B-I-1 Syrováľač

Ukážeme si dve rôzne vzorové riešenia, každé poňalo úlohu z iného konca, obe sú však rovnako efektívne čo sa týka času aj pamäti. Každé riešenie sa dá čítať samostatne.

Riešenie spredu

Postupne prechádzame ulicu. Vždy, keď stretneme nejakého konzumenta, pošleme mu syr od naposledy videného producenta rovnakého typu, ktorý ešte nikoho nenakŕmil.

To znie jednoducho, musíme však matematicky dokázať, že takto dostaneme riešenie s najmenšou celkovou prekotúľanou vzdialenosťou. Tvrdenie dokážeme sporom. Ak by nebolo pravdivé, musí existovať optimálne riešenie, v ktorom producent na pozícii p_1 vyrobil syr pre konzumenta na pozícii k_1, ale existoval ešte iný producent tohto druhu syra na pozícii p_2, takej, že $p_1 < p_2 < k_1$ a medzi p_2 a k_1 už nie sú žiadni producenti ani konzumenti tohto druhu syra. Keďže takýchto riešení môže existovať viac, vyberme to, ktoré má najmenší počet takýchto trojíc p_1, p_2, k_1 a toto riešenie nazvime R_{min}.

Ak producent na p_2 nevyrobil syr pre nikoho, vieme zlepšiť prekotúľanú vzdialenosť o $p_2 - p_1$, čiže by sme mali spor s tým, že riešenie R_{min} bolo optimálne.
Ak producent na p_2 vyrobil syr pre nejakého iného konzumenta, ten musí byť na pozícii $k_2 > k_1$. Potom ale existuje riešenie R_2, ktoré vymení tieto dve zásielky, a syr z p_1 doručí na k_2 a z p_2 doručí na k_1. Toto riešenie má rovnakú celkovú prekotúľanú vzdialenosť, má však menej trojíc p_1, p_2, k_1, keďže jednu z nich sme touto výmenou odstránili. To je ale spor s tým, ako sme vybrali R_{min}. □

Máme teda dokázané, že riešenie je správne, už ho len treba efektívne naprogramovať.
Použijeme zásobník – dátovú štruktúru, ktorá si pamätá v akom poradí sme do nej vkladali prvky a vie nám vždy vrátiť posledný vložený prvok. Keď do zásobníka budeme vkladať producentov zľava doprava, budeme vedieť ľahko nájsť najpravejšieho producenta, ktorý ešte nikoho nenakŕmil. Nachádza sa totiž na vrchu zásobníku.
A aby sa nám jednotlivé typy syrov neplietli, vytvoríme si samostatný zásobník pre každý typ. Následne prejdeme našou postupnosťou. Keď narazíme na producenta, vložíme ho do príslušného zásobníka, keď narazíme na konzumenta, vyberieme vrchné číslo z daného zásobníka a pripočítame prekotúľanú vzdialenosť k celkovej odpovedi.

Listing programu (C++)

```
#include <cctype>
#include <iostream>
#include <vector>
#include <stack>
using namespace std;

int main() {
  string ulica;
  cin >> ulica;

  vector<stack<int>> posledna_vyroba(26);

  long long odp = 0;
  for (unsigned i = 0; i < ulica.size(); ++i) {
    char pism = ulica[i];
    if (isupper(pism)) {
      posledna_vyroba[pism - 'A'].push(i);           // pribudol novy producent
    } else {
      odp += i - posledna_vyroba[pism - 'a'].top();  // naposledy videneho producenta pouzijeme...
      posledna_vyroba[pism - 'a'].pop();             // ... a kedze je uz pouzity, zabudneme na neho
    }
  }

  cout << odp << "\n";
}
```

Časová aj pamäťová zložitosť je $O(n)$. Pri pamäti si môžeme uvedomiť, že v najhoršom prípade bude dokopy vo všetkých zásobníkoch najviac n prvkov.

