Programiranje 2, letnji semestar 2023/4

Nizovi i pokazivači

Stefan Nikolić

Prirodno-matematički fakultet, Novi Sad
18.03.2024.

Sa prošlog časa

```
unsigned n = 100;
for(unsigned i = 1; i <= 6; ++i)
{
        unsigned uzajamno_prostih = 0;
        unsigned broj_parova = 0;
        for(unsigned a = 1; a <= n; ++a)
        {
                for(unsigned b = 1; b < a; ++b)
                {
                        ++broj_parova;

                        //Dovoljno je da posmatramo samo neuredjene parove

                        unsigned nzd = 1;
                        for(unsigned d = 2; d <= sqrt(b); ++d)
                        {
                                if(!(a % d) && !(b % d))
                                {
                                        nzd = d;
                                        //Posto nam je bitno samo da
                                        //nadjemo neki delilac > 1,
                                        //mozemo da prekinemo petlju
                                        //cim se to desi.

                                        break;
                                }
                        }
                        if (nzd == 1)
                        {
                                uzajamno_prostih++;
                        }
                }
        }
        cout << "p(" << n << ") = " <<
                double(uzajamno_prostih) / broj_parova << endl;
```

To nije verovatnoća koju tražimo

Sa prošlog časa

```cpp
unsigned n = 100;
for(unsigned i = 1; i <= 6; ++i)
{
        unsigned uzajamno_prostih = 0;
        unsigned broj_parova = 0;
        for(unsigned a = 1; a <= n; ++a)
        {
                for(unsigned b = 1; b < a; ++b)
                {
                        ++broj_parova;

                        //Dovoljno je da posmatramo samo neuredjene parove

                        unsigned nzd = 1;
                        for(unsigned d = 2; d <= sqrt(b); ++d)
                        {
                                if(!(a % d) && !(b % d))
                                {
                                        nzd = d;
                                        //Posto nam je bitno samo da
                                        //nadjemo neki delilac > 1,
                                        //mozemo da prekinemo petlju
                                        //cim se to desi.

                                        break;
                                }
                        }
                        if (nzd == 1)
                        {
                                uzajamno_prostih++;
                        }
                }
        }
        cout << "p(" << n << ") = " <<
                double(uzajamno_prostih) / broj_parova << endl;
```

$$a = 121, b = 77$$

$$d \in [2, \sqrt{77} = 8,775)$$

Nikada ne uočavamo 11,
pa 121, 77 računamo u uzajamno proste

Sa prošlog časa

```
    unsigned uzajamno_prostih = 0;
    for(unsigned a = 1; a <= n; ++a)
    {
            for(unsigned b = 1; b < a; ++b)
            {
                    //Dovoljno je da posmatramo samo neuredjene parove

                    unsigned nzd = 1;
                    for(unsigned d = 2; d <= b; ++d)
                    {
                            if(!(a % d) && !(b % d))
                            {
                                    nzd = d;
                                    //Posto nam je bitno samo da
                                    //nadjemo neki delilac > 1,
                                    //mozemo da prekinemo petlju
                                    //cim se to desi.

                                    break;
                            }
                    }
```

```
                            if (nzd == 1)
                            {
                                    uzajamno_prostih++;
                            }
                    }
            }
            cout << "p(" << n << ") = " <<
                    double(uzajamno_prostih * 2 + 1) / (n * n) << endl;

            //Za svaki prost par (a, b), imamo i prost par (b, a) i dodajemo jos (1, 1)

            n *= i % 2 ? 5 : 2;
    }

    return 0;
}
```

Sa prošlog časa

```
p(100) = 0.6087
p(500) = 0.608924
p(1000) = 0.608383

real    0m0.399s
user    0m0.399s
sys     0m0.000s
```

Optimizacicijom koda smo ubrzali program $> 4\times$

Da li možemo još nešto da poboljšamo?

Ljudi su proučavali mehanizaciju računanja mnogo pre pojave automatskih računara

Peta verzija

```cpp
#include <iostream>
using namespace std;

int main()
{
        unsigned n = 100;
        for(unsigned i = 1; i <= 6; ++i)
        {
                unsigned uzajamno_prostih = 0;
                for(unsigned a = 1; a <= n; ++a)
                {
                        for(unsigned b = 1; b < a; ++b)
                        {
                                unsigned a_kopija = a;
                                unsigned b_kopija = b;

                                for(; b_kopija > 0;)
                                {
                                        //Ne znamo koliko puta ce petlja biti izvrsena
                                        //a nemamo ni bilo kakvu drugu svrhu od brojaca
                                        //pa nam je dovoljan samo uslov

                                        unsigned zamena = b_kopija;
                                        b_kopija = a_kopija % b_kopija; // b <- a mod b
                                        a_kopija = zamena;              // a <- b
                                }
                                if (a_kopija == 1)
                                {
                                        uzajamno_prostih++;
                                }
                        }
```

Peta verzija

```
p(100) = 0.608722
p(500) = 0.608925
p(1000) = 0.608383

real    0m0.040s
user    0m0.040s
sys     0m0.000s
```

Poboljšanje algoritma je ubrzalo program $\sim 40\times$

Za veće vrednosti n, razlika je još mnogo značajnija

- računska složenost naivnog algoritma: $\mathcal{O}(n^3)$
- računska složenost nakon upotrebe Euklidovog algoritma: $\mathcal{O}(n^2 \log_{10} n)$

Ako pretpostavimo da je jedna konstanta dominantna, možemo je i izmeriti

Naivni NZD

$$t_1(n) = k_1 \times n^3$$

$$t_1(100) = 0,000308365$$

```
p(100) = 0.6087

real    0m0.003s
user    0m0.003s
sys     0m0.000s
```

$$t_1(1000) = 0,308365$$

```
p(1000) = 0.608383

real    0m0.333s
user    0m0.332s
sys     0m0.000s
```

```
p(10000) = 0.60795

real    5m8.444s
user    5m8.365s
sys     0m0.012s
```

$$k_1 = t_1(10000)/10000^3 = 3,08365 \times 10^{-10}$$

$$t_1(100000) = 85,66h$$

Euklidov NZD

$$t_2(n) = k_2 \times n^2 \log_{10} n$$

```
p(100) = 0.6087

real    0m0.002s
user    0m0.002s
sys     0m0.000s
```

$$t_2(100) = 0,0002$$

Za male vrednosti n, pretpostavka o dominantnoj konstanti ne važi

```
p(1000) = 0.608383

real    0m0.032s
user    0m0.028s
sys     0m0.004s
```

$$t_2(1000) = 0,0304275$$

```
p(10000) = 0.60795

real    0m4.057s
user    0m4.051s
sys     0m0.004s
```

$$k_2 = t_2(10000)/(4 \times 10000^2) = 1,01425 \times 10^{-8}$$

$$t_2(100000) = 507,125$$

```
p(100000) = 0.60793

real    8m41.721s
user    8m40.253s
sys     0m0.096s
```

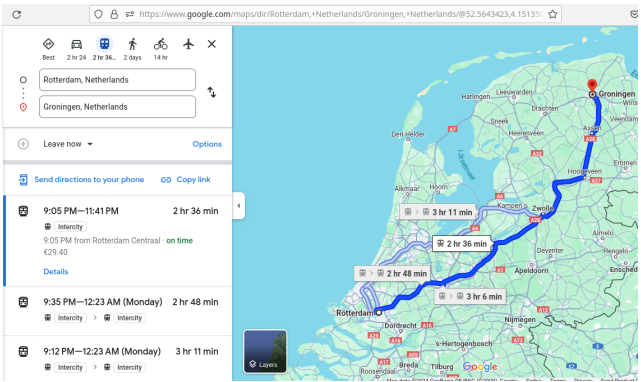
Edsger Dajkstra (Edsger Dijkstra)

Nakon što sam programirao oko tri godine, imao sam razgovor sa A. van Vijngardenom, mojim tadašnjim šefom u Matematičkom centru u Amsterdamu. Bio je to razgovor na kom ću mu biti zahvalan dok sam živ. Suština je bila u tome da je trebalo da istovremeno studiram teorijsku fiziku na Univerzitetu u Lajdenu, a kako mi je bivalo sve teže i teže da kombinujem te dve aktivnosti, trebalo je da donesem odluku: ili da prestanem da programiram i postanem pravi, ozbiljan teorijski fizičar, ili da nastavim svoje studije fizike do tek formalnog završetka, uz minimalni trud, i da postanem... Da, šta? Programer? Ali je li to bilo zanimanje vredno poštovanja? Na kraju krajeva, šta je to bilo programiranje? Gde je bilo smislen korpus znanja koji bi programiranje podržao kao intelektualno ozbiljnu disciplinu? Sećam se prilično živo koliko sam zavideo svojim kolegama iz oblasti hardvera koji su, kada bi ih neko upitao za stručna znanja, barem mogli da kažu da znaju sve o vakumskim cevima, pojačavačima i sličnom. S druge strane, ja sam se pri susretu sa tim pitanjem nalazio bez odgovora. Pun nedoumica, pokucao sam na vrata van Vijngardenove kancelarije i upitao ga da li mogu „na trenutak" da razgovaram sa njim. Kada sam, posle nekoliko sati, napustio njegov kabinet, bio sam druga osoba. Nakon što je pažljivo saslušao moje probleme, složio se da do tada nije bilo nečeg poput programerske discipline. Zatim je dodao da su automatski računari tu da ostanu, da smo mi tek na početku, i zar ne bih ja mogao da budem jedna od tih osoba pozvanih da programiranje učine ozbiljnom disciplinom u godinama koje slede? To je bila prekretnica u mom životu i završio sam svoje studije fizike samo formalno, što sam brže mogao. Jedna pouka ove priče je, naravno, da moramo da budemo veoma pažljivi kada mlađim ljudima dajemo savete; ponekad ih i slede!

Novi problem: najkraće putanje u grafovima

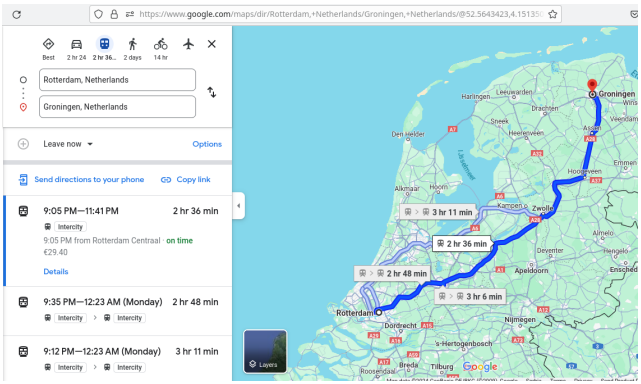
Vratimo se u 1956.

Vratimo se u 1956.



Dajkstru je mučio sledeći problem: kako da nađe najkraću putanju vozom između Roterama i Groningena?

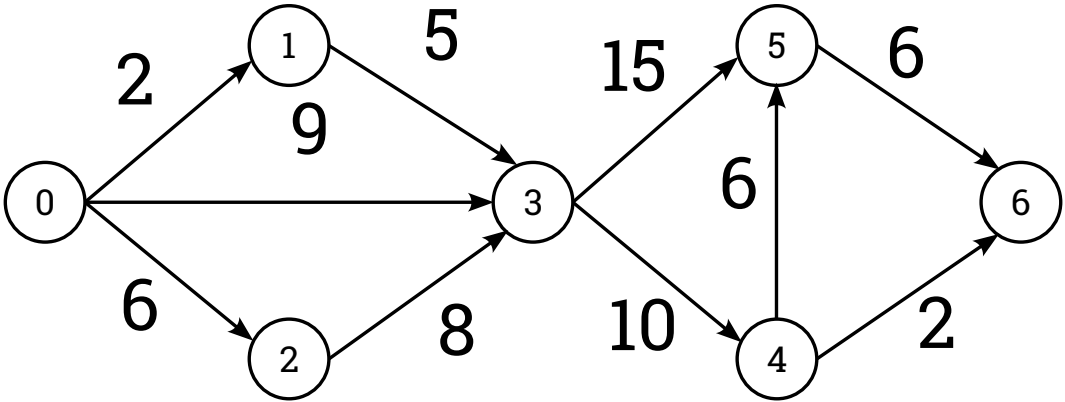
Vratimo se u 1956.



A onda i uopšteno, između bilo koja dva grada

Dajkstrin algoritam

Kako možemo predstaviti železničku mrežu?



Najkraća putanja od 0 do 6: (0, 1, 3, 4, 6)

najkraća putanja od 0 do 6: (0, 1, 3, 4, 6)

Jedna implementacija Dajkstrinog algoritma u C++-u

```cpp
#include <iostream>
#include <climits>

#define V 7
#define NOT_REACHED -1
#define INF UCHAR_MAX

using namespace std;

int main()
{
        unsigned char adj[V][V] = {
                                    {0, 2, 6, 9, INF, INF, INF},
                                    {INF, 0, INF, 5, INF, INF, INF},
                                    {INF, INF, 0, 8, INF, INF, INF},
                                    {INF, INF, INF, 0, 10, INF, INF},
                                    {INF, INF, INF, INF, 0, 6, 2},
                                    {INF, INF, INF, INF, INF, 0, 6},
                                    {INF, INF, INF, INF, INF, INF, 0}
                                   };

        const unsigned char s = 0;
        const unsigned char t = 6;
```

Najpre predstavimo graf otežanom matricom susednosti

Nizovi

Deklaracija nizova

tip identifikator[dužina niza];

Dužina niza bi trebalo da bude konstanta poznata prilikom kompilacije[1]

[1] Iako neki kompajleri dozvoljavaju specifikaciju dužine pomoću promenljive, zbog portabilnosti koda, to se ne preporučuje.

Operator indeksiranja

Da bismo pristupili i-tom elementu niza, koristimo operator indeksiranja:

x = a[i]; //i mora biti iz opsega $[0, \textit{duzina niza})$
a[i] = y;

Opseg niza je od 0 do dužina niza - 1

Primer deklaracije i indeksiranja

```
#include <iostream>
#define LEN 10
using namespace std;

int main()
{
        int a[LEN];

        for(unsigned i = 0; i < LEN; ++i)
        {
                cout << "Unesite " << i << "-ti element niza: ";
                cin >> a[i];
        }

        cout << "Uneti niz ispisan u obrnutom poretku:\n";
        for(int i = LEN - 1; i >= 0; --i)
                cout << a[i] << endl;

        return 0;
}
```

Primer deklaracije i indeksiranja

```
Unesite 0-ti element niza: 7
Unesite 1-ti element niza: 3
Unesite 2-ti element niza: 4
Unesite 3-ti element niza: 6
Unesite 4-ti element niza: 2
Unesite 5-ti element niza: 1
Unesite 6-ti element niza: 9
Unesite 7-ti element niza: 8
Unesite 8-ti element niza: 11
Unesite 9-ti element niza: 19
Uneti niz ispisan u obrnutom poretku:
19
11
8
9
1
2
6
4
3
7
```

Indeksiranje izvan opsega

Pristup elementima niza van opsega nije definisan

Nekada će dovesti do greške u izvršenju

```cpp
#include <iostream>
#define LEN 10
using namespace std;

int main()
{
        int a[LEN];

        for(unsigned i = 0; i <= LEN; ++i)
        {
                cout << "Unesite " << i << "-ti element niza: ";
                cin >> a[i];
        }

        cout << "Uneti niz ispisan u obrnutom poretku:\n";
        for(int i = LEN; i >= 0; --i)
                cout << a[i] << endl;

        return 0;
}
```

Nekada će dovesti do greške u izvršenju

```
Unesite 0-ti element niza: 7
Unesite 1-ti element niza: 3
Unesite 2-ti element niza: 2
Unesite 3-ti element niza: 4
Unesite 4-ti element niza: 5
Unesite 5-ti element niza: 11
Unesite 6-ti element niza: 6
Unesite 7-ti element niza: 9
Unesite 8-ti element niza: 10
Unesite 9-ti element niza: 20
Unesite 10-ti element niza: 33
/bin/bash: line 1:  4747 Segmentation fault      (core dumped) ./run
```

Najčešće je to **Segmentation Fault**. O tome ćemo uskoro govoriti detaljnije.

