Лекція № 6

ОЦІНКА СКЛАДНОСТІ АЛГОРИТМІВ

Основними показниками складності алгоритму є час, необхідний для вирішення завдання, і об'єм необхідної пам'яті.

Розмір входу - деяке число, що характеризує об'єм даних для класу завдань. *Складність алгоритму* це функція розміру входу.

Складність алгоритму може бути різною при одному і тому ж розмірі входу, але різних вхідних даних. Тому розглядають поняття часової складності алгоритму *в гіршому, середньому* або *кращому* випадку. Зазвичай оцінюють складність у гіршому випадку.

Визначення. Часова складність алгоритму (у гіршому випадку) — це функція від розміру вхідних даних, яка дорівнює максимальній кількості елементарних операцій, що виконуються алгоритмом для вирішення екземпляра завдання вказаного розміру.

Аналогічно поняттю *часової складності у гіршому випадку* визначається поняття ***часової складності алгоритму в кращому випадку***.

Часова складність алгоритму *в середньому* випадку – *це середній час роботи алгоритму*, тобто математичне очікування часу роботи алгоритму.

Максимальна часова складність відповідає випадку, коли вибрані вхідні дані породжують найбільш довге виконання алгоритму.

Слід зазначити, що існує ***практична складність*** - точна міра часу обчислень і об'єму пам'яті для *конкретної моделі* обчислювальної машини, і ***теоретична складність***, яка більш незалежна від практичних умов виконання алгоритму і дає порядок величини.

ПРИКЛАД 7.1. Розглянемо алгоритм, що визначає для будь-якого цілого n>1 його *найбільший дільник* max_d, відмінний від самого n (якщо n - просте, то результат - одиниця).

```
i:=n-1;
while (n mod i <> 0) do i:=i-1;
   max_d:=i;
```

Цикл обов'язково завершується, оскільки n(mod 1) = 0.

Оцінимо час виконання алгоритму.

Алгоритм має вигляд:

I_1;while (condition) do I_2;I_3;

Хай t_1, t_2, t_3 - часи виконання операторів I_1, I_2 і I_3 відповідно. Припустимо, що цикл while представлений в машині звичайним способом за допомогою одного умовного і одного безумовного переходу, що

вимагають відповідно часів t_y і $t_б$. Тоді час виконання визначається як

$$t= t_1+t_3+m*(t_y+t_2+t_б)+ t_y \qquad (7.1)$$

де m - число повторень циклу.

Введемо поозначення $P=t_1+t_3+t_y$; $Q=t_y+t_2+t_б$ и підставимо їх в (7.1).

Отримаємо $t=P+Q*m$.

“Гірший випадок”: вхідний параметр n - число просте.

Тоді число виконань циклу $m_A=n-2$.

Час виконання алгоритму визначається у вигляді:

$P+Q*(n-2) = Q*n+(P-2*Q)$,

Деталізація *практичної складності* веде до обчислення постійних P, Q, визначуваних конкретною ЕОМ і залежних від часу виконання її команд.

Теоретична складність дозволяє порівняти два алгоритми. Коли **n** велике

$Q_An+(P_A-2Q_A)\sim Q_An$

Таким чином, у гіршому випадку часова складність алгоритму пропорційна n (позначення: $t_A=O(n)$).

По аналогії з часовою складністю, визначають ***просторову складність алгоритму***, тільки тут говорять не про кількість елементарних операцій, а про об'єм використовуваної пам'яті.

Визначення**. **Ємкісна (або просторова) складність у гіршому випадку – функція розміру входу, дорівнює максимальній кількості елементів пам'яті, до яких було звернення при рішенні завдань даного розміру. Для просторової складності також розрізняють максимальну і середню просторову складність.

ПРИКЛАД 7.2. Хай часова складність деякого завдання має порядок 2^n. Відомо, що при n=10 завдання вирішується за одну хвилину. Збільшимо n до 100. Час виконання зросте в $2^{100}/2^{10}=2^{90}$ разів. Враховуючи, що 1 рік складається з 525600 хвилин, легко підрахувати, що завдання вирішуватиметься більш ніж 1020 років.

