Zadania przed kolokwium I - rozwiązania

Zadanie 1
Drzewo przedziałowe:k

Zadanie 2
Posortować mniejszy zbiór, a potem sprawdzać dla każdego elementu z większego zbioru binary searchem, czy jest w mniejszym zbiorze. Złożoność: O(mlog(m) + nlog(m)) = O(log(m) * (m + n))

Zadanie 3
Nieoptymalnie: sprawdzać elementy po kolei metodą pop() - O(k*log(n)).
Optymalnie:

Zauważmy, że z definicji kopca minimum każdy korzeń poddrzewa jest mniejszy od wszystkich swoich dzieci. Dotyczy to zarówno korzenia całego kopca, jak i wszystkich poddrzew. Wynika z tego, że jeżeli korzeń jest >= x, to całe poddrzewo musi być >= x (skoro nic nie może być tam mniejszego, to może być tylko większe lub równe).

Zauważmy także, że nie interesuje nas wartość wierzchołków przed k-tym. Co więcej, nie interesuje nas nawet wartość k-tego wierzchołka - interesuje nas tylko jego relacja z x, czy jest od niego mniejszy, czy większy lub równy.

Można przejść po drzewie kopca (przydatne są tutaj funkcje pomocnicze get_left_child(i), get_right_child(i) oraz get_parent(i)) przeszukując dzieci wszystkich wierzchołków o wartości mniejszej od x, aż:

a) znajdziemy przynajmniej k wierzchołków o wartości < x; jeżeli tak się stanie, to k-ty wierzchołek musiał być mniejszy od x, a więc zwracamy fałsz
b) wyczerpiemy wierzchołki o wartości < x, zanim znaleźliśmy ich k; jeżeli tak się stanie, to dalsze wierzchołki muszą być >= x, a zatem k-ty też, więc zwracamy prawdę

Złożoność: sprawdzamy tylko dzieci wierzchołków o wartości < x, a tych jest co najwyżej k (jeżeli byłoby więcej, to przerywamy zgodnie z opisem powyżej); każde może mieć co najwyżej 2 dzieci, więc odwiedzamy co najwyżej 3k wierzchołków, a więc mamy O(k).

Zadanie 4
Wystarczy posortować liczby i brać zawsze pary liczb: najbardziej po lewej + najbardziej po prawej. W ten sposób zawsze sumujemy najmniejszą i największą liczbę. Należy jeszcze udowodnić, że taki algorytm zachłanny da prawidłowy wynik, ale jest to proste, bo wystarczy pokazać, że suma dowolnej pary “wewnętrznych” elementów będzie mniejsza lub równa sumie pierwszej pary, najbardziej “zewnętrznej” (arr[0] + arr[-1]). Złożoność: O(n)

Zadanie 5
Fajnie byłoby użyć wyszukiwania binarnego, ale problemem jest to, że nie znamy prawej granicy liczb. Można ją szybko znaleźć, stosując prostą iterację: skaczemy o i do przodu, sprawdzamy czy jesteśmy na wartości None lub wartości > x - jeżeli tak, to ta pozycja jest naszym prawym końcem i zaczynamy binary search, w przeciwnym wypadku i *= 2 i

skaczemy znowu. W ten sposób robimy takie skakanie o kolejne potęgi liczby 2, dzięki czemu w log(n) znajdziemy prawy koniec (być może przeskoczymy n-ty indeks, wchodząc na None, ale nie zaszkodzi to rzędowi złożoności). Następnie wykonujemy binary search, szukając x. Dostajemy ostatecznie O(log(n)).

Zadanie 6
Uwaga: Maciej Czyjt udowodnił, że oryginalne rozwiązanie jest odpowiednie dla liczb całkowitych lub dla braku unikatowości dla liczb naturalnych. Dla parami różnych liczb naturalnych przedstawione przez niego poniżej rozwiązanie działa w O(1).

