1 Uvod u Mathematica-u

Šta je Mathematica?

Mathematica je komputacijski software koji se koristi u naučnim, inžinjerskim, matematičkim i drugim poljima tehničke komputacije.

Stvorio ju je Stephen Wolfram, a razvoj paketa nastavila firma Wolfram Research.

Mathematica se u matematici koristi na mnoge različite načine - kao pomoć prilikom istraživačkog rada, za različite numeričke metode, ali i kao samostalan programski jezik koji ima mnogobrojne veze sa drugim programskim jezicima i programskim paketima, kao što su C, .NET/Link, JAVA, SQL, OpenOffice, Acrobat, itd. Neke od korisnih svojstava Mathematica-e su:

- Biblioteka elementarnih matematičkih funkcija;
- Biblioteka posebnih matematičkih funkcija;
- Alati za manipulaciju matricama i podacima:
- Podrška za kompleksne brojeve, te proizvoljnu preciznost;
- 2D i 3D vizuelizacija podataka i funkcija;
- Izračunavanje sistema jednačina, diofantskih jednačina, ODJ, PDJ, diferencijalnih algebarskih jednačina itd;
- Numerički i simbolički alati za diskretni i neprekidni račun;
- Statističke biblioteke;
- Programski jezik koji podržava proceduralne, funkcionalne i objektno-orijentisane konstrukcije;
- Alati za obradu slike;
- Alati za vizuelizaciju i analizu grafova;
- Alati za kombinatoriku;
- još mnogo, mnogo više;

1.1 Kakav će ovo biti kurs?

Kakav će ovo biti kurs?

Ne možemo tvrditi da ćemo u ovom kursu moći niti blizu pokriti sve moguće aspekte Mathematica-e, ali ćemo na početku pokušati dati jedan opći pregled mogućeg korištenja ovog programskog paketa u programerske i matematičke svrhe.

Na početku ćemo se koncentrirati se na slijedeće:

- Mathematica kao kalkulator;
- Funkcije u Mathematici;

- Manipulacija listama i matricama;
- Grafika u Mathematici;
- Primjena u rješavanju diferencijalnih jednačina;

Kasnije ćemo posmatrati mnoge naprednije forme u Mathematici, kao što su

- Rješavanje diferencijalnih jednačina;
- Rješavanje sistema algebarskih i diferencijalnih jednačina;
- Algebarske metode;
- Primjene u numeričkoj matematici;
- Neke druge teme (zavisno od vremena).

Ko sam ja?

Vedad Pašić

MMath University of Sussex

PhD University of Bath

Trenutno: Prirodno-matematički Fakultet, Univerzitet u Tuzli

email: vedad.pasic@untz.ba

Web: http://www.vedad.com.ba/pmf/

1.2 Rad u Mathematica-i

Rad u Mathematica-i

Mathematica je podijeljena na dva dijela - *kernel* i *frontend*. Kernel interpretira izraze (u Mathematica kodu) i vraća rezultatne izraze.

Frontend nam daje GUI, koji omugućava kreiranje i editovanje Notebook dokumenata (ekstenzija nb) koji sadrže kod sa prettyprintingom, formatovanim tekstom, grafikom, komponentama GUI-a, tabele pa čak i zvuk.

Većina standardnih mogućnosti procesiranja teksta su podržane, iako ima samo jedan nivo "undo"-a (na žalost!). Dokumenti se mogu struktuirati koristeći se hijerarhijom ćelija, a mogu se prezentovati i u prezentacijskom okruženju (koji mi ovdje ne koristimo).

Veoma korisna mogućnost, koju ćemo prezentovati nešto kasnije, je eksportovanje notbooka u različite formate kao što je recimo TeX ili PDF.

Pogledajmo skupa kako to izgleda - pokrenimo Mathematica-u.

1.3 Funkcije u Mathematica-i

Funkcije u Mathematica-i

Do sada smo vidjeli kako se koristiti raznim funkcijama vec ugrađenim u Mathematica-u.

No naravno da će od velikog značaja biti vidjeti kako definisati sopstvene funkcije u ovom programskom paketu!

Nekoliko je ključnih stvari za upamtiti:

- Ne počinjite imena svojih funkcija sa velikim slovom kako biste izbjegli koliziju sa već definisanim funkcijama.
- Dajte imena svojim funkcijama koja imaju smisla.
- Funkcije se definišu sa uglastim zagradama.
- Pazite da ne definišete dvije funkcije sa istim imenom!

