7. tekmovanje ACM v znanju računalništva

Predstavitev nalog in rešitev

1.1 Prepletene besede

- Naloga: predelaj niz *s* v nov niz *t* tako, da premakneš velike črke na začetek, male na konec, njihov medsebojni vrstni red pa se ne spremeni
  - `PbREPeseLEdTiEloNO` $\rightarrow$ `PREPLETENObesedilo`
- Rešitev:
  - Recimo, da imamo $s = s[0]\ s[1] \ldots s[n-1]$
  - Naredimo dva prehoda po njem
    - V prvem kopiramo v *t* velike črke, v drugem male
  - *j* = 0;
    for (*i* = 0; *i* < *n*; *i*++) **if** (*s*[*i*] je velika) *t*[*j*++] = s[*i*];
    for (*i* = 0; *i* < *n*; *i*++) **if** (*s*[*i*] je mala) *t*[*j*++] = s[*i*];

1.2 Manjkajoča števila

- Naloga:
  - Iz datoteke beri naraščajoče zaporedje naravnih števil
  - Sproti izpiši tista, ki manjkajo
  - Primer: 4, 5, 6, 9, 11 $\rightarrow$ 4, 5, 6, (7), (8), 9, (10), 11
- Rešitev:
  - V neki spremenljivki si zapomnimo nazadnje izpisano število, recimo *z*
  - Ko preberemo naslednje število, recimo *n*,
    najprej izpišemo tista med *z* in *n*
  - *z* = *z* + 1;
    while (*z* < *n*) {
    izpiši *z* v oklepajih; *z* = *z* + 1; }
    izpiši *n*;

1.3 Kazenski stavek

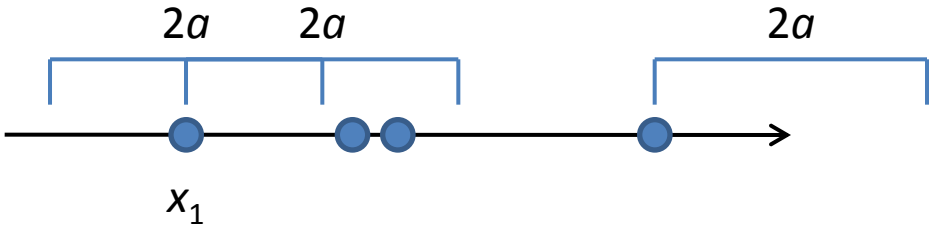
- Dan je dolg niz *s*, iščemo najkrajši tak *t*, da je
  s = *t* + " " + *t* + " " + *t* + … + *t* + " " + *t*
- Primer:
  s = "ab ab ab" $\rightarrow$ t = "ab"
  s = "ab ab ab " $\rightarrow$ t = s = "ab ab ab "
- Naj bo *n* dolžina niza *s*.
  Preizkusimo vse možne dolžine *t*-ja od 1 do *n*:
  - Naj bo *d* trenutna dolžina *t*-ja. Preverimo:
  - Ali je *n* + 1 večkratnik števila *d* + 1?
  - Ali je *s*[*d*] presledek?
  - Ali je *s*[*i*] = *s*[i – *d* – 1] za vse *i* > *d*?
  - Če vse to drži, smo našli pravi *d* (in *t* = *s*[0] … *s*[*d* – 1])

1.4 Mase

- Naloga:
  - Imamo tehtnico z napako $\pm a$
    - Torej, če tehtamo predmet z maso m, dobimo v resnici neko meritev z intervala $[m - a, m + a]$
  - Stehtali smo več predmetov in dobili zaporedje meritev $x_1, x_2, \ldots, x_n$ (v naraščajočem vrstnem redu)
  - Koliko najmanj različnih mas imajo ti predmeti?
- Primer: $a = 5$
  - Meritve 15, 24, 26 $\rightarrow$ vsaj dve različni masi (npr. 15 in 25)
  - Meritve 15, 24, 25 $\rightarrow$ vsaj ena različna masa (20)

