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

1.1 Zadanie

1. Z in-order zápisu binárneho vyhľadávacieho stromu vieme tento strom jednoznačne zrekonštruovať.
2. Ak do vektoru, v ktorom už je n prvkov, postupne vložím n ďalších, bude zaručene celková časová zložitosť $O(n)$.
3. Dve polia obsahujú po n celých čísel. V čase $O(n \log n)$ sa dá zistiť, či sa niektoré číslo nachádza v oboch poliach.
4. Máme haldu s minimom v koreni, obsahujúcu n prvkov. Na nájdenie *maximálneho* prvku v nej treba čas $\Omega(n \log n)$.
5. Ak budeme v MergeSorte pole zakaždým deliť na štvrtinu a tri štvrtiny, bude časová zložitosť naďalej $O(n \log n)$.
6. Na paličke je n mravcov. Chcú sa stretnúť na jednom mieste. Tým miestom, pre ktoré bude súčet mravcami prejdených vzdialeností minimálny, je aritmetický priemer ich súradníc (t.j. ťažisko množiny mravcov).
7. Nech f je funkcia taká, že $f(n)$ je n-té Fibonacciho číslo. Potom f patrí do $O(2^n)$.
8. Ak pri HashSete zmeníme poradie, v ktorom doň vkladáme prvky, nezmeníme tým celkový počet kolízií.

1.2 Riešenie

1. FALSE. In-order zápisom BST je usporiadaná postupnosť prvkov, ktoré obsahuje. A tá o jeho tvare nič nehovorí.

2. TRUE. Dokonca celá postupnosť vkladania od prázdneho vektoru až po vektor obsahujúci $2n$ prvkov dokopy zaberie len $O(n)$ krokov.

3. TRUE. Napríklad prvé pole usporiadame a následne pre každý prvok druhého poľa pomocou binárneho vyhľadávania zistíme, či je aj v prvom poli. (Existuje aspoň 5 iných, rovnako efektívnych spôsobov.)

4. FALSE. Stačí jednoducho prejsť celé pole, v ktorom je halda uložená, a nájsť v ňom maximum. Toto celé vieme v čase $O(n)$.

5. TRUE. Na každej úrovni stromu rekurzie je celková práca $O(n)$ Najhlbšia vetva stromu rekurzie bude mať hĺbku približne $\log_{4/3} n$.

   (Mimochodom, $(3/4)^3 = 27/64 < 1/2$, preto $\log_{4/3} n < 3 \log_2 n$. Slovne: najhlbšia vetva bude nanajvýš trikrát taká hlboká ako pri klasickom MergeSorte.)

6. FALSE. Intuitívne napr. pre mravce na súradniciach 0, 1, 2, 3 a 1000 je optimálne miesto stretnutia niekde v okolí 0, nie niekde v okolí 200.

   Dotyčným optimálnym miestom stretnutia je *medián* súradníc, dôkaz bol na prednáške. Hlavná myšlienka: keď miesto stretnutia posunieme z mediánu súradníc inam, viacerým mravcom tým uškodíme ako pomôžeme.

7. TRUE. Dokonca platí $f(n) < 2^n$, čo sa ľahko dokáže napr. indukciou.

   (Pozor, otázka sa pýtala na *veľkosť* Fibonacciho čísel, nie na *časovú zložitosť* ich výpočtu!)

8. TRUE, ale vlastne FALSE.

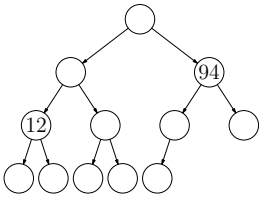
   Na body stačila odpoveď, že počet kolízií je počet dvojíc prvkov, ktoré sa zahešujú na to isté miesto tabuľky. A táto definícia v sebe nikde nezohľadňuje poradie, v ktorom ich vkladáme. (Ak napr. riešime kolízie zreťazením, na konci tak či onak dostaneme rovnako dlhé spájané zoznamy, len prvky v rámci konkrétneho zoznamu nemusia byť v tom istom poradí.)

