**Лекція 6 Складність алгоритмів**

Складність алгоритмів зазвичай оцінюють **за часом виконання** або **за об’ємом використаної пам’яті**.

В обох випадках складність залежить від розміру вхідних даних масиву: масив зі 100 елементів буде оброблений швидше ніж аналогічний з 1000. При цьому точний час мало кого цікавить, так як він залежить від процесора, типу даних, мови програмування та інших параметрів. Важливою є **асимптотична складність**, тобто складність при умові прямування розміру вхідних даних до нескінченності.

Припустимо, деякому алгоритму треба виконати ***4n3+7n*** умовних операцій, щоб обробити ***п*** елементів вхідних даних. При збільшенні ***п*** на підсумковий час роботи алгоритму значно більше буде впливати піднесення ***п*** до кубу, ніж множення на 4 або додавання **7*п*** . Тоді говорять, що часова складність цього алгоритму дорівнює ***О(n3),*** тобто **залежить від розміру вхідних даних кубічно.**

Використання букви ***О*** (це називають **О-нотацією**) прийшло з математики, де її використовують для порівняння асимптотичного поводження функцій. Формально ***O(f(n))*** означає, що час роботи алгоритму (або об’єм використаної пам’яті) зростає в залежності від об’єму вхідних даних не швидше, ніж деяка константа, помножена на ***f(n).***

**Приклади**

**O(n)** — лінійна складність

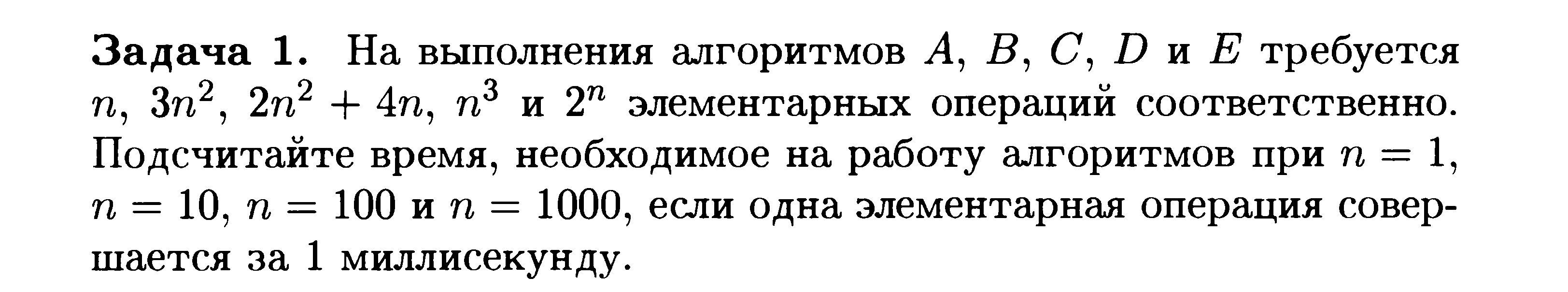
Таку складність мають, наприклад**, алгоритм пошуку найбільшого елемента в невідсортованому масиві.** Нам треба буде пройтись по всім ***n*** елементам масива, щоб зрозуміти, який з них є максимальним.

**O(log n)** — логарифмічна складність

Приклад — **бінарний пошук**. Якщо масив **відсортований**, ми можемо перевірити, чи є в ньому деяке конкретне значення, методом ділення навпіл. В підсумку перевіримо ***log n*** елементів.

**O(n2)** — квадратична складність

Таку складність має, наприклад, **алгоритм сортування вставками**. В канонічній реалізації він представляє собою два вкладених цикла: один, щоб проходити по всьому масиву, а другий, щоб знаходити місце черговому елементу в уже відсортованій частині. Таким чином, кількість операцій буде залежати від розміру масива як ***n\*n***, тобто ***n2***.