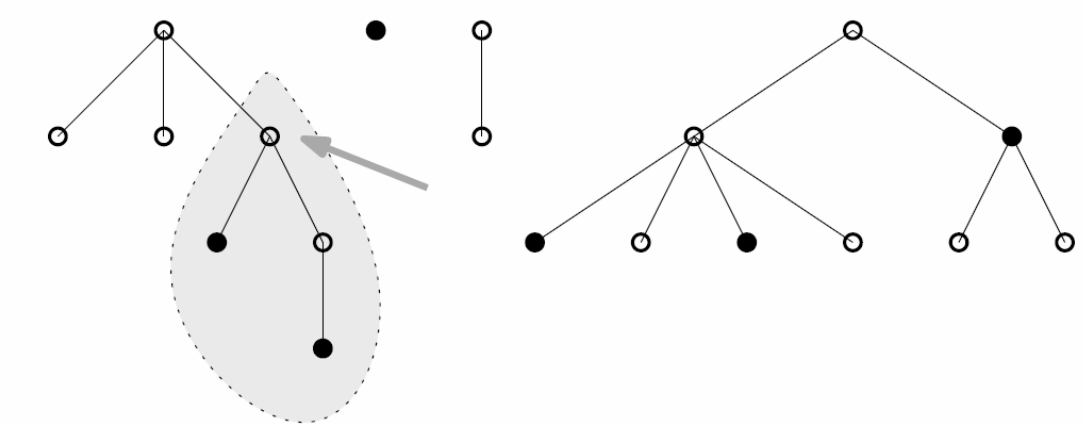
1.4 Mase

- Rešitev: požrešni algoritem
  - Če ima nek izdelek maso m, nam "pokrije" vse meritve, ki ležijo na intervalu $[m - a, m + a]$
  - Izbrati moramo čim manj takih m-jev, da bodo njihovi intervali skupaj pokrili vse meritve
  - Da pokrijemo najmanjšo meritev, x_1, potrebujemo neko meritev m z območja $[x_1 - a, x_1 + a]$
    - Manjšega m od $x_1 + a$ nima smisla jemati, ker s tem ne bomo pokrili ničesar, česar ne bi pokrili tudi z $m = x_1 + a$
  - Pobrišimo meritve, ki jih pokrije ta izdelek (torej vse z maso $\leq x_1 + 2a$), in nadaljujmo pri najmanjši od preostalih meritev



1.5 V Afganistan!

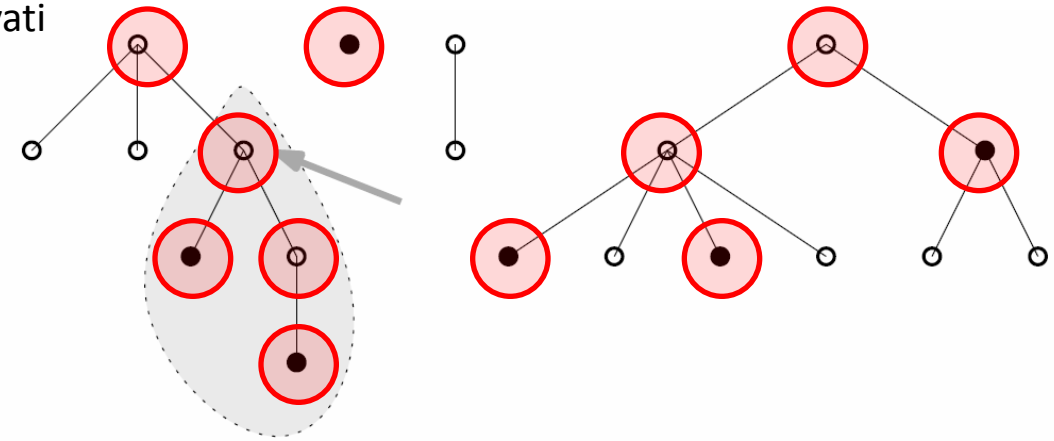
- Imamo množico vojakov
  - Urejeni so v hierarhije
    (vsak ima največ enega neposredno nadrejenega)
  - Nekatere hočemo poslati v Afganistan (črni krožci na sliki)
  - Če pošljemo nekomu ukaz, naj gre v Afganistan,
    odpelje s seboj še celo svoje poddrevo
  - Koliko najmanj ukazov moramo izdati?
    - (Nič ni narobe, če so ukazi taki, da gredo poleg črnih krožcev v Afganistan tudi nekateri beli krožci.)



1.5 V Afganistan!

- Če smo *x*-u poslali ukaz in ima ta *x* nekega neposredno nadrejenega, recimo *y*,
  - bi lahko ta ukaz namesto *x*-u poslali *y*-u, pa bi šli še vedno v Afganistan vsi tisti vojaki kot prej, pa mogoče še kakšen dodatni (npr. *y* sam)
  - Torej ni nobene koristi od tega, da pošljemo ukaz vojaku, ki ima nadrejenega (torej ni sam v vrhu svoje hierarhije)
  - Moramo torej le poiskati korene dreves, ki vsebujejo kakšno črno točko
- Imejmo tabelo *b*, ki za vsakega vojaka pove, ali je treba njega ali kakšnega (posredno ali neposredno) podrejenega poslati v Afganistan
  - Na začetku postavimo *b*[*u*] = **false** za vse točke *u*;
  - Za vsako črno točko *u*:

    v = *u*
    while (*v* ≥ 0 **and not** *b*[*v*]) {
      b[*v*] = **true**; *v* = *Nadrejeni*; }
  - Tako ni treba iste točke obiskovati po večkrat
  - Na koncu pošljemo ukaz vsem takim *u*, ki imajo *b*[*u*] == **true** in nimajo nadrejenega.



