Písomná skúška z ADŠ dňa 20. 12. 2013

1 (True or False) and Justify [20 bodov]

1.1 Zadanie

O každom tvrdení uveďte, či je pravdivé, a svoj názor stručne (ideálne 1 veta, max 3) zdôvodnite.

1. Mergesort deliaci pole na 1/3 a 2/3 má aj v najhoršom možnom prípade časovú zložitosť $\Theta(n \log n)$.
2. Pomocou vyvaž. binárneho stromu vieme efektívne implementovať abstraktnú dátovú štruktúru „prioritná fronta“.
3. Pomocou binárnej haldy vieme efektívne implementovať abstraktnú dátovú štruktúru „usporiadaná množina“.
4. Máme dve n-prvkové polia. V čase $O(n \log n)$ vieme zistiť, či je jedno permutáciou druhého.
5. Existuje implementácia zásobníku, pri ktorej veľkosť nie je vopred obmedzená (inak ako množstvom dostupnej pamäte) a „push“ aj „pop“ majú zaručene konštantnú (t.j. nie len amortizovane konštantnú) časovú zložitosť.
6. Máme usporiadané pole obsahujúce k^3 prvkov. Zistiť, či sa v ňom nachádza prvok x, vieme v čase $O(\log k)$.
7. Na vygenerovanie všetkých podmnožín danej n-prvkovej množiny treba čas $\Omega(n!)$.
8. $\Theta(n \log n) \subseteq O(n^2)$.

1.2 Riešenie

1. TRUE. Na každej úrovni stromu rekurzie sa spraví nanajvýš lineárne množstvo práce. (Na prvých úrovniach je presne lineárne, odkedy niektoré vetvy skončia je množstvo práce menšie.) A najhlbšia vetva stromu rekurzie má ešte stále hĺbku logaritmickú od n (exaktne je to $\log_{3/2} n$).

2. TRUE. Vieme v čase $O(\log n)$ aj vložiť ľubovoľný prvok, aj nájsť a vybrať maximum/minimum.

3. FALSE. V halde napr. nevieme efektívne vyhľadávať prvok, resp. vymazať prvok s danou hodnotou.

   (Častým argumentom, ktorý ale dostal mierne menej bodov, bolo, že pomocou haldy nevieme efektívne prejsť cez prvky v usporiadanom poradí. Technically, toto nie je úplne pravda, keďže vieme v ešte-stále-efektívnom čase $O(n \log n)$ vytvoriť z haldy usporiadané pole. Je to však pomalšie a menej praktické ako u vyhľadávacích stromov, a navyše sa to nedá kombinovať s inými operáciami.)

4. TRUE. Obe usporiadame (napr. MergeSortom) a následne overíme či sú identické.

   (Za implicitného predpokladu, že prvky poľa vieme porovnávať aj operátorom $<$. Ten v zadaní nedopatrením chýbal, ale vyzerá, že problémy to nespravilo.)

5. TRUE. Pomocou pointrov a spájaného zoznamu.

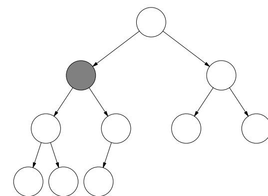
6. TRUE. Totiž $\log k^3 = 3 \log k$, a teda časová zložitosť binárneho vyhľadávania bude $O(\log k)$.

7. FALSE. Stačí nám čas $O(n \cdot 2^n)$, a $n \cdot 2^n$ je pomalšie rastúca funkcia ako $n!$.

8. TRUE. Vľavo sú funkcie rastúce rádovo tak rýchlo ako $n \log n$, vpravo sú všetky nanajvýš kvadratické funkcie, a medzi tie patria aj funkcie zľava. Formálne napr.: $\Theta(n \log n) \subseteq O(n \log n)$ z definície Θ, a následne $O(n \log n) \subseteq O(n^2)$ vďaka rýchlosti rastu dotyčných funkcií.

2 Pohľad zvnútra: halda [5+5 bodov]

2.1 Zadanie



Na obrázku je binárna halda s maximom v koreni. Sú v nej uložené čísla 1 až 10.
a) Doplňte čísla do haldy tak, aby vyfarbené pole malo čo najväčšie číslo.
b) Doplňte čísla do haldy tak, aby vyfarbené pole malo čo najmenšie číslo.
Stručne zdôvodnite, prečo tam väčšie ani menšie číslo od tých vašich byť nemôže.