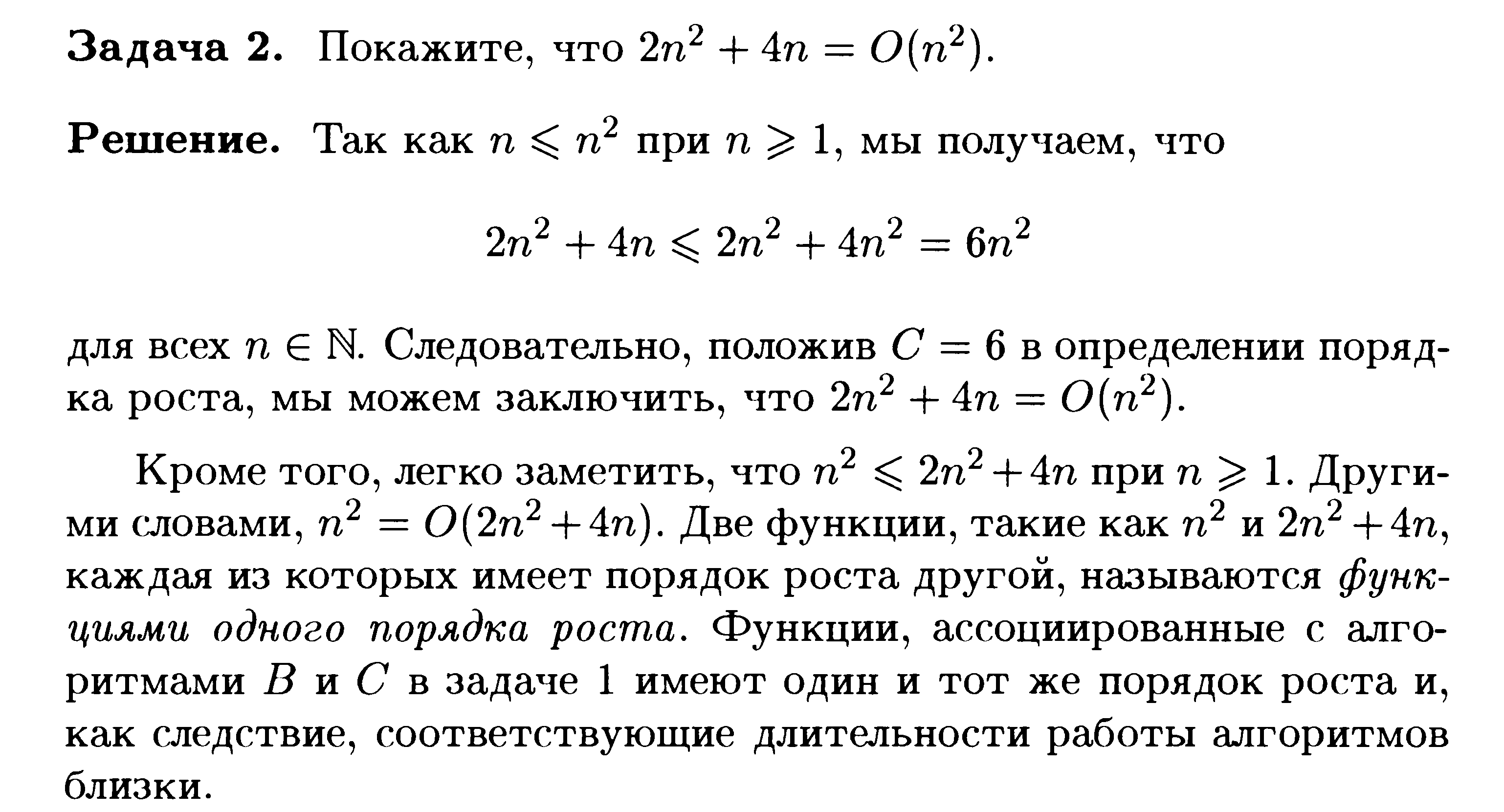




Як видно з таблиці, існує якісна різниця між формулами, які включають в себе степені ***п*** (**поліноміальні функції**), й тими, в яких ***п*** є показником (**експоненціальні функції**). Поліноміальні функції відрізняються одна від одної величиною показника старшого степеня. Функції з однаковими показниками старших степенів, то робочий час відповідних алгоритмів «майже однаковий», наприклад у алгоритмів ***В*** і ***С***.

**Означення.** Припустимо, що функції **f(n)** та **g(n)** вимірюють ефективність двох алгоритмів, зазвичай їх називають функціями часової складності. Говорять, що ***порядок зростання функції*** **f(n)** не більший, ніж у **g(n),** якщо знайдеться така додатна константа С, що  для всіх достатньо великих значень ***п***. Цей факт прийнято позначати як .

Зауважимо, що  означає не одну, а цілий клас функцій, які зростають не швидше, ніж . Тому знак «=» тут треба розуміти як знак .



Ми можемо визначити ієрархічну структуру на множині функцій, кожна з яких має більший порядок зростання ніж попередня. Один з прикладів такої ієрархії має вигляд



Цю послідовність можна деталізувати, вставляючи в неї нові функції. Покажемо, наприклад, що функцію  можна вставити між та , а функцію  - між  та .

