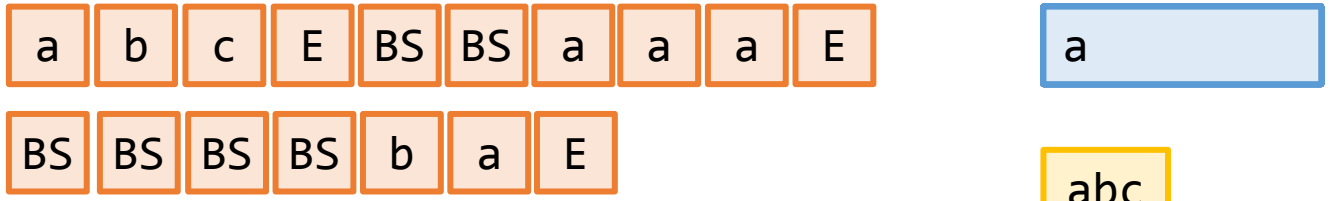
RTK 2016 – Naloge in rešitve

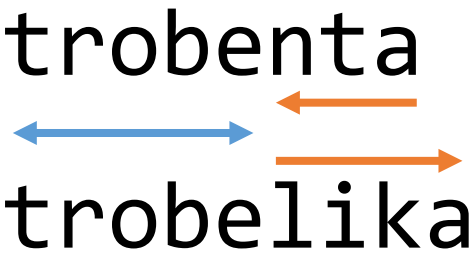
Janez Brank

1.1 Tipkanje



- Natipkati moramo dano zaporedje besed
  - Besedo tipkamo v vnosno okno
  - Ko pridemo do konca besede, pritisnemo Enter
  - Nato lahko pobrišemo zadnjih nekaj znakov (Backspace) in natipkamo preostanek naslednje besede itn.
  - Koliko pritiskov na tipke potrebujemo?
- Primer: abc, aaaa, ba
  - 17 pritiskov: a, b, c, Enter, BS, BS, a, a, a, Enter BS, BS, BS, BS, b, a, Enter

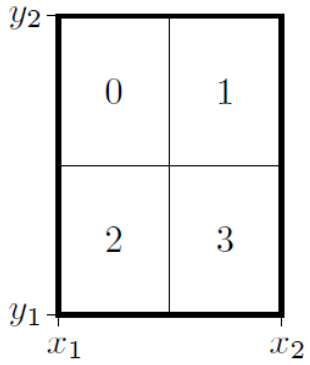
1.1 Tipkanje



- Rešitev:
  - Za dve zaporedni besedi moramo pogledati,
    kako dolgi sta in v koliko prvih znakih se ujemata
  - Če sta dolgi d_1 in d_2 ter se ujemata v prvih u znakih,
    potrebujemo za drugo besedo $(d_1 - u) + (d_2 - u) + 1$ pritiskov
  - Z zanko gremo po vseh besedah in seštevamo pritiske na tipke

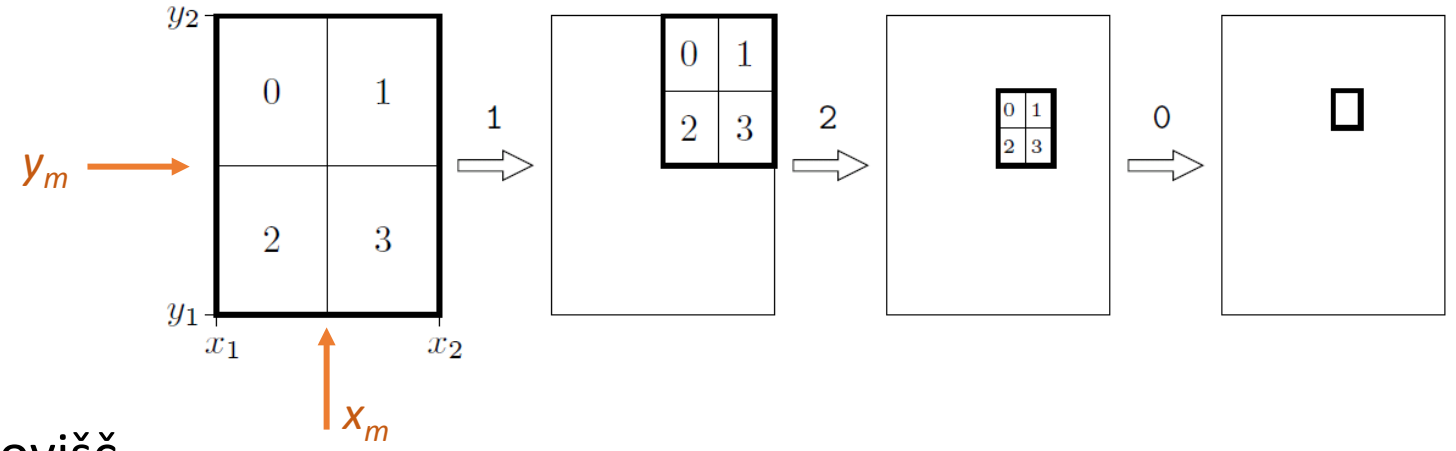
```
stPritiskov = 0
prejsnja = ""; d1 = 0
for beseda in vhodniSeznam:
    u = 0; d2 = len(beseda)
    while u < d1 and u < d2 and prejsnja[u] == beseda[u]:  u += 1
    stPritiskov += d1 – u + d2 – u + 1
    prejsnja = beseda; d1 = d2
print(stPritiskov)
```

