1. Rekurencja (recursion): *anagramy*; *rekurencyjne wyszukiwanie binarne*.
2. Sortowanie przez scalanie (merge sort).

1. Rekurencja

Rekurencja jest techniką programowania w której funkcja (lub też funkcja składowa) wywołuje samą siebie. "This might sound like a strange thing to do, or even a catastrophic mistake. (…) Like pulling yourself up by your bootstraps (you do have bootstraps, don't you?)"

Przykładem problemu w którym rekurencja może być zastosowana jest problem obliczenia kolejnych liczb tzw. trójkątnych (liczby w których związków mistycznych szukali starożytni matematycy z grupy Pitagorasa). Kila pierwszych liczb tworzy następujący szereg: 1, 3, 6, 10, 15, 21, … (szereg jest złożony z nieskończenie wielu wyrazów; jaka liczba powinna być następna?)

Drugi wyraz szeregu powstaje z dodania 2 do pierwszego wyrazu szeregu. Trzeci wyraz szeregu powstaje z dodania 3 do drugiego, itd. Nazwa liczby trójkątne bierze się od liczby identycznych obiektów które można ułożyć w kształt trójkąta.

Rekurencyjne obliczenie n-tej liczby trójkątnej jest podobne do znanej z życia codziennego „spychologii" (lub też zlecenia pracy podwykonawcy). Przykładowo chcemy obliczyć 9-tą liczbę trójkątną. Wiemy, że 9-ta liczba trójkątna wynosi tyle samo co 9 + ósma liczba trójkątna, wiec dzwonimy do Harry'ego i pytamy go o ósmą liczbę trójkątną. Kiedy Harry oddzwoni dodamy jego odpowiedź do 9 i otrzymamy w ten sposób 9-tą liczbę trójkątną.
Harry wie, że ósma liczba trójkątna wynosi tyle samo co 8 plus siódma liczba trójkątna wiec dzwoni do Sally i prosi ją o obliczenie siódmej liczby trójkątnej. Proces ten jest kontynuowany tak długo aż do wyznaczenia zostanie 1-wsza liczba trójkątna, która z założenia wynosi jeden.

Przykład 1 (C++) Program stosujący rekurencję do obliczenia liczby trójkątnej.

[triangle.cpp]

```
// triangle.cpp
// evaluates triangular numbers
#include <iostream>
using namespace std;

int main()
{
    int theNumber;
    int triangle(int);
    cout << "Enter a number: ";
    cin >> theNumber;
    int theAnswer = triangle(theNumber);
    cout << "Triangle=" << theAnswer << endl;
    return 0;
}  // end main()
//-------------------------------------------------------------
```

```
int triangle(int n)
{
    if(n==1)
        return 1;
    else
        return  ( n + triangle(n-1) );
}
```

Anagramy słów a rekurencja

Przypuśćmy, że chcemy otrzymać listę anagramów dla zadanego słowa. Listę anagramów możemy otrzymać wypisując wszystkie kombinacje liter podanego słowa (tylko część z nich będzie miało sens). Przykładowo możemy poszukać anagramów dla angielskiego słowa cat:

cat,
    ,cta
atc,
    ,act
tca,
    ,tac

Koncepcja otrzymywania anagramów
Zakładamy, że szukamy anagramów z słowa złożonego z n liter.

1. Szukamy anagramów z ostatnich n-1 liter,
2. Przesuwamy wszystkie litery w lewo (litera z pierwszej pozycji wędruje na ostatnią (RotateLeft))
3. Wykonujemy wszystkie kroki n-krotnie.

Przykład 2 (C++) Program stosujący rekurencję do wypisania anagramów wpisanego słowa.

[anagramy.cpp]

```
#include <iostream>
#include <string>
#include <cstdlib>

using std::cin;   using std::cout;
using std::endl;  using std::string;

class word
{
    private:
        string slowo;
        int n;
        int liczbawystapien;

    public:
        word(string s):slowo(s),n(slowo.length()),liczbawystapien(0)
        {}

    void rotate(int size){
        int position=n-size;
        char temp=slowo[position];
        for(int j=position+1;j<n;j++){
            slowo[j-1]=slowo[j];
        }
```

```
        slowo[n-1]=temp;
    }

    void anagram(int size){
        if(size==1)
            return;

        for(int j=0;j<size;j++){
            anagram(size-1);
            if(size==2){
                cout<<slowo<<" ";
                liczbawystapien++;
                if(liczbawystapien%6==0){
                    cout<<endl;
                }
            }
            rotate(size);
        }
    }
};
int main(){
    cout<<"Wpisz slowo: ";
    string ts;
    cin>>ts;
    int dlugoscslowa=ts.length();

    word s(ts);
    s.anagram(dlugoscslowa);

    system("pause");
    return 0;
}
```

