


Структури от данни и алгоритми

Софтуерно Инженерство
Зимен семестър 2019-2020г

Преподавателски екип

- Лектор: д-р Милен Чечев 
 - 15 години опит като Софтуерен инженер/Data Scientist/Ръководител екип. [LinkedIn](#)
 - 10 години опит като преподавател: AI, ML, RecSys, СДА
- Петко Митков
 - Завършил Софтуерно инженерство
 - Любима книга за алгоритми: [SICP](#)
- Димитър Сейков
 - 3-ти курс Софтуерно Инженерство
 - 2 години професионален опит
- Васил Жухов
 - 3-ти курс Софтуерно Инженерство
 - Full Stack Developer
- Иван Лучев
 - Софтуерно Инженерство 3-ти курс
 - [Github](#)
- Александър Станев
 - 3-ти курс Компютърни Науки
 - Състезател по информатика
- Ангел Цанков
 - 3-ти курс Информатика
 - Състезател по информатика
- Николай Цонев
 - Приложна математика 3-ти курс
 -
- Михаил Михайлов
 - 3-ти курс Компютърни науки
 - Състезател по информатика

Организация на курса

Аудиторни часове: 45 часа лекции + 30 часа упражнения

Текущ контрол: 8 контролни по време на лекции и 14 задания за самостоятелна работа

Важно! - носете си лаптопи за контролните! Ако нямате ваш личен лаптоп, попитайте колеги или приятели да ви услужат.

Оценяване:

- Текущ контрол: 60%
- Финален изпит: 40%

Организация на курса

Мудъл сайт: <https://learn.fmi.uni-sofia.bg/course/view.php?id=5505>

Slack канал: <https://app.slack.com/client/TG8PZA4RF/CNHE981H7>

За контролни и задачи за самостоятелна работа ще ви трябва акаунт наименован “Фамилия_фн” в [hackerrank](#) (ако фамилията е по-дълга от 10 символа се допуска използване на “име_фн”)

Текущият контрол е важна част от процеса на усвояване на материала по Структури от данни и алгоритми и е задължителен за оформяне на оценка по предмета.

Очаквани предизвикателства (за преподавателския екип)

- Голям брой участници в курса 120 (+50-60 от минали години)
- Пропуски в знанията от предишните курсове по програмиране
- Неравномерно разпределение на студентите в групите за упражнения (поради присъствие на студенти от минали години или невъзможност на студент да присъства на упражненията на своята група)
- Проверката на $140 \times 8 = 1120$ контролни и $140 \times 14 = 1960$ задания за самостоятелна работа
- Мотивиране и ангажиране на всички участници в курса

Очаквани предизвикателства (за студентите в курса)

- Отделяне на достатъчно време за участие в курса 3+2+5 часа седмично
- Усвояване на материала
- Намиране на лаптоп за контролните по време на лекции
- Редовно участие в контролни и задачи за самостоятелна подготовка.
- Подаване на обратна връзка към преподавателският екип

За първата седмица използвайте този адрес:

<https://learn.fmi.uni-sofia.bg/mod/feedback/view.php?id=124812>

Учебни/Неучебни дни

Октомври						
п	в	с	ч	п	с	н
	1	2	3	4	5	6
7	8	9	10	11	12	13
14	15	16	17	18	19	20
21	22	23	24	25	26	27
28	29	30	31			

Ноември						
п	в	с	ч	п	с	н
				1	2	3
4	5	6	7	8	9	10
11	12	13	14	15	16	17
18	19	20	21	22	23	24
25	26	27	28	29	30	

Декември						
п	в	с	ч	п	с	н
						1
2	3	4	5	6	7	8
9	10	11	12	13	14	15
16	17	18	19	20	21	22
23	24	25	26	27	28	29
30	31					

Януари						
п	в	с	ч	п	с	н
		1	2	3	4	5
6	7	8	9	10	11	12
13	14	15	16	17	18	19
20	21	22	23	24	25	26
27	28	29	30	31		

Февруари						
п	в	с	ч	п	с	н
					1	2
3	4	5	6	7	8	9
10	11	12	13	14	15	16
17	18	19	20	21	22	23
24	25	26	27	28	29	

Конспект

1. Оценка на сложност за алгоритми.
2. Алгоритми за сортиране (Bubble, Selection, Insertion, Merge, Quick, Counting)
3. Алгоритми за търсене (Linear, Binary, Ternary Search)
4. Свързан списък. Специфични особености. Реализации.
5. Стек и опашка. Shunting-yard algorithm.
6. Дървета.
 - Двоично дърво.
 - Балансирани дървета. 2-3-4, Red-Black, AVL, Treap
 - Heap. Priority Queue
 - B-tree
7. Граф
 - Какво е граф и основни представяния и имплементации
 - Алгоритми за търсене в дълбочина и широчина в граф
 - Топологично сортиране
 - Цикъл в граф. Ойлеров и Хамилтонов цикъл в граф.
 - Алгоритми за Минимално покриващо дърво (Prim, Kruskal)
 - Търсене на най-кратък път в граф. Наивен алгоритъм. Алгоритъм на Dijkstra.

