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

1.1 Zadanie

1. Existuje algoritmus, ktorý ľubovoľné n-prvkové pole, v ktorom je len 47 rôznych hodnôt, usporiada v čase $O(n)$.
2. Rozmiestnenie prvkov v halde nezávisí od poradia, v akom sme ich vkladali.
3. Rozmiestnenie prvkov v binárnom vyhľadávacom strome nezávisí od poradia, v akom sme ich vkladali.
4. O usporiadanom poli s n^2 prvkami vieme v čase $\Theta(\log n)$ zistiť, či obsahuje daný prvok x.
5. Najväčší prvok v binárnom vyhľadávacom strome je aj v pre-order, aj v in-order, aj v post-order zápise posledný.
6. V halde s minimom v koreni pre ľubovoľné prvky x, y platí: ak $x > y$, tak x je aspoň tak hlboko ako y.
7. Každý (aj nevyvážený) binárny vyhľadávací strom s n prvkami má hĺbku $\Omega(\log n)$.
8. Ak v obci s n domami nie sú žiadne dva susedné od seba viac ako 100 m, tak na obec stačí $\lceil n/11 \rceil$ zastávok.

1.2 Riešenie

1. TRUE. Napríklad CountSort: zistíme si, aké hodnoty sa v poli nachádzajú, pre každú spočítame, koľkokrát sa tam nachádza, a na záver pomocou týchto počtov vyplníme výstupné, usporiadané pole.

2. FALSE. Keď do prázdnej minimovej haldy postupne vložíme hodnoty $(1, 2, 3)$, dostaneme inú haldu ako keď vložíme $(1, 3, 2)$.

3. FALSE. Čo vložíme prvé, to bude celý čas v koreni. (Tvrdenie by neplatilo ani vtedy, ak by sme uvažovali vyvažované AVL stromy.)

4. TRUE. Binárne vyhľadávanie má takúto časovú zložitosť, lebo $\log n^2 = 2 \log n$.

5. FALSE. V post-order zápise je posledným vypísaným prvkom ten v koreni, čo síce môže byť najväčší zo všetkých, ale zväčša ním nebude. (Iba v in-order zápise tvrdenie zo zadania platí. Rozmyslite si sami, prečo nemusí platiť v pre-order zápise.)

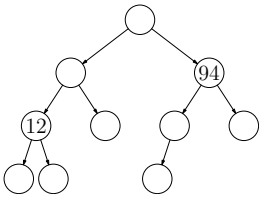
6. FALSE. Napr. minimová halda, ktorá má v koreni hodnotu 1, v ľavom podstrome hodnoty 2, 3, 4 a v pravom podstrome hodnoty 5, 6, 7.

7. TRUE. Slovne, tvrdenie zo zadania hovorí „má hĺbku aspoň rádovo $\log n$“. A to musí platiť, napr. preto, že každý strom hĺbky k má nanajvýš $2^{k+1} - 1$ vrcholov.

8. TRUE. Náš pažravý algoritmus vyrobí takéto pokrytie. Spolu s prvým domom pokryjeme všetky, ktoré sú nanajvýš 1000 m za ním, a takých je aspoň 10 (alebo všetky po koniec dediny, ak sa nám už minuli).

2 Pohľad zvnútra: BST [5+5 bodov]

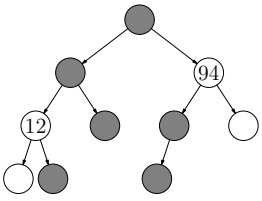
2.1 Zadanie



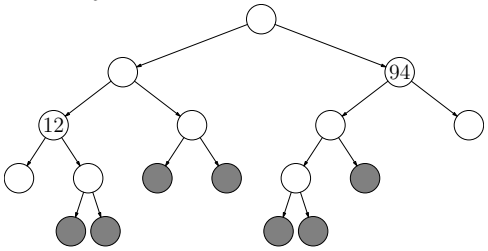
Na obrázku je binárny vyhľadávací strom obsahujúci 10 navzájom rôznych prvkov.
a) Vyfarbite všetky vrcholy, kde sa môže nachádzať prvok s hodnotou 47.
b) V strome nie je prvok s hodnotou 42. Dokreslite všetky možnosti, kde môže pribudnúť nový vrchol, keď prvok 42 do tohto stromu vložíme.