Riešenie odzadu

Ulicu môžeme prechádzať odzadu a v každom momente si pamätať, koľkým konzumentom ešte musíme doručiť syr a nenašli sme pre nich producenta. V zadaní máme sľúbené, že každého konzumenta vieme nakŕmiť. Takže vždy, keď prechádzame po nejakom metri ulice medzi domami, vieme, koľko syrov budeme musieť po tomto metri prekotúľať – presne toľko, koľko konzumentov za nami ešte čaká na svoj syr.
Pre každý meter ulice takto zistíme, koľko syrov musíme po tomto metri prekotúľať. Sčítaním týchto čísel dostaneme odpoveď – celkovú prekotúľanú vzdialenosť. (Namiesto toho, aby sme sa pre každý syr pozreli na to, po koľkých metroch cesty ho budeme kotúľať, sme sa pre každý meter cesty pozreli na to, koľko syrov po ňom budeme kotúľať. Oba súčty zjavne musia mať tú istú hodnotu.)

Príklad so vstupom `BAAba`: Po poslednom metri ulice, teda medzi `BAAb` a `a`, musíme prekotúľať jeden syr: ten idúci do `a`. Po predposlednom metri musíme prekotúľať dva syry. Po treťom metri od konca, teda tom medzi `BA` a `Aba`, už opäť len jeden syr: ten idúci do `b`. A aj po metri na začiatku cesty budeme kotúľať ten istý jeden syr. To je dokopy $1 + 2 + 1 + 1 = 5$ metrov.

Ako to efektívne naprogramovať? Pre každý druh syra si pamätáme, koľko konzumentov máme nenakŕmených. Počítadlo zvyšujeme, keď nájdeme malé písmenko, a znižujeme, keď nájdeme veľké písmenko. Nikdy však počítadlo pre daný typ syra nemôže klesnúť pod nulu – ak by sa tak malo stať, necháme ho na nule. (Toto sa stane práve vtedy, keď nájdeme producenta, ktorého syr sa nepoužije.)

Listing programu (Python)

```
syry = input()
hladni = [0] * 26
kotulanie = 0
for c in reversed(syry):
    if c >= 'a':
        hladni[ord(c) - ord('a')] += 1
    else:
        if hladni[ord(c) - ord('A')] > 0:
            hladni[ord(c) - ord('A')] -= 1
    kotulanie += sum(hladni)
print(kotulanie)
```

V tomto riešení veľakrát počítame súčet d čísel, kde d je počet druhov syra. Ak to chceme zrýchliť, môžeme si vyrobiť druhé počítadlo, ktoré si budeme pamätať súčet čísel v jednotlivých počítadlách a zmeníme ho vždy, keď sa zmení jedno z nich.

Listing programu (Python)

```
syry = input()
hladni = [0] * 26
spolu_hladni = 0
kotulanie = 0
for c in reversed(syry):
    if c >= 'a':
        hladni[ord(c) - ord('a')] += 1
        spolu_hladni += 1
    else:
        if hladni[ord(c) - ord('A')] > 0:
            hladni[ord(c) - ord('A')] -= 1
            spolu_hladni -= 1
    kotulanie += spolu_hladni
print(kotulanie)
```

Časová aj pamäťová zložitosť tohto vylepšeného riešenia je $O(n)$, teda lineárna od dĺžky vstupu.

B-I-2 Eliminácia

Môžeme si zahrať hru s koláčmi na malom vstupe, a zistiť ako Andráš a Sandra prídu na optimálne ťahy.

Majme teda koláče 9 2 4 6 3 5 8 7, a pozrime sa na ne z Andrášovho pohľadu.
Andráš má dve možnosti – buď nechá prvú polovicu koláčov, a pustí na ťah Sandru s koláčmi 9 2 4 6, alebo nechá druhú polovicu koláčov, a pustí ju na ťah s koláčmi 3 5 8 7. Nie je však na prvý pohľad jasné, s ktorým

