Основи програмирања

Предавање 11 - сложеност алгоритама

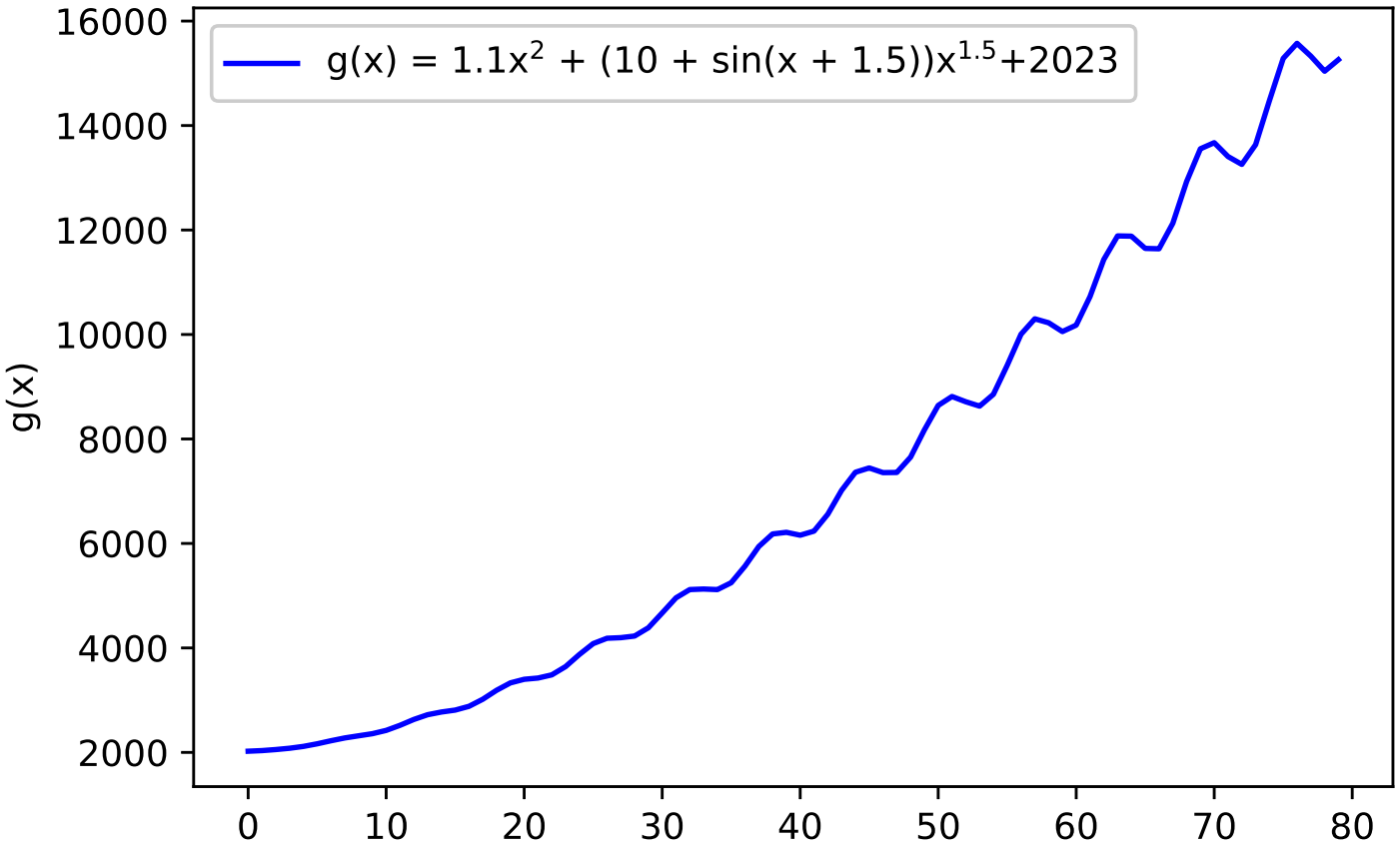
Лазар Илић
iliclazar15@gmail.com

Факултет инжењерских наука
Универзитет у Крагујевцу

децембар 2023.

Садржај

Асимптотска анализа



За функцију $g(x)$ кажемо следеће: $g(x) = \theta(x^2)$.

Асимптотска анализа

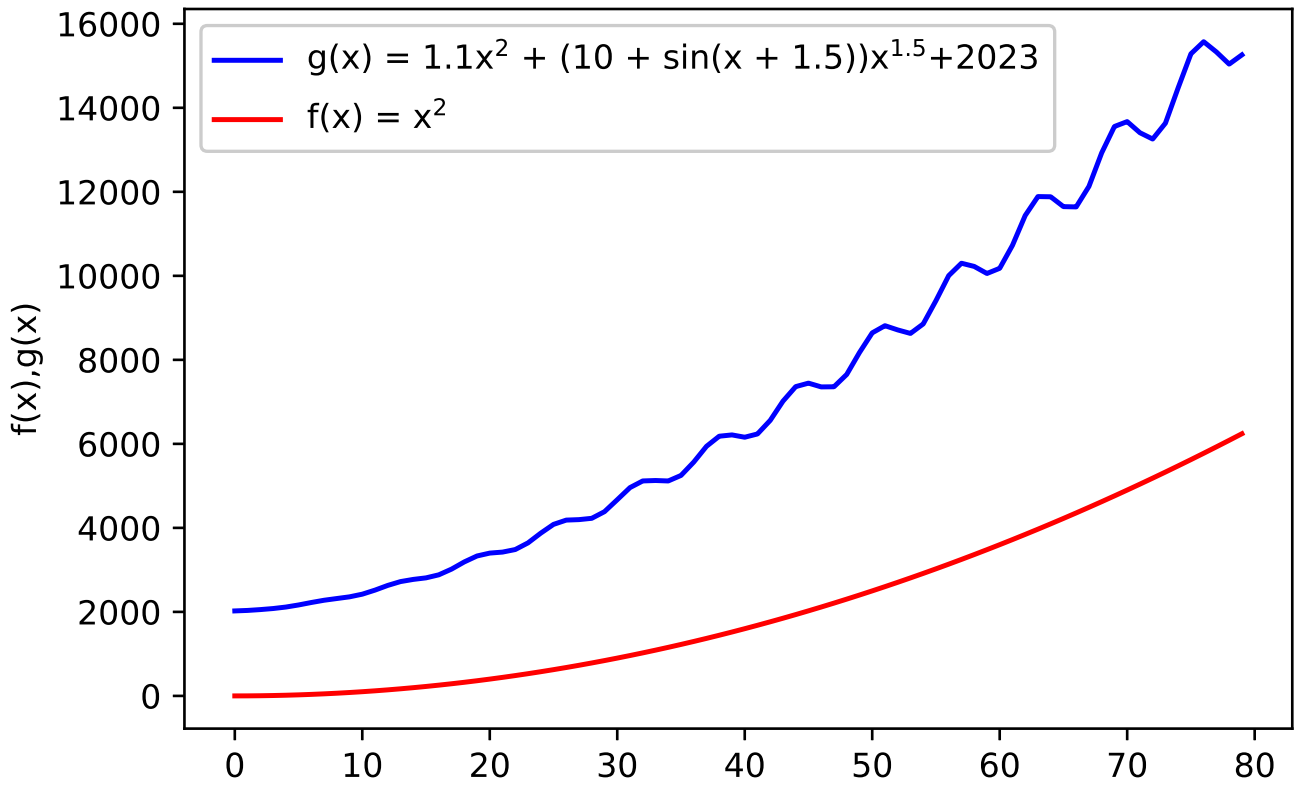
На шта се односи θ? Овде говоримо о *ограничењима* функције, и то:

- Ω - нотација за доњу границу асимптотског раста функције,
- $\mathcal{O}$ - нотација за горњу границу асимптотског раста функције,
- θ - нотација за обострано ограничење (и горња и доња граница) асимптотског раста функције.

Зашто баш x^2?

