Dizajn algoritama i rekurzija

Slajdovi za predmet Osnove programiranja

Katedra za informatiku, Fakultet tehničkih nauka, Novi Sad

2022.

Ciljevi

- osnovne tehnike za analizu efikasnosti algoritama
- pojam pretraživanja, linearna i binarna pretraga
- rekurzivne funkcije
- pojam sortiranja; selection sort i merge sort
- analiza algoritama; reizračunljivi i nerešivi problemi

Pretraživanje

- pretraživanje je proces pronalaženja date vrednosti u kolekciji podataka
- primer: program za evidentiranje studenata
    - pronalaženje podataka o studentu na osnovu broja indeksa

Primer pretrage

- funkcija za pretragu

```
def search(x, nums):
    # nums je lista brojeva, x je broj koji se traži
    # vraća indeks u listi gde se nalazi x,
    # ako x nije u listi vraća -1
```

- primeri upotrebe:

```
>>> search(4, [3, 1, 4, 2, 5])
2
>>> search(7, [3, 1, 4, 2, 5])
-1
```

Pretraga i Python

- testiranje da li se `x` nalazi u sekvenci: operator `in`

```
if x in nums:
    # uradi nešto
```

- pozicija `x` u listi pomoću funkcije `index`

```
>>> nums = [3, 1, 4, 2, 5]
>>> nums.index(4)
2
```

Naša funkcija vs Python funkcija

- naša funkcija vraća -1 ako ne nađe element u listi
- Python-ova funkcija `index` baca izuzetak ako ne nađe element u listi
- mogli bismo napraviti naš `search` tako da koristi `index`,
- hvata izuzetak i vraća -1 u tom slučaju

```
>>> nums = [3, 1, 4, 2, 5]
>>> nums.index(4)
2
```

Naša funkcija vs Python funkcija $_2$

- prethodni primer može da posluži,
- ali nas zanima kako Python implementira `index`

Strategija 1: linearna pretraga

- primer: data nam je neuređena lista brojeva
- zanima nas da li je broj 13 u toj listi
- kako da to ispitamo?

- krenemo od početka liste
- poredimo svaki element liste sa 13
- ako naiđemo na 13, vratimo indeks
- ako dođemo do kraja liste i ne nađemo 13, vratimo -1

Linearna pretraga

- linearna pretraga: prolazimo redom kroz sve elemente liste
- prekidamo pretragu kad nađemo podatak ili kada dođemo do kraja liste

```
def search(x, nums):
    for i in range(len(nums)):
        if nums[i] == x: # pronađen
            return i
    return -1  # došli do kraja, nije pronađen
```

- jednostavan algoritam
- pristojne performanse za male (kratke) liste

Linearna pretraga $_2$

- Pythonov operator `in` i funkcija `index` vrše linearnu pretragu
- ako je lista vrlo velika, pretraga može da traje

- isplati se organizovati podatke tako da ne testiramo sve elemente liste

Binarna pretraga

- ako su podaci sortirani, ne moramo testirati sve elemente liste
- kad naiđemo na element koji je veći od tražene vrednosti
- ne moramo testirati elemente iza njega!

- u proseku, možemo uštedeti polovinu testova

Binarna pretraga $_2$

- igra pogađanja zamišljenog broja:
- igrač A zamisli broj između 1 i 100, a igrač B ga pogađa
- za svaki pokušaj igrač A kaže da li je zamišljeni broj veći ili manji od broja koji je pokušao igrač B

- isplati se prvo probati 50; tri ishoda:

a) pogodili smo broj :)
b) ako je odgovor da je broj veći od 50, odbacujemo opseg 1-50
c) u suprotnom odbacujemo opseg 50-100

- ako je odgovor pod b), sledeći pokušaj nam je broj 75...

Binarna pretraga $_2$

- svaki put kao pogodak biramo sredinu trenutnog opsega
- time polovimo opseg
- ovakav pristup se zove binarna pretraga

- binarna jer delimo opseg na dva dela

Binarna pretraga $_3$

- ovo možemo koristiti za naš problem
- treba da pamtimo indekse za početak i kraj tekućeg opsega
- inicijalne vrednosti indeksa:
  - `low = 0`
  - `high = len(list)-1`

Binarna pretraga $_4$

- glavni deo algoritma je petlja koja poredi srednji element opsega sa `x`
- ako je `x` manji od srednjeg,
  `high` se pomera na kraj prve polovine
- ako je `x` veći od srednjeg,
  `low` se pomera na početak druge polovine
- ako je `x` jednak srednjem – našli smo broj

- petlja se završava ako
  - smo našli `x`
  - nema više elemenata za test (`low > high`)

Binarna pretraga $_5$

```
def search(x, nums):
    low = 0
    high = len(nums) - 1
    while low <= high:           # dok opseg nije širok 0
        mid = (low + high)//2 # indeks srednjeg elementa
        item = nums[mid]
        if x == item:            # ako smo ga našli
            return mid
        elif x < item:           # x je u prvoj polovini
            high = mid - 1
        else:                    # x je u gornjoj polovini
            low = mid + 1
    return -1                    # nismo našli x
```

Poređenje algoritama

- koji algoritam je bolji – linearna ili binarna pretraga?
    - linearna je jednostavnija
    - binarna je efikasnija jer ne testira sve elemente
- (intuitivno) možemo očekivati da linearna pretraga radi bolje za kratke liste, a binarna za dugačke
- kako da budemo sigurni?

