Opisi algoritama

Zadatke, test primjere i rješenja pripremili: Frane Kurtović, Alen Rakipović, Gustav Matula, Ivan Katanić i Ante Đerek. Primjeri implementiranih rješenja su dani u priloženim izvornim kodovima koji nužno ne odgovaraju u svim detaljima ovdje opisanim algoritmima.

Zadatak: Kuponi

Predložio: Ante Đerek
Potrebno znanje: naredba `if`, `for` petlja, stringovi

Lako se vidi da je za kupone istog tipa svejedno kojih im redoslijedom Frane primjenjuje. Također, vrijedi da je uvijek bolje upotrijebiti kupone tipa "Y`%`" prije kupona tipa "X`kn`". Dakle, jedan pristup za rješavanje zadatka je sljedeći:

1. Pročitaj ulaz i razluči iznos i tip svakog kupona.
2. Prođi kroz sva tri kupona i primjeni one koji su tipa "Y`%`".
3. Prođi kroz sva tri kupona i primjeni one koji su tipa"X`kn`".

Razlučivanje iznosa i tipa se svodi na jednostavnu manipulaciju stringovima u programskom jeziku Python, a u programskom jeziku C++ jednostavno najprije pročitamo broj pa zatim tip kupona.

Zadatak: Burek

Predložio: Alen Rakipović
Potrebno znanje: `for` petlja, niz brojeva, sort, pohlepni algoritam

Prvo sortirajmo težine, i nazovimo taj niz T. Neka je $S(i)$ najveći indeks za koji vrijedi $T(S(i)) \leq T(i)+10$, tj. najdesniji (najteži) burek kojeg možemo pokupiti ako kupimo burek i.

Promatrajmo što učiniti za $T(0)$, njega moramo ili kupiti ili će biti poklonjen kupnjom nekog drugog bureka. Najdesniji burek koji može pokupiti i $T(0)$ je $S(0)$, stoga je najisplativije kupiti burek $S(0)$. Kada kupimo taj burek, na poklon dobivamo i sve koji su za 10 veći od njega, tj. sve do indeksa $S(S(0))$. Na taj način smo kupili prvi burek i odredili sve bureke koji se dobiju na poklon s njim. Taj postupak treba napraviti i za prvi idući burek, te to ponavljati dok se ne dođe do kraja niza.

Zadatak: Asm

Predložio: Ante Đerek
Potrebno znanje: ad-hoc, nizovi, for petlja, stringovi

Ključni dio rješenja je odabir strukture podataka koja će sadržavati matematičke izraze tj. polinome koji se pojavljuju u registrima. Postoji više različitih pristupa koji su dovoljno efikasni obzirom na relativno malena ograničenja u zadatku. Jedan pristup je da svaki polinom p pamtimo kao niz od 100 brojeva $p_1, \ldots, p_{100}$, gdje je p_k koeficijent uz potenciju x^k. Aritmetičke operacije implementiramo na sljedeći način:

- Polinomi p i q se zbrajaju tako da se zbroje odgovarajući koeficijenti $s_k = p_k + q_k$.
- Polinomi p i q se množe tako da se s dvije `for` petlje razmatraju svi parovi i, j te se koeficijent uz $i + j$ u rezultatu uveća za $p_i p_j$.

Prilikom implementacije aritmetičkih operacija potrebno je voditi pažnju o tome da je moguće da se polinom množi tj. zbraja sam sa sobom.

Za one koji žele više: Za zadani n, konstruiraj program od n naredbi koji računa polinom s *najvećim mogućim* brojem različitih potencija.

Zadatak: Gosti

Predložili: Alen Rakipović, Ivan Katanić
Potrebno znanje: dinamičko programiranje, isprobavanje svih kombinacija, rekurzivno pretraživanje, bitmaske

Fokusirajmo se na jednu vrstu hrane koju treba poslužiti gostima, označimo njeno vrijeme posluživanja s t. Ključna opservacija je da ako odaberemo goste koje ćemo poslužiti i redoslijed u kojem ćemo ih poslužiti tada raspored posluživanja možemo raditi pohlepnim algoritmom.