8.8 Хеширане

- Какво е хеш функция. Свойства на хеш функциите. Какво е колизия. Видове хеш функции. Как да се справяме с колизии. Приложения на хеширането.
- Структури от данни използващи хеш функции: Хеш Таблици и Хеш Мапове.
- Bloom Filters - идея, основни свойства и приложения в реалния свят.

Книги

- [Introduction to Algorithms](#), 3rd Edition (The MIT Press)Jul 31, 2009
by Thomas H. Cormen and Charles E. Leiserson
- [Algorithms](#) (4th Edition)Mar 19, 2011
by Robert Sedgewick and Kevin Wayne
- [The Algorithm Design Manual](#), Nov 5, 2010
by Steven S Skiena

Въведение

Какво е Алгоритъм?

- Последователност стъпки за решаване на проблем.

Какво е “Структура от данни”?

- Начин за организиране на данни във формат удобен за ползване.

За какво са ни необходими алгоритми и структури от данни ?

- За да можем да решим реални проблеми от заобикалящият ни свят

Трябва ли да уча алгоритми, за които знам, че вече има готови имплементации?

- Да, защото всеки алгоритъм, който научаваме ни развива мисленето и може да бъде основа за решаване на по-сложен проблем.

Сложност на алгоритми

- Какво е сложност на алгоритъм?
- Защо е необходимо да можем да оценим сложността на алгоритъм?
- Сложност по време / Сложност по памет
- Начини за определяне на сложност

Видове сложност на алгоритъм

- Сложност по време
- Сложност по памет

Видове оценка на сложност

- В най-лошия случай (песимистична)
Какъв е максималният възможен брой операции (единици памет), които могат да са нужни на алгоритъма, за да реши задачата?
- В най-добрия случай (оптимистична)
Какъв е минималният възможен брой операции (единици памет), които може да извърши (използва) алгоритъмът, за да реши задачата?
- В средния случай (средна)
Ако считаме, че всеки възможен вход е равновероятен, какво е “средното аритметично” на броя операции (единици памет), които трябва при всички възможни входове?
- При многократно изпълнение (амортизирана)
Ако алгоритъмът ще се извиква няколко пъти в рамките на дадена програма, колко операции (единици памет) средно ще са му необходими за едно извикване?

Начини за определяне на сложност

Сложност

Дефиниция

$$f(x) = O(g(x)) \quad \exists(N > 0) \exists x_0 \forall(x > x_0) (|f(x)| \leq N g(x))$$

$$f(x) = o(g(x)) \quad \forall(N > 0) \exists x_0 \forall(x > x_0) (|f(x)| \leq N |g(x)|)$$

$$f(x) = \Theta(g(x)) \quad \exists(N > 0) \exists(M > 0) \exists x_0 \forall(x > x_0) (N g(x) \leq f(x) \leq M g(x))$$

$$f(x) = \Omega(g(x)) \quad \exists(N > 0) \exists x_0 \forall(x > x_0) (f(x) \geq N g(x))$$

Какво всъщност означава това?

Big O - За големи стойности на входните данни каква функция може да се използва за да се ограничат отгоре броя на необходимите операции (пренебрегвайки константи с които може да се умножи функцията)