Wyszukiwanie binarne

Napisz program realizujący poszukiwanie pewnej zadanej liczby x w tablicy posortowanej od wartości minimalnych do maksymalnych. Metoda poszukiwania bazować ma na tzw. *przeszukiwaniu binarnym*. Metoda ta polega na następującej obserwacji:

a) Podzielmy tablicę o rozmiarze n na połowę:
   - t[0], t[1] … t[n/2-1]; t[n/2], t[n/2+1],…,t[n-1].
   - jeśli x=t[n/2], to element x został znaleziony,
   - jeśli x<t[n/2], to element x (o ile istnieje) znajduje się w lewej połowie tablicy
     ➔ analizuj ją,
   - jeśli x>t[n/2], to element x (o ile istnieje) znajduje się w prawej połowie tablicy
     ➔ analizuj ją.

b) 12-elementowa tablica zawiera liczby: {1, 2, 6, 18, 20, 23, 29, 32, 34, 39, 40, 41}.
   Szukamy liczby 18.

Przykład 3 (C++). Program realizujący wyszukiwanie binarne.

[p3.cpp]

```
#include <iostream>
#include <cstdlib>

/* program realizujacy przeszukiwanie binarne
tablicy uprzednio posortowanej */

using namespace std;
int t[12]={1,2,6,18,20,23,29,32,34,39,40,41};

//przeszukiwanie binarne
void pb(int * m, int szuk, int left, int right){
    int mid=(left+right)/2;
    cout<<mid<<" ("<<m[mid]<<"),"<<left<<" "<<right<<endl;
    if(szuk==m[mid]){
            cout<<"Element "<<szuk<<" zostal
      znaleziony.\n***************\n";
            return;
    }
    else{
        if(szuk!=m[mid]&&right==left){
                cout<<"Element "<<szuk<<" nie zostal znaleziony.\n";
        }
    }
    if(right>left){
        if(szuk<m[mid]){
                right=mid-1;
        }
        else{
            left=mid+1;
        }
        pb(m,szuk,left,right);
    }
}

int main(){

int s=0;
for(int i=1;i<12;i++){
    s=t[i];
    cout << "Znajdowanie elementu s="<<s<<" w tablicy t[]:\n";
    pb(t,s,0,11);
}
cout<<"\n ################ \n";
for(int i=1;i<12;i++){
    s=t[i]+41;
    cout << "Znajdowanie elementu s="<<s<<" w tablicy t[]:\n";
    pb(t,s,0,11);
}
    s=3;
    cout << "Znajdowanie elementu s="<<s<<" w tablicy t[]:\n";
    pb(t,s,0,11);
cout<<endl;

system("pause");
return 0;
}
```

2. Sortowanie przez scalanie

Przykład 4 (C++) Realizacja sortowania przez scalanie.

[merge.cpp]

```
//merge.cpp
//demonstrates merging two arrays into a third
#include <iostream>
using namespace std;

void merge( int[], int, int[], int, int[] );
void display(int[], int);               //prototypes

int main()
{
    int arrayA[] = {23, 47, 81, 95};        //source A
    int arrayB[] = {7, 14, 39, 55, 62, 74}; //source B
    int arrayC[10];                         //destination

    merge(arrayA, 4, arrayB, 6, arrayC);    //merge A+B-->C
    display(arrayC, 10);                    //display result
    return 0;
}  //end main()

void merge( int arrayA[], int sizeA,        //merge A and B into C
int arrayB[], int sizeB,
int arrayC[] )
{
    int aDex=0, bDex=0, cDex=0;
    while(aDex < sizeA && bDex < sizeB)     //neither array empty
        if( arrayA[aDex] < arrayB[bDex] )
            arrayC[cDex++] = arrayA[aDex++];
        else
            arrayC[cDex++] = arrayB[bDex++];
    while(aDex < sizeA)                     //arrayB is empty,
        arrayC[cDex++] = arrayA[aDex++];    //but arrayA isn't
    while(bDex < sizeB)                     //arrayA is empty,
        arrayC[cDex++] = arrayB[bDex++];    //but arrayB isn't
}  //end merge()

void display(int theArray[], int size)      //display array
{
    for(int j=0; j<size; j++)
    cout << theArray[j] << " ";
    cout << endl;
}
```

Przykład 5 (C++) Realizacja sortowania przez scalanie (wersja rekurencyjna).