2.2 Riešenie

a) Sú to vrcholy vyfarbené na nasledujúcom obrázku:



b) Sú to nové vyfarbené vrcholy na nasledujúcom obrázku:



3 Pohľad zvnútra: Podobné k sebe [15 bodov]

3.1 Zadanie

V poli $A[0..n-1]$ máme prvky neznámeho typu. Máme funkciu, ktorá nám pre dvojicu prvkov v konštantnom čase povie, či sa podobajú. Podobnosť je symetrická, ale nie nutne tranzitívna. Úlohou je preusporiadať toto pole tak, aby platilo, že každé dva po sebe idúce prvky sa na seba podobajú (alebo zistiť, že sa to nedá).
Dokážte, že *každý* algoritmus riešiaci túto úlohu musí mať časovú zložitosť $\Omega(n \log n)$.

3.2 Riešenie

Ide o maskovanú úlohu hľadania tzv. Hamiltonovskej cesty v grafe – prvky sú vrcholy grafu, funkcia nám o dvojici vrcholov hovorí, či sú spojené hranou. Nie je známy *vôbec žiadny* algoritmus s polynomiálnou časovou zložitosťou, riešiaci túto úlohu. (Keby ste ale vedeli dokázať, že taký algoritmus neexistuje, slušne si zarobíte `:-)`.)
Naša úloha je ale výrazne jednoduchšia, dôkaz je analógiou dôkazu pre triedenie.
Zvoľme si pevné n. Pre každú permutáciu n prvkov existuje konkrétny vstup taký, že dotyčná permutácia a jej reverz sú jeho jedinými dvomi riešeniami. Preto musí každý program riešiaci našu úlohu vedieť dať aspoň $n!/2$ rôznych výstupov. A to znamená, že v najhoršom možnom prípade sa musí postupne pozrieť na aspoň $\lceil \log_2(n!/2) \rceil$ dvojíc prvkov. No a na to treba $\Omega(n \log n)$ krokov výpočtu.

Iné riešenie. Vieme dokonca dokázať, že potrebná časová zložitosť je $\Omega(n^2)$, a to argumentom „niekedy je nutné zistiť podobnosť pre veľmi veľa dvojíc prvkov“.
Nech $n = 2k$. Uvažujme vstup, pre ktorý sú podobné každé dva prvky s číslami 0 až $k-1$ a taktiež každé dva s číslami k až $2k-1$. Tvrdíme, že pre tento vstup ľubovoľný algoritmus riešiaci našu úlohu spraví $\Omega(n^2)$ krokov. To preto, že sa musí pre každú dvojicu prvkov (i, j) takú, že $i < k \leq j$, opýtať, či sa podobajú. Jedine tak totiž overí, že naozaj neexistuje žiadny spôsob ako prvky usporiadať.

4 Pohľad zvnútra: multiset [15 bodov]

4.1 Zadanie

Dostali ste knižnicu, v ktorej je implementovaný `ordered_set`. To je dátová štruktúra, v ktorej viete reprezentovať usporiadanú množinu: efektívne robiť operácie `insert(x)` (vlož `x` ak tam ešte nie je), `erase` (vymaž `x` ak tam je) a `lower_bound(x)` (nájdi najmenší prvok $\geq$ `x`).
Pomocou tejto dátovej štruktúry implementujte `multiset`: dátovú štruktúru pre množinu s násobnými prvkami. Tá by mala podporovať (aspoň) operácie `insert(x)`, `erase_one(x)`, `erase_all(x)` (vymaž jeden výskyt `x`, resp. vymaž všetky výskyty `x`) a `count(x)` (zisti, koľkokrát obsahuje `x`).
Ideálne riešenie okrem volaní metód `ordered_setu` spraví pri každej operácii len $O(1)$ iných krokov výpočtu. Ak takéto riešenie neviete nájsť, nájdite najlepšie, aké viete – prinajhoršom nejako implementujte `multiset` úplne bez použitia `ordered_setu`.

4.2 Riešenie

Multiset vieme ľahko implementovať pomocou asociatívneho poľa: ku každému prvku (kľúču) si budeme pamätať celé číslo (hodnotu) udávajúce, koľkokrát máme dotyčný prvok v našom multisete.
Použijeme teda rovnaký postup ako keď pomocou `setu` implementujeme asociatívne pole: budeme mať `ordered_set`, do ktorého budeme vkladať záznamy – usporiadané dvojice (prvok, celé číslo).
Teraz popíšeme implementáciu jednotlivých metód nášho `multisetu`. Ten budeme označovať `M`, ním používaný `ordered_set` usporiadaných dvojíc budeme označovať `S`.

