**REGEX ESPRESSIONE :**  String pattern = "[(\\d+|[\\+\\-\\\*\\/\\(\\)])]((\\d+|%5b\\+\\-\\*\\/\\(\\)%5d))+";

**REGEX INTERI :**  String INTERO="\\-?\\d+";

**REALI:** String REALE="\\-?(\\d+|(\\d+)?\\.\\d+)([eE][\\-\\+]?\\d{1,3})?[DdFf]?";

**REGEX IDENTIFICATORE JAVA :** String IDENTIFICATORE="[a-zA-Z\_][a-zA-Z0-9\_\\$]\*";

**REGEX POLINOMI :** String MONOMIO="(\\d+|x(\\^\\d+)?|\\d+x(\\^\\d+)?)";

String SEGNO="[\\+\\-]";

String POLINOMIO="\\-?"+MONOMIO+"("+SEGNO+MONOMIO+")\*";

**REVERSE POLISH NOTATION**

**public** **void** visitaPerLivelli(List<T> ls) {

CodaConcatenata<Nodo<T>> coda=**new** CodaConcatenata<>();

coda.offer(radice);

**while** (!coda.isEmpty()) {

Nodo<T> n=coda.poll();

ls.add(n.info);

**if** (n.fs!=**null**)

coda.offer(n.fs);

**if** (n.fd!=**null**)

coda.offer(n.fd);

}

}

**MERGE SORT**

**public** **static** <T **extends** Comparable<? **super** T>> **void** mergeSort(T[] v) {

**if** (v.length==0)

**throw** **new** IllegalArgumentException();

**if** (v.length==1)

**return**;

*mergeSortIte*(v,0,v.length-1);

}

**private** **static** <T **extends** Comparable<? **super** T>> **void** mergeSort(T[] v, **int** inf, **int** sup) {

**if** (inf<sup) {

**int** med = (inf+sup)/2;

*mergeSort*(v,inf,med);

*mergeSort*(v,med+1,sup);

*merge*(v,inf,med,sup);

}

}

**private** **static** <T **extends** Comparable<? **super** T>> **void** merge(T[] v, **int** inf, **int** med, **int** sup) {

T[] aus=(T[]) **new** Comparable[sup-inf+1];

**int** i=inf, j=med+1, k=0; //k è l'indice di aus che parte da zero obv

**while** (i<=med && j<=sup) {

**if** (v[i].compareTo(v[j])<0) { //v[i] è più piccolo v[j]

aus[k]=v[i];

i++; k++;

}

**else** {

aus[k]=v[j]; j++; k++;

}

}

//gestione del residuo sul primo o sul secondo segmento

**while** (i<=med) {

aus[k]=v[i]; i++; k++;

}

**while** (j<=sup) {

aus[k]=v[j]; j++; k++;

}

//riscrivo aus su v

**for** (**int** h=0; h<aus.length; ++h) {

v[h+inf]=aus[h];

}

}

**private** **static** **enum** Op {***m***, ***ms***};

**private** **static** <T **extends** Comparable<? **super** T>> **void** mergeSortIte(T[] v, **int** inf, **int** sup) {

**class** Frame {

Op x; **int** inf; **int** med; **int** sup;

**public** Frame(Op y, **int** in, **int** me, **int** su) {

x=y; inf=in; sup=su; med=me;

}

}

Stack<Frame> stack = **new** StackConcatenato<>();

stack.push(**new** Frame(Op.***ms***,inf,0,sup));

**while** (!stack.isEmpty()) {

Frame ad=stack.pop();

**if** (ad.x==Op.***m***)

*merge*(v,ad.inf, ad.med, ad.sup);

**else** {

**if**(ad.inf<ad.sup) { //altrimenti gira all'infinito

ad.med=(ad.inf+ad.sup)/2;

stack.push(**new** Frame(Op.***m***,ad.inf, ad.med,ad.sup));

stack.push(**new** Frame(Op.***ms***,ad.med+1,0,ad.sup));

stack.push(**new** Frame(Op.***ms***,ad.inf,0,ad.med));

}

}

}

}

**HEAP SORT**

**public** **static** <T **extends** Comparable<? **super** T>> **void** heapSort(T[] v) {

Heap<T> h=**new** Heap<>(v.length); //non trabocca mai se ci metto esattamente la length di v

//prima fase: riempimento heap

**for**(**int** i=0; i<v.length; ++i)

h.add(v[i]);

//seconda fase: svuoto l'heap

**for** (**int** i=0; i<v.length; i++)

v[i]=h.remove(); //ci mette di volta in volta il minimo

//complessità di heap sort è nlogn

//fisso l'estremo superiore trovo il massimo dell'array e lo metto nell'ultima posizione fino all'inizio di v