   V skutočnosti je však odpoveď iná. Vyššie uvedená odpoveď totiž má v sebe oproti zadaniu jeden dodatočný predpoklad: že sa počas celého procesu používa jedna a tá istá hešovacia funkcia. Skutočné HashSety sú zväčša implementované tak, že vždy, keď sa hešovacia tabuľka dostatočne zaplní, alokuje sa nová väčšia a do nej sa všetky prvky zo starej tabuľky prehešujú. Pri takejto implementácii môžu rôzne poradia vkladania prvkov viesť k rôznemu počtu kolízii.

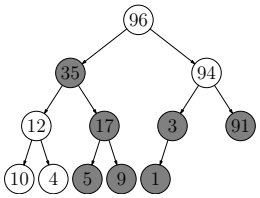
2 Pohľad zvnútra: halda [4+6 bodov]

2.1 Zadanie

Na obrázku je binárna halda s maximom v koreni.
a) Vyfarbite všetky vrcholy, kde sa môže nachádzať prvok s hodnotou 47.
b) Vyplňte všetky prázdne vrcholy navzájom rôznymi prirodzenými číslami tak, aby vznikla korektná halda.



2.2 Riešenie



Prvok s hodnotou 47 sa nemôže nachádzať na ceste hore z prvku 94, ani na žiadnej z ciest dodola z prvku 12.

3 Pohľad zvnútra: prasa strikes back [15 bodov]

3.1 Zadanie

V pamäti máme pole $A[1..n]$. Toto pole bolo voľakedy usporiadané v ostro rastúcom poradí. Potom ale prišlo prasa. A tá sviňa nám toto pole zrotovala o niekoľko pozícií doprava. Príklad: pole $A = (6, 47, 1, 2, 4)$ bolo zrotované o 2 pozície. Napíšte čo najefektívnejšiu funkciu, ktorá pre dané x zistí, či sa v poli A nachádza. Dobré riešenie by to malo zvládnuť bez toho, aby sa pozrelo na väčšinu políčok poľa A.
Pomôcka: Asi to bude jednoduchšie, ak najskôr zistíte, o koľko tá sviňa zrotovala pole. Ale ide to aj bez toho.

3.2 Riešenie

Ak je prvok poľa A väčší alebo rovný $A[1]$, budeme ho volať *veľký*, inak ho budeme volať *malý*. V poli A je najskôr niekoľko (aspoň jeden) veľkých prvkov a následne niekoľko (možno aj nula) malých.
Na určenie presného počtu veľkých prvkov použijeme binárne vyhľadávanie. Pri implementácii môžeme použiť polootvorený interval: ľavú hranicu inicializujeme na 1 (prvok $A[1]$ je určite veľký) a pravú na $n + 1$ (o prvku $A[n + 1]$ si predstavíme, že je malý).
Akonáhle nájdeme počet veľkých prvkov, máme pole A rozdelené na dve rastúce časti. V tej z nich, ktorá môže obsahovať x, ho druhým binárnym vyhľadávaním nájdeme.
Celková časová zložitosť tohto algoritmu je zjavne $\Theta(\log n)$.
Implementácia v Pythone:

```
def najdi(x):
    global A
    # na rozdiel od popisu v programe indexujeme od 0
    lo, hi = 0, len(A)
    while hi-lo>1:
        med = (lo+hi) // 2
        if A[med] >= A[0]: lo = med
        else:              hi = med
    # a teraz najdeme x v spravnej polovici
    if x >= A[0]: lo, hi = 0, hi
    else:         lo, hi = lo, len(A)
    while hi-lo>1:
        med = (lo+hi) // 2
        if A[med] <= x: lo = med
        else:           hi = med
    if A[lo]==x: return lo
    return -1
```

