

Основи програмирања

Вежбе 12

Исидора Грујић
isidora@uni.kg.ac.rs

Лазар Илић
iliclazar15@gmail.com

Факултет инжењерских наука
Универзитет у Крагујевцу

децембар 2023.

1 Алгоритми сортирања

2 Алгоритми претраживања

Под сортирањем подразумевамо преуређивање низа (или генерално итерабилног објекта) тако да важи следеће:

$$N_0 < N_1 < N_2 < \dots < N_{n-1}$$

Где је N_i i -ти члан низа, n дужина низа, а $<$ произвољни оператор који пореди два члана (сетити се примера са ламбда кључевима).

Сортирање може у многоне олакшати решавање неких проблема, али имати у виду цену сортирања, односно временску, као и меморијску комплексност алгоритама за сортирање.

У наставку ћемо се упознати са следећим алгоритмима сортирања:

- Сортирање избором - *Selection sort*
- Сортирање мехуром - *Bubble sort*
- Сортирање уметањем - *Insertion sort*
- Сортирање спајањем - *Merge sort*

Сортирање избором - *Selection sort*

Идеја: у сваком моменту низ посматрати као дводелну структуру - сачињену од сортираног дела и несортираног дела. У почетку је читав низ несортиран, а сортиран део је празан, док је након примене алгоритма сортирања обрнуто.

У свакој итерацији одредити најмањи члан несортираног дела низа, и њега заменити са првим чланом несортираног дела; чиме се суштински помера граница између сортираног и несортираног дела.

Погледати следеће илустрације: [слика](#) и [анимација](#).

Сортирање избором - *Selection sort*

```
def sortiranjeIzborom(N):  
    for i in range(len(N) - 1):  
        minIndeks = i  
        minVrednost = N[i]  
        j = i + 1  
        while j < len(N):  
            if minVrednost > N[j]:  
                minIndeks = j  
                minVrednost = N[j]  
            j = j + 1  
        privremeno = N[i]  
        N[i] = N[minIndeks]  
        N[minIndeks] = privremeno
```

Процедура која сортира низ такозваним сортирањем избором.

Која је сложеност овог алгоритама?

Сортирање избором - *Selection sort*

```
def sortiranjeIzborom(N):  
    for i in range(len(N) - 1):  
        minIndeks = i  
        minVrednost = N[i]  
        j = i + 1  
        while j < len(N):  
            if minVrednost > N[j]:  
                minIndeks = j  
                minVrednost = N[j]  
            j = j + 1  
        privremeno = N[i]  
        N[i] = N[minIndeks]  
        N[minIndeks] = privremeno
```

Процедура која сортира низ такозваним сортирањем избором.

Која је сложеност овог алгоритама?

Сортирање мехуrom - *Bubble sort*

Идеја: Кренувши од почетка низа (члана нултог индекса), поредити вредности два суседна члана, и уколико нису у одговарајућем поретку, заменити им места; затим прећи на следећи пар чланова.

Понављати овај поступак све док није потребно извршити још ротација, односно, док низ није сортиран.

Погледати следеће илустрације: [слика](#) и [анимација](#).

Сортирање мехуrom - *Bubble sort*

```
def sortiranjeMehurom(N):  
    zamena = False  
    while not zamena:  
        zamena = True  
        for j in range(1, len(N)):  
            if N[j-1] > N[j]:  
                zamena = False  
                privremeno = N[j]  
                N[j] = N[j-1]  
                N[j-1] = privremeno
```

Процедура која сортира низ такозваним сортирањем мехуrom.

Која је сложеност овог алгоритма?

Сортирање мехуrom - *Bubble sort*

```
def sortiranjeMehurom(N):  
    zamena = False  
    while not zamena:  
        zamena = True  
        for j in range(1, len(N)):  
            if N[j-1] > N[j]:  
                zamena = False  
                privremeno = N[j]  
                N[j] = N[j-1]  
                N[j-1] = privremeno
```

Процедура која сортира низ такозваним сортирањем мехуrom.

Која је сложеност овог алгоритма?

Сортирање уметањем - *Insertion sort*

Идеја: "паковати" чланове низа редом, сваки на своје место. Слично првом алгоритму, у сваком моменту низ посматрати као дводелну структуру - сачињену од сортираног дела и несортираног дела.

У свакој итерацији, узети вредност из несортираног дела, а затим је сместити на њој одговарајуће место у сортираном делу (односно, наћи први мањи члан од тренутног, а онда тренутни уметнути на одговарајући индекс). Ово потенцијално укључује померање чланова сортираног дела!