1.2 Zoom



- Začnemo s pravokotnikom $[x_1, x_2] \times [y_1, y_2]$
  - Razdelimo ga na četrtine in se premaknemo v eno od njih
  - Ta korak večkrat ponovimo
  - Izpiši koordinate končnega pravokotnika
  - Zaporedje premikov je podano kot niz znakov `0`, `1`, `2`, `3`

1.2 Zoom



- Rešitev:
  - Izračunajmo koordinati razpolovišč
    $x_m = (x_1 + x_2) / 2, \quad y_m = (y_1 + y_2) / 2$
  - Če je naša četrtina ena od levih dveh (0 ali 2), se x_2 premakne na x_m, sicer se x_1 premakne na x_m
  - Če je naša četrtina ena od zgornjih dveh (0 ali 1), se y_1 premakne na y_m, sicer se y_2 premakne na y_m
  - To ponavljamo v zanki po vhodnem nizu

```
for c in s:
    xm = (x1 + x2) / 2; ym = (y1 + y2) / 2
    if c == "0" or c == "2": x2 = xm
    else: x1 = xm
    if c == "0" or c == "1": y1 = ym
    else: y2 = ym
print("[%g, %g] * [%g, %g]" % (x1, x2, y1, y2))
```

1.3 Zaklepajski izrazi

- Niz je oklepajski izraz, če:
  - So vsi znaki le oklepaji in zaklepaji `( ) [ ] { } < >`
  - Vsak oklepaj ima pripadajoč zaklepaj,
    podniz med njima je tudi sam zase oklepajski izraz
  - Primeri: `<<>>`, `[()(<>)]()`, `{[{}()]<>}`
  - Niso oklepajski izrazi: `(()(`, `([)]`, `<>><<>`
- Naloga: dobimo niz zaklepajev in zvezdic
  - Spremeni zvezdice v oklepaje tako, da nastane oklepajski izraz
  - Primer: `**]*))` → `([](())`
  - Včasih to ni mogoče: `***>*)`

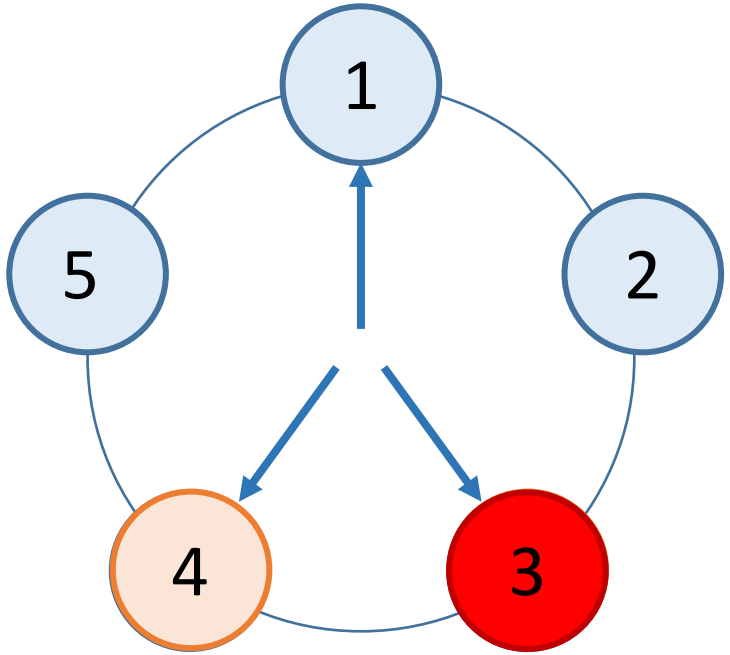
1.3 Zaklepajski izrazi

- Rešitev:
  - Niz pregledujmo od desne proti levi
  - Na skladu (seznam ali tabela) hranimo trenutno odprte zaklepaje
  - Ko preberemo nov zaklepaj, ga dodamo na vrh sklada
  - Ko preberemo zvezdico, jo moramo spremeniti v tak oklepaj, ki bo zaprl zaklepaj z vrha sklada (tega ob tem pobrišemo)

    sklad = [ ]; *n* = len(*s*)
    for *i* = n – 1, n – 2, …, 0:
    if *s*[*i*] je zaklepaj **then** *sklad*.append(*s*[*i*])
    else if *sklad* je prazen **then** *niza ni mogoče popraviti*
    else: *c* = *a*.pop(); *s*[*i*] = pripadajoč oklepaj k zaklepaju *c*
    if *sklad* ni prazen **then** *niza ni mogoče popraviti*

1.4 Izštevanka

- Otroci stojijo v krogu, oštevilčeni od 1 do *n*, vsak ima dve življenji
  - Izštevanka se začne pri otroku 1 in šteje *k* korakov naprej od njega
  - Otrok, pri katerem se štetje ustavi, izgubi eno življenje
  - Štetje se nadaljuje od njega naprej z za 1 večjim *k*-jem
  - Ugotovi, kdo prvi izgubi obe življenji
- Primer: $n = 5$, $k = 3$