2.2 Riešenie

Číslo v koreni musí byť od vyznačeného vrcholu väčšie, preto tam môže byť len 9 a menej. Všetky čísla v podstrome pod vyznačeným vrcholom musia byť od neho menšie, preto tam môže byť len 6 a viac. Aj pre 6 aj pre 9 ľahko nájdeme prípustné vyplnenie celej haldy.

3 Pohľad zvnútra: najdrahší prvok [2+8 bodov]

3.1 Zadanie

Budeme sa hrať nasledovnú hru. Mám pole a v ňom n rôzne drahých vecí, očíslovaných od 0 do $n - 1$. V každom kole hry mi poviete dve rôzne čísla vecí a ja vám poviem, ktorá z nich je drahšia. Cieľom hry je nájsť najdrahšiu vec.
Príklad: Mám veci s cenami $(10, 47, 1, 2, 42)$. Opýtate sa na veci číslo 0 a 3. Odpoviem: drahšia je vec číslo 0.
a) Napíšte program, ktorý bude hrať túto hru a v najhoršom prípade položí najmenší možný počet otázok.
b) Dokážte, že program, ktorý ste napísali, má požadovanú vlastnosť.

3.2 Riešenie a)

Nasledujúci program vždy položí presne $n-1$ otázok.

```
kde_je_max = 0
for i in range(1,n): kde_je_max = drahsia( kde_je_max, i )
print( kde_je_max )
```

3.3 Riešenie b)

Vyskytlo sa niekoľko „dôkazov“, ktoré len slovne prerozprávali, čo ich algoritmus z časti a) robí, a následne uzavreli, že keby sa ľubovoľné porovnanie z neho vynechalo, nefungoval by. Toto nie je dôkaz. Dôkaz musí ukázať, že neexistuje žiaden (ani principiálne ináč fungujúci!) algoritmus, ktorý by bol od vášho lepší.
Veľmi obľúbená bola tiež *nesprávna* argumentácia: „na každý prvok sa potrebujem niekedy pozrieť, a teda musím spraviť aspoň $n-1$ porovnaní“. Prečo je táto argumentácia nesprávna? Preto, že každým porovnaním sa pozrieme hneď na dva prvky! A teda z argumentu „na každý prvok sa potrebujem niekedy pozrieť“ vyplýva len slabšie dolné ohraničenie: $\lceil n/2 \rceil$ porovnaní. V správnom dôkaze bolo treba kľúčovú myšlienku sformulovať poriadnejšie.

Správna argumentácia pomocou kandidátov:
Na začiatku každý prvok kandiduje na to byť globálnym maximom. Každou otázkou spomedzi kandidátov vylúčime nanajvýš jedného: toho, o ktorom sme sa práve dozvedeli, že je od niektorého iného prvku lacnejší. Keďže na konci musí byť kandidát len jeden, musíme položiť aspoň $n-1$ otázok.
(Ak sa na porovnávania dívame ako na zápasy, tak množinu kandidátov tvoria tie prvky, ktoré ešte nikdy neprehrali. Kto prehrá, vypadne. Aby ostal len jeden neporazený, musí každý z ostatných aspoň raz prehrať. Z takéhoto pohľadu tiež ľahko vidíme, že ľubovoľný optimálny algoritmus musí v každej otázke porovnávať dva prvky, ktoré v danom okamihu patria medzi kandidátov.)

Správna argumentácia pomocou grafu (čo je vlastne to isté, len si to ináč predstavujeme):
Predstavme si graf, ktorého vrcholy sú čísla 0 až $n-1$. Každú otázku si zaznačíme ako hranu medzi vrcholmi, ktoré sme porovnávali. Ak položíme menej ako $n-1$ otázok, bude mať náš graf aspoň 2 komponenty súvislosti. No a o vzájomnej veľkosti prvkov, ktoré ležia v rôznych komponentoch, nič nevieme, a teda nevieme ani povedať, v ktorom z komponentov leží globálne maximum.
(Rozmyslite si, že každý komponent obsahuje aspoň jedného kandidáta: najväčší z prvkov v ňom.)

4 Pohľad zvnútra: binárny strom [20 bodov]

4.1 Zadanie

Napíšte (nejaký, čím efektívnejší, tým lepšie) algoritmus, ktorý k danému vrcholu binárneho vyhľadávacieho stromu nájde vrchol s nasledujúcou väčšou hodnotou. Inými slovami, napíšte (do detailov) program, ktorý sa spustí vždy, keď inkrementujem iterátor ukazujúci na prvok v `set`e. (Vstupom je ukazovateľ na vrchol v strome, návratovou hodnotou je ukazovateľ na iný vrchol. Detaily uloženia stromu v pamäti si vhodne zvoľte.)

