

Dokumentacija projekta – Facto

Općenito o projektu:

Projekt Facto je program koji korisniku omogućuje izračunavanje faktoriijela proizvoljnog broja na tri različita načina rada. Prvi od njih je onaj klasični način, takozvani “normal” te je u tom načinu rada moguće izračunati maksimalnu faktoriijelu broja 30. Drugi način je “precise” u kojem je u teoriji moguće izračunati faktoriijelu bilo kojeg broja, no zbog “hardverske limitacije”, ovaj način rada je ograničen na faktoriijelu broja 1 000 001. Zadnji način rada ovog programa je izračunavanje faktoriijela koristeći Stirlingovu formulu za aproksimaciju faktoriijela nekog broja te je tako i taj način rada ograničen na faktoriijelu broja 30. Program je napisan u programskom jeziku C. Također, napisan je bigint library koji omogućuje u teoriji izračunavanje bilo kojeg faktoriijela broja.

Matematička podloga:

Općenito:

Faktoriijel prirodnog broja n je umnožak svih prirodnih brojeva koji su manji ili jednaki n .

$$n! = 1 * 2 * 3 * \dots * (n-1) * n$$

Zapisuju se kao $n!$ gdje vrijedi: $n \in \mathbb{N}_0$

Čita se: 'en faktoriijel'

Koriste se u kombinatorici, algebri, teoriji brojeva i drugom.

Neka od svojstava faktoriijela su:

- $0! = 1$
- Svi faktoriijeli veći od 2 su parni brojevi jer su barem u jednom trenutku pomnoženi s 2
- Svi faktoriijeli veći od 5 završavaju s 0 na kraju jer je $5! = 120$ tako da svaki slijedeći će završiti s 0.

Za faktoriijele vrijedi svojstvo rekurzivna relacija koje glasi:

$$n! = n * (n - 1)! \text{ gdje vrijedi: } n \geq 1$$

Primjeri računanja faktoriijela:

$$1! = 1$$

$$2! = 1 * 2 = 2$$

$$5! = 1 * 2 * 3 * 4 * 5 = 120$$

$$10! = 1 * 2 * 3 * 4 * 5 * 6 * 7 * 8 * 9 * 10 = 3\,628\,800$$

$$24! = 1 * 2 * 3 * \dots * 23 * 24 = 620448401733239439360000$$

Algoritmi za množenje

Uvod: Ovo je skup algoritama koji su nam pali na pamet kako bi množili velike brojeve koji ne stanu u klasične tipove podataka. Ovi algoritmi su nam omogućili da možemo izračunati velike faktorijske kao npr. 1000000!.

Algoritam 1:

U prvoj verziji algoritma, za reprezentaciju broja koristili smo strukturu pod nazivom bigint. Ova struktura sadrži nekoliko varijabli koje prate trenutno stanje broja i jedan dinamički alocirani niz varijabli long long tipa, niz po potrebi povećavamo ovisno o veličini broja.

```
struct bigint {  
    int chunks_used;  
    long long *num;  
    size_t num_size;  
};
```

Kako bi pohranili broj u danu strukturu, odnosno u niz long long varijabli, pretvorili smo broj u binarni oblik. Pri pretvorbi u binarni oblik broj dijelimo na chunkove (dijelove) veličine 31 bit, ako koristimo 64 bitne long long varijable.

Veličinu chunka zapravo računamo prema formuli:

$$(\text{sizeof}(\text{varijabla}) / 2) - 1$$

Ovakva veličina nam omogućuje da pomnožimo dva chunka koji sadrže maksimalnu vrijednost, odnosno svi bitovi su im jedinice, te nakon toga pribrojimo još jedan takav umnožak. I to sve bez da premašimo vrijednost koju možemo pohraniti u varijablu.

Za pretvorbu u binarni oblik koristimo *Divide by 2* algoritam. Kada broj primimo od korisnika on se nalazi u obliku ASCII znamenki. Kopiramo taj niz znakova u novi niz nad kojim ćemo vršiti operacije prilikom pretvorbe.

Zatim u petlji dijelimo taj niz znamenaka sa 2, znamenku po znamenku kao da se radi o velikom broju što i jest. Pri svakom dijeljenju dobivamo ostatak 1 ili 0. Te ostatke jedan po jedan raspoređujemo u chunkove. Dok pravi program koristi za pohranu chunkova koristi long long varijable, zbog jednostavnosti objašnjenja zamislimo da su u pitanju char varijable čija je duljina 8 bitova.