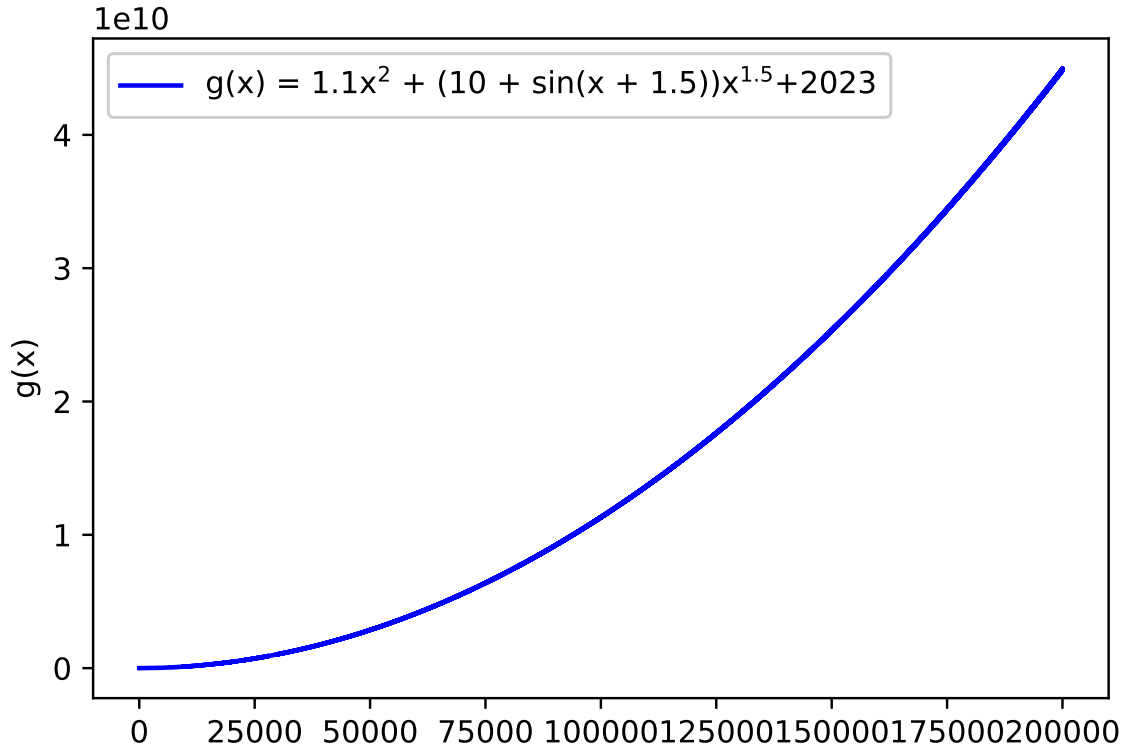
- Посматрати само **доминантан** члан функције,
- Занемарити мултипликативне константе које стоје уз доминантан члан функције.

Зашто баш x^2?



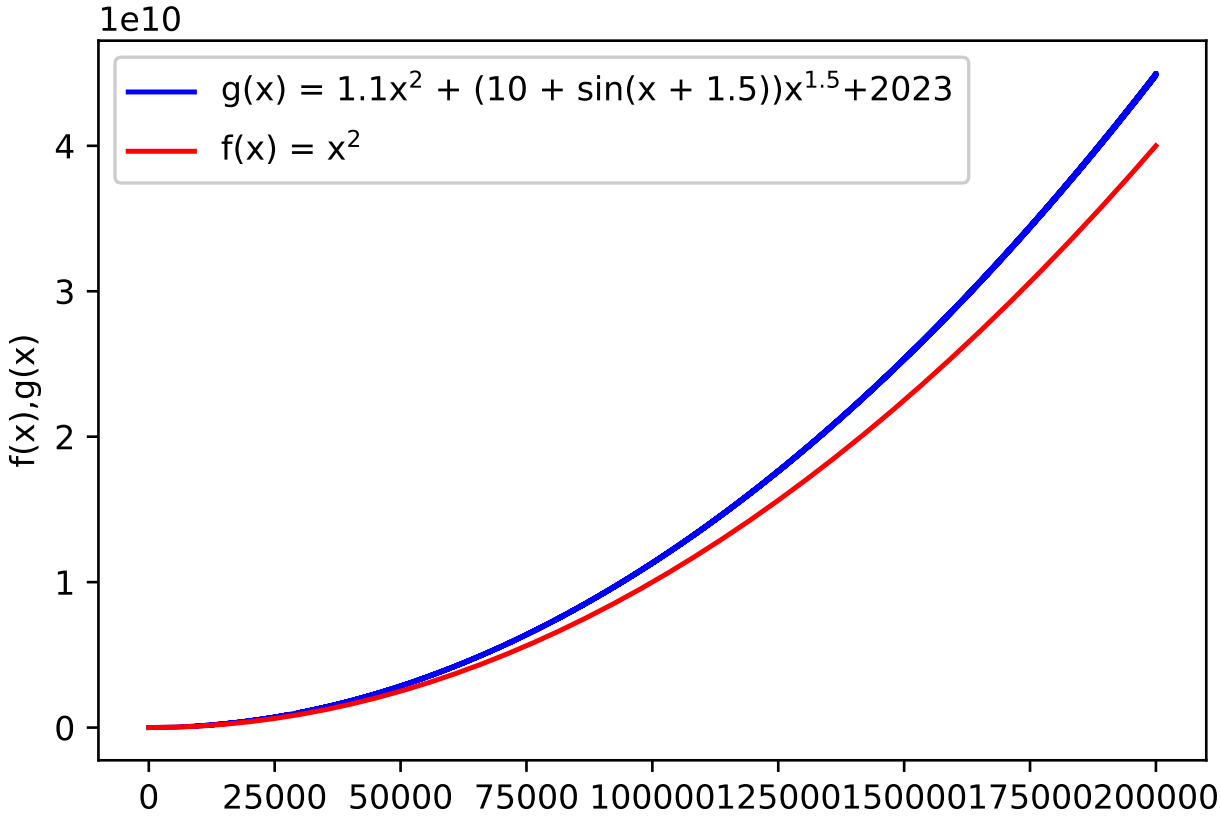
На основу овог графика и нема неког смисла наша одлука да изаберемо баш x^2. Кључ асимптотске анализе је у томе што је потребно посматрати понашање функције $g(x)$ када $x \to \infty$!

Други поглед на функцију $g(x)$



Када "одзумирамо" график, увиђамо да се мање примећује таласаста природа функције $g(x)$ (узрокована синусом који је део функције), те да наша функција сада више "личи" на x^2, и то је управо зато што x^2 и јесте **најдоминантнији** члан функције.

Други поглед на функције $g(x), f(x)$



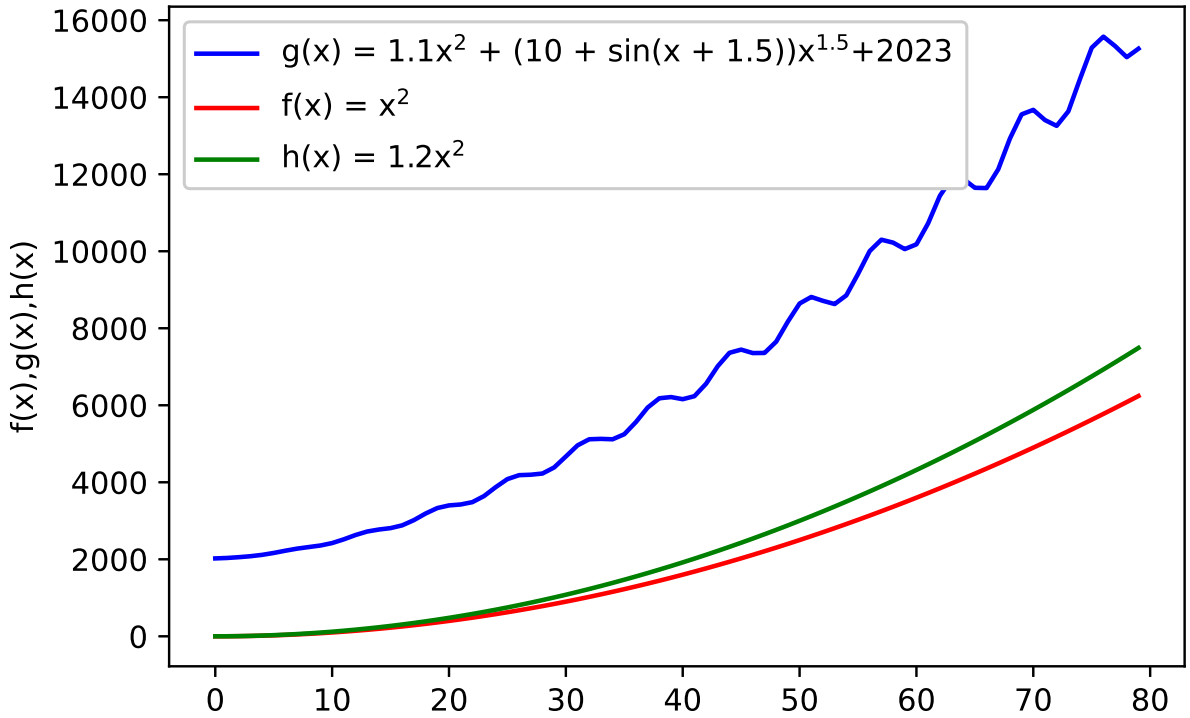
Видимо да се $f(x)$ на графику налази испод $g(x)$, и рекли смо да функције "личе" једна на другу, при чему би ове функције биле још "ближе" једна другој када $x \to \infty$.

