

Testing of Algorithms of sorting and searching.

There's the results I have achieved, with using the standard "System collection" algorithms.

11	1st type		2nd type		3rd type		4th type	
SORTING	com	per	com	per	com	per	com	per
Sorting by exchanges (bubble Sort)	190	100	190	0	190	186	190	159
Sorting by choice	190	19	190	19	190	19	190	19
Improved algorithm of sorting by exchange(v1)	190	100	19	0	190	186	190	159
Improved algorithm of sorting by exchange(v2)	165	100	19	0	189	186	177	159
Sorting by inserts	77	100	19	0	86	186	89	159
Pyramidal sorting	133	70	139	78	139	64	135	63
Quick sorting by Hoar	111	26	78	13	84	23	110	26

And here are the results of some of My algorithms:

11	1st type		2nd type		3rd type		4th type	
SORTING	com	per	com	per	com	per	com	per
Sorting by exchanges (bubble Sort)	145	100	38	0	190	186	183	159
Sorting by choice (My improvement)	190	17	190	0	190	11	190	14
Improved algorithm of sorting by exchange(v1)	145	100	19	0	190	186	183	159
Pyramidal sorting(My implementation)	165	100	169	105	154	86	155	88
Quick sorting by Hoar(Median)	102	30	78	13	84	23	108	25
Quick sorting by Hoar(My implementation)	151	25	118	4	117	12	126	20
Quick sorting by Hoar(My implementation with median)	133	22	118	4	164	12	133	22

Sorting by exchanges

Improvement in Sorting By exchanges is made by adding a second loop, that checks the elements in an opposite direction. This makes the compares more effective and thus make's them less than in a "one direction" bubble sorting and there's less of "empty runs" if the table is already sorted. This kind of sorting is otherwise called "Shaker sorting".

Here's the code in pascal:

```

Program Bub1Sort;
Const
  n = 20;
Var
  a : array[1..n] of Data;
  i, j, left, right, prev, exc, cmp : Integer;
  b : Data;
Begin
  exc := 0;
  cmp := 0;
  left := 1;

```

```

    right := n - 1;

Repeat
    For j := right downto left do
        begin
            cmp := cmp + 1;
            If a[j] > a[j+1]
            then begin
                exc := exc + 1;
                b := a[j+1];
                a[j+1] := a[j];
                a[j] := b;
                prev := j;
            end;
        end;

    left := prev + 1;
    For j := left to right do
        begin
            cmp := cmp + 1;
            If a[j] > a[j+1]
            then begin
                exc := exc + 1;
                b := a[j+1];
                a[j+1] := a[j];
                a[j] := b;
                prev := j;
            end;
        end;
    right := prev - 1;
Until left > right;
End.

Program BublSort1v;
Const
    n = 20;
Var
    a : array[1..n] of Data;
    i, j, left, right, prev, exc, cmp : Integer;
    TempMem : Data;
    isOrd : Boolean;

Begin
    exc := 0;
    cmp := 0;
    left := 1;
    right := n - 1;

Repeat
    isOrd := True;
    For j := right downto left do

```

```

begin
    cmp := cmp + 1;
    If a[j] > a[j+1]
    then begin
        exc := exc + 1;
        TempMem := a[j+1];
        a[j+1] := a[j];
        a[j] := TempMem ;
        prev := j;
        isOrd := False;
    end;
end;
left := prev + 1;
if isOrd then left := right + 1;
For j := left to right do
begin
    cmp := cmp + 1;
    If a[j] > a[j+1] then
    begin
        exc := exc + 1;
        TempMem := a[j+1];
        a[j+1] := a[j];
        a[j] := TempMem ;
        prev := j;
        isOrd := False;
    end;
end;
right := prev - 1;
Until isOrd or (left > right)
End.

```

Sorting by choice

The improvement of sorting by exchanges was just a one line command, to not make change's if an element is already in place, this eliminated the exchanges between the same elements of the table. The code look like this:

```

Program SelectSort;
Const
    n = 20;
Var
    a : array[1..n] of Data;
    i, j, MinInd, exc, cmp : Integer;
    b : Data;
Begin
    exc := 0;
    cmp := 0;

    For i := 1 to n - 1 do begin
        MinInd := i;

```

```

    For j := i + 1 to n do
        begin
            cmp := cmp + 1;
            If a[j] < a[MinInd]
            then MinInd := j;
        end;

    If MinInd <> i then
    begin
        exc := exc + 1;
        b := a[MinInd];
        a[MinInd] := a[i];
        a[i] := b;
    end;
end;
End.