Poređenje algoritama $_2$

- jedan način: da probamo oba algoritma za liste različite dužine i merimo vreme
  - linearna je brža za liste kraće od 10 elemenata
  - mala razlika za liste od 10-1000 elemenata
  - binarna je znatno brža za liste od 1000+ elemenata
    (za milion elemenata: binarna u proseku 0.0003s, linearna 2.5s)

Poređenje algoritama $_3$

- ovakvi testovi zavise od
  - vrste računara
  - količine memorije
  - brzine procesora
  - itd.
- umesto štoperice, možemo da analiziramo algoritme
- algoritam koji obavi manji broj koraka je brži

Poređenje algoritama $_4$

- kako brojimo „korake“ u algoritmu?
- posmatramo zavisnost broja koraka od veličine kolekcije podataka ili složenosti proračuna

- za pretragu: složenost zavisi od veličine kolekcije

Poređenje algoritama $_5$

- koliko koraka je potrebno za pretragu u listi dužine n?
- šta se dešava kada n postane veliko?

Poređenje algoritama $_6$

- za linearnu pretragu:
  - za listu od 10 elemenata – max 10 poređenja
  - za listu od 20 elemenata – max 20 poređenja
  - za listu od 30 elemenata – max 30 poređenja

- broj operacija linearno zavisi od dužine liste n
- $\rightarrow$ algoritam koji ima linearnu vremensku zavisnost

Poređenje algoritama $_7$

- za binarnu pretragu:
  - neka lista ima 16 elemenata; u svakom ciklusu polovina se eliminiše – prvi put eliminiše se 8 elemenata
  - nakon drugog ciklusa ostane još 4 elementa
  - nakon trećeg ciklusa ostane još 2 elementa
  - nakon četvrtog ciklusa ostane 1 element

- ako binarna pretraga obavi i ciklusa
- može da pronađe element u listi dužine 2^i

Poređenje algoritama $_8$

- koliko elemenata se testira za listu dužine n?
- treba rešiti $n = 2^i$ po i, odnosno
- $i = log_2 n$

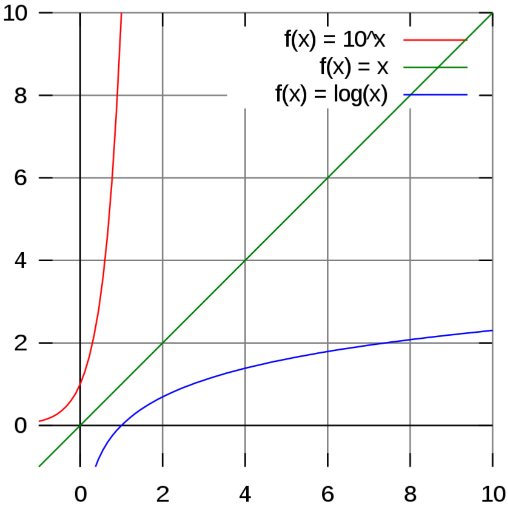
- binarna pretaga ima logaritamsku vremensku zavisnost
- vreme potrebno za pronalaženje raste sa logaritmom dužine liste

Poređenje algoritama $_8$

- telefonski imenik Njujorka: 12 miliona imena
- pronalaženje datog imena u imeniku:
- najviše $log_2 12000000 = 24$ pokušaja!

- za linearnu pretragu treba u proseku 6000000 pokušaja

Linearna vs logaritamska vs eksponencijalna funkcija



Pretraga u Pythonu

- zašto Python koristi linearnu pretragu?
- binarna pretraga očekuje sortiranu listu
- ako lista nije sortirana, prvo se mora sortirati

Podeli i vladaj

- binarna pretraga deli problem na dva dela
- princip podeli i vladaj (divide et impera):
- podeli problem na potprobleme koji su manja verzija početnog

Podeli i vladaj: binarna pretraga

- za binarnu pretragu početni opseg je cela lista
- testiramo srednji element…
- ako smo pogodili – kraj
- ako nismo – biramo polovinu i ponavljamo pretragu

Podeli i vladaj: binarna pretraga $_2$

```
def search(x, nums, low, high):
    if low > high:
        return -1
    mid = (low + high)//2
    if x == nums[mid]:
        return mid
    elsif x < nums[mid]:
        search(x, nums, low, mid-1)
    else:
        search(x, nums, mid+1, high)
```

- ova verzija nema petlju!
- poziva samu sebe!

Rekurzivna definicija

- definicija nečega koja se referiše na samu sebe je cirkularna definicija
  - ne treba definisati pojmove pomoću njih samih
- rekurzivna definicija, za razliku od cirkularne, ima slučajeve kada se ne definiše pomoću sebe same

- algoritam za binarnu pretragu koristi sebe samog
- „poziva“ istu funkciju unutar njene definicije
- ali ne za sve slučajeve!

