Упражнение №1 по САА

Оценка и сложност на алгоритми

Оценката на сложността на един алгоритъм дава реална представа за неговата бързина и ефективност посредством математическия апарат. Когато разглеждаме даден компютърен алгоритъм, се интересуваме най-общо от три негови свойства:

- простота (и елегантност)
- коректност
- бързодействие

Докато първото от тях всеки може да "премери" интуитивно (и донякъде субективно), то за последните две е необходим много по-задълбочен анализ. Оценката на алгоритмите ни дава възможност лесно и точно да определяме ефективността на даден алгоритъм, да сравняваме алгоритми, да повишаваме бързодействието им чрез премерени и точни промени.

Нека разгледаме следния програмен фрагмент:

```
1)     n = 100;
2)     sum = 0;
3)     for (i = 0; i < n; i++)
4)          for (j = 0; j < n; j++)
5)               sum++;
```

Интересува ни колко бързо ще работи горната програма. Това, което можем да направим експериментално, е да проверим за какво време ще завърши работата си. За да изследваме по-общо нейното поведение, можем да я изпълним за други стойности на *n*. *Таблица 1.* показва зависимостта между размера на входните данни и скорост на изпълнение.

Таблица 1. Време за изпълнение на програмния фрагмент

Размер на входа	Време за изпълнение
10	0,000001 сек.
100	0,0001 сек.
1000	0,01 сек.
10000	1,071 сек.
100000	106,543 сек.
1000000	10663,6 сек.

Вижда се, че когато увеличаваме *n* десет пъти, времето за изпълнение на програмата се увеличава 100 пъти.

Нека изследваме по-задълбочено горния фрагмент. На редове 1) и 2) има статична инициализация, която отнема константно време. Да го означим с *a*. За операциите $i = 0$ и i++, както и за проверката $i < n$, отново е необходимо константно време (всяка от тях

представлява константен брой инструкции на процесора), ще го означим съответно с b, c, d. На ред 4) времената, необходими за операциите $j = 0$, $j < n$ и j++, означаваме с e, f, g. Последно, операцията на ред 5) също изисква константно време: нека бъде h.

С така въведените означения не е трудно да се пресметне общото време на работа на програмата за произволна стойност на n:

$$a + b + n.c + n.d + n.(e + n.f + n.g + n.h) =$$
$$= a + b + n.c + n.d + n.e + n.n.f + n.n.g + n.n.h =$$
$$= n^2.(f + g + h) + n.(c + d + e) + a + b$$

Припомняме, че a, b, c, d, e, f, g, h са константи. Нека означим:

$$i = f + g + h$$
$$j = c + d + e$$
$$k = a + b$$

(тук i и j нямат общо с използваните във фрагмента по-горе променливи). Така алгоритъмът се изпълнява за време:

$$i.n^2 + j.n + k$$

Константите i, j и k са *важни* за бързодействието на алгоритъма, но не са *определящи*. На практика когато изследваме ефективността на даден алгоритъм, ние не се интересуваме от тях. Тези константи зависят предимно от машинното представяне на нашата програма, както и от бързината на машината, на която тя се изпълнява.

Нещо повече, когато изследваме поведението на нашия алгоритъм, можем да игнорираме дори и едночлените $j.n$ и k и да оставим единствено този, в който n участва в най-висока степен. Целта на това "опростяване" е да оставим само *най-значимата* характеристика за даден алгоритъм, т.е. функцията, от която в най-голяма степен зависи времето на изпълнение, т.е. която нараства най-бързо, когато се увеличава размерът на входните данни.

Задача за упражнение:

Задача 1. Дадени са три алгоритъма със сложности съответно $5n^2 - 7n + 13$, $3n^2 + 15n + 100$ и $1000n$. Кой от тях следва да се използва при входни данни с размер до: 100; 1000; 10000; 1000000?

Асимптотична нотация

Когато се интересуваме от сложността на алгоритъм, най-често се интересуваме как се държи при достатъчно голям размер n на входните данни. При формалното оценяване на сложността на алгоритмите ще се интересуваме от поведението им при n, клонящо към безкрайност. Сложността на даден алгоритъм ще описваме с функции от вида $f : N \rightarrow N$. (Ще припомним, че с N означаваме множеството на естествените числа: 0,1,2,...). Понякога ще работим с функции, дефинирани върху *подмножество* на N,

например валидни само за четни n или пък от дадена стойност нататък. Ще използваме и реални функции, например log n, като по принцип ще подразбираме тяхно ограничение върху N. Последните случаи не са особено "чисти" от теоретична гледна точка, но спестяват много трудности.

Дефиниция 1. $O(F(n)) = \{f(n) \mid \exists c\ (c > 0), \exists n_0(c): \forall n > n_0 : 0 \leq f(n) \leq c.F(n)\ \}$

Т.е. $O(F(n))$ е множеството от функции f, за които съществува константа c ($c > 0$) такава, че $f(n) \leq c.F(n)$, за всички *достатъчно големи* стойности на n, т.е. съществува константа n_0 (евентуално зависеща от c), за която горното неравенство е изпълнено за всяко $n > n_0$. Така $O(F)$ определя множеството на всички функции, които *нарастват* ***не*** *по-бързо* от F.