- Metóda `M.count(x)`: Zavoláme metódu `S.lower_bound` s parametrom `(x,0)`. Ak nám vráti prvok `(x,y)`, vrátime na výstup `y`, inak (ak nám vráti prvok s väčším kľúčom, prípadne žiaden) vrátime na výstup 0.
- Metóda `M.insert(x)`: Zavoláme metódu `M.count(x)`, zistíme si `y`. Ak je `y` kladné, zavoláme `S.erase` s parametrom `(x,y)`. Zavoláme `S.insert` s parametrom `(x,y+1)`.
- Metódy `M.erase_*` implementujeme analogicky.

5 Pohľad zvonka: stíhanie mŕtvych čiar [(7+7+6) bodov + bonus]

5.1 Zadanie

Janku čaká n povinností. O každej z nich vie čas t_i (v sekundách), ktorý jej zaberie, a deadline d_i (v sekundách odteraz, $d_i \geq t_i$), dokedy to musí byť hotové. Činnosť na povinnostiach môže ľubovoľne prerušovať – ak má povinnosť, ktorá trvá 100 sekúnd, môže spraviť 47.7 sekundy teraz a zvyšných 52.3 neskôr.
a) Navrhnite pažravý algoritmus, ktorý zistí, či vie Janka stihnúť všetky povinnosti. Odhadnite jeho časovú zložitosť.
b) Dokážte správnosť vášho pažravého algoritmu (napr.: „Určite nič nepokazím, ak ... lebo ...“)
c) Ťažšia úloha: Každá povinnosť má aj čas začiatku $z_i \leq d_i - t_i$, odkedy sa vôbec dá na nej pracovať.
Upravte algoritmus z časti a) tak, aby riešil aj túto úlohu. (6 bodov za čokoľvek polynomiálne, bonus za efektívne)

5.2 Riešenie

Správny pažravý algoritmus: Vždy robím na tej povinnosti, ktorá má najskôr deadline, a to až kým ju celú nespravím alebo neuplynie dotyčný deadline.
Dôkaz správnosti: Nech existuje nejaký rozvrh, pri ktorom Janka všetko stihne. Pozrime sa na ľubovoľné dve po sebe vykonávané činnosti. Ak robila najskôr na povinnosti s neskorším deadline a potom na povinnosti so skorším, zjavne nič nepokazíme, ak tieto dve činnosti medzi sebou vymeníme. (Neovplyvní to nič pred nimi ani po nich. A výmenu určite môžeme spraviť – rozmyslite si, že ak sme v pôvodnom rozvrhu neprekročili žiaden deadline, neprekročíme ho ani teraz.)
Konečným počtom takýchto výmen vieme ľubovoľný vyhovujúci rozvrh prerobiť na taký, kde Janka postupne pracuje na úlohách v poradí podľa ich deadline. A teda ak existuje korektné riešenie, tak náš pažravý postup korektné riešenie určite vyrobí.

Ťažší problém zas nie je o toľko ťažší – len sa pri rozhodovaní vždy pozeráme iba na tie úlohy, ktoré už môžeme riešiť. Jediná zmena oproti predchádzajúcemu riešeniu je teda nasledovná: ak sa počas toho, ako riešime jednu úlohu, „zjaví“ iná so skorším deadline, tak tú práve riešenú prerušíme a pustíme sa do tej novej.
Inými slovami, algoritmus môžeme implementovať tak, že otázku „čo odteraz robiť?“ si Janka bude zodpovedať nie len v čase 0 a časoch, kedy práve doriešila niektorú úlohu, ale taktiež v každom z časov z_i.

Existuje implementácia s časovou zložitosťou $O(n \log n)$ – simulujeme činnosť Janky, pričom úlohy, ktoré už môže robiť a ešte nesplnila, si pamätáme v halde usporiadané podľa deadline.

6 Pohľad zvonka: problém vreca [4 × 5 bodov + bonus]

6.1 Zadanie

Každý, kto už skúšal narvať kopu vecí do batohu či kufra, vie, že nezáleží len na našej nosnosti, ale aj na objeme vecí. Náš zlodej má pred sebou n vecí. Každá vec má celočíselný objem v_i (v decilitroch), celočíselnú hmotnosť m_i (v stovkách gramov) a cenu c_i. Zlodej má so sebou vrece, do ktorého sa zmestia veci s celkovým objemom nanajvýš V. Zlodej v ňom zvláda odniesť veci s celkovou hmotnosťou nanajvýš M. Nájdite najdrahšiu množinu vecí, ktoré zvláda odniesť.
Príklad: pre $V = 700$, $M = 650$, $n = 3$ a veci s $(v_i, m_i, c_i) = (500, 20, 40), (100, 35, 42), (200, 430, 47)$ vezme veci 2 a 3.

