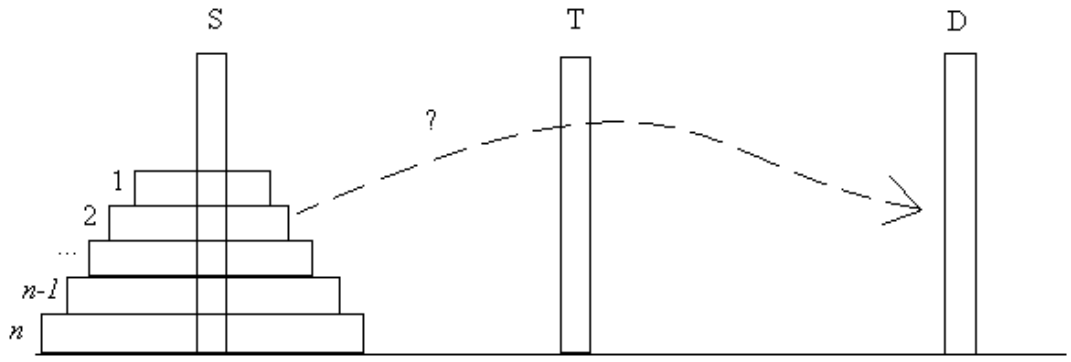
ALGORITMI I STRUKTURE PODATAKA – GRUPA 9
DODACI UZ ČETVRTI I PETI TJEDAN NASTAVE
Robert Manger, 14. ožujka 2007.

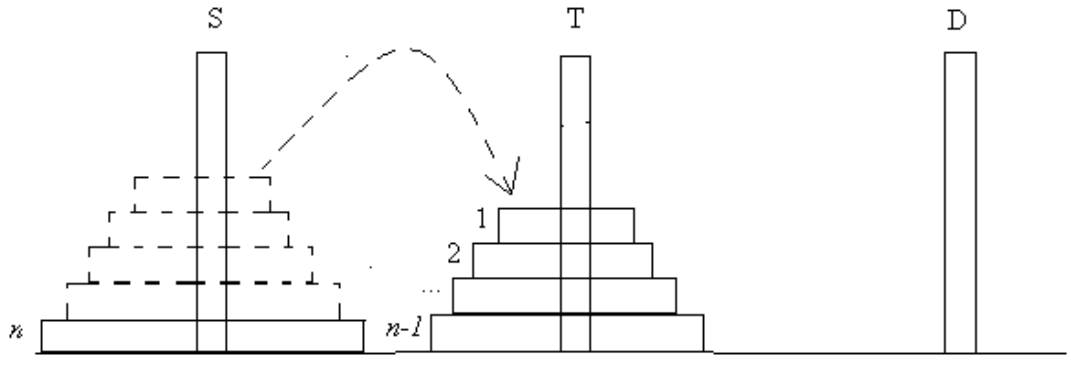
Hanojski tornjevi

Najprije opisujemo problem. Postoje tri štapa, S (source, izvor), D (destination, odredište) i T (temporary, pomoćni). Na štap S nabodeno je *n* diskova različitih veličina tako da je manji disk uvijek iznad većeg. Potrebno je premjestiti sve diskove na D, jedan po jedan. Pritom su dozvoljena samo takva premještanja gdje se disk s vrha jednog štapa stavlja na prazni drugi štap, ili na veći disk koji je na vrhu drugog štapa.