Prvog gosta poslužujemo čim ranije, u trenutku 1 ili u trenutku a_i gdje je i njegova oznaka. Recimo da je prvi gost završio s jelom u trenutku T, drugog gosta onda poslužujemo opet čim ranije, u trenutku T ili u trenutku a_j (što god je veće) gdje je j njegova oznaka. Naravno, ako je T veći od b_j tada ga nije moguće poslužiti.

Na sličan način poslužujemo i preostale goste. Zadatak dalje rješavamo dinamičkim programiranjem: D_S nam kaže najmanji trenutak takav da je do tog trenutka moguće poslužiti goste iz podskupa S (S je bitmaska kojom prigodno opisujemo neki podskup gostiju). Ako ih nije moguće poslužiti tada je $D_S = \infty$.

Kako izračunati D_S? Fiksirajmo gosta (označimo ga s j) iz podskupa S kojeg ćemo posljednjeg poslužiti, zamislimo sada da provodimo gore opisani pohlepni algoritam. Ono što je potrebno učiniti je na isti način prvo poslužiti sve goste iz S osim j (rekurzivno), i zatim čim ranije poslužiti gosta j. Tako za posljednjeg gosta probamo sve goste koje zaista možemo poslužiti i uzmemo ono rješenje koje daje najmanje vrijeme. Ako sa S_j označimo S bez gosta j, $S_j = S \setminus \{j\}$ onda vrijedi:

$$D_S = \min(\infty, \max(D_{S_j}, a_j) + t;\ \text{za } j \text{ iz } S \text{ i } D_{S_j} \leq b_j)$$

Vrijednosti D_S računamo od manjih skupova S prema većima, tako da su oni koji se koriste u relaciji D_S već izračunati. Za konačno rješenje potrebno je naći podskup S najviše gostiju koji je moguće poslužiti tj. bitmasku S s najviše jedinica za koju vrijedi $D_S < \infty$. Vremenska složenost rješenja je $O(MN2^N)$. Za detalje pogledajte priloženi kod.

Zadatak: Popust

Predložio: Ante Đerek
Potrebno znanje: naredba `if`, `for` petlja, stringovi

Zadatak možemo svesti na zadatak *Kuponi* nakon što s tri `for` petlje isprobamo sve mogućnosti za tri kupona. Međutim još je lakše isprobati sve moguće kombinacije i redosljede te odabrati najpovoljniji:

1. Pročitaj ulaz i razluči iznos i tip svakog kupona.
2. Pomoći tri `for` petlje odaberi prvi, drugi i treći kupon (vodeći računa da su različiti).
3. Primjeni redom odabrane kupone, izračunaj cijenu i usporedi je s do sada najpovoljnijom.

Razlučivanje iznosa i tipa se svodi na jednostavnu manipulaciju stringovima u programskom jeziku Python, a u programskom jeziku C++ jednostavno najprije pročitamo broj pa zatim tip kupona.

Za one koji žele više: Riješite zadatak u kojemu je potrebno odabrati k od n kupona za k i n do 10^7.

Zadatak: Cache

Predložio: Ante Đerek
Potrebno znanje: ad-hoc, polja, `for` petlja

Prvi korak rješenja se sastoji od pronalaska rednog broja bloka, za sve adrese iz ulaza – redni broj bloka je jednak rezultatu cjelobrojnog dijeljenja s veličinom pojedinog bloka (64). Priručnu memoriju mozemo predstaviti s dvije liste - jedna za redne brojeve blokova i druga za vrijeme pristupa pojedinom bloku.

