E1. Z definice dokažte: $\forall k \in N, k \geq 1 : n^k \in o(n^{k+1})$.

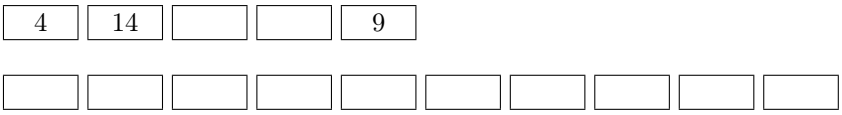
E2. Do třídy `List` dopište metodu `void remove(Object o)`, která odstraní zadaný prvek ze spojového seznamu. Prvky porovnávejte pomocí `equals`, dejte si pozor, abyste správně ošetřili všechny krajní případy!

```
class List {
    Node first;

    List() { first = new Node(); }
    void add(Object o) {
        first = new Node(o, first, first.previous);
    }
}

class Node {
    Object contents;
    Node next, previous;

    Node(Object c, Node n, Node p) {
        contents = c;
        next = n;
        previous = p;
        next.previous = this;
        previous.next = this;
    }

    Node() {
        next = this;
        previous = this;
    }
}
```

E3. Třída `Heap` je naimplementovaná špatně. Vyznačte kde a vysvětlete proč. Za špatná **nepovažujte** ta řešení, která sice jsou zbytečně neefektivní, ale nezvyšují časovou složitost operací `add` a `removeMin` nad $O(\log n)$. Jinými slovy: vyznačte kde a napište jak je porušen invariant haldy.

```
class Heap {
    int[] contents = new int[5];
    int size = 0;

    public void add(int e) {
        if (contentsTooSmall()) enlarge();
        int index = size;
        contents[size++] = e;
        while (index > 0) {
            int indexOfParent = (index - 1) / 2;
            if (contents[index] < contents[indexOfParent])
                swap(index, indexOfParent);
            index = indexOfParent;
        }
    }

    public int removeMin() {
        // uzivatel nesmi volat removeMin na prazdne halde
        int result = contents[0];
        size--;
        contents[0] = contents[size];
        int index = 0;
        while (index < size) {
            int indexOfChild = index * 2 + 1;
            if (contents[index] > contents[indexOfChild])
                swap(index, indexOfChild);
            index = indexOfChild;
        }
        if (contentsTooBig()) shrink();
        return result;
    }

    private void swap(int i, int j) {
        int temp = contents[i];
        contents[i] = contents[j];
        contents[j] = temp;
    }

    // kod zbylych metod
    // predpokladejte, ze je spravne
}
```

E4. Rozptylovací tabulka používá uzavřené rozptylování s funkcí $f(x) = x \bmod delka\_tabulky$, po překročení 75% kapacity tabulky se zvětšuje na dvojnásobek. Nakreslete stav tabulky po vykonání následující sekvence operací (počáteční stav tabulky je na obrázku): $insert(16), insert(24), insert(22), remove(4)$.



E5. Spočítejte asymptotickou časovou složitost funkce `f` v závislosti na `n`.

```
void f(int n) {
    for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++) {
            array[i][j]++;
            if (i == j) for (int k = 0; k < n; k++)
                array[i][j] += k;
        }
}
```

E6. Na malém dopravním letišti neexistuje něco jako pravidelný odlet/přílet, místo toho se vzletová/přistávací dráha (jen jedna) přiděluje tomu, kdo si o ni zažádal a má největší počet pasažérů (předpokládáme, že letadla ve vzduchu mají vždy dost paliva). Jakou datovou strukturu (z těch, co jsme probírali) byste zvolili pro evidenci všech žádostí a výběr žádosti s největším počtem pasažérů? Svůj výběr odůvodněte a v pseudokódu naznačte způsob použití. Jak by se tento způsob změnil, kdyby letiště mělo víc drah?