Big-O

$O(N!)$

$O(2^N)$

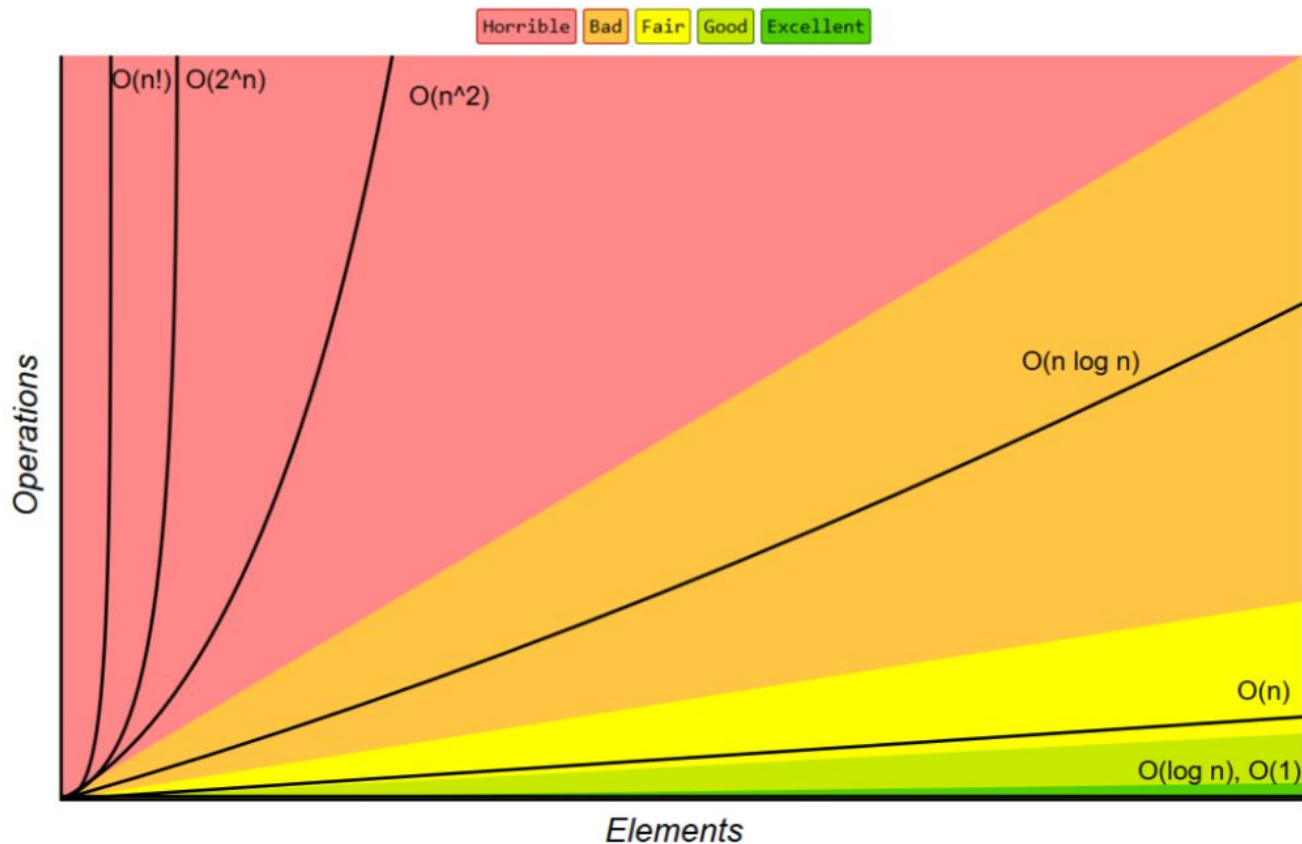
$O(N^2)$

$O(N)$

$O(N \log N)$

$O(\log N)$

$O(1)$



Big-O in practice

	1	2	3	10	100	1000	1 000 000
$O(1)$	1	1	1	1	1	1	1
$O(\log N)$	1	1	2	4	7	10	20
$O(N)$	1	2	3	10	100	1000	1 000 000
$O(N \log N)$	1	2	5	34	665	9964	19 931 569
$O(N^2)$	1	4	9	100	10 000	1 000 000	10^{12}
$O(2^N)$	1	4	8	1024	$> 10^{30}$	$> 10^{301}$	$> 10^{301029}$
$O(N!)$	1	2	6	3 628 800	$> 10^{157}$	$> 10^{2567}$	$> 10^{5565708}$

Примери

Константна сложност - знаем точният отговор и го връщаме веднага.

Логаритмична сложност - данните, които трябва да претърсим са разделени, така че търсейки на всяка стъпка изхвърляме половината от възможностите.

Линейна сложност - обхождаме всички данни и връщаме намереният резултат.

Как да измерим сложността на алгоритъм?

- С оценка на сложностите на различните компоненти по-брой операции/памет
- Емпирично
 - Увеличаваме линейно големината на входните данни и измерваме време/памет

Пример

Задача 1: В подаден масив със числа намерете дали се среща точно определено число.