4.2 Riešenie

Algoritmus sme si ukazovali na prednáške. Rozlíšime dva prípady: Ak máme pravého syna, najbližší väčší prvok je najmenší z prvkov v našom pravom podstrome. Ak pravého syna nemáme, najbližší väčší prvok je niektorý náš predok – presnejšie, najbližší náš predok, ktorý je od nás väčší. (To vieme spoznať buď porovnaním hodnôt, alebo jednoducho tak, že je to prvý náš predok, pre ktorého platí, že sme v jeho ľavom podstrome.)
Implementácia v Pythone:

```
def next(kde):
    if kde.right:
        kde = kde.right
        while kde.left: kde = kde.left
        return kde
    else:
        while True:
            if not kde.up: return None
            kam = kde.up
            if kde == kam.left: return kam
            kde = kam
```

Keďže ideme len dodola, resp. len dohora v strome, časová zložitosť je nanajvýš priamo úmerná jeho hĺbke – vo vyváženom strome to teda bude $O(\log n)$.

5 Pohľad zvonka: identifikátory [15 bodov]

5.1 Zadanie

Na vstupe je postupnosť n objektov; každý objekt je nejaký b-bitový reťazec. Dva objekty považujeme za rovnaké len ak sa presne rovnajú ich bitové reprezentácie. Objektom by sme chceli priradiť identifikátory od 0 po $k-1$, kde k je počet rôznych objektov na vstupe. Napíšte čo najefektívnejší program, ktorý načíta dotyčných n objektov, priradí im identifikátory a vypíše postupnosť n identifikátorov zodpovedajúcu vstupu. Odhadnite jeho časovú zložitosť.
Príklad: pre vstup `Jaro`, `Zuza`, `Jaro`, `Fero` je jedným z možných výstupov postupnosť 1, 0, 1, 2.

5.2 Riešenie 1: usporiadaná mapa

Objekty typu popísaného v zadaní vieme ľahko porovnávať – či už testovať na rovnosť, alebo dokonca sa na ne dívať ako na b-bitové čísla a vedieť v prípade, že nenastáva rovnosť, povedať aj, ktoré je menšie. No a toto už stačí napr. na to, aby sme naše objekty použili ako kľúče do mapy.
Pre jednoduchosť budeme pri implementácii predpokladať, že naše objekty sú jednoducho reťazce. (Rozmyslite si, že štandardné operátory `==` a `<` pre reťazce v skutočnosti robia presne to, čo sme popísali v predchádzajúcom odseku.)
V mape si môžeme pamätať ku každému kľúču-objektu identifikátor, ktorý sme mu pridelili. Keď nám príde na vstupe ďalší objekt, pozrieme sa, či už takýto kľúč v mape máme. Ak áno, priradená hodnota je identifikátor, ktorý máme vypísať. A ak nie, tak sme takýto objekt ešte nikdy nevideli. Vtedy inkrementujeme globálne počítadlo (ktoré počíta, koľko *rôznych* objektov sme už videli), novému objektu priradíme príslušné poradové číslo a zapamätáme si túto skutočnosť v našej mape.
Časová zložitosť takéhoto riešenia bude $O(bn \log n)$, pretože spracúvame postupne n objektov, pri každom z nich spravíme $O(\log n)$ porovnaní pri hľadaní kľúča v mape, a každé porovnanie vieme spraviť v čase $O(b)$.

```
#include <iostream>
#include <string>
#include <map>
using namespace std;

int main() {
    int next_free_id = 0;
    map<string,int> ID;
    string name;

    while (cin >> name) {
        if (!ID.count(name)) ID[name] = next_free_id++;
        cout << ID[name] << endl;
    }
}
```

5.3 Riešenie 2: hešovanie

Namiesto usporiadanej mapy môžeme samozrejme využiť hešovanie: v čase $O(b)$ spočítame hešovaciu hodnotu objektu a potom, v očakávanom prípade, urobíme konštantný počet porovnaní objektov. Takéto riešenie nám teda síce nevie dať dobrú *záruku* časovej zložitosti, ale pri použití vhodnej hešovacej funkcie bude jeho *očakávaná* časová zložitosť $O(bn)$.