3.3 Iné riešenie

Obe časti predchádzajúceho riešenia sa dá robiť súčasne. Na začiatku sa pozrieme, či je hľadaná hodnota x veľká alebo malá. Následne robíme binárne vyhľadávanie, pričom zakaždým, keď sa pozrieme na nejaký prvok poľa A, vieme povedať, či musí byť x naľavo alebo napravo od neho.
(Sú štyri prípady: Ak je hľadané x veľké a políčko, na ktoré sme sa práve pozreli, malé, je x v ľavej časti. A naopak, ak je x malé a políčko veľké, treba x hľadať napravo. A v prípadoch, kedy sú aj x aj políčko veľké (alebo oba malé) porovnáme ich hodnoty a podľa toho sa rozhodneme.)

4 Pohľad zvnútra: podmnožiny podmnožín [15 bodov a bonus]

4.1 Zadanie

Veliteľ má posádku tvorenú n vojakmi, očíslovanými 1 až n. Potrebuje sa rozhodnúť, ktorí vojaci budú v noci spať a ktorí hliadkovať. A o každom hliadkujúcom vojakovi sa potrebuje rozhodnúť, či bude hliadkovať pri bráne alebo okolo plota. Napíšte program, ktorý veliteľovi vypíše všetky možnosti, ktoré má na výber. (Ak máte program s optimálnou časovou zložitosťou, dostanete bonusové body, ak ju navyše správne odhadnete a dokážete, že je optimálna.)

4.2 Riešenie

Priamo podľa názvu: vygenerujeme všetky podmnožiny vojakov. Pre každú podmnožinu si povieme, že toto sú vojaci, ktorí idú hliadkovať. Následne z nich vyberieme všetkými spôsobmi podmnožinu tých, ktorí budú hliadkovať pri bráne. Na generovanie podmnožín môžeme použiť buď rekurziu, alebo jednoduchú iteráciu po príslušnú mocninu dvoch. Implementácia v Pythone:

```
for hliadkuju in range(2**n):
    subset = [ x for x in range(n) if (hliadkuju & 2**x) ]
    for brana in range(2**len(subset)):
        for i in range(len(subset)):
            if brana & 2**i:
                print 'vojak', subset[i]+1, 'pri brane'
            else:
                print 'vojak', subset[i]+1, 'okolo plotu'
        print
```

4.3 Iné riešenie

Pre každého vojaka máme tri možnosti: buď nehliadkuje vôbec, alebo hliadkuje pri bráne, alebo okolo plotu. Implementácia v Pythone pomocou rekurzie:

```
aktualne_brana = []
aktualne_plot  = []

def vyber(koho):
    if koho==0:
        print aktualne_brana, aktualne_plot
    else:
        # tohto vojaka nepouzijeme
        vyber(koho-1)

        # tohto vojaka dame ku brane
        aktualne_brana.append(koho)
        vyber(koho-1)
        aktualne_brana.pop()

        # tohto vojaka dame ku plotu
        aktualne_plot.append(koho)
        vyber(koho-1)
        aktualne_plot.pop()

vyber(n)
```

4.4 Bonus

Z druhého riešenia je zjavné, že existuje presne 3^n možných rozvrhov.
Tiež ľahko dokážeme, že celková veľkosť výstupu je $\Theta(n3^n)$. Ak vypisujeme všetkých vojakov, je to zjavné: pre každý rozvrh vypíšeme n vojakov. Ale platí to aj vtedy, ak vypisujeme vždy len vojakov, ktorí hliadkujú: totiž každý vojak hliadkuje v 2.3^{n-1} možnostiach.
Každý program riešiaci túto úlohu má teda časovú zložitosť $\Omega(n3^n)$.
O druhom programe je zjavné, že jeho časová zložitosť je $\Theta(n3^n)$, je teda optimálny. Takúto istú časovú zložitosť má aj prvý náš program, dôkaz je založený na identite $\sum_{i=0}^{n} \binom{n}{i} 2^i = 3^n$. Tá vyplýva napr. z binomickej vety pre $(2+1)^n$.

