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

1.1 Zadanie

1. Ak by sme pri MergeSorte delili pole na 4 približne rovnaké časti, ešte stále by mal časovú zložitosť $\Theta(n \log n)$.
2. Majme dátovú štruktúru vektor, ktorá už obsahuje n prvkov. Potom postupnosť operácií „vlož prvok na koniec“ a „odstráň prvok z konca“ má vždy časovú zložitosť $O(1)$.
3. Majme dátovú štruktúru vektor, ktorá už obsahuje n prvkov. Potom postupnosť operácií „odstráň prvok z konca“ a „vlož prvok na koniec“ má vždy časovú zložitosť $O(1)$.
4. Ak máme v úlohe o stabilných manželstvách m mužov a z žien, tak existuje množina $\min(m, z)$ disjunktných manželstiev, ktorá je stabilná.
5. Z neusporiadaného poľa veľkosti n ($n > 47$) sa v čase $O(n)$ dá vyrobiť binárny vyhľadávací strom s hĺbkou $\leq n/3$.
6. Máme dve polia veľkosti n. V čase $O(n \log n)$ sa dá zistiť, či obsahujú tú istú množinu prvkov.
7. Máme dve polia veľkosti n. V čase $O(n \log n)$ sa dá zistiť, či existuje nejaký prvok, ktorý leží v oboch poliach.
8. Máme usporiadané pole n celých čísel. V čase $O(\log n)$ sa dá zistiť, koľko z nich je kladných.

1.2 Riešenie

1. ÁNO. Nová rekurencia by bola $T(n) = 4T(n/4) + \Theta(n)$, a tá má stále riešenie $T(n) = \Theta(n \log n)$. (Hĺbka stromu rekurzie je $\log_4 n$ a na každej jeho úrovni je celková práca lineárna.)

   Iné zdôvodnenie: v pôvodnom MergeSorte sme pole rozdelili na polovicu, a potom každú z tých polovíc na polovice, čím sme vlastne pôvodné pole rozdelili na štvrtiny. Tento algoritmus teda robí presne to isté ako pôvodný MergeSort.

2. NIE. Pri vkladaní prvku na koniec môže nastať situácia, kedy už naň nemáme voľné miesto a je potrebné alokovať dvakrát viac pamäte.

3. ÁNO. Odobrať prvok vieme v konštantnom čase (stačí zmenšiť počítadlo prvkov) a následne ho pridať tiež (máme istí, že sa do pamäte zmestí, lebo je tam voľné miesto po odobratom prvku).

   (Uznáva sa aj odpoveď NIE, ak uvažujeme takú implementáciu vektoru, ktorá prealokuje pamäť na polovičnú, akonáhle počet prvkov v ňom klesne pod štvrtinu alokovanej pamäte.)

4. ÁNO. Dá sa dokázať, že priamo algoritmus z prednášky takúto sadu manželstiev aj v asymetrickom prípade vyrobí.

   Iný dôkaz: BUNV nech $m < z$, potom si stačí domyslieť $z - m$ mužov, ktorých každá žena chce menej ako tých reálnych. Teraz nájdeme stabilné manželstvá a potom tie z nich, ktoré obsahujú fiktívnych mužov, zahodíme.

5. NIE. Ak by sme toto vedeli, tak vieme v čase $O(n)$ triediť, čo nejde. Podmienka „s hĺbkou $\leq n/3$“ je úplne irelevantná, odpoveď by bola NIE aj bez nej.

6. ÁNO. Polia usporiadame, následne v lineárnom čase vyhádžeme duplikáty a výsledky porovnáme.

7. ÁNO. Polia usporiadame a potom obe naraz prechádzame ako pri Merge.

   Iné riešenie: usporiadame len jedno z polí a následne v ňom prvky druhého poľa binárne vyhľadávame.

8. ÁNO. Binárnym vyhľadávaním nájdeme prvé kladné číslo.

2 Pohľad zvnútra: halda [5 + (6 alebo 10) bodov]

2.1 Zadanie

a) Popíšte a/lebo nakreslite, ako presne by ste do poľa uložili klasickú binárnu haldu (s minimom v koreni).

b1) Napíšte ako pseudokód alebo kus programu funkciu, ktorá do takto uloženej haldy vloží nový prvok.

b2) Napíšte ako pseudokód alebo kus programu funkciu, ktorá z takto uloženej haldy vyberie najmenší prvok.
(Vyberte si jedno z b1 a b2. Za b2 je viac bodov.)