(Dosť veľa ľudí o tomto riešení bohužiaľ tvrdilo buď „časová zložitosť je $O(bn)$“ alebo dokonca „amortizovaná časová zložitosť je $O(bn)$“. Oboje je nesprávne. Pri hešovaní nikdy nemáme záruku, že všetko naozaj prebehne s nanajvýš konštantným počtom kolízii. A amortizovaná časová zložitosť je niečo principiálne iné – aj pri tej nás zaujíma *zaručený* odhad časovej zložitosti, len tentokrát nie jednej operácie ale ich dostatočne dlhej postupnosti.)

Jednoduchú implementáciu v C++ dostaneme, keď v predchádzajúcom riešení namiesto `map` použijeme `unordered_map`. Ekvivalentná implementácia v Pythone:

```
next_free_id = 0
ID = {}

from sys import stdin
for name in stdin.read().split():
    if name not in ID:
        ID[name] = next_free_id
        next_free_id += 1
    print( ID[name] )
```

6 Pohľad zvonka: lyžovačka [5 × 5 bodov]

6.1 Zadanie

Máme kopec. Na kopci je n významných bodov, očíslovaných od 0 po $n-1$. Vyššie číslo = vyššia nadmorská výška. Bod 0 je spodok a $n-1$ je vrch vleku. Medzi niektorými dvojicami bodov vedú zjazdovky, a to vždy zhora dole. Po zjazdovkách sa odvšadiaľ dá dostať do 0. Našim cieľom je zistiť dĺžku najdlhšej postupnosti zjazdoviek vedúcej z bodu $n-1$ do bodu 0.
Na vstupe je dané dvojrozmerné pole D. Hodnota $D[i][j]$ je buď dĺžka zjazdovky idúcej z i do j, alebo 0 ak taká zjazdovka neexistuje.

a) Uvažujme jednoduchý pažravý algoritmus: ak stojím v bode, z ktorého ide ďalej viacero zjazdoviek, vyberiem si vždy najdlhšiu z nich. Dokážte alebo vyvráťte správnosť tohto algoritmu.
b) Napíšte rekurzívnu funkciu, ktorá náš problém vyrieši tak, že bude postupne skúšať všetky možné spôsoby, ako ísť z bodu $n-1$ do bodu 0. Odhadnite zhora časovú zložitosť tohto riešenia.
c) Vhodne pridajte do riešenia podúlohy b memoizáciu. Odhadnite zhora časovú zložitosť takto vylepšeného riešenia. (Hint: už pri riešení podúlohy b sa oplatí myslieť na to, že následne bude treba riešiť túto podúlohu.)
d) Prepíšte riešenie podúlohy c na dynamické programovanie (t.j. ekvivalentné iteratívne riešenie nepoužívajúce rekurziu).
e) Popíšte, ako upraviť algoritmus z podúlohy c alebo d tak, aby sme nielen zistili maximálnu možnú celkovú dĺžku, ale aj vedeli efektívne zostrojiť (jednu možnú) optimálnu postupnosť zjazdoviek.

6.2 Riešenie podúlohy a)

Najmenší protipríklad má $n = 3$. Uvažujme napr. zjazdovku $2 \to 0$ dĺžky 3 a zjazdovky $2 \to 1$ a $1 \to 0$ obe dĺžky 2. Pažravý algoritmus by šiel zjazdovkou dĺžky 3, avšak optimálna cesta má dĺžku 2+2=4.

6.3 Riešenie podúlohy b)

Pri riešení tejto podúlohy bolo treba rozmýšľať. Ak ste na ňu nešli správne, mohlo sa vám stať, že ste v nasledujúcich podúlohách nemali šancu. V nasledujúcom texte uvádzam ten správny spôsob. Na konci týchto vzorových riešení nájdete dodatok o tom, ako sa túto podúlohu riešiť nemalo.

Nech som v bode x. Ak ešte nie som dole, musím si vybrať nejakú zjazdovku, ktorou pôjdem ďalej. Nech ma tá dovedie do bodu i. Následne budem stáť pred podobným problémom ako mám teraz: budem v bode i a budem sa chcieť dostať dole pomocou čo najdlhšej cesty.
Takto sme práve objavili súvis medzi riešením celého problému a riešením podobných problémov, ktoré sú menšie, jednoduchšie (lebo sme bližšie ku spodku svahu, a teda máme menej možností na výber). Ak chceme nájsť najlepšie riešenie pre bod x, stačí vyskúšať postupne všetky možnosti, ktorou zjazdovkou ísť ďalej, a zakaždým (rekurzívne) vyriešiť problém ako z konca dotyčnej zjazdovky optimálne pokračovať ďalej.
Túto rekurzívnu úvahu ľahko prepíšeme do programu:

```
def najdlhsia(x):
    best = 0
    for i in range(x):
        if D[x][i] > 0:
            best = max( best, D[x][i] + najdlhsia(i) )
    return best

print(najdlhsia(n-1))
```

