Bubblesort

```
void bubblesort (int* array, int N){
    int t;   //Dummy Element
    bool changed=true;

    while (changed==true){  //Nur weitermachen, solange Array unsortiert
        changed=false;
        for (int i=0;i<N;i++){
            if(array[i]>=array[i+1]){  //swappen
                t=array[i];
                array[i]=array[i+1];
                array[i+1]=t;
                changed=true;
                }
            }
        }
    return;}
```

Hanoi

```
void carry_disc(char source,char target){

    cout << source << " to " << target << '\n';
    return;
    }

void hanoi (int n, char source,char target,char other)
{
  if (n>0)
  {
        hanoi(n-1,source,other,target);
        carry_disc(source,target);
        hanoi(n-1,other,target,source);
        }
        else return;
}
```

Fibonacci

```
int fib(int N){

      if (N==0)
        return 0;
      if (N==1)
        return 1;

      return (fib(N-1)+fib(N-2));
      }
```

Quicksort

```
    void swap(int& a, int& b){  //swaps Elements
        int temp=b;
        b=a;
        a=temp;}

    void quicksort(int* array, int links, int rechts){  //links= linker Rand des Array,etc
        if (rechts-links>1){ //Sortieren im Teilarray nicht möglich, weil zu klein
         int pivot = array[links];    //Pivot an erster Stelle des Array wählen
         int l=links+1;
         int r=rechts;
         while(l<r){  //solange Grenzen nicht übereinander laufen
                if (array[l]<= pivot){
                    l++;}
                else {r--;
                     swap(array[l],array[r]);
                     }
                }
                l--;
         swap(array[links],array[l]);
         quicksort(array,links,l);  //Rekursiv den linken Teil abarbeiten
         quicksort(array,r,rechts);  //Rekursiv den rechten Teil abarbeiten
         }
    return;}
```

Fibonacci (mit Ausgabe)

```
#include<iostream>
using namespace std;

            char tab= '\t';
            int i;

void forward(int n, int depth){
            for(i=0;i<depth;i++){cout << tab;}
            if(depth==0){cout << "fib("<< n << ")"<< endl;}
            else{cout << "--> fib("<< n << ")"<< endl;}
            }

void backward(int n,int back, int depth){
            for(i=0;i<depth;i++){cout << tab;}
            if(depth==0){cout << "Endergebnis:"<< back<< endl;}
            else{cout << "<--"<< back<< endl;}
            }

int fib(int n, int depth){
            int back; //wert von fib in gegebener Tiefe depth
            forward(n,depth);
                        if(n==0) back=0;
                        if(n==1) back=1;
                        if(n>1) back=fib(n-1,depth+1)+fib(n-2,depth+1);
                        backward(n,back,depth);
            return back;
            }



int main(){
```

```
	int n;
	int depth=0;
	cout << "Fibonacci-Folge fuer wievieltes Glied? ";
	cin >> n;
		fib(n,depth);
	}
```