Faktorijel pomoću rekurzije

- u matematici se rekurzivne definicije često koriste
- faktorijel: $5! = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1 = 5 \cdot 4!$
- odnosno $n! = n \cdot (n-1)!$

- specijalni slučaj: $0! = 1$
- konačna definicija:

$$n! = \begin{cases} 1 & \text{ako } n = 0 \\ n(n-1)! & \text{inače} \end{cases}$$

- zašto ovo nije cirkularna definicija?
- kada dođemo do $0!$ to je osnovni slučaj koji se rešava bez rekurzije

Rekurzivne definicije

- korektne rekurzivne definicije imaju dve bitne osobine:
  - postoji jedan ili više osnovnih slučajeva za koje nije potrebna rekurzija
  - svi lanci rekurzije se na kraju svedu na neki osnovni slučaj
- recept: svaka rekurzija treba da operiše nad manjom verzijom početnog problema
- vrlo mala verzija početnog problema koja se može rešiti bez rekurzije postaje osnovni slučaj

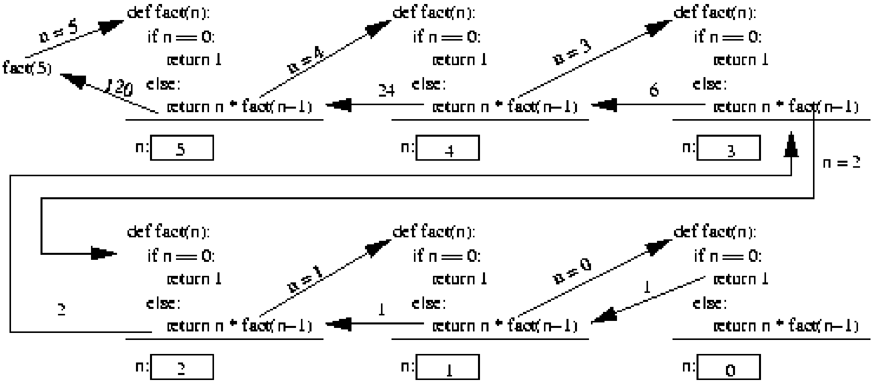
Faktorijel pomoću rekurzije

- ranije smo napravili faktorijel pomoću akumulatora
- sada pomoću rekurzije:

```
def fact(n):
    if n == 0:
        return 1
    else:
        return n * fact(n-1)
```

Faktorijel pomoću rekurzije $_2$

- svaki poziv funkcije ima svoje parametre i lokalne promenljive!



Primer: obrtanje redosleda znakova u stringu

- za svaku Python listu na raspolaganju je funkcija `reverse`
- ako hoćemo da konvertujemo string?

1. pretvorimo sting u listu znakova
2. okrenemo listu
3. pretvorimo listu u string

Primer: obrtanje redosleda znakova u stringu $_2$

- korišćenjem rekurzije možemo da obavimo ovu operaciju bez pomoćne liste
- polazna ideja za rekurziju:

1. podelimo string na prvi znak i ostatak stringa
2. okrenemo ostatak stringa
3. prvi znak dodamo na kraj okrenutog stringa

Primer: obrtanje redosleda znakova u stringu $_3$

```
def reverse(s):
    return reverse(s[1:]) + s[0]
```

- isečak `s[1:]` vraća sve osim prvog znaka
- okrenemo ovaj isečak i dodamo prvi znak `s[0]` na kraj

Primer: obrtanje redosleda znakova u stringu $_4$

```
>>> reverse("Hello")
...
  File "<stdin>", line 2, in reverse
RuntimeError: maximum recursion depth exceeded
```

- šta se desilo?
- funkcija ne sadrži osnovni slučaj koji se ne oslanja na rekurziju
- funkcija beskonačno poziva samu sebe: svaki poziv `reverse` dalje poziva `reverse`

Primer: obrtanje redosleda znakova u stringu $_5$

- svaki poziv funkcije troši malo memorije
- Python zaustavlja rekurziju posle 1000 poziva
- šta će biti naš osnovni slučaj?
- okretanje praznog stringa daje isti rezultat!

Primer: obrtanje redosleda znakova u stringu $_6$

```
def reverse(s):
    if s == "":
        return s
    else:
        return reverse(s[1:]) + s[0]

>>> reverse("Hello")
'olleH'
```

Primer: anagrami

- anagram se dobija menjanjem redosleda slova u reči
- pravljenje anagrama je specijalni slučaj generisanja svih permutacija nekog niza

- pristup iz prošlog primera:

1. iseci prvi znak iz stringa
2. ubaci prvi znak u sve moguće pozicije anagrama koji je dobijen od ostatka stringa

Primer: anagrami $_2$

- neka je originalni string `"abc"`
- isecanjem `"a"` ostaje nam `"bc"`
- rezultat generisanja anagrama za `"bc"` su stringovi `"bc"` i `"cb"`
- da bismo formirali anagram originalnog stringa, ubacićemo ”a” na sve moguće pozicije u okviru dva manja anagrama
- `["abc", "bac", "bca", "acb", "cab", "cba"]`

Primer: anagrami $_2$

- i u ovom primeru prazan string može poslužiti kao osnovni slučaj za rekurziju

```
def anagrams(s):
    if s == "":
        return [s]
    else:
        ans = []
        for w in anagrams(s[1:]):
            for pos in range(len(w)+1):
                ans.append(w[:pos]+s[0]+w[pos:])
        return ans
```