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 „vezmem ju“ a „nevezmem ju“.)
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) Dokážte alebo vyvráťte správnosť pažravého algoritmu:
Pre každú vec vypočítame jej výhodnosť $\varphi_i = c_i \,/\, \max(v_i/V, m_i/M)$ a následne do vreca po jednej vkladáme veci, pričom zakaždým spomedzi vecí, ktoré ešte pripadajú do úvahy, vyberieme tú s najväčšou výhodnosťou.
e) Za bonusové body: Ktorý z algoritmov z častí b) a c) by ste si v praxi vybrali a prečo?

6.2 Riešenie

Stav niekde uprostred skúšania vieme popísať tromi číslami: o koľkých veciach sme sa ešte nerozhodli, či ich chceme; koľko voľného objemu ešte máme vo vreci; koľko hmotnosti ešte zvládame uniesť. Toto teda budú aj parametre našej rekurzívnej funkcie. Tá bude zodpovedať otázku: „Aký najlepší zisk viem mať z vecí s číslami $1, \ldots, a$, ak môžu mať celkový objem do b a celkovú hmotnosť do c?“
V programe to bude vyzerať nasledovne:

```
# globalne premenne: N, veci[1..N]

def najlepsi_zisk(a,b,c):
    if a==0: return 0 # neostali uz ziadne veci, nebude ziadny zisk
    odpoved1 = najlepsi_zisk(a-1,b,c) # najlepsi zisk ak vec cislo a nezoberiem
    odpoved2 = 0
    if (veci[a].objem <= b) and (veci[a].hmotnost <= c):
        odpoved2 = najlepsi_zisk(a-1, b-veci[a].objem, c-veci[a].hmotnost) + veci[a].cena
    return max(odpoved1, odpoved2)

# a volanie, ktoreho vystup nas zaujima
print najlepsi_zisk(N,V,M)
```

Pridanie memoizácie je teraz len mechanická činnosť:

```
# globalne premenne: N, veci[1..N], memo[1..N,0..V,0..M] inicializovane na -1

def najlepsi_zisk(a,b,c):
    if a==0: return 0 # neostali uz ziadne veci, nebude ziadny zisk
    if memo[a,b,c] != -1: return memo[a,b,c] # toto sme uz niekedy pocitali
    odpoved1 = najlepsi_zisk(a-1,b,c) # najlepsi zisk ak vec cislo a nezoberiem
    odpoved2 = 0
    if (veci[a].objem <= b) and (veci[a].hmotnost <= c):
        odpoved2 = najlepsi_zisk(a-1, b-veci[a].objem, c-veci[a].hmotnost) + veci[a].cena
    memo[a,b,c] = max(odpoved1, odpoved2) # zapamatame si spocitanu hodnotu
    return memo[a,b,c]

# a volanie, ktoreho vystup nas zaujima
print najlepsi_zisk(N,V,M)
```

V najhoršom možnom prípade budeme musieť počas behu algoritmu zodpovedať každú z $\Theta(NVM)$ možných otázok. A každú z nich vieme zodpovedať v konštantnom čase, preto aj celková časová zložitosť nášho algoritmu je $O(NVM)$.
Ten istý algoritmus zapísaný ako dynamické programovanie:

```
# globalne premenne: N, veci[1..N], memo[0..N,0..V,0..M] inicializovane na 0

for a in 1..N:
    for b in 0..V:
        for c in 0..M:
            memo[a,b,c] = memo[a-1,b,c]
            if (veci[a].objem <= b) and (veci[a].hmotnost <= c):
                tmp = memo[a-1, b-veci[a].objem, c-veci[a].hmotnost] + veci[a].cena
                if tmp > memo[a,b,c]:
                    memo[a,b,c] = tmp

print memo[N,V,M]
```

Pažravý algoritmus je zovšeobecnením algoritmu „zober vec s najväčšou jednotkovou cenou“ do dvoch rozmerov. Funguje preň protipríklad z prednášky, stačí navyše položiť $V = M$ a pre každú vec $v_i = m_i$. (Alebo $V = \infty$ a pre každú vec $v_i = 1$, takže nás objemy nijak neobmedzujú.)
Uvažujme tri veci, ktorých (cena,hmotnosť) sú nasledovné: (10,51), (9,50), (9,50). Pre $M = 100$ pažravý algoritmus vyberie prvú vec, optimálne je však zobrať zvyšné dve.