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

1.1 Zadanie

1. Ak v 99-prvkovom poli 470× vymením dva rôzne, náhodne vybrané prvky, môžem na konci dostať ľubovoľnú permutáciu.
2. Z haldy sa dá vyrobiť binárny vyhľadávací strom (nie nutne vyvážený) v čase $O(n)$.
3. Ak má n-vrcholový binárny strom aspoň $n/2$ listov, tak je jeho hĺbka $O(\log n)$.
4. Každý binárny strom hĺbky $\leq 2\log_2 n$ je nutne vyvážený (v zmysle def. pre AVL stromy).
5. V halde s minimom v koreni pre ľubovoľné prvky x, y platí: ak $x > y$, tak x je aspoň tak hlboko ako y.
6. Pole obsahujúce n celých čísel z množiny $\{0, 1, \ldots, 47n - 1\}$ vieme usporiadať v čase asymptoticky lepšom ako $n \log n$.
7. Pole obsahujúce n celých čísel z množiny $\{0, 1, \ldots, n^{47} - 1\}$ vieme usporiadať v čase asymptoticky lepšom ako $n \log n$.
8. Keď do poľa veľkosti 1 000 000 zahešujem 10 000 prvkov, pravdepodobnosť aspoň jednej kolízie je väčšia ako 1/2.

1.2 Riešenie

1. FALSE. Výsledná permutácia bude určite párna (t.j. bude mať párny počet inverzií). Polovica všetkých možných permutácií sa teda na výstupe nemá ako objaviť.

   Obľúbená chyba: prečítať v zadaní slovo „470“ ako „najviac 470“. Potom je správnou odpoveďou, že ľubovoľnú permutáciu vieme vyrobiť na nanajvýš 98 výmen.

2. FALSE. Keby sme toto vedeli, vedeli by sme aj triediť porovnávaním v čase rádovo lepšom ako $n \log n$: z neusporiadaného poľa vyrobíme v čase $O(n)$ haldu (to vieme), potom použijeme konštrukciu zo zadania a na záver výsledný strom v čase $O(n)$ vypíšeme in-order (aj to vieme).

3. FALSE. Neplatí napr. pre strom, ktorý vznikne, keď bez vyvažovania vkladáme postupnosť 2, 1, 4, 3, 6, 5, 8, 7, ...

4. FALSE. Existujú malé protipríklady, napr. strom, ktorý vznikne, keď bez vyvažovania vložíme postupnosť 1, 2, 3, 4. Neplatí to ani vo všeobecnosti, napr. pre 2^n vrcholov môžeme mať strom, ktorého koreň má prázdny pravý podstrom a v ľavom podstrome je úplný binárny strom tvorený $2^n - 1$ vrcholmi.

5. FALSE. Najmenší protipríklad má 4 vrcholy, vznikne napr. postupným vložením 1, 2, 4 a 3 do haldy.

6. TRUE. Zafunguje asi hociktorý špecifický sort, napr. CountSort.

7. TRUE. Zapísané v n-kovej sústave majú tieto čísla ≤ 47 cifier, časová zložitosť RadixSortu v n-kovej sústave bude teda $O(n)$.

   Obľúbená chyba: Tvrdiť, že to nejde, lebo nejaký špecifický postup (zväčša CountSort) nezafunguje.

8. TRUE. Vyplýva z narodeninového paradoxu.

2 Pohľad zvnútra: stromy a haldy [5+5+5 bodov + 10 bonus]

2.1 Zadanie

a) Nakreslite *všetky* možné binárne stromy, pre ktoré platí: množina vrcholov je $\{1, 2, 3, 4, 5\}$, je to binárny vyhľadávací strom a zároveň aj minimová halda (t.j., každý vrchol spĺňa podmienku haldy). Zdôvodnite, prečo iné nevyhovujú.

b) Binárnu haldu s n prvkami vieme uložiť do poľa $A[0..n-1]$ tak, že pre vrchol uložený na indexe i sú jeho synovia (ak existujú) uložení na indexoch $2i + 1$ a $2i + 2$. Aké veľké pole (presne!) by sme potrebovali, ak by sme doň chceli vedieť rovnakým spôsobom uložiť ľubovoľný n-vrcholový binárny vyhľadávací strom?

c) Aké veľké pole (asymptoticky) by sme potrebovali v časti b), ak by bolo zaručené, že ten strom má hĺbku $\leq 2\log_2 n$?

d) BONUS: Aké veľké pole by sme potrebovali v časti b), ak by bolo zaručené, že ten strom je vyvážený?