Primer: anagrami $_3$

- lista se koristi za akumuliranje rezultata
- spoljašnja `for` petlja iterira kroz sve anagrame isečka od `s`
- unutrašnja petlja za svaku poziciju u anagramu ubacuje slovo i kreira novi string
- unutrašnja petlja se ponavlja sve do `len(w)+1` da bi novo slovo moglo da se ubaci i na kraj anagrama

Primer: anagrami $_4$

`w[:pos]+ s[0] + w[pos:]`

- `w[:pos]` vraća isečak od `w` sve do (ali ne uključujući) pozicije `pos`
- `w[pos:]` vraća sve od pozicije `pos` do kraja
- ubacivanje `s[0]` zapravo ubacuje znak na poziciju `pos`
- unutrašnja petlja se ponavlja sve do `len(w)+1` da bi novo slovo moglo da se ubaci i na kraj anagrama

Primer: anagrami $_5$

```
>>> anagrams("abc")
['abc', 'bac', 'bca', 'acb', 'cab', 'cba']
```

- broj različitih anagrama jednak je faktorijelu dužine stringa

Primer: brzo stepenovanje

- jedan način da izračunamo a^n je da pomnožimo a ukupno n puta
- ovo se može lako napraviti pomoću akumulatorske petlje:

```
def loopPower(a, n):
    ans = 1
    for i in range(n):
        ans = ans * a
    return ans
```

Primer: brzo stepenovanje $_2$

- ovaj problem možemo rešavati i strategijom podeli-i-vladaj
- znamo da je $2^8 = 2^4 \cdot 2^4$
- ako znamo koliko je 2^4, možemo izračunati 2^8 pomoću jednog množenja
- dalje, $2^4 = 2^2 \cdot 2^2$
- dalje $2^2 = 2 \cdot 2$

- izračunali smo $2 \cdot 2 = 4$, $4 \cdot 4 = 16$, $16 \cdot 16 = 256 = 2^8$ pomoću samo tri množenja!

Primer: brzo stepenovanje $_3$

- znamo da je $a^n = a^{n/2} \cdot a^{n/2}$
- šta ako je n neparan broj?
- $2^9 = 2^4 \cdot 2^4 \cdot 2^1$

- opšti zakon:

$$a^n = \begin{cases} a^{n//2} \cdot a^{n//2} & \text{ako je } n \text{ paran broj} \\ a^{n//2} \cdot a^{n//2} \cdot a & \text{ako je } n \text{ neparan broj} \end{cases}$$

Primer: brzo stepenovanje $_3$

- ovde se oslanjamo na celobrojno deljenje: $9//2 = 4$
- za rekurzivni algoritam treba nam osnovni slučaj
- kako budemo polovili n doći ćemo do nule, jer $1//2 = 0$
- dalje $a^0 = 1$ za svako a osim nule

Primer: brzo stepenovanje $_4$

```
def recPower(a, n):
    if n ==  0:
        return 1
    else:
        factor = recPower(a, n//2)
        if n%2 == 0:     # n je paran
            return factor * factor
        else:            # n je neparan
            return factor * factor * a
```

- privremena promenljiva `factor` je uvedena da ne bismo dva puta morali da izračunavamo $a^{n//2}$ više od jedanput

Primer: binarna pretraga

- ispitamo srednji element u listi, i potom pretražujemo donju ili gornju polovinu liste
- osnovni slučajevi za rekurziju: kada smo našli podatak ili kada smo suzili pretragu na praznu listu
- rekurzivni pozivi će poloviti opseg koji je „još u igri“
- svaki poziv funkcije će tražiti element između indeksa `low` i `high`

Primer: binarna pretraga $_2$

```
def recBinSearch(x, nums, low, high):
    if low > high:    # nema više gde da se traži
        return -1
    mid = (low + high)//2
    item = nums[mid]
    if item == x:
        return mid
    elif x < item:    # traži u donjoj polovini
        return recBinSearch(x, nums, low, mid-1)
    else:             # traži u gornjoj polovini
        return recBinSearch(x, nums, mid+1, high)
```

Primer: binarna pretraga $_3$

- da bi se prethodna funkcija lakše koristila možemo napraviti novu funkciju koja „sakriva“ parametre `low` i `high`

```
def search(x, nums):
    return recBinSearch(x, nums, 0, len(nums)-1)
```

Rekurzija vs iteracija

- postoje sličnosti između iteracije (petlje) i rekurzije
- sve što se može rešiti pomoću petlje, može i pomoću rekurzije
- neki programski jezici imaju samo rekurziju!
- neki problemi koji se lako rešavaju rekurzijom su komplikovani za rešavanje petljom
- ...i obrnuto

Rekurzija vs iteracija $_2$

- izračunavanje faktorijela i binarna pretraga na sličan način koriste petlje i rekurziju i efikasnost im je slična
- kod stepenovanja je drugačije:
  - verzija sa petljom je linearne složenosti
  - verzija sa rekurzijom je logaritamske složenosti
  - razlika u efikasnosti je kao između linearne i binarne pretrage
- da li će rekurzivna rešenja uvek biti brža ili bar jednako brza kao iterativna?