Všimnite si, že vlastne ani netreba špeciálne ošetrovať bod 0 ako špeciálny prípad: maximálnu vzdialenosť z neho ďalej inicializujeme na 0, a keďže nemáme kam ďalej ísť, tak na 0 aj ostane.
Voľný odhad časovej zložitosti sa dá spraviť napr. nasledovne: v každom bode máme nanajvýš n možností, kam ísť ďalej, a takýchto rozhodnutí máme na každej ceste nanajvýš n, preto je celková časová zložitosť $O(n^n)$.
Tento odhad je síce korektný, ale je príliš voľný. Na lepší odhad si všimnime, že každá cesta zhora dole je jednoznačne určená množinou bodov, ktoré navštívime. A teda možných ciest je len $O(2^n)$ – teda toľko, ako všetkých podmnožín danej množiny bodov. No a časová zložitosť nášho algoritmu je zjavne priamo úmerná počtu možných ciest.

6.4 Riešenie podúlohy c)

Celý výpočet hodnoty vyššie definovanej funkcie `najdlhsia(n-1)` pozostáva z exponenciálneho množstva volaní funkcií, no každá zo zavolaných funkcií je `najdlhsia(i)` pre i od 0 po $n-2$. No a práve tejto neefektívnosti (zbytočného počítania tej istej funkčnej hodnoty veľakrát) sa vieme zbaviť pridaním memoizácie.
Pridanie memoizácie je čisto mechanický proces, pri ktorom už netreba rozmýšľať. Len skontrolujeme, či sme už niekedy daný podproblém riešili (t.j. v našom prípade či už poznáme dĺžku najdlhšej cesty z bodu x dole) a ak nie, tak ho vyriešime a riešenie si zapamätáme.

```
memo = [ None for i in range(n) ]

def najdlhsia(x):
    if memo[x] is not None:
        return memo[x]
    best = 0
    for i in range(x):
        if D[x][i] > 0:
            best = max( best, D[x][i] + najdlhsia(i) )
    memo[x] = best
    return best

print(najdlhsia(n-1))
```

Takto upravený kód každý z n podproblémov vyrieši len raz. No a vyriešenie konkrétneho podproblému (nerátajúc potrebné rekurzívne volania) prebehne v čase $O(n)$. Celková časová zložitosť je preto $O(n^2)$.

6.5 Riešenie podúlohy d)

Prepis z memoizovanej rekurzie na dynamické programovanie je tiež záležitosť takmer mechanická. Jediné, pri čom potrebujeme rozmýšľať, je určenie správneho poradia, v ktorom riešenia podproblémov počítať.
V našom prípade na zistenie dĺžky najdlhšej cesty z x dole potrebujeme poznať dĺžku najdlhšej cesty pre body, do ktorých sa z x vieme priamo dostať – a tie majú všetky číslo menšie ako x. Takže nám stačí jednoducho riešiť podproblémy postupne od 0 po $n-1$. Pri tomto poradí ich riešenia bude platiť, že vždy, keď riešime nejaký podproblém, tak už poznáme všetko, čo na jeho vyriešenie potrebujeme.
Zodpovedajúci program:

```
memo = [ -1 for i in range(n) ]

for x in range(n):
    best = 0
    for i in range(x):
        if D[x][i] > 0:
            best = max( best, D[x][i] + memo[i] )
    memo[x] = best

print( memo[n-1] )
```

Všimnite si, že „telo" vonkajšieho for-cyklu je v podstate izomorfné s telom funkcie `najdlhsia` z predchádzajúcej podúlohy.