1.4 Izštevanka

- Rešitev:
  - Simulirati moramo potek izštevanja
  - *k* mest naprej od otroka *x* je otrok *x* + *k*
  - Če presežemo *n*, je koristno vzeti ostanke po deljenju z *n*
    - Iz *n* samega sicer tako nastane 0 – na to bomo pazili pri izpisu

```
x = 1; ziv = [2] * n
while ziv[x]:
    x = (x + k) % n; ziv[x] –= 1
if x == 0: x = n
print(x)
```

1.5 H-indeks

- Dan je seznam (tabela) nenegativnih celih števil
  - Poišči največji *h*, za katerega velja,
    da je v seznamu vsaj *h* števil, ki so $\geq h$
- Primer: 6, 5, 4, 3, 2, 5, 10, 5, 7
  - Tu je vsaj 5 števil, ki so ≥ 5 (celo 6 je takih števil)
  - Tu NI vsaj 6 števil, ki so ≥ 6 (samo 3 so taka)
  - Rezultat je torej 5

1.5 H-indeks

- Preprosta (in neučinkovita) rešitev:
  - Za nek konkreten h lahko z zanko po seznamu preštejemo, koliko števil je ≥ h
  - Če jih je dovolj, lahko poskusimo s h + 1, nato s h + 2 in tako naprej
  - Potrebujemo torej še eno zunanjo zanko po h

```
h = 0
while True:
    h += 1;  k = 0
    for x in seznam:
        if x >= h: k += 1
    if k < h: return h – 1
```

- Boljše rešitve:
  - Največji primeren *h* poiščemo z bisekcijo
  - Ali pa seznam najprej uredimo

2.1 Sorodstvo

- Imamo množico parov (r_i , s_i) = (leto rojstva, leto smrti) za *n* ljudi
  - Oseba *j* je *morebitni otrok* osebe *i*, če je $r_i + d \leq r_j \leq s_i$
  - Konstanta *d* je podana
  - Sestavi najdaljšo možno verigo ljudi,
    v kateri je vsak človek morebitni otrok prejšnjega v verigi

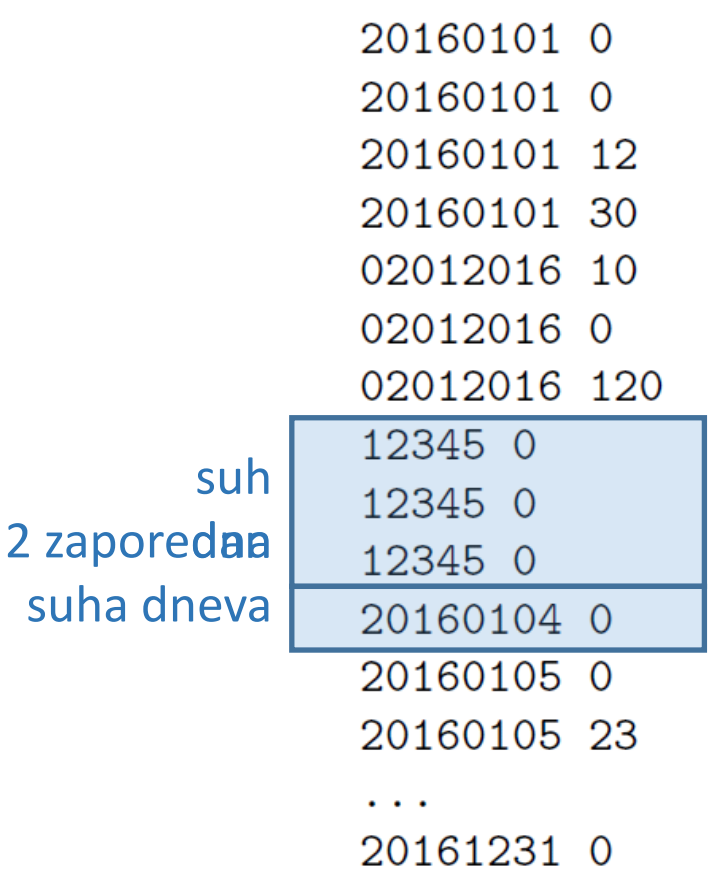
2.1 Sorodstvo

```
for i = 1, 2, …, n:
    a[i] = 1
    for j = 1, 2, …, i – 1:
        if r[i] + d <= r[j] <= s[i]:
            a[i] = max(a[i], 1 + a[j])
```

- Rešitev:
  - Koristno je za začetek urediti vhodne podatke padajoče po letu rojstva r_i
  - Naj bo a_i dolžina najdaljše primerne verige, ki se začne pri človeku i
  - Recimo, da je j naslednji člen te verige
    - To je seveda mogoče le, če je $r_i + d \leq r_j \leq s_i$
  - Preostanek verige mora biti torej čim daljša veriga z začetkom pri j
  - Ta pa je dolga a_j
  - Tako dobimo $a_i = 1 + \max \{ a_j : r_i + d \leq r_j \leq s_i \}$
  - To je koristno računati po padajočem letu rojstva, tako da imamo, ko računamo a_i, že izračunane vse potrebne a_j
  - Izboljšava: nad tabelo a zgradimo drevo z maksimumi po 2, 4, 8, … zaporednih elementov
    - Tako lahko hitreje poiščemo max a_j po več zaporednih j-jih