Pogledajmo kako to izgleda u praksi.

2 Liste

Jedna od fundamentalnih struktura u bilo kojem programskom jeziku, pogotovo funkcionalnom, su *liste*.

Ukratno, liste su u Mathematici ekvivaletne nizovima u proceduralnim jezicima, kao što je C, ali sa mnogo više funkcionalnosti!

Već smo vidjeli neke liste u Mathematici a da možda toga nismo bili ni svjesni.

U Mathematici su one apsolutno neizbježne! Sto im moramo posvetiti maksimalnu pažnju.

Logički operatori i kondicionali

Logički operatori su naravno fundament bilo kojeg programskog jezika i Mathematica nije izuzetak u tom smislu.

Mathematica se ne razlikuje bitno po ovom pitanju od drugih programskih jezika.

Obratiti pažnju dakako - jednakost "=" se koristi za pripisivanje vrijednosti! Dakle, kako se onda ispituje jednakost dvije promjenljive?

Pomoću "==" dvostruke jednakosti.

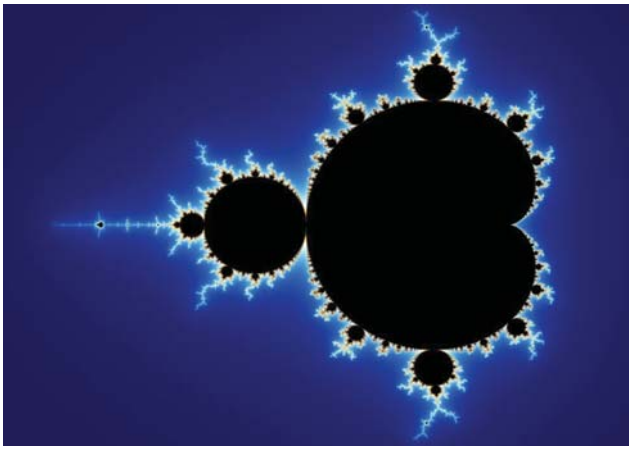
2.1 Logički operatori

Nejednakost i generalno negacija u Mathematici se postiže pomoću "!" uskličnika. To jest, "!=" je ekvivalentno sa $\neq$.

Operatori poređenja su jasni:

$$<=, <, >=, > .$$

Primjetite da Mathematica (u svojim novijim verzijama) automatski pravi operatore u formu na koju smo navikli - budite oprezni sa ovim!



na kraju, trebaju nam operatori $\wedge$ i $\vee$. Oni se dobivaju pomoću komandi && i || respektivno. Korištenjem ovih poredbenih i logičkih operatora, možete da dobijete isključivo dva odgovora - u logici predstavljenih sa T i N. U svakom programskom jeziku, ove se vrijednosti nazivaju booleans (ili Booleove vrijednosti).

Skoro bez izuzetka se označavaju sa TRUE i FALSE.

Pogledajmo kako ovo izgleda u praksi.

2.2 Kondicionali

Skoro nikakvo ozbiljno programiranje ne bi bilo moguće bez kondicionala.

Kondicionali nam omogućuju da zavisno od određenih vrijednosti uradimo neku od više mogućih stvari. Standardni naziv za ove strukture je "If Then Else".

Na žalost, u Mathematica-i je ova struktura dosta nejasne sintakse i treba biti veoma oprezan.

$$If[test, then, else, unknown]$$

je forma kondicionalne funkcije If koja radi "then", ako je "test" True, "else", ako je "test" False i "unknown" ukoliko je test neznana vrijednost.

3 Fraktali

3.1 Uvod u fraktale

Mandelbrotov skup

Definicija fraktala

Rečeno informalno, fraktal je grub ili fragmentiran geometrijski oblik koji se može podijeliti u dijelove od kojih je svaki (baem približno) umanjena kopija originala.

Ova osobina se naziva 'samo-sličnost'.

Riječ dolazi od latinskog *fractus*, što znači slomljen i termin je 1975. godine izmislio Benoît Mandelbrot.

Matematički fraktal je zasnovan na jednačini koja prolazi kroz iteraciju.

