КАТЕДРА: КОМПЮТЪРНИ СИСТЕМИ И ТЕХНОЛОГИИ
ДИСЦИПЛИНА: АЛГОРИТМИ СТРУКТУРИ ОТ ДАННИ
ДИСЦИПЛИНА: СИНТЕЗ И АНАЛИЗ НА АЛГОРИТМИ

ЛАБОРАТОРНО УПРАЖНЕНИЕ № 4

ТЕМА: Оценка и сложност на алгоритмите

ЦЕЛ:

Целта на упражнението е студентите да се запознаят с понятието сложност на алгоритъм. Те ще се научат да измерват времето за изпълнение на алгоритмите в езика C/C++. След упражнението студентите би следвало да могат да преценяват кой алгоритъм е по добър в дадени ситуации.

I. ТЕОРЕТИЧНА ЧАСТ

1. Оценка за 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 дава възможност лесно и точно да определя ефективността на даден алгоритъм, да се сравняват алгоритми, да се повишава бързодействието им чрез премерени и точни промени.

Разгледайте 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++;
```

За да разберем бързодействието на горния програмен код е възможно експериментално, да се провери за какво време ще завърши работата си. За да се изследва по-общо нейното поведение, може да се изпълни за различни стойности на n. В табл.1. се показва зависимостта между размера на входните данни и скорост на изпълнение.

Табл.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 се увеличава *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: $p=f+g+h$, $z=c+d+e$, $k = a + b$. Така алгоритъмът се изпълнява за време:

$$p.n^2 + z.n + k$$

Константите *p, z* и *k* са важни за бързодействието на алгоритъма, но те не са определящи. На практика когато се изследва 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 *z.n* и *k* и да оставим единствено този, в който *n* участва в най-висока степен. Целта на това "опростяване" е да се остави 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.

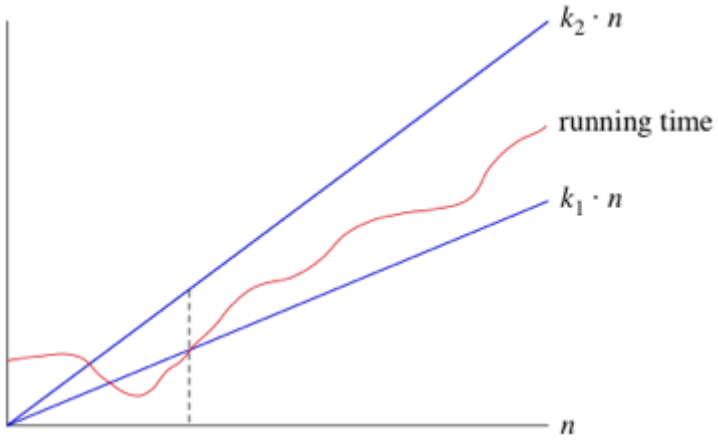
2. Асимптотична нотация

Съществуват няколко форми на асимптотични нотации: *Big-θ нотация, Big-O нотация, Big-Ω нотация*.

- Big-θ нотация (Big theta нотация)

Нотацията се бележи с гръцката буква *θ*, като във скоби се посочва само значителната част от броя на операциите за даден алгоритъм. За посочения по – горе пример е $\theta(n^2)$.

Ако времето за изпълнение на един алгоритъм е $\theta(n)$, то това означава че когато n стане достатъчно голямо, времето за изпълнение е поне $k_1.n$ и най–много $k_2.n$ за константи k_1 и k_2. На фиг.1 е показана диаграма за $\theta(n)$.



Фиг.1. Диаграма за θ(n)

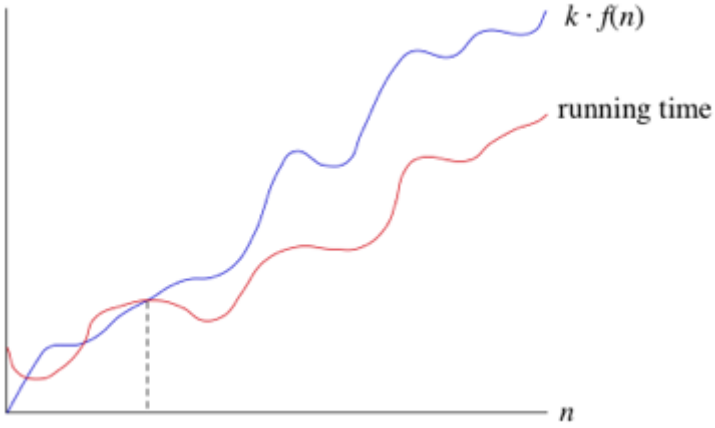
За малки стойности на n, сравнението между времето за изпълнение на $k_1.n$ и $k_2.n$ е без значение. Но когато n стане достатъчно голямо – достигне пунктираната линия или е от дясната ѝ страна – времето за изпълнение трябва да е между $k_1.n$ и $k_2.n$. Докато тези константи k_1 и k_2 съществуват, казваме, че времето за изпълнение е $\theta(n)$.