Binary Tree

```
#include <iostream>
using namespace std;

struct Tree {
	Tree *left, *right;
	int key;

	Tree() {
		left=right=0;
		key=-1;
	}
};

void Insert(Tree* &tree, int value) {
	if (tree==0) {
		tree = new Tree;
		tree->key=value;
		return;
	}

	if (tree->key>value) {
		Insert(tree->left, value);
	} else {
		Insert(tree->right, value);
	}
}

void Delete(Tree* &tree, int value) {
	if (tree==0) return;
	if (tree->key == value) {

		// test if one of the leaves
		// is equal to null
		if ((tree->left==0) || (tree->right==0)) {
			// store the element to delete in a temporary
			Tree* elem = tree;
			if (tree->left==0) {
				tree=tree->right;
			} else {
				tree=tree->left;
			}

			delete elem;
			return;
		}
		// second case where both subtrees are
		// not empty.
		// search the smallest element in the
		// right subtree

			Tree **p = &tree->right;
			while ((*p)->left != 0) {
				// get the address of the left tree
				p=&((*p)->left);
			}

			// store the minimum in a temporary pointer
			Tree *tmp = (*p);
			// store the element to delete in a temporary
			Tree *elem = tree;

			// remove the minimum out of the tree by
			// draging up the right subtree of the minimum
			(*p)=(*p)->right;
			// link the minimum into the tree where we
			// wanted to delete
			tmp->left=tree->left;
			tmp->right=tree->right;
			tree=tmp;

			// delete the node from the tree
			delete elem;

	} else {
		// delete recursively in the tree
		if (tree->key>value) {
			Delete(tree->left, value);
		} else {
			Delete(tree->right, value);
		}
	}
}

int SearchMinimum(Tree* tree) {
	// assumend all values bigger than 0
	if (tree==0) return 0;

	if (tree->left!=0) {
		return SearchMinimum(tree->left);
	}
	return tree->key;
}




int SearchMaximum(Tree* tree) {
	// assumend all values bigger than 0
	if (tree==0) return 0;

	if (tree->right!=0) {
		return SearchMaximum(tree->right);
	}
	return tree->key;
}
```

```
int Sum(Tree* tree) {
	if (tree==0) return 0;

	return tree->key + Sum(tree->left) + Sum(tree->right);
}

void print(Tree* tree) {
	if (tree!=0) {
		cout << tree->key << endl;
		print(tree->left);
		print(tree->right);
	}
}
```

Linked List

```
struct List {
	List* next;
	int value;

	List() {
		value=-1;
		next=0;
	}
};

void Insert(List *&current, int value) {
	if (current==0) {
		// we are at the tail of the list
		// so just insert the value

		current = new List;
		current->value=value;

		return;
	}

	if (current->value>=value) {
		// the value of the current element is bigger
		// so insert the new value just before

		List *elem = new List;
		elem->value=value;

		elem->next=current;
		current=elem;
	} else {
		// go on recursively

		Insert(current->next, value);
	}
}

void Delete(List *&list, int value) {

	// handle empty list
	if (list==0) return;

	// handle the head of the list
	if (list->value==value) {
		List *tmp=list;
		list=list->next;

		delete tmp;
		return;
	}

	List *cur=list;
	while (cur->next != 0) {
		if (cur->next->value == value) {
			List *tmp = cur->next;
			cur->next=cur->next->next;
			delete tmp;

			return;
		}

		cur=cur->next;
	}
}

void printList(List* list) {
	if (list!=0) {
		cout << list->value << endl;
		printList(list->next);
	}
}

bool contains(List* l1, List* l2) {
	if (l1==0 || l2==0) {
		return false;
	}

	List* t1=l1;
	List* t2=l2;

	while(t1!=0 && t2!=0 && t1->value==t2->value) {
		t1=t1->next;
		t2=t2->next;
	}

	if (t2==0) {
		return true;
	}

	return contains(l1->next, l2);
}

void evaluate(bool b) {
	if (b) {
		cout << "true" << endl;
	} else {
		cout << "false" << endl;
	}
}
```

Klassen - Vererbung

```
#include <iostream>
using namespace std;

class CPolygon {
  protected:
    int width, height;
  public:
    void set_values (int a, int b)
      { width=a; height=b;}
  };

class CRectangle:   public CPolygon {
  public:
    int area ()
      { return (width * height); }
  };

class CTriangle:   public CPolygon {
  public:
    int area ()
      { return (width * height / 2); }
  };

int main () {
  CRectangle rect;
  CTriangle trgl;
  rect.set_values (4,5);
  trgl.set_values (4,5);

  cout << rect.area() << endl;
  cout << trgl.area() << endl;

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

Mehrfachvererbung:

```
class CRectangle: public CPolygon, public COutput {
...
```

Class Time

```
#include <iostream>
using namespace std;

class Time{
private:
  int hour;
  int min;
public:
  Time(){hour =0; min =0;};
  Time & setHour(int h){hour =h; return *this;};
  Time & setMinute(int m){min =m; return *this;};

  void print_time(){cout << hour<< '\t' << min  << endl;};
};
```

```
int main(){
  Time t;
  t.setHour(3);
  t.print_time();
  t.setHour(2).setMinute(51);
  t.print_time();

  cin.get();
  return 0;
}
```