2.1 Ovce

- Naloga:
  - Imamo n ovac in bi jih radi postrigli v času t.
  - Imamo m strižcev, pri čemer i-ti med njimi za striženje ene ovce porabi t_i časa in prejme p_i plačila.
  - Cena elektrike za striženje je c denarja na enoto časa.
  - Kako razporediti ovce med strižce, da bo najceneje?
- Rešitev: požrešni algoritem
  - Strošek za nas, če nam eno ovco postriže i-ti strižec, je $q_i = p_i + c\ t_i$
  - Če je $q_i < q_j$ in premaknemo eno ovco od strižca j k strižcu i, prihranimo nekaj denarja (namreč $q_j - q_i$)
  - Torej uredimo strižce po naraščajočem q_i in dajmo vsakemu toliko ovac, kolikor jih le lahko obdela

    $s = 0$; (* *skupna cena* *)

    pregleduj strižce po naraščajočem q_i :

    naj bo i trenutni strižec;

    $n_i = \min\{ n, \lfloor t / t_i \rfloor \}$ (* *toliko ovac bo postrigel* *)

    $s = s + n_i\ q_i$; $n = n - n_i$;

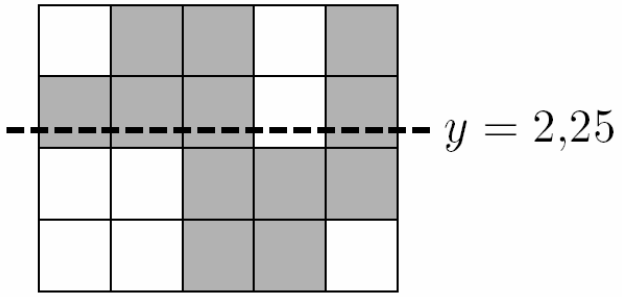
2.2 Spričevala

- Dan je seznam spričeval – trojic ⟨*leto*, *šola*, *ravnatelj*⟩ (že urejen po letih)
  - Poišči primere, kjer:
    - Je za isto šolo v istem letu navedenih več ravnateljev
    - Je isti ravnatelj v istem letu naveden pri več šolah
- Imejmo tabelo *sola*[*r*], ki pove, na kateri šoli smo videli ravnatelja *r*
  - Na začetku leta postavimo vse *sola*[*r*] na –1
  - Za vsako trojico (*leto*, *s*, *r*):
    if (*sola*[*r*] == –1) *sola*[*r*] = *s*;
    else if (*sola*[*r*] != *s*) izpiši opozorilo;
  - Tako smo poskrbeli za opozorila, če je isti ravnatelj naveden v istem letu pri več šolah. Drugi del naloge gre na enak način, le šole in ravnatelji so v zamenjanih vlogah.
  - Gornji postopek ima še dve slabosti:
    - Za isto leto in ravnatelja lahko izpiše po več opozoril (kar naloga prepoveduje)
    - Za vsako leto gre v celoti po tabeli *sola*[*r*], da jo inicializira (neučinkovitost)
  - Vpeljimo še eno tabelo, *rLeto*[*r*], ki pove, v katerem letu smo nazadnje videli ravnatelja *r*
    - Zdaj torej na začetku leta ni treba pospravljati tabele *sola*[*r*], ker bomo že iz tabele *rLeto* videli, katera polja se nanašajo na letošnje leto
    - Vrednost *sola*[*r*] == –1 pa lahko zdaj uporabimo za primere, ko smo opozorilo že izpisali
  - Na začetku seznama postavimo vse *rLeto*[*r*] na –1
  - Za vsako trojico (leto, *s*, *r*):
    if (*rLeto*[*r*] != *leto*) *rLeto*[*r*] = *leto*, *sola*[*r*] = *r*;
    else if (*sola*[*r*] != –1 **and** *sola*[*r*] != *s*) { izpiši opozorilo; *sola*[*r*] = –1; }

2.3 Razpolovišče lika

- Dan je lik, ki ga tvorijo črna polja na črno-beli karirasti mreži
  - Potegni vodoravno premico tako,
    da ga po površini razdeli ravno na polovico
- Rešitev:
  - Z enim prehodom po mreži preštejemo črna polja
    (= ploščina lika, recimo ji *p*)
  - Nato gremo še enkrat po mreži, po vrsticah,
    in gledamo, v kateri vrstici površina doseže/preseže *p*/2

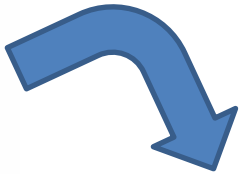
d = 0
for (*y* = 0; *y* < *VisinaMreze*; *y*++) :
  naj bo *v* število črnih polj v vrstici *y*
  if (*d* + *v* ≥ *p*/2) **return** *y* + (*p*/2 – *d*) / *v*
  d = *d* + *v*;