Definicija fraktala

Fraktal obično ima slijedeće osobine:

- Ima finu strukturu do proizvoljno malih skaliranja.
- Previše je iregularan da bi bio opisan Euclidskom geometrijom.
- Samo-sličnost.
- Njegova Hausdorffova dimenzija je veća od topološke dimenzije.
- Ima jednostavnu rekurzivnu definiciju.

Definicija fraktala

Prirodni primjeri fraktala uključuju:

- Oblaci;
- Planinski lanci;
- Munje;
- Obalni pojasevi;
- Snježne pahuljice;
- Određeno povrće (karfiol ili brokula)...

3.2 Historija fraktala

Historija fraktala

1872. se pojavljuje funkcija čiji se graf može smatrati fraktalom.

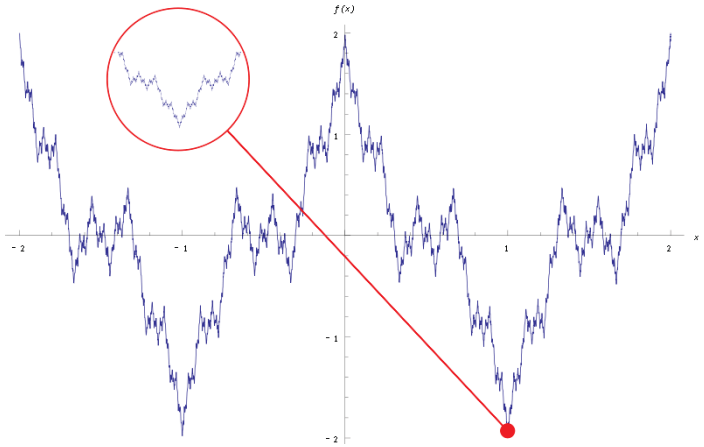
Karl Weierstrass daje primjer funkcije sa neintuitivnom osobinom da je svugdje neprekidna, a nigdje diferencijabilna!

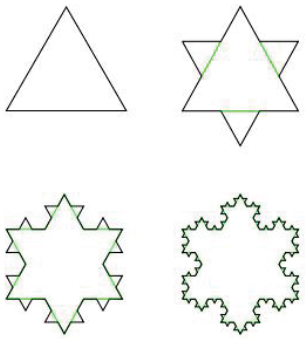
$$f(x) = \sum_{n=0}^{\infty} a^n \cos(b^n \pi x),$$

gdje je $0 < a < 1$, b pozitivan neparan cijeli broj i

$$ab > 1 + \frac{3}{2}\pi.$$

Weierstrassova funkcija





Kochova pahuljica

1904. Helge von Koch, nezadovoljan Weierstrassovom definicijom, daje mnogo više geometrijski primjer fraktala.

Kochova pahuljica

Površina Kochove pahuljice je

$$\frac{2s^2\sqrt{3}}{5}$$

gdje je s dužina jedne stranice originalnog trougla. Dakle, Kochova pahuljica ima beskonačnu granicu, a konačnu površinu! 1918 godine Bertrand Russell je priznao 'vrhunsku ljepotu' unutar nastajuće matematike fraktala.

Fraktali kompleksne ravni

Iterirane funkcije u kompleksnoj ravni su ispitivane u kasnom 19om i ranom 20om stoljeću.

Taj rad je bio djelo Henri Poincaréa, Felixa Kleina, Pierrea Fatoua i Gastona Julia-e.

Međutim bez pomoći kompjuterske grafike, nismo imali mogućnost vizeulizacije ljepote mnogih objekata koji su bili otkriveni.

3.3 Mandelbrotov skup

Mandelbrotov skup

Mandebrotov skup je skup tačaka u kompleksnoj ravni čija granica formira fraktal. Matematički, ovaj skup se definiše kao skup kompleksnih tačaka $c \in \mathbb{C}$, za koje orbita nule pod iteracijama kvadratnog kompleksnog polinoma $z_{n+1} = z_n^2 + c$ ostaje ograničena. Jasnije, kompleksni broj $c \in \mathbb{C}$ se nalazi u Mandebrotovom skupu ako, počevši od $z_0 = 0$, $|z_n|$ pod gornjom iteracijom nikada ne prelazi određeni broj, ma koliko veliko n postalo! Broj 1 nije u Mandelbrotovom skupu. No, broj i jeste!

$$0, i, (-1+i), -i, -1+i, -i, \ldots$$

No kako ovo predstaviti u Mathematica-i?