Rekurzija vs iteracija $_3$

- Fibonačijevi brojevi predstavljaju niz 1,1,2,3,5,8,...
  - sekvenca počinje sa 1,1
  - sledeći broj se računa kao zbir prethodna dva

Fibonacci iterativno

- promenljive `curr` i `prev` služe za računanje sledećeg broja u nizu
- kada ga izračunamo, postavimo `prev` na `curr` i `curr` na novoizračunati broj
- stavimo ovo u petlju i izvršimo potreban broj ciklusa

Fibonacci iterativno $_2$

```
def loopfib(n):
    curr = 1
    prev = 1
    for i in range(n-2):
        curr, prev = curr+prev, curr
    return curr
```

- iskoristili smo dvostruku dodelu
- petlja izvršava $n-2$ ciklusa jer prva dva broja već imamo

Fibonacci rekurzivno

$$fib(n) = \begin{cases} 1 & \text{ako je } n < 3 \\ fib(n-1) + fib(n-2) & \text{inače} \end{cases}$$

- lako je napisati funkciju na osnovu ovakve definicije:

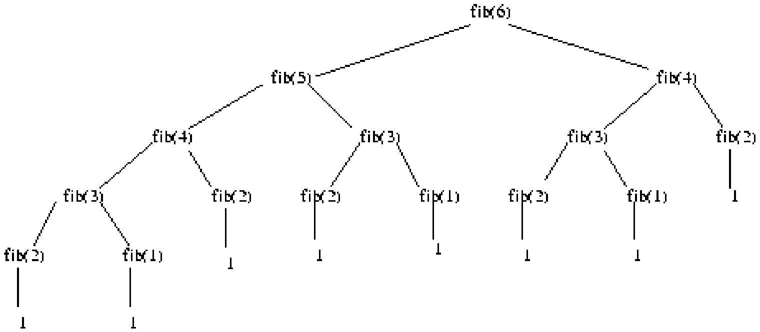
```python
def fib(n):
    if n < 3:
        return 1
    else:
        return fib(n-1)+fib(n-2)
```

Fibonacci rekurzivno $_2$

- ova funkcija poštuje sva pravila koja smo ranije postavili:
  - rekurzija se zasniva na manjim vrednostima
  - postoji osnovni slučaj koji se ne oslanja na rekurziju
- da li će rekurzivna varijanta raditi efikasno?...

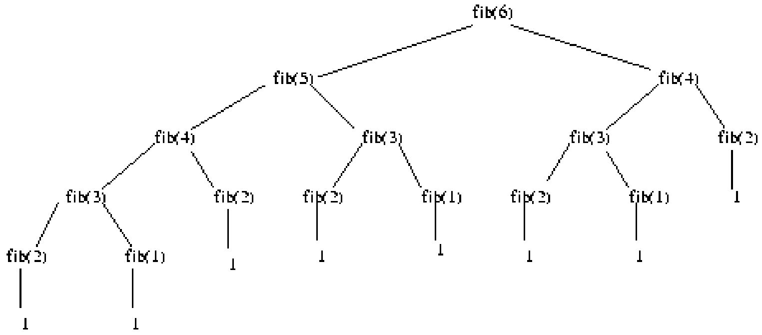
Fibonacci rekurzivno $_3$

- rekurzivno rešenje je neefikasno jer obavlja puno ponovljenih izračunavanja!



Fibonacci rekurzivno 4

- da bismo izračunali `fib(6)`, `fib(4)` će se računati dva puta, `fib(3)` će se računati 3 puta, `fib(2)` će se računati 4 puta...
- za velike brojeve ima jako puno ponovljenih proračuna



Rekurzija vs iteracija

- rekurzija je još jedan alat za rešavanje problema
- ponekad je rekurzija efikasnija od iterativnog rešenja
- u drugim slučajevima, kada su rešenja slična, iterativno rešenje je obično brže
- rekurzivna rešenja treba izbegavati ako su neefikasna, ili kada ne možemo napraviti iterativno rešenje

Algoritmi za sortiranje

- osnovni problem sortiranja:
- prerasporediti elemente liste tako da su poređani u rastućem ili opadajućem redosledu

Selection sort

- dat nam je špil karata koje treba da poređamo u rastućem redosledu
- kako da to uradimo?
- pronađemo najmanju kartu (sekvencijalno) i stavimo je na vrh špila
- ponovimo postupak za ostatak špila
- sve ovo ponavljamo dok ne sortiramo ceo špil

- ovaj algoritam je poznat kao selection sort

Selection sort $_2$

- algoritam ima petlju
- u svakom ciklusu nađemo najmanji element i premestimo ga na početak
  - za svih n elemenata: nađemo najmanji i premestimo ga na nultu poziciju
  - za elemente sa pozicijama 1 do $n-1$: nađemo najmanji i premestimo ga na prvu poziciju
  - za elemente sa pozicijama 2 do $n-1$: nađemo najmanji i premestimo ga na drugu poziciju
  - ...

Selection sort $_3$

- kada premeštamo element na nultu poziciju, ne smemo da izgubimo element koji se tamo nalazi!
- možemo da im zamenimo mesta:
  `nums[0], nums[x] = nums[x], nums[0]`
- možemo da implementiramo algoritam:
- `bottom` je pozicija za najmanji element
- `mp` je pozicija najmanjeg koga smo pronašli

Selection sort $_4$

```
def selectionSort(nums):
    n = len(nums)
    for bottom in range(n-1):
        mp = bottom          # bottom je trenutno najmanji
        for i in range(bottom+1, n): # za svaku poziciju
            if nums[i] < nums[mp]:   # ako je manji
                mp = i               # zapamti mu indeks

        # zameni mesta
        nums[bottom], nums[mp] = nums[mp], nums[bottom]
```

Selection sort $_4$

- umesto da pamtimo najmanju vrednost, pamtimo joj poziciju u promenljivoj `mp`
- nove vrednosti se testiraju poređenjem elementa sa indeksom `i` i elementa sa indeksom `mp`
- `bottom` se zaustavlja na indeksu `n-2`: kada nam ostane samo jedan element, on je i najveći!

