Горошко Ю.В.
Чернігівський національний педагогічний університет імені Т.Г. Шевченка

Про часову складність алгоритмів

Як правило, розглядувані в курсі основ програмування і алгоритмізації задачі стандартизовані, але їх недосить, щоб підготувати творчого вчителя інформатики. Тому, крім таких задач, треба розглядати і більш складні задачі, які мають нестандартне формулювання, фабулу, неочевидні методи розв'язування, так звані олімпіадні задачі. Це надасть можливість підвищити у студентів навички інформаційного моделювання, зацікавити їх даною дисципліною, підготувати до творчої педагогічної діяльності, сформувати відповідні компетентності щодо навчання основ програмування, проведення шкільних олімпіад. Якраз для таких творчих, нестандартних задач надзвичайно важливим стає етап тестування отриманої програми.

На жаль, більшість студентів нехтує цим етапом, вважаючи, що, якщо програма «запрацювала», то вона вже правильна. У найкращому випадку перевіряється правильність програми на одному з простих тестів. Тому дуже важливо навчити студентів правильно тестувати програми, добирати систему тестів таким чином, щоб охопити всі можливі нюанси щодо вхідних даних, наприклад граничні значення.

Вже на етапі побудови моделі задачі, а тим більше в ході тестування важливо звернути увагу на ефективність алгоритмів, що розробляються, зокрема на питання, що стосуються часової складності алгоритмів, оскільки часто очевидний алгоритм розв'язування задачі виконується за неприйнятно довгий проміжок часу. Потрібно навчити студентів визначати часову складність алгоритмів, знайомити їх з ефективними алгоритмами для певних класів задач (швидкими алгоритмами сортування, пошуку), а також із способами оптимізації алгоритмів.

В межах даної статті будемо вважати алгоритмом програму, що призначена виключно для виконання обчислень, без виконання інших операцій, наприклад введення-виведення. Прикладами таких обчислень можна назвати обчислення за формулами, пошук в структурах даних тощо.

За допомогою аналізу часової складності алгоритму можна з'ясувати, як буде виконуватися алгоритм при збільшенні обсягу вхідних даних.

Розглянемо один з найпростіших алгоритмів: знаходження максимального елемента масиву:

```
var a: array[1..n] of Integer;
       max: Integer;
     i: Integer;
begin
  max := a[1];
  for i := 1 to n do
    if (a[i] >= max) then
        max := a[i];
end;
```

В цьому алгоритмі можна виокремити наступні інструкції:

- пошук 1-го елемента масиву, вважати максимальним 1-й елемент масиву, надання змінній max значення 1-го елемента масиву, лічильнику `i` - значення 0 та порівняння лічильника `i` з `n` - всього 4 інструкції, які виконуються один раз поза циклом;

- пошук `i`-го елемента масиву (2 рази), порівняння `i`-го елемента масиву з `max`, порівняння `i` з `n` та збільшення `i` на одиницю, які виконуються `n` разів у циклі.

Для аналізу часової складності потрібно передбачати найгірший сценарій, тобто такий, за яким виконуватиметься найбільша кількість інструкцій - в розглядуваному випадку це буде тоді, коли масив відсортований за зростанням. В цьому випадку в тілі циклу також `n` разів виконуватиметься інструкція надання змінній `max` значення `i`-того елемента масиву, а отже всього у циклі будуть виконуватись 6 інструкцій.

Таким чином отримаємо наступну функцію від `n`:

`f(n)=4+6n`

Перший доданок не залежить від `n` і його можна відкинути.

Залишається функція `f(n)=6n`.

Також можна відкинути константу біля `n`, оскільки необхідно дослідити, як змінюються значення `f(n)` при великих значеннях `n`, що ще називають асимптотичною поведінкою.

Часову складність алгоритму позначають через `O(складність)`, таким чином для наведеного вище алгоритму пошуку максимального елемента масиву часова складність буде `O(n)`.