Доња граница асимптотског раста функције $g(x)$

Сада можемо рећи следеће: $g(x) = \Omega(f(x))$.
Овај закључак је донет на основу чињенице да се $f(x)$ понаша као доња граница функције $g(x)$ када $x \to \infty$.

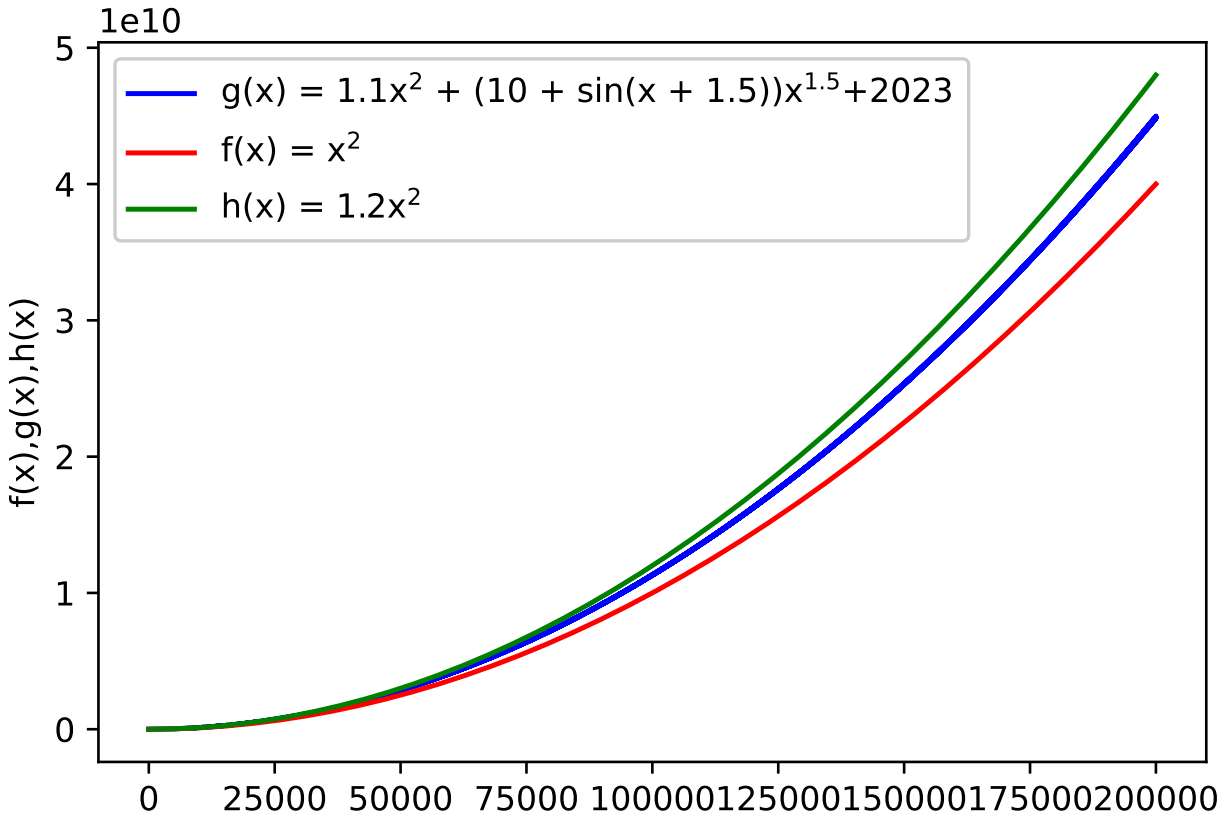
Међутим, првобитно је речено следеће: $g(x) = \theta(f(x))$. Овим се имплицира да се функција $f(x)$ истовремено понаша и као доња и као горња граница асимптотског раста функције $g(x)$.

Трећи поглед на функцију $g(x)$



Сада уводимо и функцију $h(x)$ која ће представљати горњу границу за нашу функцију $g(x)$. Изабрана је специфично функција $1.2x^2$ јер је њен фактор уз x^2 мало већи од фактора који се налази у $g(x)$ (1.1).

Шири поглед на функције $g(x), f(x), h(x)$



На "одзумираном" графику видимо да се функција $h(x)$ налази изнад $g(x)$, и да све три функције међусобно "личе".

Горња граница асимптотског раста функције $g(x)$

Сада можемо рећи следеће: $g(x) = \mathcal{O}(h(x))$.
Слично примеру са доњом границом, овај закључак је донет на основу чињенице да се $h(x)$ понаша као горња граница функције $g(x)$ када $x \to \infty$.

И даље нисмо дошли до онога што првобитно је речено:

$$g(x) = \theta(f(x)).$$

Видели смо да смо имали 2 различите функције које су ограничавале $g(x)$, како тачно онда можемо користи једну функцију уз θ?
Моћ асимптотске анализе/нотације лежи у томе што нам дозвољава да занемаримо мултипликативне константе које стоје уз доминантан члан функције.
Дакле, иако функције $f(x), h(x)$ јесу различите, оне се разликују само у коефицијенту који стоји уз x^2 (1.2 и 1). Из тог разлога, ми можемо рећи да је функција $g(x)$ **обострано** ограничена функцијом x^2, односно $g(x) = \theta(f(x))$.

Дискусија - Ω или $\mathcal{O}$ или θ

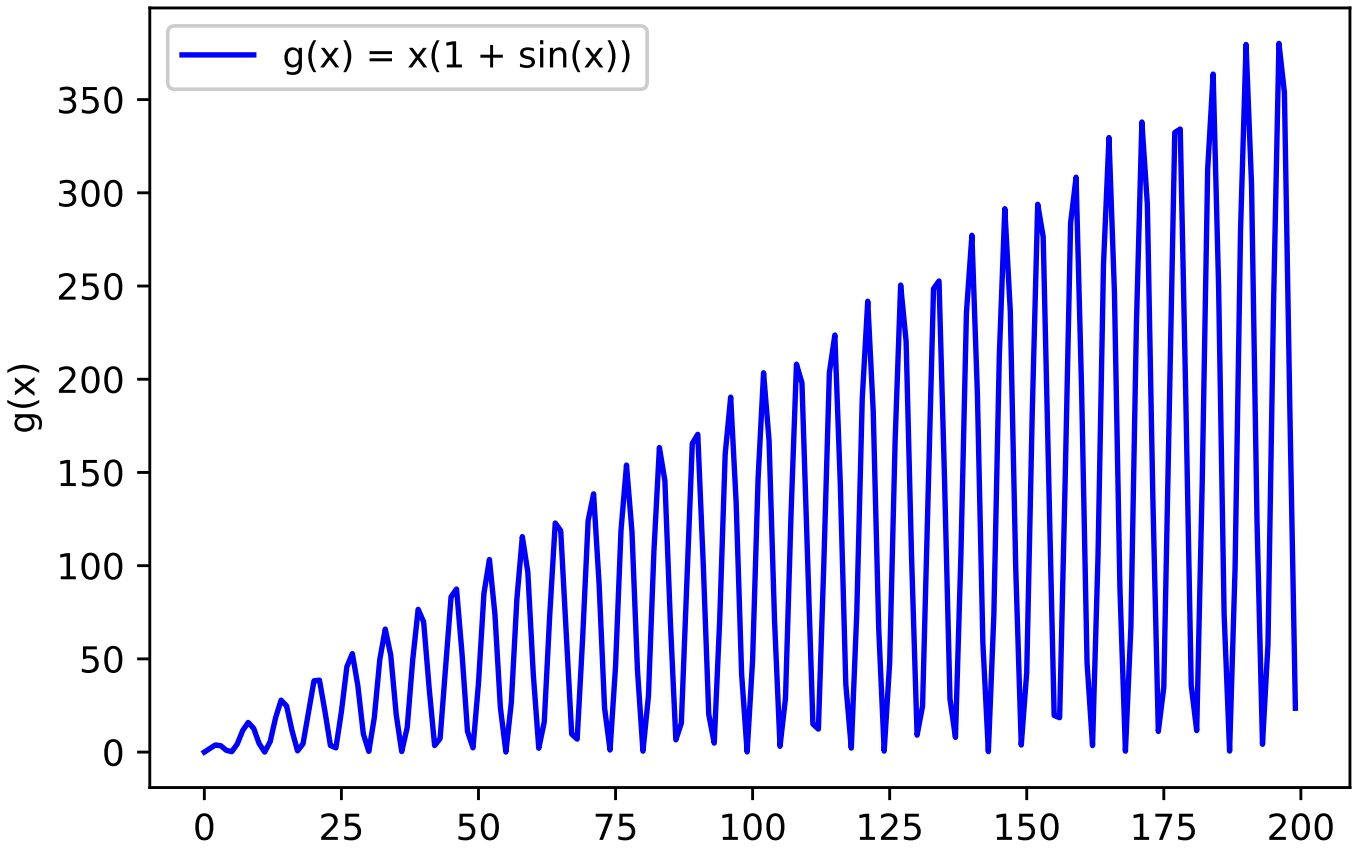
Која граница је најбитнија?