2.2 Riešenie

a) Žiaden vrchol nesmie mať ľavého syna, ten by totiž musel od otca byť menší a väčší zároveň. Strom má teda tvar spájaného zoznamu idúceho doprava dodola. Zhora dole v ňom postupne musia byť hodnoty od 5 do 1.

b) Najhorší prípad je zovšeobecnením stromu z podúlohy a). Jeho vrcholy by boli uložené na indexoch 0, 2, 6, 14, ... Vo všeobecnosti pre strom s n vrcholmi bude posledný index $2^n - 2$, a teda potrebná veľkosť poľa je $2^n - 1$.

c) Už vieme, že posledný vrchol v hĺbke k má index $2^k - 2$. Ak teda máme posledný vrchol zrovna v hĺbke $2\log_2 n$, jeho index bude zhruba $2^{2\log_2 n} = n^2$.

d) Pre vyvážené stromy si vieme odvodiť, že strom hĺbky h má aspoň $F_{h+3} - 1$ vrcholov. Ak teda máme vyvážený strom so zhruba $n = \phi^k$ vrcholmi, jeho hĺbka je v najhoršom prípade $k - 3$. Na takto hlboký strom potrebujeme pole veľkosti 2^{k-3}, čo je rádovo to isté ako 2^k. Keď dosadíme $k = \log_\phi n$ a vhodne upravíme, dostávame pre veľkosť stromu výsledok $n^{\log_\phi 2} \approx n^{1.4404}$. To je o chlp menej ako $n\sqrt{n}$.

   Obľúbená chyba: Tvrdiť, že vyvážený strom musí mať hĺbku $\lceil \log_2 n \rceil + O(1)$.

3 Pohľad zvnútra: Niekoľko najväčších [6+14 bodov]

3.1 Zadanie

Neusporiadané pole $A[0..n-1]$ obsahuje prvky, ktoré vieme porovnávať. Chceme nájsť jeho $k = \lceil \sqrt{n} \rceil$ najväčších prvkov.
a) Dokážte, že každý program riešiaci túto úlohu musí mať časovú zložitosť $\Omega(n)$, t.j., lineárnu alebo horšiu.
b) Napíšte (naozaj do detailov) čo najefektívnejší program riešiaci túto úlohu. Odhadnite jeho časovú zložitosť.

Hint: Existuje veľa optimálnych riešení. Jedno vhodne použije haldu. Druhé si pole naseká na vhodne dlhé úseky. Tretie sa použitiu celého triedenia vyhne ešte iným spôsobom.

3.2 Riešenie

a) Dolný odhad priamo vyplýva z toho, že sa program musí aspoň raz pozrieť na každé číslo na vstupe.

Podrobnejší dôkaz sporom. Predpokladajme, že ste taký program napísali. Potom existuje nejaké dostatočne veľké n také, že pre ľubovoľné pole veľkosti n existuje prvok, na ktorý sa nepozrie. Spustím teda ten váš program na poli veľkosti n obsahujúcom samé hodnoty 1. Následne si vyberiem niektorý prvok, na ktorý sa nepozrel. Ak ho zahrnul medzi vybratých k najväčších, zmením ho na 0, inak ho zmením na 2. A mám vstup, pre ktorý dal váš program nesprávny výstup.

Veľmi obľúbená chyba: zabudnúť na možnosť, ktorú som vyššie vyriešil zmenou na 0. Veľa ľudí argumentovalo len tým, že keď sa na nejaké prvky nepozriem, môže zrovna na tých pozíciách byť maximum, a teda nedám správnu odpoveď. Nášmu hypotetickému programu ale nič nebráni vrátiť na výstupe aj prvky, na ktoré sa nepozrel. Čo ak sa napr. dohodneme, že "nájsť k najväčších prvkov" znamená nájsť ich indexy?

(Za toto som strhal max. bod, keďže je to len technický detail.)

b) Na rozumne veľa bodov stačilo implementovať ľubovoľné efektívne triedenie. Existujú však aj riešenia s lineárnou časovou zložitosťou.