Розглянемо тепер програмний код для знаходження максимального елемента квадратної матриці

```
a[nxn].
   max:=a[1,1];
   for i:=1 to n do
   for j:=1 to n do
     if a[i,j]>max then
        max:=a[i,j]
```

В цьому прикладі змінна i набуває значень від 1 до n. При кожному значенні змінної i змінна j також набуває значень від от 1 до n. Таким чином для кожного з n повторень зовнішнього циклу внутрішній цикл також повторюється n разів і загальна кількість повторень тіла внутрішнього циклу буде n*n. Врахувавши наведені вище міркування про відкидання констант, можна сказати, що часова складність наведеного коду буде $O(n^2)$.

Оцінюючи часову складність алгоритму, що складається з кількох частин, що слідують одна за одною, потрібно оцінити часову складність кожної з цих частин, а потім для визначення загальної часової складності взяти найбільшу часову складність з них.

Нехай алгоритм складається з двох частин, часова складність першої частини буде n^3, а другої - n. Функція f(n) для визначення часової складності буде: $f(n)=n^3+n$, але часова складність всього алгоритму буде $O(n^3)$. Дійсно, при n=100, різниця між $n^3+n=1000100$ і $n^3=1000000$ буде тільки 100, що складає 0,01%.

Якщо алгоритм не залежіть від n, його називають алгоритмом константної складності. Будь-який алгоритм, час виконання якого не залежить від кількості вхідних даних, має константну складність, яка позначається O(1).

Далі доцільно розглянути приклади знаходження часової складності алгоритмів для наступних прикладів, використовуючи принцип відкидання констант і залишання тільки того доданку, який найшвидше зростає:

Для $f(n) = 5n + 12$ часова складність буде O(n) на основі наведених вище міркувань.

Для $f(n) = 125$ часова складність буде O(1)

Для $f(n) = n^2 + 3n + 112$ часова складність буде $O(n^2)$, оскільки n^2 зростає швидше, ніж 3n, а 3n в свою чергу, зростає швидше, ніж 112.

Для $f(n) = n^3 + nlog(n) + 2500$ часова складність буде $O(n^3)$.

Для $f(n) = n + sqrt(n)$ часова складність буде O(n), тому що n зростає швидше, ніж sqrt(n).

Для формування компетентностей з визначення часової складності алгоритмів доцільно запропонувати студентам розв'язати наступні вправи – визначити часову складність алгоритму за заданим виразом f(n):

1. $f(n) = n^6 + 3n$
2. $f(n) = 2^n + 12$
3. $f(n) = 3^n + 2^n$
4. $f(n) = n^n + n$

Отже, можна сформулювати три важливих правила для визначення часової складності алгоритмів.

1. O(k*f)=O(f)
2. O(f*g)=O(f)*O(g) або O(f/g)=O(f)/O(g)
3. O(f+g) дорівнює домінанті O(f) і O(g)

В наведених вище правилах k означає константу, а f і g - функції.

В першому правилі декларується, що постійні множники не мають значення для визначення часової складності алгоритму, наприклад O(1.5*n)=O(n).

Із другого правила слідує, що часова складність добутку дорівнює добутку часових складностей, наприклад

$$O((17*n)*n) = O(17*n)*O(n) = O(n)*O(n)=O(n*n) = O(n^2)$$

Із третього правила слідує, що для визначення часової складності суми двох функцій f(n) і g(n) потрібно вибрати більший із доданків, наприклад

$$O(n^5+n^2)=O(n^5)$$

В якості більш складного прикладу для визначення часової складності алгоритму розглянемо знаходження такої складності для алгоритму бінарного пошуку у відсортованому масиві.

Потрібно визначити позицію (індекс) заданого значення key у відсортованому за зростанням масиві a. Суть алгоритму полягає у наступному: знаходимо середній елемент масиву і порівнюємо його з key. Якщо ці два значення співпадають, то позицію значення key визначено. Якщо значення

key є меншим, ніж значення середнього елемента масиву, для подальшого розгляду обираємо лівий підмасив, в протилежному випадку - правий. З обраним підмасивом продовжуємо виконувати дії за описаним вище алгоритмом. Ці дії продовжуємо до знаходження спів падання значення key з елементом масиву.

```
Var a: array[1..n, 1..n] of integer;
function search(low, high, key: integer): integer;
var mid, data: integer;
begin
  while (low<high) do
  begin
    mid:=(low+high) div 2;
    data:=a[mid];
    if (key=data) then
    begin
      search:= mid;
      break;
    end
    else
      if (key<data) then
        high:=mid-1
      else
        low:=mid+1;
  end;
end;
```