Хай для того ж завдання вдалося знайти алгоритм з часовою оцінкою n^5. В цьому випадку при n=100 час рішення задачі зросте в $100^5/10^5=10^5$ разів, тобто складе “всього лише” два з невеликим місяця.

Визначення. *Трудомісткість алгоритму.* Під трудомісткістю алгоритму для даного конкретного входу – $F_a(N)$, розумітимемо кількість

«елементарних» операцій, здійснюваних алгоритмом для вирішення конкретної проблеми в даній формальній системі.

Асимптотичний аналіз функцій трудомісткості алгоритму[1]

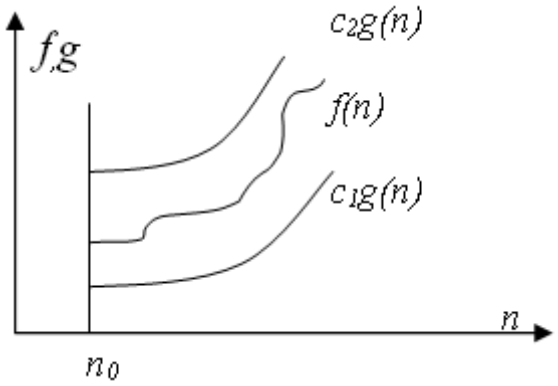
При аналізі поведінки функції трудомісткості алгоритму часто використовують прийняті в математиці асимптотичні позначення, що дозволяють показати швидкість росту функції, маскуючи при цьому конкретні коефіцієнти.

Така оцінка функції трудомісткості алгоритму називається *складністю алгоритму* і дозволяє визначити переваги у використанні того або іншого алгоритму для великих значень розмірності початкових даних.

У асимптотичному аналізі прийняті наступні позначення:

1. Оцінка Θ (тетта)

Хай f(n) і g(n) – позитивні функції позитивного аргументу, n>=1 (кількість об'єктів на вході і кількість операцій – позитивні числа), тоді:



f(n) = Θ(g(n)), якщо існують позитивні c_1, c_2, n_0, такі, що:

$c_1*g(n)=<f(n)=<c_2*g(n)$,

при $n>n_0$

Зазвичай говорять, що при цьому функція g(n) є асимптотично точною оцінкою функції f(n), оскільки за визначенням функція f(n) не відрізняється від функції g(n) з точністю до постійного множника.

Відзначимо, що з f(n) = Θ(g(n)) витікає, що g(n) = (f(n)).

Приклади:

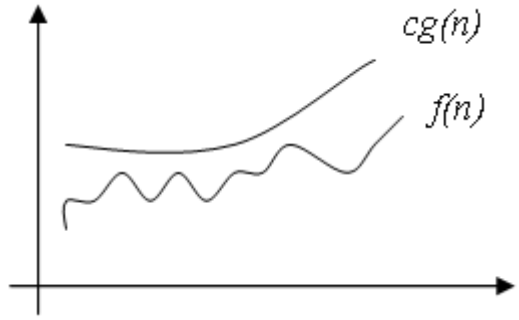
1) $f(N)=4N^2+N*\ln(N)+174$ – f(n)= $\Theta(N^2)$;

2) f(N)=Θ(1) – запис означає, що f(n) або рівна константі, не рівній нулю, або f(N) обмежена константою на ∞: f(N)= 7+1/N = Θ(1).

2. Оцінка O (О-велике)

На відміну від оцінки Θ, оцінка **O** вимагає тільки, щоб функція f(n) не перевищувала g(n) починаючи з $n>n_0$, з точністю до постійного множника:

[1] Асимптотичне позначення **O** сходить до підручника Бахмана по теорії простих чисел (Bachman, 1892), позначення **Θ, Ω,** введені Д. Кнутом (Donald Knuth).