2.4 Strukturirani podatki

- Imamo besedilo, opremljeno z značkami, podobno kot HTML ali XML, le sintaksa je preprostejša
  - Vsaka značka ali kos besedila je v svoji vrstici
  - Značke so pravilno gnezdene
  - Izpiši kose besedila, pred vsakim pa spisek trenutno odprtih značk

```
+ena
  +dve
    alfa
  -dve
  +tri
    beta
    gama
    +stiri
      delta
    -stiri
    epsilon
  -tri
  zeta
  +tri
  -tri
-ena
eta
```



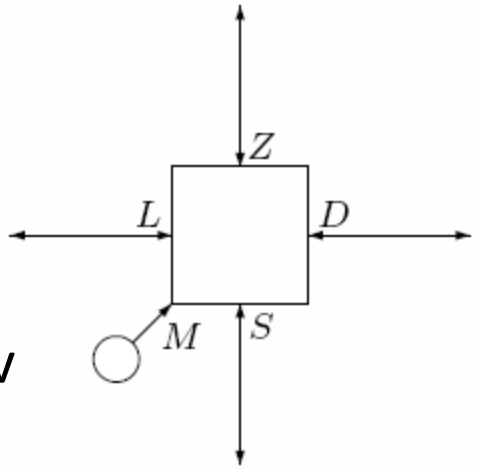
```
ena.dve alfa
ena.tri beta
ena.tri gama
ena.tri.stiri delta
ena.tri epsilon
ena zeta
 eta
```

2.4 Strukturirani podatki

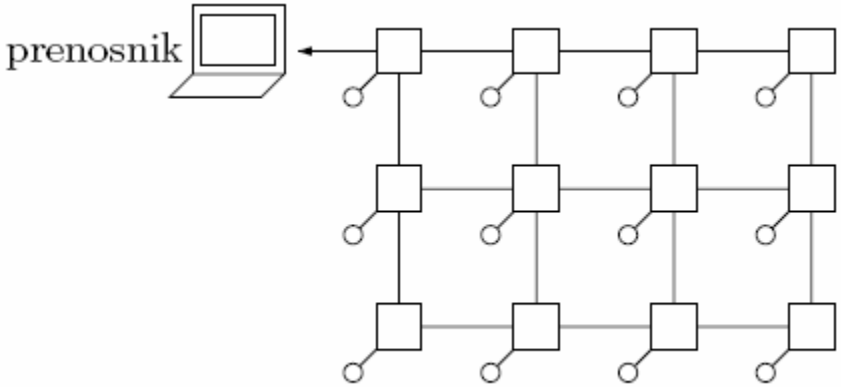
- Rešitev: vzdržujmo nek seznam (sklad) trenutno odprtih značk
  - Berimo vhodno datoteko po vrsticah
  - Če se vrstica začne na "+", je začetna značka in jo dodajmo na konec seznama
  - Če se začne na "–", je končna značka in jo pobrišimo s konca seznama
  - Sicer je to navadna vrstica z besedilom:
    - Zapeljimo se po celem seznamu in izpisujmo imena značk, med njimi pike,
      na koncu pa še trenutno vrstico

```
+ena
  +dve
    alfa
  -dve
  +tri
    beta
    gama
    +stiri
      delta
    -stiri
    epsilon
  -tri
  zeta
  +tri
  -tri
-ena
eta
```

