REKURENCJA UNIWERSALNA:

(DOWODY: http://smurf.mimuw.edu.pl/node/829)

Dla funkcji T(n) zadanej przez

$$\begin{cases} \boldsymbol{\Theta(1)}, & \boldsymbol{dla\ n\ \in \{0,1\}} \\ \boldsymbol{a * T\left(\frac{n}{b}\right) + f(n)}, & \boldsymbol{dla\ n > 1} \end{cases}$$

zachodzi:

- Jeśli $\boldsymbol{f(n) = O(n^{log_b a -})}$ dla pewnego > **0,** to $\boldsymbol{T(n) = \Theta(n^{log_b a})}$,
- Jeśli $\boldsymbol{f(n) = \Theta\left(n^{log_b a}\right)}$, to $\boldsymbol{T(n) = \Theta(n^{log_b a} lgn)}$,
- Jeśli $\boldsymbol{f(n) = \Omega(n^{log_b a +})}$ dla pewnego > **0** oraz $\boldsymbol{af\left(\frac{n}{b}\right) \le cf(n)}$ dla pewnego **c** < **1** i prawie wszystkich **n,** to $\boldsymbol{T(n) = \ \Theta(f(n))}$

2. Zastosuj twierdzenie o rekursji uniwersalnej do rozwiązania do następujących problemów:

a) $\boldsymbol{T(n) = 4T\left(\frac{n}{3}\right) + n}$

a = 4 b = 3 f(n) = n

$$\boldsymbol{n^{log_b a} = n^{log_3 4} = n^{1{,}26}}$$

$$\boldsymbol{T(n) = \Theta\left(n^{log_b a}\right) = \Theta(n^{1{,}26})}$$

b) $\boldsymbol{T(n) = 6T\left(\frac{n}{3}\right) + n^2}$

a = 6 b = 3 f(n) = n^2

$$\boldsymbol{n^{log_b a} = n^{log_3 6} = n^{1{,}63}}$$

$$\boldsymbol{T(n) = \Theta(f(n)) = \Theta(n^2)}$$

c) $\boldsymbol{T(n) = 8T\left(\frac{n}{4}\right) + n\sqrt{n}}$

a = 8 b = 4 f(n) = $\boldsymbol{n\sqrt{n}}$

$$\boldsymbol{n^{log_b a} = n^{log_4 8} = n^{1{,}5}}$$

$$\boldsymbol{T(n) = \Theta\left(n^{log_b a} lgn\right) = \Theta(n^{1{,}5} lgn)}$$

d) $\boldsymbol{T(n) = 7T\left(\frac{n}{2}\right) + n^3}$

a = 7 b = 2 f(n) = n^3

$$n^{\log_b a} = n^{\log_2 7} = n^{2,8}$$

$$T(n) = \Theta(f(n)) = \Theta(n^3)$$

e) $\boldsymbol{T(n) = T\left(\frac{n}{3}\right) + \sqrt[3]{n}}$

a = 1 b = 3 f(n) = $\sqrt[3]{n}$

$$n^{\log_b a} = n^{\log_3 1} = n^0 = 1$$

$$T(n) = \Theta(f(n)) = \Theta(\sqrt[3]{n})$$

3.

a) Napisz funkcję **merge(double t1[], int n1, double t2[], int n2, double t3[]),** która złącza posortowane tablice **t1, t2** o rozmiarach **n1, n2** w jedną posortowaną tablicę **t3**

```
void merge(double t1[ ], int n1, double t2[ ], int n2, double t3[ ])
{
   int i = 0;
   int j = 0;
   int k = 0;
   while( i < n1 && j < n2 )
   {
     if(t1[i] < t2[j])
     {
        t3[k] = t1[i];
        k++;
        i++;
     }
     else
     {
        t3[k] = t2[j];
        k++;
        j++;
     }
   }
   while(i < n1)   t3[k++] = t1[i++];
   while(j < n2)   t3[k++] = t2[j++];
}
```

b) Ile maksymalnie porównań między elementami tablicy wykonuje funkcja **merge?**

Bierzemy dwie najgorsze możliwe tablice, tj. t1 = [1, 3, 5, 7] i t2 = [2, 4, 6, 8].