A nekada će se činiti da sve radi normalno (daleko opasnije)

```cpp
#include <iostream>
#define LEN 10
using namespace std;

int main()
{
        int a[LEN];

        for(unsigned i = 0; i <= LEN; ++i)
        {
                cout << "Unesite " << i << "-ti element niza: ";
                cin >> a[i];
        }

        cout << "Uneti niz ispisan u obrnutom poretku:\n";
        for(int i = LEN; i >= 0; --i)
                cout << a[i] << endl;

        return 0;
}
```

A nekada će se činiti da sve radi normalno (daleko opasnije)

```
Unesite 0-ti element niza: 7
Unesite 1-ti element niza: 3
Unesite 2-ti element niza: 4
Unesite 3-ti element niza: 6
Unesite 4-ti element niza: 2
Unesite 5-ti element niza: 1
Unesite 6-ti element niza: 9
Unesite 7-ti element niza: 8
Unesite 8-ti element niza: 11
Unesite 9-ti element niza: 19
Unesite 10-ti element niza: 20
Uneti niz ispisan u obrnutom poretku:
20
19
11
8
9
1
2
6
4
3
7
```

To zavisi i od konkretnog pokretanja programa, konkretnih ulaznih podataka...

O dužini niza sami moramo da vodimo računa

Nažalost, za nizove u C/C++-u ne postoji pandan Python-ovom **len** operatoru

Iz tog razloga uvek moramo negde da imamo sačuvanu dužinu niza pre deklaracije

Šta je zapravo niz?

Operator adrese

Sve što može biti leva strana operatora dodele ima adresu

Operator adrese

Adresu nekog objekta[2] u memoriji dobijamo upotrebom operatora adrese, &

&identifikator

[2] Ovde ne mislimo samo na objekat u smislu instance klase. Time ćemo se baviti u drugom delu predmeta.

Primer

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

int main()
{
        int a = 7;
        cout << &a << endl;

        return 0;
}
```

Primer

U kakvom su odnosu adrese elemenata niza?

```
#include <iostream>
#define LEN 10
using namespace std;

int main()
{
        int a[LEN];
        for(unsigned i = 0; i < LEN; ++i)
                cout << "Adresa " << i << ". elementa niza a je " << &a[i] << endl;

        return 0;
}
```

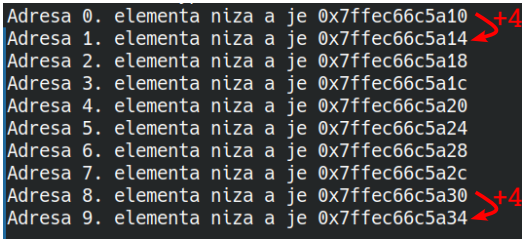
U kakvom su odnosu adrese elemenata niza?

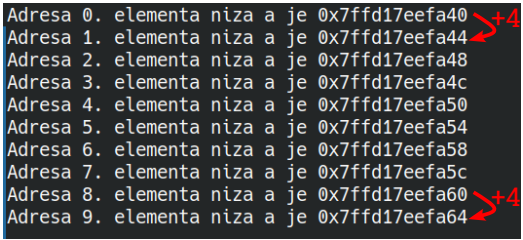
```
Adresa 0. elementa niza a je 0x7ffec66c5a10
Adresa 1. elementa niza a je 0x7ffec66c5a14
Adresa 2. elementa niza a je 0x7ffec66c5a18
Adresa 3. elementa niza a je 0x7ffec66c5a1c
Adresa 4. elementa niza a je 0x7ffec66c5a20
Adresa 5. elementa niza a je 0x7ffec66c5a24
Adresa 6. elementa niza a je 0x7ffec66c5a28
Adresa 7. elementa niza a je 0x7ffec66c5a2c
Adresa 8. elementa niza a je 0x7ffec66c5a30
Adresa 9. elementa niza a je 0x7ffec66c5a34
```

Pri svakom pozivu programa, adrese se menjaju

```
Adresa 0. elementa niza a je 0x7ffd17eefa40
Adresa 1. elementa niza a je 0x7ffd17eefa44
Adresa 2. elementa niza a je 0x7ffd17eefa48
Adresa 3. elementa niza a je 0x7ffd17eefa4c
Adresa 4. elementa niza a je 0x7ffd17eefa50
Adresa 5. elementa niza a je 0x7ffd17eefa54
Adresa 6. elementa niza a je 0x7ffd17eefa58
Adresa 7. elementa niza a je 0x7ffd17eefa5c
Adresa 8. elementa niza a je 0x7ffd17eefa60
Adresa 9. elementa niza a je 0x7ffd17eefa64
```

Ali je razmak između adresa susednih elemenata konstantan





Heksadecimalni brojevi (pozicioni sistem sa osnovom 16)

$0 = 0_{10}$	$4 = 4_{10}$	$8 = 8_{10}$	$c = 12_{10}$
$1 = 1_{10}$	$5 = 5_{10}$	$9 = 9_{10}$	$d = 13_{10}$
$2 = 2_{10}$	$6 = 6_{10}$	$a = 10_{10}$	$e = 14_{10}$
$3 = 3_{10}$	$7 = 7_{10}$	$b = 11_{10}$	$f = 15_{10}$

To možemo i automatski da proverimo

```
#include <iostream>
#define LEN 10
using namespace std;

int main()
{
        int a[LEN];
        for(unsigned i = 0; i < LEN - 1; ++i)
                cout << "Razlika izmedju adresa " << i << ". i " << i + 1 << ". elementa niza a je "
                     << long(&a[i + 1]) - long(&a[i]) << endl;

        return 0;
}
```

To možemo i automatski da proverimo

```
Razlika izmedju adresa 0. i 1. elementa niza a je 4
Razlika izmedju adresa 1. i 2. elementa niza a je 4
Razlika izmedju adresa 2. i 3. elementa niza a je 4
Razlika izmedju adresa 3. i 4. elementa niza a je 4
Razlika izmedju adresa 4. i 5. elementa niza a je 4
Razlika izmedju adresa 5. i 6. elementa niza a je 4
Razlika izmedju adresa 6. i 7. elementa niza a je 4
Razlika izmedju adresa 7. i 8. elementa niza a je 4
Razlika izmedju adresa 8. i 9. elementa niza a je 4
```

Zašto baš 4?

```
#include <iostream>
#define LEN 10
using namespace std;

int main()
{
        int a[LEN];
        for(unsigned i = 0; i < LEN - 1; ++i)
                cout << "Razlika izmedju adresa " << i << ". i " << i + 1 << ". elementa niza a je "
                     << long(&a[i + 1]) - long(&a[i]) << endl;

        return 0;
}
```

Da li će uvek biti 4?

```
#include <iostream>
#define LEN 10
using namespace std;

int main()
{
        short a[LEN];
        for(unsigned i = 0; i < LEN - 1; ++i)
                cout << "Razlika izmedju adresa " << i << ". i " << i + 1 << ". elementa niza a je "
                     << long(&a[i + 1]) - long(&a[i]) << endl;

        return 0;
}
```

Da li će uvek biti 4?

```
Razlika izmedju adresa 0. i 1. elementa niza a je 2
Razlika izmedju adresa 1. i 2. elementa niza a je 2
Razlika izmedju adresa 2. i 3. elementa niza a je 2
Razlika izmedju adresa 3. i 4. elementa niza a je 2
Razlika izmedju adresa 4. i 5. elementa niza a je 2
Razlika izmedju adresa 5. i 6. elementa niza a je 2
Razlika izmedju adresa 6. i 7. elementa niza a je 2
Razlika izmedju adresa 7. i 8. elementa niza a je 2
Razlika izmedju adresa 8. i 9. elementa niza a je 2
```

Koju hipotezu možemo da postavimo?

Koju hipotezu možemo da postavimo?

Razmak između adresa uzastopnih elemenata niza odgovara veličini tipa elemenata u broju bajtova

Kako da proverimo tu hipotezu?

Kako da proverimo tu hipotezu?

```cpp
#include <iostream>
using namespace std;

int main()
{
        cout << "Velicina tipa int u broju bajta je " << sizeof(int) << endl;
        cout << "Velicina tipa short u broju bajta je " << sizeof(short) << endl;

        return 0;
}
```

Kako da proverimo tu hipotezu?

```
Velicina tipa int u broju bajta je 4
Velicina tipa short u broju bajta je 2
```

Još jedan eksperiment

```
#include <iostream>
#define LEN 10
using namespace std;

int main()
{
        int a[LEN];

        cout.width(30); cout << "Adresa 0. elementa niza a je " << &a[0] << endl;
        cout.width(30); cout << "Adresa niza a je " << &a << endl;
        cout.width(30); cout << "a je " << a << endl;

        return 0;
}
```

Još jedan eksperiment

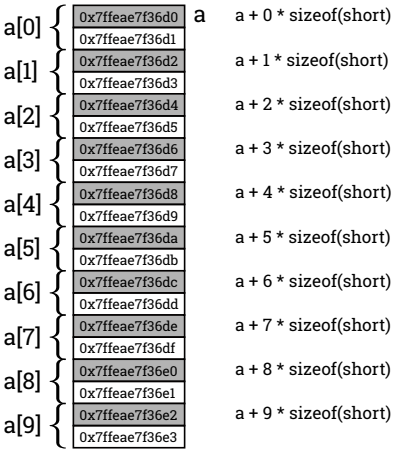
```
Adresa 0. elementa niza a je 0x7ffc69f60eb0
            Adresa niza a je 0x7ffc69f60eb0
                         a je 0x7ffc69f60eb0
```

Šta je zapravo niz?

Zaključci iz eksperimenata

short a[LEN];

Rezerviši u memoriji
prostor veličine LEN * sizeof(short)
i nazovi adresu početka tog prostora a

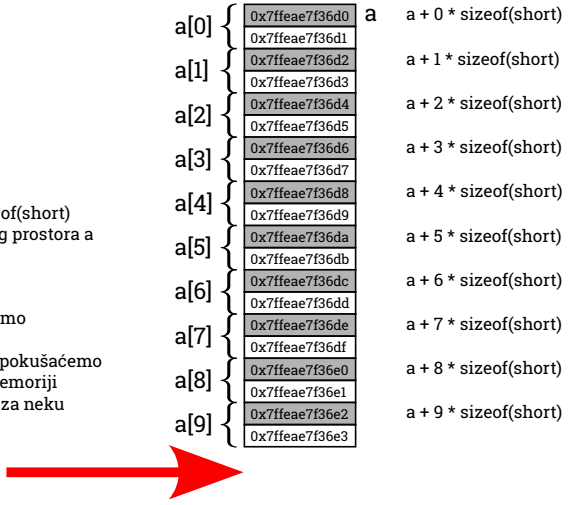


Zaključci iz eksperimenata

short a[LEN];

Rezerviši u memoriji
prostor veličine LEN * sizeof(short)
i nazovi adresu početka tog prostora a

Ako pokušamo da pristupimo
elementu van opsega niza
(nepostojećem elementu), pokušaćemo
da pristupimo lokaciji u memoriji
koja je možda rezervisana za neku
drugu promenljivu
a svakako sadrži nama
nepoznatu vrednost



Ozbiljna greška: Buffer Overflow

https://en.wikipedia.org/wiki/Buffer_overflow

'EDIA
cyclopedia

Search Wikipedia | Search

Create account Log in

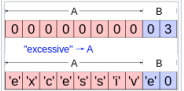
Buffer overflow

34 languages

Article Talk | Read Edit View history Tools

From Wikipedia, the free encyclopedia

In programming and information security, a **buffer overflow** or **buffer overrun** is an anomaly whereby a program writes data to a buffer beyond the buffer's allocated memory, overwriting adjacent memory locations.



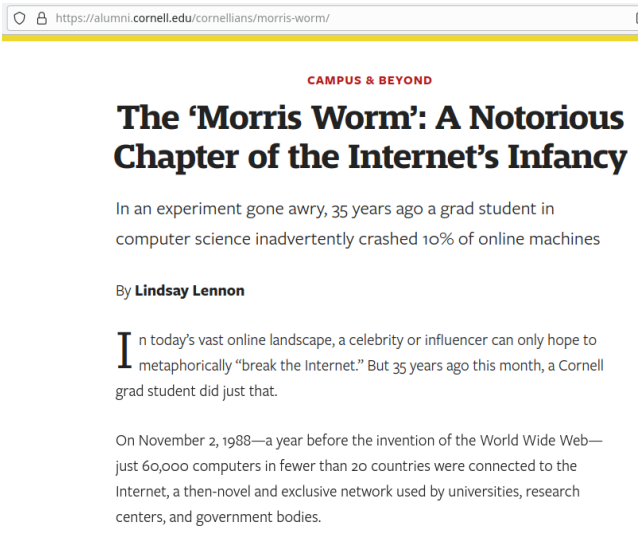
Visualization of a software buffer overflow. Data is written into A, but is too large to fit within A, so it *overflows* into B.

Buffers are areas of memory set aside to hold data, often while moving it from one section of a program to another, or between programs. Buffer overflows can often be triggered by malformed inputs; if one assumes all inputs will be smaller than a certain size and the buffer is created to be that size, then an anomalous transaction that produces more data could cause it to write past the end of the buffer. If this overwrites adjacent data or executable code, this may result in erratic program behavior, including memory access errors, incorrect results, and crashes.

Exploiting the behavior of a buffer overflow is a well-known security exploit. On many systems, the memory layout of a program, or the system as a whole, is well defined. By sending in data designed to cause a buffer overflow, it is possible to write into areas known to hold executable code and replace it with malicious code, or to selectively overwrite data pertaining to the program's state, therefore causing behavior that was not intended by the original programmer. Buffers are widespread in operating system (OS) code, so it is possible to make attacks that perform privilege escalation and gain unlimited access to the computer's resources. The famed Morris worm in 1988 used this as one of its attack techniques.

Programming languages commonly associated with buffer overflows include C and C++, which provide no built-in protection against accessing or overwriting data in any part of memory and do not automatically check that data written to an array (the built-in buffer type) is within the boundaries of that array. Bounds checking can prevent buffer overflows, but requires additional code and processing time. Modern operating systems use a variety of

Ozbiljna greška: Buffer Overflow



Pokazivači

Rekli smo da identifikator niza sadrži adresu početka prostora rezervisanog za niz

C/C++ omogućuje čuvanje adresa u promenljivama i u opštem slučaju

Te promenljive nazivamo *pokazivačima* (eng. *pointer*)

Deklaracija pokazivača

tip na čiju adresu pokazivač pokazuje * identifikator;

Primer

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

int main()
{
        int a = 7;
        int* b;

        b = &a;

        cout << "Vrednost promenljive a = " << a << ". Adresa promenljive a = " << &a << endl;
        cout << "Vrednost promenljive b = " << b << ". Adresa promenljive b = " << &b << endl;

        return 0;
}
```

Primer

```
Vrednost promenljive a = 7. Adresa promenljive a = 0x7ffd0578a5d4
Vrednost promenljive b = 0x7ffd0578a5d4. Adresa promenljive b = 0x7ffd0578a5d8
```

Primetimo da je adresa promenljive b za tačno 4 (sizeof(int) na ovom računaru) veća od adrese promenljive a. To je u ovom slučaju bila odluka kompajlera, ali u nekom drugom ne mora da bude. Nikada ne treba računati na određeni odnos adresa nezavisnih promenljivih.