Jeżeli liczby są parami różne, to dla j-tego indeksu na lewo są liczby tylko < j, a na prawo tylko > j. Jeżeli byłoby inaczej, to na 100% warunek zadania nie zachodzi. Wystarczy zatem sprawdzić, czy A[0] == 0. Jeżeli by tak nie było, to na pewno dla żadnego i nie zachodzi A[i] == i, bo nie zaczynalibyśmy od 0, a więc wszystkie liczby byłyby "przesunięte" względem swojego własnego indeksu.

Dla uogólnionego przypadku dowolnych liczb całkowitych poniższe stare rozwiązanie jest prawidłowe:
Skoro tablica jest posortowana, to dla j-tego indeksu:

- jeżeli A[j] < j, to jeżeli istnieje A[i] == i, to musi być na prawo
- jeżeli A[i] > j, to jeżeli istnieje A[i] == i, to musi być na lewo

Wystarczy więc wykonać przeszukiwanie binarne z decyzją o tym, gdzie przeszukiwać, podejmowaną zgodnie z powyższymi zasadami. Złożoność: O(log(n))

Zadanie 7
https://leetcode.com/problems/maximum-gap/solution/

Zadanie 8
Najpierw sortujemy punkty. Później będziemy używać dwóch sum odległości (elementów tablicy): wcześniejszych od aktualnego (prefix) oraz późniejszych od aktualnego (suffix). Dzięki nim wystarczy jedno przejście po tablicy, a będziemy cały czas znali odległości do wcześniejszych i późniejszych elementów.

Dla i-tego elementu wcześniejsze elementy mają położenia A[0], A[1], … , A[i-1], a późniejsze A[i+1], A[i+2], …, A[n - 1] (bo tablica ma n elementów). Suma odległości do wcześniejszych punktów to zatem (A[i] - A[0]) + (A[i] - A[1]) + … + (A[i] - A[i - 1]), czyli:
i * A[i] - (A[0] + A[1] + … + A[i - 1]). Analogicznie dla elementów późniejszych.

Zauważmy, że jeżeli będziemy trzymali "aktualny" prefix i suffix, czyli "aktualną" sumę elementów wcześniejszych i późniejszych, to wzory sprowadzają się do prefix + (i - 1) * A[i] oraz suffix + (len(a) - i) * A[i]. Dlatego też podczas iteracji po tablicy na początku prefix = 0, suffix = sum(arr), a potem będziemy zwiększać prefix i zmniejszać suffix o A[i], przechodząc dalej. W ten sposób będziemy mieli w pamięci O(1) i kosztem czasowym O(1) aktualną sumę elementów wcześniejszych i późniejszych.

Policzenie odległości sprowadza się do zastosowania wcześniej wyprowadzonych wzorów: dodajemy odległości do elementów wcześniejszych i do elementów późniejszych, nadążamy za tym dla którego indeksu i jest ona najmniejsza i to zwracamy.

Złożoność: O(nlog(n)) ze względu na sortowanie, sama ta iteracja to O(n)

Zadanie 9

Sortujemy A i B, i dla każdego elementu C, przeprowadzamy szukanie idąc dwoma wskaźnikami: jednym z początku A, drugim z końca B. w zależności od tego, czy suma elementów wskazywanych przez wskaźniki w A i B jest większa lub mniejsza niż element z C, to przesuwamy odpowiedni wskaźnik. Jak natrafimy na równą sumę, to raportujemy o sukcesie.

Inne podejście polega na posortowaniu C i dla każdej pary z A i B szukamy sumy binarnie.

Złożoności: dla wielkości tablic odpowiednio a, b i c:

1) pierwsze podejście: O(alog(a) + blog(b) + c(a+b))
2) drugie podejście: O(clog(c) + a*b*log(c)) = O(log(c) * (a*b + c))

Zadanie 10

Sortujemy tablicę. Wyznaczamy tablicę sum prefixowych i suffixowych. Każde zapytanie, to wyszukanie binarnie, gdzie w tablicy zaczynają się elementy, takie, że nawet po dodaniu sumy wszystkich dotychczasowych x z zapytań, będą liczby ujemne. Potem, dla fragmentu liczb dodatnich:

odpowiednia suma prefixowa + odpowiednia liczba* akumulator wartości x. Dla części ujemnej to samo, tylko przemnażamy przez -1. Złożoność O(nlog(n)).