Algorytmy i Struktury Danych
Kolokwium 3 (3. VI 2020)

Format rozwiązań

Rozwiązanie każdego zadania musi składać się z opisu algorytmu (wraz z uzasadnieniem poprawności i oszacowaniem złożoności obliczeniowej) oraz jego implementacji. Zarówno opis algorytmu jak i implementacja powinny się znajdować w tym samym pliku Pythona (rozszerzenie `.py`). Opis powinien być na początku pliku w formie komentarza (w pierwszej linii w komentarzu powinno być imię i nazwisko studenta). Implementacja musi być zgodna z szablonem kodu źródłowego dostarczonym wraz z zadaniem. Nie dopuszczalne jest w szczególności:

1. zmiana nazwy funkcji implementującej algorytm lub listy jej argumentów,
2. modyfikacja testów dostarczonych wraz z szablonem,
3. wypisywanie na ekranie jakichkolwiek napisów innych, niż wypisywane przez dostarczony kod (ew. napisy dodane na potrzeby diagnozowania błędów należy usunąć przed wysłaniem zadania).

Dopuszczalne jest natomiast:

1. korzystanie z następujących elementarnych struktur danych: krotka, lista, kolejka `collections.deque`,
2. korzystanie z wbudowanych algorytmów sortowania,
3. korzystanie ze struktur danych dostarczonych razem z zadaniem.

Wszystkie inne algorytmy lub struktury danych (w tym słowniki) wymagają implementacji. Dopuszczalne jest oczywiście implementowanie dodatkowych funkcji pomocniczych w pliku z szablonem rozwiązania. Jeśli ktoś zaimplementuje standardowe drzewo BST, to może w analizie zakładać, że złożoność operacji na nim jest rzędu $O(\log n)$.

Zadania niezgodne z powyższymi ograniczeniami otrzymają ocenę 0 pkt. Rozwiązania w innych formatach (np. `.PDF`, `.DOC`, `.PNG`, `.JPG`) z definicji nie będą sprawdzane i otrzymają ocenę 0 punktów, nawet jeśli będą poprawne.

Proszę pamiętać, że rozwiązania trochę wolniejsze niż oczekiwane, ale za to poprawne, mają szanse na otrzymanie 1 punktu. Rozwiązania próbujące osiągnąć jak najlepszą złożoność, ale zaimplementowane błędnie otrzymają 0 punktów. Proszę mierzyć siły na zamiary!

Testowanie rozwiązań

Żeby przetestować rozwiązanie zadania 1 należy wykonać:

```
python3 zad1.py
```

Żeby przetestować rozwiązanie zadania 2 należy wykonać:

```
python3 zad2_testy.py
```

[2pkt.] Zadanie 1.

Szablon rozwiązania: `zad1.py`
Drzewo przedziałowe: `inttree.py`

Dana jest lista `I` przedziałów domkniętych $[a_1, b_1]$, $[a_2, b_2], \ldots, [a_n, b_n]$. Napisz funkcję `intervals( I )`, która oblicza dla każdego $i \in \{1, 2, \ldots, n\}$ długość najdłuższego ciągłego przedziału, który można osiągnąć sumując wybrane przedziały spośród pierwszych i przedziałów z listy. Funkcja powinna być możliwie jak najszybsza.

Przedziały reprezentowane są w postaci listy par. Funkcja powinna zwrócić listę liczb, w której i-ty element to długość poszukiwanego najdłuższego przedziału zbudowanego z pierwszych i elementów wejścia. Na przykład, dla listy odcinków:

```
[(1,3),(5,6),(4,7),(6,9)]
```

rozwiązaniem jest lista `[2, 2, 3, 5]` zawierająca długości odcinków: `[1, 3]`, `[1, 3]`, `[4, 7]` oraz `[4, 9]`.