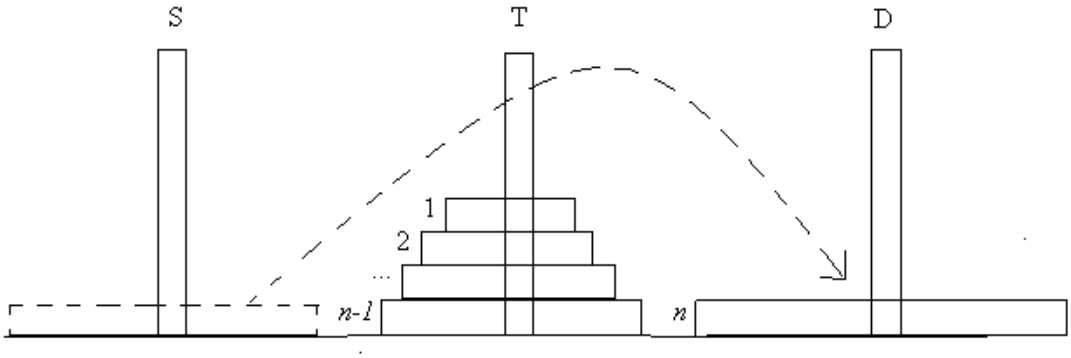


Dalje opisujemo rekurzivno rješenje problema. Pretpostavimo da već znamo riješiti problem za *n*-1 diskova. Tada se rješenje za *n* diskova dobiva u sljedeća tri koraka.

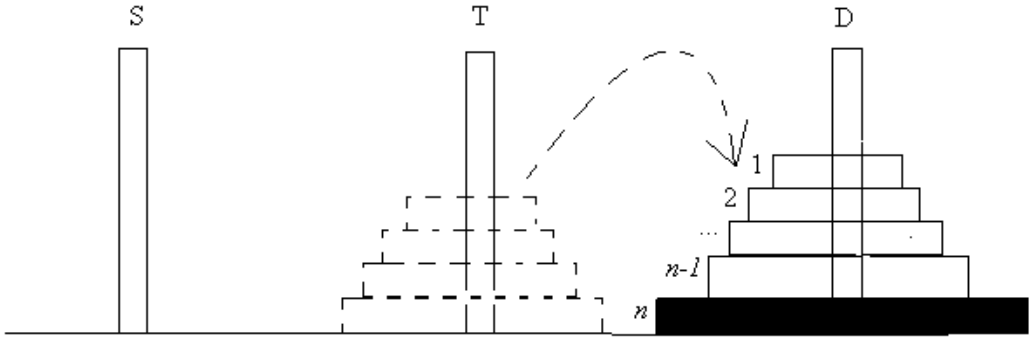
1. Rješavanjem problema za *n*-1 diskova, prebacimo gornjih *n*-1 diskova sa S na T koristeći D kao pomoćni štap.



2. Izravno premjestimo najveći *n*-ti disk sa S na D.



3. Rješavanjem još jednog problema za *n*-1 diskova, prebacimo *n*-1 diskova s T na D (dakle na *n*-ti disk) koristeći S kao pomoćni štap.



Vidimo da rješavanje problema s *n* diskova zahtijeva rješavanje dva problema s *n*-1 diskova. Budući da povećanje *n* za 1 udvostručuje količinu posla, složenost ovog algoritma je očito $O(2^n)$.

Složenost pri rekurziji – primjer programa RazneSloženosti

U ovom primjeru povezujemo ono što smo prethodno naučili o složenosti algoritama i o rekurziji. Bavit ćemo se rješavanjem jednog malo kompliciranijeg studijskog problema. Taj problem riješit ćemo na više načina pomoću različitih algoritama. Analizirat ćemo vremensku složenost svakog od algoritama. Vidjet ćemo da se složenosti bitno razlikuju, to jest imaju različite redove veličine. Također ćemo vidjeti da rekurzija može dati vrlo efikasno rješenje čija korektnost je još uvijek očigledna. Doduše, pokazat ćemo da postoji i još efikasnije ne-rekurzivno rješenje, no ono ima manu da je prilično nerazumljivo, dakle takvo da bi njegovu točnost morali opravdati matematičkim dokazom.

