САА – Упражнение 6

<u>Определяне и оценка на сложност на алгоритъм</u>

1. ***Размер на входните данни*** – това е величина, която показва какво е количеството на входните данни на изследвания алгоритъм.

Нека е дадена задача, в която размерът на входните данни е определен от дадено цяло число *n*.

пример 1.

Даден е масив с *n* елемента, който трябва да бъде сортиран. Размерът на входните данни се определя от броя на дадените числа *n*.

пример 2.

При алгоритмите за операции с квадратни матрици *n x n*, се казва, че размерът на задачата е *n*, въпреки, че броят на елементите в матрицата е n^2.

пример 3.

При алгоритмите за графи *n* може да е броят на възлите или броят на ребрата или сумата от броя на възлите и броя на ребрата. Макар, че обектът е един и същ, *n* е различно за различните алгоритми. Стойността на *n* зависи от алгоритъма и от това, кои са най-трудоемките операции в него.

2. ***Определяне на времева сложност на алгоритъм***

Времето Т за изпълнение на даден алгоритъм може да се представи като функция на размера *n* на задачата – т.е. T(n) и това е времевата сложност на алгоритъма.

Елементарен оператор

Сложността на елементарен оператор е константа, т.е. $O(1)$.

Елементарен оператор е такъв, който се извършва за константно време, независещо от обема на обработваната информация. Елементарни операции в общия случай са присвояването, събирането, умножението и др. Когато обаче се работи със стоцифрени числа, умножението не е елементарна операция.

Последователност от оператори

Времевата сложност на последователност от оператори се определя от сложността на по-бавния от тях. Ако операторът s_1 със сложност F_1 е следван от оператора s_2 със сложност F_2, то:

$$T(s_1) \in O(F_1), T(s_2) \in O(F_2) \Rightarrow T(\mathtt{s_1;\ s_2}) \in \max(O(F_1), O(F_2))$$

Композиция (влагане) на оператори

При влагане на оператор в областта на действие на друг оператор сложността се пресмята като произведение от сложностите им, т.е.

$$T(s_1) \in O(F_1),\ T(s_2) \in O(F_2) \Rightarrow T(\mathtt{s_1\{s_2\}}) \in O(F_1.F_2))$$

if-конструкции

```
if (p)
      s1;
else
      s2;
```

Ако сложностите на p, s_1 и s_2 са $O(P)$, $O(F_1)$, $O(F_2)$, то сложността на показания фрагмент е $\max(O(P), O(F_1), O(F_2))$, т. е. сложността на най-бързо нарастващата функция измежду P, F_2 и F_3. (Приемаме, че условието p не е константа, т.е. в зависимост от входните данни е възможно да бъде както истина, така и лъжа).

$$T(p) \in O(P),\ T(s_1) \in O(F_1)\ ,\ T(s_2) \in O(F_2) \Rightarrow T(\mathtt{if\ (p)\ s_1;\ else\ s_2}) \in \max(O(P), O(F_1), O(F_2))$$

Цикли

Нека е даден цикъла:

```
int fact = 1;
for (int i = 1; i <= n; i++)
      fact = fact * i;
```

Може да се приеме, че тялото на цикъла отнема константно време c, независещо от n. Сложността на оператора за цикъл for е $O(n)$. Тогава по правилото за композицията за сложността на целия цикъл се получава $O(c.n)$, т.е. $O(n)$. Като се прибави и сложността на началната инициализация преди цикъла (която има сложност $O(1)$), се получава $O(1+n)$. В крайна сметка сложността се оказва $O(n)$.

Вложени цикли

```
int sum = 0;
for (int i = 0; i < n; i++)
  for (int j = 0; j < n; j++)
    sum++;
```

Сложността при два или повече вложени цикли с взаимно независими броячи може да се изведе лесно. В случая на два вложени цикъла от фрагмента по-горе тя е $f = n.O(g)$, където g е сложността на вътрешния цикъл, $g = O(n) \Rightarrow f = O(n.n) = O(n^2)$.

```
int sum = 0;
for (int i=0; i<n-1; i++)
  for (int j=i; j<n; j++)
    sum++;
```

Въпреки, че в този пример `sum++` ще се изпълни $\frac{n.(n-1)}{2}$ пъти, сложността на фрагмента ще бъде $O(n^2)$, (ако се приеме, че `sum++` е със сложност $O(1)$).