Prema formuli možemo izračunati da će tada veličina chunka biti: $8 / 2 - 1 = 3$

Sada recimo da imamo broj 69, prvo ćemo koristeći *Divide by 2* algoritam, računamo znamenku po znamenku binarnog oblika i pritom grupiramo znamenke po 3, jer je to naša veličina chunka.

```
-----
69 / 2 = 34   ostatak 1
34 / 2 = 17   ostatak 0
17 / 2 = 8    ostatak 1  chunk 0
-----
8 / 2 = 4     ostatak 0
4 / 2 = 2     ostatak 0
2 / 2 = 1     ostatak 0  chunk 1
-----
1 / 2 = 0     ostatak 1
                        0
                        0  chunk 2
-----
```

Dakle za pohranu ovog broja potrebna su nam 3 chunka po 3 bita, s time da kako posljednji chunk nismo cijeli ispunili dodajemo vodeće nule. Ovakav zapis nam omogućava da vršimo računske operacije.

Pa tako recimo da želimo izračunati koliko je $69 * 21 = 1449$. Kako bi to učinili koristimo jednostavan školski algoritam za množenje, samo što umjesto znamenki imamo chunkove. Kako bi bilo što jednostavnije za programirati počinjemo od prvog chunka oba broja.

Broj 21 pretvoren u chunkove po 3 je:

A_chunk_0: 101 = 5
A_chunk_1: 010 = 2

A broj 69 kao što smo već ustanovili je:

B_chunk_0: 101 = 5
B_chunk_1: 000 = 0
B_chunk_2: 001 = 1

Sada će nam trebati još jedna bigint struktura odnosno niz chunkova. Za početak množimo sve chunkove broja B nulnim chunkom broja A i rezultate spremamo u chunkove broja RES.

RES_chunk_0: B_chunk_0 * A_chunk_0 => $5 * 5$ => 25 => 11001
RES_chunk_1: B_chunk_1 * A_chunk_0 => $0 * 5$ => 0 => 000
RES_chunk_2: B_chunk_2 * A_chunk_0 => $1 * 5$ => 5 => 101

Kao što možete vidjeti prvi broj sada premašuje veličinu chunka tako da moramo odrezati višak i prenijeti ga na prvi sljedeći chunk.

```
RES_chunk_0: 11 001 => 001      => 001
RES_chunk_1:   000 => 000 + 11 => 011
RES_chunk_2:   101 => 101      => 101
```

Sada smo pomnožili broj B sa nižim chunkom broja A, te s obzirom da broj A ima samo 2 chunka moramo se pozabaviti još samo jednim. Situacija se sada blago komplicira, rezultate više nećemo samo spremati u chunkove RES broja, već ćemo ih zbrajati sa već postojećim vrijednostima, i pri tome moramo napraviti pomak za jedan chunk u dalje, kao što činimo i kod pisanog množenja.

```
RES_chunk_0: 001                                     => 001
RES_chunk_1: 011 + (B_chunk_0 * A_chunk_1) => 3 + (5 * 2)  => 13 => 1101
RES_chunk_2: 101 + (B_chunk_1 * A_chunk_1) => 5 + (0 * 2)  => 5  => 101
              (B_chunk_2 * A_chunk_1) =>      (1 * 2)  => 2   => 010
```

Vidimo da ponovo moramo popraviti chunkove, jer jedan od njih opet ima veću vrijednosti nego što bi trebao, stoga prenosimo vrijednosti na sljedeći. Također vidimo da nam je potreban još jedan chunk u broju RES kako bi rezultat stao.

```
RES_chunk_0: 001 => 001      => 001
RES_chunk_1: 1 101 => 101     => 101
RES_chunk_2: 101 => 101 + 1 => 110
RES_chunk_3: 010 => 010      => 010
```

I to je to, nakon što smo popravili sve chunkove dobili smo svoj rezultat, ostali je još samo pretvoriti ga u decimalni oblik. To činimo koristeći algoritam sličan *Divide by 2* samo što ovoga puta množimo i dodajemo ostatak. Krećemo od znamenke sa najvećom težinom i množimo je sa 2 za svaku sljedeću binarnu znamenku, te joj dodajemo trenutnu binarnu znamenku (ovo radimo u ASCII char nizu).