6.6 Riešenie podúlohy e)

Okrem toho, že si budeme pamätať, akú dĺžku má optimálne riešenie, zapamätáme si aj voľbu nasledujúcej zjazdovky, ktorá k nemu viedla. Asymptotickú časovú zložitosť algoritmu to neovplyvní.

```
memo   = [ None for i in range(n) ]
odkial = [ None for i in range(n) ]

def najdlhsia(x):
    if memo[x] is not None:
        return memo[x]
    best, prev = 0, None
    for i in range(x):
        if D[x][i] > 0:
            curr = D[x][i] + najdlhsia(i)
            if curr > best:
                best, prev = curr, i
    memo[x]   = best
    odkial[x] = prev
    return best

print(najdlhsia(n-1))
kde = n-1
while kde is not None:
    print(kde)
    kde = odkial[kde]
```

(Niektoré odovzdané riešenia si pre každé `x` pamätali celú optimálnu cestu z neho dole. Toto je síce korektné, ale neefektívne – zbytočne nám to o rád zhorší časovú zložitosť. Totiž na mieste, kde my robíme `odkial[x] = prev`, by toto riešenie muselo zakaždým skopírovať celú starú cestu a potom k nej pridať nový vrchol.)

7 Ťažšie bonusové úlohy

7.1 Zadania

B1 (10 bodov) Keby sme v úlohe 3 chceli nájsť aj najlacnejšiu aj najdrahšiu vec, koľko najmenej otázok nám zaručene stačí?

B2 (6 bodov) V úlohe 6 uvažujme zložitejší pažravý algoritmus: Jednotkovou cenou zjazdovky z i do j nazveme hodnotu $D[i][j]/(i-j)$. Ak stojím v bode, z ktorého ide ďalej viacero zjazdoviek, vyberiem si tú s najväčšou jednotkovou cenou. Dokážte alebo vyvráťte správnosť tohto algoritmu.

B3 (4 body) V úlohe 6 predpokladajme, že namiesto poľa D je na vstupe zoznam jednotlivých zjazdoviek. Počet zjazdoviek označme m a predpokladajme, že m je rádovo menšie ako n^2. Navrhnite čo najlepšiu dátovú štruktúru na ich uloženie a povedzte, ako sa jej použitie prejaví na časovej a pamäťovej zložitosti riešenia podúloh 6c a 6d.

7.2 Riešenie podúlohy B2

Nebolo sa treba zľaknúť, stále fungovali veľmi jednoduché protipríklady. Medzi najmenšie patrí zjazdovka z 2 do 0 s cenou 3, zjazdovka z 2 do 1 s cenou 1 a zjazdovka z 1 do 0 s cenou 3.

7.3 Riešenie podúlohy B3

Riedky graf je vhodné si uložiť ako tzv. zoznam okolí vrcholov: pre každý vrchol budeme mať jeden vektor, v ktorom si uložíme z neho vychádzajúce hrany (v našom prípade teda zjazdovky idúce z príslušného bodu dodola). Takto nám bude stačiť pamäť $O(n+m)$. No a náš algoritmus následne upravíme tak, aby sa namiesto skúšania všetkých možných nasledujúcich *vrcholov* pozeral len na *hrany*, ktoré z aktuálneho vrcholu naozaj vedú. Takéto riešenie, po pridaní memoizácie, sa na každú hranu pozrie len raz. Odtiaľ vyplýva, že aj časová zložitosť sa zlepší na $O(n+m)$.

7.4 Riešenie podúlohy B1

Optimálny algoritmus potrebuje približne $3n/2$ otázok.
Začneme tým, že si prvky rozdelíme do dvojíc (a možno jeden zvýši). V každej dvojici prvky porovnáme. Ostalo nám približne $n/2$ prvkov, ktoré „vyhrali“ (boli väčšie), a približne $n/2$ prvkov, ktoré „prehrali“. (Ak bolo n nepárne, tak ten jeden zvyšný prvok patrí do oboch skupín.) No a následne medzi „víťazmi“ nájdeme maximum a medzi „porazenými“ zase minimum. Ľahko nahliadneme, že takto nájdené hodnoty sú nutne globálnym maximom a minimom – totiž globálne maximum nemohlo po prvom kole skončiť medzi porazenými, a vice versa.
Prečo je tento algoritmus optimálny? Na začiatku máme n prvkov, ktoré môžu byť maximom aj minimom. Vo všeobecnosti počas riešenia vieme stav popísať ako trojicu (a, b, c), kde a je počet prvkov, ktoré môžu byť maximom aj minimom, b je počet prvkov, ktoré môžu byť len maximom (už vyhrali, ešte neprehrali) a c je naopak počet prvkov, ktoré môžu byť len minimom. My sa teda chceme dostať zo stavu $(n, 0, 0)$ do stavu $(0, 1, 1)$.
Rozmyslime si teraz nasledovné pozorovania:

- Prvky, ktoré nemôžu byť maximom ani minimom, nemá zmysel porovnávať. V najhoršom prípade sa totiž z takýchto porovnaní nikdy nič nedozvieme. (Ak takýto prvok x porovnáte napr. s prvkom y, ktorý ešte môže byť maximom, odpoviem vám, že y je väčší, a nič sa nezmenilo.)
- Z rovnakého dôvodu nemá zmysel porovnávať dva prvky, z ktorých môže jeden byť len maximom a druhý len minimom.
- Porovnaním dvoch kandidátov len na maximum (resp. dvoch kandidátov len na minimum) sa zbavíme jedného z nich. Teda prejdeme zo stavu (a, b, c) do stavu $(a, b-1, c)$ (resp. do stavu $(a, b, c-1)$).

- Ak porovnáme kandidáta x len na maximum s kandidátom y na oboje, tiež sa v najhoršom možnom prípade dozvieme len jednu vec: x je väčší. V takomto prípade jediná nová informácia je, že y nie je maximum. A teda sa presunieme zo stavu (a, b, c) do stavu $(a-1, b, c+1)$. Symetricky pre minimum.
- Jedine ak porovnáme dvoch kandidátov na oboje (teda dva prvky, ktoré sme ešte nikdy s ničím neporovnali), máme istotu, že vylúčime dve veci: jeden prestane byť kandidátom na maximum a druhý kandidátom na minimum. Táto operácia nás teda dostane zo stavu (a, b, c) do stavu $(a-2, b+1, c+1)$.

No a to už je všetko. Všimnime si totiž hodnotu $2a + b + c$ (teda počet kandidátov na maximum plus počet kandidátov na minimum). Tá je na začiatku rovná $2n$, na konci má byť rovná 2. Zmenšiť ju vieme o 2 operáciou posledného typu, alebo o 1 niektorými inými typmi operácií. No a operáciu posledného typu vieme použiť nanajvýš $\lfloor n/2 \rfloor$-krát, potom sa nám minú prvky, o ktorých nič nevieme.
Slovne: bez ohľadu na to, aký algoritmus použijeme, vieme počas neho položiť nanajvýš $\lfloor n/2 \rfloor$ otázok takých, že naraz vylúčime dvoch kandidátov. Každou ďalšou otázkou už vieme vylúčiť nanajvýš jedného, a tak treba otázok aspoň (zhruba) $3n/2$.
(Úmyselne som v tomto vzorovom riešení nepísal správne všetky celé časti a ± 1, ktoré by bolo treba na jeho exaktnú formuláciu. Vhodne si domyslite, kam patria a že nič podstatné nemenia.)

8 Dodatky

8.1 Riešenie 3 úlohy 5: univerzálne hešovanie

Obe riešenia uvedené vyššie stačili na plný počet bodov za túto úlohu. Pre zaujímavosť však uvádzame aj lepšiu verziu riešenia s hešovaním. Namiesto toho, aby sme sa spoľahli, že preddefinovaná hešovacia funkcia bude dobre fungovať pre naše objekty (čo v praxi môže a nemusí byť pravda), definujeme si vlastnú, a to pomocou univerzálneho hešovania. Presnejšie si teda definujeme celú triedu hešovacích funkcií, z ktorých si jednu pri spustení programu náhodne vyberieme.

```
#include <unordered_map>
#include <iostream>
#include <cassert>
#include <string>
#include <random>
using namespace std;

template<unsigned max_bits>
class UniversalHasher {
    vector<size_t> hash_data;

    public:
    UniversalHasher() {
        // vyrobime si dobry pseudonahodny generator
        std::random_device rd;
        std::mt19937 gen(rd());
        std::uniform_int_distribution<> dis(0,255);

        // vygenerujeme si dostatocny pocet nahodnych hodnot z rozsahu velkosti size_t
        while (hash_data.size() < max_bits) {
            size_t random_size_t = 0;
            for (unsigned i=0; i<sizeof(size_t); ++i) {
                random_size_t <<= 8;
                random_size_t |= dis(gen);
            }
            hash_data.push_back( random_size_t );
        }
    }

    size_t operator() (const string &key) const {
        assert( 8*key.size() <= hash_data.size() );
        size_t answer = 0;
        for (unsigned i=0; i<key.size(); ++i) {
            unsigned char c = key[i];
            for (unsigned j=0; j<8; ++j) if (c & 1<<j) answer ^= hash_data[8*i+j];
        }
        return answer;
    }
};

class StringComparator {
    public:
    bool operator() (const string &A, const string &B) const { return A==B; }
};

int main() {
    int next_free_id = 0;
    unordered_map< string, int, UniversalHasher<9000>, StringComparator > ID;
    string name;

    while (cin >> name) {
        if (!ID.count(name)) ID[name] = next_free_id++;
        cout << ID[name] << endl;
    }
}
```