Погледати следеће илустрације: [слика](#) и [анимација](#).

Сортирање уметањем - *Insertion sort*

```
def sortiranjeUmetanjem(N):  
    for i in range(1, len(N)):  
        trenutno = N[i]  
        j = i - 1  
        while j >= 0 and trenutno < N[j]:  
            N[j+1] = N[j]  
            j = j - 1  
        N[j+1] = trenutno
```

Процедура која сортира низ такозваним сортирањем уметањем.

Која је сложеност овог алгоритма?

Сортирање уметањем - *Insertion sort*

```
def sortiranjeUmetanjem(N):  
    for i in range(1, len(N)):  
        trenutno = N[i]  
        j = i - 1  
        while j >= 0 and trenutno < N[j]:  
            N[j+1] = N[j]  
            j = j - 1  
        N[j+1] = trenutno
```

Процедура која сортира низ такозваним сортирањем уметањем.

Која је сложеност овог алгоритма?

Сортирање спајањем - *Merge sort*

Идеја: Искористити такозвани „подели па владај” приступ:

- ❶ Најпре, ако је дужина низа или 0 или 1, већ је сортиран.
- ❷ Затим, ако је у низу више од једног члана, поделити га на два подниза и сортирати најпре сваки од њих понаособ.
- ❸ Коначно, спојити резултате:
 - ❶ спајање се састоји од поређења првих чланова оба низа и премештања мањег од њих на крај крајње сортираног низа.
 - ❷ када је један од низова празан, само ископирати други низ.

Погледати следеће илустрације: [слика](#) и [анимација](#).

Сортирање спајањем - *Merge sort*

```
def спајање(лева, десна, пореди):
    резултат, i, j = [], 0, 0
    while i < len(лева) and j < len(десна):
        if пореди(лева[i], десна[j]):
            резултат.append(лева[i])
            i = i + 1
        else:
            резултат.append(десна[j])
            j = j + 1
    while (i < len(лева)):
        резултат.append(лева[i])
        i = i + 1
    while (j < len(десна)):
        резултат.append(десна[j])
        j = j + 1
    return резултат
```

Процедура за спајање два сортирана подниза у један

Сортирање спајањем - *Merge sort*

```
import operator

def sortiranjeSpajanjem(N, poredi = operator.lt):
    if len(N) < 2:
        return N[:]
    sredina = int(len(N)/2)
    leva = sortiranjeSpajanjem(N[:sredina], poredi)
    desna = sortiranjeSpajanjem(N[sredina:], poredi)
    return spajanje(leva, desna, poredi)
```

Процедура која користи сортирање спајањем да сортира низ.

Која је сложеност овог алгоритма?

Бржи алгоритам (временска комплексност му је нижа од претходних),
али је меморијски захтевнији!

Сортирање спајањем - *Merge sort*

```
import operator

def sortiranjeSpajanjem(N, poredi = operator.lt):
    if len(N) < 2:
        return N[:]
    sredina = int(len(N)/2)
    leva = sortiranjeSpajanjem(N[:sredina], poredi)
    desna = sortiranjeSpajanjem(N[sredina:], poredi)
    return spajanje(leva, desna, poredi)
```

Процедура која користи сортирање спајањем да сортира низ.

Која је сложеност овог алгоритма?

Бржи алгоритам (временска комплексност му је нижа од претходних),
али је меморијски захтевнији!

Сортирање спајањем - *Merge sort*

```
import operator

def sortiranjeSpajanjem(N, poredi = operator.lt):
    if len(N) < 2:
        return N[:]
    sredina = int(len(N)/2)
    leva = sortiranjeSpajanjem(N[:sredina], poredi)
    desna = sortiranjeSpajanjem(N[sredina:], poredi)
    return spajanje(leva, desna, poredi)
```

Процедура која користи сортирање спајањем да сортира низ.

Која је сложеност овог алгоритма?

Бржи алгоритам (временска комплексност му је нижа од претходних),
али је меморијски захтевнији!

1 Алгоритми сортирања

2 Алгоритми претраживања

Алгоритми за претраживање или алгоритми претраге су методе проналаска члана са тачно одређеним особинама у неком скупу.

Скуп свих чланова се обично назива домен претраге.

У наставку ћемо се упознати са следећим алгоритмима претраге:

- Линеарна претрага
- Бинарна претрага

Линеарна претрага

```
def линеарнаПретрага(niz, x):  
    for e in niz:  
        if e == x:  
            return True  
    return False
```

Процедура која вржи претрагу низа редоследом од првог до последњег члана не би ли се утврдило да ли је елемент присутан у низу или није.