$\exists c>0,\ n_0>0: 0=<f(n)=<c*g(n),\ \forall n>n_0$

Взагалі, запис **O**(g(n)) позначає клас функцій, таких, що всі вони ростуть не швидше, ніж функція g(n) з точністю до постійного множника, тому іноді говорять, що g(n) мажорує функцію f(n).

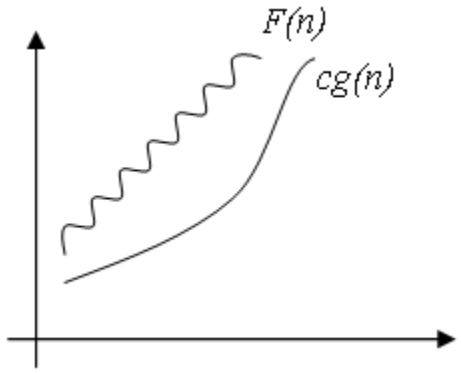
Наприклад, для всіх функцій:

f(n)=1/n, f(n)= 12, f(n)=3*n+17, f(n)=n*Ln(n), f(n)=6*n^2 +24*n+77 буде справедлива оцінка O(n^2)

Указуючи оцінку O є сенс указувати найбільш «близьку» мажоруючу функцію, оскільки, наприклад для f(n)= n^2 справедлива оцінка O(n^2), проте вона не має практичного сенсу.

3. Оцінка Ω (Омега)

На відміну від оцінки **O**, оцінка Ω є оцінкою знизу – тобто визначає клас функцій, які ростуть не повільніше, ніж g(n) з точністю до постійного множника:



$\exists c>0,\ n_0>0: 0=<c*g(n)=<f(n)$

Наприклад, запис Ω(n*Ln(n)) позначає клас функцій, які ростуть не повільніше, ніж g(n)= n*Ln(n), в цей клас потрапляють всі поліноми із ступенем більше одиниці, так само як і всі степеневі функції з основою більше одиниці.

Відзначимо, що не завжди для пари функцій справедливе одне з асимптотичних співвідношень, наприклад для $f(n)=n^{1+\sin(n)}$ і $g(n)=n$ не виконується жодне з асимптотичних співвідношень.

У асимптотичному аналізі алгоритмів розроблені спеціальні методи отримання асимптотичних оцінок, особливо для класу рекурсивних алгоритмів. Очевидно, що Θ оцінка є більш переважною, ніж оцінка O. Знання асимптотики поведінки функції трудомісткості алгоритму, тобто його складності, дає можливість робити прогнози по вибору раціональнішого з погляду трудомісткості алгоритму для великих розмірностей початкових даних.

Трудомісткість алгоритмів і часові оцінки

1. Елементарні операції в мові запису алгоритмів

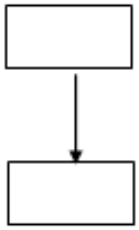
Для отримання функції трудомісткості алгоритму, представленого у формальній системі деякої абстрактної машини необхідно уточнити поняття «елементарних» операцій, співвіднесених з мовою високого рівня. Як такі «елементарні» операції пропонується використовувати наступні:

1. Просте присвоєння: a = b;
2. Одновимірна індексація a[i]: (адреса (a)+i*довжина елементу);
3. Арифметичні операції: (*, /, - +);
4. Операції порівняння: a < b;
5. Логічні операції {OR, AND, NOT}.

Виключимо команду переходу за адресою, вважаючи її пов'язаною з операцією порівняння в конструкції розгалуження.

Після введення елементарних операцій аналіз трудомісткості основних алгоритмічних конструкцій в загальному вигляді зводиться до наступних положень:

1. *Конструкція «Слідування»*



Трудомісткість конструкції є сумою трудомісткостей блоків, розташованих один за одним.

$F_{\text{слідування}}=f_1+..+fk$, де k – кількість блоків.