koláčom by v týchto prípadoch hra vlastne skončila. Nevieme len tak povedať, ktorou polovicou by mala hra pokračovať, aby bol výsledok čo najväčší (čo Andráš chce).
Na to by sme sa museli spýtať podobnú otázku – ak je na ťahu Sandra s koláčmi 9 2 4 6, ktorú polovicu má zahodiť, aby hra skončila s čo najmenším koláčom? Čo ak sa hrá s koláčmi 3 5 8 7?
V prvom prípade môže pustiť Andráša na ťah s koláčmi 9 2 alebo 4 6, v druhom prípade má zas na výber či ho pustí vyberať z koláčov 3 5 alebo 8 7. Toto sa už Andrášovi ľahko rozhoduje, keďže pri dvoch koláčoch platí, že ten koláč, ktorý nechá, bude výsledok hry. Andráš teda nechá teda vždy ten väčší z dvoch koláčov.
Vráťme sa teraz späť k Sandre a prípadu, kde je na ťahu s koláčmi 9 2 4 6. Sandra teraz môže rozmýšľať nasledovne: „Ak nechám Andrášovi prvé dva koláče, on z nich nechá koláč 9. Ak mu nechám druhé dva, on z nich nechá koláč 6. Toto sú teda tie dve možnosti, ktoré mám teraz na výber. No a keďže chcem čo najmenší výsledok, musím mu nechať druhú polovicu.“
Podobnou úvahou zistíme, že z koláčov 3 5 8 7 môže Sandra nechať 3 5 alebo 8 7, pričom prvá voľba vedie k tomu, že Andráš nechá koláč 5 a druhá k tomu, že nechá koláč 8. Čiže tu sa Sandre oplatí zahodiť druhú polovicu a tým dosiahnuť, že výsledkom hry bude 5.
No a teraz sa už môžeme vrátiť ku Andrášovmu ťahu na začiatku hry. Preň vie Andráš zopakovať tú istú úvahu: Ak pustí Sandru na ťah s prvou polovicou koláčov, 9 2 4 6, hra skončí s koláčom 6. Ak ju pustí na ťah s koláčmi 3 5 8 7, hra skončí zasa s koláčom 5. No a teraz si už Andráš vie optimálne vybrať: nechá Sandre prvú polovicu, keďže potom na konci zostane väčší koláč.

Na vyriešenie úlohy treba už len efektívne implementovať tento postup.
Jedna priamočiara možnosť je vytvoriť funkciu, ktorá dostane pole koláčov a hráča, ktorý je na ťahu a zistí, ako by hra skončila z takéhoto stavu. Výsledok (teda číslo koláča, ktorý ostane na konci) táto funkcia vráti.
Výsledok zistíme rekurzívne. Ak má na vstupe iba jeden koláč, hra skončila, nie je sa ako rozhodovať, ten koláč je výsledok hry. Inak naša funkcia dva krát zavolá samú seba, s prvou a druhou polovicou koláčov, a s druhým hráčom. Dostane tak koláče, ktoré by vyhrali dané menšie hry. Z nich potom vyberie lepši koláč podľa preferencie hráča (väčší ak je na ťahu Andráš, menší ak je na ťahu Sandra), a ten vráti ako výsledok.
Stačí nám potom zavolať našu funkciu na celom vstupe s Andrášom na ťahu a vypísať čo nám vráti.
Aby sme zistili našu časovú a pamäťovú zložitosť, najprv si rozmyslime, koľko krát našu funkciu zavoláme. Práve raz ju zavoláme na celom poli koláčov. Tá dva krát zavolá samú seba, raz na prvej a raz na druhej polovici koláčov. V oboch volaniach zavoláme funkciu dva krát, dokopy raz na každej štvrtine koláčov. A tak ďalej, až práve raz zavoláme našu funkciu na každom z našich koláčov (pole dĺžky jedna). Ak $n = 2^k$ (čo máme sľúbené v zadaní), zavoláme ju dokopy $2^0 + 2^1 + 2^2 + \cdots + 2^k$ krát. Pre mocniny dvojky platí, že tento súčet sa rovná $2^{k+1} - 1$, čo je teda $2n - 1$. Našu funkciu teda zavoláme $O(n)$ krát.
Naša časová a pamäťová zložitosť závisí teraz od toho, koľko práce v každom volaní vykonáme a koľko pamäte alokujeme. Jedna z efektívnych možností je posielať len ľavý a pravý index, určujúc úsek poľa koláčov zo vstupu. Z nich vieme v $O(1)$ vypočítať stred úseku (a teda pravý/ľavý index ľavej/pravej polovice nášho úseku), a potrebujeme len $O(1)$ pamäte.
Vieme tak dosiahnuť riešenie s časovou aj pamäťovou zložitosťou $O(n)$.