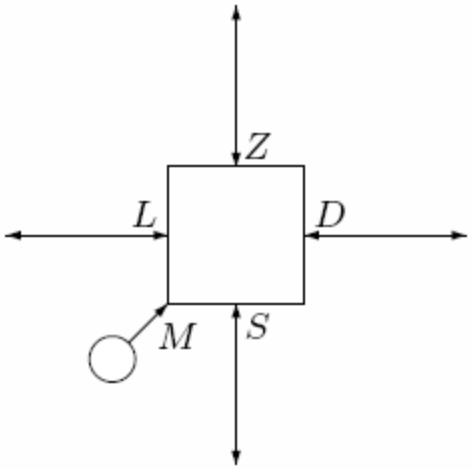
2.5 Največji pretok



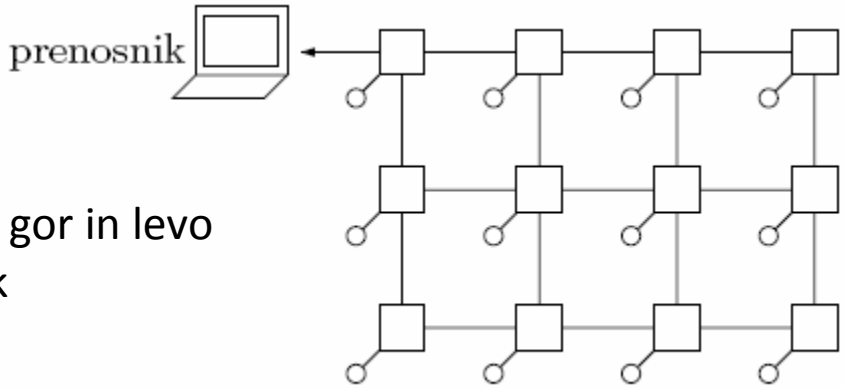
- Imamo mrežo enostavnih merilnih računalnikov
  - Vsak ima 1 kanal, ki ga povezuje z merilnikom (*M*), in 4 kanale do sosedov v mreži (*L*, *D*, *Z*, *S*)
  - Tako so povezani v pravokotno mrežo
  - Na zgornji levi računalnik se priklopimo s prenosnikom
  - Radi bi odčitali maksimum vseh dosedanjih meritev v celotni mreži
  - Napiši program, ki se bo izvajal na vsakem računalniku in poskrbel za to



2.5 Največji pretok



- Rešitev:
  - V neki spremenljivki *m* hranimo največjo nam znano meritev doslej
  - Program teče v zanki:
    - V vsakem koraku preberemo vrednosti iz *M*, *D*, *S* in si zapomnimo največjo:
      m = max{ *m*, *Beri*(*M*), *Beri*(*D*), *Beri*(*S*) }
    - Nato *m* pošljemo zgornjemu in levemu sosedu:
      Pisi(*L*, *m*); *Pisi*(*Z*, *m*)
    - Tako vsak računalnik pozna maksimum po vseh, ki ležijo spodaj in desno od njega
    - Največje vrednosti se sčasoma širijo gor in levo po mreži in tako dosežejo računalnik v zgornjem levem kotu

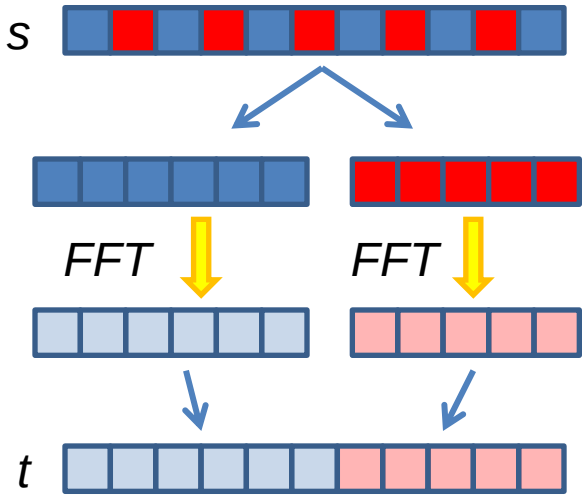


3.1 de-FFT permutacija

- Dana je funkcija *FFT* za šifriranje nizov. Definirana je rekurzivno:
  - Če imamo niz $s = s_1\ s_2\ s_3\ \ldots$, definirajmo
    $FFT(s) = FFT(s_1\ s_3\ s_5\ \ldots) + FFT(s_2\ s_4\ s_6\ \ldots)$
  - Pri nizu dolžine 1 pa $FFT(s_1) = s_1$.
  - Naloga: računaj obratno funkcijo (za dešifriranje)
  - Primer:

    $FFT(\mathrm{abcde}) = FFT(\mathrm{ace}) + FFT(\mathrm{bd})$
    $= FFT(\mathrm{ae}) + FFT(\mathrm{c}) + FFT(\mathrm{b}) + FFT(\mathrm{d})$
    $= FFT(\mathrm{a}) + FFT(\mathrm{e}) + \mathrm{c} + \mathrm{b} + \mathrm{d}$
    $= \mathrm{a} + \mathrm{e} + \mathrm{c} + \mathrm{b} + \mathrm{d} = \mathrm{aecbd}$

3.1 de-FFT permutacija



- Tudi dešifriramo lahko rekurzivno
  - Naj bo s prvotni niz, t pa šifrirani niz: $t = FFT(s)$
    - Očitno sta enako dolga, recimo 11 znakov
  - Spomnimo se, da je FFT najprej razbil