Која је сложеност овог алгоритма?

Линеарна претрага

```
def линеарнаПретрага(niz, x):  
    for e in niz:  
        if e == x:  
            return True  
    return False
```

Процедура која вржи претрагу низа редоследом од првог до последњег члана не би ли се утврдило да ли је елемент присутан у низу или није.

Која је сложеност овог алгоритма?

Бинарна претрага

Алгоритам двојне (бинарне) претраге низа:

- 1 изабрати индекс i који дели низ N на две половине,
- 2 затим испитати да ли је средњи члан низа $N[i] == x$,
- 3 уколико није, испитати да ли је $N[i]$ веће или мање од x ,
- 4 у зависности од одговора претражити леву/десну половину.

Нови случај такозваних „подели па владај” алгоритма:

- поделити проблем на мање, односно простије потпроблеме (краће низове), уз неке додатне једноставне операције.
- одговор на мање проблеме је одговор на почетни проблем.

Да ли ово ради за сваки низ?
Неопходно је да је низ N сортиран пре претраге!

Бинарна претрага

Алгоритам двојне (бинарне) **претраге** низа:

- 1 изабрати индекс i који дели низ N на две половине,
- 2 затим испитати да ли је средњи члан низа $N[i] == x$,
- 3 уколико није, испитати да ли је $N[i]$ веће или мање од x ,
- 4 у зависности од одговора претражити леву/десну половину.

Нови случај такозваних „подели па владај” алгоритма:

- поделити проблем на мање, односно простије потпроблеме (краће низове), уз неке додатне једноставне операције.
- одговор на мање проблеме је одговор на почетни проблем.

Да ли ово ради за сваки низ?

Неопходно је да је низ N сортиран пре претраге!

Бинарна претрага

Алгоритам двојне (бинарне) претраге низа:

- 1 изабрати индекс i који дели низ N на две половине,
- 2 затим испитати да ли је средњи члан низа $N[i] == x$,
- 3 уколико није, испитати да ли је $N[i]$ веће или мање од x ,
- 4 у зависности од одговора претражити леву/десну половину.

Нови случај такозваних „подели па владај” алгоритма:

- поделити проблем на мање, односно простије потпроблеме (краће низове), уз неке додатне једноставне операције.
- одговор на мање проблеме је одговор на почетни проблем.

Да ли ово ради за сваки низ?

Неопходно је да је низ N сортиран пре претраге!

Бинарна претрага

```
def pretraga(N, x):  
    def dvojnaPretraga(N, x, d, g):  
        if d == g:  
            return N[d] == x  
        s = d + int((g - d)/2)  
        if N[s] == x:  
            return True  
        if N[s] > x:  
            return dvojnaPretraga(N, x, d, s-1)  
        else:  
            return dvojnaPretraga(N, x, s + 1, g)  
  
    if len(N) == 0:  
        return False  
    else:  
        return dvojnaPretraga(N, x, 0, len(N)-1)
```

Рекурзивна процедура двојне претраге низа по некој вредности.

Бинарна претрага

```
def pretraga(N, x):  
    def dvojnaPretraga(N, x, d, g):  
        if d == g:  
            return N[d] == x  
        s = d + int((g - d)/2)  
        if N[s] == x:  
            return True  
        if N[s] > x:  
            return dvojnaPretraga(N, x, d, s-1)  
        else:  
            return dvojnaPretraga(N, x, s + 1, g)  
  
    if len(N) == 0:  
        return False  
    else:  
        return dvojnaPretraga(N, x, 0, len(N)-1)
```

Рекурзивна процедура двојне претраге низа по некој вредности.

Која је сложеност овог алгоритма?

Бинарна претрага

```
def pretraga(N, x):  
    def dvojnaPretraga(N, x, d, g):  
        if d == g:  
            return N[d] == x  
        s = d + int((g - d)/2)  
        if N[s] == x:  
            return True  
        if N[s] > x:  
            return dvojnaPretraga(N, x, d, s-1)  
        else:  
            return dvojnaPretraga(N, x, s + 1, g)  
  
    if len(N) == 0:  
        return False  
    else:  
        return dvojnaPretraga(N, x, 0, len(N)-1)
```

Рекурзивна процедура двојне претраге низа по некој вредности.

Која је сложеност овог алгоритма?

Да ли је "исплативије" користити линеарну претрагу, или комбиновати бинарну претрагу и неки алгоритам сортирања, рецимо сортирање спајањем?