2.2 Riešenie

Uloženie haldy do poľa je vysvetlené v študijných materiáloch. Pri implementácii predpokladáme, že globálne pole H, obsahujúce haldu, je indexované od 1. V globálnej premennej n je uložený aktuálny počet prvkov v halde.

```
def insert(x):
    n = n+1
    H[n] = x
    kde = n
    kým je (kde > 1) a (H[kde] < H[kde/2]):
        vymeň H[kde] a H[kde/2]
        kde = kde/2
```

```
def erase():
    výstup = H[1]
    H[1] = H[n]
    n = n-1
    kde = 1
    do nekonečna opakuj:
        kam = kde
        ak (2*kde   <= n) a (H[2*kde]   < H[kam]): kam = 2*kde
        ak (2*kde+1 <= n) a (H[2*kde+1] < H[kam]): kam = 2*kde+1
        ak kam == kde: return výstup
        vymeň H[kde] a H[kam]
        kde = kam
```

3 Pohľad zvnútra: hľadanie maxima a minima [7+4+3+6+5 bodov]

3.1 Zadanie

Ibrahim má pole a v ňom n prvkov, nie nutne navzájom rôznych. Pre jednoduchosť budeme predpokladať, že $n = 2^k$.

a) Dokážte, že na nájdenie maxima je vždy potrebné spraviť aspoň $n - 1$ porovnaní. Inými slovami, dokážte, že pre každý algoritmus, ktorý spraví menej ako $n - 1$ porovnaní, existuje vstup, pre ktorý nedá správnu odpoveď.

b) Ibrahim potrebuje nájsť aj maximum, aj minimum. Keby hľadal každé zvlášť, potrebuje $2n - 2$ porovnaní. On ale vymyslel algoritmus, ktorý naraz nájde obe: Rozdelíme pole na dve polovice, samostatne v každej nájdeme maximum a minimum a z tých potom vypočítame globálne maximum a minimum. Kostra jeho programu:

```
def minmax(A):
    ak dĺžka A je 2:
        <niečo sa stane>
    inak:
        min1, max1 = minmax(prvá polovica A)
        min2, max2 = minmax(druhá polovica A)
        <niečo iné sa stane>
```

Doplňte optimálne časť „niečo sa stane“.

c) Doplňte optimálne časť „niečo iné sa stane“.

d) Ibrahimov algoritmus na poli s n prvkami spraví $an + b$ porovnaní. Nájdite a a b. Ak $a \geq 2$, niečo robíte zle.

e) Zuzanka sa opýtala: prečo sme ako základný prípad použili „dĺžka A je 2“ a nie až „dĺžka A je 1“? Odpovedzte.

3.2 Riešenie

a) Na začiatku máme n kandidátov na maximum. Po každej otázke vieme vylúčiť nanajvýš jedného z nich, preto treba aspoň $n - 1$ otázok.

Iný dôkaz: otázky, ktoré kladieme, si môžeme značiť ako hrany v n-vrcholovom grafe. Kým nie je tento graf súvislý, tak si nemôžeme byť istí odpoveďou – lebo nevieme porovnať veľkosti prvkov v rôznych jeho komponentoch.

b,c)
```
def minmax(A):
    ak dĺžka A je 2:
        # stačí nám jedno porovnanie
        nech a, b sú prvky poľa A
        ak a<b: return a,b
        inak: return b,a
    inak:
        min1, max1 = minmax(prvá polovica A)
        min2, max2 = minmax(druhá polovica A)
        # stačia nám dve porovnania
        return min(min1,min2), max(max1,max2)
```

Všimnite si, že v prípade, kedy je dĺžka 2, je **nesprávne** písať `return min(a,b), max(a,b)`, to sú totiž dve porovnania, nie jedno!

d) Celkový počet porovnaní spĺňa rekurenciu $T(2) = 1$ a $T(n) = 2T(n/2) + 2$.

Dosadením dostávame $T(4) = 4$. Hľadáme teda také a a b, aby $2a + b = 1$ a $4a + b = 4$. Odtiaľ dostávame $a = 3/2$ a $b = -2$. Skúškou sa ľahko presvedčíme, že $T(n) = (3n/2) - 2$ skutočne spĺňa odvodenú rekurenciu.

(Existuje veľa iných spôsobov, ako sa z rekurencie dopracovať ku číslam a a b, tento mi pripadal najstručnejší. Iná argumentácia: prípad „ak je dĺžka 2“ sa vykoná $n/2$-krát a zakaždým spravíme jedno porovnávanie, opačný prípad sa teda vykoná $((n/2) - 1)$-krát a zakaždým spravíme dve porovnávania.)

e) Všimnime si, že oproti naivnému riešeniu sme ušetrili $n/2$ porovnaní. Kde sa toto ušetrenie udeje? Práve na poliach dĺžky 2, pre ktoré nám stačí jediné porovnanie.

