1 Сложност

1.1 Сложност по време

Какъвто и проблем да решаваме рано или късно стигаме до използването на някакъв алгоритъм. След като измислим нашия алгоритъм естествено трябва да си отговорим на въпроса "Този алгоритъм ефикасен ли е?". Дали няма някой алгоритъм, който поради една или друга причина ще работи по - бързо от текущия? А всъщност какво разбираме под "по - бързо"?
В случая не става въпрос за физическо време. Една от грешките при анализа е бързината на алгоритъма да се обърка с производителността на машината. Колко бързо се изпълнява един алгоритъм във физическо време зависи от много фактори, като например върху какъв хардуер го изпълняваме, какви процеси се изпълняват, какъв е компилаторът, как компилираме и тн. Поради тази причина е доста по - важно да разгледаме как се държи този алгоритъм с увеличение на сложността на задачата. Мерим сложността в дискретно време.

1.1: Сложност по време

Сложността по време е функция по големината на входа.

Неформално се стремим да дадем някаква оценка на алгоритъма спрямо броя елементарни операции които той извършва. Обикновено не е необходимо да се намери точния брой операции, най - важното е как броя на операциите се изменя с увеличаване на размера на задачата.
Нека с n означаваме големината на входа. От обща култура вероятно вече сте виждали означения от типа $O(n)$ или $\Theta(n^2)$, но едва ли сте виждали $O(5$ минути и 32 секунди$)$.
Когато разглеждаме сложността по време на алгоритмите разглеждаме три случая:

1. Най - добър случай (Best case)
2. Среден случай (Avarage case)
3. Най - лош случай (Worst case)

Най - ценна оценка е средната сложност. Средното време за работа обаче често се изчислява сложно, та ще се концентрираме върху най - лошия случай.
Разбирането на сложността е ценно при работата със структури от данни. Познаването на сложностите ни помага да решим дали да използваме дадената структура от данни или да изберем друга, по - подходяща за дадената задача.

1.2 Асимптотични нотации

Разбрахме, че се опитваме да дадем някаква оценка на броя примитивни операции които нашия алгоритъм извършва. Използвайки асимптотичните нотации можем да изразяваме сложността на алгоритмите приблизително.

1.2: Асимптотична нотация Θ

Нека g(n) е функция.
$\Theta(g(n)) = \{f(n) | \exists c_1, c_2 > 0 \exists n_0 > 0 \forall n \geq n_0 : 0 \leq c_1 * g(n) \leq f(n) \leq c_2 * g(n)\}$

1.3: Асимптотична нотация O

Нека g(n) е функция.
$O(g(n)) = \{f(n) | \exists c > 0 \exists n_0 > 0 \forall n \geq n_0 : 0 \leq f(n) \leq c * g(n)\}$

Какво всъщност е Θ? Поглеждайки дефиницията разбираме, че това е безкрайно множество от функции. Неформално за $f(n)$ твърдим, че когато n стане достатъчно голямо стойността ѝ е поне $c_1 * g(n)$, но не повече от $c_2 * g(n)$ за някакви положителни константи c_1 и c_2. За малки стойности на n не се интересуваме какво се случва, но веднъж n стане ли достатъчно голямо то стойността трябва да остава в тази граница. Ако искаме да кажем, че някаква функция принадлежи на множеството е прието да пишем $f(n) = \Theta(g(n))$ а не $f(n) \in \Theta(g(n))$.
За Θ можем да мислим като за точна граница, докато за O мислим като горна граница на сложността. Тоест, работейки с O за $f(n)$ твърдим, че след някое достатъчно голямо n сложността ѝ не надминава $c * g(n)$ за някоя положителна константа c. Съществуват и други асимптотични

нотации, но тях ще оставим за друг курс.
В анализа на сложността се прави следното

1.4: допускане

Ако изразът за сложност на даден алгоритъм е сума, разглеждаме най - бързо растящата функция измежду събираемите, в асимптотичния смисъл.

Примерно $\frac{n^2}{16} + 11n + 17 = O(n^2)$.
Също, при асимптотичен анализ можем да игнорираме константите.
Примерно $\frac{n^2}{2} = O(n^2)$

1.3 Примери

За да стане по - ясно ще разгледаме няколко алгоритъма и ще определим неформално тяхната сложност. Нека първо разгледаме алгоритъма *линейно търсене*.

```
bool linearSearch(const std::vector<int>& v, int elem) {
    for(int i = 0; i < v.size(); i++)
        if(v[i] == elem)
            return true;
    return false;
}
```

Каква е сложността на този алгоритъм?