**SELECTION SORT**

**public** **static** **void** selectionSort(**int**[] v) {

**for** (**int** j=v.length-1; j>0; j--) {

**int** indMax=0;

**for** (**int** i=1; i<=j; i++) //devo arrivare a j perchè potrebbe essere esso stesso il massimo

**if** (v[i]>v[indMax]) indMax=i;

**int** park=v[j];

v[j]=v[indMax];

v[indMax]=park;

}

}

//stesso per i double

**public** **static** **void** selectionSort(**double**[] v) {

**for** (**int** j=v.length-1; j>0; j--) {

**int** indMax=0;

**for** (**int** i=1; i<=j; i++)

**if** (v[i]>v[indMax]) indMax=i;

**double** park=v[j];

v[j]=v[indMax];

v[indMax]=park;

}

}

//stesso per i comparable

**public** **static** <T **extends** Comparable<? **super** T>> **void** selectionSort(T[] v) { //posso instanziare una classe che implementa comparable

**for** (**int** j=v.length-1; j>0; j--) {

**int** indMax=0;

**for** (**int** i=1; i<=j; i++) //devo arrivare a j perchè potrebbe essere esso stesso il massimo

**if** (v[i].compareTo(v[indMax])>0) indMax=i;

**if** (indMax == j) **continue**;

T park=v[j];

v[j]=v[indMax];

v[indMax]=park;

}

}

**BUBBLE SORT**

**public** **static** **void** bubbleSort(**int**[] v)

**boolean** scambi=**true**;

**int** limite=v.length-1;

**int** ius=-1; //indice ultimo scambio

**while** (scambi) {

scambi=**false**;

**for** (**int** i=0; i<limite; i++)

//mi serve fare il confronto tra penultimo e ultimo

**if** (v[i]>v[i+1]) {

**int** tmp=v[i]; v[i]=v[i+1]; v[i+1]=tmp;

ius=i; scambi=**true**;

//faccio tante scansioni a coppie fino a quando non ho scambi

}

limite=ius;

//non lo decremento di uno ma lo faccio diventare l'ultima posizione

}

}

//stesso per i double

**public** **static** **void** bubbleSort(**double**[] v) {

**boolean** scambi=**true**;

**int** limite=v.length;

**int** ius=-1;

**while** (scambi) {

scambi=**false**;

**for** (**int** i=0; i<limite; i++)

**if** (v[i]>v[i+1]) {

**double** tmp=v[i]; v[i]=v[i+1]; v[i+1]=tmp;

ius=i; scambi=**true**;

}

limite=ius;

}

}

//stesso per i comparable

**public** **static** <T **extends** Comparable<? **super** T>> **void** bubbleSort(T[] v) {//confronta 1 e 2, 2 e 3, etc e li mette in ordine

**boolean** scambi=**true**;

**int** limite=v.length-1; //in questo modo accorcio di volta in volta la scansione, correttezza funzionale

**int** ius=-1; //indice ultimo scambio

**while** (scambi) {

scambi=**false**;

**for** (**int** i=0; i<limite; i++) //mi serve fare il confronto tra penultimo e ultimo

**if** (v[i].compareTo(v[i+1])>0) {

T tmp=v[i]; v[i]=v[i+1]; v[i+1]=tmp;

ius=i; scambi=**true**; //faccio tante scansioni a coppie fino a quando non ho scambi

}

limite=ius; //non lo decremento di uno ma lo faccio diventare l'ultima posizione

}

}

**INSETION SORT**

//ordino in base ad un elemento ad i, tutti i maggiori a destra tutti i minori a sinistra

**public** **static** **void** insertionSort(**int**[] v) {

**for** (**int** i=0; i<v.length; i++) {

**int** x=v[i];

**int** j=i;

**while** (j>0 && v[j-1]>x) {

v[j]=v[j-1]; j--; //sposto in avanti v[j-1]

}

v[j]=x; //pongo x in v[j]

}

}

//stesso per i double

**public** **static** **void** insertionSort(**double**[] v) {

**for** (**int** i=0; i<v.length; i++) {

**double** x=v[i];

**int** j=i;

**while** (j>0 && v[j-1]>x) {

v[j]=v[j-1]; j--;

}

v[j]=x;

}

}

//stesso per i comparable

**public** **static** <T **extends** Comparable<? **super** T>> **void** insertionSort(T[] v) {

**for** (**int** i=0; i<v.length; i++) {

T x=v[i];

**int** j=i;

**while** (j>0 && v[j-1].compareTo(x)>0) {

v[j]=v[j-1]; j--;

}

v[j]=x;

}

}

**POP E PUSH DI STACK CONCATENATO**

**public** **void** push(T x) {

Nodo<T> n=**new** Nodo<>();

n.info=x; n.next=testa;

testa=n; size++;

}

**public** T pop() {

**if** (testa==**null**)

//lista vuota

**throw** **new** NoSuchElementException();

T x=testa.info;

testa=testa.next;

size--;

**return** x;

}