Pa prije svega trebamo definisati funkciju Mandelbrot, koja vraća broj iteracija kojih moramo proći kako bi $|z_n|$ pod gornjom iteracijom prešao određenu vrijednost.

Pošto smo naravno ograničeni, neka je taj ograničavajući broj 2, a maksimalni broj iteracija 100.

Dakle, recimo

$$Mandelbrot[zc\_] := Module[\{z = 0, i = 0\},$$

$$While[i < 100 \&\& Abs[z] < 2, z = z^2 + zc; i++]; i];$$

Probajmo ovu funkciju na gornjim primjerima. Očito, od interesa su nam u stvari oni brojevi koji su "ni tamo, ni ovamo" dakle koji dosta dugo ne divergiraju, pa onda to učine! Recimo, broj

$$c = -1.2 + 0.193i$$

tek poslije 83 iteracije pređe vrijednost 2, dok već broj $c = -1.2 + 0.2i$ poslije 18 iteracija učini isto. Ovi granični brojevi su oni koji u stvari formiraju granicu Mandelbrotovog skupa! No kako ih prikazati?

Funkcija $DensityPlot$ - veoma slična $ContourPlot$-u.

3.4 Julia skup

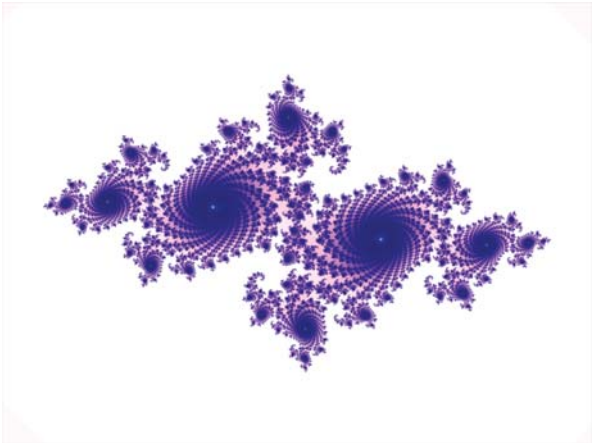
Julia skup

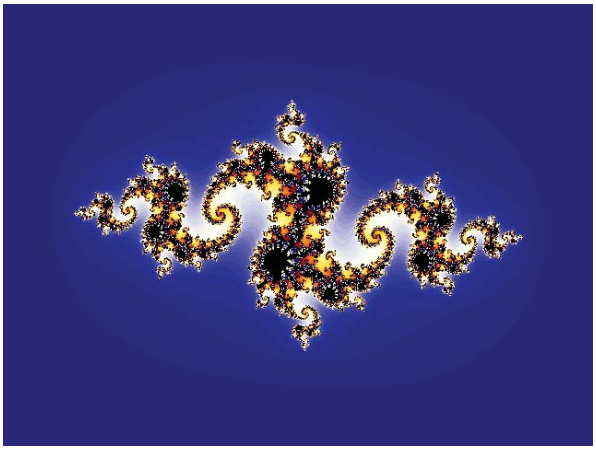
U kompleksnoj dinamici, Julia skup $J(f)$ holomorfične funkcije f se informalno sastoji od onih tačaka čije se dugoročno ponašanje pod ponovljenim iteracijama funkcije f može drastično promjeniti pod proizvoljno malim perturbacijama.

Veoma popularan dinamički sistem je dat sa porodicom kvadratnih polinoma, koji su naravno poseban slučaj racionalnih preslikavanja. Kvadratni polinomi se izražavaju kao

$$f_c(z) = z^2 + c,$$

gdje je c kompleksan parametar. Ovo se fundamentalno dakako razlikuje od prethodnog skupa i proizvodi čitav niz različitih skupova. Primjenimo sličan pristup kao maloprije.