2.2 Suhi dnevi

- Dobimo zaporedje parov (dan, meritev) z neke merilne postaje
  - Meritve so urejene po času, za vsak dan v letu je 1 ali več meritev
  - Dan je suh, če so vse meritve tistega dne enake 0
  - Preštej, koliko je suhih dni
  - In kako dolga je najdaljša strnjena skupina suhih dni



2.2 Suhi dnevi

- Ko beremo meritve, si zapomnimo:
  - Trenutni dan
  - *b* = Ali je ta dan suh?
  - *s* = Število suhih dni doslej
  - *z* = Dolžina trenutne skupine zaporednih suhih dni (s trenutnim dnem vred)
- To, ali je dan res suh, vemo šele, ko preberemo vse njegove meritve
  - Če ima naslednja prebrana meritev drugačen datum, vemo, da je dosedanjega dneva konec
    - Če je bil ta dan suh, povečamo *s* in *z* za 1
    - Sicer postavimo *z* na 0
    - Postavimo *b* na **true**, da bomo pripravljeni na branje naslednjega dneva
  - Pazimo na robne primere
    - Zadnji dan se konča z EOF, za njim ni novega dne z drugačnim datumom

2.3 Virus

- Imamo n = 1000 zgoščenk, na eni je virus
  - Na računalniku lahko poženemo eno ali več zgoščenk, čez čas bomo videli, ali je bila kakšna od njih okužena ali ne (ne pa tudi, katera)
  - Tak poskus lahko na posameznem računalniku naredimo le enkrat dnevno
  - Radi bi ugotovili, na kateri je virus
    - (1) Z enim računalnikom v čim manj dneh
    - (2) V enem dnevu s čim manj računalniki

2.3 Virus

- Če imamo en sam računalnik
  - Razdelimo zgoščenke na dve skupini po 500, poženimo vse iz ene skupine
  - Odvisno od tega, ali se je okužil ali ne, lahko eno od skupin izločimo iz postopka
  - Preostalo skupino razdelimo na dve še manjši skupini po 250, poženimo vse iz ene od teh podskupin
  - In tako naprej, po največ 10 poskusih nam ostane "skupina" z eno samo zgoščenko, na tej je torej virus
- Če imamo en sam dan (in več računalnikov)
  - Oštevilčimo zgoščenke od 1 do 1000
  - Zapišimo te številke v dvojiškem zapisu z 10 biti: npr. $123_{10} = 00011\ 11011_2$
  - Na računalniku *k* poženimo vse tiste zgoščenke, ki imajo na *k*-tem bitu enico
  - Iz tega, kateri se okužijo, kateri pa ne, rekonstruiramo številko okužene zgoščenke

2.4 Analiza enot

količine enote

```
h : m
g : m / s s
v : m / s
```

- Danih je nekaj definicij, ki povedo,
  v kakšnih enotah se izkažajo določene fizikalne količine
  - Oznake količin in enot so posamezne male črke
  - Na desni strani je lahko ulomek s števcem in imenovalcem
  - Enota je lahko navedena večkrat (npr. s^2 je predstavljena kot par `s s`)
- Nato dobimo še formulo, v kateri nastopajo te količine
  - Na levi in na desni strani sta ulomka (lahko tudi brez imenovalca)
  - Naloga: preveri, ali se enote na obeh straneh formule ujemajo

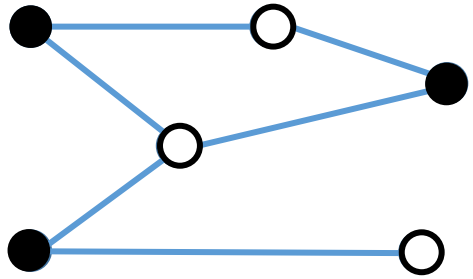
2.4 Analiza enot

- Rešitev:
  - Ker so enote in količine predstavljene s po eno malo črko, imamo največ 26 enot in 26 količin
  - Pripravimo si tabelo 26×26, v kateri bo element $T[k][e]$ hranil stopnjo enote e v količini k (lahko je negativna, če se pojavi v imenovalcu)
  - Ko beremo formulo na koncu, lahko računamo stopnjo enot v formuli:

```
for e in range(26): stopnja[e] = 0;
levo = true; stevec = true;
for c in formula:
    if c == '=': levo = false; stevec = true
    else if c == '#': stevec = false
    else if 'a' <= c <= 'z':
        k = c – 'a'; if levo == stevec then s = 1 else s = –1
        for e in range(26): stopnja[e] += s * T[k][e]
```

  - Na koncu preverimo, če so vse stopnje enake 0

2.5 Žužki



- Dan je neusmerjen graf
  - Ali lahko vsako točko pobarvamo z eno od dveh možnih barv tako, da nobena povezava ne bo povezovala dveh točk enake barve?
- Rešitev:
  - Rešitev, če obstaja, gotovo ni enolična
    (črne točke spremenimo v bele in obratno)
  - Izberimo si torej za začetek poljubno točko *u* in jo pobarvajmo črno
  - Za vse u-jeve sosede zdaj ni druge možnosti, kot da jih pobarvamo belo
  - Za *njihove* sosede zdaj ni druge možnosti, kot da jih pobarvamo črno
  - Tako nadaljujemo, dokler ni pobarvan cel graf