REZULTAT: 010 110 101 001 => 010110101001 => 10110101001

$0 * 2 + 1 = 1$
 $1 * 2 + 0 = 2$
 $2 * 2 + 1 = 5$
 $5 * 2 + 1 = 11$
 $11 * 2 + 0 = 22$
 $22 * 2 + 1 = 45$
 $45 * 2 + 0 = 90$
 $90 * 2 + 1 = 181$
 $181 * 2 + 0 = 362$
 $362 * 2 + 0 = 724$
 $724 * 2 + 1 = 1449$

Naš konačni rezultat je 1449, što je točan rezultat, možete provjeriti na kalkulatoru :)

Naravno prava implementacija ima neke manje razlike u odnosu na ovdje objašnjenu verziju, s obzirom da množi char niz znamenki, ali princip funkcioniranja je isti.

Algoritam 2:

Objašnjenje algoritma za množenje brojeva A i B

Brojevi:

Broj	Način spremanja	Maksimalna veličina
A	svaka znamenka se sprema u c++ unsigned long int vector	Proizvoljna (Ovisi o memoriji računala)
B	cijeli broj se sprema u unsigned long int tip podatka	$2^{64}-1$

Spremanje broja A:

Broj A se pretvara u vector na način da se svaka znamenka spremi kao element vectora počevši od znamenke najmanje težine

Primjer: Broj 20286 bi bio zapisan kao

indeks	0	1	2	3	4
vrijednost	6	8	2	0	2

Izračunavanje produkta:

Pri računanju produkta brojeva A i B vrijednost produkta se sprema u broj A. Tako da se izvorna vrijednost broja A gubi.

Postupak:

Broj B množimo sa svakom znamenkom broja A i vrijednost tog mjesta spremamo u varijablu gdje je malo prije bila spremljena znamenka.

Primjer: $5746 * 30$

Broj A				
indeks	0	1	2	3
vrijednost	6	4	7	5

Broj B	
vrijednost	30

Broj B množimo sa svim znamenka broja A.

Sada imamo:

Broj A				
indeks	0	1	2	3
vrijednost	180	120	210	150

Nakon množenja potrebno je novo dobivene vrijednosti ponovno pretvoriti u znamenke. To radimo na način da vrijednost podijelimo sa 10. Ostatak pri dijeljenju spremamo umjesto vrijednosti, a cjelobrojni dio dijeljenja spremamo u drugu varijablu npr. carry.

1.znameka

Broj A				
indeks	0	1	2	3
vrijednost	0	120	210	150

Broj B	
vrijednost	18

Zatim prije nego ponovimo isti postupak za iduću znamenku moramo joj prvo pribrojiti carry. Tako radimo sve dok ne dožemo do zadnje znamenke.

2.znameka

1.korak

Broj A				
indeks	0	1	2	3
vrijednost	0	120	210	150

Carry	
vrijednost	18

2.korak

Broj A				
indeks	0	1	2	3
vrijednost	0	138	210	150

Carry	
vrijednost	0

Broj A				
indeks	0	1	2	3
vrijednost	0	8	210	150

Carry	
vrijednost	13

3.znameka

Broj A				
indeks	0	1	2	3
vrijednost	0	8	3	150

Carry	
vrijednost	22

4.znameka

Broj A				
indeks	0	1	2	3
vrijednost	0	8	3	2

Carry	
vrijednost	17

U slučaju da nam na kraju carry nije jednak 0 moramo dodati još elemenata vectoru i na novo kreirane indexe zapisati carry na isti način kako smo to učinili primarno za broj A.

Prije:

Carry	
vrijednost	17

Broj A				
indeks	0	1	2	3
vrijednost	0	8	3	2

Poslije:

Broj A						
indeks	0	1	2	3	4	5
vrijednost	0	8	3	2	7	1

Na kraju imamo rezultat spremljen u vectoru i ispisujemo ga od zadnjeg indeksa prema prvom.

Alogritam 3:

Prva verzija ovog algoritma mogla je dosta brzo množiti i zbrajati (zbrajanje nije objašnjeno, ali je jednostavnije), ali je imala jedan bitan problem. Pretvorba u i iz binarnog oblika potrebnog za računanje trajalo je jako dugo za velike brojeve kakvi nastanu računanjem faktoriijela.

Kako bi riješili taj problem odlučili smo brojeve umjesto po binarnim znamenkama u chunkove sjeći po dekadskim znamenkama pomoći operacija dijeljenja i mod-a.

Veličinu chunka određujemo na sličan način, i dalje koristimo formulu:

$$(\text{sizeof}(\text{varijabla}) / 2) - 1$$

samo što sada umjesto da kažemo chunk ima 31 bit, moramo izračunati koliko najviše dekadskih znamenki može imati broj, a da i dalje stane u 31 bit.