Dereferenciranje pokazivača

Pokazivači ne bi bili naročito korisni ako ne bismo mogli da preko njih pristupimo memorijskoj lokaciji na adresi koju sadrže

Operator dereferenciranja je *: *identifikator

Primer

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

int main()
{
        int a = 7;
        int* b;

        b = &a;

        cout << "Vrednost promenljive a = " << a << ". Adresa promenljive a = " << &a << endl;
        cout << "Vrednost promenljive b = " << b << ". Adresa promenljive b = " << &b << endl;
        cout << "Vrednost promenljive na adresi " << b << " je " << *b << endl;

        return 0;
}
```

Primer

```
Vrednost promenljive a = 7. Adresa promenljive a = 0x7ffec7ff1e04
Vrednost promenljive b = 0x7ffec7ff1e04. Adresa promenljive b = 0x7ffec7ff1e08
Vrednost promenljive na adresi 0x7ffec7ff1e04 je 7
```

Dereferencirane pokazivače možemo koristiti i za promenu vrednosti promenljivih

```cpp
using namespace std;

int main()
{
        int a = 7;
        int* b;

        b = &a;

        cout << "Vrednost promenljive a = " << a << ". Adresa promenljive a = " << &a << endl;
        cout << "Vrednost promenljive b = " << b << ". Adresa promenljive b = " << &b << endl;
        cout << "Vrednost promenljive na adresi " << b << " je " << *b << endl;

        *b = 17; //Upisi ono sto je sa desne strane operatora dodele u lokaciju cija je adresa sadrzana u b

        cout << "Nova vrednost promenljive a je " << a << endl;
        //U nasem programu, ime te lokacije je a

        ++(*b);

        cout << "Poslednja vrednost promenljive a je " << a << endl;

        return 0;
}
```

Dereferencirane pokazivače možemo koristiti i za promenu vrednosti promenljivih

```
Vrednost promenljive a = 7. Adresa promenljive a = 0x7fff5acd54d4
Vrednost promenljive b = 0x7fff5acd54d4. Adresa promenljive b = 0x7fff5acd54d8
Vrednost promenljive na adresi 0x7fff5acd54d4 je 7
Nova vrednost promenljive a je 17
Poslednja vrednost promenljive a je 18
```

Dereferencirane pokazivače možemo koristiti i za promenu vrednosti promenljivih

Takva indirektna izmena vrednosti promenljivih može biti jako korisna
(to ćemo videti na sledećem predavanju kada budemo radili potprograme)

No nekad može biti i opasna, jer možemo slučajno izmeniti neku vrednost

Pokazivači na konstantu vrednost

Da bismo to predupredili, možemo deklarisati pokazivač kao pokazivač na konstantnu vrednost:

const tip na čiju adresu pokazivač pokazuje * identifikator;

Primer

```
using namespace std;

int main()
{
        int a = 7;
        const int* b;

        b = &a;

        cout << "Vrednost promenljive a = " << a << ". Adresa promenljive a = " << &a << endl;
        cout << "Vrednost promenljive b = " << b << ". Adresa promenljive b = " << &b << endl;
        cout << "Vrednost promenljive na adresi " << b << " je " << *b << endl;

        *b = 17; //Upisi ono sto je sa desne strane operatora dodele u lokaciju cija je adresa sadrzana u b

        cout << "Nova vrednost promenljive a je " << a << endl;
        //U nasem programu, ime te lokacije je a

        ++(*b);

        cout << "Poslednja vrednost promenljive a je " << a << endl;

        return 0;
}
```

Primer

```
pointer.cpp:15:12: error: assignment of read-only location ‘* b’
   15 |         *b = 17; //Upisi ono sto je sa desne strane operatora dodele u lokaciju cija je adresa sadrzana u b
      |         ~~~^~~~
pointer.cpp:20:12: error: increment of read-only location ‘* b’
   20 |         ++(*b);
      |           ~^~~
```

Ako pokušamo da promenimo vrednost promenljive upotrebom pokazivača koji je deklarisan kao pokazivač na konstantnu vrednost, kompajler će prijaviti grešku

Šta znači konstantna vrednost?

```cpp
#include <iostream>
using namespace std;

int main()
{
        int a = 7;
        const int* b;

        b = &a;

        cout << "Vrednost promenljive a = " << a << ". Adresa promenljive a = " << &a << endl;
        cout << "Vrednost promenljive b = " << b << ". Adresa promenljive b = " << &b << endl;
        cout << "Vrednost promenljive na adresi " << b << " je " << *b << endl;

        a = 17;

        cout << "Nova vrednost promenljive a je " << a << endl;

        ++a;

        cout << "Poslednja vrednost promenljive a je " << a << endl;

        return 0;
}
```

Deklaracija pokazivača kao pokazivača na konstantnu vrednost ne znači da promenljiva na koju pokazuje mora biti konstantna, već samo sprečava njenu promenu upotrebom datog pokazivača

Šta znači konstantna vrednost?

```
Vrednost promenljive a = 7. Adresa promenljive a = 0x7fff72724304
Vrednost promenljive b = 0x7fff72724304. Adresa promenljive b = 0x7fff72724308
Vrednost promenljive na adresi 0x7fff72724304 je 7
Nova vrednost promenljive a je 17
Poslednja vrednost promenljive a je 18
```

Deklaracija pokazivača kao pokazivača na konstantnu vrednost ne znači da promenljiva na koju pokazuje mora biti konstantna, već samo sprečava njenu promenu upotrebom datog pokazivača

Konstantni pokazivači

```cpp
#include <iostream>
using namespace std;

int main()
{
        int a = 7;
        int* const b = NULL;

        b = &a;

        cout << "Vrednost promenljive a = " << a << ". Adresa promenljive a = " << &a << endl;
        cout << "Vrednost promenljive b = " << b << ". Adresa promenljive b = " << &b << endl;
        cout << "Vrednost promenljive na adresi " << b << " je " << *b << endl;

        a = 17;

        cout << "Nova vrednost promenljive a je " << a << endl;

        ++a;

        cout << "Poslednja vrednost promenljive a je " << a << endl;

        return 0;
}
```

Kako je pokazivač promenljiva, i njega je moguće deklarisati kao konstantnog

Konstantni pokazivači

```
pointer.cpp: In function ‘int main()’:
pointer.cpp:9:11: error: assignment of read-only variable ‘b’
    9 |          b = &a;
      |          ~~^~~~

shell returned 1
```

Tada nam kompajler neće dozvoliti da promenimo adresu uskladištenu u pokazivač

Konstantni pokazivači

```cpp
#include <iostream>
using namespace std;

int main()
{
        int a = 7;
        int* const b = &a;

        cout << "Vrednost promenljive a = " << a << ". Adresa promenljive a = " << &a << endl;
        cout << "Vrednost promenljive b = " << b << ". Adresa promenljive b = " << &b << endl;
        cout << "Vrednost promenljive na adresi " << b << " je " << *b << endl;

        *b = 17;

        cout << "Nova vrednost promenljive a je " << a << endl;

        ++(*b);

        cout << "Poslednja vrednost promenljive a je " << a << endl;

        return 0;
}
```

Ali možemo menjati vrednost promenljive na koju pokazivač pokazuje

Konstantni pokazivači

```
Vrednost promenljive a = 7. Adresa promenljive a = 0x7ffd81fa8f44
Vrednost promenljive b = 0x7ffd81fa8f44. Adresa promenljive b = 0x7ffd81fa8f48
Vrednost promenljive na adresi 0x7ffd81fa8f44 je 7
Nova vrednost promenljive a je 17
Poslednja vrednost promenljive a je 18
```

Ali možemo menjati vrednost promenljive na koju pokazivač pokazuje

Konstantni pokazivači na konstantne vrednosti

```cpp
#include <iostream>
using namespace std;

int main()
{
        int a = 7;
        const int* const b = &a;

        int c;

        b = &c;

        cout << "Vrednost promenljive a = " << a << ". Adresa promenljive a = " << &a << endl;
        cout << "Vrednost promenljive b = " << b << ". Adresa promenljive b = " << &b << endl;
        cout << "Vrednost promenljive na adresi " << b << " je " << *b << endl;

        *b = 17;

        cout << "Nova vrednost promenljive a je " << a << endl;

        ++(*b);

        cout << "Poslednja vrednost promenljive a je " << a << endl;

        return 0;
}
```

Naravno, možemo i kombinovati dva const modifikatora

Konstantni pokazivači na konstantne vrednosti

```
pointer.cpp: In function ‘int main()’:
pointer.cpp:11:11: error: assignment of read-only variable ‘b’
   11 |         b = &c;
      |         ~~^~~~
pointer.cpp:17:12: error: assignment of read-only location ‘*(const int*)b’
   17 |         *b = 17;
      |         ~~~^~~~
pointer.cpp:21:12: error: increment of read-only location ‘*(const int*)b’
   21 |         ++(*b);
      |           ~^~~

shell returned 1
```

Naravno, možemo i kombinovati dva const modifikatora

NULL

Upotreba pokazivača koji sadrži nepoznatu adresu može dovesti do remećenja podataka

Zato C/C++ pruža mogućnost postavljanja pokazivača koji trenutno nisu u upotrebi na adresu koja sigurno nije korišćena za skladištenje neke promenljive

Ime te adrese je NULL i ona je najčešće jednaka baš nuli, mada to zavisi od konkretnog sistema

NULL

```cpp
#include <iostream>
using namespace std;

int main()
{
        int* a = NULL;

        cout << "a = " << a << endl;

        return 0;
}
```

NULL

NULL

```cpp
#include <iostream>
using namespace std;

int main()
{
        int* a = NULL;

        cout << "a = " << a << endl;

        cout << *a << endl;

        return 0;
}
```

Kako NULL nije validna adresa, dereferenciranje NULL pokazivača dovodi do Segmentation Fault-a

NULL

```
a = 0
/bin/bash: line 1: 24989 Segmentation fault      (core dumped) ./run
```

Kako NULL nije validna adresa, dereferenciranje NULL pokazivača dovodi do Segmentation Fault-a

Pokazivačka aritmetika

Identifikator niza sadrži adresu prvog elementa

```
#include <iostream>
#define LEN 10
using namespace std;

int main()
{
        short a[LEN];

        unsigned val = 1;
        for(unsigned i = 0; i < LEN; ++i)
        {
                a[i] = val;
                val *= 2;
        }

        for(unsigned i = 0; i < LEN - 1; ++i)
                cout << a[i] << ", ";
        cout << a[LEN - 1] << endl;

        cout << "Prvi element niza a je " << a[0] << endl;
        cout << "A moze i ovako " << *a << endl;

        return 0;
}
```

Identifikator niza sadrži adresu prvog elementa

```
1, 2, 4, 8, 16, 32, 64, 128, 256, 512
Prvi element niza a je 1
A moze i ovako 1
```

Identifikator niza sadrži adresu prvog elementa

```cpp
#include <iostream>
#define LEN 10
using namespace std;

int main()
{
        short a[LEN];

        unsigned val = 1;
        for(unsigned i = 0; i < LEN; ++i)
        {
                a[i] = val;
                val *= 2;
        }

        for(unsigned i = 0; i < LEN - 1; ++i)
                cout << a[i] << ", ";
        cout << a[LEN - 1] << endl;

        cout << "Prvi element niza a je " << a[0] << endl;
        cout << "A moze i ovako " << *(a + 0 * sizeof(short)) << endl;

        return 0;
}
```

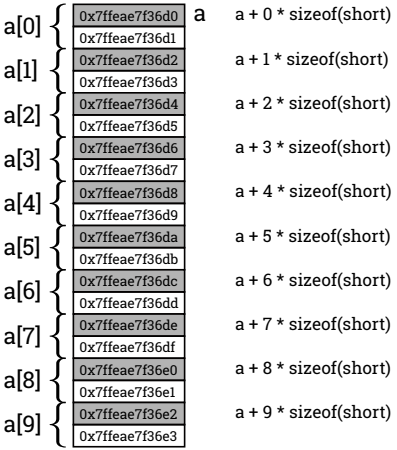
Identifikator niza sadrži adresu prvog elementa

```
1, 2, 4, 8, 16, 32, 64, 128, 256, 512
Prvi element niza a je 1
A moze i ovako 1
```

Setimo se adresa elemenata niza u memoriji

short a[LEN];

Rezerviši u memoriji
prostor veličine LEN * sizeof(short)
i nazovi adresu početka tog prostora a



Da li će ovo dati očekivan rezultat?

```cpp
#include <iostream>
#define LEN 10
using namespace std;

int main()
{
        short a[LEN];

        unsigned val = 1;
        for(unsigned i = 0; i < LEN; ++i)
        {
                a[i] = val;
                val *= 2;
        }

        for(unsigned i = 0; i < LEN - 1; ++i)
                cout << a[i] << ", ";
        cout << a[LEN - 1] << endl;

        cout << "Treci element niza a je " << a[2] << endl;
        cout << "A moze i ovako " << *(a + 2 * sizeof(short)) << endl;

        return 0;
}
```

Ipak neće

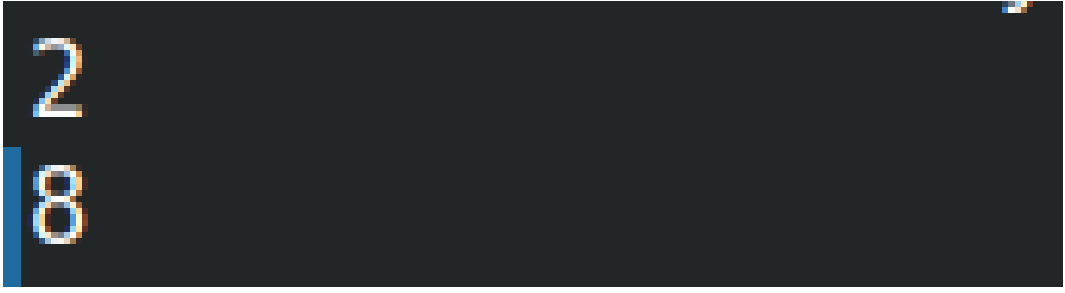
```
1, 2, 4, 8, 16, 32, 64, 128, 256, 512
Treci element niza a je 4
A moze i ovako 16
```

Umesto elementa na poziciji 2, dobili smo element na poziciji 4

Šta se dogodilo?