У програмирању, и инжењерингу генерално, потребно је увек посматрати **најгори** случај!

С тим у вези, можемо рећи, са становишта алгоритама најзначајнија нам је **горња** граница, односно нотација са $\mathcal{O}$.

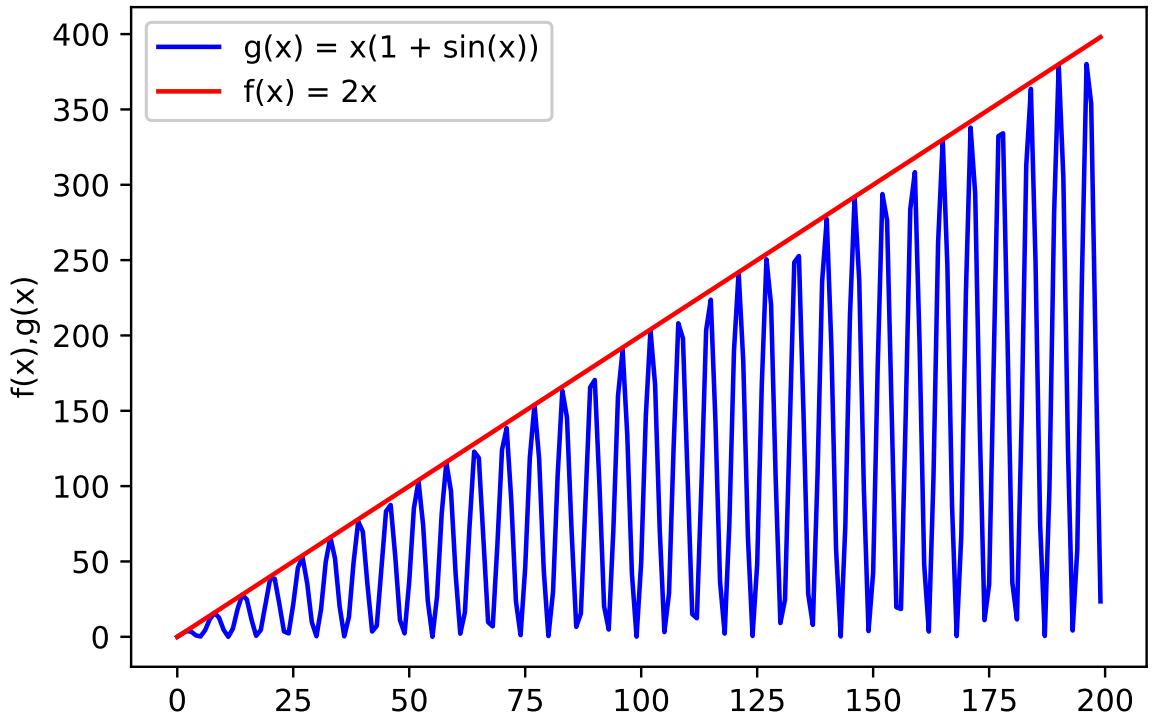
Неретко се у литератури, или по форумима из области програмирања користи $\mathcal{O}$ при чему се мисли на θ, из разлога што људе мрзи да изговарају/пишу тета, као и због тога што се за добар део проблема поклапају горња и обострана граница. Уједно смо рекли да је нама свакако од интереса **горња** граница.

Специфични случај за Ω



Како бисте ограничили $g(x)$ у овом случају?

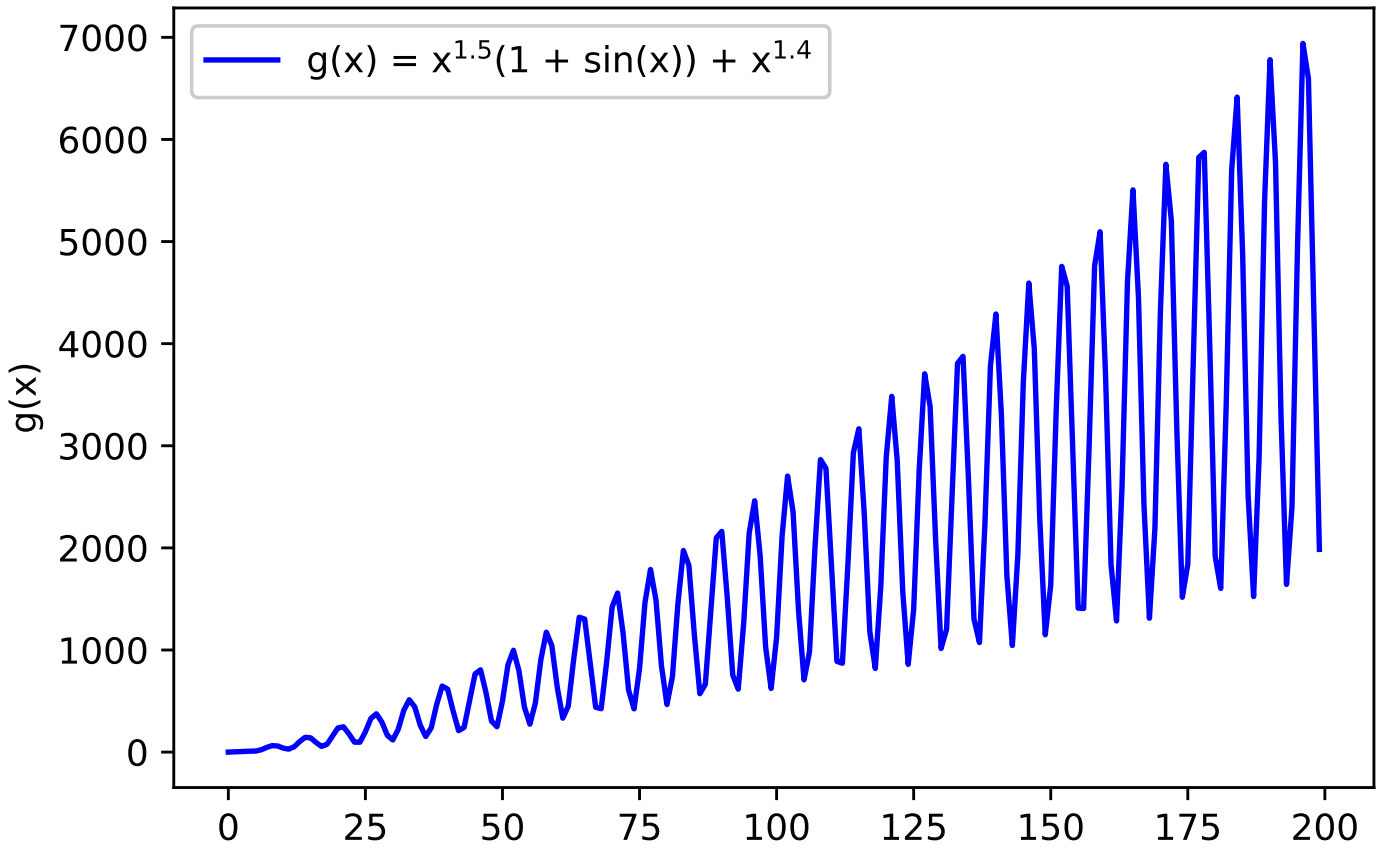
Специфични случај за Ω - наставак



Горња граница за ову функцију може бити $2x$ (обзиром на то да је максимална вредност синуса 1, те ће коефицијент уз x бити максимално 2).
Доња граница би била 0 (ова функција бесконачно много пута у свом домену достиже вредност 0), међутим ово није валидна вредност за ограничење функције (аналогно томе, није валидно узети бесконачност за горњу границу).