5 Pohľad zvonka: realitná kancelária [7/13/20 bodov]

5.1 Zadanie

Realitná kancelária ponúka na svojej webstránke na predaj byty. Uvažujme niekoľko typov operácií:

1. `navstevnik(s)`: Keď príde návštevník na stránku, dostane otázku, ako drahý byt hľadá. Zadá sumu s a stránka mu ukáže k bytov, ktorých cena je najbližšia sume s. (Číslo k je rádovo menšie ako počet n všetkých bytov.)
2. `novy_byt(adresa,cena)`: Realitka má nový byt na predaj, treba ho pridať do ponuky.
3. `predane(adresa)`: Byt na danej adrese bol predaný, treba ho z ponuky odstrániť.

Vyberte si jednu z možností:
a) (max. 7 bodov) Navrhnite čo najlepšiu dátovú štruktúru umožňujúcu efektívne robiť operácie typu 1.
b) (max. 13 bodov) Navrhnite čo najlepšiu dátovú štruktúru umožňujúcu efektívne robiť operácie typu 1 a 2.
c) (max. 20 bodov) Navrhnite čo najlepšiu sadu dátových štruktúr umožňujúcu efektívne robiť operácie typu 1, 2 aj 3.

Bez ohľadu na vybranú možnosť: Slovne popíšte implementáciu navrhovaného riešenia pomocou dátových štruktúr z prednášky. Odhadnite časovú zložitosť jednotlivých operácií ako funkciu n a k.

5.2 Riešenie podúlohy a)

Ak sa množina bytov nemení, je najjednoduchším riešením mať byty v obyčajnom poli, usporiadané podľa ceny. Keď príde požiadavka, binárnym vyhľadávaním nájdeme miesto zodpovedajúce zadanej sume s a z jeho okolia vypíšeme k bytov. Časová zložitosť každej operácie je $O(k + \log n)$.
Podrobnejší popis výpisu správnych bytov: Nech x je poradové číslo posledného bytu s cenou $\leq s$. Položme $y = x + 1$. Teraz k-krát zopakujeme: Ak je byt číslo x bližšie cenou ku s ako byt číslo y, vypíš byt x a zmenši x, inak vypíš byt y a zväčši y.

5.3 Riešenie podúlohy b)

Ak potrebujeme aj do množiny vedieť pridávať nové byty, najlepšou voľbou je Set: usporiadaná množina, implementovaná ako vyvažovaný binárny vyhľadávací strom. Byty si v ňom budeme udržiavať usporiadané podľa ceny.
Vloženie nového bytu vieme realizovať v čase $O(\log n)$. Výpis bytov pre návštevníka má časovú zložitosť $O(k \log n)$, lebo potrebujeme k-krát posunúť iterátor na predchádzajúci/nasledujúci byt a každé posunutie iterátora má časovú zložitosť $O(\log n)$.
(Existuje aj šikovnejšia implementácia BST, v ktorej si v každom vrchole navyše pamätáme ukazovatele na predchádzajúci a nasledujúci prvok v in-order poradí. V takomto strome vieme vypísať byty s časovou zložitosťou $O(k + \log n)$. V praxi aj v písomke je však dostatočne dobré aj predchádzajúce riešenie.)