2.5 Žužki

- Zapišimo ta postopek malo podrobneje:

```
for u = 0, 1, …, n – 1: barva[u] = –1
for u = 0, 1, …, n – 1:
    if barva[u] ≥ 0 then continue;   // že pobarvana
    barva[u] := 0; Q := {u};
    while Q ni prazna:
        v := poljubna točka iz Q; pobriši v iz Q;
        za vsako v-jevo sosedo w:
            if barva[w] == barva[v] then return false; // barvanje ni mogoče
            else if barva[w] < 0 then
                barva[w] = 1 – barva[v]; dodaj w v Q
```

3.1 Letala

- Imamo n zabojnikov z maso $m_1, m_2, \ldots, m_n$
- In k letal z nosilnostjo $c_1, c_2, \ldots, c_k$
  - Letalo c_j lahko nosi zabojnik m_i le, če je $c_j \geq m_i$
  - Vsako letalo lahko nese le en zabojnik na enkrat in poleti le enkrat na dan
- V koliko dneh lahko prepeljemo vse zabojnike?

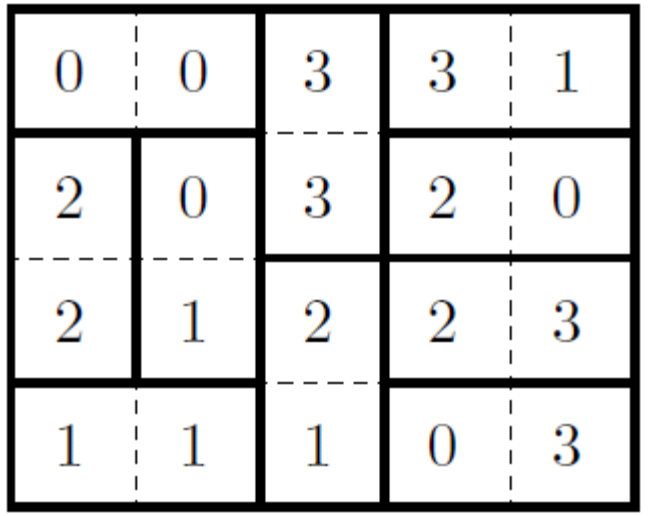
3.1 Letala

- Uredimo letala padajoče po nosilnosti, zabojnike padajoče po masi
- Ali lahko vse zabojnike prepeljemo v t dneh?
  - Najmočnejše letalo (c_1) naj pelje najtežjih t zabojnikov
  - Naslednje letalo (c_2) naj pelje naslednjih t zabojnikov itd.
  - Če se to ne izide (dobi kakšno letalo pretežak zabojnik), potem v t dneh ne gre
    - Zakaj? Tega zabojnika ne moremo dati nobenemu šibkejšemu letalu, saj bo temu še bolj pretežak
    - Močnejšemu letalu pa tudi ne, saj so ta že polno zasedena, naše letalo pa jih ne more razbremeniti, ker so mu njihovi zabojniki še bolj pretežki
  - Ni treba preverjati vseh zabojnikov, ampak le najtežjega pri vsakem letalu
    - Pri c_1 je to m_1, pri c_2 je to m_{t+1}, pri c_3 je to m_{2t+1} itd.
- Najmanjši primerni t lahko poiščemo z bisekcijo

3.1 Letala

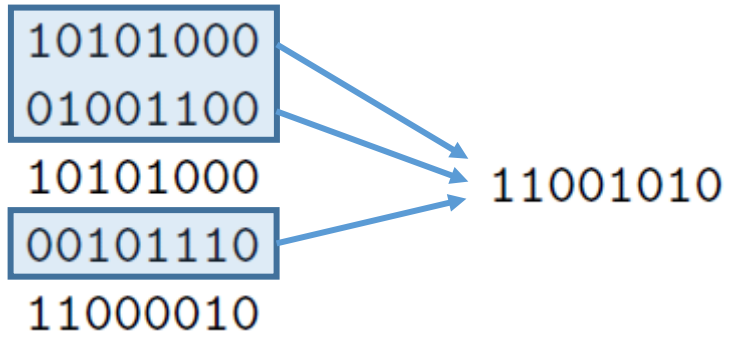
- Še ena rešitev:
  - Preštejmo zabojnike, ki jih lahko pelje le najmočnejše letalo
    - Recimo, da jih je n_1 – torej potrebujemo vsaj n_1 dni
  - Preštejmo zabojnike, ki jih lahko peljeta le najmočnejši dve letali
    - Recimo, da jih je n_2 – torej potrebujemo vsaj $\lceil n_2 / 2 \rceil$ dni
  - Itd. – v splošnem, če je n_p zabojnikov, ki jih lahko pelje le najmočnejših p letal, potrebujemo vsaj $\lceil n_p / p \rceil$ dni
  - Med vsemi temi spodnjimi mejami za t vzemimo največjo

3.2 Dominosa



- Dana je pravokotna mreža števil
  - Razdeliti jo hočemo na domine tako,
    da vsaka celica pripada natanko eni domini in da so vse domine različne
- Rešitev: rekurzija
  - Poiščimo prvo nepokrito celico (najvišje, najbolj levo)
  - Preizkusimo obe možnosti – domino lahko tvori z desno ali s spodnjo sosedo
    - Če smo tako domino že uporabili, v to smer ne nadaljujemo
    - V globalni spremenljivki hranimo podatke o tem, katere domine smo že uporabili
  - Preostanek mreže zapolnimo z rekurzivnim klicem