Julia skup

Julia skup

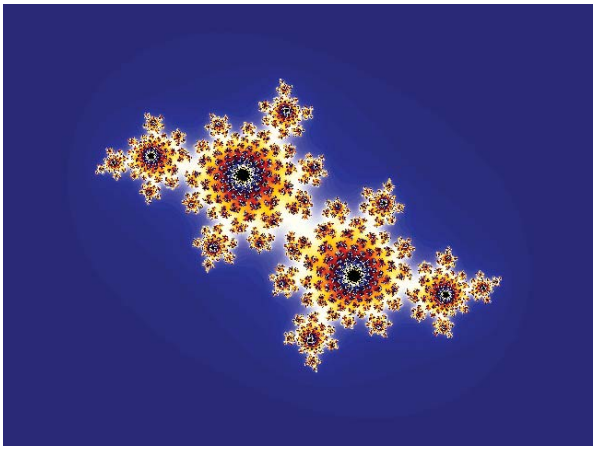
Julia skup

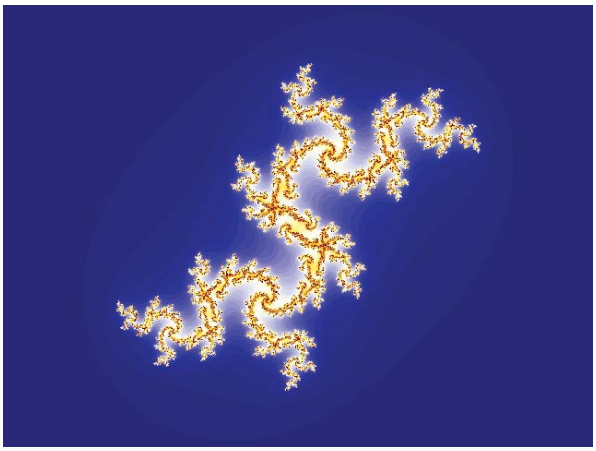
Julia skup

4 MergeSort

4.1 Rekurzija

Rekurzija





- Rekurzija je tehnika u programiranju koja nam omogućava rješavanje brojnih problema na vrlo jednostavan način.
- Veoma je drugačija od standardnog pristupa programiranja na strukturama podataka.
- Nas naravno interesuje primarno primjena rekurzije na liste.
- Rekurzija je česta metoda riješavanja problema rješavanjem podproblema sličnog tipa.

Faktorijel - primjer rekurzije

- Najjednostavniji primjer razlike standarnog pristupa i rekurzivnog pristupa je funkcija koja računa faktorijel datog broja.
- Klasični pristup:
- $faktorijel[n\_] := Module[\{rezultat\}, rezultat = 1; For[i = 1, i <= n, i++, rezultat = rezultat * i]$; rezultat]

Faktorijel - primjer rekurzije

- Rekurzivni pristup
- $faktorijel[n\_] := Module[\{\}, If[n \leq 1, Return[n], Return[n*faktorijel[n-1]]]]$
- Vidimo da u nekim slučajevima rekurzija ima mnogo jednostavniji oblik, gdje se samo brinemo o baznom, najjednostavnijem slučaju, a inace samo pozivamo isti program na "manjem" inputu.
- Jasne paralele sa matematičkom indukcijom!

Još jedan primjer rekurzije

- Još jedan očit primjer je algoritam za najveći zajednički djelilac.
- Iterativna (klasična) metoda:
- $gcd[x\_, y\_] := Module[\{r, x1, y1\}, x1 = x; y1 = y; While[y1! = 0, r = Mod[x1, y1]; x1 = y1; y1 = r]; Return[x1]]$
- Rekurzivna metoda
- $gcd[x\_, y\_] := Module[\{\}, If[y == 0, Return[x], Return[gcd[y, Mod[x, y]]]]]$

4.2 Podijeli pa osvoji

Podijeli pa osvoji

- U kompjuterskoj nauci, zavadi pa osvoji je važan oblik dizajna algoritama.
- Radi tako što rekurzivno razbija problem na dva ili više podproblema istog tipa, sve dok ovi ne postanu tako jednostavni da se mogu riješiti direktno.
- Riješenja ovih podproblema se onda kombinuju kako bi dali rješenja sveukupnog problema.

Podijeli pa osvoji

- Ova je tehnika baza za mnoge djelotvorne algoritme svih vrsta, kao što je sortiranje i diskretne Fourierove transformacije.
- Njena primjena na numeričke algoritme se obično naziva binarno razdvajanje.