Когато се разглежда сложност на алгоритмите, се използват още две основни означения: $\Theta(F)$ и $\Omega(F)$.

Дефиниция 2. $\Omega(F(n)) = \{f(n) \mid \exists c\ (c > 0), \exists n_0\ (c): \forall n > n_0 : f(n) \geq c.F(n) \geq 0\}$

Т.е. $\Omega(F)$ е множеството от функции $f(n)$, за които $f(n) \geq c.F(n)$ за всяко $n > n_0$. Така $\Omega(F)$ включва всички функции, които *нарастват* ***не*** *по-бавно* от F.

Дефиниция 3.

$\Theta(F(n)) = \{f(n) | \exists c_1\ (c_1 > 0), \exists c_2(c_2 > 0), \exists n_0(c_1,c_2): \forall n \geq n_0 : 0 \leq c_1.F(n) \leq f(n) \leq c_2.F(n)\}$

Непосредствено от дефиницията следва, че $\Theta(F) = O(F) \cap \Omega(F)$. Т.е. $\Theta(F)$ съдържа всички функции, които нарастват *толкова бързо, колкото* и F (с точност до константен множител).

Нарастване на основните функции

При оценка на сложността на алгоритми най-често се използват следните функции: c, log n, n, n log n, n^2, n^3, n^k, 2^n, $n!$, n^n. Тук сме ги подредили по-скорост на нарастване. За да придобие читателят по-добра представа за скоростта им на нарастване, прилагаме *таблица 2.*, показваща стойностите на функциите при различни стойности на аргумента им n.

Таблица 2. Нарастване на някои по-често използвани функции

Функция	***Стойност***				
	$n = 1$	$n = 2$	$n = 10$	$n = 100$	$n = 1000$
5	5	5	5	5	5
log n	0	1	3,32	6,64	9,96
n	1	2	10	100	1000
n log n	0	2	33,2	664	9966
n^2	1	4	100	10000	10^6
n^3	1	8	1000	10^6	10^9
2^n	2	4	1024	10^{30}	10^{300}
$n!$	1	2	3628800	10^{157}	10^{2567}
n^n	1	4	10^{10}	10^{200}	10^{3000}

Сортиране и търсене

Често при работа с големи еднотипни данни се налага въвеждането на някаква наредба с цел по-лесната им обработка. Подредбата на елементите би могла да ни даде значително по-ефективен алгоритъм за търсене в сравнение със случая, когато данните не са подредени.

Прието е процесът на пренареждане (пермутиране по подходящ начин) на елементите на някакво множество от обекти в определен ред да се нарича сортиране. Сортирането е основна дейност с широка сфера на приложение: речници, телефонни указатели, справочни индекси и въобще навсякъде, където се налага бързо търсене и намиране на различни обекти. Сортирането е неделима част от нашето ежедневие – зад всяко подреждане, от бюрото на което седим, до гардероба, чантата ни и т.н. се крие някакъв вид сортиране.

Класически алгоритми за сортиране

Съществуват три класически елементарни универсални методи за сортиране чрез сравнение: чрез вмъкване, чрез избор и по метода на мехурчето. В основата на всеки един от тях стои проста идея, позволяваща бърза и ясна реализация. Елементарните методи за сортиране са ефективни при сравнително малък брой елементи (около 20) и често се използват на практика. За съжаление, при по-голям брой елементи скоростта им рязко пада, поради което се налага използване на други методи. Действително, и трите елементарни метода се характеризират с алгоритмична сложност $\Theta(n2)$, което е доста по-бавно в сравнение със сложността $\Theta(n.\log 2\ n)$, ха-рактерна за съвременните методи за сортиране като пирамидалното или бързото сортиране. Въпреки това, елементарните методи имат своето място, тъй като за достатъчно малки последователности те са по-ефективни и, както ще видим по-долу, често се използват в хибридни варианти с цел повишаване на бързодействието. Така например, широко използван подход при бързото сортиране е, при достигане до дял с достатъчно малко елементи да се използва по-прост метод като сортиране чрез вмъкване.

(*вече би трябвало да сте ги научили по ПИК-овете, а за по-сигурно ще ги разглеждате и на лекции*)

Предварително дефинирани структура от данни и метод за размяна:

```
#define MAX 100
struct CElem {
     int key;
     /* ............. Някакви данни ............. */
} m[MAX];


/* Разменя стойностите на *x1 и *x2 */
void swap(struct CElem *x1, struct CElem *x2)
{ struct CElem tmp = *x1; *x1 = *x2; *x2 = tmp; }
```

Бързо сортиране на Хоор

Идеята на алгоритъма, предложен от Хоор, е да изберем някакъв елемент *x* и да разделим масива на два дяла: ляв, в който елементите са по-малки от *x*, и десен, в който са по-големи. Прилагаме същия алгоритъм за лявата и дясната част, намалявайки постепенно лявата и дясната граница на разглежданите подмасиви, докато не достигнем до интервали, съдържащи единствен елемент. След приключване работата на алгоритъма, масивът ще бъде сортиран. (*Защо?*)

Да означим с *q* индекса на *x* в масива, т.е. x = m[q]. Нека сортираме подмасива m[l,l+1,...,r] и нека partition() го разделя на две части: лява (m[l,l+1,...,q]) и дясна (m[q+1,q+2,...,r]), и връща *q* като резултат. Забележете, че разделянето почти сигурно е свързано с размени на елементи, т.е. partition() *не търси* елемент *x* в масива, ами го *избира* и извършва разделяне на две области относно стойността му. По-долу ще видим как може да стане това.

