CS101 Computer Programming and Utilization

Milind Sohoni

June 13, 2006
So far

What is sorting

Bubble Sort

Other Sorts

Searching
The story so far ...

- We have seen various control flows.
- We have seen multi-dimensional arrays and the `char` data type.
- We saw the use of functions and calling methods.
- We have seen structs and file input/output.

This week...

Sorting and Searching
A Problem

Recall that we have the struct:

```c
struct student
{
    char name[6];
    char roll[8];
    int hostel;
}
```

Suppose next that we have a list (array) of students and we wish to

- Insert into that list.
- Delete from that list.
- Check if present in that list.
A Problem

Recall that we have the struct:

```c
struct student {
    char name[6];
    char roll[8];
    int hostel;
}
```

Suppose next that we have a list (array) of students and we wish to

- Insert into that list.
- Delete from that list.
- Check if present in that list.

It is then clear that we must store the array in a sorted order.
What does sorting mean?

- Every element of the list must have a key on which the sorting will be done.
- For two keys $k_1$ and $k_2$, we must have that either $k_1 < k_2$, or $k_1 = k_2$ or $k_1 > k_2$. This is called total ordering.
- Sorting then means that arranging the in the order $s_1, \ldots s_n$ so that $k_1 \leq k_2 \leq \ldots \leq k_n$.

For our example, the alphabetical ordering is the required total order.
Sorting

Let us assume, for simplicity, we have a struct called key and a function which implements the total order:

\[
\text{int CompareKey(key k1, k2), which returns}
\]

- 1 if \( k_1 > k_2 \).
- 0 if \( k_1 = k_2 \).
- -1 if \( k_1 < k_2 \).

Our problem then is to sort an array of keys.

Let us first write this CompareKey for char name[6].
Sorting

Let us assume, for simplicity, we have a `struct` called `key` and a function which implements the total order:

- `int CompareKey(key k1, k2)`, which returns
  - 1 if \( k_1 > k_2 \).
  - 0 if \( k_1 = k_2 \).
  - -1 if \( k_1 < k_2 \).

Our problem then is to sort an array of keys.

Let us first write this `CompareKey` for `char name[6]`.

```c
#include <iostream.h>
int main()
{
    char name1[7], name2[7];
    int i;
    cout << "student names?\n";
    cin >> name1 >> name2;
    for (i=0; i<7; i=i+1)
    {
        if (name1[i]<name2[i])
        {
            cout << "-1 \n"; return 0;
        }
        if (name1[i]>name2[i])
        {
            cout << "1 \n"; return 0;
        }
    } // of for
    cout << "0 \n"; return 0;
}
```

Milind Sohoni ()
CS101 Computer Programming and Utilization
June 13, 2006
Sorting

Whats happening?

If \( x_1 > x_2 \) return 1.

If \( x_1 < x_2 \) return -1.

If \( x_1 = x_2 \) then increment i.

If \( i \) == 7 then return 0.

#include <iostream.h>
int main()
{
    char name1[7], name2[7];
    int i;
    cout << "student names?\n";
    cin >> name1 >> name2;
    for (i=0; i<7; i=i+1)
    {
        if (name1[i]<name2[i])
        {
            cout << "-1 \n"; return 0;
        }
        else if (name1[i]>name2[i])
        {
            cout << "1 \n"; return 0;
        }
    } // of for
    cout << "0 \n"; return 0;
}
Bubble Sort

Now that key comparison is clear, let us now sort an array of integers. Obviously this algorithm can be used to sort any key.

We look at the bubble sort whose basic step is the flip(i):

- Compare two adjacent elements $x_{i-1}, x_i$.
- If $x_i > x_{i-1}$ then interchange.

... ... 4 3 ... ...

↓

... ... 3 4 ... ...
Bubble Sort

Now that key comparison is clear, let us now sort an array of integers. Obviously this algorithm can be used to sort any key.

We look at the bubble sort whose basic step is the flip(i):

- Compare two adjacent elements $x_{i-1}, x_i$.
- If $x_i > x_{i-1}$ then interchange.