5.4 Riešenie podúlohy c)

V čom je problém pri mazaní bytov? V tom, že usporiadané ich máme podľa ceny, zatiaľ čo pri mazaní bytu poznáme len jeho adresu. Aby sme vedeli zmazať byt zo Setu, potrebujeme ho tam efektívne nájsť. A na to treba k adrese vedieť rýchlo určiť cenu príslušného bytu.
Na toto môžeme použiť napr. HashMap (asociatívne pole), v ktorom si ku každej adrese budeme pamätať cenu príslušného bytu. Vymazanie bytu spomedzi ponúkaných potom spravíme na dva kroky: najskôr si pomocou HashMapy zistíme ku adrese cenu, a následne si v Sete nájdeme záznam s príslušnou cenou a adresou a odstránime ho. Očakávaná časová zložitosť je $O(\log n)$ kvôli mazaniu zo Setu.
(Iné riešenie je pamätať si v HashMape priamo ukazovatele do Setu. Ešte iné riešenie je použiť namiesto HashMapy usporiadanú Mapu, časovú zložitosť to nepokazí.)
Implementácia v C++:

```
// kod je pisany podla noveho standardu C++11
// v g++ je skompilovatelny so switchom -std=gnu++11, resp. -std=gnu++0x
#include <iostream>
#include <sstream>
#include <set>
#include <unordered_map>
#include <utility>
using namespace std;

const int INFINITY = 1234567890;
const int K = 3;
set< pair<int,string> > byty_podla_ceny;
unordered_map< string, int > cena_ku_adrese;

int main() {
    // inicializacia
    byty_podla_ceny.insert( make_pair(-INFINITY,"") );
    byty_podla_ceny.insert( make_pair(+INFINITY,"") );

    while (true) {
        string cmd; getline(cin,cmd);

        if (cmd=="novy_byt") {
            string adresa; getline(cin,adresa);
            string tmp; getline(cin,tmp);
            int cena; stringstream(tmp) >> cena;
            byty_podla_ceny.insert( make_pair( cena, adresa ) );
            cena_ku_adrese[ adresa ] = cena;
        }

        if (cmd=="predane") {
            string adresa; getline(cin,adresa);
            int cena = cena_ku_adrese[adresa];
            cena_ku_adrese.erase( adresa );
            byty_podla_ceny.erase( make_pair( cena, adresa ) );
        }

        if (cmd=="navstevnik") {
            string tmp; getline(cin,tmp);
            int s; stringstream(tmp) >> s;
            auto y = byty_podla_ceny.lower_bound( make_pair(s+1,"") );
            auto x = y; --x;
            for (int i=0; i<K; ++i) {
                if (x->first==-INFINITY && y->first==+INFINITY) break; // mame menej ako K bytov v ponuke
                if (x->first==-INFINITY) { ++y; continue; }
```

```
                if (y->first==+INFINITY) { --x; continue; }
                if (abs(s-x->first) < abs(s-y->first)) --x; else ++y;
            }
            for (auto it=++x; it!=y; ++it) cout << it->second << " za cenu " << it->first << endl;
        }
    }
}
```

6 Pohľad zvonka: problém vreca v krajine hojnosti [4 × 5 bodov]

6.1 Zadanie

V sklade je n typov vecí. Každá vec i-teho typu má kladnú celočíselnú hmotnosť m_i (v gramoch) a kladnú cenu c_i. Vecí každého typu je veľmi veľa. Do skladu prišiel zlodej s dostatočne veľkým vrecom. V ňom zvláda odniesť veci s celkovou hmotnosťou nanajvýš M. Nájdite najdrahšiu sadu vecí, ktoré zvláda odniesť. Príklad: pre $M = 65000$, $n = 3$ a typy vecí s $(m_i, c_i) = (30000, 1000), (16000, 120), (4047, 1)$ zlodej vezme 2 veci prvého typu a 1 vec tretieho typu.

a) Napíšte (ako pseudokód alebo kus programu) rekurzívny algoritmus skúšajúci všetky možnosti ako vybrať veci do vreca. (Začne napr. tým, že pre n-tú vec postupne rekurzívne vyskúša možnosti „ešte jednu takúto vezmem“ a „už žiadnu takúto nevezmem“.) Zdôvodnite, že váš program pre ľubovoľný možný vstup naozaj skončí.

b) Pridajte do predchádzajúceho algoritmu memoizáciu tak, aby vznikol algoritmus s časovou zložitosťou polynomiálnou od počtu vecí n. Odhadnite jeho časovú zložitosť.

c) Uveďte ekvivalentný algoritmus, ktorý túto úlohu rieši pomocou dynamického programovania.