Бързото сортиране би могло да се запише най-общо така (ТУК Масивът m[] не се подава като параметър, а се разглежда като глобална променлива: така се пести място в стека и се печели скорост, тъй като функцията е рекурсивна.):

```
void quickSort(int l, int r)
{
     int q;
     if (l < r) {
          q = partition(l,r);
          quickSort(l,q);
          quickSort(q+1,r);
     }
}
```

Нека предположим, че въз основа на някаква формула вече сме пресметнали *q*. (На първо време ще избираме най-левия или най-десния елемент на масива преди разделянето.) Ще отбележим, че m[q] съдържа стойността на *x преди* разделянето. След разделянето *q* ще бъде граничен индекс: вляво от него (включително) ще имаме елементи, чийто ключ не надвишава стойността на *x*, а вдясно ключове, по-големи от *x*. Възниква следващият съществен въпрос: Как да извършваме разделянето? Съществуват най-общо два различни подхода. Следва първият:

```
unsigned partition(int l, int r) {
     int q, j, x;
     q = l - 1; x = m[r].key;
     for (j = l; j <= r; j++)
          if (m[j].key <= x) {
               q++;
               swap(m+q,m+j);
          }
          if (q == r) /* Всички елементи са <= x. Областта намалява с 1. */
     q--;
     return q;
}
```

Как работи предложеният метод? Избира се елемент x, по който да се извършва разделя-нето. В лявата част на масива трябва да останат елементи, по-малки или равни на x, а в дясната — строго по-големи от x. Разглежданият дял от масива m[l,l+1,...,r] се преглежда отляво-на-дясно, като при това в лявата му част се изгражда постоянно разширяваща се област от елементи, по-малки или равни на x. Десният край на областта се определя от q. Когато j достигне края r на дяла, q ще сочи границата между двете области. (*Забележка:* При реална програма е добре обръщението към функцията swap() да се замени с нейния код с цел по-голяма ефективност.)

Друг възможен метод е да се използват два индекса i и j, показващи границите на *две* непрекъснато разширяващи се области от *двата* края към центъра. На всяка стъпка на алгоритъма се прави опит за разширяване на лявата област надясно, докато това е възможно, т. е. докато вдясно от нея стои елемент, по-малък или равен на *x*. Същото се извършва и за дясната област (двата *while*-цикъла). След това се разменят местата на двата елемента-прегради за лявата и дясната области, които са спрели разширяването им, и процесът се повтаря отначало. Приключва при “срещане” на двете области, т. е. когато двете им граници се разминат. Индексът i сочи десния край на лявата област, а j — левия край на дясната:

```
unsigned partition(int l, int r) {
     unsigned i, j, x;
     i = l; j = r; x = m[l].key;
     do {
          while (x > m[i].key) i++;
          while (x < m[j].key) j--;
          if (i <= j) {
               swap(m+i,m+j);
               i++;
               j--;
          }
     }while (i <= j);
     return j;
}
```

След обединяване на функциите partition() и quickSort() в едно, получаваме (за първия вариант):

```
void quickSort(int l, int r) {
     int i,j,x;
     i = l-1; x = m[r].key;
     for (j = l; j <= r; j++)
          if (m[j].key <= x) {
               i++;
               swap(m+i,m+j);
          }
     if (i == r) /* Всички елементи са <= x. Областта намалява с 1. */
          i--;
     if (l < i)
          quickSort(l,i);
```

```
     if ((i+1) < r) /***/
          quickSort(i+1,r); /***/
}
```

За втория вариант имаме:

```
void quickSort(int l, int r) {
     int i, j, x;
     i = l;
     j = r;
     x = m[r].key;
     do {
          while (x > m[i].key) i++;
          while (x < m[j].key) j--;
          if (i <= j) {
               swap(m+i, m+j);
               i++;
               j--;
          }
     } while (j >= i);
     if (j > l)
          quickSort(l, j);
     if (i < r) /***/
          quickSort(i, r); /***/
}
```

Втория вариант с избор на разделител по средата на масива, и предаване на целия масив като параметър.

```
void quickSort(int arr[], int left, int right)
 {
  int i = left, j = right;
  int tmp;
  int pivot = arr[(left + right) / 2];

  /* partition */
  while (i <= j) {
        while (arr[i] < pivot)   i++;
        while (arr[j] > pivot)   j--;
        if (i <= j) {
              tmp = arr[i];
              arr[i] = arr[j];
              arr[j] = tmp;
              i++;
              j--;
    }
}
/* recursion */
if (left < j)
    quickSort(arr, left, j);
if (i < right)
        quickSort(arr, i, right);
}
```