Naš studijski problem glasi ovako. Zadano je polje cijelih brojeva:

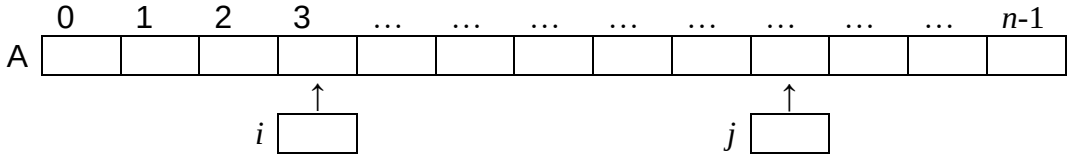
A[0], A[1], A[2], … A[n-1].

Ti brojevi mogu biti pozitivni, negativni ili nule. Potrebno je pronaći najveću moguću vrijednost sume niza uzastopnih članova polja. Dakle gledamo sume oblika:

A[i] + A[i+1] + …. + A[j],

za neke i, j, $i \leq j$. Htjeli bi odrediti i, j tako da suma bude maksimalna. Dozvoljavamo i „prazni“ niz uzastopnih članova polja, dakle sumu koja se sastoji od nula pribrojnika i koja je po definiciji jednaka 0. Zbog postojanja praznog niza, rješenje problema je uvijek ≥ 0. Primijetimo da u slučaju kad su svi brojevi u polju A negativni prazni niz sa sumom 0 postaje rješenje problema. Također, primijetimo da u slučaju kad postoji barem jedan ne-negativni broj rješenje možemo konstruirati kao ne-prazan niz.

Prvo rješenje studijskog problema svodi se na izravno ispitivanje svih mogućih nizova uzastopnih članova polja.



Algoritam se realizira kao tri ugniježđene for petlje. Brojač i u vanjskoj petlji predstavlja indeks početnog člana niza. Brojač j u srednjoj petlji predstavlja indeks zadnjeg člana niza. Variraju se sve mogućnosti za i, j takve da je $i \leq j$. Unutrašnja petlja s brojačem k služi za zbrajanje elemenata A[i], A[i+1], … , A[j]. Suma trenutno promatranog niza uspoređuje se s najboljom do tada pronađenom sumom. Prazni niz je uzet u obzir na taj način što je varijabla koja pamti najbolju do tada pronađenu sumu inicijalizirana na 0.

Algoritam je zapisan kao funkcija MaxPodSumaNiza3( ) unutar programa RazneSlozenosti. Budući da se MaxPodSumaNiza3( ) sastoji od 3 ugniježđene petlje, od kojih svaka ima u najgorem slučaju n koraka, očito je da je složenost te funkcije $O(n^3)$.

Drugo rješenje studijskog problema dobiva se poboljšanjem prvog rješenja. Primijetimo da se niz koji se promatra u nekom koraku srednje petlje neznatno razlikuje od niza koji se promatrao u prethodnom koraku iste petlje. Naime, niz u trenutnom koraku izgleda ovako:

A[*i*], A[*i*+1], … , A[*j*-1], A[*j*],

Dok je niz u prethodnom koraku bio:

A[*i*], A[*i*+1], … , A[*j*-1].

Suma niza u trenutnom koraku može se dobiti iz zapamćene sume iz prethodnog koraka jednim zbrajanjem, to jest tako da zapamćenoj sumi još pribrojimo A[*j*]. Dakle unutrašnja petlja s brojačem *k* je suvišna.

Novi algoritam zapisan je kao funkcija MaxPodSumaNiza2( ) unutar programa RazneSlozenosti. Tekst od MaxPodSumaNiza2( ) dobiven je iz teksta MaxPodSumaNiza3( ) izbacivanjem unutrašnje petlje s brojačem *k* i premještanjem naredbe za inicijalizaciju trenutne sume. Budući da se MaxPodSumaNiza2( ) sastoji od 2 ugniježdene petlje, od kojih svaka ima u najgorem slučaju *n* koraka, složenost te funkcije je $O(n^2)$. Točnijim prebrajanjem dobili bi da je broj operacija proporcionalan s

$n + (n\text{-}1) + (n\text{-}2) + \ldots + 1 = n(n\text{-}1)/2 = O(n^2)$.