Listing programu (Python)

```
n = int(input())
kolace = [int(x) for x in input().split()]

ANDRAS, SANDRA = True, False

# Ako hra skonci, ak je hrac na tahu a hra sa s kolacmi na indexoch [L,R) ?
def vysledok_hry(L, R, hrac):
    if L+1 == R:
        return kolace[L]

    stred = L + (R-L)//2
    zahod_pravo = vysledok_hry(stred, R, not hrac)
    zahod_lavo  = vysledok_hry(L, stred, not hrac)

    if hrac == ANDRAS:
        return max(zahod_lavo, zahod_pravo)
    else:
        return min(zahod_lavo, zahod_pravo)

print(vysledok_hry(0, n, ANDRAS))
```

Alternatívnym riešením je riešiť úlohu odzadu. Zistíme si, ktorý z hráčov bude na ťahu posledný. Potom zo všetkých dvojíc koláčov vyberieme ten, ktorý by z danej dvojice nechal vyhrať tento hráč. Následne zo všetkých dvojíc týchto koláčov vyberieme ten, ktorý by nechal vyhrať ten druhý hráč. A tak ďalej, až skončíme s jediným koláčom, ktorý je výsledok hry. Aj toto riešenie vieme implementovať s časovou a pamäťovou zložitosťou $O(n)$.

Listing programu (C++)

```
#include<algorithm>
#include<iostream>
using namespace std;

int n;
int A[1100047];

int main() {
    cin >> n;
    for (int i = 0; i < n; ++i) cin >> A[i];

    int parity = 0;
    for(int jump = 1; jump < n; jump *= 2) {
        parity = 1-parity;
    }

    for(int jump = 1; jump < n; jump *= 2) {
        parity = 1-parity;
        for(int i = 0; i < n; i += jump) {
            A[i] = parity ? min(A[i], A[i+jump]) : max(A[i], A[i+jump]);
        }
    }

    cout << A[0] << endl;
}
```

B-I-3 Predvolebné fotenie

Pozrime sa najskôr na jednoduché riešenie, v ktorom náš program postupne skontroluje celú zadanú časť múru, pri ktorej sa chce kandidát odfotiť. To vieme odsimulovať jedným cyklom, ktorý prechádza intervalom, až kým nenarazí na úsek, ktorý má inú ako požadovanú výšku. Keď budeme mať šťastie, hneď prvý úsek múru v tomto intervale bude iný a náš cyklus môže skončiť. Ak však bude mať veľa úsekov za sebou takú istú výšku, akú žiada kandidát, budeme musieť prejsť oveľa väčšou časťou, prípadne až celým intervalom. A keďže túto možnosť nemôžeme vylúčiť, musíme počítať s časovou zložitosťou $O(mn)$, pretože pre každého kandidáta budeme musieť prinajhoršom prejsť celým múrom.

Naše riešenie teda vieme vylepšiť, ak nájdeme spôsob, akým vieme preskočiť časť múru s úsekmi rovnakej výšky, aby sme ich nemuseli zakaždým kontrolovať. Na to nám poslúži, ak si pre každý úsek múru spočítame, kde sa nachádza najbližší ďalší úsek, ktorý má od neho inú výšku. Napríklad, ak by sme mali postupnosť výšok $(4, 4, 4, 4, 4, 7, 3, 4)$, tak by sme si pre ňu mohli spočítať postupnosť $(6, 6, 6, 6, 6, 7, 8, -)$. Číslo v druhej postupnosti označuje pozíciu najbližšieho ďalšieho iného úseku. Pre prvú hodnotu 4 je to výška 7 na pozícii 6. A pre poslednú hodnotu už ďalší úsek neexistuje, preto tam máme iba pomlčku.