A Phase(N-1) is a sequence of flips:

$flip(1), flip(2), \ldots, flip(N-1)$

\[
\begin{array}{cccccccc}
3 & 1 & 4 & 3 & 1 & 4 \\
\downarrow & & & & & & &
\end{array}
\]

\[
\begin{array}{cccccccc}
1 & 3 & 4 & 3 & 1 & 2 \\
\downarrow & & & & & & &
\end{array}
\]

\[
\begin{array}{cccccccc}
1 & 3 & 4 & 3 & 1 & 2 \\
\downarrow & & & & & & &
\end{array}
\]

\[
\begin{array}{cccccccc}
1 & 3 & 3 & 4 & 1 & 2 \\
\downarrow & & & & & & &
\end{array}
\]

\[
\begin{array}{cccccccc}
1 & 3 & 3 & 1 & 4 & 2 \\
\downarrow & & & & & & &
\end{array}
\]

\[
\begin{array}{cccccccc}
1 & 3 & 3 & 1 & 2 & 4 \\
& & & & & & &
\end{array}
\]
Thus, we see that:

- At the end of Phase(N-1), the largest element is in the last location.

We may thus run:

- Phase(N-1) which fixes the (N-1)-th element.
- Phase(N-2) which fixes the (N-2)-th element.

\[ \vdots \]

- Phase(1) which fixes the (1)-th element. and obtain:
Thus, we see that:

- At the end of Phase(N-1), the largest element is in the last location.

We may thus run:

- Phase(N-1) which fixes the (N-1)-th element.
- Phase(N-2) which fixes the (N-2)-th element.

... 
- Phase(1) which fixes the (1)-th element. and obtain:

\[
\begin{array}{ccccccc}
3 & 1 & 4 & 3 & 1 & 4 \\
\downarrow & \text{Phase(5)} & \downarrow & \text{Phase(4)} & \downarrow & \text{Phase(3)} & \downarrow & \text{Phase(2)} & \downarrow & \text{Phase(1)} \\
1 & 3 & 3 & 1 & 2 & 4 \\
1 & 3 & 1 & 2 & 3 & 4 \\
1 & 1 & 2 & 3 & 3 & 4 \\
1 & 1 & 2 & 3 & 3 & 4 \\
1 & 1 & 2 & 3 & 3 & 4 \\
\end{array}
\]

This sorts the array.
#include <iostream.h>
void sort(int c[], int N) {
    int i, j, temp;
    for (i=N-1; i>=1; i=i-1) // beginning Phase (i)
        for (j=1; j<=i; j=j+1) // beginning flip (j)
            if (c[j] < c[j-1])
                { temp = c[j]; c[j] = c[j-1];
                c[j-1] = temp;
                };
}

i is counting phase.

j is counting flip.

array c is passed by reference.
#include <iostream.h>
void sort(int c[],int N)
{
    int i,j,temp;
    for (i=N-1;i>=1;i=i-1)
        // beginning Phase (i)
        for (j=1;j<=i;j=j+1)
            // beginning flip (j)
            if (c[j]<c[j-1])
                { temp=c[j]; c[j]=c[j-1];
                  c[j-1]=temp;
                }
    }

\* i is counting phase.
\* j is counting flip.
\* array c is passed by reference.

**Question**  How many steps does it take to sort an array of size N

**Answer**  \((N – 1) + (N – 2) + \ldots + 1 = \frac{N(N – 1)}{2}\)

*Thus it takes quadratic, i.e., \(O(N^2)\) time to bubble-sort.*
Other Sorts

There are faster ways to sort:

- Merge-Sort, Heap-Sort, \( O(N \log N) \).
- Quick-Sort, expected time \( O(N \log N) \).

All of these are fairly simple but clever. We will look at Merge-Sort though, not in detail.
Other Sorts

There are faster ways to sort:

- **Merge-Sort, Heap-Sort, $O(N \log N)$**.
- **Quick-Sort**, expected time $O(N \log N)$.

All of these are fairly simple but clever. We will look at **Merge-Sort** though, not in detail. **Merge** has three basic steps:

- **Split** the given array $A$ into two equal halves $A_1$ and $A_2$.
- **Recursively**, sort $A_1$, $A_2$ to get sorted $B_1$, $B_2$.
- **Merge** $B_1$, $B_2$ to get sorted $B$. 

---

**Diagram Explanation**

- **Split**: Divide the array into two halves.
- **Recursive Sort**: Perform sorting on each half recursively.
- **Merge**: Combine the sorted halves into a single sorted array.
Other Sorts

There are faster ways to sort:
- **Merge-Sort**, **Heap-Sort**, $O(N \log N)$.
- **Quick-Sort**, expected time $O(N \log N)$.