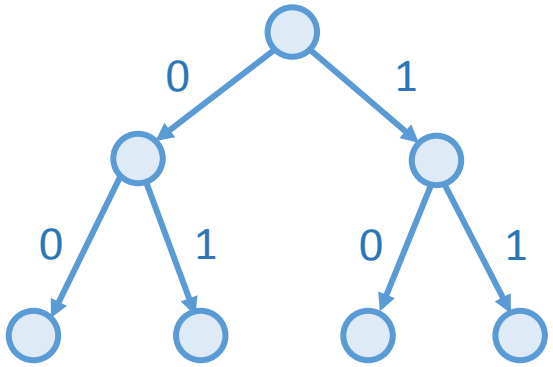
3.3 Galaktična zavezništva



- Dobimo n (do 7500) binarnih nizov, dolgih po m (≤ 50) bitov
- Izbrati moramo tri nize a, b, c tako, da bo $xor(a, b, c)$ največji možni
  - Xor ima prižgan bit tam, kjer ima liho mnogo izmed nizov a, b, c na tistem mestu prižgan bit

- Preprosta rešitev (za majhne primere): [40% točk]
  - Nize predstavimo kot 64-bitna cela števila (**long long**)
  - Vse možne trojice pregledamo s tremi gnezdenimi zankami
  - Žal je vseh trojic približno $n^3 / 6$

3.3 Galaktična zavezništva

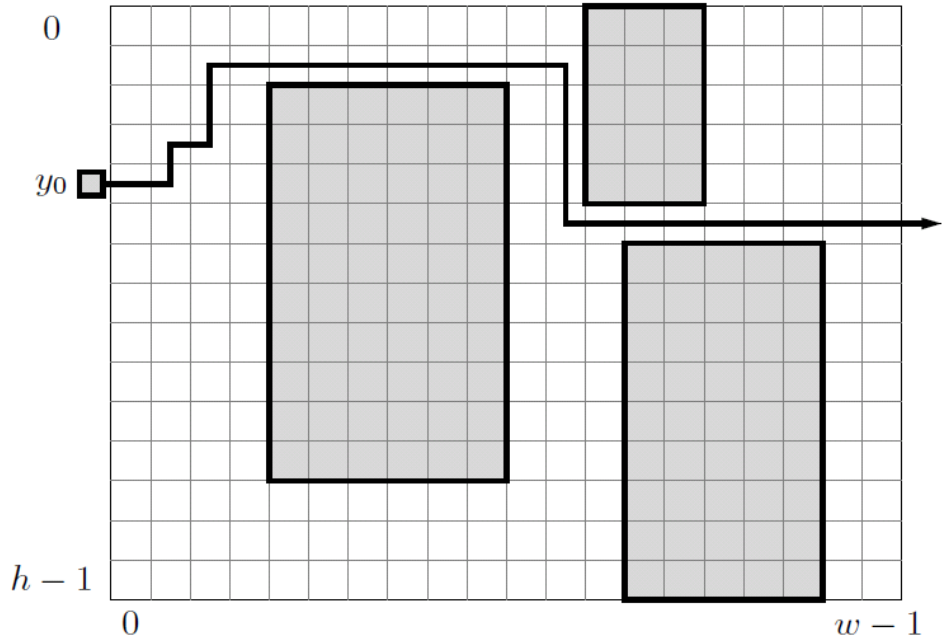


- Razmislimo najprej o lažji nalogi:
  kako bi izbrali *a*, *b* tako, da bi bil *xor*(*a*, *b*) največji?
  - Zložimo nize v drevo (*trie*)
  - Koristno je vzeti *a* iz levega poddrevesa, *b* pa iz desnega
    - Tako se bosta v prvem bitu razlikovala in bo imel *xor*(*a*, *b*) tam enico
    - Če to ne gre (ker je eno od poddreves prazno), ponovimo enak razmislek en nivo nižje
  - Podobno razmišljamo tudi kasneje
    - Hkrati se spuščamo po dveh vejah drevesa – ena bo pripeljala do *a*, druga do *b*
    - Če je le mogoče, skušamo vsakič nadaljevati pot tako, da bi se *a* in *b* razlikovala v istoležnem bitu
      - Pri enem vozlišču gremo v levo poddrevo, pri drugem v desno (ali obratno)
    - Če to ne gre, pač gremo pri obeh v levo ali obeh v desno poddrevo
    - Če imamo za nadaljevanje dve možnosti, preizkusimo obe (z rekurzijo)