Ak by sme vedeli vypočítať tieto hodnoty, riešenie je následne jednoduché. Pre zadaný interval $\langle z, k\rangle$ sa najprv pozrieme na hodnotu na pozícii z. Ak je táto hodnota iná ako zadané h rovno vypíšeme výsledok. A ak sa rovná h, tak pomocou vypočítaných hodnôt skočíme rovno na najbližšie iné číslo a skontrolujeme, či leží v intervale $\langle z, k\rangle$. Ak áno, vypíšeme ho, ak nie, znamená to, že všetky úseky v intervale mali výšku h. Takéto riešenie zistí pre každého kandidáta výsledok v konštantnom čase – čo vieme formálne zapísať ako $O(1)$.

Ostáva nám už len vymyslieť, ako si predpočítať najbližšie úseky inej výšky. Budeme ich chcieť spočítať postupne všetky, pričom pôjdeme od konca múru. Ak chceme zistiť pozíciu najbližšieho iného úseku k pozícii i, najprv sa pozrieme na pozíciu $i + 1$. Ak je na tejto pozícii iná výška ako na pozícii i, odpoveď je $i + 1$, to je totiž najbližšia pozícia s inou výškou. A ak sú tieto výšky rovnaké, tak tým, že múr prechádzame odzadu, tak pre $i + 1$ sme už najbližšiu pozíciu s inou výškou vypočítali, a táto pozícia je najbližšia aj pre pozíciu i. V jednom prechode so zložitosťou $O(n)$ teda vieme spočítať všetky potrebné hodnoty.

Celková časová zložitosť nášho riešenia bude $O(n+q)$ – v čase $O(n)$ predpočítame najbližšie iné úseky a každú z q otázok potom vyriešime v konštantom čase. Pamätať si potrebujeme jednak celú postupnosť a tiež predpočítané hodnoty, pamäťová zložitosť je preto $O(n)$.

Listing programu (Python)

```
n, q = map(int, input().split())
vysky = list(map(int, input().split()))

najblizsia_ina = [-1] * n

for i in reversed(range(n-1)):
    if vysky[i] != vysky[i+1]:
        najblizsia_ina[i] = i+1
    else:
        najblizsia_ina[i] = najblizsia_ina[i+1]

for _ in range(q):
    z, k, h = map(int, input().split())
    z, k = z - 1, k - 1
    if vysky[z] != h:
        print(z + 1)
    else:
        if najblizsia_ina[z] == -1 or najblizsia_ina[z] > k:
            print(-1)
        else:
            print(najblizsia_ina[z] + 1)
```

B-I-4 Pohovor

Pre zopakovanie, našou úlohou je zmeniť číslo n na číslo 1 pomocou čo najmenšieho počtu operácií: pripočítanie jednotky, odčítanie jednotky, a v prípade, že je číslo párne, vydelenie dvoma.

Podúloha a)