Когато използваме нотацията *big–θ*, това означава, че съществува асимптотична строга граница на времето за изпълнение. "Асимптотична", защото има значение само за големи стойности на n. "Строга граница", защото времето за изпълнение е ограничено от константи отгоре и отдолу.

- Big-O нотация

Нотацията се бележи с главна буква *O* като във скоби се посочва само значителната част от броя на операциите за даден алгоритъм. За горния пример се бележи по следния начин $O(n^2)$. Тя се използва за да направи асимптотично ограничение отгоре.

Ако времето за изпълнение е $O(f(n))$, то за достатъчно голямо n времето за изпълнение е най-много $k.f(n)$ за константа k. Това означава, че времето за изпълнение нараства най – много до $k.f(n)$. На фиг.2. е показана диаграма за $O(f(n))$.



Фиг.2. Диаграма за Big – O нотация

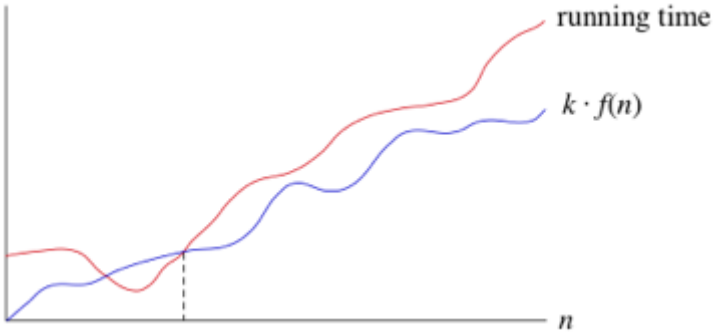
Нотацията се използва за асимптотични горни граници, тъй като тя дава горна граница на нарастването на времето за изпълнение за достатъчно големи входни данни.

Ако кажем, че времето за изпълнение е $\theta(f(n))$ в дадена ситуация, то е също и $O(f(n))$. Обратното обаче не е задължително да е вярно. Това е така понеже $\theta(f(n))$ задава асимптотична строга граница, а $O(f(n))$ само горна граница.

- Big-Ω нотация (Big Omega нотация)

Тази нотация се бележи с гръцката буква Ω. Използва се за задаване на асимптотична долна граница.

Ако времето за изпълнение е $\Omega(f(n))$, тогава за достатъчно голямо n времето за изпълнение е поне $k.f(n)$ за някаква константа k. На фиг.3. е представена диаграмата на тази нотация.



Фиг.3. Диаграма за Big-Ω нотация

Точно както $\theta(f(n))$ автоматично предполага каква е горната граница за $O(f(n))$, то също автоматично предполага и долната граница за $\Omega(f(n))$.

3. 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 придобиете по-добра представа за скоростта им на нарастване, разгледайте табл. 2., показваща стойностите на функциите при различни стойности на аргумента им n.

Табл.2. Нарастване на някои по-често използвани функции

Функция	Стойност				
	n=1	n=2	n=10	n=100	n=1000
$c=5$	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	9960
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}

4. Измерване на времето за изпълнение на алгоритмите.

За измерване на времето се използва метод който засича текущото време. За да определим времето за изпълнение на някакъв фрагмент от код, то трябва да се измери текущото време непосредствено преди него и веднага след него. Накрая се вадят двете засечени времена и се получава времето за изпълнение на съответния фрагмент.

В езика C/C++ за измерване на текущото време може а се използва функцията now() достъпна от клас „high_resolution_clock“. За да се използва е необходимо в началото на кода да се включи библиотеката „chrono“. Засичането на текущото време става по следния начин:

```
std::chrono::high_resolution_clock::now();
```

Резултата върнат за текущото време може да се запише в нова променлива от тип auto.

auto start=std::chrono::high_resolution_clock::now();

По същия начин може да се засече текущото време след края на фрагмента чиито код засичаме, записан в друга променлива. Двете времена се изваждат и се получава изчислимото време. За да се преобразува времето в подходящ тип (мили, микро, нано) секунди се използва функция duration от същата библиотека. Функцията може да се използва по следния начин:

```
std::chrono::duration_cast<microseconds>(stop - start);
```

Това би върнало резултата в микросекунди. За да не се пише всеки път “std::chrono::” пред използваните функции то бихме могли след библиотеките да напишем:

using namespace std::chrono;

II. ПРАКТИЧЕСКА ЧАСТ

ЗАДАЧА1: *Измерете времето за изпълнение на програмния фрагмент от примера в теоретичната част. Тествайте го при различни входни данни за n.*

КОД:

```
#include <iostream>
#include <chrono>
using namespace std;
using namespace std::chrono;

……..
auto start= high_resolution_clock::now();

int n = 1000;
int sum = 0;
for (int i = 0; i < n; i++)
      for (int j = 0; j < n; j++)
      sum=sum+i;

auto stop= high_resolution_clock::now();
auto duration= duration_cast<microseconds>(stop - start);
……..

cout<<”Sumata e”<<sum;
cout<<"Vremeto za izpalnenie e: "<<duration.count()<<" microseconds"<<endl;
```

ПОЯСНЕНИЕ

В началото са дефинирани необходимите библиотеки. В Променливата старт се засича текущото време за изпълнение преди програмния фрагмент, а в променлива стоп се – текущото време след програмния фрагмент. Чрез duration_cast се преобразува измереното време в съответния тип секунди и cout се отпечатва.

Времето за изпълнение зависи и от процесора(машината на която се изпълнява), затова при различни компютърни конфигурации може да покаже различен резултат. Ако входните данни за n е малка стойност то времето за изпълнение може да е под 1 микро секунда и в такъв случай ще показва 0. При увеличение стойността на n ще забележите и увеличение на времето за изпълнение.

III. Въпроси и задача за самостоятелна работа.

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? Обосновете отговора си.
2. Тествайте примерната задача в практическата част. Каква е Big O нотацията на сложност за тази задача.
3. Измерете времето на предходния алгоритъм като пробвате да добавите още един вложен for цикъл и отделно като премахнете вложените цикли(така че да се получи алгоритъм който търси само сумата на числата от 1 до n). Как се изменя времето за изпълнение на алгоритъма и неговата Big O нотацията на сложност.
4. Редактирайте предходния алгоритъм така че вместо сумата да търси произведението на числата от 1 до n. Измерете времето за изпълнение. Кой от двата алгоритъма е по бърз (за намиране на сумата или за произведението).
5. Измерете времето за изпълнение на алгоритъма в задачата за намиране на простите числа от предходното упражнение. Увеличете диапазона за търсене на простите числа многократно. Какво забелязвате във времето за изпълнение? Каква е нотацията на сложност за този алгоритъм?