Selection sort $_5$

- selection sort se lako implementira
- pristojno radi za liste skromne dužine
- nije efikasan

Merge sort

- primena podeli-i-vladaj pristupa na sortiranje
- recimo da smo dobili špil karata da sortiramo

1. podelimo špil na dva dela
2. sortiramo prvu polovinu
3. sortiramo drugu polovinu
4. spojimo dve sortirane polovine (tako da rezultat bude sortiran)

Merge sort $_2$

- proces kombinovanja dve sortirane liste u jednu zove se spajanje (merging)
- korak 1: podeli špil na dva dela
    - ovo je lako: koristićemo isecanje liste
- korak 4: spajanje sortiranih polovina
    - uporedimo dve karte na vrhu svake polovine; manja od njih će biti na vrhu celog špila
    - kada sklonimo najmanju kartu, ponovo poredimo najmanje karte u obe polovine
    - ponavljamo postupak dok ne potrošimo jednu polovinu; ostatak druge polovine dodamo na kraj špila

Merge sort $_3$

- `lst1` i `lst2` su dve pod-liste
- `lst3` je rezultat

Merge sort $_4$

```
def merge(lst1, lst2, lst3):
    # indeksi tekućih pozicija u svakoj od lista
    i1, i2, i3 = 0, 0, 0  # svi počinju od početka
    n1, n2 = len(lst1), len(lst2)

    # dok u obe polovine ima elemenata
    while i1 < n1 and i2 < n2:
        if lst1[i1] < lst2[i2]: # vrh iz lst1 je manji
            lst3[i3] = lst1[i1] # kopiraj ga u lst3 na kraj
            i1 = i1 + 1
        else:                   # vrh iz lst2 je manji
            lst3[i3] = lst2[i2] # kopiraj ga u lst3 na kraj
            i2 = i2 + 1
        i3 = i3 + 1             # pomeri indeks za lst3
```

Merge sort $_5$

```
# Sada smo ispraznili lst1 ili lst2. Iskopiraj ostatak iz
# liste koja nije prazna u lst3.

    # kopiraj elemente iz lst1 ako ih ima
    while i1 < n1:
        lst3[i3] = lst1[i1]
        i1 = i1 + 1
        i3 = i3 + 1

    # kopiraj elemente iz lst2 ako ih ima
    while i2 < n2:
        lst3[i3] = lst2[i2]
        i2 = i2 + 1
        i3 = i3 + 1
```

Merge sort $_6$

- možemo da podelimo listu na polovine
- i da spojimo podeljene liste nazad
- kako da sortiramo polovine? – novim deljenjem
- liči na rekurziju!

- treba nam osnovni slučaj kada rekurzija prestaje
- i da rekurzivni pozivi uvek operišu nad manjom varijantom početnog problema

Merge sort $_7$

- na kraju deobe ćemo doći do liste dužine 1
- ta lista je sortirana!
- da skiciramo rekurziju:

```
if len(nums) > 1:
    podeli listu na dva dela
    mergeSort(prva polovina)
    mergeSort(druga polovina)
    spoj dve liste u jednu
```

Merge sort 8

```python
def mergeSort(nums):
    n = len(nums)
    if n > 1:
        m = n//2
        nums1, nums2 = nums[:m], nums[m:]
        mergeSort(nums1)
        mergeSort(nums2)
        merge(nums1, nums2, nums)
```

Selection sort vs merge sort

- imamo dva algoritma za sortiranje; koji da koristimo?
- složenost postupka sortiranja zavisi od dužine liste
- treba da odredimo u koliko koraka svaki algoritam obavi sortiranje

Selection sort vs merge sort $_2$

- da krenemo sa selection sortom
- lista je dužine n
- za pronalaženje najmanjeg elementa, testiramo svih n elemenata
- u sledećem ciklusu testiramo $n-1$ elemenata
- u sledećem ciklusu testiramo $n-2$ elemenata
- ...
- ukupan broj iteracija je:

$$n + (n-1) + (n-2) + (n-3) + \ldots + 1$$

Selection sort vs merge sort $_3$

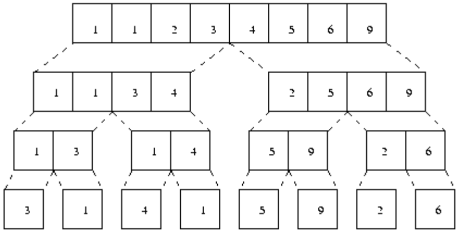
- vreme potrebno selection sortu da sortira listu dužine n
- je proporcionalno sumi prvih n prirodnih brojeva, odnosno

$$\frac{n(n+1)}{2}$$

- u ovoj formuli se pojavljuje n^2
- što znači da je broj potrebnih koraka proporcionalan kvadratu dužine liste
- $\rightarrow$ algoritam kvadratne složenosti