Treće rješenje dobiva se primjenom rekurzije. Podijelimo polazno polje po sredini na dva podjednako dugačka dijela:

A[0], A[1], …, A[*n*/2-1] | A[*n*/2], …, A[*n*-1].

Tada se niz uzastopnih članova polja koji predstavljaju rješenje problema može nalaziti:
- u lijevoj polovici polja,
- u desnoj polovici polja,
- na granici, tako da dijelom zahvaća lijevu a dijelom desnu polovicu polja.

Da bi riješili problem, moramo ispitati sva tri slučaja, usporediti rješenja, te uvažiti najbolje rješenje. Maksimalne sume za prva dva slučaja mogu se dobiti rekurzivnim pozivima istog algoritma na lijevom odnosno desnom dijelu polja. Maksimalna suma za treći slučaj može se dobiti (u linearnom vremenu) ovako:
- u lijevom dijelu polja nađe se najveća suma niza koji „dodiruje“ desnu granicu lijevog dijela polja:

variramo fiksirano
↓ … (zbrajamo) … ↓
A[0], …, …, …, …, …, A[*n*/2-1] | …

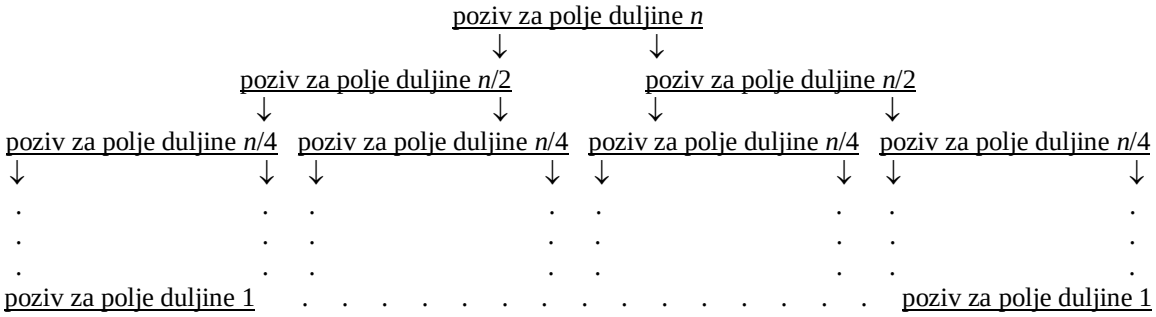
- u desnom dijelu polja nađe se najveća suma niza koji „dodiruje“ lijevu granicu desnog dijela polja:

fiksirano variramo
↓ … (zbrajamo) … ↓
… | A[*n*/2], …, …, …, …, …, A[*n*-1]

– te dvije najveće sume iz dvaju različitih dijelova polja zbrojimo.

Konkretan primjer rekurzivnog rješavanja problema s poljem duljine n=8 nalazi se u službenoj prezentaciji na slajdu 33. Cijeli algoritam zapisan je u programu RazneSlozenosti( ) kao funkcija MaxPodSumaNizaLog( ), odnosno kao pomoćna funkcija MaxPodSuma( ). Algoritam neće sam sebe pozivati onda kad je duljina polja jednaka 1.

Da bi ocijenili složenost rekurzivnog algoritma, primijetimo da se njegovi rekurzivni pozivi mogu prikazati binarnim stablom:

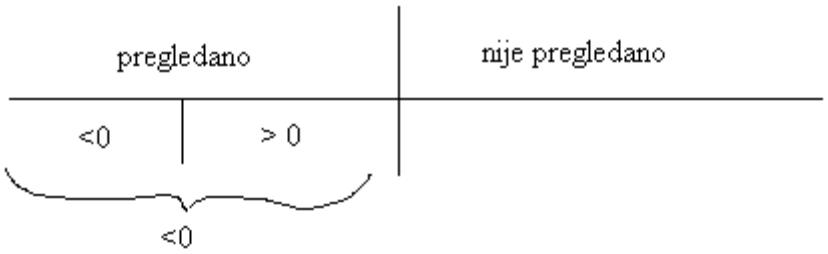


Budući da je broj operacija u svakom rekurzivnom pozivu proporcionalan s duljinom razmatranog (pod)polja, te budući da je ukupan broj elemenata za sva (pod)polja na istoj razini jednak n, slijedi da je zbroj složenosti za sve pozive na istoj razini $O(n)$. Broj razina je $(\log_2 n + 1)$. Dakle ukupna složenost je $O(n\ (\log_2 n + 1)) = O(n \log n)$.