2. *Конструкція «Розгалуження»*

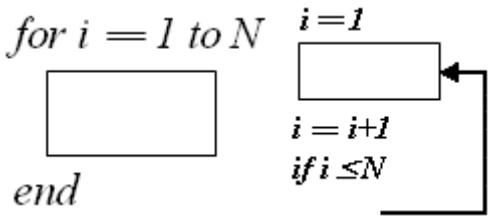
if(I) then

else

Загальна трудомісткість конструкції «Розгалуження» вимагає аналізу вірогідності виконання переходів на блоки «*Then*» і «*Else*» і визначається як:

$F_{розгалуження}=f_{then}*p+f_{else}*(1-p)$

3. *Конструкція «Цикл»*



Після зведення конструкції до елементарних операцій її трудомісткість визначається як:

$F_{цикл} = 1+3*N+N*f_{тіло\ циклу}$

Приклади аналізу простих алгоритмів

Приклад 1 *Задача підсумовування елементів квадратної матриці*

```
SUMM (A, N; Sum)
Sum =0
For i = 1 to N
For j = 1 to N
Sum = Sum + A[i,j]
end for
Return (Sum)
End
```

Алгоритм виконує однакову кількість операцій при фіксованому значенні N, і отже є кількісно-залежним. Застосування методики аналізу конструкції «Цикл» дає:

$F_a(n)=1+1+ N*(3+1+ N*(3+4))=7N^2+4* N +2 = \Theta(N^2)$

Відмітимо, що під N розуміється лінійна розмірність матриці, тоді як на вхід алгоритму подається N^2 значень.

Приклад 2 *Задача пошуку максимуму в масиві*

```
MAXS (S,N; Max)
Max = S[1]
For i = 2 to N
if Max < S[i]
then Max = S[i]
end for
return Max
End
```

Даний алгоритм є кількісно-параметричним, тому для фіксованої розмірності початкових даних необхідно проводити аналіз для гіршого, кращого і середнього випадку.

А). Гірший випадок

Максимальна кількість переприсвоювань максимуму (на кожному проході циклу) буде в тому випадку, якщо елементи масиву відсортовані за збільшенням. Трудомісткість алгоритму в цьому випадку становить:

$F^{\wedge}_a(n)=1+1+1+ (N-1)*(3+2+2)=7*N - 4 = \Theta(n)$.

Б) Кращий випадок

Мінімальна кількість переприсвоювань максимуму (жодного на кожному проході циклу) буде в тому випадку, якщо максимальний елемент розташований на першому місці в масиві. Трудомісткість алгоритму в цьому випадку рівна:

$F^{\vee}_a(n)=1+1+1+ (N-1)*(3+2)=5*N - 2 = \Theta(n)$.

В) Середній випадок

Алгоритм пошуку максимуму послідовно перебирає елементи масиву, порівнюючи поточний елемент масиву з поточним значенням максимуму. На черговому кроці, коли є видимим k-ий елемент масиву, переприсвоювання максимуму відбудеться, якщо в підмасиві з перших k елементів максимальним елементом є останній. Очевидно, що у разі рівномірного розподілу початкових даних вірогідність того, що максимальний з k елементів розташований в певній (останній) позиції дорівнює 1/k. Тоді в масиві з n елементів загальна кількість операцій переприсвоювання максимуму визначається як:

$$\sum_{i=1}^{N}\frac{1}{i} = H_N \approx Ln(N) + \gamma, \gamma \approx 0{,}57$$

Величина H_N називається N-bм гармонійним числом. Таким чином, точне значення (математичне очікування) середньої кількості операцій присвоєння в алгоритмі пошуку максимуму в масиві з N елементів визначається величиною H_N (на нескінченній кількості випробувань), тоді:

$F_a(n)=1+(N-1)*(3+2)+2*(Ln(N)+\gamma)=5*N + 2*Ln(N)-4+2*\gamma= \Theta(n)$.

Порівняння двох алгоритмів по їх функції трудомісткості вносить деяку помилку до отримуваних результатів. Основними причинами цієї помилки є те, що елементарні операції зустрічаються з різною частотою, і існує відмінність в часах виконання елементарних операцій на реальному процесорі, породжена різними алгоритмами їх реалізації. Таким чином, виникає задача переходу від функції трудомісткості до оцінки часу роботи алгоритму на конкретному процесорі.