    $s = s_1\ s_2\ \ldots\ s_{11}$

    na dva pol krajša niza

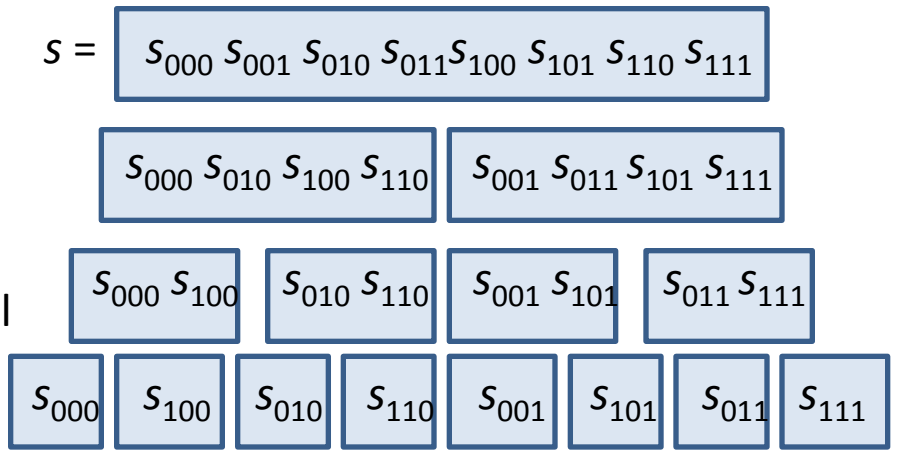
    $s_1\ s_3\ \ldots\ s_9\ s_{11}$ in $s_2\ s_4\ \ldots\ s_8\ s_{10}$,

    šifriral vsakega posebej in šifri staknil:

    $t = FFT(s) = FFT(s_1\ s_3\ \ldots\ s_9\ s_{11}) + \mathrm{FFT}(s_2\ s_4\ \ldots\ s_8\ s_{10})$
    - Prvi del je dolg 6 znakov, drugi pa 5
    - Torej je prvih 6 znakov t-ja ravno šifra niza $FFT(s_1\ s_3\ \ldots\ s_9\ s_{11})$, preostalih 5 znakov t-ja pa je šifra niza $FFT(s_2\ s_4\ \ldots\ s_8\ s_{10})$
    - Torej lahko t razsekamo na ta dva kosa, vsakega posebej dešifriramo (rekurzivni klic) in tako dobimo $s_1\ s_3\ \ldots\ s_9\ s_{11}$ ter $s_2\ s_4\ \ldots\ s_8\ s_{10}$, iz njiju pa potem s
  - V splošnem, če imamo niz dolžine k,
    ga moramo razdeliti na prvih $\lceil k/2 \rceil$ znakov in preostalih $\lfloor k/2 \rfloor$ znakov.

3.1 de-FFT permutacija

- Lahko pa nalogo rešimo tudi drugače
- Recimo, da je s dolg $n = 2^b$ znakov
  - Primer: $b = 3$, $n = 8$
  - Štejmo indekse od 0 naprej: $s = s_0\ s_1 \ldots s_7$
  - Oz. v dvojiškem zapisu: $s = s_{000}\ s_{001}\ s_{010} \ldots s_{111}$
  - Če zapišemo *i* kot *b*-bitno število v dvojiškem zapisu in preberemo bite z desne proti levi, dobimo ravno indeks, na katerem se s_i nahaja v $FFT(s)$!
  - Slika kaže primer za $n = 8$, z indukcijo pa se da dokazati, da velja za vse potence števila 2
  - Torej lahko za vsak znak iz $FFT(s)$ kar **izračunamo**, s katerega indeksa v nizu *s* je prišel



3.1 de-FFT permutacija

- Kaj pa, če *n* (dolžina *s*-ja) ni potenca števila 2?
  - Naj bo 2^b prva potenca števila 2, ki je večja od *n*
  - Podaljšajmo v mislih *s* do te dolžine z nekimi posebnimi znaki, recimo #; tako podaljšanemu nizu recimo *s'*
  - Izračunajmo *FFT* tako podaljšanega niza in nato iz nje pobrišimo znake #
  - Izkaže se, da dobimo ravno *FFT* prvotnega niza
  - Primer: *s* = abcde $\rightarrow$ podaljšamo v *s'* = abcde###
    $\rightarrow$ *FFT*(abcde###) = aec#b#d#
    $\rightarrow$ pobrišemo znake # in dobimo aecbd, kar je res enako *FFT*(abcde)
  - Postopek je zdaj lahko tak: če imamo *t* = *FFT*(*s*) in bi radi dobili *s*,
    for (*i* = 0, *j* = 0; *i* < 2^b; i++)
    zapiši *i* kot *b*-bitno dvojiško število in obrni vrstni red bitov;
    dobljenemu indeksu recimo *k*
    vemo torej, da je *i*-ti znak v *FFT*(*s'*) nastal iz *k*-tega znaka niza *s'*
    if (*k* $\geq$ *n*) **continue**; (* *ta znak je # in ga v t sploh ni* *)
    s[*k*] = *t*[*j*++] (* *sicer pa je i-ti znak v FFT(s') ravno naslednji znak t-ja,*
    kajti t je enak nizu, ki ga dobimo, če iz FFT(s') pobrišemo vse # *)

3.2 Potovanje

- Imamo *n* črpalk na ravni cesti od zahoda proti vzhodu
  - *i*-ta črpalka ima položaj x_i in zalogo goriva g_i
  - Za vsako črpalko ugotovi, kako daleč lahko pridemo, če začnemo pri njej s praznim tankom (kapaciteta je neomejena)
- Preprosta rešitev:
  - Pri vsaki možni začetni črpalki z simuliramo celotno potovanje:

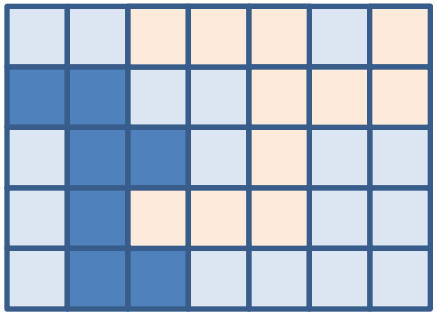
    $i = z$; $g = g_i$
    while ($i < n$) {
    if ($x_{i+1} - x_i > g$) **break;** (* *ne moremo doseči naslednje črpalke* *)
    $g = g - (x_{i+1} - x_i) + g_{i+1}$; $i = i + 1$; }
    $potovanje_z = x_i - x_z + g$
  - Lahko ima časovno zahtevnost $O(n^2)$
  - Gre pa tudi hitreje

3.2 Potovanje

- Izboljšava:
  - Ko računamo potovanje z začetkom pri z, bi bilo koristno že kaj vedeti o potovanjih, ki se začnejo pri kasnejših črpalkah
  - Recimo, da če začnemo pri z s praznim tankom, lahko dosežemo črpalko p_z (in imamo še h_z goriva, ko jo dosežemo), ne pa tudi kakšne od kasnejših črpalk
  - To lahko računamo po padajočih z-jih:

    $i = z$; $g = 0$; $p_z = z$
    while ($i < n$) {
    if ($p_i > i$) { $g = g + g_i$; $i = p_i$; **continue**; }
    if ($x_{i+1} - x_i > g + g_i$) **break;**
    $g = g - (x_{i+1} - x_i) + g_{i+1}$; $i = i + 1$; }
    $p_z = i$; $h_z = g$; $potovanje_z = x_i - x_z + g_i + g$
  - Zakaj tu ne more priti do časovne zahtevnosti $O(n^2)$?
    - Pot od z do p_z prehodimo le enkrat,
      v bodoče bomo tam naredili le neposreden skok $z \rightarrow p_z$
    - Lahko si predstavljamo sklad, naša zanka je pobrisala nekaj vrhnjih elementov (do p_z) in nato dodala na vrh z
    - Vsak element le enkrat dodamo in enkrat pobrišemo $\rightarrow O(n)$

3.3 Leteči pujsi



- Karirasta mreža $w \times h$, črna polja so opeke, bela so prosta
  - Opeka je *stabilna*, če leži v najbolj spodnji vrstici ali pa se z levim, desnim ali spodnjim robom dotika kakšne druge stabilne opeke
  - Naloga: za vsako opeko ugotovi, koliko opek postane nestabilnih, če jo izbrišemo
- Preprosta rešitev:
  - Za vsako opeko pregledamo celo mrežo od te višine navzgor in na novo računamo stabilnost opek v tabeli $s[x, y]$

```
for (y = y0; y >= 0; y--) {
    for (x = 0; x < w; x++)
        s[x, y] = opeka[x, y] and (s[x – 1, y] or s[x, y + 1]);
    for (x = w – 1; x >= 0; x––)
        s[x, y] |= s[x + 1, y] and opeka[x, y]; }
```

  - Slabost: $O(wh)$ dela pri vsaki opeki (x_0, y_0), skupaj $O(w^2 h^2)$

3.3 Leteči pujsi

- Naj bo *prečka* strnjena skupina opek v isti vrstici, ki na levi in desni meji na prazna polja ali rob mreže
  - Prečka je v celoti stabilna ali v celoti nestabilna
  - Prečka dobi svojo stabilnost le od spodaj, z leve ali desne je ne more

- Ko pobrišemo opeko (x_0, y_0):
  - Njena prečka razpade na dva dela, za vsakega pogledamo, ali je od spodaj še podprt
  - Vse druge prečke v tej vrstici (in vse prečke v nižjih vrsticah) so še vedno stabilne
  - Tako dobimo nek interval nestabilnosti v vrstici y_0
  - Ko poznamo interval nestabilnosti $x_1..x_2$ v vrstici $y + 1$, si oglejmo vrstico y:
    - Vse prečke, ki ležijo v celoti znotraj $x_1..x_2$, so zdaj nestabilne
    - Na levem koncu $x_1..x_2$ imamo lahko prečko, ki delno že sega levo od x_1; če je ta od spodaj podprta še kje levo od x_1, ostane stabilna, sicer pa ne
    - Podobno je na desni
    - Tako imamo nov interval nestabilnosti v vrstici y