Informaciju da li je pojedini blok priručne memorije zauzet, odnosno koji točno blok radne memorije sadrži pamtimo u jednom polju. Također, u drugom polju za svaki blok pamtimo zadnje vrijeme pristupa odnosno redni broj zadnje adrese koju smo čitali iz bloka. Potrebno iterirati kroz sve blokove (iz prethodnog koraka) te u svakoj iteraciji razmatrati dva slučaja:

1. Blok se nalazi u priručnoj memoriji – ovo je pogodak i ispišemo redni broj bloka iz priručne memorije te osvježimo vrijeme pristupa za taj blok.
2. Blok se ne nalazi u priručnoj memoriji – ovo je promašaj i potrebno je dodati blok u priručnu memoriju (sto rezultira s dva dodatna slučaja).
   (a) U priručnoj memoriji ima mjesta za dodatni blok – blok se doda na prvo slobodno mjesto, te se ispiše redni broj bloka iz priručne memorije, te se osvježi vrijeme pristupa za taj blok u priručnoj memoriji.
   (b) Nema mjesta za dodatni blok – iz memorije je potrebno izbaciti blok iz kojeg se čitalo najkasnije u prošlosti pretraživanjem po listi pristupa, na njegovo mjesto se doda novi blok te osvježi vrijeme pristupa za taj blok

Za one koji žele više: U zadatku je opisan jedan od algoritama za održavanje stanja priručne memorije (engl. Least recently used) Proučite ostale algoritme (`en.wikipedia.org/wiki/Cache_algorithms`) te pokušajte implementirati neke od njih.

Zadatak: Poli

Predložio: Ante Đerek
Potrebno znanje: ad-hoc, binarni zapis broja, nizovi, for petlja, stringovi

Prvi dio rješenja je čitanje ulaza i rastavljanje zapisa polinoma, što se može napraviti na nekoliko načina. Primjerice, možemo najprije rastaviti zapis na pribrojnike (npr. u Python-u koristimo funkciju `split`, dok u C++ pomoću jedne `for` petlje tražimo sljedeći znak "`+`" i koristimo funkciju `substr`). Za svaki pribrojnik određujemo vrijednosti a i k tako da rastavljamo string na dva dijela – prije i poslje malog slova "`x`". Ukoliko string nije prazan, pretvaramo ga u broj (naravno najprije uklonimo znak potenciranja iz drugog stringa).

Nakon analize zapisa polinom je opisan nizom $p_1, \ldots, p_{10}$, gdje je p_k koeficijent uz potenciju x^k.

U drugom dijelu rješenja konstruiramo program koji računa zadani polinom. Za polinome sa malenim koeficijentima mogući su različiti ad-hoc pristupi. Jedan način je sljedeći:

- Dodamo naredbu `add d1 d0`, u ovom trenutku `d1` sadrži x.
- Unutar `for` petlje dodajemo naredbu `mul d1 d0`, generirajući redom sve potencije x^k
  - Kada se u `d1` nalazi x^k, dodamo p_k naredbi `add d9 d1`, tako da se `d9` uveća za $p_k x^k$.
- Dodamo naredbu `mul d0 d8` tako da `d0` postavimo na nulu.
- Konačno rješenje preselimo iz `d9` u `d0` naredbom `add d0 d9`.

Za velike koeficijente ovaj pristup daje previše naredbi – primjerice za koeficijent 1000 trebati će nam 1000 `add` naredbi u koraku 2. (a) te je potrebno generirati efikasnije programe.

Ključna ideja je da se za svaki koeficijent razmatra njegov binarni zapis te se pomoću njega napravi program koji ga izgrađuje sa relativno malo naredbi. Pretpostavimo da je $(b_n \ldots b_0)_2$ binarni zapis prirodnog broja a odnosno da vrijedi $a = b_n 2^n + \ldots + b_0 2^0$. Također, pretpostavimo da se u registru `d3` nalazi x^k. Pribrojnik ax^k možemo generirati na sljedeći način:

- Postavimo `d4` na nulu naredbom `mul d4 d8`.

- Unutar `for` petlje dodajemo `add d3 d3`, generirajući redom polinome oblika $2^i x^k$.
  - Kada se u `d3` nalazi $2^i x^k$ onda se razmatra binarna znamenka b_i te ako je ona jednaka 1 dodaje se naredba `add d4 d3`
- Pribrojimo ax^k rješenju naredbom `add d9 d4`

Kombinirajući ova dva postupka dobivamo rješenje koje generira oko 250 naredbi u najgorem slučaju.

