GRUPA II

1. Napisz kod funkcji `int segment(wel L)` operującej na jednokierunkowej liście nieuporządkowanej `L`, która zwraca długość najdłuższego segmentu w `L`, tj. ciągu jej elementów utworzonego przez kolejne liczby `int`. Np. dla listy

$$1 \to 2 \to 3 \to 5 \to 3 \to 2 \to 3 \to 4 \to 5 \to 3 \to \texttt{NULL}$$

funkcja powinna zwrócić wartość 4, bo najdłuższy segment to $2 \to 3 \to 4 \to 5$.

2. Nieuporządkowana lista jednokierunkowa zawiera liczby dodatnie. Napisz kod funkcji `usunw(wel *L)` usuwającej z niej te elementy, które są większe od sumy wszystkich swoich poprzedników, przy czym pierwszy element pozostaje zawsze bez zmian. A więc po zadziałaniu tej funkcji powinien być spełniony warunek

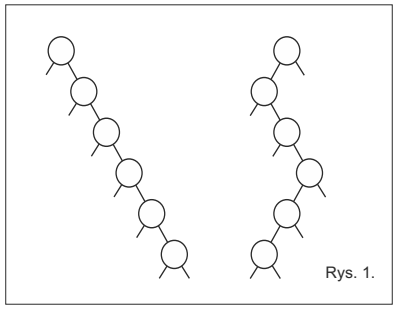
$$\text{dla każdego } \; 1 < k \leq n \qquad x_1 + x_2 + \cdots + x_{k-1} \; \geq \; x_k \,,$$

gdzie n jest liczbą elementów pozostałych na liście `L`.

3. Napisz funkcję logiczną `int powt(wel L)`, która zwraca wartość 1, gdy jednokierunkowa nieuporządkowana lista cykliczna wskazywana przez `L` zawiera powtarzające się elementy, lub 0 w przeciwnym razie.

4. Napisz kod rekursywenej funkcji `int suma_liści(weld T)`, która jako wartość zwraca sumę liczb zapisanych w liściach drzewa `T`. Gdy drzewo jest puste, funkcja zwraca wartość 0.

5. Napisz funkcję `int degen(weld T)`, która zwraca wartość 1 jeśli drzewo `T` ma zdegenerowaną strukturę, efektywnie identyczną ze zwykłą listą jednokierunkową (tak jak na rysunku).



GRUPA III

1. Napisz procedurę `void min2front(wel *L)`, która odnajduje minimalny element nieuporządkowanej listy jednokierunkowej `L` i przenosi go na jej początek. Wszystkie inne elementy listy pozostają na swoim miejscu.

2. Wykorzystaj wywołania funkcji `min2front` z poprzedniego zadania do posortowania nieuporządkowanej listy. Jaka jest pesymistyczna ocena złożoności (wyrażona liczbą porównań i przestawień elementów listy względem jej długości n) takiego sposobu sortowania? Odpowiedź uzasadnij.

3. Funkcja `int max_powt(wel C)` operuje na nieuporządkowanej liście cyklicznej `C`. Jej wartością ma być maksymalna liczba powtórzeń pojedynczego elementu na tej liście. Np. dla listy

$$2 \to 5 \to 3 \to 2 \to 6 \to 3 \to 3 \to 5 \to 4 \to 3 \to 1 \to 5 \to \text{(wskaźnik na pierwszy element)}$$

wynikiem powinna być liczba 4 bo "3" występuje 4 razy na tej liście.

4. Napisz funkcję `void usun_l(weld *T)`, która usuwa wszystkie liście z drzewa `*T`. Jeśli korzeń jest liściem, czyli jedynym elementem drzewa, wynikiem jest drzewo puste.

5. Należy porównać 2 *uporządkowane* drzewa pod kątem występowania w nich wspólnych elementów. Napisz funkcję logiczną `int wspólne(weld T1, weld T2, int k)`, która zwraca wartość 1 jeśli `T1` i `T2` mają przynajmniej `k` wspólnych elemenów. Interesuje mnie rozwiązanie optymalne, które przegląda każde z drzew nie więcej niż 1 raz. Można wykorzystać wywołania funkcji `weld min_drzewa(weld T)` oraz `weld następnik(weld T)`, takich jak omawiane na ćwiczeniach bez konieczności pisania ich kodów.