```cpp
#include <iostream>
using namespace std;

int main()
{
        short a = 7;
        short* p = &a;

        cout << sizeof(short) << endl;
        cout << long(p + 2 * sizeof(short)) - long(p) << endl;
        //Ocekujemo razliku u adresama od 2 * sizeof(short) = 4 (na ovom racunaru)

        return 0;
}
```

Šta se dogodilo?



U pokazivačkoj aritmetici tip pokazivača određuje množilac celobrojnog operanda

```cpp
#include <iostream>
using namespace std;

int main()
{
        short a = 7;
        short* p = &a;

        cout << sizeof(short) << endl;
        cout << long(p + 2) - long(p) << endl;
        //Ocekujemo razliku u adresama od 2 * sizeof(short) = 4 (na ovom racunaru)

        return 0;
}
```

U pokazivačkoj aritmetici tip pokazivača određuje množilac celobrojnog operanda

```
2
4
```

Jedan opštiji primer

```
#include <iostream>
#include <cstdio>
using namespace std;

int main()
{
        char a_char;
        short a_short;
        int a_int;
        long a_long;
        long long a_long_long;

        char* p_char = &a_char;
        short* p_short = &a_short;
        int* p_int = &a_int;
        long* p_long = &a_long;
        long long* p_long_long = &a_long_long;
```

Jedan opštiji primer

```
        cout << "CHAR\n----------------\n";

        cout << "p_char + 1 - p_char = " << long(p_char + 1) - long(p_char) << endl;
        cout << "sizeof(char) = " << sizeof(char) << "\n\n";

        cout << "SHORT\n----------------\n";
        cout << "p_short + 1 - p_short = " << long(p_short + 1) - long(p_short) << endl;
        cout << "sizeof(short) = " << sizeof(short) << "\n\n";

        cout << "INT\n----------------\n";
        cout << "p_int + 1 - p_int = " << long(p_int + 1) - long(p_int) << endl;
        cout << "sizeof(int) = " << sizeof(int) << "\n\n";

        cout << "LONG\n----------------\n";
        cout << "p_long + 1 - p_long = " << long(p_long + 1) - long(p_long) << endl;
        cout << "sizeof(long) = " << sizeof(long) << "\n\n";

        cout << "LONG LONG\n----------------\n";
        cout << "p_long_long + 1 - p_long_long = " << long(p_long_long + 1) - long(p_long_long) << endl;
        cout << "sizeof(long long) = " << sizeof(long long) << "\n\n";

        return 0;
}
```

Jedan opštiji primer

```
CHAR
----------------
p_char + 1 - p_char = 1
sizeof(char) = 1

SHORT
----------------
p_short + 1 - p_short = 2
sizeof(short) = 2

INT
----------------
p_int + 1 - p_int = 4
sizeof(int) = 4

LONG
----------------
p_long + 1 - p_long = 8
sizeof(long) = 8

LONG LONG
----------------
p_long_long + 1 - p_long_long = 8
sizeof(long long) = 8
```

Indeksiranje dereferenciranjem pokazivača

```cpp
#include <iostream>
#define LEN 10
using namespace std;

int main()
{
        short a[LEN];

        unsigned val = 1;
        for(unsigned i = 0; i < LEN; ++i)
        {
                a[i] = val;
                val *= 2;
        }

        for(unsigned i = 0; i < LEN - 1; ++i)
                cout << a[i] << ", ";
        cout << a[LEN - 1] << endl;

        cout << "Treci element niza a je " << a[2] << endl;
        cout << "A moze i ovako " << *(a + 2) << endl;

        return 0;
}
```

Indeksiranje dereferenciranjem pokazivača

```
1, 2, 4, 8, 16, 32, 64, 128, 256, 512
Treci element niza a je 4
A moze i ovako 4
```

U opštem slučaju

a[i] $\iff$ *(a + i)

Za upotrebom pokazivačke aritmetike retko ima potrebe i ona je često izvor grešaka

U C++-u ipak postoje neke razlike između identifikatora nizova i običnih pokazvača

Primer

```cpp
#include <iostream>
#define LEN 10
using namespace std;

int main()
{
        short a[LEN];

        for(unsigned i = 0; i < LEN; ++i)
                a[i] = 2 * LEN - i;

        for(unsigned i = 0; i < LEN - 1; ++i)
                cout << a[i] << ", ";
        cout << a[LEN - 1] << endl;

        return 0;
}
```

Primer

```
20, 19, 18, 17, 16, 15, 14, 13, 12, 11
```

Primer: identifikatoru niza ne možemo dodeliti drugi niz

```
#include <iostream>
#define LEN 10
using namespace std;

int main()
{
        short a[LEN];

        for(unsigned i = 0; i < LEN; ++i)
                a[i] = 2 * LEN - i;

        for(unsigned i = 0; i < LEN - 1; ++i)
                cout << a[i] << ", ";
        cout << a[LEN - 1] << endl;

        short b[LEN] = a;

        return 0;
}
```

Primer: identifikatoru niza ne možemo dodeliti drugi niz

```
niz_cp.cpp: In function ‘int main()’:
niz_cp.cpp:16:24: error: array must be initialized with a brace-enclosed initializer
   16 |         short b[LEN] = a;
      |                        ^

shell returned 1
```

Primer: ni izvan inicijalizacije

```
#include <iostream>
#define LEN 10
using namespace std;

int main()
{
        short a[LEN];

        for(unsigned i = 0; i < LEN; ++i)
                a[i] = 2 * LEN - i;

        for(unsigned i = 0; i < LEN - 1; ++i)
                cout << a[i] << ", ";
        cout << a[LEN - 1] << endl;

        short b[LEN];
        b = a;

        return 0;
}
```

Primer: ni izvan inicijalizacije

```
niz_cp.cpp: In function ‘int main()’:
niz_cp.cpp:17:11: error: invalid array assignment
   17 |         b = a;
      |         ~~^~~

shell returned 1
```

Primer: pokazivaču možemo

```
#include <iostream>
#define LEN 10
using namespace std;

int main()
{
        short a[LEN];

        for(unsigned i = 0; i < LEN; ++i)
                a[i] = 2 * LEN - i;

        for(unsigned i = 0; i < LEN - 1; ++i)
                cout << a[i] << ", ";
        cout << a[LEN - 1] << endl;

        short* b;
        b = a;

        return 0;
}
```

Primer: pokazivaču možemo

```
20, 19, 18, 17, 16, 15, 14, 13, 12, 11
```

Primer: pokazivač nakon toga možemo koristiti za pristupanje elementima niza

```cpp
#include <iostream>
#define LEN 10
using namespace std;

int main()
{
        short a[LEN];

        for(unsigned i = 0; i < LEN; ++i)
                a[i] = 2 * LEN - i;

        for(unsigned i = 0; i < LEN - 1; ++i)
                cout << a[i] << ", ";
        cout << a[LEN - 1] << endl;

        short* b;
        b = a;

        for(unsigned i = 0; i < LEN - 1; ++i)
                cout << b[i] << ", ";
        cout << b[LEN - 1] << endl;

        return 0;
}
```

Primer: pokazivač nakon toga možemo koristiti za pristupanje elementima niza

```
20, 19, 18, 17, 16, 15, 14, 13, 12, 11
20, 19, 18, 17, 16, 15, 14, 13, 12, 11
```

Primer: to uključuje i izmene

```
#include <iostream>
#define LEN 10
using namespace std;

int main()
{
        short a[LEN];

        for(unsigned i = 0; i < LEN; ++i)
                a[i] = 2 * LEN - i;

        for(unsigned i = 0; i < LEN - 1; ++i)
                cout << a[i] << ", ";
        cout << a[LEN - 1] << endl;

        short* b;
        b = a;

        b[3] = 1000;

        for(unsigned i = 0; i < LEN - 1; ++i)
                cout << b[i] << ", ";
        cout << b[LEN - 1] << endl;

        for(unsigned i = 0; i < LEN - 1; ++i)
                cout << a[i] << ", ";
        cout << a[LEN - 1] << endl;


        return 0;
}
```

Primer: to uključuje i izmene

```
20, 19, 18, 17, 16, 15, 14, 13, 12, 11
20, 19, 18, 1000, 16, 15, 14, 13, 12, 11
20, 19, 18, 1000, 16, 15, 14, 13, 12, 11
```

Primetimo da a i b pokazuju na isti fizički prostor u memoriji. b nam je dakle samo drugo ime za a.

Ako nam je potrebna kopija, moramo deklarisati novi niz i prekopirati sadržaj

```cpp
#include <iostream>
#define LEN 10
using namespace std;

int main()
{
        short a[LEN];

        for(unsigned i = 0; i < LEN; ++i)
                a[i] = 2 * LEN - i;

        for(unsigned i = 0; i < LEN - 1; ++i)
                cout << a[i] << ", ";
        cout << a[LEN - 1] << endl;

        short b[LEN];
        for(unsigned i = 0; i < LEN; ++i)
                b[i] = a[i];

        b[3] = 1000;

        for(unsigned i = 0; i < LEN - 1; ++i)
                cout << b[i] << ", ";
        cout << b[LEN - 1] << endl;

        for(unsigned i = 0; i < LEN - 1; ++i)
                cout << a[i] << ", ";
        cout << a[LEN - 1] << endl;

        return 0;
}
```

Ako nam je potrebna kopija, moramo deklarisati novi niz i prekopirati sadržaj

```
20, 19, 18, 17, 16, 15, 14, 13, 12, 11
20, 19, 18, 1000, 16, 15, 14, 13, 12, 11
20, 19, 18, 17, 16, 15, 14, 13, 12, 11
```

Inicijalizacija nizova

tip identifikator[dužina niza] = {element0, element1, ...}

Primer

```cpp
#include <iostream>
#define LEN 24

using namespace std;

int main()
{
        unsigned a[LEN] = {17, 23, 4, 11, 120, 147, 79, 44, 31, 9, 12, 27, 3, 99, 234, 37, 84, 102, 65, 201, 13, 42, 6, 16};

        for(unsigned i = 0; i < LEN - 1; ++i)
                cout << a[i] << ", ";
        cout << a[LEN - 1] << endl;

        return 0;
}
```

Primer

```
17, 23, 4, 11, 120, 147, 79, 44, 31, 9, 12, 27, 3, 99, 234, 37, 84, 102, 65, 201, 13, 42, 6, 16
```

Višedimenzioni nizovi

```
#include <iostream>
#include <climits>

#define V 7
#define NOT_REACHED -1
#define INF UCHAR_MAX

using namespace std;

int main()
{
        unsigned char adj[V][V] = {
                                    {0, 2, 6, 9, INF, INF, INF},
                                    {INF, 0, INF, 5, INF, INF, INF},
                                    {INF, INF, 0, 8, INF, INF, INF},
                                    {INF, INF, INF, 0, 10, INF, INF},
                                    {INF, INF, INF, INF, 0, 6, 2},
                                    {INF, INF, INF, INF, INF, 0, 6},
                                    {INF, INF, INF, INF, INF, INF, 0}
                                   };

        const unsigned char s = 0;
        const unsigned char t = 6;
```

Višedimenzioni nizovi: deklaracija

tip identifikator[dimenzija 0][dimenzija 1][dimenzija 2][...;

Višedimenzioni nizovi: indeksiranje

identifikator[indeks 0][indeks 1][indeks 2][...;

Primer

```
67                  for(unsigned char v = 0; v < V; ++v)
68                  {
69                          if(adj[u][v] < INF)
70                          {
71                                  if(dist_from_s[v] > dist_from_s[u] + adj[u][v])
72                                  {
73                                          reached_from[v] = u;
74                                          dist_from_s[v] = dist_from_s[u] + adj[u][v];
75                                  }
76                          }
77                  }
78          }
79
```

Neke operacije nad nizovima

Potraga za najlepšim draguljem

Broj odgovara lepoti dragulja

```
17, 23, 4, 11, 120, 147, 79, 44, 31, 9, 12, 27, 3, 99, 234, 37, 84, 102, 65, 201, 13, 42, 6, 16
```

Patuljak Gimli traži najlepši dragulj. Kako da mu pomognemo?

Ideja

1. Uzmi prvi dragulj na koji naiđeš i proglasi ga najlepšim
2. Svaki sledeći dragulj na koji naiđeš uporedi sa onim koji ti je u ruci
3. Ako ti je taj lepši, uzmi njega i odbaci onaj koji si ranije imao
4. Kada prođeš sve dragulje, znaćeš da u ruci imaš najlepši

Gimli je malo lenj...

- Dobro, pa koliko ću to poređenja morati da napravim?

Moraćemo da napravimo $n - 1$ poređenja, gde je n dužina niza

Naš algoritam ima složenost $\mathcal{O}(n)$ (linearan je)[3]

[3] Preciznije, $\Theta(n)$, jer će i u najgorem i u najboljem slučaju zahtevati $n - 1$ poređenja, ali to je sada detalj.

Implementacija u C++-u

```cpp
#include <iostream>
#define LEN 24

using namespace std;

int main()
{
        unsigned a[LEN] = {17, 23, 4, 11, 120, 147, 79, 44, 31, 9, 12, 27, 3, 99, 234, 37, 84, 102, 65, 201, 13, 42, 6, 16};

        unsigned max_val = a[0];
        unsigned max_ind = 0;
        for(unsigned i = 1; i < LEN; ++i)
        {
                if(a[i] > max_val)
                {
                        max_val = a[i];
                        max_ind = i;
                }
        }

        cout << "Najveci element niza ";
        for(unsigned i = 0; i < LEN; ++i)
                cout << a[i] << ", ";

        cout << "\nje " << max_val << ", na poziciji " << max_ind << endl;

        return 0;
}
```

Implementacija u C++-u

```
Najveci element niza 17, 23, 4, 11, 120, 147, 79, 44, 31, 9, 12, 27, 3, 99, 234, 37, 84, 102, 65, 201, 13, 42, 6, 16,
je 234, na poziciji 14
```

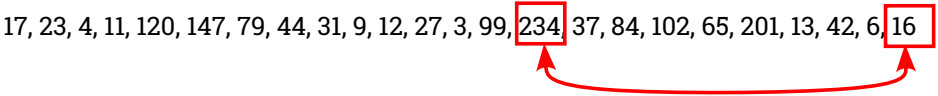
Kako bismo našli k najvećih elemenata niza?

```cpp
#include <iostream>
#define LEN 24

using namespace std;

int main()
{
        unsigned a[LEN] = {17, 23, 4, 11, 120, 147, 79, 44, 31, 9, 12, 27, 3, 99, 234, 37, 84, 102, 65, 201, 13, 42, 6, 16};

        unsigned max_val = a[0];
        unsigned max_ind = 0;
        for(unsigned i = 1; i < LEN; ++i)
        {
                if(a[i] > max_val)
                {
                        max_val = a[i];
                        max_ind = i;
                }
        }

        cout << "Najveci element niza ";
        for(unsigned i = 0; i < LEN; ++i)
                cout << a[i] << ", ";

        cout << "\nje " << max_val << ", na poziciji " << max_ind << endl;

        return 0;
}
```

Korak 1: Nađemo maksimum niza

17, 23, 4, 11, 120, 147, 79, 44, 31, 9, 12, 27, 3, 99, 234, 37, 84, 102, 65, 201, 13, 42, 6, 16

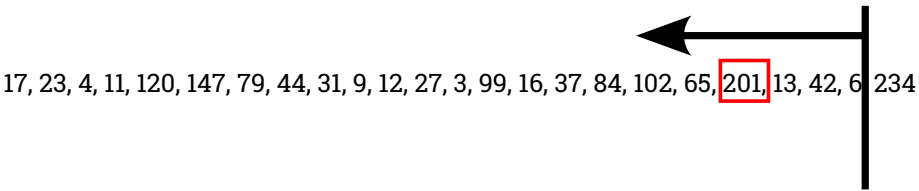
Korak 2: Zamenimo maksimum sa elementom na poziciji $n - 1$ (krajnjim desnim)



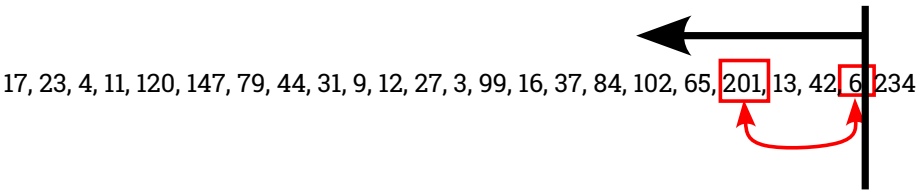
Korak 2: Zamenimo maksimum sa elementom na poziciji $n - 1$ (krajnjim desnim)

17, 23, 4, 11, 120, 147, 79, 44, 31, 9, 12, 27, 3, 99, 16, 37, 84, 102, 65, 201, 13, 42, 6, 234

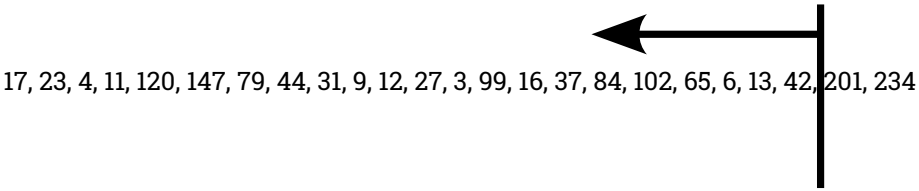
Korak 3: Tražimo maksimum ostatka niza i menjamo ga sa elementom na poziciji $n - 2$



Korak 3: Tražimo maksimum ostatka niza i menjamo ga sa elementom na poziciji $n - 2$



Nastavljamo dalje dok ne završimo svih k iteracija



k najvećih elemenata će se izdvojiti na kraju niza

Gimli je malo lenj...

- Dobro, pa koliko ću to sad poređenja morati da napravim?!

Analiza složenosti

Za najveći element imamo $n - 1$ poređenja

Za drugi najveći $n - 2$

Za treći najveći $n - 3$

...

Za k-ti najveći $n - k$

Analiza složenosti

Ukupno: $kn - \sum_{i=1}^{k} i = kn - (k+1)\frac{k}{2}$

Ako smatramo da k nije $f(n)$, algoritam će i dalje biti linearan u odnosu na dužinu niza

Implementacija u C++-u