Za one koji žele više: Riješite zadatak ako se polinom može sastojati od najviše 10 pribrojnika koji mogu biti stupnja do 1000.

Zadatak: Sažetko

Predložio: Frane Kurtović
Potrebno znanje: stabla, stingovi, parsiranje, rekurzija, dinamičko programiranje

Pogledajmo prvo kako riješiti zadatak kad se traži složenost cijelog sažetka ($a = 1$ i $b = n$). Definicija sažetaka je rekurzivna pa je prirodno razmišljati o rekurzivnom rješenju.

Označimo s $L(s)$ najljeviji, a s $R(s)$ najdesniji znak u uzorku definiranom sažetkom s.

- Ako je s uzorak, $L(s)$ je najljeviji znak uzorka.
- Ako su s_1 i s_2 sažeci, $L(s_1 s_2) = L(s_1)$.
- Ako je s sažetak, $L(p(s)) = L(s)$.

Time smo potpuno definirali L, dok se R se definira analogno.

Označimo složenost sažetka s sa $C(s)$. Definirajmo $|s| = C(s) + 1$ ($|s|$ sada označava broj uzastopnih grupa istih znakova u uzorku definiranom sažetkom s).

- Ako je s uzorak, $|s|$ računamo direktno (jednostavno prebrojimo grupe).
- Ako su s_1 i s_2 sažeci, $|s_1 s_2| = |s_1| + |s_2| - [R(s_1) == L(s_2)]$. Ovjde koristimo notaciju: $[e]$ je 1 ako je e istina, 0 ako je laž.
- Ako je s sažetak, $|p(s)| = p|s| - (p-1)[L(s) == R(s)]$.

Uz ove relacije problem se svodi na parsiranje ulaznog stringa. Za detalje parsiranja pogledajte kod službenog rješenja.

Problem je nešto teži ako imamo zadani poduzorak čiju složenost moramo izračunati. Označimo s $N(s)$ ukupnu duljinu uzorka definiranog sažetkom s. $N(s)$ možemo definirati rekurzivno slično već definiranim funkcijama.

Sada možemo definirati općenitiju funkciju $|s|_l^r$ koja nam govori broj uzastopnih grupa u poduzorku od znaka l do r uzorka definiranog sažetkom s.

- Ako je s uzorak, $|s|_l^r$ računamo direktno.
- Ako su s_1 i s_2 sažeci, za $|s_1 s_2|_l^r$ imamo nekoliko slučajeva:
  1. Ako $r \leq N(s_1)$, onda je $|s_1 s_2|_l^r = |s_1|_l^r$ (interval je u potpunosti u s_1).
  2. Ako $l > N(s_1)$, onda je $|s_1 s_2|_l^r = |s_2|_{l-N(s_1)}^{r-N(s_1)}$ (interval je u potpunosti u s_2).
  3. Inače je interval razlomljen između s_1 i s_2.

$$|s_1 s_2|_l^r = |s_1|_l^{N(s_1)} + |s_2|_1^{r-N(s_1)} - [R(s_1) == L(s_2)].$$

- Ako je s sažetak, $|p(s)|_l^r$ računamo na sljedeći način: U intervalu $[l, r]$ imat ćemo $q = \lfloor (r-l+1)/N(s) \rfloor$ cjelovitih pojavljivanja s, sufiks od s duljine $S = (N(p(s)) - l + 1) \bmod N(s)$ lijevo, te prefiks od s duljine $P = r \bmod N(s)$ desno. Pri izračunu pazimo da za jednake rubne znakove smanjimo rješenje (za detalje vidi izvorni kod rješenja). Alternativno, budući da su ograničenja u zadatku mala, $|p(s)|_l^r$ možemo izračunati petljom koja ide kroz svih p primjeraka i kombinira ih.
- Dodatno, $|s|_l^r$ je 0 ako je $l > r$.

Pri implementaciji korisno je prvo parsirati ulazni string i u memoriji ga predstaviti kao stablo. Definirane funkcije se tada lako računaju na stablu dinamičkim programiranjem.