Prvé možné riešenie: V lineárnom čase z poľa spravíme haldu s maximom v koreni. Následne k-krát z tejto haldy vyberieme najväčší prvok. Časová zložitosť $\Theta(n + \sqrt{n}\log(n)) = \Theta(n)$. Implementovať stačí bublanie prvku dodola, to sa použije aj pri výrobe haldy, aj pri následnom vyberaní.

Druhé možné riešenie: Rozdelíme pole na k úsekov dĺžky zhruba k. V každom úseku nájdeme maximum. Následne k-krát zopakujeme: nájdi najväčšie spomedzi maxím úsekov, to vyber, a následne v danom úseku nájdi nové maximum. Časová zložitosť $\Theta(k^2) = \Theta(n)$.

Tretie možné riešenie: Existuje lineárny algoritmus na nájdenie k-teho najväčšieho prvku. Jeho použitím prerozdelíme pole tak, aby na k-tej pozícii od konca bol správny prvok a napravo od neho samé väčšie alebo rovné prvky. Tie potom tvoria práve hľadanú množinu. (Základná myšlienka algoritmu: robíme to isté ako QuickSort, vždy sa však rekurzívne zavoláme len na tú časť poľa, ktorá obsahuje index, ktorý nás zaujíma.)

Implementácia druhého riešenia v Pythone:

```
from random import randint
from math import ceil, sqrt

N = 12345678
A = [ randint(0,2**30-1) for n in range(N) ]

K = int(ceil(sqrt(N)))

# nasekame pole na useky dlzky sqrt(N)
useky = []
for k in range(K): useky.append( A[ k*K : (k+1)*K ] )

# v kazdom useku najdeme maximum
def najdi_maximum(usek):
    for i in range(len(usek)):
        if usek[i]>usek[-1]: usek[i],usek[-1] = usek[-1],usek[i]

for usek in useky: najdi_maximum(usek)

# postupne vyrabame odpoved
odpoved = []
while len(odpoved)<K:
    best = 0
    for k in range(K):
        if useky[k][-1] > useky[best][-1]: best=k
    odpoved.append( useky[best][-1] )
    useky[best].pop()
    najdi_maximum( useky[best] )
    if len(useky[-1])==0: useky.pop() # posledny usek sa nam moze minut

print odpoved
```

4 Pohľad zvonka: Porovnaj tri naraz [8+12 bodov]

4.1 Zadanie

V poli $A[0..n-1]$, pričom $n \geq 3$, máme navzájom rôzne prvky neznámeho typu. Prvky majú navzájom rôzne veľkosti, tie ale nepoznáme. Chceli by sme pole usporiadať podľa veľkosti prvkov. Jediný spôsob, ako vieme prvky porovnávať, je ale

zvláštny: Máme funkciu, ktorej dáme tri *rôzne* indexy do poľa a dozvieme sa (v konštantnom čase), ktorý z tých troch prvkov je najmenší, ktorý je prostredný a ktorý najväčší.
a) Napíšte program, ktorý túto úlohu vyrieši s optimálnou časovou zložitosťou.
b) Dokážte, že asymptotická časová zložitosť vášho programu je optimálna.
(V oboch podúlohách je OK sa bez rozpisovania odvolávať na algoritmy a dôkazy z prednášok.)

4.2 Riešenie

a) Použijeme ľubovoľný optimálny algoritmus pre triedenie porovnávaním, napr. MergeSort. Jediné, čo potrebujeme navyše, je funkcia pre porovnanie dvoch prvkov. Tú vieme spraviť tak, že porovnáme tie dva, ktoré nás zaujímajú, a nejaký tretí, je jedno aký (len nech je iný). Ako súčasť odpovede sa dozvieme porovnanie našich dvoch prvkov, to dáme na výstup.

   Obľúbená chyba: Odbiť to celé s tým, že „urobím MergeSort deliaci pole na tri časti". OK, to si predstaviť viem, ale čo sa potom stane pri mergovaní, keď spájam tri usporiadané polia a v jednom z nich sa mi už minuli prvky? Ako spojím tie zvyšné dva?

b) Vieme, že každý algoritmus triedenia porovnávaním má časovú zložitosť $\Omega(n \log n)$. Ak by existoval efektívnejší algoritmus riešiaci zadanú úlohu, nutne by existoval aj rovnako efektívny algoritmus triedenia porovnávaním – totiž každé porovnanie trojice vieme naskladať z troch normálnych porovnaní.