U slučaju chunka od 31 bita, u chunk stane dekadski broj od maksimalno 10 znamenki.

Sada vratimo se na naš stari primjer, ali ovaj put koristit ćemo varijable veličine dva bajta, odnosno 16 bita. Prema formuli možemo izračunati da je veličina chunka 7 bita, u koje stane najveći binarni broj 127, što znači da ćemo koristiti dvije dekadске znamenke.

Uzmimo za primjer broj 4002, kako bi ga podijelili na chunkove jednostavno dijelimo:

$$\text{chunk 0: } 4002 / 1 \% 100 \Rightarrow 02$$

$$\text{chunk 1: } 4002 / 100 \% 100 \Rightarrow 40$$

Kao što možete vidjeti proces je mnogo jednostavniji i nama za shvatiti, ali i procesoru za izvršiti, što znači da će postupak pretvorbe biti puno jednostavniji i brži.

Čisto radi primjera pomnožimo brojeve $4002 * 2004 = 8020008$. Kada oba broja podijelimo u chunkove kao prije dobivamo:

$$\text{A_chunk_0: } 2023 / 1 \% 100 \Rightarrow 04$$

$$\text{A_chunk_1: } 2023 / 100 \% 100 \Rightarrow 20$$

$$\text{B_chunk_0: } 4002 / 1 \% 100 \Rightarrow 02$$

$$\text{B_chunk_1: } 4002 / 100 \% 100 \Rightarrow 40$$

Kao i prije opet množimo sve chunkove broja B nultim chunkom broja A i rezultat spremamo u broj RES:

$$\text{RES_chunk_0: } \text{B_chunk_0} * \text{A_chunk_0} \Rightarrow 02 * 04 \Rightarrow 08$$

RES_chunk_1: $B_chunk_0 * A_chunk_1 \Rightarrow 40 * 04 \Rightarrow 160$

U ovom slučaju već sada moramo prenijeti dio chunka jer smo već prešli maksimalnu vrijednosti koju nam je dopušteno pohraniti u chunk.

RES_chunk_0: $08 \Rightarrow 08 \Rightarrow 08$

RES_chunk_1: $1\ 60 \Rightarrow 60 \Rightarrow 60$

RES_chunk_2: $00 \Rightarrow 00 + 1 \Rightarrow 01$

Sada kao i u verziji 1 množimo sve chunkove broja B drugim chunkom broja A, i pritom radimo posmak.

RES_chunk_0: $08 \Rightarrow 08$

RES_chunk_1: $60 + (B_chunk_0 * A_chunk_1) \Rightarrow 60 + (02 * 20) \Rightarrow 60 + 40 \Rightarrow 100$

RES_chunk_2: $01 + (B_chunk_1 * A_chunk_1) \Rightarrow 01 + (40 * 20) \Rightarrow 01 + 800 \Rightarrow 801$

I za kraj moramo ponovo prenijeti višak na sljedeći chunk:

RES_chunk_0: $08 \Rightarrow 08 \Rightarrow 08$

RES_chunk_1: $1\ 00 \Rightarrow 00 \Rightarrow 00$

RES_chunk_2: $8\ 01 \Rightarrow 01 + 1 \Rightarrow 02$

RES_chunk_3: $00 \Rightarrow 00 + 8 \Rightarrow 08$

I kada poredamo chunkove po težini dobivamo rezultat:

REZULTAT: $08\ 02\ 00\ 08 \Rightarrow 08020008 \Rightarrow 8020008$

Kako bi ispisali rješenje u dekadskom obliku više nije potrebna pretvorba iz binarnog u dekadski oblik na način kao prije, već možemo ispisivati znamenke jednog po jednog chunka, redom od chunka najveće težine, koristeći / i % operatore.

Ovakav postupak značajno ubrzava cijeli proces, i omogućuje nam da uz žrtvovanje malo više prostora u odnosu na prvu verziju znatno ubrzamo pretvorbu.

Naš konačni bigint library, koji koristimo za računanje faktoriijela velikih brojeva bez gubljenja preciznosti, koristi verziju 2 ovog algoritma.

Usporedba algoritama

Uvod:

Kao što je već prije objašnjeno napravili smo 3 algoritma. U ovom dijelu dokumentacije ćemo ih usporediti.