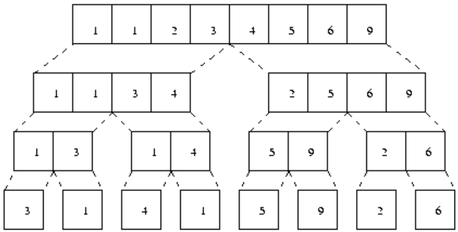
Selection sort vs merge sort $_4$

- sada analiziramo merge sort
- lista se podeli na dve polovine, svaka se sortira i potom se spoje
- pravo mesto gde se sortiranje odvija je funkcija `merge`
- ova slika pokazuje kako se sortira lista `[3,1,4,1,5,9,2,6]`
- idući od dole, treba da kopiramo n vrednosti u drugi nivo



Selection sort vs merge sort $_5$

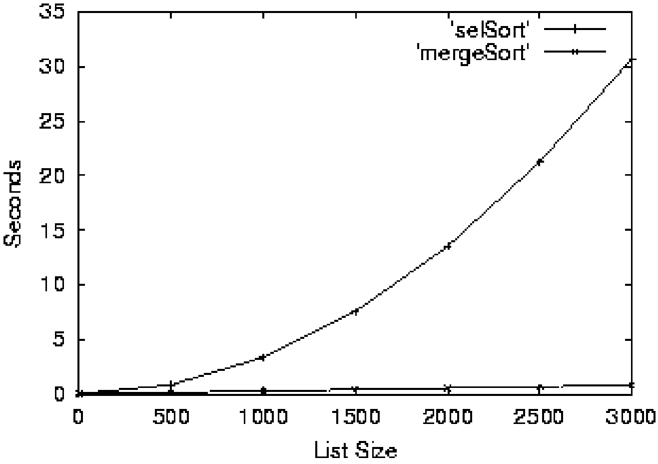
- od drugog prema trećem nivou treba opet kopirati n vrednosti
- svaki nivo podrazumeva kopiranje n vrednosti
- koliko ima nivoa?
- na osnovu analize binarne pretrage, znamo da ih ima $log_2 n$
- ukupan broj koraka za sortiranje je $n \cdot log_2 n$



Selection sort vs merge sort $_6$

- koji algoritam je bolji: selection sa n^2 ili merge sa $nlog_2n$ operacija?
- za kratke liste selection sort može biti brži jer je kôd jednostavniji
- kada n poraste...
- funkcija $nlog_2n$ raste znatno sporije od n^2

Selection sort vs merge sort $_7$

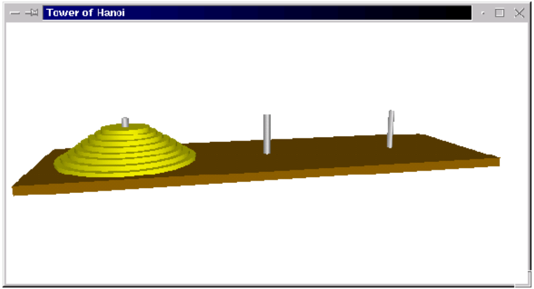


Složeni problemi

- pomoću podeli-i-vladaj pristupa možemo napraviti efikasne algoritme za sortiranje i pretragu
- podeli-i-vladaj i rekurzija su moćne tehnike za dizajn algoritama
- ali nemaju svi problemi efikasno rešenje!

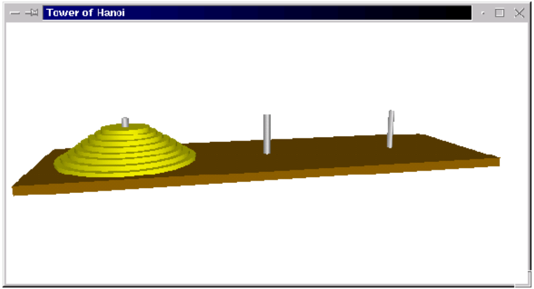
Hanojske kule

- imamo tri stuba i 64 koncentrična diska različitih prečnika, poređanih u obliku kupe
- treba premestiti diskove sa jednog stuba na drugi pomoću tri pravila:
  - samo jedan disk može da se pomera u jednom trenutku
  - disk se sme spustiti samo na stub, ne može se ostaviti sa strane
  - veći disk se ne sme spustiti na manji



Hanojske kule $_2$

- ako stubove označimo sa A, B i C
- operacije za premeštanje su
  - premesti disk sa A na C
  - premesti disk sa A na B
  - premesti disk sa C na B



Hanojske kule $_3$

- jednostavni slučajevi:
- 1 disk
    - premesti disk sa A na C
- 2 diska
    - premesti disk sa A na B
    - premesti disk sa A na C
    - premesti disk sa B na C

Hanojske kule $_4$

- 3 diska
  - da bismo premestili najveći na C, treba prvo da sklonimo 2 manja diska
  - ova dva diska čine kupu visine 2, što znamo da rešimo

Hanojske kule $_5$

- algoritam: premesti kulu sa n diskova
  1. premesti kulu visine $n-1$ sa početnog položaja na pomoćni
  2. premesti kulu visine 1 na krajnji položaj
  3. premesti kulu visine $n-1$ sa pomoćnog položaja na krajnji
- osnovni slučaj: premeštanje kule visine 1