Дійсно,

у нас 

, так як 

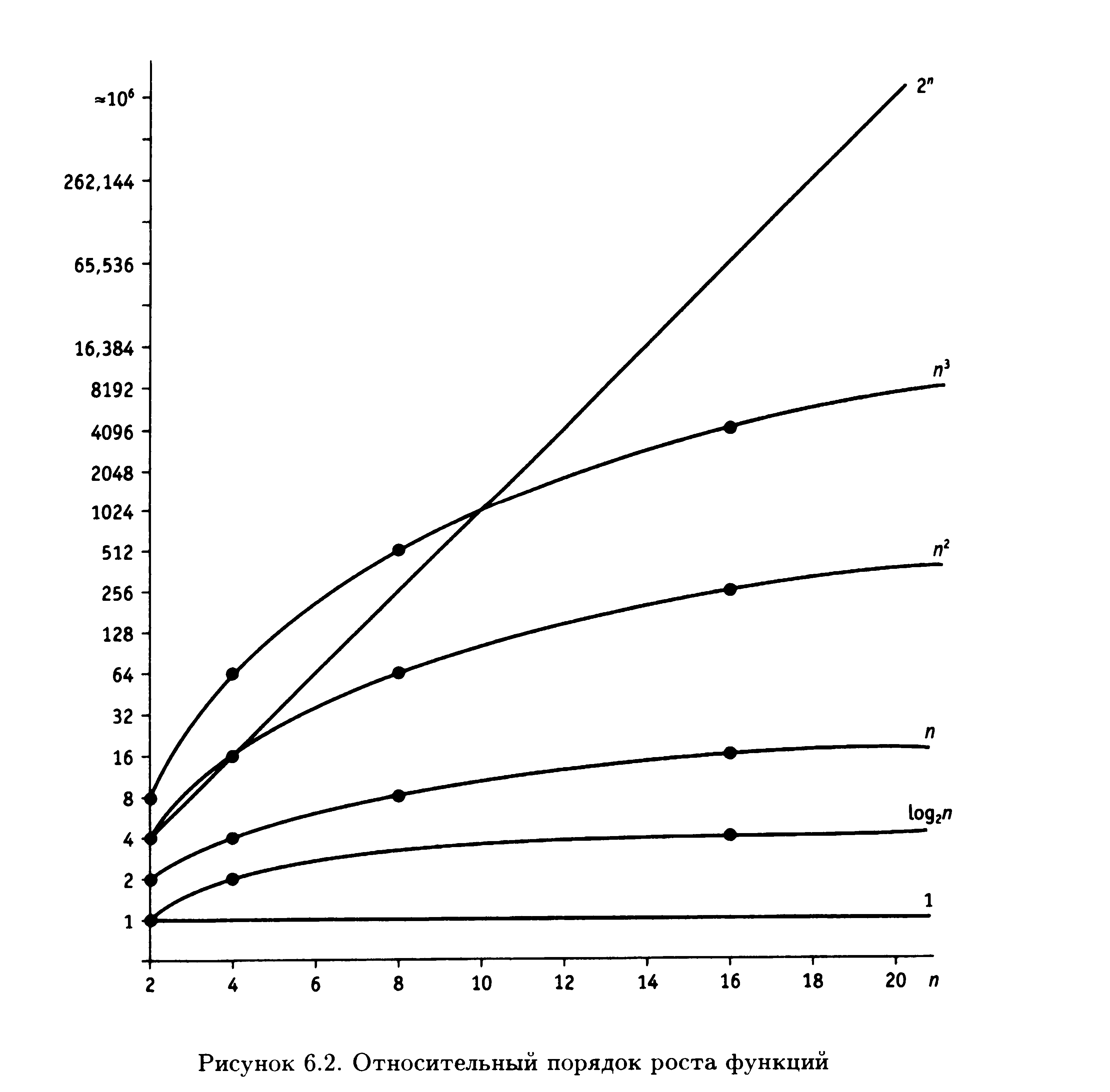
, так як 

**Аналогично**,

, так як 

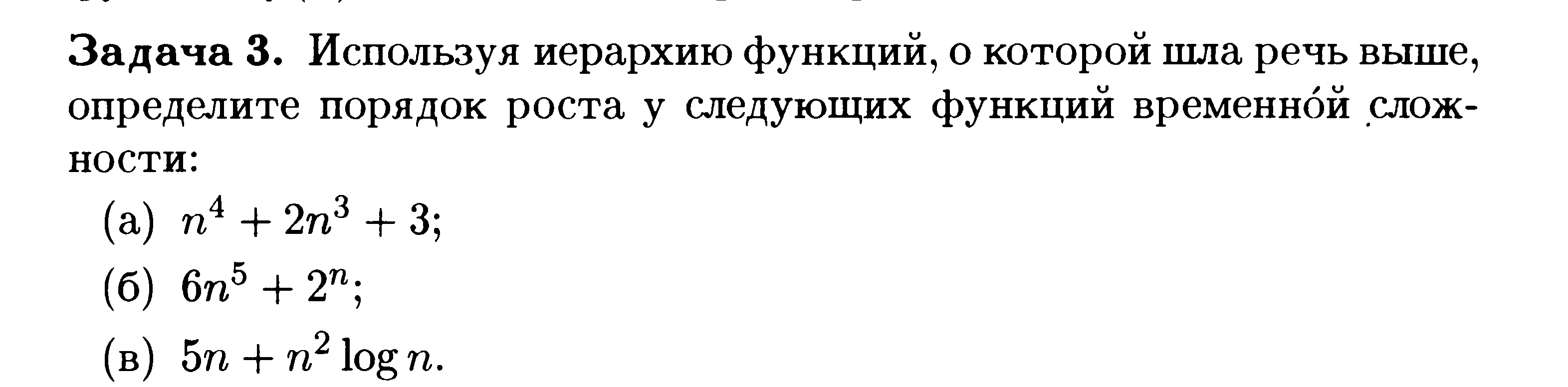
, так як .

В цій ієрархії важливо те, що просуваючись від її лівого краю до правого, ми зустрічаємо функції все більшого порядку зростання. Тобто, чим правіше в цій послідовності