Način usporedbe: Za uspoređivanje različitih programa koristili smo evaulator koji se koristi na natjecanjima iz algoritama, a može se pronaći na stranici informatika.azoo.hr. Evaulator je modificiran na način da više ne traži korisnika za da pritisne enter između svakog isprobavanja nego automatski prelazi na sljedeći testni ulaz. Algoritme smo testirali na 10 primjera, a to su faktorijski brojeva:

Izmjereni faktorijski brojevi									
5	10	100	1000	10000	20000	50000	100000	200000	1000000

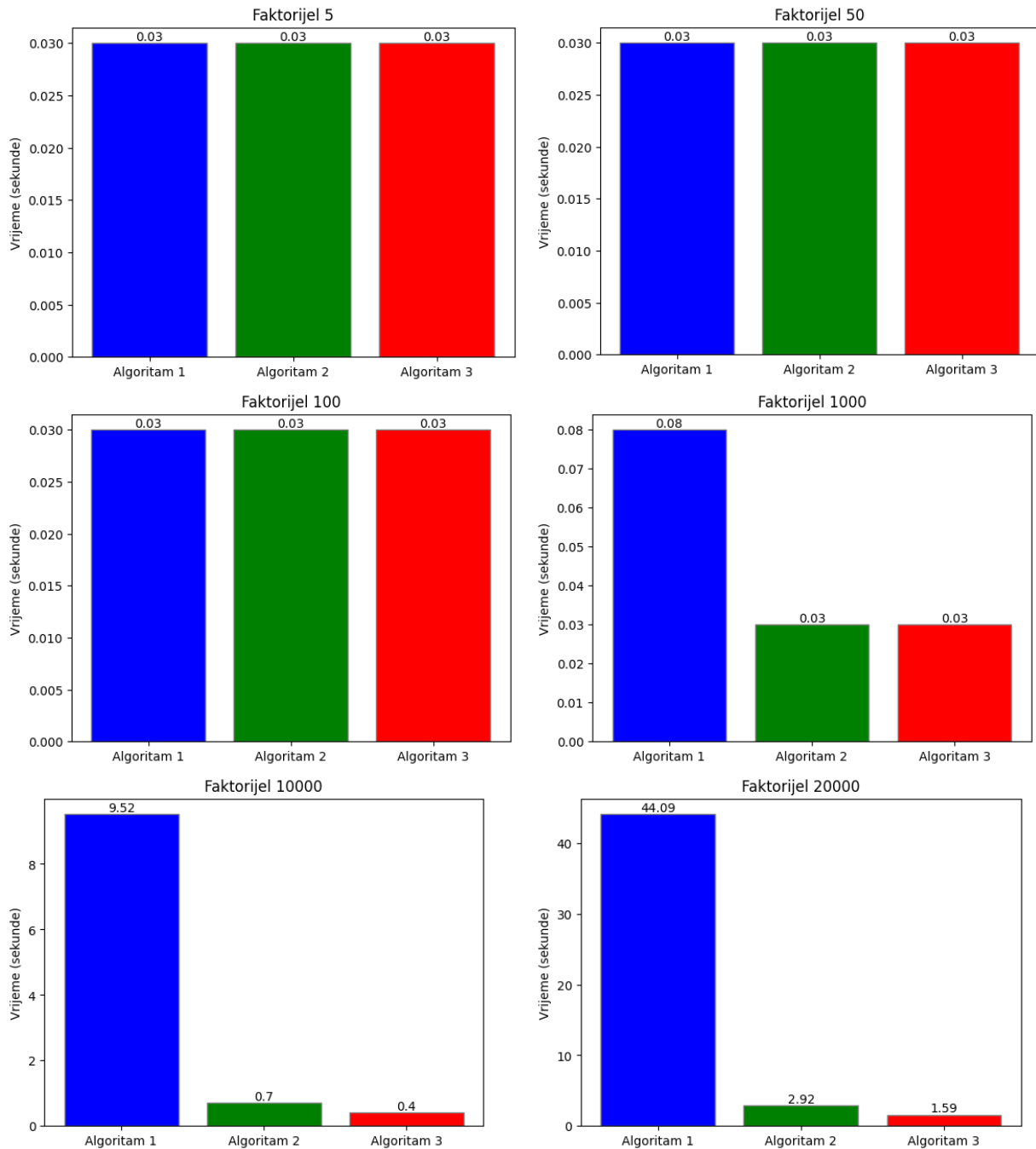
Rezultati mjerenja:

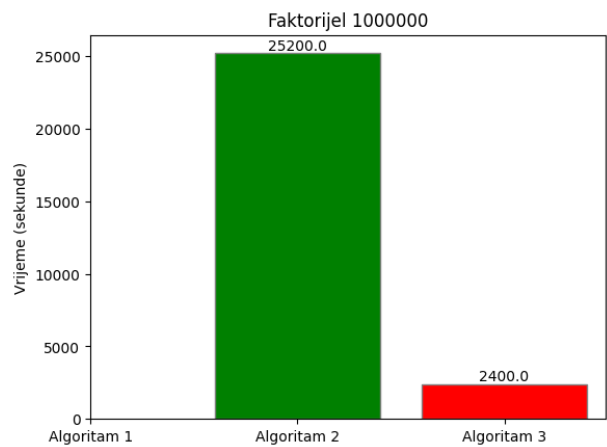
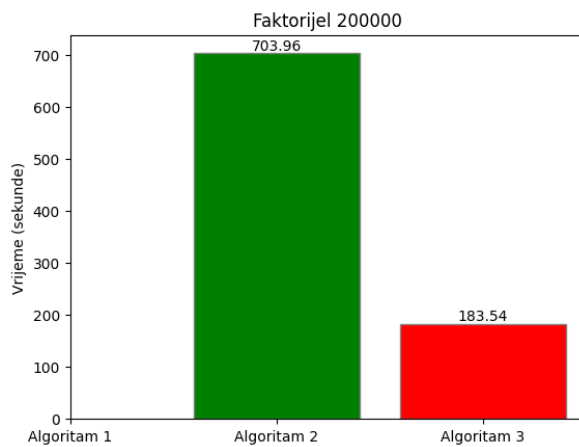
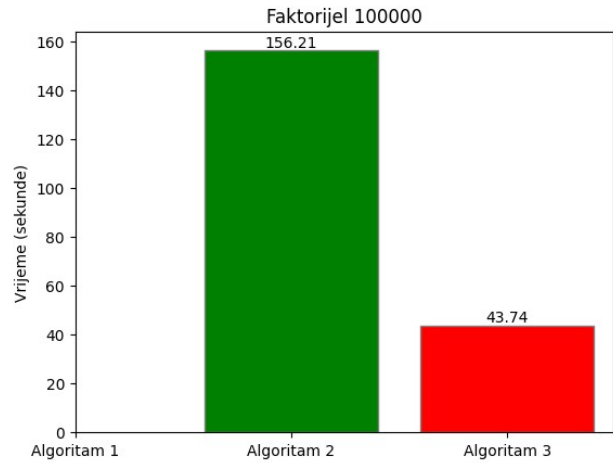
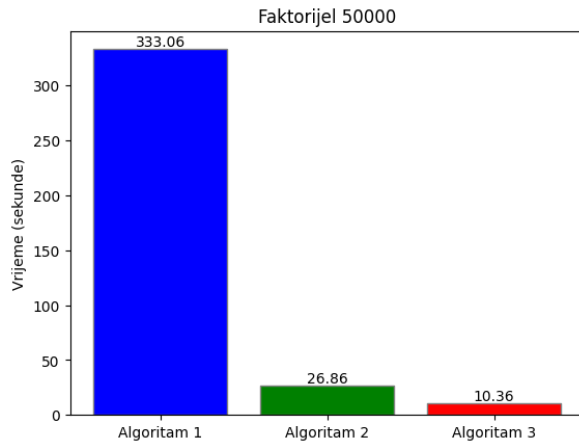
Vrijeme izvršavanja svakog algoritma(sekunde)									
Faktorijski broj	5	10	100	1000	10000	20000	50000	100000	1000000
Algoritam 1	0.03	0.03	0.03	0.08	9.52	44.09	333.06	?	?
Algoritam 2	0.03	0.03	0.03	0.03	0.7	2.92	26.86	156.21	2400(40 min)
Algoritam 3	0.03	0.03	0.03	0.03	0.4	1.59	10.36	43.74	Oko 7 sati

*Oznaka ? znači da bi se program predugo izvršavao da bi ga mogli izmjeriti

Vizualizacija:

Nakon dobivenih podataka odlučili smo vizualizirati vrijeme izvršavanja za svaki algoritma uz pomoć pythona i library-a





Zaključak:

Ovim testiranjem smo primijetili da se razlika između razlika između vremena algoritama sve više mijenja kako dolazimo do većih faktorijela. Čak ne možemo primijetiti razliku između izvršavanja za faktorije do 100. Analizom svih podataka zaključujemo da je algoritam 3 najbolja opcija te da ćemo ga koristiti u finalnoj verziji programa.