   Obľúbená chyba: zamotať sa v argumentácii. Tu to chcelo jasne vysloviť a dokázať implikáciu: AK by sme vedeli riešiť našu úlohu efektívnejšie, TAK by sme vedeli triediť *porovnávaním* efektívnejšie. Veľa ľudí namiesto toho len inými slovami zopakovalo riešenie podúlohy a). To NIE JE to, čo sme tu chceli. A argument „podúlohu a) sme vyriešili MergeSortom a o tom vieme, že je optimálny" tiež nemôžete použiť – MergeSort je optimálny pre klasické triedenie porovnávaním, vy ste teraz mali *dokázať*, že je optimálny aj pre riešenie tejto novej úlohy.

5 Pohľad zvonka: DNA [5 × 5 bodov + 5 bonus]

5.1 Zadanie

Dve vlákna DNA na seba pasujú, ak majú rovnakú dĺžku a obsahujú komplementárne dusíkaté bázy (C ku G, A ku T). Teda napr. $GATTACA$ a $CTAATGT$ na seba pasujú. Toto môžeme zovšeobecniť. *Nekompatibilita* vláken α a β bude počet elementárnych zmien potrebných na to, aby α a β na seba pasovali. Povolené elementárne zmeny budeme uvažovať nasledovné: i) vynechanie niektorej bázy z vlákna, ii) vloženie novej bázy kamkoľvek do vlákna, iii) zmena ľubovoľnej bázy na inú.
Napr. nekompatibilita vláken $\alpha = GGGAGGT$ a $\beta = CTCCCCT$ je 3. Jeden optimálny spôsob úprav: z α vynecháme 4. bázu (A), potom do α na druhú pozíciu vložíme novú bázu A, a nakoniec v β poslednú bázu zmeníme z T na A. Dostaneme $\alpha' = GAGGGGT$ a $\beta' = CTCCCCA$ a tie už na seba pasujú.
V tejto úlohe sa skúsite dopracovať k efektívnemu algoritmu, ktorý pre dané dva reťazce spočíta ich nekompatibilitu.

a) Dokážte, že sa nekompatibilita žiadnej dvojice reťazcov nezmení, ak zakážeme elementárne operácie typu ii) – dovolíme teda len mazať bázy a meniť existujúce bázy na iné.
b) Napíšte (ako pseudokód alebo kus programu) rekurzívny algoritmus skúšajúci všetky možnosti ako postupne upravovať vlákna pomocou operácií typu i) a iii). Vhodné je začať napr. od konca: Ako môže vyzerať optimálna postupnosť úprav ak na seba posledné znaky pasujú? A ako, keď nepasujú?
c) Pridajte do predchádzajúceho algoritmu memoizáciu tak, aby vznikol algoritmus s časovou zložitosťou polynomiálnou od počtu vecí n. Odhadnite jeho časovú zložitosť.
d) Uveďte ekvivalentný algoritmus, ktorý túto úlohu rieši pomocou dynamického programovania.
e) Dokážte alebo vyvráťte správnosť pažravého algoritmu: Ak je niektorý reťazec prázdny, zmažem všetky písmená druhého. Ak na seba prvé písmená oboch pasujú, oba zmažem, ale nepočítam to medzi operácie. Ak nie a reťazce sú rôznej dĺžky, zmažem prvé písmeno dlhšieho. A ak na seba nepasujú prvé písmená dvoch rovnako dlhých reťazcov, zmením jedno z nich tak, aby už pasovali.
f) BONUS: V praxi čakáme, že budeme spracúvať hlavne dvojice vláken, ktoré na seba skoro pasujú. Ktorý z algoritmov v častiach c), d) a e) je najvhodnejší a prečo?

5.2 Riešenie

a) Majme ľubovoľné optimálne riešenie, v ktorom sme v nejakom kroku použili operáciu ii). BUNV nech sme vložili nejaký znak x do reťazca α. Zjavne na konci tento znak niekde v α bude a niekde v druhom reťazci β bude znak y, ktorý mu zodpovedá. Ak by sme namiesto vloženia x do α v poslednom kroku úprav zmazali z β znak y, dostaneme rovnako dobré (teda tiež optimálne) riešenie, v ktorom už je ale o jedno použitie operácie ii) menej.