4.3 MergeSort

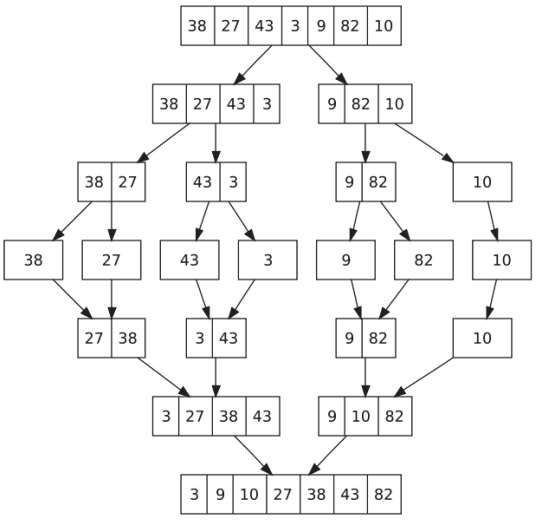
MergeSort

- Do sada smo samo vidjeli iterativne algoritme sortiranja. Sada je red na jedan rekurzivni algoritam, koji se koristi tehnikom podijeli pa vladaj.
- MergeSort uzima kao input jednu listu i vraća listu sortiranih elemenata te liste. Radi tako što od originalne liste napravi *dvije* liste, otprilike podjednake dužine, te ponovo sebe pozove na te *obje liste*.
- Rekurzija se zaustavlja kada dobijemo listu dužine 1.
- Nakon što sve svedemo na skup lista dužine jedan, spajamo ih ponovo, na pametan način, tj. *sortirano*.

Grafički prikaz

Pseudo-Code

- $funkcija\ MergeSort(m)$ $promjenljive\ lijevo, desno, rezultat$ $if\ Length(m) \leq 1$ $return\ m$ $else$ $sredina = length(m)/2$ za svako x u m do pozicije $sredina$ dodaj x u lijevo za svako x u m od pozicije $sredina$ dodaj x u desno $lijevo = mergesort(lijevo)$ $desno = mergesort(desno)$ $rezultat = merge(lijevo, desno)$ $return\ rezultat$



Funkcije merge

- Naravno, tu je ta misteriozna funkcija $Merge$ koja spaja dvije liste u složenu listu.
- Primjetite da je algoritam napravljen na takav način da funckija $Merge$ samo dobija *već sortirane liste* da spoji, što uveliko olakšava posao.
- Sjetite se da je lista od jednog elementa već sortirana.

Grafički prikaz funkcije merge

Funkcija merge - Pseudocode

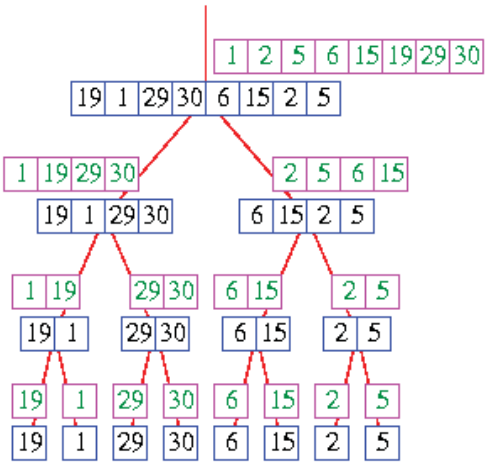
- $funkcija\ merge(lijevo, desno)$ $promjenljive\ i, j, rezultat$ $while(i \leq length(lijevo)$ && $j \leq length(desno))$ $if(lijevo[i] \leq desno[j])\ then$ Dodaj lijevo[i] u rezultat $i++$ $else$ Dodaj desno[j] u rezultat $j++$ Prekopiraj sve preostale elemente iz $lijevo$ ili $desno$ u $rezultat$

5 QuickSort

5.1 Uvod

QuickSort

- QuickSort je još jedan Podijeli pa osvoji algoritam za sortiranje.
- Samo ime kaže da očekujemo da u praksi ovaj algoritam bude najbrži od svih.



- Vrlo dobro znan algoritam koji je razvio C.A.R. Hoare koji, u prosjeku, traje $O(n \log n)$ provjera kako bi sortirao listu dužine n.