Testiranje programa:

Tesiranje programa odradili smo jednostavnom usporedbom dobivenih rezultata s onim vrijednostima koje nam je kalkulator izračunao.

Način rada "normal":

Testni primjer 1:

- Program: $10! = 3628800$
- Kalkulator: $10! = 3628800$

Testni primjer 2:

- Program: $24! = 620448401733239439360000$
- Kalkulator: $24! = 6.204484017 * 10^{23}$

Testni primjer 3:

- Program: $30! = 265252859812191058647452510846976$
- Kalkulator: $30! = 2.652528598 * 10^{32}$

Način rada "precise":

Testni primjer 1:

- Program: $48! =$
 $12413915592536072670862289047373375038521486354677760000000000$
- Kalkulator: $48! = 1.241391559 * 10^{61}$

Testni primjer 2:

- Program: $60! =$
 $83209871127413901442763411832233643807541726063612459524492776964096000$
 000000000000
- Kalkulator: $60! = 8.320987113 * 10^{81}$

Testni primjer 3:

- Program: $90! =$
 $14857159644817614973095227336208257378855699612846887669422168637049853$
 $93094065876545992131370884059645617234469978112000000000000000000000$
- Kalkulator: $90! = \text{Math ERROR}$

Način rada "stirling":

Testni primjer 1:

- Program: $10! = 3598695.62$
- Kalkulator $10! = 3598695.619$

Testni primjer 2:

- Program: $24! = 618297927022794799841280$
- Kalkulator: $24! = 6.18297927 * 10^{23}$

Testni primjer 3:

- Program: $30! = 264517095922965156800687262138368$
- Kalkulator: $30! = 2.645170959 * 10^{32}$

Upute za korisničko sučelje:

Na prvom prikazu se vidi naslov našeg rada. Rad se zove **Facto**

Prva komanda za pomoć korisniku je komanda *"help"*.

```
[Faded]
A simple factorial calculator
Made by BrownBird Team <3

Type help for the list of commands and program description

[NORMAL] >>
```

Nakon upisa komande *"help"* izlistavaju se sve naredbe i kompletna pomoć za korisnika.

```
[NORMAL] >> help

Facto is a simple factorial calculator. It can be used to calculate
factorials in 3 different ways, each with it's own way and precision.
Following commands are supported:

help
    Prints this help message

<number>!
    Calculates factorial of given <number> in current mode

mode
    Displays in which mode program is operating in

normal
    Switches to normal mode, this mode uses double variables to store
    result of factorial and thus can calculate up to 30!

precise
    Switches to precise mode, this mode uses big integers to calculate
    factorial and should be able to calculate factorial of any number,
    but because of hardware limitations this mode is limited to 1000001!

stirling
    Switches to stirling mode, this mode approximates value of factorial
    using stirling's formula, it also uses doubles like the normal mode and
    is limited to 30!

out <path>
    Saves result of factorial calculation into specified file, if called
    without an argument prints current output file. If you wish to reset
    output back to standard output, type STDOUT as filename

version
    Prints current program version

exit
    Finishes the execution of the program

good luck :)

[NORMAL] >>
```


S novo prikazanim komandama možemo početi računati.

Naredba <number>!

- izračunava faktoriјelu od proizvoljnog broja

<pre>[NORMAL] >> 2! 2! = 2.00 Took 0.000622s to execute [NORMAL] >> _</pre>	<pre>[NORMAL] >> 25! 25! = 15511210043330983907819520.00 Took 0.001906s to execute [NORMAL] >></pre>
---	--

Program se sastoji od više modova. Modovi su: normal, precise, striling.

Sada koristimo normal mode koji je prikazan u zagradama [NORMAL].

Problem sa Normal i striling modom je taj što računaju do broja 30.

To je prikazano u primjeru:

```
[NORMAL] >> 31!  
facto error: Maximum factorial [NORMAL] mode can calculate is 30!  
Took 0.003299s to execute  
[NORMAL] >> _
```

Za pogled na trenutajući mod rada koristimo komandu mode

```
[NORMAL] >> mode  
Facto is running in [NORMAL] mode  
[NORMAL] >> _
```

Za promjenu moda možemo napisati komande: normal, precise ili striling

```
[NORMAL] >> precise
Switching mode to [PRECISE]
[PRECISE] >>
```

```
[STIRLING] >> normal
Switching mode to [NORMAL]
[NORMAL] >>
```

```
[PRECISE] >> stirling
Switching mode to [STIRLING]
[STIRLING] >>
```