V tejto podúlohe máme dokázať, že pre každé kladné n existuje také optimálne riešenie, ktoré nikdy nepoužije bezprostredne po sebe dve a viac operácií `+1` a `-1`.
Pozrime sa teda na to, čo by sa stalo, ak by sme v našej postupnosti mali viac takýchto operácií po sebe. Ako prvé si môžeme uvedomiť, že sa nikdy neoplatí používať operácie `+1` a `-1` hneď po sebe. Ak totiž k číslu n najprv 1 pripočítame, a potom ju odpočítame, dostaneme opäť číslo n. Mohli sme teda radšej nič nerobiť, ušetriť si použitie dvoch operácií a dosiahnuť ten istý výsledok. Môžeme teda z uvažovania vynechať také postupnosti, kde sa po sebe nachádzajú operácie `+1` a `-1`.
Predstavme si teda situáciu, v ktorej použijeme za sebou viac operácií `+1`. Keďže vďaka nim hodnota n rastie, aby sme sa dostali k výsledku 1, budeme musieť použiť aj inú operáciu. Práve sme si však ukázali, že to nemôže byť operácia `-1`, musí to byť teda operácia `/2`. To znamená, že začíname hodnotou n, potom niekoľkokrát, povedzme si, že k-krát, pripočítame jednotku, čím dostaneme hodnotu $n + k$, a tú následne vydelíme dvoma na hodnotu $\frac{n+k}{2}$.
Nech je k párne. Výsledné číslo potom vieme pekne zapísať ako $\frac{n}{2} + \frac{k}{2}$. Vieme toto číslo vytvoriť z hodnoty n aj na menej operácií ako $k + 1$? Áno, stačí najprv použiť operáciu `/2` a potom $\frac{k}{2}$-krát operáciu `+1`, čím použijeme iba $\frac{k}{2} + 1$ operácií. To ale znamená, že v optimálnom riešení sa takáto postupnosť operácií vyskytnúť nemôže, lebo by sme ju vedeli nahradiť kratšou postupnosťou operácií.
Čo v prípade, že k je nepárne? Funguje tá istá úvaha, akurát sa na výsledné číslo pozrieme ako na $\frac{n+1}{2} + \frac{k-1}{2}$, teda najprv použijeme operáciu `+1`, potom `/2`, a nakoniec $\frac{k-1}{2}$-krát `+1`. Namiesto $k + 1$ operácií nám stačí $\frac{k-1}{2} + 2$ operácií.
Pri operácii `-1` bude myšlienka úplne rovnaká. Ak sme najprv použili k operácií `-1`, a potom operáciu `/2`, vieme dostať to isté číslo tak, že najprv číslo vydelíme 2, a potom odpočítame iba $\frac{k}{2}$ jednotiek. Samozrejme, pri nepárnom k najprv odčítame jednu 1, stále nám však postačí menší počet operácií.
Ostáva už len jedna špeciálna situácia, čo keď po operáciách `-1` nenasleduje operácia `/2`, pretože sme sa dostali na hodnotu 1? Tu si len uvedomíme, že posledná operácia zmení hodnotu n z 2 na 1, takže poslednú `-1` môžeme nahradiť za `/2` a výsledok ani počet operácií tým nezmeníme, akurát už budeme môcť použiť odôvodnenie vyššie. A áno, pre hodnotu $n = 3$ existujú dve optimálne postupnosti. V jednej z nich použijeme dve `-1` za sebou. Nám však stačilo ukázať to, že vždy existuje aj taká postupnosť operácií, kde toto nenastane a taká aj pre $n = 3$ existuje.

Podúloha b)

V druhej podúlohe si máme predstaviť, že zoberieme všetky postupnosti spĺňajúce podmienku z podúlohy a), ktoré menia číslo n na číslo 1, a zapíšeme si všetky čísla, ktoré dostaneme aplikáciou týchto postupností. Úlohou je odhadnúť, koľko týchto čísel bude.
Tú istú otázku si môžeme sformulovať aj nasledovne: Predstavme si, že sme naprogramovali a spustili rekurzívny program, ktorý hľadá optimálne riešenie našej pôvodnej úlohy („ako z čísla n na najmenej krokov vyrobiť číslo 1?“) tak, že skúša generovať a priebežne aj vyhodnocovať práve všetky postupnosti vyššie popísaného tvaru. Nás teraz zaujíma, koľko *rôznych* čísel by tento program aspoň raz uvidel počas svojho behu.
Spomínaný program môže vyzerať napríklad nasledovne:

Listing programu (Python)

```
def rekurzia(n):
    print(n)
    if n == 1:
        # Ak uz sme na n == 1, netreba ziadne dalsie operacie.
        return 0
    if n % 2 == 0:
        # Ak je n parne, musime ho vydelit dvoma.
        # Optimalne riesenie teda zacne tymto krokom a potom pouzijeme optimalne riesenie pre n/2.
        return 1 + rekurzia(n // 2)
    else:
        # Ak je n neparne, musime prave raz bud pripocitat alebo odpocitat 1 a potom novu (parnu) hodnotu
        # vydelit dvoma. Vyskusame obe moznosti a vyberieme si lepsiu z nich.
        return 1 + min(rekurzia(n + 1), rekurzia(n - 1))
```

Môžeme začať veľmi hrubým odhadom. Na začiatočné číslo n nanajvýš raz použijeme operáciu $+1$. Potom už aktuálne číslo musíme vydeliť dvoma, čím vznikne hodnota omnoho menšia ako n. Opakovaním tejto úvahy ľahko nahliadneme, že žiadna z hodnôt, ktoré dostaneme, nebude väčšia ako $n+1$. Všetky hodnoty, ktoré teda teoreticky môžeme uvidieť, ležia v rozsahu od 1 po $n+1$, a teda ich je $O(n)$ – t.j. nanajvýš rádovo n.