[1, 4, 3, 2,  5, 7, 6] , $x=5$

Решение 1

// лекциите и примерите в тях няма да са обвързани с конкретен език за програмиране! На упражненията езика за програмиране ще е конкретен - C++!

```
boolean isFound(int[] arr, int x){  
    boolean found = false;  
    for( int i = 0 ; i < arr.length ; i++){  
        if(arr[i] == x){  
            found = true;  
        }  
    }  
    return found;  
}
```

Решение 2

```
boolean isFound(int[] arr, int x){  
    boolean found = false;  
    for( int i = 0 ; i < arr.length ; i++){  
        for(int j = 0 ; j < arr.length;j++){  
            if(arr[j] == x){  
                found = true;  
            }  
        }  
    }  
    return found;  
}
```


Решение 3

```
boolean isFound(int[] arr, int x){  
    boolean found = false;  
    for( int i = 0 ; i < arr.length ; i++){  
        for(int j = i ; j < arr.length;j++){  
            if(arr[j] == x){  
                found = true;  
            }  
        }  
    }  
    return found;  
}
```

Решение 4

```
boolean isFound(int[] arr, int x){  
    boolean found = false;  
    for( int i = 0 ; i < arr.length ; i+=10){  
        for(int j = i ; j < arr.length;j++)  
            if(arr[j] == x){  
                found = true;  
            }  
        }  
    }  
    return found;  
}
```

Кое решение да ползваме ?

- Всички те решават задачата мигновено за всеки масив, който въвеждаме от конзолата...
- Решение 4 изглежда най-сложно, може би е най-доброто ?
- Решение 1 изглежда най-просто, но дали е най-бързото все пак в първият цикъл на Решение 4 имаме доста по-бързо растящ брояч
- За да разберем кое решение е по-добро следва да направим оценка за сложността му.

Пример 2

Имаме масив с числа, някой от които се повтарят - търсим кое числото, което се среща най-често в масива

Решение 1

```
int most_frequent = arr[0];
int frequency = 1;
for(int i = 0; i < arr.length ; i++){
    int i_freq = 0;
    for(int j = 0 ; j < arr.length; j++){
        if([arr[i]==arr[j]){
            i_freq++;
        }
    }
    if(i_freq > frequency){
        frequency = i_freq;
        most_frequent = arr[i];
    }
}
```

Решение 2

```
for(int i = 0; i < arr.length;i++){
    for(int j = i+1 ; j < arr.length; j++){
        if(arr[i]>arr[j]){ swap(arr,i,j); }
    }
}
int most_frequent = arr[0]; int max_frequency = 1;
int current = arr[0]; int current_frequency = 1;
for(int i = 1 ; i < arr.length; i++){
    if(current != arr[i]){
        if(current_freq > max_frequency){
            max_frequency = current_frequency;
            most_frequent = arr[i-1];
        }
        current = arr[i];
        current_frequency = 0;
    }
    current_frequency++;
}
```

Кое решение е по-добро?

- Първото понеже е по-кратко?
- Второто понеже е с “по-кратък” вътрешен цикъл?
- ?

Каква е сложността на примера?

```
long mysteryFunc( int n) {  
    if(n == 0) {  
        return 0;  
    }  
    if(n == 1) {  
        return 1;  
    }  
    return mysteryFunc(n-1) + mysteryFunc(n-2);  
}
```

Отговор: $O(2^n)$

Да се запомни!

За оценка с нотацията голямо O използваме само най-голямата функция и премахваме всички константи

Примери:

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

$$O(8n+4n^3 + 7) = O(n^3)$$

$$O(n^n + 999 \cdot n^{999}) = O(n^n)$$

Изводи

- За да може да сравним различни решения на една задача трябва да може да оценим сложността на решението.
- Сложността на алгоритъм, най-често оценяваме сложността в най-лошият случай с Big O нотацията.
- При оценяване с Big O нотацията, премахваме всякакви константи и свободни членове от функциите.
- $O(1) < O(\log(n)) < O(n) < O(n \cdot \log(n)) < O(n^2)$

Следващи стъпки

1. Входящ тест в модул
2. Задачи за самостоятелна подготовка (очаквано време за работа 3-5 часа. **Включват се в крайното оценяване!**). К След крайният срок решенията ще се публикуват и няма да се приемат закъснели решения.
3. Упражнения по СДА (Използвайте ефективно часовете за да разберете всичко, което е останало неясно от лекции и да започнете със задачите за самостоятелна подготовка)
4. Подготовка за контролно на 22.10.2018 (Оценяване на сложност на алгоритми, сравнение на алгоритми, алгоритми за сортиране.)

Важно!

Контролните и задачите за самостоятелна работа са изключително важен процес за развиване на уменията за решаване на алгоритмични проблеми, не се допуска никакво преписване! Ако някой има нужда от помощ, може да му помогнете с обяснение как да си реши задачата(извън залата за контролно), но не и да му давате готово код за решението на контролно или домашно.

Време за входящ тест/анкета!

Може да направите теста от лаптоп или мобилен телефон.