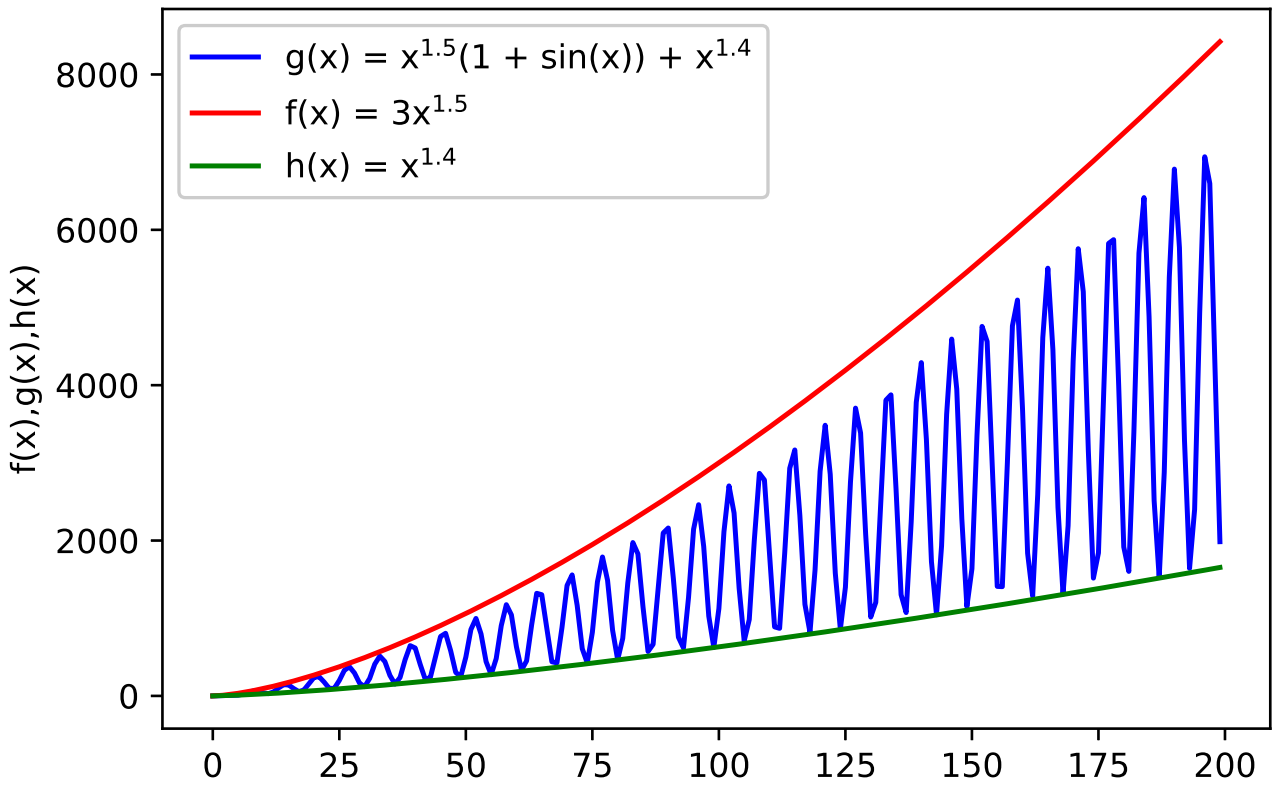
Шта је са θ?

Специфични случај за θ



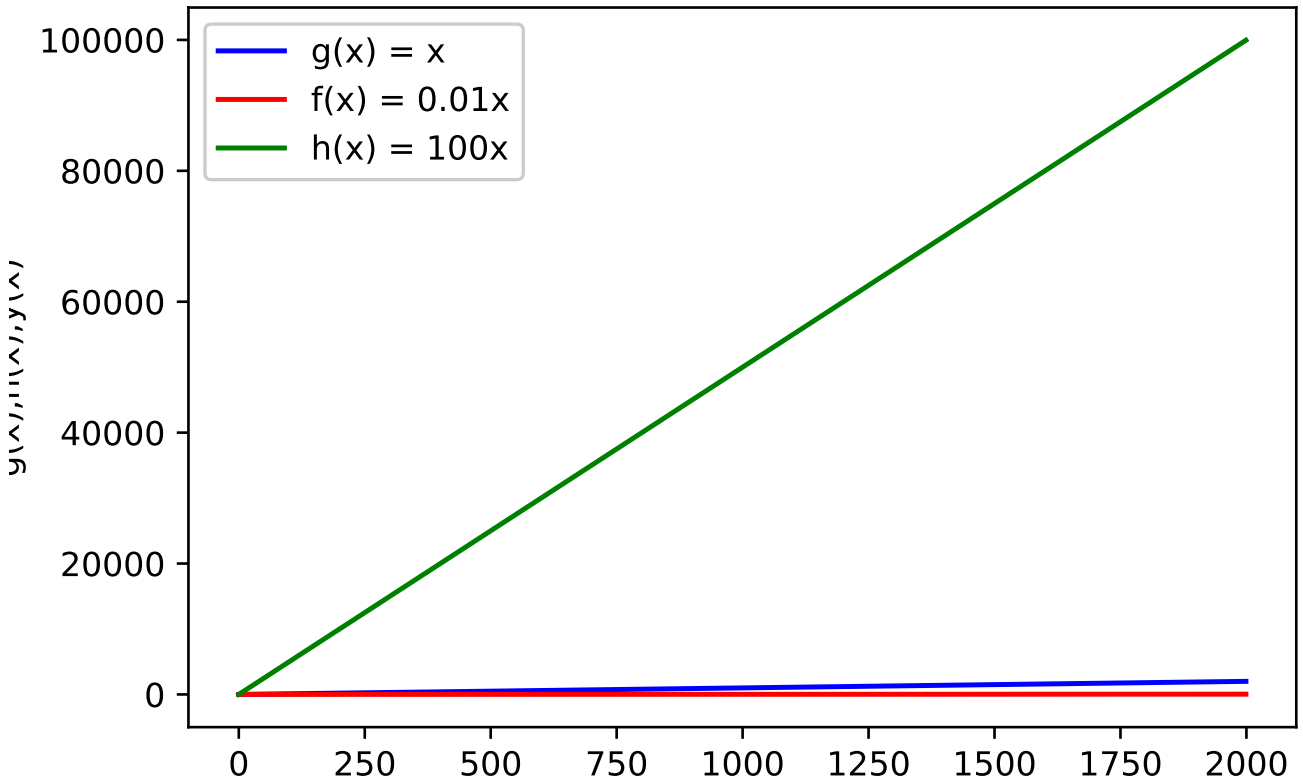
Како бисте ограничили $g(x)$ у овом случају?

Специфични случај за θ - наставак



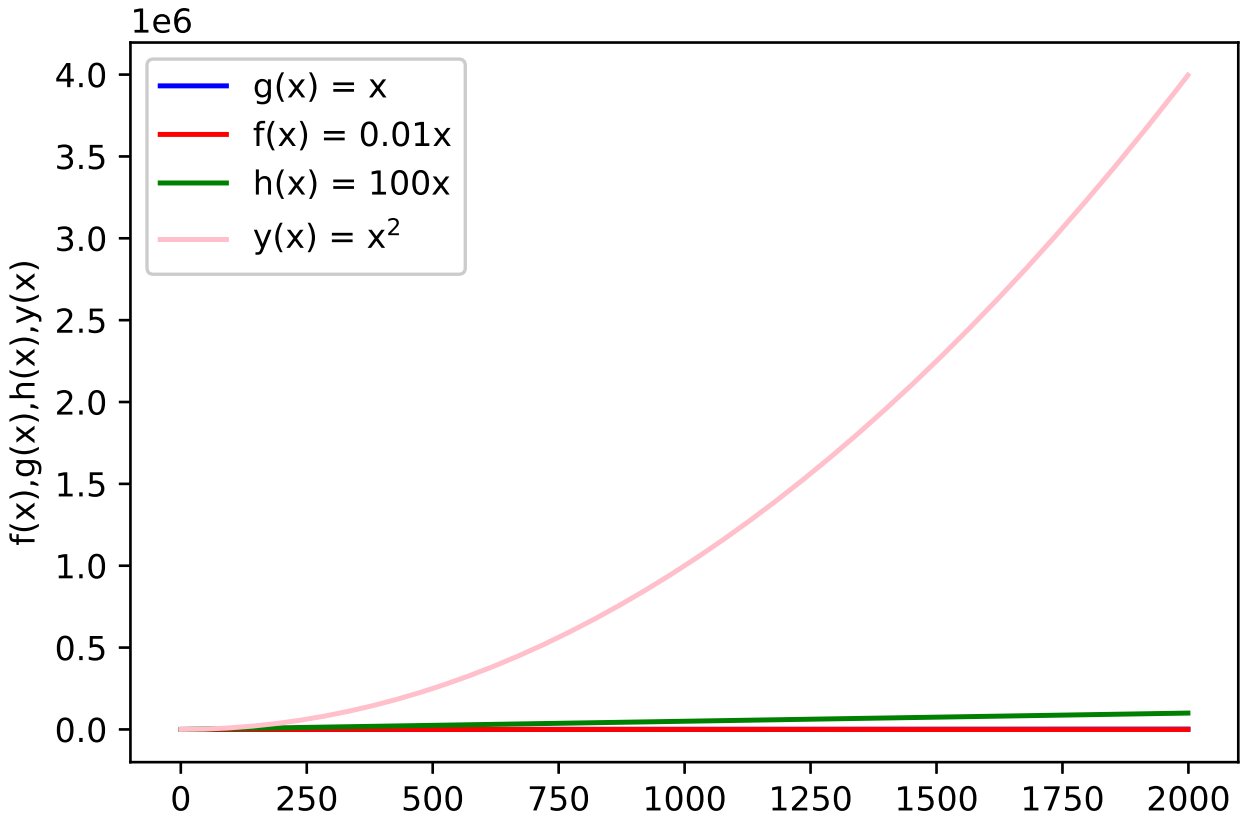
У овом случају се горња и доња граница разликују, али не само у мултипликативним константама, већ постоји разлика у експонентима x-ева, стога не можемо да дефинишемо θ за ово $g(x)$!