QuickSort

- Medjitim, u najgorem slučaju, ovaj će algoritam napraviti $O(n^2)$ usporedbi.
- Pametnom implementacijom ćemo izbjeći ovaj slučaj.
- Tipično je QuickSort značajno brži u praksi od ostalih $O(n \log n)$ algoritama (kao što je MergeSort), zato što se njegova unutrašnja petlja može vrlo efektivno implementirati na većini arhitektura.
- Na primjerima iz stvarnog svijeta se mogu napraviti izbori u dizajnu kako bismo izbjegli slučajeve koji zahtjevaju kvadratno vrijeme.

QuickSort

- QuickSort je sortiranje komparacijom i u djelovornim implementacijama NIJE stabilan algoritam.
- QuickSort je razvio Hoare 1960. godine dok je radio za malu Britansku firmu Elliott Brothers, koja je razijala načne računare.
- Na sličan način kao MergeSort, QuickSort koristi Podijeli pa vladaj strategiju, tako što dijeli listu na dvije podliste, ali na značajno drugačiji način.

5.2 Algoritam

QuickSort Algoritam

Algoritam radi po slijedećim koracima:

- Izaberimo element iz liste, koji zovemo *pivot*.
- Organiziraj članove liste u dvije liste $lijevo$ i $desno$, tako da su elementi manji od pivota u lijevo, a veći u desno (elementi jednaki pivotu mogu ići bilo gdje).
- Rekurzivno sortiraj podliste $lijevo$ i $desno$.
- Bazni slučaj rekurzije su liste dužine nula ili jedan, koje su naravno, već sortirane.

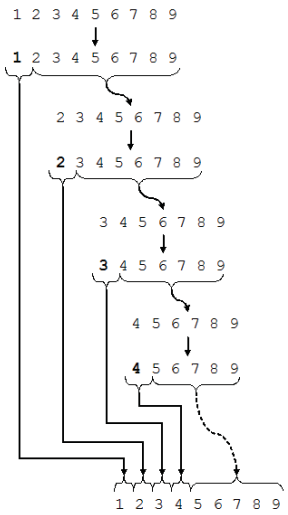
QuickSort PseudoCode

- $funkcija\ QuickSort(m)$
  $promjenljive\ pivot, lijevo, desno, isto, rezultat$
  $if\ Length(m) = 1$
  $return\ m$
  $else$
  odaberi neki element iz liste da bude pivot
  za svako x u listi m
  ako je $x < pivot$ dodaj x u $lijevo$
  ako je $x > pivot$ dodaj x u $desno$
  ako je $x = pivot$ dodaj x u $isto$
  $rezultat = spoji(quicksort(lijevo), isto, quicksort(desno))$
  $return\ rezultat$

QuickSort PseudoCode

- Ovo je bio najprimitivniji oblik quicksorta.
- Problem je što, kao i MergeSort, koristi jako puno memorije - svaki put zauzima 3 nove liste.
- Takodjer je izbor pivota jako bitna stvar - kod prvog pokušaja izaberite prvi element liste.
- No tada se može dogoditi slijedeći slučaj.

Grafički prikaz funkcije quicksort



QuickSort PseudoCode

- Jedno od rješenja je da *randomiziramo* izbor pivota, tj. svaki put izaberemo nasumični pivot.
- No ovo sad čini algoritam nestabilnim.
- Ali ipak očekujemo da će u većini slučajeva algoritam raditi dobro!

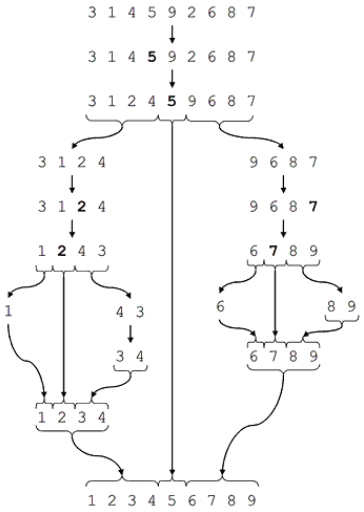
Grafički prikaz funkcije quicksort - randomizirano

QuickSort Random

- Primjera radi, sortirajući već sortiranu listu dužine 1000, prvi algoritam je računao 11.917 sekundi, dok je randomizirana verzija računala 0.41 sekundi.
- Pod ovim podrazumjevamo *nestabilnost*, ali stoga je možemo izbjeći pametnom implementacijom.