Розв'язання:

Для знаходження розв'язку потрібно визначити m - кількість повторень тіла циклу в алгоритмі. У першому повторенні тіла циклу розглядається весь масив. У кожному наступному повторенні розмір підмасиву, що розглядається, стає у два рази меншим. Отже набір розмірів підмасивів має наступний вигляд:

$$n,\ n/2^1,\ n/2^2,\ n/2^3,\ n/2^4,\ldots,\ n/2^m$$

Значення m в цій послідовності повинно задовольняти нерівності $n/2^m<2$, звідки слідує $n<2^{m+1}$.

Оскільки m - це перше ціле число, для якого $n/2^m<2$, то правильною повинна бути і наступна нерівність $n/2^{m-1}>=2$, звідки слідує $2^m=<n$.

Із обох цих нерівностей слідує, що $2^m=<n<2^{m+1}$. Візьмемо логарифм за основою два від кожної частини нерівності і отримаємо

$$m=<\log_2 n<m+1$$

Отже, можна зробити висновок що часова складність розглянутого алгоритму $O(\log_2 n)$.

Зауважимо, що оскільки логарифми з різними основами від одного і того значення відрізняються один від одного на константу, у виразі для часової складності часто пишуть «просто» логарифм, без уточнення його основи.

Розглянемо тепер часову складність алгоритму пошуку у глибину у зв'язному графі, що заданий матрицею суміжності. a[n, n] - матриця суміжності, no_visit[n] — масив, в якому відмічаються відвідані вершини.

```
var a: array[1..n, 1..n] of integer;
    no_visit: array[1..n] of boolean;
    i: integer;
{ відмітимо всb вершини графа невідвіданими: }
for i:=1 to n do
  no_visit:=true;

procedure deep(v: integer);
var i: integer;
begin
  { відмітимо v-ту вершину відвіданою: }
  no_visit[v]:=false;
  { розглянемо всі ще невідвідані вершини,
    суміжні з v-тою }
  for i:=1 to n do
    if ((a[v,i]>0) and no_visit[i]) then
      deep(i);
end;
```

Наведена процедура пошуку в глибину є рекурсивною. Часову складність цієї процедури можна

визначити за наступними міркуваннями.

Оскільки для кожної вершини графа, яка ще не відвідана, виклик процедури `deep()` виконується тільки раз, то всього таких викликів буде стільки, скільки вершин в графі. При кожному виклику цієї процедури кількість операцій, що будуть виконані, пропорційна кількості ребер, інцидентних цій вершині, а, отже загальна кількість операцій при всіх викликах процедури `deep()` буде пропорційна загальній кількості ребер.

Якщо позначити кількість вершин через `V`, а кількість ребер через `E`, то тоді часова складність алгоритму пошуку в ширину буде `O(V+E)`.