Појашњење - занемаривање мултипликативних константи



Евидентно је да постоји разлика у датим графицима функција, иако се оне разликују "само" у мултипликативним константама, како их онда можемо тек тако занемарити, као што налаже асимптотска анализа/нотација?

Појашњење - занемаривање мултипликативних константи



Као што је већ речено, неопходно је посматрати понашање функција када $x \to \infty$, а у тим случајевима на раст функције много више утиче доминантан члан функције него мултипликативна константа уз њега (јер важи да је $x >> c$).

Сажетак - асимптотска анализа функција

Потребно је запамтити следеће нотације за ограничење функција:

- Ω - нотација за доњу границу асимптотског раста функције,
- $\mathcal{O}$ - нотација за горњу границу асимптотског раста функције,
- θ - нотација за обострано ограничење (и горња и доња граница) асимптотског раста функције.

Најбитнија је **горња** граница, јер је од интереса посматрати **најгори** случај; дакле фокусираћемо се надаље на $\mathcal{O}$.

Фокусирати се на доминантан члан функције, и занемарити мултипликативне константе које стоје поред њега.

У наставку ћемо упоредити неке од сложености са којима се најчешће срећемо у програмирању.

Садржај

Како применити асимптотску анализу на алгоритме?

Видели смо да је асимптотска анализа моћна алатка која нам олакшава анализе наизглед компликованих функција, али како да то применимо на алгоритме?
Уједно, како уопште "мерити" ефикасност програма/алгоритма?
Прво и основно, неопходно је да алгоритам ради **тачно**, односно да на крају извршавања враћа тачан резултат, али постоје случајеви где је јако битна и брзина извршавања (аутопилот за авион/хеликоптер, или системи у аутомобилима са аутономним режимом вожње,...).
Поред потрошње у виду времена, не треба занемарити ни количину меморије коју програм троши! Битно је напоменути да је циљ направити компромис између читљивости и ефикасности самог кода.
Ако нас интересује време извршавања, ми можемо користити штоперицу, и тако пратити време које је потребно да се програм изврши. Да ли је ово добро, и зашто не?
Овакав начин би занемарио бројне факторе који утичу на брзину извршавања, попут брзине рачунара, варијације међу интерпретаторима/преводиоцима језика, као и различите вредности улазних података; што би довело до различитих резултата за исти алгоритам!

Идеја - универзална метрика

Посматраћемо рачунар као **машину са случајним приступом(МСП)**, и под тим моделом израчунавања претпостављамо да се:

- сви кораци извршавају секвенцијално један за другим.
- корак је операција која захтева константно време:
  - доделе вредности
  - поређења
  - аритметичко-логичке операције
  - приступ објектима у меморији

Поделом на ове основне кораке, можемо поредити ефикасност алгоритама који решавају исти проблем!
Ово такође гарантује исте резултате независно од хардвера рачунара, као и преводиоца/интерпретатора, али остаје зависност од дужине улазних променљивих процедуре, или како се чешће назива - величине проблема (N).

Најчешће виђене класе сложености алгоритама

За улаз процедуре N и константу k запис:

- $\mathcal{O}(1)$ — означава константно време извршавања
- $\mathcal{O}(\log N)$ — означава логаритамско време извршавања
- $\mathcal{O}(N)$ — означава линеарно време извршавања
- $\mathcal{O}(N \log N)$ — означава логлинеарно време извршавања
- $\mathcal{O}(N^2)$ — означава квадратно време извршавања
- $\mathcal{O}(N^3)$ — означава кубно време извршавања
- $\mathcal{O}(N^k)$ — означава полиномијално време извршавања
- $\mathcal{O}(k^N)$ — означава експоненцијално време извршавања

Пар примера асимптотске анализе

Размотрићемо неколико функција које зависе од величине проблема (N), а затим их сврстати у одговарајућу класу сложености.

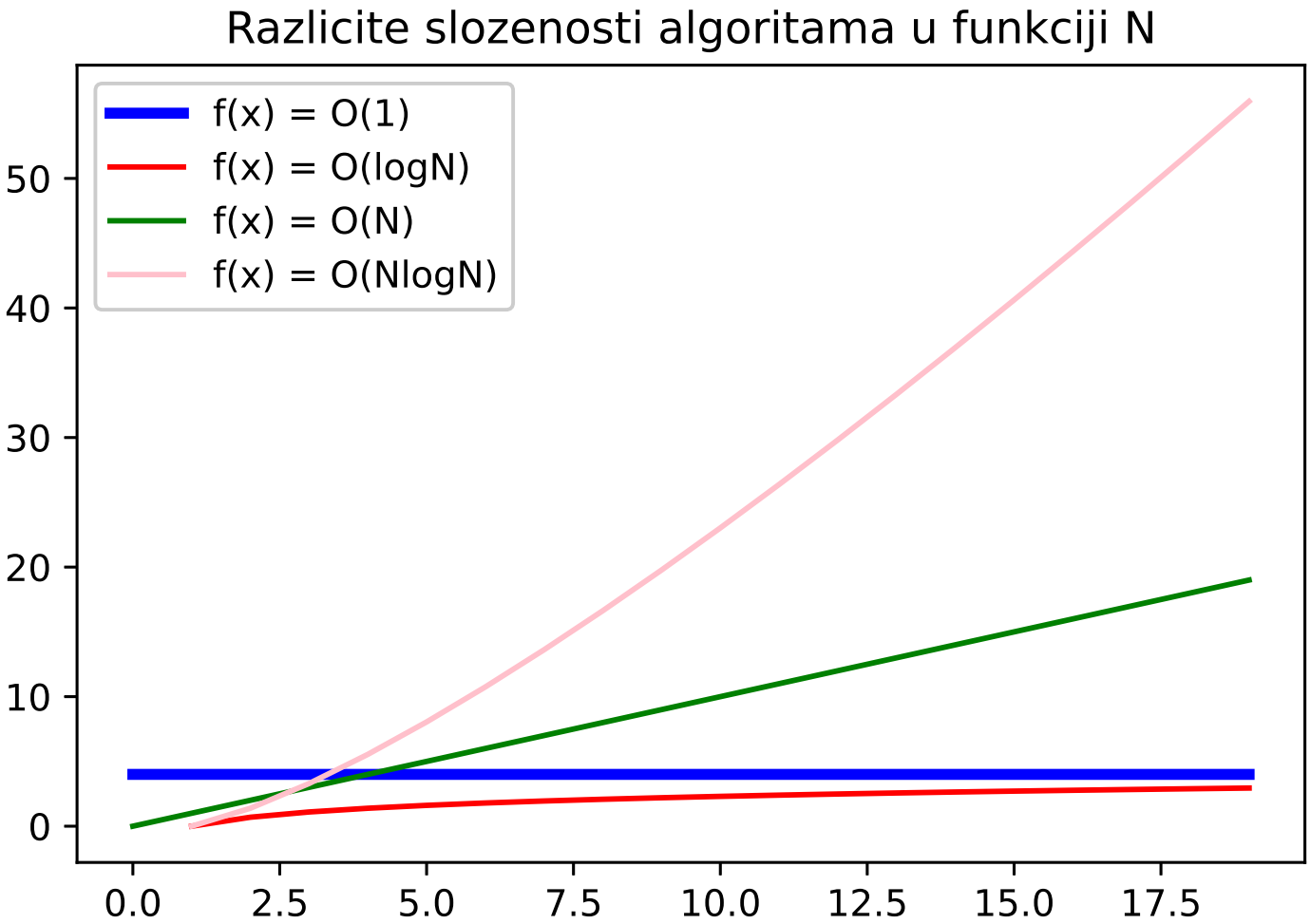
$f(N)$:

- 10^{80}
- $(20N)^7$
- $log\, N^{100}$
- $(1 + 2 + 3 + ... + N)$

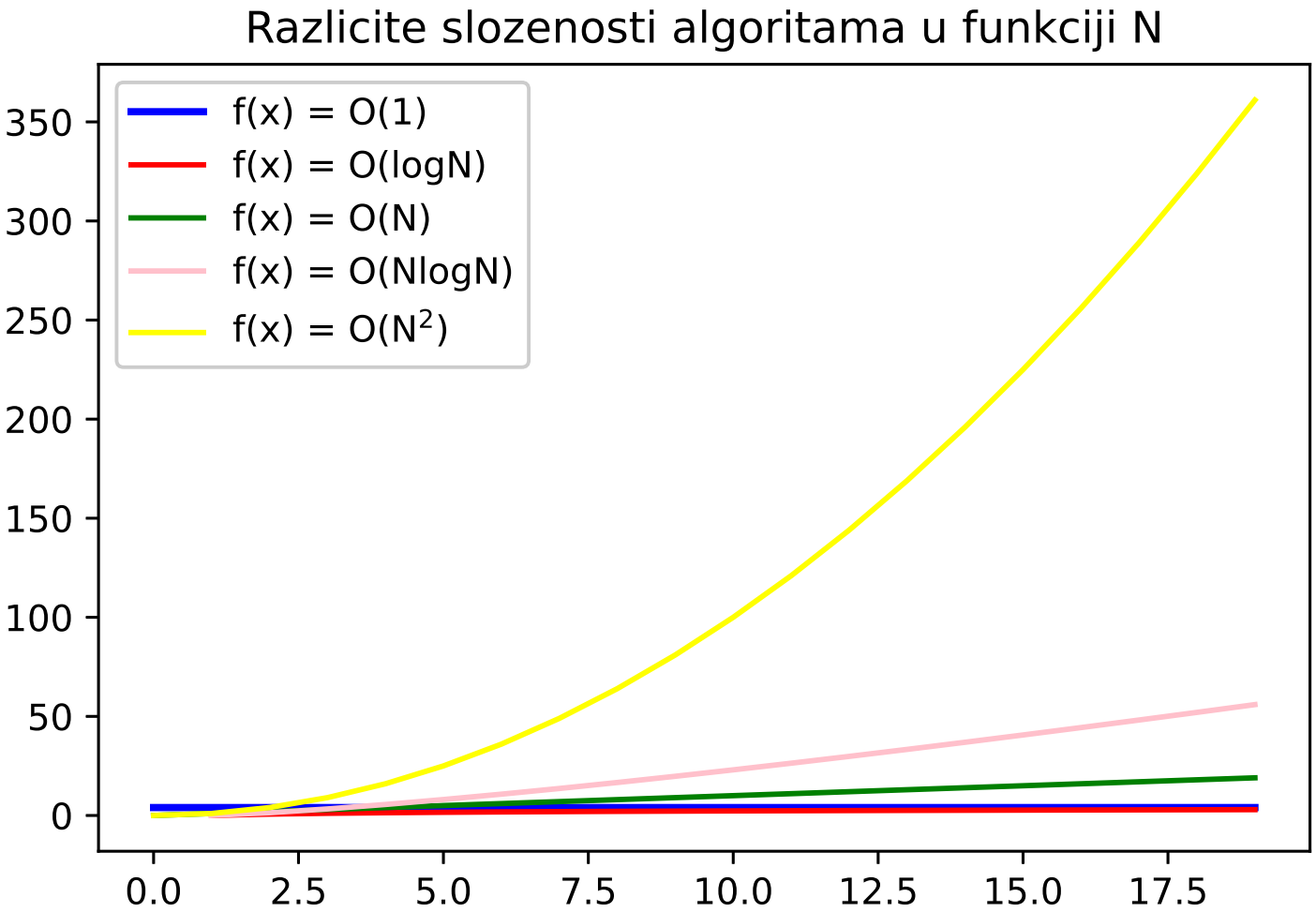
$\mathcal{O}(?)$:

- $\mathcal{O}(1)$
- $\mathcal{O}(N^7)$
- $\mathcal{O}(logN)$
- $\mathcal{O}(N^2)$

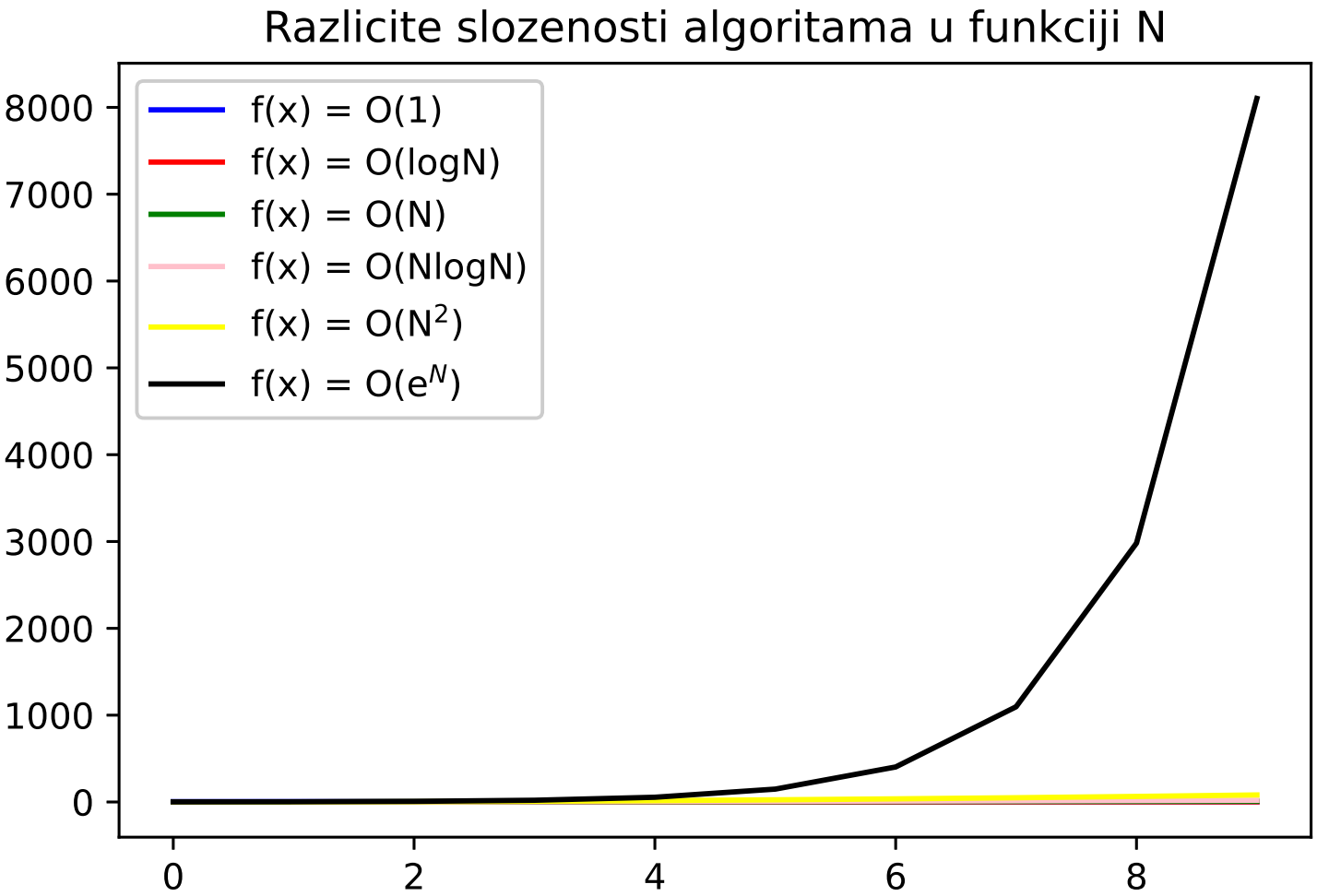
Класе сложености алгоритама - графици



Класе сложености алгоритама - графици



Класе сложености алгоритама - графици



Константна сложеност алгоритма

Сложеност не зависи од улаза.