  - Koristno je, če si v nekih tabelah izračunamo:
    - Za vsako opeko: levo in desno krajišče njene prečke
    - Za vsako prosto celico: x najbližje opeke levo in desno od nje
    - Za vsako celico: koliko je opek levo od nje v njeni vrstici
    - Potem bomo novi interval nestabilnosti vsakič izračunali v času $O(1)$, tudi opeke na njem bomo prešteli v času $O(1)$
    - To je potem $O(h)$ dela za vsak (x_0, y_0), celoten algoritem ima zahtevnost $O(w\,h^2)$

3.4 Nakup parcele

- Imamo karirasto mrežo $w \times h$,
  v vsakem polju je število
- Vanjo na vse možne načine postavimo pravokotno okno velikosti $n \times m$
- Pri vsakem položaju vzamemo najmanjše število v oknu
- Te minimume seštejemo (po vseh položajih okna)

8	5	5	8	1
6	4	8	3	8
8	2	8	7	7

4 3 1
2 2 3

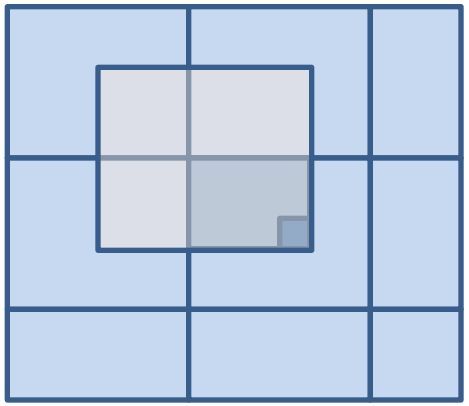
4 + 3 + 1 + 2 + 2 + 3 = 15

3.4 Nakup parcele

- Razdelimo našo mrežo na “celice” velikosti $(n - 1) \times (m - 1)$
- Za vsako polje izračunajmo minimum po vseh tistih, ki so v njegovi celici zgoraj in levo od njega

  $$ZL[x, y] = \min\{ ZL[x - 1, y], ZL[x, y - 1], a[x, y] \}$$

- Podobno tudi za zgoraj desno, spodaj levo, spodaj desno
- Naše okno $n \times m$ pade v točno štiri celice, njegovi preseki z njimi so točno taka pravokotna območja, s kakršnimi smo se ukvarjali zgoraj
  - Zdaj moramo le vzeti minimum po štirih takih območjih
- Časovna zahtevnost: $O(wh)$ za pripravo tabel ZL itd., nato $O(1)$ za vsak položaj okna, skupaj torej $O(wh)$



3.5 Rotacija

- Imamo niz (pravzaprav cikel) števk $s = s_1\, s_2 \ldots s_n$
- Radi bi ga ciklično zamaknili tako, da bi dobil čim večjo vrednost
- Primer: 425747 $\rightarrow$ 747425
- Da se ne bomo preveč mučili s cikli, lahko nalogo zapišemo tudi takole: iščemo leksikografsko največji podniz dolžine n v nizu $t := s_1\, s_2 \ldots s_n\, s_1\, s_2 \ldots s_{n-1}$

3.5 Rotacija

- Leksikografsko največji podniz dolžine n se vsekakor začne z leksikografsko največjim podnizom dolžine $n - 1$ in tako naprej proti krajšim nizom
  - Torej poiščimo najprej leksikografsko največje podnize krajših dolžin in bomo nato pogledali, katere od njih se da podaljšati v leksikografsko največji podniz malo večje dolžine
- Postopek:

  $d = 1$; $c = \max \{ t_i : i \in 1..n \}$; $L = \{ i : t_i = c \}$
  while ($d < n$ **or** $|L| > 1$):
  (* *Invarianta: za vsak $i \in L$ velja, da je $t_i \ldots t_{i+d-1}$ leksikografsko največji podniz dolžine d v t.* *)
  $c = \max \{ t_{i+d} : i \in n \}$; $L' = \{ i \in L : t_{i+d} = c \}$
  $L = L'$; $d = d + 1$;

- Slabost tega postopka: lahko se zgodi, da gre zanka vse do $d = n$ in da je v L ves čas kar $O(n)$ indeksov (npr. če so v t vse številke enake) $\rightarrow$ porabimo $O(n^2)$ časa

3.5 Rotacija

- Izboljšava:
  - Naj bo *x* leksikografsko največji podniz dolžine *d*
  - V *L* imamo indekse, pri katerih se začnejo pojavitve *x*-a v *t*
  - Recimo, da je to najmanjši tak *d*, pri katerem se zgodi, da se dve ali več pojavitev *x*-a “sprime” skupaj
  - V taki skupini sprijetih pojavitev lahko vse razen prve zavržemo!
  - Zaradi te spremembe je pri *d* lahko v seznamu *L* največ $O(n/d)$ indeksov, če to seštejemo po vseh *d*, dobimo $O(n \log n)$, takšna je zdaj tudi časovna zahtevnost