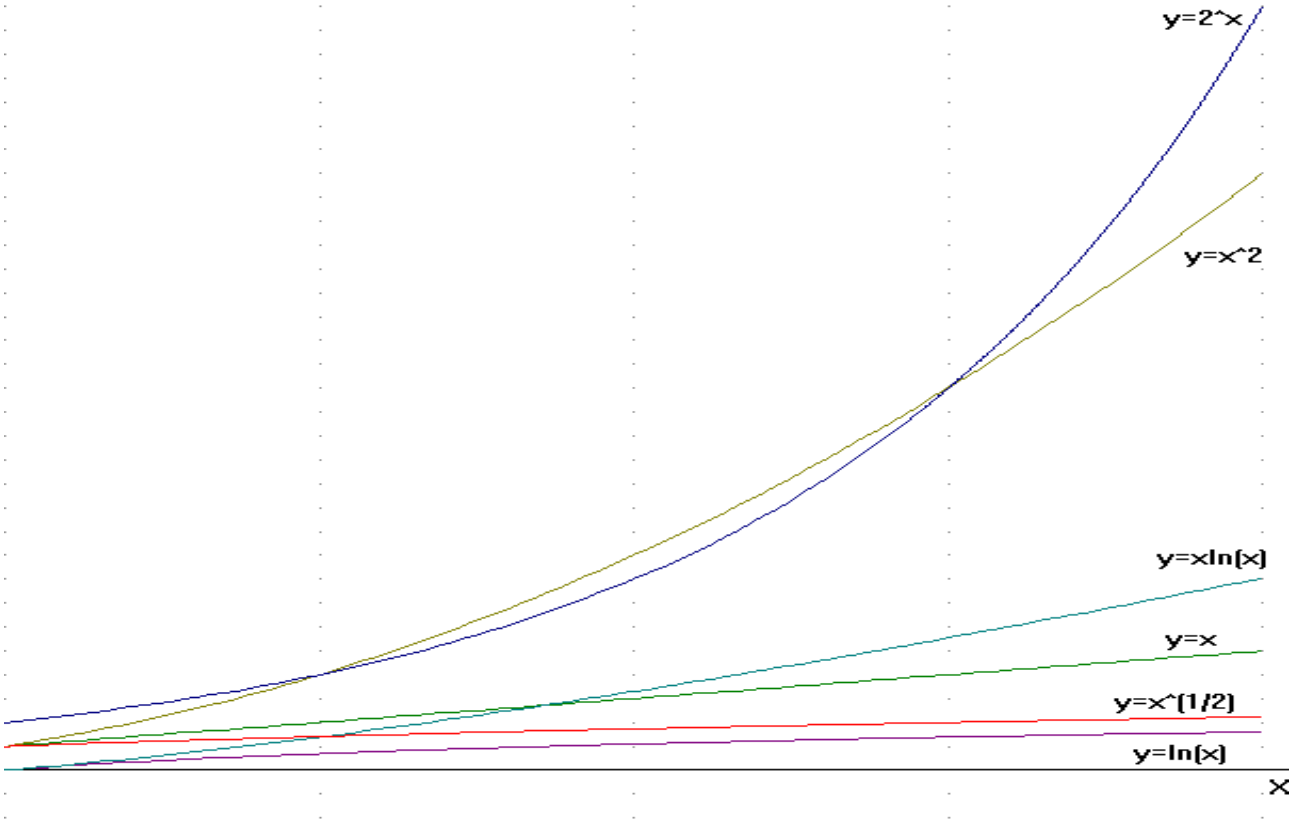

Рис. 1

Оскільки при розв'язуванні багатьох задач потрібно використовувати сортування масивів, доцільно ознайомити студентів із часовими складностями алгоритмів сортування масиву. Так найбільш прості і зрозумілі студентам алгоритми сортування «бульбашка» та сортування вибором є неефективними щодо часової складності, яка для них є $O(n^2)$. Алгоритм швидкого сортування (quick sort), що є в багатьох бібліотеках підпрограм, в загальному випадку має часову складність $O(n*\log(n))$, проте на деяких «незручних» наборах даних його часова складність збільшується до $O(n^2)$, чим часто користуються при добиранні таких даних у тестах, наприклад на олімпіадах з інформатики. Часова складність $O(n*\log(n))$ гарантується, коли використовуються більш складні алгоритми сортування, наприклад сортування «купою» (heap sort).

На рис. 1 наведено графіки деяких функцій, які часто фігурують у виразах часової складності алгоритмів (при побудові вибиралось деяке фіксоване значення відповідної константи). Перелічимо ці та інші часто використовувані функції у порядку зростання часової складності (`c` - константа, `n` - змінна).

1. c
2. $\log(n)$
3. n^c, $0<c<1$
4. n
5. $n \log(n)$
6. n^c, $c>1$
7. c^n, $c>1$
8. $n!$

Слід зауважити, що основним недоліком О-функцій є їх надмірна «грубість», пов'язана з відкиданням констант. Якщо за алгоритмом А розв’язування певної задачі відбувається за $0.001*n$

сек, а в той же час за алгоритмом В для цього потрібно `1000*n` сек, то фактично алгоритм В в мільйон разів більш повільний, ніж алгоритм А. Проте ці алгоритми мають однакову часову складність `O(n)`. Особливо перевагу алгоритму А можна буде відчути при малих значеннях `n`. Може бути і так, що особливо ефективно реалізований алгоритм із складністю $O(n^2)$ при малих `n` буде виконуватися швидше, ніж недостатньо ефективно реалізований алгоритм розв'язування тієї самої задачі з часовою складністю `O(n)`.