Vyššie spravený odhad je v skutočnosti veľmi voľný. Napríklad sme vôbec nevzali do úvahy, že pre veľké n určite neuvidíme v podstate žiadnu z hodnôt medzi $n/2$ a n – čiže skoro polovicu celého rozsahu možných hodnôt.
Pokúsme sa preto spraviť nejaký tesnejší odhad. Pôjdeme postupne po jednotlivých rekurzívnych volaniach a budeme sa pozerať, ktoré hodnoty kedy dosiahneme.
Začnime pekne na začiatku. Vtedy máme jednu hodnotu: n. A máme pred sebou jednu z dvoch možností.
Ak n je párne, rekurzívne sa zavoláme len na jednu hodnotu: $n/2$.
Ak je n nepárne, náš algoritmus sa rekurzívne zavolá na $n+1$ a $n-1$. Obe tieto hodnoty sú párne a preto je ďalší krok algoritmu v oboch z nich opäť jednoznačný: delíme dvoma. V jednej z nich s výsledkom $\frac{n+1}{2}$, v druhej $\frac{n-1}{2}$.
Môžeme si všimnúť, že rozdiel týchto dvoch čísel je 1. Dostali sme teda dve po sebe idúce čísla. Nutne teraz musí platiť, že jedno z nich je párne a druhé nepárne.
Nech je párne napríklad $\frac{n+1}{2}$. (Opačný prípad by mal veľmi podobný rozbor.) V ďalšom kole z tohto párneho čísla teda delením dvoma vznikne číslo $\frac{n+1}{4}$. Pre nepárne číslo $\frac{n-1}{2}$ máme opäť dve možnosti: $\frac{n-1}{2}+1=\frac{n+1}{2}$ a $\frac{n-1}{2}-1=\frac{n-3}{2}$. Toto sú obe párne čísla. Navyše je jedno z nich rovnaké ako hodnota, ktorú sme už videli – $\frac{n+1}{2}$. To ale znamená, že síce si náš algoritmus myslí, že skúma novú vetvu možných postupností, v skutočnosti v nej ale bude znova dostávať také isté čísla, aké už boli v inej vetve.
Navyše, po vydelení čísla $\frac{n-3}{2}$ dvoma dostaneme číslo $\frac{n-3}{4}$, ktoré sa od $\frac{n+1}{4}$ líši práve o 1. A tým sme v rovnakej situácii ako predtým, akurát naše čísla sú zhruba štvrtinou n.
Koľko rôznych čísel teda dokopy vyskúšame? V okolí čísla n to boli najviac 3 čísla – n, $n+1$ a $n-1$. Aj v okolí $\frac{n}{2}$ to boli najviac 3 čísla (v rozobratej možnosti to boli $\frac{n+1}{2}$, $\frac{n-1}{2}$ a $\frac{n-3}{2}$). A ľahko nahliadneme, že rovnako to musí pokračovať aj ďalej: navštívime tri po sebe idúce čísla v okolí $\frac{n}{4}$, z nich sa dostaneme k takejto trojici čísel pri $\frac{n}{8}$, a tak ďalej.
Po $\log_2 n$ opakovaniach vyššie uvedeného postupu sa už dostaneme k číslu 1. Celkový počet rôznych čísel, ktoré náš rekurzívny program navštívi, teda nemôže byť väčší ako $3\log_2 n$.

Podúloha c)

Algoritmus z podúlohy b) je na riešenie tejto podúlohy ešte príliš pomalý. Rôznych vetiev, ktorými sa totiž vie pustiť je až n, keďže sa ich počet zhruba každým druhým krokom zdojnásobí vyskúšaním oboch možností `+1` a `-1`. Vďaka podúlohe b) však vieme, že hoci rôznych vetiev je veľa, dokopy prechádzajú neustále cez tie isté čísla, ktorých je výrazne menej, iba $O(\log n)$.