(Rovnako dobrý algoritmus vieme zapísať aj nerekurzívne. Rozdelíme prvky do dvojíc, v každej dvojici nájdeme menší prvok (porazeného) a väčší prvok (víťaza). Potom globálne maximum je maximom víťazov a globálne minimum je minimom porazených.)

Ak by sme našu implementáciu zmenili na `ak dĺžka A je 1: return a,a` (kde `a` je jediný prvok `A`), práve o toto ušetrenie by sme prišli. Pre takto pokazený algoritmus by počet porovnaní spĺňal rekurenciu $T(1) = 0$ (alebo ekvivalentne $T(2) = 2$) a $T(n) = 2T(n/2) + 2$. A tej riešenie už je, ako ľahko nahliadneme, $T(n) = 2n - 2$.

4 Pohľad zvonka: lenivý stavbár [15 bodov]

4.1 Zadanie

Ukrajinec Sergej betónuje terasu, ale sa mu nechce. Spraviť a vyliať za miešačku betónu mu trvá presne hodinu. Pozná presné časy $t_1, \ldots, t_n$ (v sekundách od začiatku zmeny), kedy sa na neho príde pozrieť stavbyvedúci. Napíšte (ako pseudokód alebo kus programu) algoritmus, ktorý vypočíta, koľko najmenej miešačiek betónu musí Sergej spraviť, aby ho stavbyvedúci zakaždým videl, ako pracuje. Dokážte správnosť vášho programu a odhadnite jeho časovú zložitosť.
Príklad: ak príde kontrola o 9:47, 10:12 a 13:50, tak stačia dve miešačky: napr. od 9:30 do 10:30 a od 13:47 do 14:47.

4.2 Riešenie

Toto je v podstate tá istá úloha ako prvá úloha o zastávkach v dedine: Máme niekoľko bodov a treba ich pokryť najmenším počtom intervalov.
Vieme teda spraviť napríklad nasledovnú úvahu: Uvažujme prvý príchod stavbyvedúceho. Vtedy musí Sergej pracovať. Kedy začal? Zjavne sa oplatí začať tesne pred príchodom stavbyvedúceho – skorším začiatkom nemáme čo získať, môžeme len stratiť. Takto pokryjeme niekoľko prvých návštev stavbyvedúceho. Ak nám ešte ostali nejaké nepokryté, zopakujeme celú úvahu odznova.
Celý algoritmus teda vyzerá nasledovne: Ak časy $t_1, \ldots, t_n$ náhodou nie sú na vstupe usporiadané, usporiadame ich. Na toto potrebujeme čas $\Theta(n \log n)$. Zvyšok algoritmu je lineárny: prechádzame časy, pričom si pamätáme, dokedy ich už máme pokryté. Vždy, keď narazíme na nepokrytý čas, začneme novú miešačku.

```
def najmenej_miešačiek(časy T):
    sort(T)
    počet_miešačiek = 0
    koniec_pokrytia = min(T) - 1
    pre každé t v T:
        # (v usporiadanom poradí, začínajúc najmenším časom)
        ak t > koniec_pokrytia:
            počet_miešačiek += 1
            koniec_pokrytia = t + 3600
    return počet_miešačiek
```

5 Pohľad zvonka: jednozubé pole [10 alebo 5 bodov]

5.1 Zadanie

Pole $A[0..n-1]$ voláme *jednozubé*, ak existuje nanajvýš jedno i také, že $A[i] > A[i+1]$. (Každé jednozubé pole je teda takmer usporiadané vzostupne.) Eleonóra potrebuje často usporadúvať veľké polia. Navyše však vie, že veľa spomedzi jej polí je jednozubých. Preto sa jej nepáči MergeSort, ktorý má na každom vstupe veľkosti n časovú zložitosť $\Theta(n \log n)$. Rozhodnite a dokážte, či existuje triediaci algoritmus s nasledujúcimi dvomi vlastnosťami:

- Na každom možnom poli veľkosti n spraví $O(n \log n)$ krokov.
- Ak je ale vstupné pole jednozubé, bude časová zložitosť nášho algoritmu len $O(n)$.

(Ľahšia verzia za 5 bodov: zmeňte si v predchádzajúcej vete slovo „jednozubé“ na slovo „usporiadané“.)

5.2 Riešenie

Taký algoritmus existuje, vyzerá napríklad nasledovne:

1. V $O(n)$ overíme, či je vstupné pole jednozubé.
2. Ak je jednozubé, špeciálnym algoritmom ho usporiadame v $O(n)$.
3. Ak nie, použijeme MergeSort, čím ho usporiadame v $\Theta(n \log n)$.