Пооперaційний аналіз

Ідея пооперaційного аналізу полягає в отриманні пооперaційної функції трудомісткості для кожної з використовуваних алгоритмом елементарних операцій з урахуванням типів даних. Наступним кроком є експериментальне визначення середнього часу виконання даної елементарної операції на конкретній обчислювальній машині. Очікуваний час виконання алгоритму розраховується як сума добутків пооперaційної трудомісткості на середні часи виконання операцій:

$$T_A(N) = \sum F_{aoni}(N) * \bar{t}_{oni}$$

Приклад пооперaційного часового аналізу

У ряді випадків саме пооперaційний аналіз дозволяє виявити тонкі аспекти раціонального застосування того або іншого алгоритму вирішення задачі. Розглянемо задачу множення двох комплексних чисел:

(a+bi)*(c+di)=(ac - bd) + i(ad + bc)=e + fi

Алгоритм A1 (пряме обчислення e, f)	***Алгоритм A2 (обчислення e, f за три множення)***
MultComplex1 (a, b, c, d; e, f) *e←a*c - b*d* *f←a*d + b*c* *Return (e, f)* *End.* f_{A1} =8 операцій f_* =4 операцій $f_\pm$ =2 операцій $f_\leftarrow$ =2 операцій	*MultComplex2 (a, b, c, d; e, f)* *z1←c*(a + b)* *z2←b*(d + c)* *z3←a*(d - c)* *e←z1 - z2* *f←z1 + z3* *Return (e, f)* *End.* f_{A2} =13 операцій f_* =3 операцій $f_\pm$ =5 операцій $f_\leftarrow$ =5 операцій

Пооперaційний аналіз цих двох алгоритмів не складний, і його результати приведені праворуч від запису відповідних алгоритмів.

По сукупній кількості елементарних операцій алгоритм А2 поступається алгоритму А1, проте в реальних комп'ютерах операція множення вимагає більшого часу, чим операція додавання, і можна шляхом пооперaційного аналізу відповісти на питання: за яких умов алгоритм А2 краще ніж алгоритм А1?

Введемо параметри q і r, що встановлюють співвідношення між часом виконання операції множення, додавання і присвоєння для операндів дійсного типу. Тоді ми можемо привести часові оцінки двох алгоритмів до часу виконання операції додавання/віднімання:

$T_{*}=q*T_{+}$, $q>1$;

$T_{=}=r*T_{+}$, $r<1$

Тоді приведені до T_{+} часові оцінки мають вигляд:

$T_{A1} = 4* q*T_{+}+2*T_{+}+2*r*T_{+} = T_{+} *(4*q+2+2*r)$;

$T_{A2} = 3* q*T_{+}+5*T_{+}+5*r*T_{+} = T_{+} *(3*q+5+5*r)$;

Рівність часів буде досягнута за умови:

$4*q+2+2*r = 3*q+5+5*r$,

звідки:

$q = 3 + 3*r$

і отже при $q>3+3*r$ алгоритм А2 працюватиме ефективніше.

Таким чином, якщо середовище реалізації алгоритмів А1 і А2 (мова програмування, обслуговуючий його компілятор і комп'ютер на якому реалізується завдання) таке, що час виконання операції множення двох дійсних чисел більш ніж втричі перевищує час складання двох дійсних чисел, в припущенні, що $r<<1$, а це реальне співвідношення, то для реалізації переважніший алгоритм А2.

Звичайно, виграш в часі малий, якщо перемножується тільки два комплексні числа, проте, якщо цей алгоритм є частиною складної обчислювальної задачі з комплексними числами, що вимагає істотно значимої за часом кількості множень, то виграш в часі може бути відчутним. З тільки що проведеного аналізу зрозуміло, що оцінка такого виграшу на одне множення комплексних чисел становить :

$\Delta T = (q–3–3*r)*T_{+}$