lektion10 inför tentan
lektion10
#include<stdexcept>

Topics

Search

linearSearch
BinarySearch (presorted)

Sort

BubbleSort
InsertionSort
SelectionSort
QuickSort

==========
Search
==========

-[Linear & Binary Search]-

The linear search is a very simple algorithm. Sometimes called a sequential search, it uses a loop to sequentially step through an array, starting with the first element. It compares each element with the value being searched for, and stops when either the value is found or the end of the array is encountered. If the value being searched for is not in the array, the algorithm will search to the end of the array.

<Abstract>

```
found = false
position = -1
index = 0
while index < number of elements and found is false
      if list[index] is equal to search value
            found = true
            position = index
      End if
      Add 1 to index
End while
Return position
```

<\Abstract>

<C++>

```
template <typename T>//only one function at a time.
int linear_search(T arr[], int SIZE, T key)
{
    int pos=-1;
    for(int i=0; i < SIZE; i++)
    {
        if(arr[i]==key)
        {
            pos=i;
        }
    }
    return pos +1;
}
```

<\C++>

The binary search is a clever algorithm that is much more efficient than the linear search.Its only requirement is that the values in the array be in order. Instead of testing the array's first element, this algorithm starts with the element in the middle. If that element happens to contain the desired value, then the search is over.

Otherwise, the value in the middle element is either greater than or less than the value being searched for. If it is greater than the desired value then the value (if it is in the list) will be found somewhere in the first half of the array.

If it is less than the desired value then the value (again, if it is in the list) will be found somewhere in the last half of the array. In either case, half of the array 's elements have been eliminated from further searching.
If the desired value wasn't found in the middle element, the procedure is repeated for the half of the array that potentially contains the value.

For instance, if the last half of the array is to be searched, the algorithm immediately tests its middle element. If the desired value isn't found there, the search is narrowed to the quarter of the array that resides before or after that element. This process continues until the value being searched for is either found or there are no more elements to test.

<Abstract>

```
binear search (sorted array)
  0123456789
  find middle
  search for Key
  key < middle?
  look at greater part than middle
    6789
  new middle
     89
  return 6+1
```

<\Abstract>

<C++>

template <typename T>//only one function at a time.

```
int binary_search(T arr[], int SIZE, T key)
{
    int start =0;
    int end = SIZE -1;
    int mid=(start+end)/2;
    while(end >= start && arr[mid] != key)
    {
        if(key > arr[mid])
        {
            start = mid +1;
        }
        else
        {
            end = mid-1;
        }
        mid=(start+end)/2;
    }
    if(start > end)
    {
        mid= -1;
    }
    return mid+1;
}
```

<\C++>

==========

Sorting

==========

---[BubbleSort]---

<abstract>

```
Do
  Set swap flag to false
  For count =0 to the next-to-last array subscript
      If array[count] is greater than array[count + 1]
            Swap the contents of array[count] and array[count+1]
            Set swap flag to true
      End If
  End For
While the swap flag is true
```

<\abstract>

<C++>

```
template <typename T>
void bubble_sort(T arr[], int SIZE)
{
    T tmp;
    for(int i=0; i < SIZE-1; i++)
    {

        for(int j=0; j < SIZE-1;j++)
        {

          if(arr[j] > arr[j+1])
            {

              tmp = arr[j];
              arr[j] = arr[j+1];
              arr[j+1] = tmp;

            }

        }

    }
}
```

```
                                    // alt 2 better
void boubbleSortArray(int array[], int size)
{

      int temp;
      bool swap;
      do
      {
         swap = false;
         for (int count = 0; count < (size-1); count++)
         {
            temp = array[count];
            array[count] = array[count + 1];
            array[count + 1] = temp;
            swap = true;
         }
      }
}while(swap);
```