QuickSort na mjestu

- Kao što smo vidjeli, slično algoritmu MergeSort, svaki put kada zovemo QuickSort, napravimo čak tri nove liste!
- To je bespotrebno trošenje memorije i usporavanje algoritma, stoga želimo to izbjeći i quicksort primjeniti samo na originalnoj listi.
- Algoritam će raditi na isti način, no sada samo pamteći *poziciju* pivota i izmjeniti mjesta većim i manjim elementima u listi.
- Sada ćemo napisati odvojenu funkciju $particija$, koja će samo rasporedjivati elemente unutar liste.



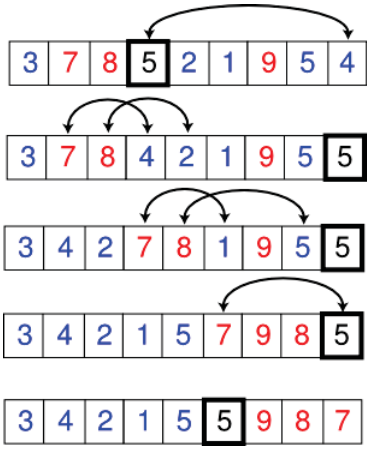
Particija PseudoCode

- $funkcija\ particija(m, lijevo, desno, pivotIndex)$
  $promjenljive\ pivot, noviIndex$
  $pivot := m(pivotIndex)$
  zamjeni $m(pivotIndex)$ i $m(desno)$
  $noviIndex := lijevo$
  za svaki element x liste m izmedju $lijevo$ i $desno$
    ako je $x < pivot$
      zamjeni x i $m(noviIndex)$
      $noviIndex++$
  zamjeni $m(noviIndex)$ i m(desno)
  $return\ \{noviIndex, m\}$

Grafički prikaz funkcije particija

QuickSort PseudoCode sa particijom

- $funkcija\ quicksort(m, lijevo, desno)$
  $if(lijevo > desno)$
    $return\ m$
  $else$
    izaberi poziciju pivota - npr. pivotIndex:=lijevo, ili randomiziraj!

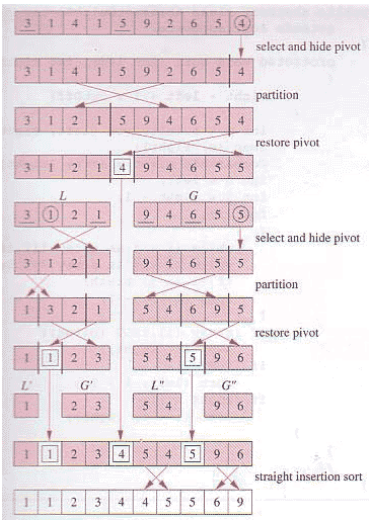


zamjeni $m(pivotIndex)$ i $m(desno)$
$noviIndex := lijevo$
za svaki element x liste m izmedju $lijevo$ i $desno - 1$
  ako je $x < pivot$
    zamjeni x i $m(noviIndex)$
    $noviIndex++$
zamjeni $m(noviIndex)$ i m(desno)
$quicksort(m, lijevo, noviIndex - 1)$
$quicksort(m, noviIndex + 1, desno)$
$return\ m$

Grafički prikaz funkcije quicksort

Problem

- Prilikom implementacije ove vrste algoritma, primjetićete da je, barem u Mathematici, sporiji od standardne implementacije!
- Razlog tomu je previše bespotrebnih zamjena, koje se lako mogu izbjeći.
- Kako to izbjeći? Nekoliko zgodnih "If"-ova može donekle pomoći.
- No optimalno rješenje je korištenje 2 pointera, ili indexa, umjesto jednog!



Problem

- Dakle, nakon izbora pivota i njegovog premještaja na kraj dijela liste koji ispitujemo, stavimo 2 index pointera, npr $desniIndex = desno - 1$ i $lijeviIndex = lijevo$.
- Sve dok je element na $lijeviIndex$ manji (ili jednak) od pivota, povećaj $lijeviIndex$.
- Sve dok je element na $desniIndex$ veći od pivota, smanji $desniIndex$.
- Ako je $lijeviIndex < desniIndex$ nakon ovoga, tek ONDA zamjeni elemente na pozicijama $lijeviIndex$ i $desniIndex$.
- Nastavi proces dok nije $lijeviIndex > desniIndex$.
- Vrati pivota na pravu poziciju, tj. $lijeviIndex$.