стоїть функція, тим швидше зростають її значення в порівнянні з ростом ***п***. Це демонструють графіки функцій

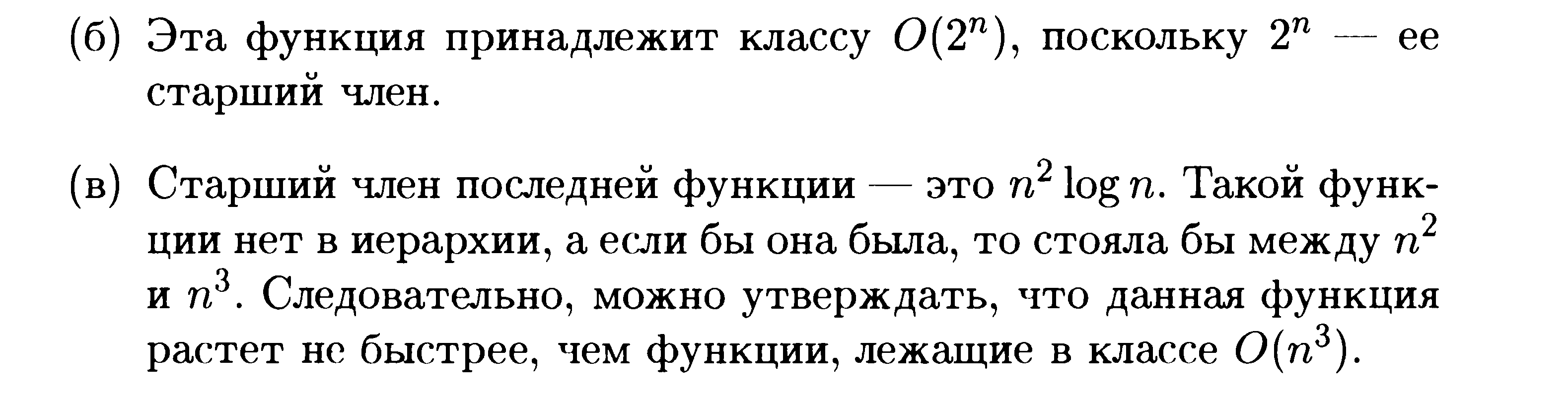


Для більш складних формул функції ми виділяємо в формулі доданок з найвищим зростанням (так званий старший член) і співставляємо цій функції відповідну функцію в ієрархії.