[mergeSort.cpp]

```
//mergeSort.cpp
//demonstrates recursive merge sort
#include <iostream>
#include <vector>
using namespace std;

class DArray
{
    private:
        vector<double>(theVect);            //vector of doubles
        int nElems;                         //number of data items
        void recMergeSort(vector<double>, int, int);
        void merge(vector<double>, int, int, int);

    public:
        DArray(int max) : nElems(0)         //constructor
        {
            theVect.resize(max);            //size vector
        }

        void insert(double value)           //put element into array
        {
            theVect[nElems] = value;        //insert it
            nElems++;                       //increment size
        }

        void display()                      //displays array contents
        {
            for(int j=0; j<nElems; j++)     //for each element,
                cout << theVect[j] << " ";  //display it
            cout << endl;
        }

        void mergeSort()                    //called by main()
        {                                   //provides workspace
            vector<double>(workSpace);
            workSpace.resize(nElems);
            recMergeSort(workSpace, 0, nElems-1);
        }
};  //end class DArray
//--------------------------------------------------------------
void DArray::recMergeSort(vector<double> workSpace,
int lowerBound, int upperBound)
{
    if(lowerBound == upperBound)        //if range is 1,
        return;                         //no use sorting
    else
    {                                   //find midpoint
        int mid = (lowerBound+upperBound) / 2;
                                        //sort low half
        recMergeSort(workSpace, lowerBound, mid);
                                        //sort high half
        recMergeSort(workSpace, mid+1, upperBound);
                                        //merge them
        merge(workSpace, lowerBound, mid+1, upperBound);
    }  //end else
```

```
}  //end recMergeSort()

void DArray::merge(vector<double> workSpace, int lowPtr,
int highPtr, int upperBound)
{
    int j = 0;                          //workspace index
    int lowerBound = lowPtr;
    int mid = highPtr-1;
    int n = upperBound-lowerBound+1;    //# of items

    while(lowPtr <= mid && highPtr <= upperBound)
        if( theVect[lowPtr] < theVect[highPtr] )
            workSpace[j++] = theVect[lowPtr++];
        else
            workSpace[j++] = theVect[highPtr++];

    while(lowPtr <= mid)
        workSpace[j++] = theVect[lowPtr++];

    while(highPtr <= upperBound)
        workSpace[j++] = theVect[highPtr++];

    for(j=0; j<n; j++)
        theVect[lowerBound+j] = workSpace[j];
}  //end merge()

int main()
{
    const int maxSize = 100;          //array size
    DArray arr(maxSize);              //create "array"

    arr.insert(64);                   //insert items
    arr.insert(21);
    arr.insert(33);
    arr.insert(70);
    arr.insert(12);
    arr.insert(85);
    arr.insert(44);
    arr.insert(3);
    arr.insert(99);
    arr.insert(0);
    arr.insert(108);
    arr.insert(36);

    arr.display();                    //display items
    arr.mergeSort();                  //merge-sort the array
    arr.display();                    //display items again
    return 0;
}  //end main()
```

Anagramy raz jeszcze

Program z przykładu nr 2 wyposażamy w kontener przechowujący znalezione anagramy oraz obliczamy czas znajdowania anagramów (zmiany wyróżnione żółtym tłem).

[anagramy2.cpp]

```
include <iostream>
#include <string>
#include <cstdlib>
#include <vector>
#include <ctime>

using std::cin;
using std::cout;
using std::endl;
using std::string;
using std::vector;

class word
{
    private:
        string slowo;
        int n;
        int liczbawystapien;
        vector<string> anagrams;

    public:
        word(string s):slowo(s),n(slowo.length()),liczbawystapien(0)
        {}

    void rotate(int size){
        int position=n-size;
        char temp=slowo[position];
        for(int j=position+1;j<n;j++){
            slowo[j-1]=slowo[j];
        }
        slowo[n-1]=temp;
    }

    void anagram(int size){
        if(size==1)
            return;

        for(int j=0;j<size;j++){
            anagram(size-1);
            if(size==2){
                anagrams.push_back(slowo);
                liczbawystapien++;
            }
            rotate(size);
        }
    }

    int howmany(){
        return liczbawystapien;
    }
```

```
    void display(){
        for(unsigned int i=0;i<anagrams.size();i++){
            cout<<anagrams[i]<<" ";
            if(i%6==0){
                cout<<endl;
            }
        }
    }
};


int main(){
    clock_t start1, finish1;
    clock_t clock(void);  //Returns the processor time consumed by the
                          // program
                          //clock_t is the type returned by clock.
    double duration1;


    cout<<"Wpisz slowo: ";
    string ts;
    cin>>ts;
    int dlugoscslowa=ts.length();

    word s(ts);

    start1 = clock();
    s.anagram(dlugoscslowa);
    finish1 = clock();
    duration1 = static_cast<double>(finish1 - start1)/CLOCKS_PER_SEC;

    //s.display();          //wyswietlanie zajmuje dodatkowy czas

    cout<<endl<<"Znaleziono "<<s.howmany()<<" anagramow slowa \'"<<ts
    <<"\' w "<<duration1<<" sekund"<<endl;
    /*============================================================*\
    //Dla slowa 1234567890 znajduje anagramy w okolo 1.3 sekundy
    //w przypadku proby zaalokowania zbyt duzej pamieci moze
    //wyrzucic wyjatek: "terminate called after throwing an instance of
    //'std::bad_alloc' what(): std::bad_alloc".
    \*============================================================*/


    system("pause");
    return 0;
}
```

Opracowano na podstawie:
Robert Lafore – "Teach Yourself Data Structures And Algorithms In 24 Hours"