d) Existujú situácie, kedy sa môžeme pozrieť na hmotnosti a ceny vecí a z nich priamo usúdiť, že niektoré veci určite v optimálnom riešení nepoužijeme. Pre nasledujúcu sadu typov vecí nájdite dva typy vecí, ktoré zlodej určite kradnúť nebude (bez ohľadu na M). Vyslovte kritérium, ktoré ste použili, všeobecne. Dokážte jeho správnosť. Máme $n = 7$ typov vecí s $(m_i, c_i) = (3200, 1001), (447, 1), (3000, 1000), (1600, 120), (1310, 98), (2320, 84), (415, 1)$.

6.2 Riešenie podúlohy a)

Rekurzívna funkcia bude mať dva parametre: koľko ešte máme voľnej nosnosti a spomedzi koľkých typov vecí ešte vyberáme. Návratová hodnota bude maximálny zisk, ktorý v danej situácii môžeme ešte mať za veci, ktoré naberieme.

```
# globalne premenne N, M_max, M[0..N-1], C[0..N-1]

def najlepsi_zisk(vaha,veci):
    if veci==0: return 0
    odpoved = najlepsi_zisk(vaha,veci-1)
    if M[veci-1] <= vaha:
        odpoved = max( odpoved, C[veci-1] + najlepsi_zisk(vaha-M[veci-1],veci) )
    return odpoved

print najlepsi_zisk(M_max,N)
```

Všimnite si, že oproti verzii, kedy z každej veci máme k dispozícii len jeden kus, sa program takmer vôbec nelíši. Jediný rozdiel je v tom, že ak vec vyberieme, použijeme rekurzívne volanie `najlepsi_zisk(vaha-M[veci-1],veci)` namiesto `...,veci-1)` – teda si ponecháme možnosť vybrať ďalšiu rovnakú vec.

6.3 Riešenie podúlohy b)

```
# globalne premenne N, M_max, M[0..N-1], C[0..N-1]
memo = {}

def najlepsi_zisk(vaha,veci):
    if veci==0: return 0
    if (vaha,veci) in memo: return memo[ (vaha,veci) ]
    odpoved = najlepsi_zisk(vaha,veci-1)
    if M[veci-1] <= vaha:
        odpoved = max( odpoved, C[veci-1] + najlepsi_zisk(vaha-M[veci-1],veci) )
    memo[ (vaha,veci) ] = odpoved
    return odpoved

print najlepsi_zisk(M_max,N)
```

Časová zložitosť je zjavne $O(Mn)$.

6.4 Riešenie podúlohy c)

```
# globalne premenne N, M_max, M[0..N-1], C[0..N-1], memo[0..M][0..N]

for vaha in range(M+1):
    for veci in range(N+1):
        if veci==0:
            memo[vaha][veci] = 0
        else:
            odpoved = memo[vaha][veci-1]
            if M[veci-1] <= vaha:
                odpoved = max( odpoved, C[veci-1] + memo[ vaha-M[veci-1] ][veci] )
            memo[vaha][veci] = odpoved

print memo[M_max][N]
```

6.5 Riešenie podúlohy d)

Definujme, že typ vecí (m_1, c_1) je *aspoň tak výhodný* ako typ (m_2, c_2), ak súčasne platí $m_1 \leq m_2$ aj $c_1 \geq c_2$.
Tvrdenie: Ak máme medzi ponúkanými typmi vecí dva typy také, že (m_1, c_1) je aspoň tak výhodný ako (m_2, c_2), tak existuje optimálne riešenie, v ktorom druhý typ vôbec nepoužijeme.
Dôkaz: Zoberme ľubovoľné optimálne riešenie. Všetky predmety druhého typu (ak tam nejaké sú) môžeme vymeniť za predmety prvého typu: cena tým neklesne, hmotnosť tým nestúpne.
Pre inštanciu zo zadania týmto kritériom vieme vylúčiť typy vecí $(447, 1)$ a $(2320, 84)$.