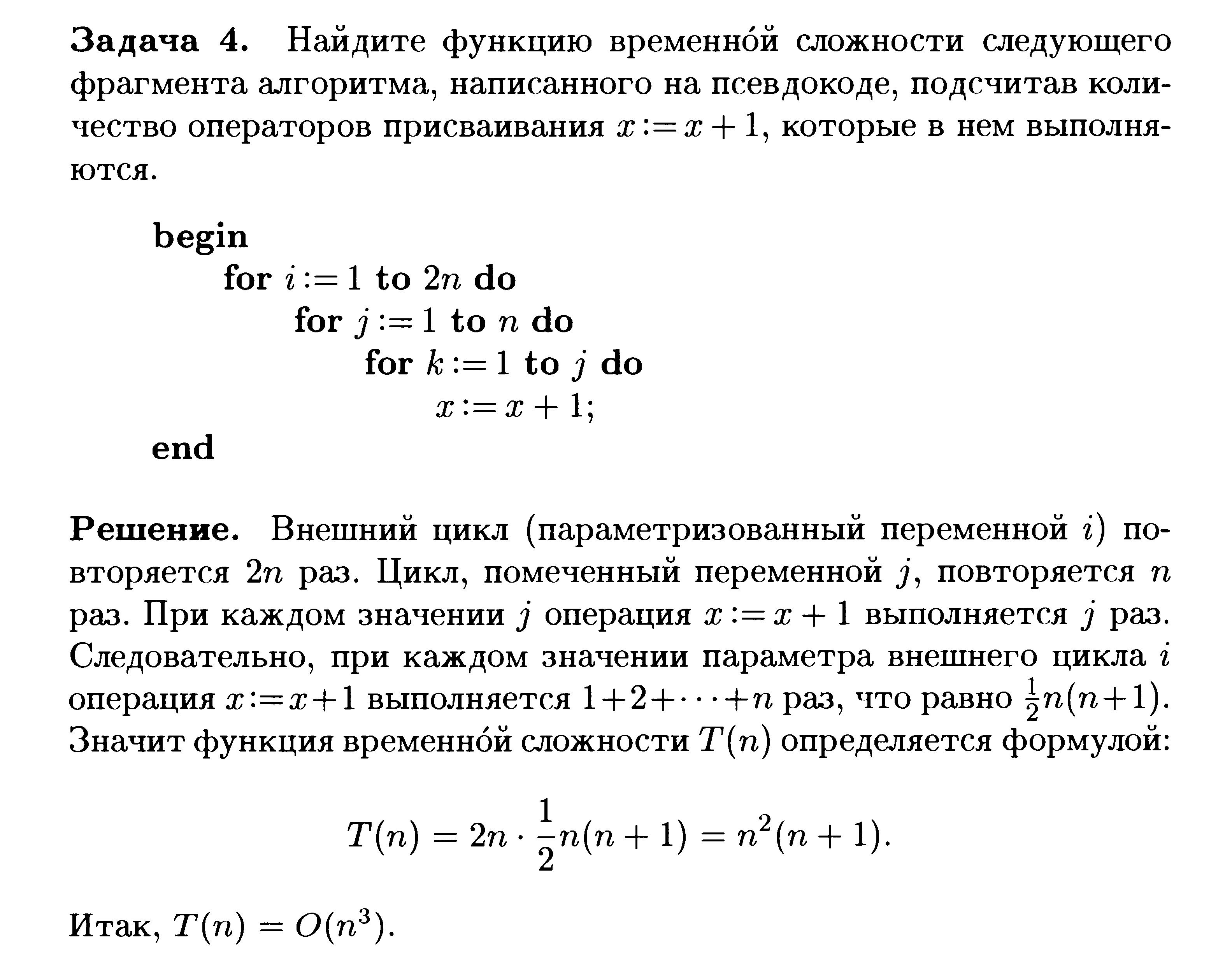
Наприклад, для функції  маємо , ,. Старшим є доданок . Отже, , або, задана функція має той же порядок зростання, що і функція .







Зауважимо, що при обчисленні часової складності будь-якого алгоритму необхідно з’ясувати, що слід взяти в якості параметру ***п*** і які елементарні операції слід враховувати.



**Задача 5 Показать, что бинарное (двоичное) возведение в степень — это приём, позволяющий возводить любое число в n-ую степень за O(\log n)умножений.**

Действительно, от степени n мы переходим, если она чётна, к n/2, а иначе — к n-1. Понятно, что всего будет не более 2 \log n переходов, прежде чем мы придём к n = 0.

 , .

Таким образом, мы получили алгоритм, работающий за O (\log n)умножений.