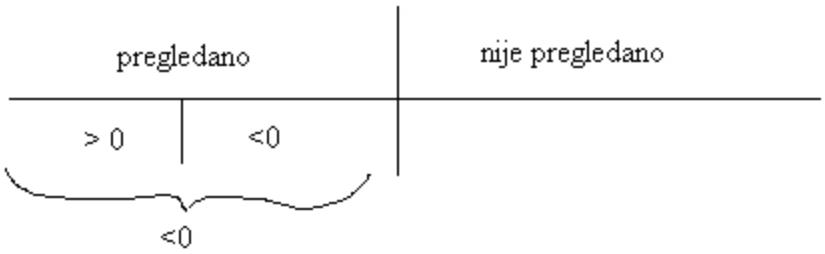
Četvrto rješenje studijskog primjera ima složenost $O(n)$ i svodi se na jednu petlju koja redom pribraja elemente polja tekućoj sumi. Kad god ta suma ispadne negativna, ona se ponovo postavlja na 0. Pamti se najveća vrijednost tekuće sume koja se pojavila tijekom cijelog postupka. Ta najveća vrijednost je rješenje problema.

Korektnost ovog postupka nije sasvim očigledna, i trebalo bi je matematički dokazati. Ključ za razumijevanje korektnosti je činjenica da se negativna tekuća suma pretvara u 0. Opravdanje za tu promjenu ide otprilike ovako.

– Ako je u trenutku pojave negativne sume već bilo pronađeno rješenje problema, ono je već zapamćeno i nastavak rada algoritma je nevažan.
– Ako u trenutku pojave negativne sume još nije bilo pronađeno rješenje problema, tada se to rješenje nalazi u još nepregledanom dijelu polja. Dodavanje pregledanog dijela polja samo bi pokvarilo rješenje jer je njegova suma negativna, a eventualni pozitivni brojevi nalaze se na lijevom kraju pregledanog dijela, pa ih ne možemo dodati bez negativnih brojeva na desnom kraju pregledanog dijela.



Ili još detaljnije: situacija s gornje slike, gdje na desnom rubu pregledanog dijela polja postoje pozitivni brojevi, nije moguća. Naime tada bi na lijevom rubu morali postojati negativni brojevi koji cijelu sumu pregledanog dijela čine negativnom, no onda bi već prije suma tog lijevog kraja bila pretvorena u 0.



Situacija s ove slike je moguća, ali nam ne pomaže. Naime da bi uključili pozitivne brojeve s lijevog kraja pregledanog dijela, morali bi uključiti i negativne brojeve s desnog kraja, pa bi zbroj uključenih brojeva iz pregledanog dijela bio <0 i pokvario bi rješenje sastavljeno od nepregledanih brojeva.

Složenost pri rekurziji – primjer programa ModPolja

U ovom primjeru opet ćemo se baviti rješavanjem jednog problema, te ćemo taj problem riješiti na više načina pomoću različitih algoritama. Za razliku od prošlog primjera, svi algoritmi sada će imati istu (linearnu) složenost. No analiza aposteriori (mjerenje vremena izvršavanja) pokazat će da se stvarna brzina algoritama ipak dosta razlikuje. Potvrdit će se da rekurzivni algoritam radi sporije od ne-rekurzivnog algoritma s istom apriori ocjenom.

Naš novi studijski problem glasi ovako. Zadano je uzlazno sortirano polje cijelih brojeva. Treba odrediti takozvani **mod** tog polja, dakle član koji se najčešće pojavljuje, te njegovu učestalost. Na primjer, u sljedećem polju je mod 4, a njegova učestalost je 5.