Dokonca sme si úplne presne všimli, v čom je problém: niektoré vetvy výpočtu, v ktorých máme párnu hodnotu x, sa môžu vyhodnotiť dvakrát – raz, keď ku x prídeme z $2x$ vydelením dvoma, a druhýkrát, keď ku x prídeme z $x \pm 1$ pripočítaním alebo odpočítaním jedničky. Druhýkrát robiť ten istý výpočet je zbytočné (nijaké nové riešenie nenájdeme), len to náš program spomaľuje.

To nás privedie k poslednému vylepšeniu – memoizácii. Pre každé číslo x, ktoré náš algoritmus cestou stretne, nás zaujíma odpoveď na tú istú otázku: „Najmenej koľko krokov potrebujem na to, aby som toto x prerobil na jednotku?“ Každá táto otázka má svoju jednoznačne určenú správnu odpoveď. Akonáhle ju raz pre konkrétne x zistíme, nemá zmysel ju v budúcnosti počítať znova – určite dostaneme tú istú odpoveď.

Napríklad optimálne riešenie pre $x = 4$ má dva kroky. Toto vieme zistiť tak, že rekurzívne vyskúšame všetky možnosti. No a akonáhle sa toto dozvieme, môžeme si túto hodnotu zapamätať. Hocikedy v budúcnosti, keď pri riešení nejakého väčšieho n naša rekurzívna funkcia navštívi hodnotu 4, nemusíme už znova počítať to isté. Namiesto toho môžeme rovno odpovedať zapamätanou hodnotou: „najlepšie riešenie pre túto hodnotu má dva kroky“.

No a toto spravíme pre *úplne všetky* hodnoty, ktoré naša rekurzívna funkcia navštívi. Keď *prvýkrát* navštívime nejakú hodnotu x, urobíme presne to isté ako doteraz: vyskúšame všetky možnosti pre nasledujúci krok a pre každú z nich sa rekurzívne zavoláme. Keď táto rekurzia dobehne, dozvieme sa optimálne riešenie pre toto x. Toto riešenie si *zapamätáme do budúcna*. No a keď *neskôr znova* navštívi naša rekurzia túto istú hodnotu x, už nebudeme zbytočne znova počítať to isté. Namiesto toho rovno odpovieme zapamätanou hodnotou optimálneho riešenia pre toto x.

Pre každú z $O(\log n)$ hodnôt, ktoré naša rekurzívna funkcia navštívi, teda raz vykonáme telo našej rekurzívnej funkcie a tým vypočítame jej optimálne riešenie. Dokopy teda takto upravený program postupne vypočíta $O(\log n)$ medzivýsledkov. Taká istá je teda aj jeho časová zložitosť.

Implementovať toto vylepšenie vieme napríklad nasledovne: Do riešenia z podúlohy b) pridáme asociatívne pole (hešovaciu tabuľku) s názvom `mem`. Do tej si budeme ukladať už vypočítané výsledky: keď sa dozvieme, že pre vstup x je optimálne riešenie y, tak do `mem` ku *kľúču* `x` priradíme hodnotu `y`.

Listing programu (Python)

```
mem = {}

def rekurzia(n):
    if n == 1:
        return 0
    if n in mem:
        # Ak sme uz tuto hodnotu niekedy stretli, mame uz zapamatane.
        # ake ma optimalne riesenie. Rovno vratime zapamatanu hodnotu.
        return mem[n]

    # Ak sme sa dostali sem, hodnota n je nova a jej riesenie este nevieme.
    # Rekurzivnymi volaniami ho teda zistime.
    if n % 2 == 0:
        odpoved = 1 + rekurzia(n // 2)
    else:
        odpoved = 1 + min(rekurzia(n + 1), rekurzia(n - 1))

    # Zistene riesenie si zapamatame a vratime ho.
    mem[n] = odpoved
    return odpoved

n = int(input())
print(rekurzia(n))
```

TRIDSIATY ÔSMY ROČNÍK OLYMPIÁDY V INFORMATIKE

Príprava úloh: Michal Anderle, Michal Forišek, Ján Hozza, Andrej Korman
Recenzia: Michal Forišek
Slovenská komisia Olympiády v informatike
Vydal: NIVAM – Národný inštitút vzdelávania a mládeže, Bratislava 2022