3.3 Galaktična zavezništva

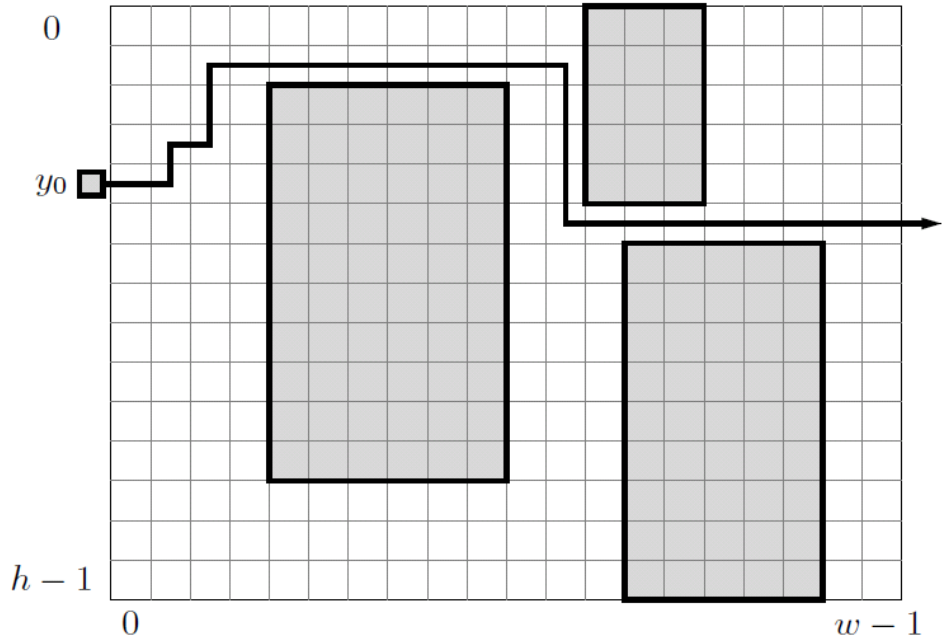
- Kako zdaj rešimo problem za tri nize?
  - Ovijemo prejšnji postopek v še eno zanko:
    za vsak niz *c*:
    pobriši *c* iz drevesa;
    poišči taka *a* in *b*, ki maksimizirata *xor*(*a*, *b*, *c*)
    dodaj *c* nazaj v drevo;
    - Od dobljenih rešitev (pri vseh *c*) vrnemo največjo
  - Kakšna je razlika med maksimiziranjem *xor*(*a*, *b*) in *xor*(*a*, *b*, *c*) pri fiksnem *c*?
    - Prej smo si vedno želeli, da bi se *a* in *b* v istoležnem bitu razlikovala
    - Zdaj pa, če ima *c* na nekem mestu enico, je bolje, če se *a* in *b* tam ujemata

3.4 Asteroidi



- V karirasti mreži $w \times h$ celic je n pravokotnih asteroidov
  - Začnemo v $(0, y_0)$, premikati se smemo le desno, gor in dol
  - Doseči hočemo desni rob s čim manj navpičnimi premiki

3.4 Asteroidi

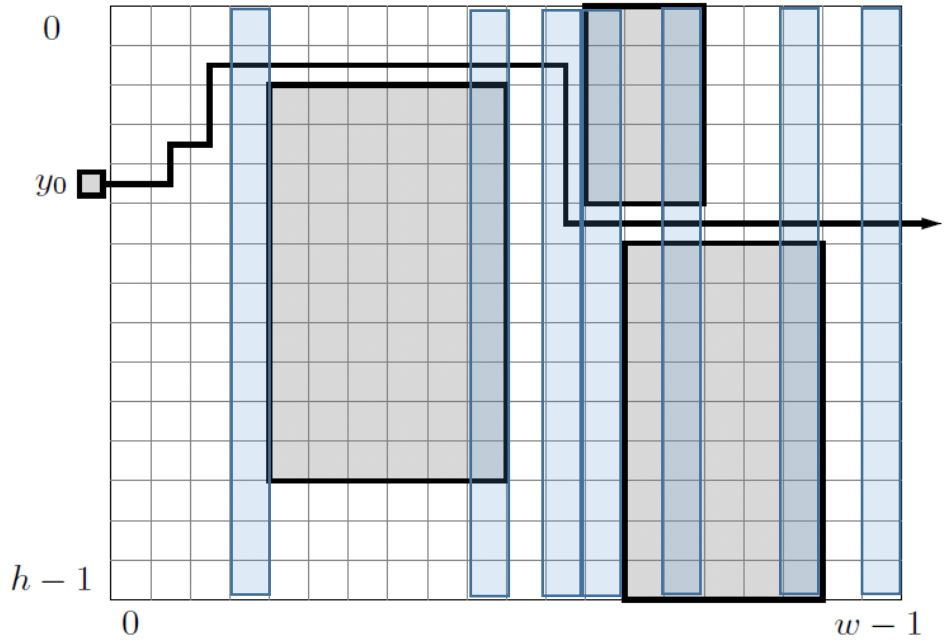


- Preprosta rešitev $O(wh)$ [40% točk]:
  - Celo mrežo predstavimo s tabelo $w \times h$ elementov
    $a[y][x]$ pove, če je na tej celici asteroid
  - Ker se ne moremo premikati v levo, lahko najkrajše poti iščemo po stolpcih od najbolj levega proti najbolj desnemu

```
for (y = 0; y < h; y++) d[y][–1] = (y == y0) ? 0 : ∞;
for (x = 0; x < w; x++) {
    if (a[y][x]) d[y][x] = d[y][x – 1]; else d[y][x] = ∞;
    for (y = 1; y < h; y++) d[y][x] = max(d[y][x], d[y – 1][x] + 1);
    for (y = h – 1; y >= 0; y– –) d[y][x] = max(d[y][x], d[y + 1][x] + 1); }
```