1 1 1| 2 2 2 2 |3 3 |4 4 4 4 4 |5 5 5

Primijetimo da su zbog sortiranosti sve pojave istog broja smještene zajedno u obliku jednog „intervala“. Naš zadatak svodi se na pronalaženje najduljeg takvog intervala.

Prvo rješenje našeg studijskog problema svodi se na izravno prebrajanje. Redom čitamo elemente polja i povećavamo brojač dok god je trenutno pročitani element jednak prethodnom. Kad se pojavi nova vrijednost elementa, resetiramo brojač. Usput pamtimo najveću postignutu vrijednost brojača te odgovarajuću vrijednost elementa. Algoritam je zapisan kao funkcija mode0( ) unutar programa modPolja.

Drugo rješenje našeg studijskog problema dobiva se primjenom rekurzije. Zamislimo da treba naći mod i učestalost u polju A[0], A[1], A[2], … A[*n*-1] duljine *n*. algoritam najprije rekurzivnim pozivom nalazi mod A[*i*] i učestalost *f* u skraćenom polju A[0], A[1], A[2], … A[*n*-2] duljine *n*-1, te zatim gleda da li dodavanje *n*-tog elementa A[*n*-1] može promijeniti rezultat dobiven na prvih *n*-1 elemenata.

	0	1	…	…	…	…	…	…	…	…	…	*n*-2	*n*-1
A									*x*	*x*	*x*	*x*	*y*

Kad n-ti element može promijeniti rezultat?

- Ako je A[*n*-1] ≠ A[*n*-2] , tada zadnji element u polju započinje novi interval, koji zbog svoje „kratkoće“ (duljine 1) ne može biti bolji od intervala koji je bio pronađen u kraćem polju. Dakle za A[*n*-1] ≠ A[*n*-2] nema promjene rezultata.
- Ako je A[*n*-1] == A[*n*-2] , tada se zadnji element u polju priključuje zadnjem intervalu. Taj zadnji interval odredit će novi mod (ili stari mod s povećanom učestalosti) ako i samo ako je njegova duljina jednaka *f*+1. To je opet onda i samo onda ako je A[*n*-1- *f*] == A[*n*-1]. Primijetimo da uvjet A[*n*-1- *f*] == A[*n*-1] ujedno osigurava da je A[*n*-1] == A[*n*-2] , zbog sortiranosti.

Algoritam je zapisan kao rekurzivna funkcija rmode0( ) unutar programa ModPolja. Funkcija neće samu sebe pozvati u slučaju kad je duljina razmatranog polja jednaka 1.

Treće rješenje našeg studijskog problema može se interpretirati kao transformacija drugog rješenja, gdje je rekurzija uspješno zamijenjena iteracijom. Pamtimo trenutno najbolji mod i njegovu učestalost *f*. Čitamo redom elemente polja A[1], A[2], A[3], … . Provjeravamo da li trenutnom elementu A[*i*] prethodi *f* jednakih, to jest da li je A[*i*] == A[*i*-*f*]. Ako je uvjet ispunjen, tada smo našli interval duljine (barem) *f*+1, pa ažuriramo *f* i odgovarajući mod. Algoritam je zapisan kao funkcija rmode1( ) unutar programa ModPolja.

Očigledno je da sva tri algoritma imaju linearnu složenost O(*n*). No njihova stvarna vremena izvršavanja ipak se dosta razlikuju. Da bi eksperimentalno utvrdili ta vremena izvršavanja, program ModPolja radi sljedeće:

- Pomoću generatora slučajnih brojeva generira slučajno polje željene duljine sa željenim rasponom vrijednosti,
- To slučajno polje sortira,
- Svaku od funkcija za traženje moda pozove mnogo puta te mjeri ukupno vrijeme za višestruke pozive iste funkcije,
- Ispisuje rezultate i vremena izvršavanja svakog od algoritama.