Список використаних джерел

1. Горошко Ю.В. Система інформаційного моделювання у підготовці майбутніх учителів математики та інформатики. – Рукопис. Дисертація на здобуття наукового ступеня доктора педагогічних наук за спеціальністю 13.00.02 – теорія та методика навчання (інформатика). – Національний педагогічний університет імені М.П. Драгоманова. – Київ, 2013
2. Введение в анализ сложности алгоритмов (часть 1). – Режим доступу: http://habrahabr.ru/post/196560/ – 5.09.2014 р.
3. Оценка сложности алгоритмов. – Режим доступу : http://habrahabr.ru/post/104219/ – 5.09.2014 р.
4. Оценка программ. – Режим доступу : http://www.structur.h1.ru/ocenka.htm/ – 5.09.2014 р.
5. Знай сложности алгоритмов!. – Режим доступу: http://habrahabr.ru/post/188010/ – 5.09.2014 р.

Біляй Ю.П.

Національний педагогічний університет імені М.П. Драгоманова

Використання віртуалізованих робочих столів у навчальному процесі

З появою хмарних технологій ідея «мати доступ до чого завгодно, звідки завгодно, з чого завгодно» стала поступово втілюватися в реальність. Усього за кілька років поняття «приватна хмара», VDS, VDI і т.п. стали масово використовуватись. Тим не менш, до цих пір існує певна плутанина термінів і понять, а також ряд невирішених питань. Наприклад, що таке віртуалізація десктопів? Це технологія чи підхід? У чому відмінність віртуалізації десктопів від інфраструктури віртуальних робочих столів?

Серед усіх численних *aaS з'явилось поняття, як DaaS – Desktop as a Service. Фактично – повноцінні робочі місця, запущені десь на віддаленому сервері, з доступом до них за допомогою тонких клієнтів.

Віртуалізація десктопів або віртуалізація робочих столів – це підхід, за якого відбувається поділ робочого середовища користувача (ОС, додатки, дані) і фізичного пристрою, на якому він звик працювати (ПК, ноутбук). Завдяки цьому підходу студенти та викладачі можуть не залежати від фізичного робочого місця в аудиторії чи на кафедрі, а можуть працювати із потрібними програмами та даними з будь-якого пристрою (планшет, смартфон, тонкий клієнт і т.д.) з будь-якого місця (вдома, в дорозі або з будь-якого місця, де є доступ до мережі Інтернет).

В основі цього підходу лежить не одна конкретна технологія, а спільне використання різних технологій в галузі клієнтської віртуалізації.

Найбільш популярними з них сьогодні є:

1. Інфраструктура віртуальних робочих столів (VDI) – система, за допомогою якої можна запустити ОС користувача (Windows і Linux) на віртуальній машині на сервері в центрі опрацювання даних (ЦОД) і працювати з нею віддалено з будь-якого пристрою (Citrix XenDesktop, VMware View, Microsoft VDI, Quest vWorkspace).
2. Служби віддалених робочих столів або термінальні сервіси (Remote Desktop Services Host (RDSH) \ Terminal Services (TS)) – класичний термінальний доступ, через який надається серверна операційна система (зазвичай, Windows Server 2008 R2 або 2012) кільком користувачам в конкурентному режимі (може працювати той, хто першим завантажив профіль). Кожен з віддалених користувачів працює в своїй сесії. Найбільш популярні рішення – Citrix XenApp, Microsoft RDS, Quest vWorkspace.
3. Дистанційна фізична робоча станція (Blade PC) – потужна високопродуктивна робоча станція (часто з встановленим графічним адаптером) у форм-факторі сервера, розташована в ЦОД, через яку відбувається доступ до обчислювальних ресурсів віддаленим користувачам. Найбільш популярні рішення – Citrix HDX 3D Pro + Dell R5500, VMware View + Dell R5500.
4. Віртуалізація додатків (Application Virtualization) – постачання і виконання додатків на віртуальних машинах, термінальний сервер або ПК без звичного встановлення програми в ОС. Найбільш популярні рішення: Microsoft App-V, Citrix XenApp, VMware ThinApp.