```cpp
#include <iostream>
#define LEN 24

using namespace std;

int main()
{
        unsigned a[LEN] = {17, 23, 4, 11, 120, 147, 79, 44, 31, 9, 12, 27, 3, 99, 234, 37, 84, 102, 65, 201, 13, 42, 6, 16};

        unsigned k = 0;
        cout << "Unisite trazeni broj najvecih elemenata: ";
        cin >> k;

        cout << k << " najvecih elemenata niza ";
        for(unsigned i = 0; i < LEN; ++i)
                cout << a[i] << ", ";


        for(unsigned end = 0; end < k; ++end)
        {
                unsigned max_val = a[0];
                unsigned max_ind = 0;
                for(unsigned i = 0; i < LEN - end; ++i)
                {
                        if(a[i] > max_val)
                        {
                                max_val = a[i];
                                max_ind = i;
                        }
                }
                //I sada zamenimo trenutni maksimum sa odgovarajucom pozicijom na kraju
                a[max_ind] = a[LEN - 1 - end];
                a[LEN - 1 - end] = max_val;
        }
```

Implementacija u C++-u

```
        cout << "\nje ";
        for(unsigned i = LEN - 1; i > LEN - k; --i)
                cout << a[i] << ", ";
        cout << a[LEN - k] << endl;

        return 0;
}
```

Implementacija u C++-u

```
Unisite trazeni broj najvecih elemenata: 5
5 najvecih elemenata niza 17, 23, 4, 11, 120, 147, 79, 44, 31, 9, 12, 27, 3, 99, 234, 37, 84, 102, 65, 201, 13, 42, 6, 16,
je 234, 201, 147, 120, 102
```

Šta ćemo dobiti ako kao k unesemo n?

```cpp
#include <iostream>
#define LEN 24

using namespace std;

int main()
{
	unsigned a[LEN] = {17, 23, 4, 11, 120, 147, 79, 44, 31, 9, 12, 27, 3, 99, 234, 37, 84, 102, 65, 201, 13, 42, 6, 16};

	unsigned k = 0;
	cout << "Unisite trazeni broj najvecih elemenata: ";
	cin >> k;

	cout << k << " najvecih elemenata niza ";
	for(unsigned i = 0; i < LEN; ++i)
		cout << a[i] << ", ";


	for(unsigned end = 0; end < k; ++end)
	{
		unsigned max_val = a[0];
		unsigned max_ind = 0;
		for(unsigned i = 0; i < LEN - end; ++i)
		{
			if(a[i] > max_val)
			{
				max_val = a[i];
				max_ind = i;
			}
		}
		//I sada zamenimo trenutni maksimum sa odgovarajucom pozicijom na kraju
		a[max_ind] = a[LEN - 1 - end];
		a[LEN - 1 - end] = max_val;
	}
```

Dobićemo sortiran niz

```
Unisite trazeni broj najvecih elemenata: 24
24 najvecih elemenata niza 17, 23, 4, 11, 120, 147, 79, 44, 31, 9, 12, 27, 3, 99, 234, 37, 84, 102, 65, 201, 13, 42, 6, 16,
je 234, 201, 147, 120, 102, 99, 84, 79, 65, 44, 42, 37, 31, 27, 23, 17, 16, 13, 12, 11, 9, 6, 4, 3
```

Kolika nam je sada složenost?

$$kn - \sum_{i=1}^{k} i = kn - (k+1)\frac{k}{2}$$

$$= n^2 - n^2/2 - n/2 = n^2/2 - n/2 = \frac{n(n-1)}{2}$$

$$= \mathcal{O}(n^2)$$

Naš algoritam za sortiranje je kvadratni[4]

[4] Ponovo $\Theta(n^2)$

Ovaj algoritam nazivamo *Selection Sort*

```cpp
#include <iostream>
#define LEN 24

using namespace std;

int main()
{
	unsigned a[LEN] = {17, 23, 4, 11, 120, 147, 79, 44, 31, 9, 12, 27, 3, 99, 234, 37, 84, 102, 65, 201, 13, 42, 6, 16};

	for(unsigned beg = 0; beg < LEN; ++beg)
	{
		unsigned min_val = a[beg];
		unsigned min_ind = beg;
		for(unsigned i = beg + 1; i < LEN; ++i)
		{
			if(a[i] < min_val)
			{
				min_val = a[i];
				min_ind = i;
			}
		}
		//I sada zamenimo trenutni minimum sa odgovarajucom pozicijom na pocetku
		a[min_ind] = a[beg];
		a[beg] = min_val;
	}


	for(unsigned i = 0; i < LEN - 1; ++i)
		cout << a[i] << ", ";
	cout << a[LEN - 1] << endl;

	return 0;
}
```

Naravno, možemo ga implementirati i smeštanjem minimuma na početak niza

Šta u praksi znači da je algoritam za sortiranje složenosti $\Theta(n^2)$?

To znači da će sortiranje zahtevati kvadratni broj operacija, bez obzira na to koliko je uneti niz delimično sortiran

Kako možemo izmeriti koliko je uneti niz sortiran?

Inverzija: za svako i, j, ako je $i < j$ a $a_i > a_j$, kažemo da za a_i i a_j postoji inverzija

Najviše inverzija ima niz sortiran u obrnutom poretku

Najveći broj inverzija: $\frac{n(n-1)}{2}$

Koliko poređenja napravi Selection Sort na našem primeru?

```cpp
using namespace std;

int main()
{
	unsigned a[LEN] = {17, 23, 4, 11, 120, 147, 79, 44, 31, 9, 12, 27, 3, 99, 234, 37, 84, 102, 65, 201, 13, 42, 6, 16};

	unsigned moves = 0;
	for(unsigned beg = 0; beg < LEN; ++beg)
	{
		unsigned min_val = a[beg];
		unsigned min_ind = beg;
		for(unsigned i = beg + 1; i < LEN; ++i)
		{
			++moves;
			if(a[i] < min_val)
			{
				min_val = a[i];
				min_ind = i;
			}
		}
		//I sada zamenimo trenutni minimum sa odgovarajucom pozicijom na pocetku
		a[min_ind] = a[beg];
		a[beg] = min_val;
	}

	cout << moves << endl;
	for(unsigned i = 0; i < LEN - 1; ++i)
		cout << a[i] << ", ";
	cout << a[LEN - 1] << endl;

	return 0;
}
```

Koliko poređenja napravi Selection Sort na našem primeru?

```
276
3, 4, 6, 9, 11, 12, 13, 16, 17, 23, 27, 31, 37, 42, 44, 65, 79, 84, 99, 102, 120, 147, 201, 234
```

$\frac{24 \times 23}{2} = 276$

Koliko inverzija ima naš niz? (Iskoristimo malu pomoć Python-a)

```
def count_inversions(a):
    inversions = 0
    for i in range(0, len(a)):
        for j in range(0, i):
            if a[i] < a[j]:
                inversions += 1

    return inversions

a = [17, 23, 4, 11, 120, 147, 79, 44, 31, 9, 12, 27, 3, 99, 234, 37, 84, 102, 65, 201, 13, 42, 6, 16]

inversions = count_inversions(a)
print inversions
```

Koliko inverzija ima naš niz? (Iskoristimo malu pomoć Python-a)

Kako da iskoristimo parcijalno sortiranje da bismo uštedeli vreme?

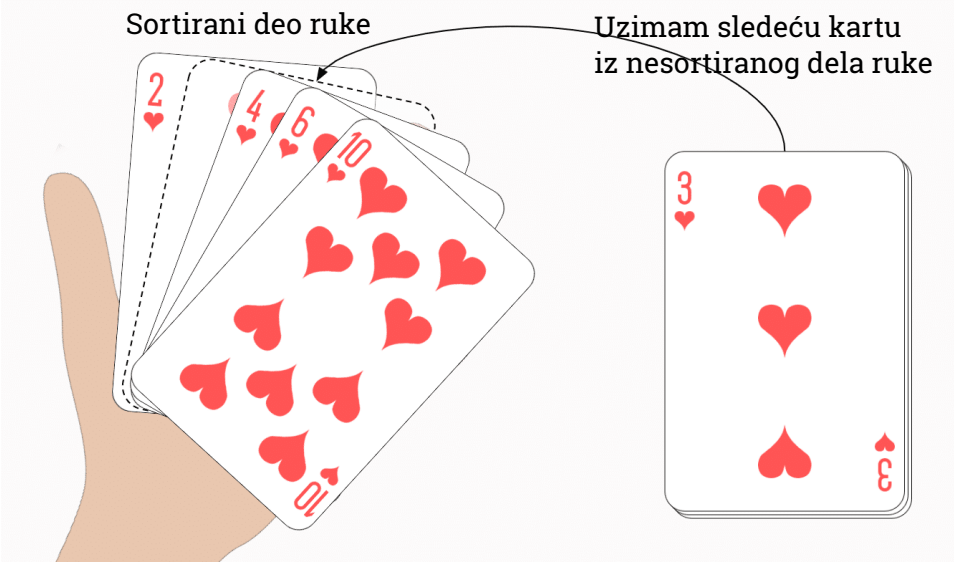
Jedne večeri, Gimli je svratio u konak da prenoći

Pa su ga neki drugi gosti ubedili da odigra partiju karata

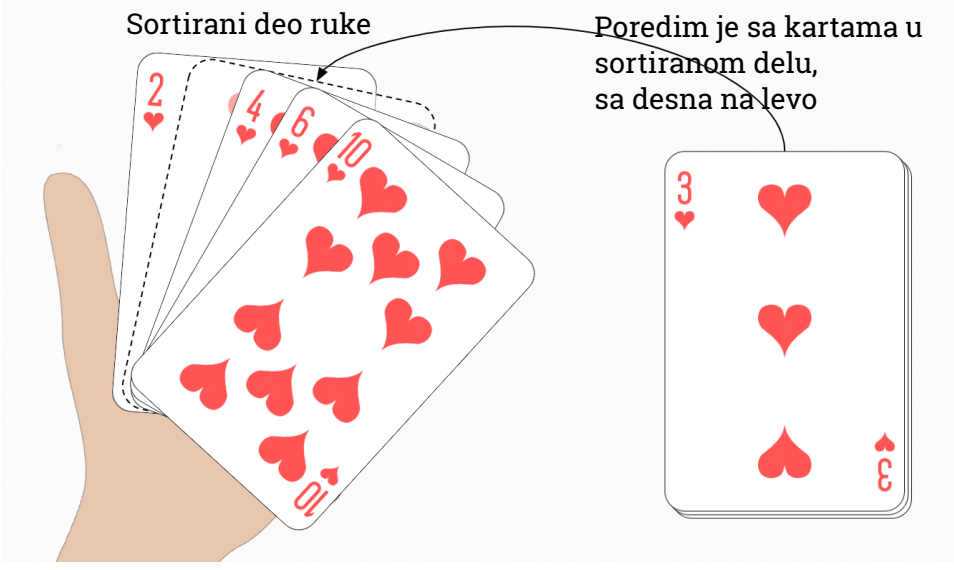
Jedne večeri, Gimli je svratio u konak da prenoći

I onda je došao na ideju

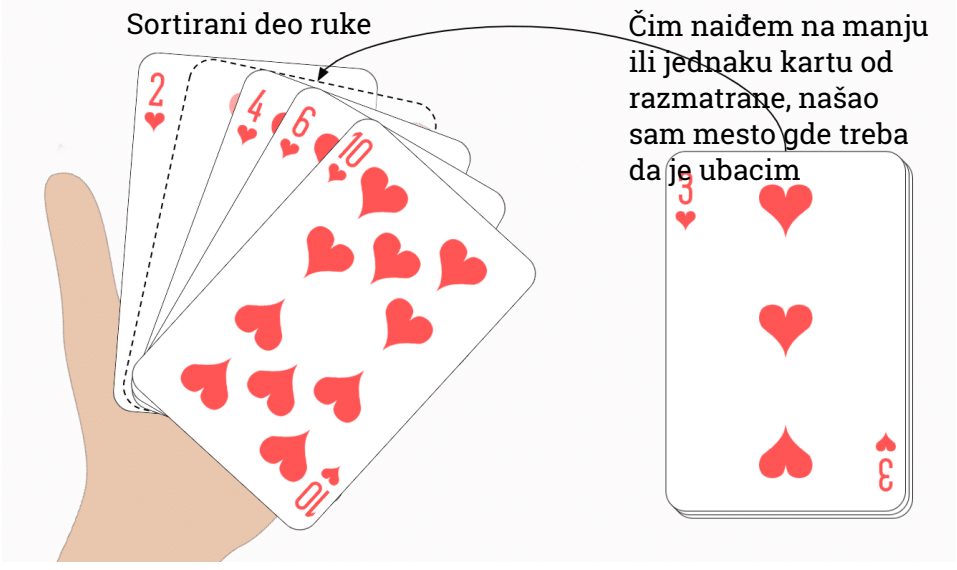
Gimlijev špil



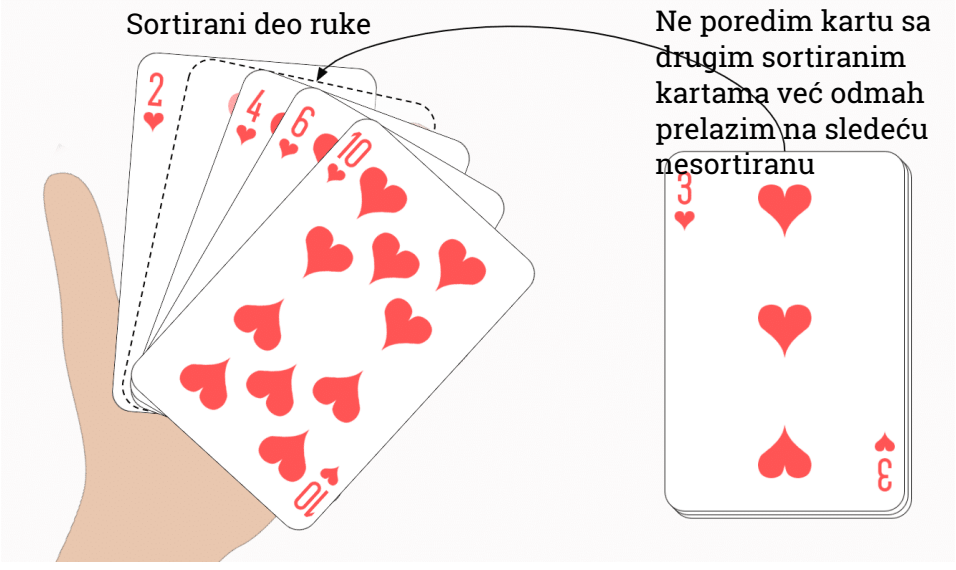
Gimlijev špil



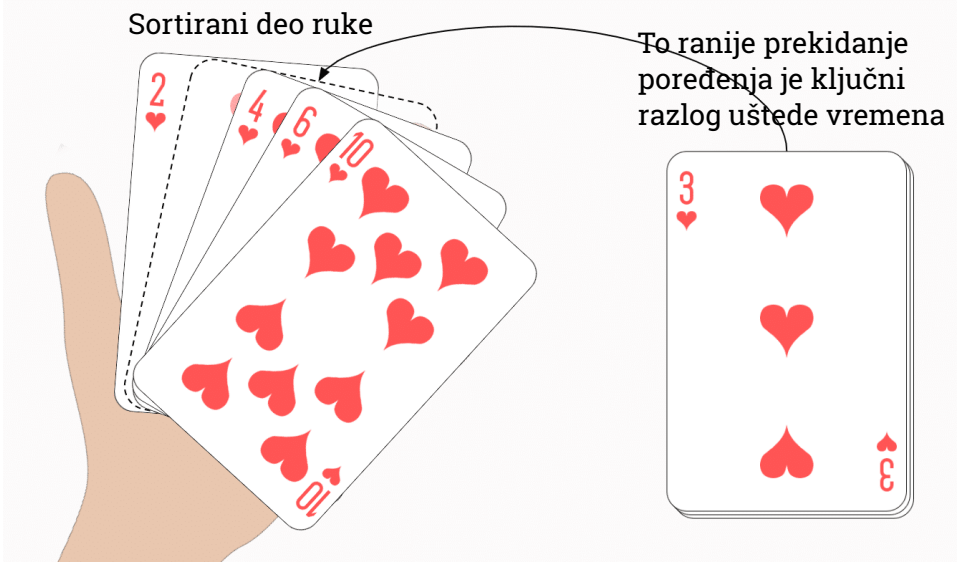
Gimlijev špil



Gimlijev špil



Gimlijev špil



Ovaj algoritam nazivamo *Insertion Sort*

Implementacija u C++-u

