# Algorytmy i Struktury danych (2021)

## Lista zadań 3 (rekurencja, drzewa, sortowanie)

1. Ile (dokładnie) porównań wykona algorytm `insertion_sort` w wersji z wartownikiem (liczbą $-\infty$ zapisaną pod adresem `t[-1]`), jeśli dane $(a_1, ..., a_n)$ o rozmiarze `n` zawierają `k` inwersji. Liczba inwersji to liczba takich par $(i, j)$, że $i < j$ i $a_i > a_j$. Jaka jest maksymalna możliwa liczba inwersji dla danych rozmiaru $n$? Wylicz "średnią" złożoność algorytmu, jaka średnią z maksymalnej i minimalnej ilości porównań jaką wykona. Uwaga: Prawdziwą średnią złożoność oblicza się, jako średnią po wszystkich możliwych permutacjach danych wejściowych.

2. (a) Ile co najwyżej porównań wykona procedura `insertion_sort` działająca na ostatnim etapie `bucket_sort` zakładając, że `bucket_sort` korzysta z `k` pomocniczych kolejek, i że do każdej z nich wpadła taka sama ilość elementów? Zakładamy wersję z wartownikiem na pozycji `t[-1]`.
   (b) Podaj uproszczony wynik dla $k = n/2$, $k = n/4$, $k = n/10$ oraz $k = \sqrt{n}$.
   Następnie każdy z tych wyników zapisz też w notacji asymptotycznej $O(f(n))$.
   (c) Jaki będzie wynik, gdy wszystkie klucze wpadną do tego samego kubełka?

3. Iteracyjna wersja procedury `mergesort` polega na scalaniu posortowanych list. Zaczyna łączenia pojedynczych elementów w posortowane pary, potem par w czwórki itd. aż do połączenia dwóch ostatnich list w jedną.
   (a) Zakładając, że rozmiar tablicy jest potęgą dwójki $n = 2^k$ oraz, że procedura merge wykonuje dokładnie tyle porównań, ile jest elementów po scaleniu, oblicz ile dokładnie porównań wykona cały algorytm.
   (b) Ile razy jest wywołana procedura `merge` w trakcie działania całego algorytmu? Jak zmieni się wynik punktu (a), jeśli założymy, że `merge` zawsze, wykonuje o jedno porównanie mniej, niż zakładaliśmy w punkcie (a)?

4. Napisz procedurę `void insertion_sort(lnode*& L)` – sortowanie przez wstawianie działające na liście jednokierunkowej. Zadbaj o to, by algorytm modyfikował jedynie pola `next` istniejących węzłów (nie używaj `new` ani `delete`). Jeśli list wejściowa jest posortowana, algorytm powinien wykonać tylko $n-1$ porównań. Wskazówka: algorytm wkładać elementy na "nową" listę w kolejności malejącej, a na końcu wywołać `reverse(L)`.

5. W pliku jest $n = 10^9$ liczb całkowitych. Ile potrzeba pamięci i dodawań by sprawdzić, która z sum $k = 10^4$ kolejnych liczb jest największa? Tych sum jest $n - k + 1$ tzn $i$-ta suma zaczyna się od $i$-tej liczby. Napisz program obliczający tę wartość tak, aby całkowity rozmiar utworzonych zmiennych wynosił około 20-bajtów (nie licząc obiektów `ifstream`), niezależnie od wartości $n$ i $k$. Argumentami programu powinny być liczba $k$ oraz nazwa pliku.
   Wskazówka: plik można czytać w dwóch miejscach jednocześnie.

6. Zakładając, że w każdym węźle drzewa BST jest dodatkowe pole `int nL`; pamiętające ilość kluczy w lewym poddrzewie, napisz funkcję `BTSnode* ity(BSTnode *t, int i)`, która zwróci wskaźnik `i`-ty (w kolejności `in order`) węzeł. Przyjmujemy, że numeracja zaczyna się od 0, czyli `ity(t,0)` to węzeł zawierający najmniejszy klucz. Dla wartości $i \geq n$ oraz $i < 0$ wynikiem funkcji powinien być `nullptr`. Pesymistyczna złożoność algorytmu powinna być równa głębokości drzewa. Nie korzystaj z rekurencji.

   Skoryguj procedurę `insert`, i konstruktor `node`, by prawidłowo aktualizowały wartości `nL` w trakcie dodawania elementów.

7. * Skoryguj procedurę `remove`, by prawidłowo aktualizowała wartości `nL` w trakcie usuwania elementów. Napisz funkcję `void remove_ity(BSTnode*&t,int i)`, która usunie `ity(t,i)` węzeł z drzewa.

Ćwiczenia do wykonania jako przygotowanie do kolokwium:

1. Napisz nierekurencyjną procedurę `int poziom(BSTnode * t, int klucz)`, której wynikiem jest poziom w drzewie `t`, na którym występuje `klucz`. Wynik 0 oznacza brak klucza w drzewie, 1 - klucz w korzeniu, 2 - w dziecku korzenia itd.

2. Napisz procedurę `int suma_do_poziomu(BSTnode * t, int poziom)`, której wynikiem jest suma kluczy znajdujących się na głębokości nie większej niż `poziom`. Przyjmujemy, że korzeń drzewa jest na poziomie 1.

3. Napisz funkcję `node* shift_sorted(node*& L)`, która od początku listy `L` odcina listę niemalejącą o największej możliwej długości i ją zwraca. Znaczy to, że cięcie jest przed pierwszym elementem mniejszym od poprzedniego lub na końcu listy.

Zadania dla ambitnych:

1. (3pkt) Napisz procedurę `void merge_sort(node*& n)` - sortowanie przez złączanie działające na liście jednokierunkowej, nie używaj rekurencji. Zadbaj by rozmiar dodatkowej pamięci nie przekroczył $\log_2 n + const$.
   Skorzystaj z procedury `lnode* merge(lnode* L1, lnode* L2)` z poprzedniej listy.

2. (3pkt) Napisz procedurę `merge` działającą na tablicy tak, aby nie wykorzystywać dodatkowego bufora, kosztem zwiększenia złożoności do $O(n \log n)$. Wskazówka: w czasie $O(n)$ można wykonać przejście $(1, 3, 5, 7, 9|2, 4, 6, 8, 10) \rightarrow (1, 2, 5|2, 4|7, 9|6, 8, 10)$, które powoduje, że każdy element lewej części jest mniejszy od każdego w prawej, a następnie rekurencyjnie wywołać `merge` dla każdej części. Jaka będzie wtedy złożoność `mergesort`?