E7. Dopište implementaci quickselectu.

```
Integer quickselect(List<Integer> list, int index) {
    // predpokladame, ze 0 <= index < list.size()
    int pivot = list.get(list.size() / 2);
    List<Integer> smaller = getSmallerElems(list, pivot);
    List<Integer> equal = getEqualElems(list, pivot);
    List<Integer> greater = getGreaterElems(list, pivot);









}

List<Integer> getSmallerElems(List<Integer> list, Integer e) {
    List<Integer> result = new ArrayList<Integer>();




    return result;
}

List<Integer> getGreaterElems(List<Integer> list, Integer e) {
    List<Integer> result = new ArrayList<Integer>();




    return result;
}

List<Integer> getEqualElems(List<Integer> list, Integer e) {
    List<Integer> result = new ArrayList<Integer>();




    return result;
}
```

D1. Předpokládejte, že při násobením matic o velikostech $n \times m$ a $m \times l$ potřebujete přesně $n \cdot m \cdot l$ operací. Ověřte si, že při násobení $(M_1 \cdot M_2) \cdot M_3$ a $M_1 \cdot (M_2 \cdot M_3)$ dostanete stejný výsledek za použití různého počtu operací a pomocí dynamického programování najděte uzávorkování minimalizující počet operací při násobení řady matic $M_1 \cdot M_2 \cdot \ldots \cdot M_n$.

D2. Napište dvourozměrnou rozptylovací tabulku s otevřeným rozptylováním. Tabulka by měla podporovat operace vložení hodnoty asociované s dvojicí klíčů (`put(Object firstKey, Object secondKey, Object value)`) a vrácení hodnoty asociované s dvojicí klíčů (`get(Object firstKey, Object secondKey)`). Tabulku skutečně naimplementujte jako dvourozměrnou, ne jako složeninu jednorozměrných tabulek, implementaci spojového seznamu psát nemusíte, hashovací kód objektu zjišťujte pomocí metody `hashCode`.

D3. Třída `Node` reprezentuje jeden uzel v binárním vyhledávacím stromu. Napište iterátor (s metodami `next` a `hasNext`), který postupně vrátí všechny prvky tohoto stromu v pořadí od nejmenšího po největší. Předpokládejte, že strom si prvky udržuje v listech i vnitřních uzlech.

```
class Node {
    Node parent; // v korenu je null
    Node left, right;
    int contents;
}
```

D4. Máte dvě setříděné sekvence a chcete z nich udělat haldu. Existuje způsob rychlejší než $n \log n$? Pokud ano, napište jaký, pokud ne, napište proč (v obou případech stačí srozumitelný slovní popis).

D5. Napište adresovatelnou haldu (klíč je celé číslo, hodnota asociovaná s klíčem obecný objekt) naimplementovanou pomocí ukazatelů. Nezapomeňte na operaci `decreaseKey`, operaci `merge` můžete vynechat.

C1. Představte si, že posloupnost prvků konvertujete na (obyčejný) binární vyhledávací strom tak, že z posloupnosti vyjmete (v konstantním čase) náhodný prvek a vložíte ho do stromu. Ukažte, že tímto způsobem v průměru postavíte strom v čase $O(n \log n)$. Tip: vzpomeňte si na průměrnou složitost quicksortu.

C2. Napište invariant, který musí platit pro každou instanci třídy `Node`. Nápověda: zamyslete se nad tím, jak jsou jednotlivé uzly na sebe napojeny.

```
class List {
    Node first;

    List() { first = new Node(); }
    void add(Object o) {
        first = new Node(o, first, first.previous);
    }
}

class Node {
    Object contents;
    Node next, previous;

    Node(Object c, Node n, Node p) {
        contents = c;
        next = n;
        previous = p;
        next.previous = this;
        previous.next = this;
    }

    Node() {
        next = this;
        previous = this;
    }
}
```

C3. V poli `array` jsou za sebou uloženy dvě setříděné posloupnosti shodné délky. Napište funkci `merge`, která tyto dvě posloupnosti v lineárním čase sloučí za použití maximálně konstantního prostoru mimo pole.

C4. Dokažte, že funkce $f(0) = 1, f(1) = 1, f(n) = f(n-1) + f(n-2)$ má exponenciální složitost. Tip: pokuste se její složitost shora a zdola omezit funkcemi $g(n) = g(n-1) + g(n-1)$ a $h(n) = h(n-2) + h(n-2)$ (nakreslete si obrázek).

C5. V automatu máte nekonečnou zásobu mincí o hodnotách $2^n, n \in N$. Když má automat vrátit částku x, postupuje tak, že najde největší n takové, že $2^n \leq x$ a poté postupuje rekurzivně pro částku $x - 2^n$, dokud nevrátí celou částku. Ukažte, že tento hladový algoritmus minimalizuje počet vrácených mincí. Tip: pravděpodobně budete muset nejprve dokázat, že tento algoritmus vrací maximálně jednu minci od každé hodnoty.