```cpp
#include <iostream>
#define LEN 24

using namespace std;

int main()
{
        unsigned a[LEN] = {17, 23, 4, 11, 120, 147, 79, 44, 31, 9, 12, 27, 3, 99, 234, 37, 84, 102, 65, 201, 13, 42, 6, 16};

        unsigned moves = 0;
        for(unsigned i = 1; i < LEN; ++i)
        {
                unsigned val = a[i];
                unsigned j = i;
                while((j > 0) && (val < a[j - 1])) //val jos nije stigao do svoje pozicije; moramo jos da idemo ulevo
                {
                        ++moves;
                        a[j] = a[j - 1]; //Pomeramo elemente udesno, da napravimo mesta za val
                        --j;
                }
                a[j] = val; //Nasli smo mesto za val, pa ga ubacujemo
        }


        cout << moves << endl;
        for(unsigned i = 0; i < LEN - 1; ++i)
                cout << a[i] << ", ";
        cout << a[LEN - 1] << endl;

        return 0;
}
```

Implementacija u C++-u

```
129
3, 4, 6, 9, 11, 12, 13, 16, 17, 23, 27, 31, 37, 42, 44, 65, 79, 84, 99, 102, 120, 147, 201, 234
```

Obzervacija: broj pomeranja odgovara broju inverzija u nizu

Da li obzervacija važi i za manje inverzija?

```python
import random

random.seed(0)

def count_inversions(a):
    inversions = 0
    for i in range(0, len(a)):
        for j in range(0, i):
            if a[i] < a[j]:
                inversions += 1

    return inversions

a = [17, 23, 4, 11, 120, 147, 79, 44, 31, 9, 12, 27, 3, 99, 234, 37, 84, 102, 65, 201, 13, 42, 6, 16]

inversions = count_inversions(a)

target_inversions = inversions / 2
while inversions > target_inversions:
    random.shuffle(a)
    inversions = count_inversions(a)

print a
print inversions
```

Ponovo u pomoć pozivamo Python

Da li obzervacija važi i za manje inverzija?

```
[6, 9, 23, 4, 3, 31, 16, 37, 11, 13, 42, 27, 102, 17, 147, 234, 79, 12, 44, 120, 84, 201, 65, 99]
64
```

Ponovo u pomoć pozivamo Python

Opet brojimo pomeranja (ovo naravno nije dokaz)

```cpp
#include <iostream>
#define LEN 24

using namespace std;

int main()
{
        unsigned a[LEN] = {6, 9, 23, 4, 3, 31, 16, 37, 11, 13, 42, 27, 102, 17, 147, 234, 79, 12, 44, 120, 84, 201, 65, 90};

        unsigned moves = 0;
        for(unsigned i = 1; i < LEN; ++i)
        {
                unsigned val = a[i];
                unsigned j = i;
                while((j > 0) && (val < a[j - 1])) //val jos nije stigao do svoje pozicije; moramo jos da idemo ulevo
                {
                        ++moves;
                        a[j] = a[j - 1]; //Pomeramo elemente udesno, da napravimo mesta za val
                        --j;
                }
                a[j] = val; //Nasli smo mesto za val, pa ga ubacujemo
        }


        cout << moves << endl;
        for(unsigned i = 0; i < LEN - 1; ++i)
                cout << a[i] << ", ";
        cout << a[LEN - 1] << endl;

        return 0;
}
```

Opet brojimo pomeranja (ovo naravno nije dokaz)

```
64
3, 4, 6, 9, 11, 12, 13, 16, 17, 23, 27, 31, 37, 42, 44, 65, 79, 84, 99, 102, 120, 147, 201, 234
```

Šta se dešava kada je niz već sortiran?

Tada nemamo pomeranja, ali ipak imamo poređenje u uslovu petlje, za svaki element niza

Stoga je u najboljem slučaju algoritam linearan

Da vidimo i najgori slučaj

```cpp
#include <iostream>
#define LEN 24

using namespace std;

int main()
{
        unsigned a[LEN] = {234, 201, 147, 120, 102, 99, 84, 79, 65, 44, 42, 37, 31, 27, 23, 17, 16, 13, 12, 11, 9, 6, 4, 3};

        unsigned moves = 0;
        for(unsigned i = 1; i < LEN; ++i)
        {
                unsigned val = a[i];
                unsigned j = i;
                while((j > 0) && (val < a[j - 1])) //val jos nije stigao do svoje pozicije; moramo jos da idemo ulevo
                {
                        ++moves;
                        a[j] = a[j - 1]; //Pomeramo elemente udesno, da napravimo mesta za val
                        --j;
                }
                a[j] = val; //Nasli smo mesto za val, pa ga ubacujemo
        }


        cout << moves << endl;
        for(unsigned i = 0; i < LEN - 1; ++i)
                cout << a[i] << ", ";
        cout << a[LEN - 1] << endl;

        return 0;
}
```

Da vidimo i najgori slučaj

```
276
3, 4, 6, 9, 11, 12, 13, 16, 17, 23, 27, 31, 37, 42, 44, 65, 79, 84, 99, 102, 120, 147, 201, 234
```

U najgorem slučaju, Insertion Sort se asimptotski ponaša jednako kao i Selection Sort

Teoriju najčešće zanima najgori slučaj, ali je u praksi prosečan slučaj često značajniji

Šta je ključni problem Insertion Sort-a?

```cpp
#include <iostream>
#define LEN 24

using namespace std;

int main()
{
        unsigned a[LEN] = {234, 201, 147, 120, 102, 99, 84, 79, 65, 44, 42, 37, 31, 27, 23, 17, 16, 13, 12, 11, 9, 6, 4, 3};

        unsigned moves = 0;
        for(unsigned i = 1; i < LEN; ++i)
        {
                unsigned val = a[i];
                unsigned j = i;
                while((j > 0) && (val < a[j - 1])) //val jos nije stigao do svoje pozicije; moramo jos da idemo ulevo
                {
                        ++moves;
                        a[j] = a[j - 1]; //Pomeramo elemente udesno, da napravimo mesta za val
                        --j;
                }
                a[j] = val; //Nasli smo mesto za val, pa ga ubacujemo
        }


        cout << moves << endl;
        for(unsigned i = 0; i < LEN - 1; ++i)
                cout << a[i] << ", ";
        cout << a[LEN - 1] << endl;

        return 0;
}
```

Šta je ključni problem Insertion Sort-a?

```cpp
#include <iostream>
#define LEN 24

using namespace std;

int main()
{
        unsigned a[LEN] = {234, 201, 147, 120, 102, 99, 84, 79, 65, 44, 42, 37, 31, 27, 23, 17, 16, 13, 12, 11, 9, 6, 4, 3};

        unsigned moves = 0;
        for(unsigned i = 1; i < LEN; ++i)
        {
                unsigned val = a[i];
                unsigned j = i;
                while((j > 0) && (val < a[j - 1])) //val jos nije stigao do svoje pozicije; moramo jos da idemo ulevo
                {
                        ++moves;
                        a[j] = a[j - 1]; //Pomeramo elemente udesno, da napravimo mesta za val
                        --j;
                }
                a[j] = val; //Nasli smo mesto za val, pa ga ubacujemo
        }


        cout << moves << endl;
        for(unsigned i = 0; i < LEN - 1; ++i)
                cout << a[i] << ", ";
        cout << a[LEN - 1] << endl;

        return 0;
}
```

Ako su neki elementi
jako daleko od svog konačnog
mesta, njihovo postavljanje na to
mesto zahteva jako veliki broj
koraka (velik broj inverzija)

Kako da rešimo taj problem?

```cpp
#include <iostream>
#define LEN 24

using namespace std;

int main()
{
        unsigned a[LEN] = {234, 201, 147, 120, 102, 99, 84, 79, 65, 44, 42, 37, 31, 27, 23, 17, 16, 13, 12, 11, 9, 6, 4, 3};

        unsigned moves = 0;
        for(unsigned i = 1; i < LEN; ++i)
        {
                unsigned val = a[i];
                unsigned j = i;
                while((j > 0) && (val < a[j - 1])) //val jos nije stigao do svoje pozicije; moramo jos da idemo ulevo
                {
                        ++moves;
                        a[j] = a[j - 1]; //Pomeramo elemente udesno, da napravimo mesta za val
                        --j;
                }
                a[j] = val; //Nasli smo mesto za val, pa ga ubacujemo
        }


        cout << moves << endl;
        for(unsigned i = 0; i < LEN - 1; ++i)
                cout << a[i] << ", ";
        cout << a[LEN - 1] << endl;

        return 0;
}
```

Ako su neki elementi
jako daleko od svog konačnog
mesta, njihovo postavljanje na to
mesto zahteva jako veliki broj
koraka (velik broj inverzija)

Broj kome tražimo mesto poredimo samo sa brojevima na velikoj udaljenosti

17, 23, 4, 11, 120, 147, 79, 44, 31, 9, 12, 27, 3, 99, 234, 37, 84, 102, 65, 201, 13, 42, 6, 16

Broj kome tražimo mesto poredimo samo sa brojevima na velikoj udaljenosti

17, 23, 4, 11, 120, 147, 79, 44, 31, 9, 12, 27, 3, 99, 234, 37, 84, 102, 65, 201, 13, 42, 6, 16

Broj kome tražimo mesto poredimo samo sa brojevima na velikoj udaljenosti

17, 23, 4, 11, 120, 147, 79, 44, 31, 9, 12, 27, 3, 99, 234, 37, 84, 102, 65, 201, 13, 42, 6, 16

Broj kome tražimo mesto poredimo samo sa brojevima na velikoj udaljenosti

17, 23, 4, 11, 120, 147, 79, 44, 31, 9, 12, 27, 3, 99, 234, 37, 84, 102, 65, 201, 13, 42, 6, 16

Broj kome tražimo mesto poredimo samo sa brojevima na velikoj udaljenosti

17, 23, 4, 11, 120, 147, 79, 44, 31, 9, 12, 27, 3, 99, 234, 37, 84, 102, 65, 201, 13, 42, 6, 16

Broj kome tražimo mesto poredimo samo sa brojevima na velikoj udaljenosti

17, 23, 4, 11, 102, 147, 79, 44, 31, 9, 12, 27, 3, 99, 234, 37, 84, 120, 65, 201, 13, 42, 6, 16

Broj kome tražimo mesto poredimo samo sa brojevima na velikoj udaljenosti

17, 23, 4, 11, 102, 147, 79, 44, 31, 9, 12, 27, 3, 99, 234, 37, 84, 120, 65, 201, 13, 42, 6, 16

Broj kome tražimo mesto poredimo samo sa brojevima na velikoj udaljenosti

17, 23, 4, 11, 102, 65 79, 44, 31, 9, 12, 27, 3, 99, 234, 37, 84, 120, 147, 201, 13, 42, 6, 16

Broj kome tražimo mesto poredimo samo sa brojevima na velikoj udaljenosti

17, 23, 4, 11, 102, 65 79, 44, 31, 9, 12, 27, 3, 99, 234, 37, 84, 120, 147, 201, 13, 42, 6, 16

Broj kome tražimo mesto poredimo samo sa brojevima na velikoj udaljenosti

17, 23, 4, 11, 102, 65 79, 44, 31, 9, 12, 27, 3, 99, 234, 37, 84, 120, 147, 201, 13, 42, 6, 16

Broj kome tražimo mesto poredimo samo sa brojevima na velikoj udaljenosti

17, 23, 4, 11, 102, 65 79, 13, 31, 9, 12, 27, 3, 99, 234, 37, 84, 120, 147, 201, 44, 42, 6, 16

Broj kome tražimo mesto poredimo samo sa brojevima na velikoj udaljenosti

17, 23, 4, 11, 102, 65 79, 13, 31, 9, 12, 27, 3, 99, 234, 37, 84, 120, 147, 201, 44, 42, 6, 16

Broj kome tražimo mesto poredimo samo sa brojevima na velikoj udaljenosti

17, 23, 4, 11, 102, 65 79, 13, 31, 9, 12, 27, 3, 99, 234, 37, 84, 120, 147, 201, 44, 42, 6, 16

Broj kome tražimo mesto poredimo samo sa brojevima na velikoj udaljenosti

17, 23, 4, 11, 102, 65 79, 13, 31, 6, 12, 27, 3, 99, 234, 37, 84, 120, 147, 201, 44, 42, 9, 16

Broj kome tražimo mesto poredimo samo sa brojevima na velikoj udaljenosti

17, 23, 4, 11, 102, 65 79, 13, 31, 6, 12, 27, 3, 99, 234, 37, 84, 120, 147, 201, 44, 42, 9, 16

Da vidimo šta smo uradili

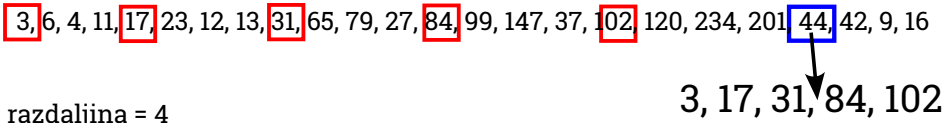
```
a = [17, 23, 4, 11, 120, 147, 79, 44, 31, 9, 12, 27, 3, 99, 234, 37, 84, 102, 65, 201, 13, 42, 6, 16]
print count_inversions(a)

a = [17, 23, 4, 11, 102, 65, 79, 13, 31, 6, 12, 27, 3, 99, 234, 37, 84, 120, 147, 201, 44, 42, 9, 16]
print count_inversions(a)
```

Da vidimo šta smo uradili