```

Pyramidal sorting

Here's a different implementation of Pyramidal (Heap) sorting, that I once have written for My diploma work, it's written in C++ but the idea is the same. Only the functions are little different and compares are made in different places, usually before calling of the function swap but not in the swap function as it is made in the "System collection" of Vint. But this implementation is not perfect, in fact it is worse than the Vint version and needs a complete rebuild, because in this code there's no improvement to be made it seems.

Here's the code in C++:

```

#include <iostream>
#include <cstdlib>
#include <ctime>

using namespace std;

typedef long T;

void generate(T [], int);
void wypisz(T [], int);
void heapSort(T [], int, int);
void buildHeap(T [], int, int);
void heapify(T [], int, int);
void zamiana(T [], int, int);
inline int left(int i);
inline int right(int i);
int isSorted(T [], int);

int main()
{
    system("cls");
    cout<<"Podaj rozmiar tablicy: ";
    int n;
    cin>>n;
    T* a;
    a = new T[n];
    cout<<"Generowanie tablicy...";
    generate(a, n);
    cout<<"Zakonczono"<<endl;
}

```

```

//wypisz(a, n);
cout<<"Sortowanie tablicy...";
clock_t pocz=clock();
heapSort(a, 0, n);
clock_t kon=clock();
if(isSorted(a, n)==1) cout<<"Tablica została pomyslnie posortowana w czasie "<<(double)(kon-
pocz)/CLOCKS_PER_SEC<<" sekund"<<endl;
    else cout<<"Tablica nie została posortowana"<<endl;
//wypisz(a, n);
system("PAUSE");
return 0;
}

void generate(T a[], int n)
{
    for(int i=0; i<n; i++)
        a[i]=rand()%1000;
}

void wypisz(T a[], int n)
{
    cout<<"Wypisuje tablice: ";
    for(int i=0; i<n; i++) cout<<a[i]<<" ";
    cout<<endl;
}

void heapSort(T a[], int root, int lenght)
{
    int heapSize=lenght;
    buildHeap(a, root, lenght);
    for(int i=lenght; i>root; i--)
    {
        zamiana(a, root, i);
        heapSize--;
        heapify(a, root, heapSize);
    }
}

void buildHeap(T a[], int root, int lenght)
{
    int heapSize=lenght;
    for(int i=lenght/2; i>=0; i--)
        heapify(a, i, heapSize);
}

void heapify(T a[], int i, int heapSize)
{
    int largest;
    int l=left(i);
    int r=right(i);
    if(l<=heapSize && a[l]>a[i]) // if l <= heapSize is not true the a[l] > a[i] compare is not being made(!)
        largest=l;
    else largest=i;
    if(r<=heapSize && a[r]>a[largest]) // the same here
        largest=r;
    if(largest!=i)
    {
        zamiana(a, i, largest);
        heapify(a, largest, heapSize);
    }
}

```

```

}

void zamiana(T a[],int x, int y)
{
    T p=a[x];
    a[x]=a[y];
    a[y]=p;
}

inline int left(int i)
{
    return 2*i;
}

inline int right(int i)
{
    return 2*i+1;
}

int isSorted(T a[], int n)
{
    int f=1;
    for(int i=0; i<n; ++i) if(a[i]>a[i+1]) f=0;
    return f;
}

```

Quick sorting by Hoar

This one has a slight improvement in the place where it chooses the element that will be a divisor between the tables of recursive calls. It chosen by gathering the first, the last and the middle element of the table and then We're choosing a median of this three elements. This improvement is most effective in randomly spread data, as You can see the most improvement was in the worst case of elements with two extreme's. The improvement because of its overhead is only effective when we are working with really big table's, where the overhead is minimized by the lesser amount of comparisons and sometimes exchanges;