Задачи

Определете (изведете) сложността на следните фрагменти код:

1.

```
int sum = 0;
for (int i = 0; i < n*n; i++)
    sum++;
```

2.

```
int sum = 0;
for (i = 0; i < n; i++)
  for (j = 0; j < n; j++)
    if (i == j)
      for (k = 0; k < n; k++)
        sum++;
```

3.

```
for (i = 0; i < n; i++)
  for (j = 0; j < n; j++)
    if (i==j)
      break;
```

4.

```
int sum = 0;
for (i = 0; i < n; i++)
  for (j = 0; j < i*i; j++)
    sum++;
```

5.

```
int sum = 0;
for (i = 0; i < n; i++)
  for (j = 0; j < i*i; j++)
      for (k = 0; k < j*j; k++)
        sum++;
```

6.

```
int sum = 0;
int h = 1;
while (h < n)
{
      sum++;
      h = 2*h;
}
```

Рекурсия

За определяне на сложността на рекурсивен алгоритъм обикновено се съставя зависимост от вида $T(n) = f(T(n-1))$. За да се намери явния вид на сложността, се налага решаване на рекурентната зависимост, което в общия случай е тежко. За щастие в някои интересни практически случаи това не е толкова трудно.

- *Факториел*

Нека е дадена функцията:

```
int fact(int n)
{
  if (n < 2)
    return 1;
  return n*fact(n-1);
}
```

В този случай рекурсията е еквивалентна на единствен цикъл от тип *for*, откъдето за сложността се получава $O(n)$.

Основна теорема

При анализ на алгоритми от типа *разделяй и владей* често възникват рекурентни зависимости от вида:

$$T(n) = a.T(n/b) + c.n^k$$

Теорема 1. Нека е дадена рекурентната зависимост $T(n) = a.T(n/b) + c.n^k$, за $n > n_0$. Тук $a \geq 1$, $b>1$, $k \geq 0$, $c>0$, $n_0 \geq 1$ и a, b, k, c, n_0 са цели числа. Тогава решението на рекурентната зависимост се дава от формулата:

$$T(n) = \begin{cases} \Theta(n^k) & a < b^k \\ \Theta(n^k \log n) & a = b^k \\ \Theta(n^{\log_b a}) & a > b^k \end{cases}$$

Пример 1:

Нека е дадена рекурентната зависимост:

$$T(1) = 1$$
$$T(n) = 3T(n/2) + n, n \geq 2$$

Като се приложи горната теорема се получава: $a=3$, $b=2$, $c=1$, $k=1$, т.е.: $3=a>b^k=2$. Това е условието за третия случай и директно се получава сложност $\Theta(n^{\log 3})$.

Пример 2:

Нека е дадена рекурентната зависимост:

$$T(1) = d$$
$$T(n) = 4T(n/2) + n^2, n \geq 2$$

Използвайки горната теорема се получава: $a=4$, $b=2$, $c=1$, $k=2$, т.е.: $4=a=b^k=2^2$. Това е условието за втория случай и директно се получава сложност $\Theta(n^2.\log_2 n)$ *(без значение каква е стойността на d).*

Задачи

1. Сложността на даден алгоритъм се задава с рекурентната зависимост: $T(1) = 2$, $T(n) = 4T(n/3) + n^{\log_3 4}, \quad n \geq 2$. Тогава:

а) $\Theta(n^2 \log n)$; б) $\Theta(n \log n)$; в) $\Theta(n^{\log_3 4} \log n)$; г) $\Theta(n^{\log_3 4})$

2. Сравнете времето за изпълнение на двете функции за пресмятане на числата на Фибоначи. Рекурсивният или итеративният вариант е по-бърз? Обосновете отговора си.

```
int fibo_r(int n)
{
	if(n <= 1)
		return 1;
	return fibo_r(n-1)+fibo_r(n-2);
}



int fibo(int n)
{
	int i, f, f1 = 1, f2 = 1;
	for(i = 2; i <= n; i++)
	{
		f = f1+f2; f2 = f1; f1 = f;
	}
	return f;
}
```