Ako prebacimo mod rada u precise možemo računati veće brojeve.

```
[PRECISE] >> 100!
100! = 9332621544394415268169923885626670049071596826438162146859296389521759999322991
56089414639761565182862536979208272237582511852109168640000000000000000000000000
Took 0.008786s to execute

[PRECISE] >>
[PRECISE] >> 345!
345! = 2421563865079234655870005369198585557012055604025865273483978326703996172017832
35931747390479136170796955315026894730122138208891348858539928184380564450802014828636
75240494802269823110125881000284687377104376400792200165127855908498047507347955446603
09396432698708731139427468423730839850291130496971971509806802549750490073058021701657
32700116984673789242915507808736051547368795426025546355584282656903020913423594718635
08627516511203478353542187151045838267239168928747525890559708487655213488727530884968
55871638500043698912947952783301034051776068834536871572902001533686253435387691487120
177669920587866285855585726554423099917844925644800000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000000000
Took 0.032915s to execute

[PRECISE] >> _
```

Ako prebacimo mod rada u stirling mode možemo računati brojeve strilingovom formulom.

```
[STIRLING] >> 20!
20! = 2422786846761136640.00
Took 0.001589s to execute

[STIRLING] >>
```

Ako želimo neki broj spremit u file na našem računalu koristit ćemo komandu out <path>

<path> = ime dokumenta pod kojim će biti spremljen broj

```
[NORMAL] >> out Broj


Output file is now set to file "Broj"

[NORMAL] >> 3!

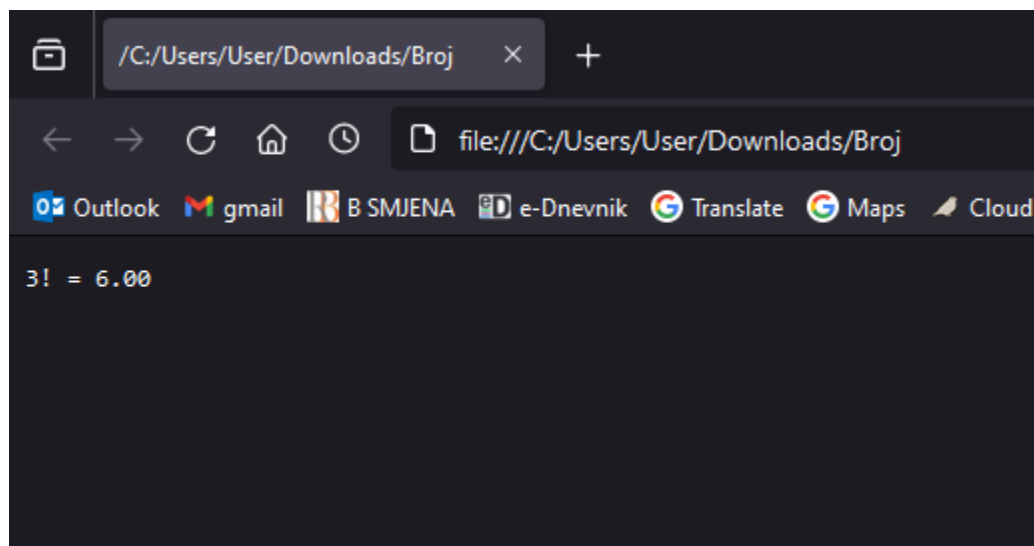
Writing result to file "Broj"
Took 0.000013s to execute

[NORMAL] >>
```

Dokument će se spremiti u Downloads pod odabranim imenom.

Name	Date modified	Type	Size
 Broj	3.12.2023. 21:15	File	1 KB

Ako otvorimo taj dokument u browseru prikazati će nam se ovako:



Za ispis trenutne verzije programa pišemo version

```
[NORMAL] >> version

Facto v1.0.0
Made by BrownBird Team <3

[NORMAL] >>
```

Podjela poslova:

Programiranje	
Algoritam 1	Roko Dobovičnik
Algoritam 2	Borna Flinta
Algoritam 3	Roko Dobovičnik
Završni program	Roko Dobovičnik

Dokumentacija	
Općenito o projektu	Luka Dimjašević
Matematička podloga	Nika Miletić
Objašnjenje algoritama	Roko Dobovičnik i Borna Flinta
Usporedba programa	Borna Flinta
Testiranje programa	Luka Dimjašević
Zaduženja članova tima	Borna Flinta
Spajanje dokumentacije	Borna Flinta

Prezentacija	Marin Cvjetković
---------------------	------------------