8.2 O nevhodných spôsoboch riešenia podúlohy 6 b)

Ako sme si už spomínali, pri riešení nasledujúcich podúloh záležalo na konkrétnej implementácii rekurzívneho riešenia. Tá naša, ktorú sme si ukázali vyššie, je dobrá tým, že parametre funkcie sú len tie vstupy, od ktorých výstupná hodnota naozaj záleží – v našom prípade teda len číslo bodu, kde sa práve nachádzame. Pre kontrast uvádzame inú možnú rekurzívnu funkciu:

```
best = 0

def skusaj(x,d):
    # som v bode x, doteraz som presiel cestu dlzky d, skusam vsetky moznosti ako ist dalej
    if x == 0:
```

```
        # uz som na spodku
        global best
        best = max( best, d )
    else:
        for i in range(x):
            if D[x][i] > 0:
                skusaj( i, d + D[x][i] )

skusaj(7,0)
print(best)
```

Tento program je tiež korektným riešením podúlohy b), dokonca má aj rovnakú časovú zložitosť. Jeho forma je ale výrazne horšia – takýto program nemáme šancu vylepšiť memoizáciou. Jedným z problémov je skutočnosť, že funkcia `skusaj` totiž nie je funkciou v matematickom zmysle, mení globálnu premennú `best`. Tohto problému by sa dalo zbaviť využitím návratovej hodnoty funkcie `skusaj`:

```
def skusaj(x,d):
    # som v bode x, doteraz som presiel cestu dlzky d, skusam vsetky moznosti ako ist dalej
    # na vystup vratim najdlhsiu moznu z najdenych ciest
    if x == 0:
        # uz som na spodku
        return d
    else:
        najviac = 0
        for i in range(x):
            if D[x][i] > 0:
                najviac = max( najviac, skusaj( i, d + D[x][i] ) )
        return najviac

print( skusaj(7,0) )
```

Takúto funkciu by sa už memoizovať dalo… len ešte tam stále máme zbytočný parameter `d`, ktorý nám len rádovo zväčší množstvo rôznych volaní našej funkcia, a tým pádom aj časovú a pamäťovú zložitosť programu.

Prečo hovoríme, že parameter `d` je zbytočný? Všimnite si, že pri skúšaní ciest z bodu `x` ďalej vôbec nezáleží na dĺžke cesty, ktorou sme do bodu `x` prišli. Volanie `skusaj(x,47)` bude skúšať presne tie isté spôsoby cesty z `x` do 0 ako volanie `skusaj(x,42)`. Len teda v prvom prípade každá cesta, ktorú budeme skúšať, bude o $47 - 42 = 5$ dlhšia, ako keď tú istú cestu budeme skúšať v druhom prípade. No a to nás práve vedie k správnej formulácii programu v podobe, ktorú sme uviedli vyššie: jediný parameter našej rekurzívnej funkcie je samotné číslo bodu, v ktorom sme, a návratová hodnota je dĺžka najdlhšej cesty z neho dole.

Dobrý spôsob, ako takúto formuláciu priamo zostrojiť: sformulujem si problém, ktorý chcem riešiť („Aká dlhá je najdlhšia cesta zhora dole?“) a ten skúsim zovšeobecniť („pre každé x, aká dlhá je najdlhšia cesta z x dole?“), pričom dobré zovšeobecnenie je také, kde medzi jednotlivými podproblémami existuje súvis („ak by najdlhšia cesta z x dole začínala hranou z x do y, musela by pokračovať najdlhšou cestou z y dole“).

Iný dobrý spôsob: Po tom, ako si sformulujem problém („Aká dlhá je najdlhšia cesta zhora dole?“), skúsim priamo hľadať dobrý rekurzívny popis jeho riešenia. To väčšinou vyzerá tak, že sa pozriem na všetky možnosti, ako toto riešenie začať, ako spraviť „prvý krok“. Ak mám šťastie, tak po prvom kroku dostanem niečo, čo sa na pôvodné zadanie dostatočne podobá. A práve v takých prípadoch má zmysel danú úlohu riešiť rekurzívne a následne toto riešenie skúsiť zefektívniť memoizáciou.