The code:

```

Program QuickSort;
Const
    n = 20;
Var
    A : array[1..n] of Data;
    k : array[1..3] of Data;
    cmp, exc : Integer;

Procedure Swap(i, j : Integer);
Var
    b : Data;
Begin
    exc := exc + 1;
    b := a[i];
    a[i] := a[j];
    a[j] := b
End;

```

```

Procedure Hoare(L, R : Integer);
Var
  left, right, i, j : Integer;
  x, b : Data;
Begin
  If L < R
  then begin
    k[1] := A[L];
    k[2] := A[(L + R) div 2];
    k[3] := A[R];

    For i := 2 downto 1 do
      For j := 1 to i do
        If k[j] > k[j+1]
        then begin
          b := k[j+1];
          k[j+1] := k[j];
          k[j] := b
        end;
      x := k[2];

    left := L;
    right := R ;
    Repeat
      cmp := cmp + 1;
      While A[left] < x do
        begin
          cmp := cmp + 1;
          left := left + 1;
        end;
      cmp := cmp + 1;
      While A[right] > x do
        begin
          cmp := cmp + 1;
          right:=right - 1;
        end;
      If left <= right
      then begin
        Swap(left, right);
        left := left + 1;
        right := right - 1;
      end
    until left > right;
    Hoare(L, right);
    Hoare(left, R)
  end
End;

```

Begin

```
    exc := 0;
    cmp := 0;
    Hoare(1, n);
End.
```

My implementation of quick sorting.

This implementation quite different, it is optimized to do less elements exchanges but make's more comparisons, but exchanges are more time consuming because data is being written so this implementation is quite fast. Using the median this algorithm speeds in some cases but it's not such fast and because its overhead in choosing median is in real time slower.

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

```
using namespace std;
```

```
typedef long T;
```

```
void generate(T [], int);
void wypisz(T [], int);
void quickSort(T [], int, int);
int partition(T [], int, int);
void zamiana(T [], int, int);
int isSorted(T [], int);
int cmp = 0, exc = 0;
```

```
int main(int argc, char* argv[])
{
    system("cls");
    cout<<"Podaj rozmiar tablicy: ";
    int n;
    cin>>n;
    T* a;
    a = new T[n];
    cout<<"Wczytywanie tablicy...\n";
    for(int i = 0; i < n; i++)
        cin >> a[i];
    //cout<<"Generowanie tablicy...";
    //generate(a, n);
    cout<<"Zakonczone"<<endl;
    wypisz(a, n);
    cout<<"Sortowanie tablicy...";
    clock_t pocz=clock();
    quickSort(a, 0, n);
    clock_t kon=clock();
    if(isSorted(a, n)==1) cout<<"Tablica zostala pomyslnie posortowana w czasie "<<(double)(kon-
pocz)/CLOCKS_PER_SEC<<" sekund"<<endl;
    else cout<<"Tablica nie zostala posortowana"<<endl;
    wypisz(a, n);
    cout << endl << "Porownan: " << cmp;
    cout << endl << "Wymian: " << exc;
    system("PAUSE");
    return 0;
}
```

```

void generate(T a[], int n)
{
    for(int i=0; i<n; i++)
        a[i]=rand()%1000;
}

```

```

void wypisz(T a[], int n)
{
    cout<<"Wypisuje tablice: ";
    for(int i=0; i<n; i++) cout<<a[i]<<" ";
    cout<<endl;
}

```

```

void quickSort(T a[], int left, int right)
{
    if(left<right)
    {
        int mid=partition(a, left, right);
        quickSort(a, left, mid);
        quickSort(a, mid+1, right);
    }
}

```

```

int partition(T a[], int left, int right)
{
    int x=a[(left+right)/2];
    int i=left-1;
    int j=right+1;
    while(true)
    {
        do { j--; cmp++; } while(a[j]>x);
        do { i++; cmp++; } while(a[i]<x);
        if(i<j) zamiana(a,i,j);
        else return j;
    }
}

```

```

void zamiana(T a[],int x, int y)
{
    exc++;
    T p=a[x];
    a[x]=a[y];
    a[y]=p;
}

```

```

int isSorted(T a[], int n)
{
    int f=1;
    for(int i=0; i<n; ++i) if(a[i]>a[i+1]) f=0;
    return f;
}

```