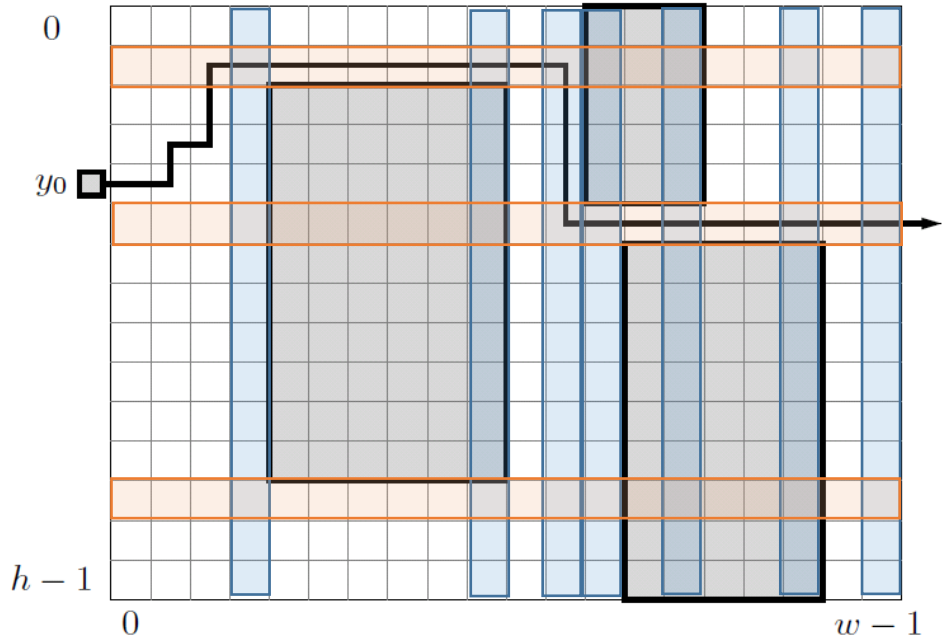
na koncu vrnemo najmanjšo vrednost $d[y][w-1]$ po vseh y

3.4 Asteroidi



- Izboljšava
  - Dva sosednja stolpca, x in $x+1$, se razlikujeta le, če se kakšen asteroid konča v x ali začne v $x+1$
  - Stolpcev je w (do 10^6), asteroidov je največ 300 $\rightarrow$ imamo velike skupine enakih stolpcev
  - V enakih stolpcih lahko delamo iste premike gor/dol
  - Zato lahko od vsake take skupine enakih stolpcev obdržimo samo najbolj desnega, po vseh ostalih se premikamo le v desno
  - Drugače s postopek skoraj nič ne spremeni
  - Časovna zahtevnost le še $O(nh)$ namesto $O(wh)$ [80% točk]

3.4 Asteroidi



- Podobno gre tudi pri vrsticah
  - Recimo, da gremo malo desno po vrstici y in nato dol v $y + 1$
  - Če se noben asteroid ne začne v vrstici $y + 1$, so v $y + 1$ prosta vsa polja, ki so bila prosta v y
  - Torej bi se lahko najprej premaknili dol v $y + 1$ in šli nato v desno po vrstici $y + 1$
  - Podoben razmislek velja tudi za premike gor
  - Torej: dovolj je obdržati vrstice, ki ležijo tik pod ali tik nad kakšnim asteroidom
  - Tako smo se omejili na presek $O(n)$ vrstic in $O(n)$ stolpcev $\rightarrow$ časovna zahtevnost $O(n^2)$ [100% točk]

3.5 Brisanje nizov

- V nizu lahko naenkrat pobrišemo več zaporednih enakih znakov
  - Kako ga s čim manj brisanji popolnoma pobrisati?
  - Primer:
    aabbbacaa → aabbbcaa → aacaa → caa → aa → ""
    aabbbacaa → aaacaa → aaaaa → ""

3.5 Brisanje nizov

- Rešitev: dinamično programiranje
  - Naj bo naš vhodni niz $s = s[0]\ s[1]\ \ldots\ s[n-1]$
  - Zastavimo si malo splošnejši problem:
    $f(i, j)$ = s koliko brisanji lahko popolnoma pobrišemo podniz $s[i\ldots j]$
  - Ko brišemo ta podniz, bo treba prej ali slej pobrisati znak $s[i]$
  - Ena možnost je, da takrat pobrišemo samo njega
    - Ostane torej brisanje podniza $s[i+1\ \ldots\ j]$: skupaj imamo $1 + f(i+1, j)$ brisanj
  - Lahko pa skupaj z njim v istem brisanju pobrišemo še kaj
    - Naj bo $s[k]$ prvi naslednji znak, ki ga pobriše to brisanje (to je možno le, če je $s[k] = s[i]$)
    - Podniz $s[i+1\ \ldots\ k-1]$ moramo torej prej pobrisati v celoti $\rightarrow$ $f(i+1, k-1)$ brisanj
    - Ostane niz $s[i]\ s[k\ldots j]$. Ker sta $s[i]$ in $s[k]$ enaka, lahko $s[i]$ pobrišemo skupaj s $s[k]$, zato je cena enaka kot za brisanje $s[k\ldots j]$. $\rightarrow f(k, j)$ brisanj
  - Torej $f(i, j) = \min \{\ 1 + f(i+1, j),$
    $f(i+1, k-1) + f(k, j)$ [za $i < k \leq j$, če $s[i] = s[k]$] $\}$
  - To računamo od krajših podnizov proti daljšim, rezultate hranimo v tabeli