1. Best case: Най - добрия случай е търсения елемент да е в началото. Тогава винаги правим константен брой примитивни операции. Следователно сложността е $O(1)$. Опитайте се да кажете колко са тези операции. Защо сложността е $O(1)$?

2. Worst case: В най - лошия случай елемента го няма. Ако допуснем, че елемента винаги е там то той ще е най - накрая. Длъжни сме да обходим целия масив. Колко проверки правим? Отговорът е, че проверките зависят от гореминатa ва вектора! Имаме сложност $O(n)$ в най - лошия случай.

Разглеждаме следния алгоритъм:

```
size_t sumPowersOfTwo(size_t n) {
    size_t result = 0;

    for(int i = 0; i <= n; i++)
        result += (1 << i);          // 2 ^ i

    return result;
}
```

Алгоритъмът връща сумата на всички степени на двойката. С нарастване на n цикъла for отговорен за сумирането прави все повече и повече итерации. В цикъла имаме само константна работа така, че сложността на този алгоритъм е $O(n)$. Можем ли да се справим по - добре?
Математиката ни казва, че можем. Знаем, че $\sum_{i=0}^{n} 2^i = 2^{n+1} - 1$. Ако това твърдение не е очевидно, можем да си представим събираемите като числа в двоична бройна система. Събираемото на позиция k има единствена единица на позиция k. Тоест сумата ще има единици на позиция [0, 1...n]. Добавете единица и проверете какво се случва.
Оказва се, че нашата сума може да се напише по следния начин:

```
size_t sumPowersOfTwo(size_t n) {
    return (1 << (n + 1)) - 1;
}
```

Каква е сложността на този алгоритъм? Ами той се състои от един ред. На този ред имаме едно събиране с единица един bitshift и едно изваждане на единица. Никоя от тези операции не става по - бавна с нарастване на n обаче. Интуитивно доказахме, че този алгоритъм има сложност $O(1)$. Кой алгоритъм ще бъде по - бърз? Очевидно константната версия на алгоритъма ще е по - бърза.

В последния пример ще разгледаме алгоритъм, който брои всички начини да изберем двама човека измежду n човека където наредбата не е от значение. Първата идея която ни хрумва може би е следната

```
size_t countAllPairs(size_t n) {
    size_t toReturn = 0;
    for(int i = 0; i < n; i++){
        for(int j = i + 1; j < n; j++){
```

```
            ++toReturn;
        }
    }
    return toReturn;
}
```

Първия човек групираме с всички, след това втория човек групираме с всички без първия и тн. Този алгоритъм е коректен. Каква е неговата сложност? При $i = 0$ алгоритъма прави $n-1$ стъпки. След това при $i = 1$ алгоритъма прави $n - 2$ стъпки. Тоест очевидно е, че общо алгоритъма прави $\sum_{i=0}^{n-1} n - i - 1$ стъпки. Това е еквивалентно на $\sum_{i=1}^{n-1} i$ или както знаем от математиката това е $\frac{n(n-1)}{2} = \frac{n^2}{2} - \frac{n}{2} = O(n^2)$.
След тези разсъждения константния алгоритъм решаващ задачата трябва да е очевиден.

```
size_t countAllPairs(size_t n) {
    return n * (n-1) / 2;
}
```

Този алгоритъм извършва едно изваждане на единица едно умножение и едно деление на две. Никоя от тези операции не става по - бавна с нарастване на входа т.е. можем да заключим, че имаме алгоритъм с константна сложност!

1.4 Сложност по памет

Сложността по памет е количеството памет (динамична или статична), което алгоритъмът използва за да работи. В примерите от горната секция всички алгоритми използват константна памет. Пример за алгоритъм, който използва $O(n)$ памет е merge sort. За да слеем два сортирани масива в линейно време заделяме допълнително памет.
Алгоритми, които работят с константна допълнителна памет, се наричат **in-place** алгоритми.

1.5 Задачи за упражнение

1. Разгледайте следните сложности: $O(1), O(n^2), O(2^n), O(\log(n)), O(n!), O(n)$. Опитайте се да ги наредите от най - добра до най - лоша.