Porównania będą się odbywać w sposób: t1[0] z t2[0] itd. Dla pierwszego przypadku, t1 będzie mniejsze, więc w następnym kroku porównanie odbędzie się dla t1[1] i t2[0]. Tutaj rezultat się odwróci. Zmiany te będą się powtarzać do końca, stąd też maksymalna liczba porównań będzie (n + m) – 1, gdzie 1 tj. ostatni element tabeli, nie porównywany z niczym.

c) Napisz funkcję **merge_sort(double t[], int n)**

```
void mergesort(double t[ ], int n)
{
  if(n > 1)
  {
    int k = n/2;
    mergesort(t, k);

    mergesort(t + k, n - k);
    merge(t, n, b, k);
  }
}
```

- ✓ **Znajdujemy środek (n/2)**
- ✓ **Mergesort dla pierwszej połowy**
- ✓ **Mergesort dla drugiej połowy**
- ✓ **Merge obydwu połów**

d) Jaka jest złożoność **mergesort (czasowa,** pamięciowa: O(n))? Udowodnij swoją odpowiedź korzystając z Twierdzenia o Rekursji Uniwersalnej.

$\boldsymbol{T(n) = 2T\left(\frac{n}{2}\right) + n;}$ **a = 2** **b = 2** **f(n) =** $\boldsymbol{n}$

$$\boldsymbol{n^{log_b a} = n^{log_2 2} = n^1 = n}$$

Druga opcja tj. $\boldsymbol{T(n) = \Theta(n^{log_b a} * logn)}$

$$\boldsymbol{T(n) = \Theta(n^{log_2 2} * logn) = \Theta(nlogn)}$$

Gdzie **a** – oznacza, że wykonujemy merge_sort dla prawej i lewej strony tablicy (stąd 2)

b – (patrz w kodzie) oznacza, że zakładamy, że dzielimy tablicę na 2 równe podproblemy

c – tj. pesymistyczny czas merge, tj. czas związany z robieniem czegoś z wynikiem rekursji (w tym przypadku wynik rekursji jest potem przerabiany przez merge

5. Dana jest definicja struktury węzła listy pojedynczo wiązanej:

```
struct lnode {
   int key;
   lnode * next;
   lnode(int k, lnode * nullptr): key(k), next(n){   }
}
```

a) Napisz funkcję int sum(lnode* L) zwracającą sumę kluczy listy L.

```
int sum(lnode *  L)
{
   int result = 0;
   while(L)
   {
      result += L → key;
      L = L → next;
   }
   return result;
}
```

b) Napisz funkcję **void prepend(lnode* &L, int x)**, która dodaje liczbę x na początku listy **L**

```
void prepend(lnode *  &L, int x)
{
   lnode *  temp = L;
   L → key = x;
   L → next = temp;
}
```

c) Napisz funkcję **int get_first(lnode* &L)**, która usuwa pierwszy element niepustej listy **L** i zwraca wartość usuniętego klucza.

```
int getfirst(lnode * &L)
{
   int result = 0;
   result = L → key;
   lnode * temp = L;
   delete L;
   L = temp → next;
   return result;
}
```

d) Napisz funkcję **void insert(lnode* &L, int x)**, która do posortowanej listy **L** dodaje liczbę **x** tak, aby lista **L** w dalszym ciągu była poprawnie posortowana. Można przyjąć, że **x** jest dodawany przed pierwszą liczbą, która jest od niego większa, a jeśli nie ma takiej liczby, to na końcu listy.

```
void insert(lnode * &L, int x)
{
  while(L)
  {
    if(x ≥ L → key)
    {
      L = L → next;
    }
    else
    {
      lnode * temp = L;
      L = new lnode(x, nullptr);
      L → next = temp;
      break;
    }
  }
}
```

e) Napisz funkcję **void usun_nieparzyste(lnode* &L)**, która z listy **L** usuwa wszystkie elementy nieparzyste.

```
void usunnieparzyste(lnode * &L)
{
  while(L)
  {
    if(L → key % 2 == 1)
    {
      lnode * temp = L;
      delete L;
      L = new lnode(temp → key, temp);
    }
    L = L → next;
  }
}
```

6. Dana jest definicja struktury węzła drzewa BST:

struct node { int key; lnode* left; lnode* right; }

a) Napisz funkcję **void destroy(node* t)**, która usunie drzewo t z pamięci

$$\boldsymbol{void\ destroy}(\boldsymbol{node} * \ \boldsymbol{t})$$
$$\{$$
$$\quad \boldsymbol{if}(\boldsymbol{t})$$
$$\quad \{$$
$$\qquad \boldsymbol{destroy}(\boldsymbol{t} \rightarrow \boldsymbol{left});$$
$$\qquad \boldsymbol{destroy}(\boldsymbol{t} \rightarrow \boldsymbol{right});$$
$$\qquad \boldsymbol{t} = \boldsymbol{NULL};$$
$$\quad \}$$
$$\}$$