ROZWIĄZANIA — GRUPA II

1.
```
int segment (wel L){
   int s=0, ms=0, y;
   if (L!=NULL){
      y = L->x;
      L = L->nast;
      s=1;
      while (L!=NULL){
         if (L->x==y+1){ s++;}
         else {
            if (s>ms) ms=s;
            s=1;
         }
         y=L->x;
         L=L->nast;
      }
   }
   return (s>ms) ? s : ms;
}
```

2.
```
void usunw(wel *L){
   wel q;
   int s;
   if (*L!=NULL){
      s = (*L)->x;
      L = &((*L)->nast);
      while (*L!=NULL){
          if ((*L)->x > s){
             q = *L;
             *L = (*L)->nast;
             free(q);
          }
          else {
             s += (*L)->x;
             L = &((*L)->nast);
          }
      }
   }
}
```

3.
```
int powt(wel L){
   wel p, q;
   if (L != NULL){
      p = L;
      do {
         for (q=p->nast; q!=L; q=q->nast)
            if (q->x == p->x) return 1;
         p = p->nast;
      } while (p != L)
   }
   return 0;
}
```

4.
```
int suma_liści(weld T){
    if (T==NULL) return 0;
    if ((T->l==NULL) && (T->p==NULL)) return T->x;
    return suma_liści(T->l) + suma_liści(T->p);
}
```

5.
```
int degen(weld T){
    if (T==NULL) return 1;
    return ((T->l==NULL) && degen(T->p)) || ((T->p==NULL) && degen(T->l));
```

ROZWIĄZANIA — GRUPA III

1.
```
void min2front(wel *L){
    wel *p, *q=L, r;
    if (*L != NULL){
       for ( p=&((*L)->nast); *p!=NULL; p=&((*p)->nast) )
          if ((*p)->x < (*q)->x) q = p;
       if (q != L){
          r = *L;
          *L = *q;
          *q = *q->nast;
          (*L)->nast = r;
       }
    }
}
```

2.
```
void sortuj(wel *L){
    if (*L != NULL){
        while ((*L)->nast != NULL) {
            min2front(L);
            L=&((*L)->nast);
        }
    }
}
```

Oszacowanie złożoności: Funkcja `min2front` wykonuje tyle kroków w poszukiwaniu minimum, ile wynosi długość listy. W pierwszym przebiegu pętli `while` w funkcji `sortuj` jest to n kroków, w drugim $n-1$ ... itd., a w ostatnim 1. Łączna liczba kroków algorytmu wynosi więc

$$n + n-1 + \cdots + 1 = \frac{n(n+1)}{2} = O(n^2).$$

3.
```
int max_powt(wel C){
   wel p=C, q;
   int k, mx=0;
   if (C != NULL){
      do {
         q = p->nast;
         k = 1;
         while (q != C){
            if ( q->x == p->x ) k++;
            q = q->nast;
         }
         if (k > mx) mx = k;
         p = p->nast;
      } while (p->nast != C);
   }
   return mx;
}
```

4.
```
void usun_l(weld *T){
    if (*T!=NULL) {
       if (((*T)->l == NULL) && ((*T)->p == NULL)) { free(*T); *T=NULL; }
       else {
          usun_l(&((*T)->l));
          usun_l(&((*T)->p));
       }
    }
}
```

5.
```
int wspólne(weld T1, weld T2, int k){
   weld p1 = min_drzewa(T1), p2 = min_drzewa(T2);
   while ((k>0) && (p1!=NULL) && (p2!=NULL)){
      while ((p1!=NULL) && (p1->x < p2->x)) p1 = następnik(p1);
      while ((p2!=NULL) && (p2->x < p1->x)) p2 = następnik(p2);
      if ((p1!=NULL) && (p2!=NULL)){
         if (p1->x == p2->x) k--;
         p1 = następnik(p1);
         p2 = następnik(p2);
      }
   }
   return (k==0);
}
```