Константно време захтевају основне операције / кораци, о којима смо већ рекли у уводу модела МСП.

Врло мало занимљивих алгоритама спада у ову класу, али са друге стране често делови алгоритама потпадају под ову класу.

Могуће је имати и петље и рекурзивне позиве, али број пролаза кроз њих, односно позива је независан од улаза у процедуру.

Логаритамска сложеност алгоритма

Сложеност расте као логаритам величине једног од улаза:

Примери логаритамске сложености:

- метода половљења интервала,
- бинарна претрага.

Пример логаритамске сложености алгоритма

Процедура за претварање природног броја у ниску:

```
def бројУниску(n):
    цифре = '012345679'
    if n == 0:
        return '0':
    рез = ''
    while n > 0:
        рез=цифре[n%10]+рез
        n=n//10
    return рез
```

Како нема позива функција (рекурзивних, нити неке друге функције) разматрати само петље.

Унутар саме петље постоји константан број корака.

Колико се укупно пута пролази кроз дату петљу?

- Колико пута неки број n могу поделити са 10?
- $\mathcal{O}(\log_{10} n)$

Линеарна сложеност алгоритама

Ова сложеност се добија при проласку кроз читав (итерабилан) објекат, односно када је неопходно посматрати све његове чланове.

Пример је процедура која рачуна збир свих децималних цифара унутар произвољне ниске:

```
def збирЦифара(n):
    збир = 0
    for s in n:
        if s.isdigit():
            збир += int(s)
    return збир
```

Сложеност алгоритма: $\mathcal{O}(|n|)$.

Још линеарне сложености алгоритама

Сложеност може зависити и од броја рекурзивних позива.

```
def факторијел(n):
    if n == 1:
        return n
    else:
        return n * факторијел(n-1)
```

Колико има рекурзивних позива?

- `факторијел(n)`, па онда `факторијел(n-1)`, па `факторијел(n-2)` и све тако до `факторијел(3)`, те `факторијел(2)` и на самом крају `факторијел(1)`.
- Сложеност сваког позива је константна, односно $\mathcal{O}(1)$.
- Асимптотска сложеност алгоритма је: $n \cdot \mathcal{O}(1) = \mathcal{O}(n)$.

Логлинеарна сложеност алгоритама

Јако много алгоритама у пракси имају логлинеарну сложеност.

Често логлинеарну сложеност имају алгоритми за сортирање.

Вратићемо се на њих мало касније.

Полиномијална сложеност алгоритама

Најчешћи полинимијални алгоритми су заправо алгоритми квадратне и кубне сложености, то јест њихова сложеност расте с квадратом односно кубом вредности, то јест дужине улаза.

Угнежђене петље су најчешће одговорне за њих.

Анализа примера квадратне сложености

```
def presek(Na, Nb):
    p = []
    for ea in Na:
        for eb in Nb:
            if ea == eb:
                p.append(ea)
    rez = []
    for e in p:
        if not e in rez:
            rez.append(e)
    return rez
```

Анализа продецуре за проналажење јединственог пресека улазних низова.

Прва угнежђена `for` петља ће се извршити тачно $|Na| \cdot |Nb| =$`len(Na)*len(Nb)` пута и ту има тачно два пута толико корака.

Друга `for` петља ће извршити највише $|Na| =$`len(Na)` пута.

Други члан врло брзо бива надвладан од првог члана.

Сложеност је: $\mathcal{O}(|Na| \cdot |Nb|)$

Експоненцијална сложеност алгоритама

Рекурзивна процедура у којој постоји више од једног рекурзивног позива за сваку улазну величину проблема:

- на пример, Ханојска кула.

Многи важни проблеми су инхерентно експоненцијални:

- нажалост, пошто је цена изузетно висока,
- води ка разматрању апроксимативних решења јер хеуристике имају далеко нижу сложеност.

Анализа примера алгоритма експоненцијалне сложености

```
def fibonaci(n):
    """
    Написати функцију која враћа нти-
    члан Фибоначијевог низа.
    Сваки члан Фибоначијевог низа се
    добија као збир претходна два његова
    члана.
    Почетни чланови низа су 0 и 1.
    """
    if n == 1:
        return 0
    if n == 2:
         return 1
    return fibonaci(n-1) + fibonaci(n-2)
```

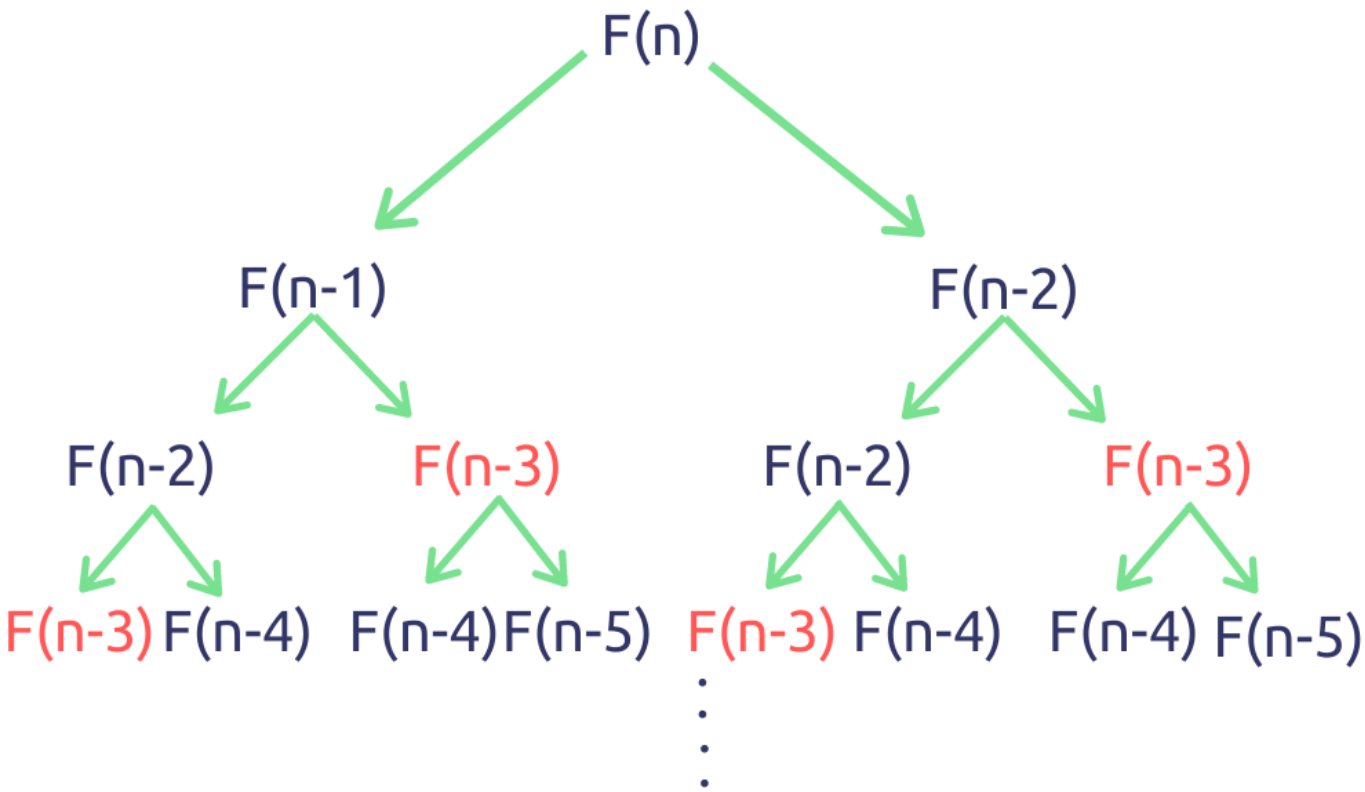
Анализа рекурзивне процедуре која рачуна n-ти члан фибоначијевог низа.

Размислити о броју рекурзивних позива функције, као и о количини редундантног израчунавања.

Један позив функције је сам по себи једноставан, и садржи константан број основних операција, али у главном резултује у 2 додатна рекурзивна позива функције!

Управо овај број рекурзивних позива расте експоненцијално, и главни је кривац за сложеност овог алгоритма.

Анализа примера алгоритма експоненцијалне сложености



Због чињенице да број рекурзивних позива функција расте експоненцијално, и сама сложеност алгоритма износи $\mathcal{O}(2^n)$.