 4) f(n)=n·log(n); 5) f(n)=n^2; 6) f(n)=2^n; 7) 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.

Формат вхідних даних:

Єдиний рядок в якому вказується номер функції.

Формат вихідних даних:

Програма повинна вивести розраховані перші 5 значень з точністю шість знаків після коми (повністю всі значення для побудови графіка повинні бути виведені у файл).

2. Побудувати графіки залежності часу виконання функцій від кількості даних за допомогою Excel. Провести аналіз і оцінку часової складності алгоритмів. Порівняти практично отримані результати з оцінкою часової складності алгоритмів.

1) У вас є **a** кубиків. Ви будуєте з них сходи так, щоб у першому ряду був 1 кубик, у другому − 2, у третьому − 3 і так далі. Напишіть функцію, яка приймає на вхід **a** і визначає, скільки повних рядів сходів можна побудувати.

2) Уявіть, що у вас є весь набір гир вагою від 1 до **a** грамів. Напишіть функцію, яка підрахує, скількома способами можна вибрати дві різні гирі так, щоб їхня сумарна вага була менше **a+5** грамів.

3) Реалізувати функцію обчислення **a**-го числа Фібоначчі за допомогою рекурсії, де a≤40, a – ціле число. Обраховуються числа Фібоначчі за виразом: F_0=0, F_1=1, F_a=F_{a-1}+F_{a-2}.

Формат вхідних даних:

Єдиний рядок вхідних даних містить 3 цілих числа, розділених пробілами:

N a y

де:

N – номер завдання (від 1 до 3);

a – вхідне значення завдання;

y – приймає 0 або 1 (1 – вивести данні для побудови графіка у файл, 0 – ні).

Формат вихідних даних:

Програма повинна вивести отриманий результат (данні для побудови графіка повинні бути виведені у файл).

Мова програмування тільки C#.