   Z toho zjavne vyplýva, že vždy existuje optimálne riešenie, pri ktorom nevkladáme nové písmená.

b) Ak na seba posledné znaky reťazcov pasujú, každé optimálne riešenie ich zjavne nechá nedotknuté. Ak nepasujú, musí prísť k nejakej zmene: buď jeden z nich zmažeme, alebo jeden (je jedno, ktorý) zmeníme, aby pasoval na druhý.

   Program v Pythone:

```
from sys import stdin
A = stdin.readline()
B = stdin.readline()

def pasuje(x,y): return x+y in ['CG','GC','AT','TA']

def solve(a,b):
    global A,B
    # najdi najlepsie riesenie pre A[:a] a B[:b]
    if a==0: return b
    if b==0: return a
    if pasuje(A[a-1],B[b-1]): return solve(a-1,b-1)
    return 1+min( solve(a-1,b), solve(a,b-1), solve(a-1,b-1) )

print solve(len(A),len(B))
```

Obľúbená chyba: Pár ľudí skúšalo mazať len z dlhšieho reťazca namiesto z oboch. Táto snaha o optimalizáciu ich paradoxne stála body, keďže s touto úpravou je už riešenie nekorektné. Viď napr. $\alpha = CCTT$ a $\beta = AAT$. V tejto situácii na seba posledné znaky α a β nepasujú a k optimálnemu riešeniu vedie len voľba zmazať posledný znak β.

c) V nasledovnej implementácii používame na memoizáciu hešovaciu tabuľku. O chlp lepšie by bolo použiť dvojrozmerné pole. To použijeme v nasledovnej podúlohe.

```
from sys import stdin
A = stdin.readline()
B = stdin.readline()
memo = {}

def pasuje(x,y): return x+y in ['CG','GC','AT','TA']

def solve(a,b):
    global A,B,memo
    # najdi najlepsie riesenie pre A[:a] a B[:b]
    if a==0: return b
    if b==0: return a
    if (a,b) in memo: return memo[(a,b)]
    if pasuje(A[a-1],B[b-1]): memo[(a,b)] = solve(a-1,b-1)
    else: memo[(a,b)] = 1+min( solve(a-1,b), solve(a,b-1), solve(a-1,b-1) )
    return memo[(a,b)]

print solve(len(A),len(B))
```

d) Mechanicky prepíšeme program z predchádzajúcej podúlohy. Jediné, čo si potrebujeme rozmyslieť, je poradie, v ktorom hodnoty `memo[a][b]` počítať. A tu teda nie je moc čo rozmýšľať, keďže priamočiare poradie po riadkoch funguje.

```
from sys import stdin
A = stdin.readline()
B = stdin.readline()
memo = [ [ 2**30 for b in range(len(B)+1) ] for a in range(len(A)+1) ]

def pasuje(x,y): return x+y in ['CG','GC','AT','TA']

for a in range(len(A)+1):
    for b in range(len(B)+1):
        if a==0:
            memo[a][b] = b
            continue
        if b==0:
            memo[a][b] = a
            continue
        if pasuje(A[a-1],B[b-1]):
            memo[a][b] = memo[a-1][b-1]
        else:
            memo[a][b] = 1+min( memo[a-1][b], memo[a][b-1], memo[a-1][b-1] )

print memo[ len(A) ][ len(B) ]
```

e) Uvažujme reťazce $\alpha = TTTCCC$ a $\beta = CAA$. Pažravý algoritmus najskôr 3× zahodí prvé písmeno α, čím dostane $\alpha' = CCC$ a $\beta = CAA$. Teraz na seba ani jedna dvojica nepasuje, takže spravíme ďalšie tri zmeny, čo je dokopy 6.

Toto riešenie ale nie je optimálne. Lepšie je napr. zmazaním štyroch znakov z konca α vyrobiť $\alpha' = TT$ a zmazaním jedného znaku zo začiatku β vyrobiť $\beta' = AA$. Tento postup potrebuje len päť operácii. (Rozmyslite si, či je optimálny.)

f) Ak čakáme, že bude potrebných málo zmien, bude efektívnejšia memoizovaná rekurzia, keďže väčšinu stavov, ktoré bude dynamické programovanie vyhodnocovať, rekurzia nikdy nenavštívi.

Obľúbená chyba: medzi výhody memoizácie uviesť menšie pamäťové nároky – to nemusí byť pravda, keďže riešenie pomocou dynamického programovania vieme upraviť tak, aby si pamätalo len dva riadky tabuľky.