Hanojske kule $_6$

- `n` je visina kule
- `source`, `dest`, `temp` su tri stuba

```
def moveTower(n, source, dest, temp):
    if n == 1:
        print("Premesti disk sa", source, "na", dest+".")
    else:
        moveTower(n-1, source, temp, dest)
        moveTower(1, source, dest, temp)
        moveTower(n-1, temp, dest, source)
```

Hanojske kule $_7$

- napravimo funkciju koja započinje postupak

```
def hanoi(n):
    moveTower(n, "A", "C", "B")
```

```
>>> hanoi(3)
Premesti disk sa A na C.
Premesti disk sa A na B.
Premesti disk sa C na B.
Premesti disk sa A na C.
Premesti disk sa B na A.
Premesti disk sa B na C.
Premesti disk sa A na C.
```

Hanojske kule $_8$

- ovo je „složen problem“
- za visinu kule n treba nam $2^n - 1$ koraka

diskova	koraka
1	1
2	3
3	7
4	15
5	31

Hanojske kule $_8$

- algoritam ima eksponencijalnu složenost
- potrebno vreme raste veoma brzo sa n

- za 64 diska, pomerajući jedan u sekundi, potrebno je 580 milijardi godina da se završi
- (starost svemira je oko 15 milijardi godina)

Hanojske kule $_9$

- algoritam za hanojske kule se lako definiše
- ali je neupotrebljiv za rešavanje problema osim za trivijalne slučajeve – intractable
- postoje problemi koji su još složeniji

Problem zaustavljanja

- pišemo program koji analizira druge programe i određuje da li imaju beskonačnu petlju
- znamo i ulaz u program koji analiziramo
- da izbegnemo slučaj da program ima beskonačnu petlju zbog određene kombinacije ulaznih podataka

Problem zaustavljanja $_2$

- specifikacija programa
  - ulaz: fajl sa Python programom; ulazni podaci za njega
  - izlaz: `"OK"` ako će se program zaustaviti, `"FAULT"` ako ima beskonačnu petlju
- npr. Python interpreter je program koji analizira programe!
- program i njegovi ulazi se mogu predstaviti kao stringovi

Problem zaustavljanja $_3$

- ne postoji algoritam koji može ispuniti ovu specifikaciju!
- nije isto kao kada kažemo da niko ne može da ga napiše
- možemo dokazati ovu tvrdnju matematičkom tehnikom koja se zove dokaz kontradikcijom

Problem zaustavljanja $_4$

- dokaz kontradikcijom: pretpostavićemo da važi suprotno od onoga što želimo da dokažemo
- ako nas ta pretpostavka dovede u kontradikciju, to znači da pretpostavka nije tačna
- tada će biti tačno ono što smo želeli da dokažemo

Problem zaustavljanja $_5$

- pretpostavimo da postoji algoritam koji može da odredi da li se neki program završava za date ulazne podatke
- ako postoji, možemo ga zapisati kao funkciju:

```
# turing.py

def terminates(program, inputData):
    # program i inputData su stringovi
    # vraća True ako će se program zaustaviti kada mu se
    # daju inputData kao ulazni podaci
```

Problem zaustavljanja $_6$

```
def main():
    lines = []
    print("Ukucaj program ('done' za kraj).")
    line = input("")
    while line != "done":
        lines.append(line)
        line = input("")
    testProg = "\n".join(lines)

    # ako se program zaustavi kad dobije samog sebe
    # kao ulazne podatke, uđi u beskonačnu petlju
    if terminates(testProg, testProg):
        while True:
            pass        # pass naredba ne radi ništa
```

Problem zaustavljanja $_7$

- prvo unosimo program sa tastature, pomoću sentinel petlje
- `join` funkcija spaja sve redove stavljajući `\n` između njih
- dobili smo višelinijski string koji sadrži ukucani program

Problem zaustavljanja 8

- `turing.py` koristi ovaj program za testiranje, ali koristi ga i kao ulazne podatke
- dakle, proveravamo da li će se program zaustaviti ako mu prosledimo samoga sebe kao ulaz
- ako se program zaustavi, `turing.py` će ući u beskonačnu petlju

Problem zaustavljanja $_9$

- ovo je sve uvod u glavno pitanje:
- šta će se desiti kada pokrenemo `turing.py` i damo mu `turing.py` da ga analizira?
- da li se `turing.py` zaustavlja kada dobije samog sebe kao ulaz?

Problem zaustavljanja 10

- u funkciji `terminates` se određuje da li će se program zaustaviti
- dva moguća slučaja:
  1. `turing.py` će se **zaustaviti** kada mu damo njega samog kao ulaz
     - funkcija `terminates` vraća `True`
     - `turing.py` potom ulazi u beskonačnu petlju
     - prema tome, `turing.py` se **ne zaustavlja** – kontradikcija
  2. `turing.py` će se **neće zaustaviti** kada mu damo njega samog kao ulaz
     - funkcija `terminates` vraća `False`
     - kada `terminates` vrati `False`, `turing.py` se završava
     - `turing.py` se **zaustavio** – kontradikcija

Problem zaustavljanja $_{11}$

- pretpostavka da postoji funkcija `terminates` nas je dovela u kontradikciju
- $\Rightarrow$ takva funkcija ne postoji!

Glava

- najvažniji kompjuter koji programer ima je onaj između ušiju