<\C++>

------[InsertionSort]--------
<abstract>

Original	34	8	64	51	32	21	Positions Moved
After $p = 1$	8	34	64	51	32	21	1
After $p = 2$	8	34	64	51	32	21	0
After $p = 3$	8	34	51	64	32	21	1
After $p = 4$	8	32	34	51	64	21	3
After $p = 5$	8	21	32	34	51	64	4

Figure 7.1 Insertion sort after each pass

<\abstract>
<C++>

```
template <typename T>//only one function at a time.
void insertion_sort(T arr[], int SIZE)
{
    T tmp;
    for(int i=0; i < SIZE; i++)
    {
        //check all previously elements and start looping at i = 1
        //if j = 0 then j-1 gives segmentationFault
        for(int j=i; j > 0 && arr[j-1] > arr[j];j--)
        {
            //swap(arr[i],arr[i+1]);
            tmp = arr[j];
            arr[j] = arr[j-1];
            arr[j-1] = tmp;
        }

    }
}
```

```
                                        //alt 2
/*
* Simple insertion sort.
*/
template <typename Comparable>
void insertionSort( vector<Comparable> & a )
{
      for( int p = 1; p < a.size( ); ++p )
      {
            Comparable tmp = std::move( a[ p ] );
            int j;
            for( j = p; j > 0 && tmp < a[ j - 1 ]; --j )
                  a[ j ] = std::move( a[ j - 1 ] );
            a[ j ] = std::move( tmp );
      }
}
```

<\C++>

----[SelectionSort]----

The selection sort scans the array,
starting at element 0,
locates the element with the smallest value.
The contents of this element are then swapped with the contents of element 0.

The algorithm then repeats the process,
but because element 0 already contains the smallest value in the array, it can be left out of the procedure .

For the second pass, the algorithm begins
the scan at element 1. It locates the smallest value in the unsorted part of the array.

Once again the process is repeated, but this time the scan begins at element 2.

<abstract>

```
For startScan= 0 to the next-to-last array subscript
     Set index to startScan
     Set minindex to startScan
     Set minValue to array{startScan}
     For index = (startScan + 1) to the last subscript in the array
          If array[index} is less than minValue
               Set minValue to array[index}
               Set minindex to index
          End If
          Increment index
     End For
     Set array[minindex} to array{startScan}
     Set array{startScan} to minValue
End For
```

<\abstract>

<C++>

```
template <typename T>
//sort low 2 big
void selection_sort(T arr[], int SIZE)
{

        T temp;
        int min_pos;
        for(int i =0; i < SIZE; i++)
        {
            //find minimum
            min_pos=i; //Could init with -1, nothing found
            //don't need to check the last one.
            for(int j=i+1; j < SIZE; j++)
            {
                if(arr[j] <= arr[min_pos])
                {
                    min_pos=j;
                }
            }
            //swap(arr[min_pos], arr[i]);
            if(min_pos != i)
            {
                //does swap.
                temp=arr[min_pos];
                arr[min_pos]=arr[i];
                arr[i]=temp;
            }
        }
}
```

<\C++>

---[QuickSort]----

<C++>

```
template <typename T>
void quickSort(T arr[], int start, int end)
{
    if(start < end)
    {
        int pivot = partition(arr, start, end);
        quickSort(arr, start, pivot-1);
        quickSort(arr, pivot+1, end);
    }

}
```

partition

```
template <typename T>
int partition(T theArray[], int start, int end)
{
    T pivotValue = theArray[start];
    int pivotPos=start;
    for(int i=start+1; i<=end;i++)
    {
        if(theArray[i]<pivotValue)
        {
            swapI(theArray[i], theArray[pivotPos+1]);
            swapI(theArray[pivotPos], theArray[pivotPos+1]);
            pivotPos++;
        }
    }
    return pivotPos;
}
```

swap

```
//swap n1 value with n2 value
template <typename T>
void swapI(T &n1, T &n2)
{
    T tmp = n1;
    n1 = n2;
    n2 = tmp;
}
```

<\C++>