```
129
111
```

Pomoću 4 zamene smo smanjili broj inverzija za 18

Umesto da sada odmah radimo klasičan Insertion Sort, smanjujemo razmak



Umesto da sada odmah radimo klasičan Insertion Sort, smanjujemo razmak

3, 6, 4, 11, 17, 23, 12, 13, 31, 65, 79, 27, 44, 99, 147, 37, 84, 120, 234, 201, 102, 42, 9, 16

razdaljina = 4

Umesto da sada odmah radimo klasičan Insertion Sort, smanjujemo razmak

3, 6, 4, 11, 17, 23, 12, 13, 31, 65, 79, 27, 44, 99, 147, 37, 84, 120, 234, 201, 102, 42, 9, 16

razdaljina = 4

6, 23, 65, 99, 120

Umesto da sada odmah radimo klasičan Insertion Sort, smanjujemo razmak

3, 6, 4, 11, 17, 23, 12, 13, 31, 42, 79, 27, 44, 65, 147, 37, 84, 99, 234, 201, 102, 120, 9, 16

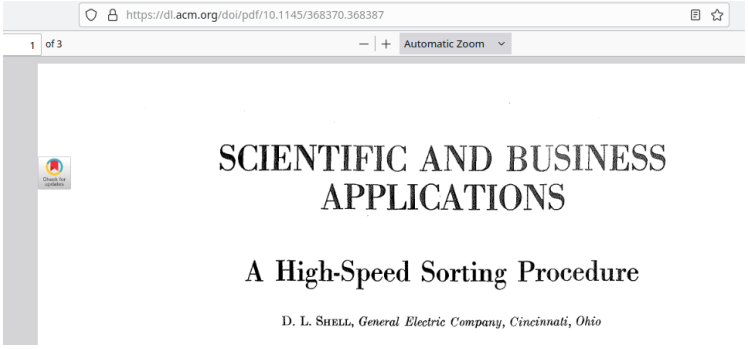
razdaljina = 4

I nastavljamo tako dok nam razmak ne opadne na 1

Ideja je da postepeno smanjujemo broj inverzija, pa se pri svakoj iteraciji Insertion Sort sa razmacima brže izvršava

Dokle god nam je rad uložen u presortiranje manji od uštede zbog smanjenja broja inverzija, bićemo na dobitku

Ovaj algoritam nazivamo *Shellsort*

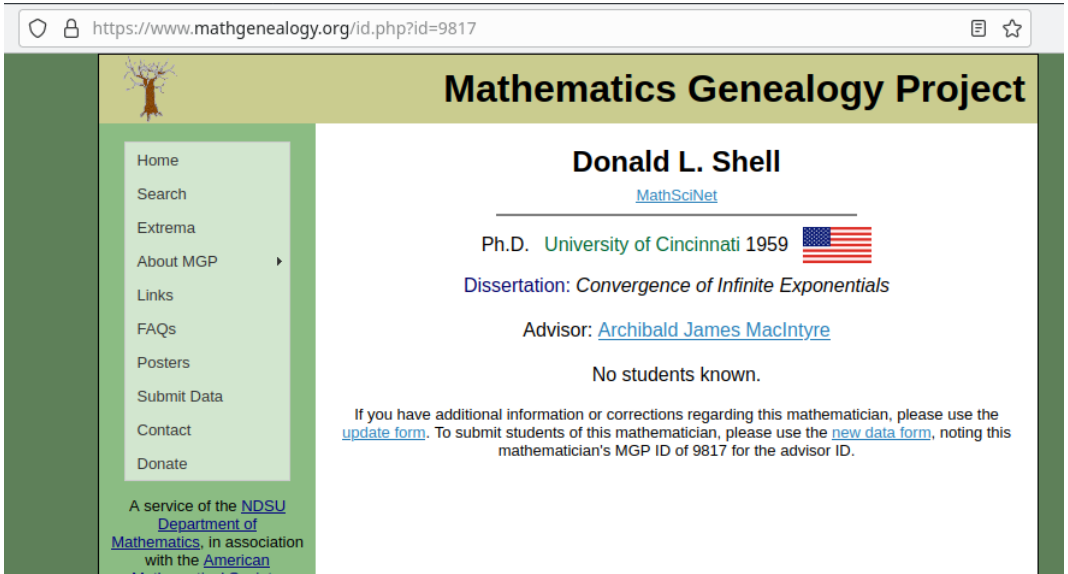


Izmislio ga je Donald Šel, krajem 50tih godina XX veka i po njemu nosi ime

Nagradno pitanje

Šta je Šel bio po obrazovanju?

Matematičar



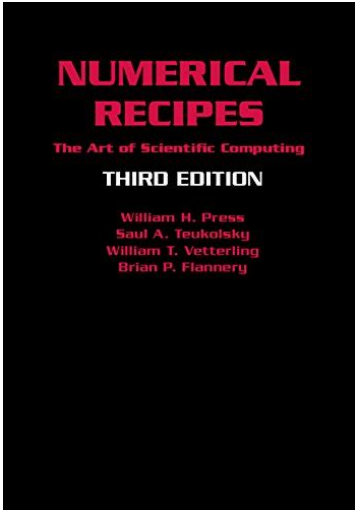
Težina analize složenosti

Međutim, ipak nije uspeo da dokaže složenost svog algoritma za predloženi niz razmaka

Ali ju je eksperimentalno izmerio

This particular method requires a negligible amount of storage space in addition to that occupied by the list of items. In addition, it operates at fairly high speed. Thus far, an analytical determination of the expected speed has eluded the writer. However, experimental use has established its speed characteristics. It appears that the time required to sort n elements is proportional to $n^{1.226}$

Dokazi za neke sekvence su usledili kasnije



The spacings between the numbers sorted on each pass through the data (8,4,2,1 in the above example) are called the *increments*, and a Shell sort is sometimes called a *diminishing increment sort*. There has been a lot of research into how to choose a good set of increments, but the optimum choice is not known. The set $\ldots, 8, 4, 2, 1$ is in fact not a good choice, especially for N a power of 2. A much better choice is the sequence

$$(3^k - 1)/2, \ldots, 40, 13, 4, 1 \tag{8.1.1}$$

which can be generated by the recurrence

$$i_0 = 1, \qquad i_{k+1} = 3i_k + 1, \quad k = 0, 1, \ldots \tag{8.1.2}$$

It can be shown (see [1]) that for this sequence of increments the number of operations required in all is of order $N^{3/2}$ for the worst possible ordering of the original data. For "randomly" ordered data, the operations count goes approximately as $N^{1.25}$, at least for $N < 60000$. For $N > 50$, however, Quicksort is generally faster.

CITED REFERENCES AND FURTHER READING:

Knuth, D.E. 1997, *Sorting and Searching*, 3rd ed., vol. 3 of *The Art of Computer Programming* (Reading, MA: Addison-Wesley), §5.2.1.[1]

Sedgewick, R. 1998, *Algorithms in C*, 3rd ed. (Reading, MA: Addison-Wesley), Chapter 8.

Šta je Donald Knut bio po obrazovanju?

https://www.nytimes.com/2018/12/17/science/donald-knuth-computers-algorithms-programming.html

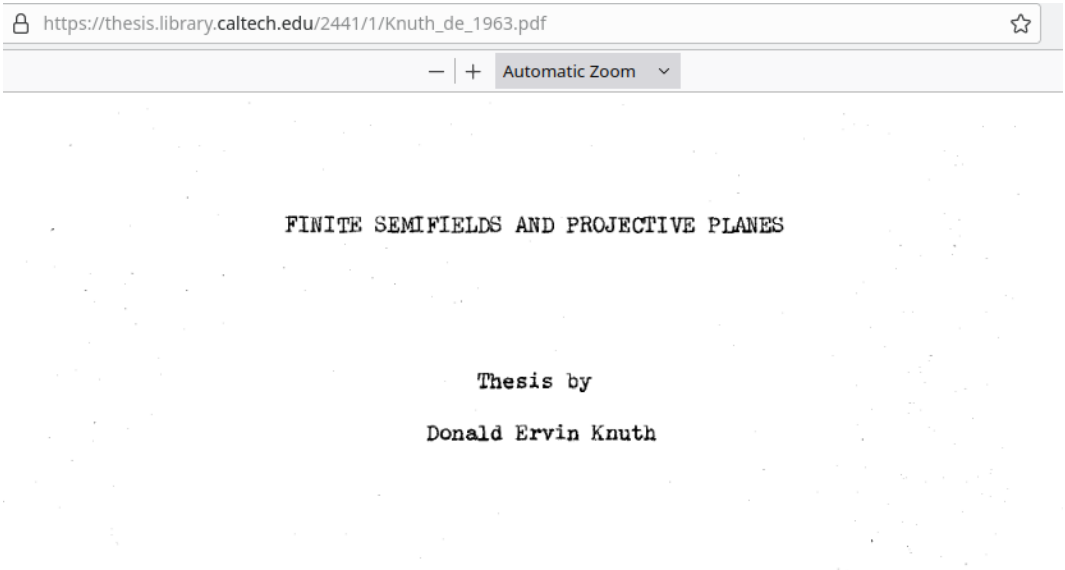
The New York Times

PROFILES IN SCIENCE

The Yoda of Silicon Valley

Donald Knuth, master of algorithms, reflects on 50 years of his opus-in-progress, "The Art of Computer Programming."

Nije bio dželaj, ali jeste na raspolaganju imao silu matematike



Kako izgleda implementacija Shellsort-a u C++-u?

Unutrašnja petlja je Insertion Sort sa razmacima

```
for(unsigned i = gap; i < LEN; ++i)
{
        unsigned val = a[i];
        unsigned j = i;
        while(val < a[j - gap]) //val jos nije stigao do svoje pozicije; moramo jos da idemo ulevo
        {
                ++moves;
                a[j] = a[j - gap]; //Pomeramo elemente udesno, da napravimo mesta za val
                j -= gap;
                if(j < gap)
                        break;
        }
        a[j] = val; //Nasli smo mesto za val, pa ga ubacujemo
}
```

U spoljašnjoj petlji menjamo razmak

a *diminishing increment sort*. There has been a lot of research into how to choose a good set of increments, but the optimum choice is not known. The set $\ldots, 8, 4, 2, 1$ is in fact not a good choice, especially for N a power of 2. A much better choice is the sequence

$$(3^k - 1)/2, \ldots, 40, 13, 4, 1 \tag{8.1.1}$$

which can be generated by the recurrence

$$i_0 = 1, \qquad i_{k+1} = 3i_k + 1, \quad k = 0, 1, \ldots \tag{8.1.2}$$

U spoljašnjoj petlji menjamo razmak

```
unsigned gap = 1;
do
{
        gap *= 3;
        ++gap;
} while(gap <= LEN);

unsigned moves = 0;
do
{
        gap /= 3;
        for(unsigned i = gap; i < LEN; ++i)
        {
                unsigned val = a[i];
                unsigned j = i;
                while(val < a[j - gap]) //val jos nije stigao do svoje pozicije; moramo jos da idemo ulevo
                {
                        ++moves;
                        a[j] = a[j - gap]; //Pomeramo elemente udesno, da napravimo mesta za val
                        j -= gap;
                        if(j < gap)
                                break;
                }
                a[j] = val; //Nasli smo mesto za val, pa ga ubacujemo
        }
} while(gap > 1);
```

```
Ukupno pomeranja 55
3, 4, 6, 9, 11, 12, 13, 16, 17, 23, 27, 31, 37, 42, 44, 65, 79, 84, 99, 102, 120, 147, 201, 234
```

Razlika je daleko uočljivija na većim nizovima

```
Ukupno pomeranja 2286334
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, 32, 33, 34, 35, 36
, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69,
 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99, 100, 101, 10
2, 103, 104, 105, 106, 107, 108, 109, 110, 111, 112, 113, 114, 115, 116, 117, 118, 119, 120, 121, 122, 123, 124, 125, 126, 127, 128,
129, 130, 131, 132, 133, 134, 135, 136, 137, 138, 139, 140, 141, 142, 143, 144, 145, 146, 147, 148, 149, 150, 151, 152, 153, 154, 155
```

Insertion Sort za nasumično sortiran niz od 3000 elemenata

Razlika je daleko uočljivija na većim nizovima

```
Ukupno pomeranja 36118
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, 32, 33, 34, 35, 36
, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69,
 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99, 100, 101, 10
2, 103, 104, 105, 106, 107, 108, 109, 110, 111, 112, 113, 114, 115, 116, 117, 118, 119, 120, 121, 122, 123, 124, 125, 126, 127, 128,
129, 130, 131, 132, 133, 134, 135, 136, 137, 138, 139, 140, 141, 142, 143, 144, 145, 146, 147, 148, 149, 150, 151, 152, 153, 154, 155
, 156, 157, 158, 159, 160, 161, 162, 163, 164, 165, 166, 167, 168, 169, 170, 171, 172, 173, 174, 175, 176, 177, 178, 179, 180, 181, 1
82, 183, 184, 185, 186, 187, 188, 189, 190, 191, 192, 193, 194, 195, 196, 197, 198, 199, 200, 201, 202, 203, 204, 205, 206, 207, 208,
```

Shellsort za nasumično sortiran niz od 3000 elemenata

Razlika je daleko uočljivija na većim nizovima

$3000^2/3000^{1,5} = 54{,}77$

$2286334/36118 = 63{,}30$

Za milion elemenata (na primer, korisnici banke), očekujemo da Shellsort sortira $1000\times$ brže od Insertion Sort-a

Zaključak

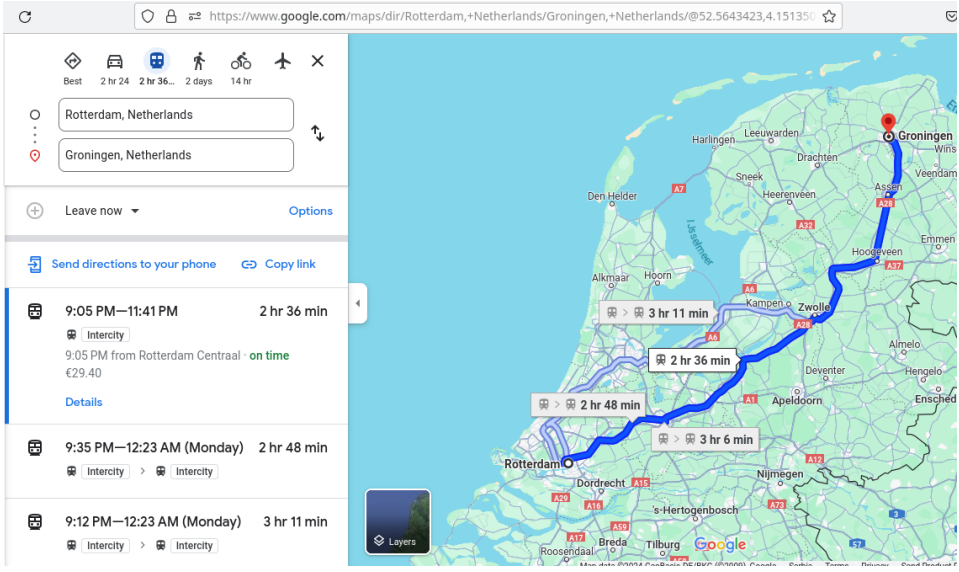
Dobro razumevanje algoritama nam pomaže da pišemo mnogo efikasnije programe

Analiza složenosti često nije trivijalna, ali i eksperimentalna procena pomaže u razumevanju

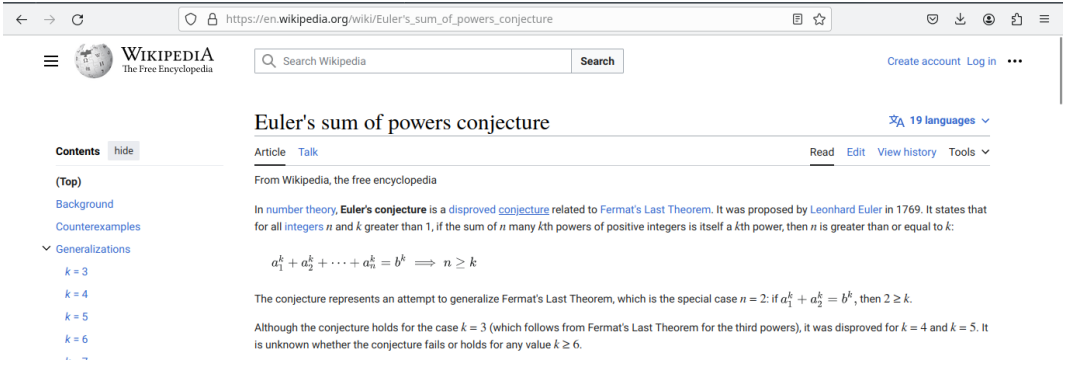
Ako počnemo od jednostavnog rešenja koje dobro razumemo, iterativnim otklanjanjem mana možemo doći do daleko boljeg rešenja

Još ćemo se vratiti na sortiranja nizova i neke druge operacije nad njima, kada za to dođe vreme

Neke primene nizova smo već videli



Neke primene nizova smo već videli



https://en.wikipedia.org/wiki/Euler's_sum_of_powers_conjecture

WIKIPEDIA
The Free Encyclopedia

Search Wikipedia | Search

Create account | Log in

Euler's sum of powers conjecture

19 languages

Article | Talk | Read | Edit | View history | Tools

Contents hide

(Top)
Background
Counterexamples
Generalizations
$k = 3$
$k = 4$
$k = 5$
$k = 6$

From Wikipedia, the free encyclopedia

In number theory, **Euler's conjecture** is a disproved conjecture related to Fermat's Last Theorem. It was proposed by Leonhard Euler in 1769. It states that for all integers n and k greater than 1, if the sum of n many kth powers of positive integers is itself a kth power, then n is greater than or equal to k:

$$a_1^k + a_2^k + \cdots + a_n^k = b^k \implies n \geq k$$

The conjecture represents an attempt to generalize Fermat's Last Theorem, which is the special case $n = 2$: if $a_1^k + a_2^k = b^k$, then $2 \geq k$.

Although the conjecture holds for the case $k = 3$ (which follows from Fermat's Last Theorem for the third powers), it was disproved for $k = 4$ and $k = 5$. It is unknown whether the conjecture fails or holds for any value $k \geq 6$.

Neke primene nizova smo već videli

Još neke ćemo videti na budućim vežbama i predavanjima

Dinamička alokacija memorije

Deklaracija nizova

tip identifikator[dužina niza];

Dužina niza bi trebalo da bude konstanta poznata prilikom kompilacije[5]

[5] Iako neki kompajleri dozvoljavaju specifikaciju dužine pomoću promenljive, zbog portabilnosti koda, to se ne preporučuje.

Šta da radimo ako ne znamo unapred dužinu niza?

Heap

Operativni sistem svakom programu u izvršenju (procesu) dodeljuje adresni opseg za alokaciju promenljivih tokom izvršenja

Taj deo memorije nazivamo **HEAP**

Po potrebi, program može od operativnog sistema da traži uvećanje heap-a, ali taj zahtev ne mora uvek biti uslišen

malloc

Funkcija za dinamičku alokaciju memorije

void* malloc(broj bajta za rezervaciju)

void u C/C++-u označava *ništa* ili *bilo šta*. void* je pokazivač na bilo šta (nema konkretan tip)

Naš primer

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

int main()
{
        unsigned n = 0;
        cout << "Unesite duzinu niza: ";
        cin >> n;

        int* a = (int*)(malloc(n * sizeof(int)));

        for(unsigned i = 0; i < n; ++i)
        {
                cout << "Unesite " << i << "-ti element niza: ";
                cin >> a[i];
        }

        cout << "Uneti niz ispisan u obrnutom poretku:\n";
        for(int i = n - 1; i >= 0; --i)
                cout << a[i] << endl;

        return 0;
}
```