Drzewo przedziałowe

W pliku `inttree.py` została Państwu dostarczona elementarna implementacja drzewa przedziałowego, tak jak było ono opisane na wykładzie. Dostępne są następujące funkcje:

1. `tree(A)` – stwórz nowe drzewo przedziałowe; przechowywane przedziały muszą być postaci $[a, b]$, gdzie liczby a i b występują w tablicy `A`; tablica `A` musi być posortowana rosnąco i nie może zawierać powtórzeń. Funkcja zwraca korzeń drzewa `T`. Złożoność: $O(|A|)$.
2. `tree_insert(T, (a,b))` – wstaw do drzewa `T` (reprezentowanego przez korzeń) przedział $[a, b]$. Złożoność: $O(\log |A|)$.
3. `tree_remove(T, (a,b))` – usuń z drzewa `T` (reprezentowanego przez korzeń) przedział $[a, b]$ (jeśli przedziału nie było w drzewie, to nic nie robi). Złożoność: $O(\log |A|)$.
4. `tree_intersect( T, x )` – zwraca listę przedziałów z drzewa `T` (reprezentowanego przez korzeń), które zawierają punkt x (niektóre przedziały mogą występować na liście dwukrotnie). Złożoność: $O(k + \log |A|)$, gdzie k to liczba zwróconych przedziałów
5. `tree_print( T )` – funkcja pomocnicza wypisująca zawartość drzewa (proszę zajrzeć do kodu, żeby zobaczyć co wypisuje).

Nie ma obowiązku korzystać z tego drzewa—można zaimplementować własne, lub użyć innej struktury danych. Jeśli ktoś zaimplementuje **klasyczne drzewo BST** to może je analizować tak, jakby operacje na nim miały **złożoność $O(\log n)$**}.

Przykład wykorzystania drzewa przedziałowego

```
from inttree import *
T = tree([1, 2, 3, 4, 5])
tree_insert(T,(1, 4))
tree_insert(T,(2, 5))
tree_print(T)
tree_remove(T,(1, 4))
tree_print(T)
tree_insert(T,(1, 3))
print(tree_intersect(T, 3))
```

[2pkt.] Zadanie 2.

Szablon rozwiązania: `zad2.py`
Plik do uruchamiania testów: `zad2_testy.py`

Dana jest tablica haszująca o rozmiarze `N` elementów oparta o adresowanie otwarte i liniowe rozwiązywanie konfliktów (z krokiem 1; dokładny kod wstawiania do tablicy znajduje się w funkcji `insert` w pliku `zad2_testy.py`). Każdy element tablicy jest obiektem zawierającym klucz (napis) oraz pole wskazujące, czy element jest zajęty:

```
class Node:
  def __init__(self, key = None, taken = False):
    self.key = key
    self.taken = taken
```

Ponadto dana jest funkcja haszująca `h(key)` przyjmująca klucz i zwracająca indeks w tablicy (w przedziale `0` do `N-1`; w pliku `zad2.py` wpisana jest konkretna wartość `N`, ale w ogólności nie należy traktować `N` jako stałej).

Niestety, w wyniku ataku komputerowego dokładnie jeden elementy tablicy haszującej zostały zmodyfikowany poprzez zmianę wartości pola `taken` na `False` oraz pola `Klucz` na `None`. Proszę zaproponować i zaimplementować funkcję:

```
def recover(hash_tab):
  ...
```

która sprawdza, czy w tablicy haszującej przekazanej przez argument wszystkie elementy z polem `taken` równym `True` mogą być poprawnie znalezione (procedura wyszukiwania w tablicy haszującej to `find` z pliku `zad2_testy.py`). Jeżeli tak nie jest, funkcja powinna „naprawić" tablicę w taki sposób, by wszystkie elementy (w których `taken == True`) były osiągalne. W każdym przypadku funkcja powinna zwrócić tablicę (potencjalnie naprawioną) jako wynik. Funkcja może używać jedynie stałego (niezależnego od `N`) rozmiaru dodatkowej pamięci operacyjnej (a więc powinna działać w miejscu - nie można dodawać pól do elementów tablicy lub utworzyć nowej tablicy haszującej). Zaproponowane rozwiązanie powinno być możliwie jak najszybsze.