All of these are fairly simple but clever. We will look at **Merge-Sort** though, not in detail. Merge has three basic steps:
- **Split** the given array $A$ into two equal halves $A_1$ and $A_2$.
- **Recursively**, sort $A_1, A_2$ to get sorted $B_1, B_2$.
- **Merge** $B_1, B_2$ to get sorted $B$.

How much time does Merge-Sort take?
Merge-Sort

- **Split** the given array $A$ into two equal halves $A_1$ and $A_2$.
- **Recursively**, sort $A_1$, $A_2$ to get sorted $B_1$, $B_2$.
- **Merge** $B_1$, $B_2$ to get sorted $B$. 

![Diagram](image)
Split the given array \( A \) into two equal halves \( A_1 \) and \( A_2 \).

Recursively, sort \( A_1 \), \( A_2 \) to get sorted \( B_1 \), \( B_2 \).

Merge \( B_1 \), \( B_2 \) to get sorted \( B \).

\[ T(N) \] is the time taken to merge-sort an \( N \)-array.

Split-ting an \( N \)-array into two equal parts is easy. At most a single for loop.

Recursive Merge: this should take time \( 2 \times T(N/2) \).

Merge is the operation of merging two sorted arrays into a single sorted array.

We will see that this takes time \( 2N \).

Thus we have:

\[
T(N) = 3N + 2T(N/2)
\]

We may expand this to check that \( T(N) = O(N \log N) \).
Let us look at the merge-operation:

- **Merge** merges two sorted arrays into a single sorted array.

We use three markers: head1, head2, tail.
Let us look at the merge-operation:

- **Merge** merges two sorted arrays into a single sorted array.

```
void merge(int B1[], B2[], B[], int N1, int N2)
{
    int tail = 0, head1 = 0, head2 = 0;
    while (head1 < N1 && head2 < N2)
    {
        if B1[head1] < B2[head2]
        {
            B[tail] = B1[head1];
            head1 = head1 + 1;
        }
        else
        {
            B[tail] = B2[head2];
            head1 = head1 + 1;
        }
        tail = tail + 1;
    } // of while
}
```

We use three markers: head1, head2, tail.
Merge

Let us look at the merge-operation:

- **Merge** merges two sorted arrays into a single sorted array.

We use three markers:
- head1
- head2
- tail

We can implement the merge-operation as follows:

```c
void merge(int B1[], B2[], B[], int N1, int N2) {
    int tail=0, head1=0, head2=0;
    while (head1<N1 && head2<N2) {
            B[tail]=B1[head1];
            head1=head1+1;
        } else {
            B[tail]=B2[head2];
            head1=head1+1;
        }
        tail=tail+1;
    }
    if (head1==N1) { push B2 }
    else {push B1};
}
```
Search

- Check if the integer $n$ occurs in a sorted array $B$ of size $N$.

The simplest way is to

- Start at the beginning and stop at the end.
Check if the integer \( n \) occurs in a sorted array \( B \) of size \( N \).

The simplest way is to

- Start at the beginning and stop at the end. \( \text{Ignore the sorting.} \)
Search

- Check if the integer $n$ occurs in a sorted array $B$ of size $N$.

The simplest way is to

- Start at the beginning and stop at the end. Ignore the sorting.

- Look at the mid-point of $B$, say it is $k$.
  - if $k = n$ done!
  - if $k < n$, Check($n$,B2).
  - if $k > n$, Check($n$,B1).
Search

- Check if the integer $n$ occurs in a sorted array $B$ of size $N$.

The simplest way is to:

- Start at the beginning and stop at the end. Ignore the sorting.

  - Look at the mid-point of $B$, say it is $k$.
  - if $k = n$ done!
  - if $k < n$, Check($n, B2$).
  - if $k > n$, Check($n, B1$).

\[ \text{B1} \quad \text{k} \quad \text{B2} \]
First ensure that \( c[hi] \neq ip \), \( c[lo] \neq ip \).

Now enter the infinite while loop.

- Compute mid and check that \( c[mid] \neq ip \).
- Check that \( lo, hi \) have a gap.

Now, redefine \( lo, hi \).

```cpp
int search(int c[], int N, int ip) {
    int lo=0, hi=N-1, done=0, mid;
    if (c[lo]==ip) return (lo);
    if (c[hi]==ip) return (hi);
    while (done==0) {
        mid=(lo+hi)/2;
        if (c[mid]==ip) return (mid);
        if (hi-lo<2) return (-1);
        if (c[mid]<ip) {
            lo=mid;
        } else {
            hi=mid;
        }
    }
}
```