Naš primer

```
Unesite duzinu niza: 7
Unesite 0-ti element niza: 4
Unesite 1-ti element niza: 2
Unesite 2-ti element niza: 5
Unesite 3-ti element niza: 3
Unesite 4-ti element niza: 1
Unesite 5-ti element niza: 6
Unesite 6-ti element niza: 7
Uneti niz ispisan u obrnutom poretku:
7
6
1
3
5
2
4
```

Neophodno je da ono što dinamički rezervišemo kasnije i oslobodimo (čim nam više ne treba)

free(pokazivač koji sadrži adresu bloka rezervisanog malloc-om);

Naš primer

```cpp
#include <iostream>
using namespace std;

int main()
{
        unsigned n = 0;
        cout << "Unesite duzinu niza: ";
        cin >> n;

        int* a = (int*)(malloc(n * sizeof(int)));

        for(unsigned i = 0; i < n; ++i)
        {
                cout << "Unesite " << i << "-ti element niza: ";
                cin >> a[i];
        }

        cout << "Uneti niz ispisan u obrnutom poretku:\n";
        for(int i = n - 1; i >= 0; --i)
                cout << a[i] << endl;

        free(a); //Ono sto rezervisemo moramo i da oslobodimo
                 //Inace ce alokator prestati da radi
                 //(curenje memorije (eng. memory leak)

        a = NULL; //Dobra praksa je i da pokazivac stavimo na NULL
                  //da ne bismo greskom pristupili memoriji kasnije
                  //alociranoj za neku drugu svrhu

        return 0;
}
```

Ručno oslobađanje promenljivih van heap-a nema smisla

```
#include <iostream>
#define LEN 10
using namespace std;

int main()
{
        int a[LEN];

        for(unsigned i = 0; i <= LEN; ++i)
        {
                cout << "Unesite " << i << "-ti element niza: ";
                cin >> a[i];
        }

        cout << "Uneti niz ispisan u obrnutom poretku:\n";
        for(int i = LEN; i >= 0; --i)
                cout << a[i] << endl;

        free(a);

        return 0;
}
```

One će automatski biti oslobođene kada im istekne opseg (više o tome kada budemo radili potprograme)

Ručno oslobađanje promenljivih van heap-a nema smisla

```
array_1.cpp: In function ‘int main()’:
array_1.cpp:19:13: warning: ‘void free(void*)’ called on unallocated object ‘a’ [-Wfree-nonheap-object]
   19 |         free(a);
      |         ~~~~^~~
array_1.cpp:7:13: note: declared here
    7 |         int a[LEN];
      |             ^
```

gcc će izdati upozorenje ali neće prijaviti grešku. Drugi kompajleri će se možda ponašati drugačije.

Dinamička alokacija višedimenzionih nizova je malo složenija

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

int main()
{
        unsigned n = 0;
        cout << "Unesite broj redova matrice: ";
        cin >> n;

        int** a = (int**)(malloc(n * sizeof(int*)));
        //Matrica je niz nizova, pa nam je potreban pokazivac na pokazivac
        //U nizu a cuvamo redove, pa je prva dimenzija n

        unsigned m = 0;
        cout << "Unesite broj kolona matrice: ";
        cin >> m;

        for(unsigned i = 0; i < n; ++i)
                a[i] = (int*)(malloc(m * sizeof(*a[i])));
                //Za svaki red alociramo novi niz duzine m
```

Dinamička alokacija višedimenzionih nizova je malo složenija

```
for(unsigned i = 0; i < n; ++i)
        for(unsigned j = 0; j < m; ++j)
        {
                cout << "Unesite " << i << ", " << j << " vrednost" << endl;
                cin >> a[i][j];
                //Dalje se ponasamo kao da je niz staticki alociran
        }

for(unsigned i = 0; i < n; ++i)
{
        for(unsigned j = 0; j < m; ++j)
                cout << a[i][j] << " ";

        cout << endl;
}
```

Dinamička alokacija višedimenzionih nizova je malo složenija

```
        for(unsigned i = 0; i < n; ++i)
                free(a[i]); //Prvo oslobadjamo svaki red. Ako oslobodimo a,
                            //izgubicemo adrese redova i memorija ce nam cureti
                            //Curenje mozemo proveriti upotrebom programa valgrind
        free(a);
        a = NULL;

        return 0;
}
:!g++ % -Ofast -o run && time valgrind ./run
```

Dinamička alokacija višedimenzionih nizova je malo složenija

```
==4169== Memcheck, a memory error detector
==4169== Copyright (C) 2002-2022, and GNU GPL'd, by Julian Seward et al.
==4169== Using Valgrind-3.20.0 and LibVEX; rerun with -h for copyright info
==4169== Command: ./run
==4169==
Unesite broj redova matrice: 4
Unesite broj kolona matrice: 3
Unesite 0, 0 vrednost
1
Unesite 0, 1 vrednost
2
Unesite 0, 2 vrednost
3
Unesite 1, 0 vrednost
4
Unesite 1, 1 vrednost
5
Unesite 1, 2 vrednost
6
Unesite 2, 0 vrednost
7
Unesite 2, 1 vrednost
8
Unesite 2, 2 vrednost
9
Unesite 3, 0 vrednost
10
Unesite 3, 1 vrednost
11
Unesite 3, 2 vrednost
12
```

Dinamička alokacija višedimenzionih nizova je malo složenija

```
1 2 3
4 5 6
7 8 9
10 11 12
==4169==
==4169== HEAP SUMMARY:
==4169==     in use at exit: 0 bytes in 0 blocks
==4169==   total heap usage: 8 allocs, 8 frees, 74,832 bytes allocated
==4169==
==4169== All heap blocks were freed -- no leaks are possible
==4169==
==4169== For lists of detected and suppressed errors, rerun with: -s
==4169== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Moramo voditi računa o curenju memorije

```
        //for(unsigned i = 0; i < n; ++i)
        //      free(a[i]); //Prvo oslobadjamo svaki red. Ako oslobodimo a,
                            //izgubicemo adrese redova i memorija ce nam cureti
                            //Curenje mozemo proveriti upotrebom programa valgrind
        free(a);
        a = NULL;

        return 0;
}
```

Moramo voditi računa o curenju memorije

```
1 2 3
4 5 6
7 8 9
10 11 12
==5334==
==5334== HEAP SUMMARY:
==5334==     in use at exit: 48 bytes in 4 blocks
==5334==   total heap usage: 8 allocs, 4 frees, 74,832 bytes allocated
==5334==
==5334== LEAK SUMMARY:
==5334==    definitely lost: 48 bytes in 4 blocks
==5334==    indirectly lost: 0 bytes in 0 blocks
==5334==      possibly lost: 0 bytes in 0 blocks
==5334==    still reachable: 0 bytes in 0 blocks
==5334==         suppressed: 0 bytes in 0 blocks
==5334== Rerun with --leak-check=full to see details of leaked memory
==5334==
==5334== For lists of detected and suppressed errors, rerun with: -s
==5334== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Šta zapravo radi alokator?

malloc i free su interfejsi ka alokatoru koji vodi računa o tome koji blokovi heap-a su slobodni

Pri pozivu malloc-a, traži prikladan slobodan blok (spram zahtevane veličine), vraća njegovu početnu adresu i beleži da je rezervisan

Po potrebi može da deli blokove na manje, ili da spaja manje susedne blokove u veće

Kada ne uspe da pronađe odgovarajući blok, šalje operativnom sistemu zahev za povećanje heap-a

Pri pozivu free-a, oslobađa dati blok

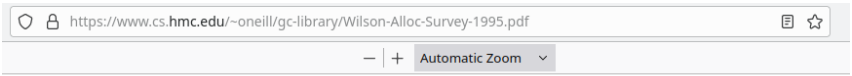
Šta zapravo radi alokator?

U svakom trenutku mora da osigura da nema preklapanja blokova dodeljenih različitim objektima

Potrebno je da što brže da odgovor na zahtev

Poželjno je da što manje rasipa memoriju (na primer, između dva zauzeta bloka je ostao suviše mali blok za bilo koju traženu alokaciju)

Jedan klasičan pregled algoritama za dinamičku alokaciju memorije



Dynamic Storage Allocation: A Survey and Critical Review* **

Paul R. Wilson, Mark S. Johnstone, Michael Neely, and David Boles***

Department of Computer Sciences
University of Texas at Austin
Austin, Texas, 78751, USA
(wilson|markj|neely@cs.utexas.edu)

Njihov dizajn je još uvek otvoren problem

Given the apparent insolubility of this problem, it may seem surprising that dynamic memory allocation is used in most systems, and the computing world does not grind to a halt due to lack of memory. The reason, of course, is that there are allocators that are fairly good in practice, in combination with most actual programs. Some allocation algorithms have been shown in practice to work acceptably well with real programs, and have been widely adopted. If a particular program interacts badly with a particular allocator, a different allocator may be used instead. (The bad cases for one allocator may be very different from the bad cases for other allocators of different design.)

The design of memory allocators is currently something of a black art. Little is known about the interactions between programs and allocators, and which programs are likely to bring out the worst in which allocators. However, one thing is clear—most programs are "well behaved" in some sense. Most programs combined with most common allocators do not squander huge amounts of memory, even if they may waste a quarter of it, or a half, or occasionally even more.

That is, *there are regularities in program behavior that allocators exploit*, a point that is often insufficiently appreciated even by professionals who design and implement allocators. These regularities are exploited by allocators to prevent excessive fragmentation, and make it possible for allocators to work in practice.

These regularities are surprisingly poorly understood, despite 35 years of allocator research, and scores of papers by dozens of researchers.

U slučaju potrebe, možete implementirati sopstveni alokator

GNU Linear Programming Kit

Reference Manual

for GLPK Version 5.0

(December 2020)

6.1.14 glp_alloc — allocate memory block

Synopsis

```
void *glp_alloc(int n, int size);
```

Description

The routine `glp_alloc` dynamically allocates a memory block of `n`×`size` bytes long. Note that:

1) the parameters `n` and `size` must be positive;

2) having been allocated the memory block contains arbitrary data, that is, it is *not* initialized by binary zeros;

3) if the block cannot be allocated due to insufficient memory, the routine prints an error message and abnormally terminates the program.

This routine is a replacement of the standard C function `malloc` and used by GLPK routines for dynamic memory allocation. The application program may use `glp_alloc` for the same purpose.

Returns

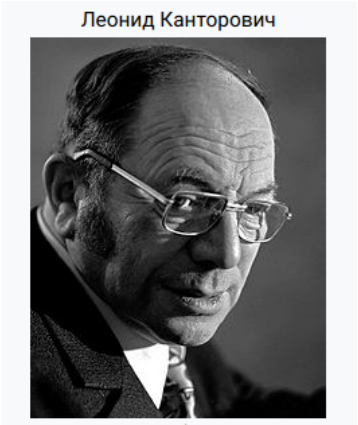
The routine `glp_alloc` returns a pointer to the memory block allocated. To free this block the routine `glp_free` (not the standard C function `free`!) should be used.

Mala digresija

Da li znate kako je nastalo linearno programiranje?

Izmislili su ga nezavisno Leonid Kantorovič u SSSR-u
i Džordž Dancig u SAD-u

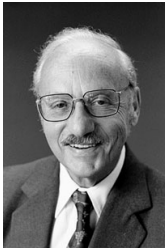
Leonid Kantorovič



Дорога жизни

Smislio je linearno programiranje 1939. za optimizaciju proizvodnje šperploče, a za vreme opsade Lenjingrada ga je koristio za proračun minimalnog rastojanja kamiona na zaleđenom jezeru Ladoga, prilikom dopremanja hrane

Džordž Dancig



George Bernard Dantzig

https://en.wikipedia.org/wiki/George_Dantzig

In 1939, a misunderstanding brought about surprising results. Near the beginning of a class, Professor Neyman wrote two problems on the blackboard. Dantzig arrived late and assumed that they were a homework assignment. According to Dantzig, they "seemed to be a little harder than usual", but a few days later he handed in completed solutions for both problems, still believing that they were an assignment that was overdue.[4][6] Six weeks later, an excited Neyman eagerly told him that the "homework" problems he had solved were two of the most famous unsolved problems in statistics.[2][4]

Nećemo rešavati najveće otvorene probleme za domaći

Ali, molim vas pogledajte šta rade funkcije calloc, realloc, memcpy i memset kao i operatori new i delete[6]

[6] Te operatore ćemo detaljnije raditi kada budemo prešli na objektno orijentisano programiranje

Hvala na pažnji