Prvý krok vieme spraviť priamo podľa definície jednozubého poľa: prejdeme ho a spočítame zuby. Navyše sa nám oplatí pre jednozubé pole si zapamätať miesto, kde bol zub. Prečo? Predstavme si nejaké jednozubé pole, napr. $(4, 7, 13, 25, 47, 1, 12, 32, 42)$. Keď ho rozstrihneme na mieste, kde je zub, dostaneme dve usporiadané polia $(4, 7, 13, 25, 47)$ a $(1, 12, 32, 42)$. A z tých už vieme ľahko v $O(n)$ zostrojiť výsledok: stačí na ne zavolať procedúru Merge z triedenia MergeSort.

```
def nájdi_zub(pole A[1..N]):
    zub = 0
    pre i od 1 po N-1:
        ak A[i] > A[i+1]:
            ak zub == 0: zub = i
            inak: return VIACZUBÉ
```

```
    ak zub == 0: return USPORIADANÉ
    inak: return zub

def usporiadaj(pole A[1..N]):
    zub = nájdi_zub(A)
    ak zub == USPORIADANÉ: return A
    ak zub == VIACZUBÉ: return MergeSort(A)
    return Merge( A[1..zub], A[zub+1..N] )
```

6 Pohľad zvonka: strojový čas [15 bodov]

6.1 Zadanie

Kleofáš má na internete pripojenú mašinu. Rozhodol sa zarábať tým, že ju bude prenajímať na rôzne účely, podobne ako to vo veľkom robí Amazon so svojim Elastic Compute Cloud.
Teraz mu prichádzajú rôzne požiadavky. Každú požiadavku tvoria dva timestampy $t_1 \leq t_2$, hovoriace, že zákazník má záujem používať mašinu v uzavretom intervale $[t_1, t_2]$. Ak Kleofáš ešte vie požiadavku prijať (t.j. požadovaný interval má prázdny prienik so všetkými skôr prijatými požiadavkami), prijme ju, inak ju odmietne.
Ako pseudokód alebo kus programu napíšte algoritmus, ktorý bude čo najefektívnejšie spracúvať požiadavky. Zdôvodnite jeho správnosť a odhadnite časovú zložitosť v závislosti od celkového počtu požiadaviek n.

Príklad: pre požiadavky $[3, 7]$, $[12, 15]$, $[2, 10]$, $[4, 13]$, $[9, 10]$ by prijal 1. a 2., odmietol 3. a 4., a prijal 5. požiadavku.

6.2 Riešenie

Požiadavky, ktoré sme už prijali, budeme mať uložené vo vyvažovanom binárnom vyhľadávacom strome, usporiadané podľa času, kedy začínajú. Keď teraz príde nová požiadavka $[a_1, a_2]$, nájdeme v čase $O(\log n)$ poslednú prijatú požiadavku $[b_1, b_2]$ takú, že $b_1 < a_1$, a (bezprostredne za ňou nasledujúcu) prvú prijatú požiadavku $[c_1, c_2]$ takú, že $c_1 \geq a_1$. Požiadavku $[a_1, a_2]$ môžeme prijať vtedy a len vtedy, ak $b_2 < a_1$ a $a_2 < c_1$. V takom prípade ju v čase $O(\log n)$ pridáme do stromu. Celková časová zložitosť algoritmu je $\Theta(n \log n)$.
Nasleduje kompletné riešenie v C++. Aby sme nemuseli ošetrovať okrajové prípady, pridáme si na začiatku medzi prijaté požiadavky dve, ktoré budú slúžiť ako zarážky.

```
#include <iostream>
#include <set>
#define INF 2012345678

class poziadavka {
    public: int zaciatok, koniec;
    poziadavka(int z, int k) : zaciatok(z), koniec(k) {} // konstruktor
};

bool operator< (const poziadavka A, const poziadavka B) { return A.zaciatok < B.zaciatok; }

std::set<poziadavka> prijate;

int main() {
    prijate.insert(poziadavka(-INF,-INF)); prijate.insert(poziadavka(INF,INF)); // zarazky

    int t1, t2;
    while (std::cin >> t1 >> t2) {
        poziadavka A(t1,t2);

        // najdeme nasledujucu prijatu C a predchadzajucu prijatu B
        std::set<poziadavka>::iterator it = prijate.lower_bound(A);
        poziadavka C = *it;
        --it;
        poziadavka B = *it;

        // ak sa A zmesti medzi ne, prijmeme ju
        if (B.koniec < A.zaciatok && A.koniec < C.zaciatok) {
            std::cout << "prijata\n";
            prijate.insert(A);
        } else {
            std::cout << "odmietnuta\n";
        }
    }
}
```