COMP 10
EXPLORING COMPUTER SCIENCE

Lecture 7

Searching and Sorting

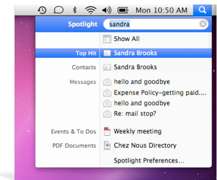
Looking for something…

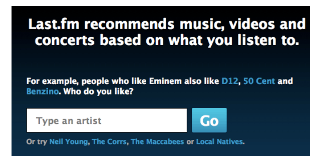
Where is the book "Modern Interiors"?

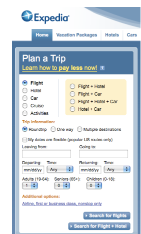
TODAY'S OUTLINE

Searching algorithms
- Linear search
- Complexity

Sorting algorithms
- Bubble sort
- Efficiency







SEARCHING

Looking for something…

Searching an array is a particularly common problem

Goal: Determine whether a particular *value* is in the array

Return its *location* in the array

SEARCHING

Looking for something…

Searching an array is a particularly common problem

Goal: Determine whether a particular *value* is in the array

Return its *location* in the array

Array to be searched: A[ ]

Number of elements: N

Where is 21?

3	12	-5	6	142	21	-17	45
A[0]	A[1]	A[2]	A[3]		...		A[N-1]

SEARCHING: PROBLEM SPECIFICATION

Let

A[ ] be the array to be searched,

N be the number of elements,

x be the value to search for (the target)

Question: Does x occur in A[ ]?

SEARCHING: PROBLEM SPECIFICATION

Let

A[ ] be the array to be searched,

N be the number of elements,

x be the value to search for (the target)

Question: Does x occur in A[ ]?

If x appears in A[0], A[1], ... , A[N-1], determine its index

that is, find the k such that A[k]==x

Otherwise, if x is not found, return -1 (error code)

LINEAR SEARCH: ALGORITHM

Function prototype

```
int LinearSearch(int A[], int size, int target)
```

Start at the beginning of the array

Examine each element until x is found, or until all elements have been examined

Where is 21?

3	12	-5	6	142	21	-17	45
A[0]	A[1]	A[2]	A[3]		...		A[N-1]

LINEAR SEARCH: CODE

```
// search an array A of size size in linear fashion

int LinearSearch (int A[ ], int size, int target) {
  int index = 0;

   while ((index < size) && (A[index] != target)){
     index++;
  }

   if (index < size){
      return index;
  }
   else{
     return -1;
  }
}
```

LINEAR SEARCH: CODE


```
// search an array A of size size in linear fashion

int LinearSearch (int A[ ], int size, int target) {
  int index = 0;

   while ((index < size) && (A[index] != target)){
     index++;
  }

   if (index < size){
      return index;
  }
   else{
     return -1;
  }
}
```


The loop condition is written so that `A[index]` is not accessed if `index >= size`

Why do we need `index < size`?

LINEAR SEARCH: CODE


```
// search an array A of size size in linear fashion

int LinearSearch (int A[ ], int size, int target) {
  int index = 0;

   while ((index < size) && (A[index] != target)){
     index++;
  }

   if (index < size){
      return index;
  }
   else{
     return -1;
  }
}
```


What does && mean?

LINEAR SEARCH: CODE

```
// search an array A of size size in linear fashion

int LinearSearch (int A[ ], int size, int target) {
  int index = 0;

   while ((index < size) && (A[index] != target)){
     index++;
  }

   if (index < size){
      return index;
  }
   else{
     return -1;
  }
}
```

Why do we need `A[index] != target`?

COMPLEXITY

```
int LinearSearch(int A[], int size, int target){
}
```

Examples:

- Where is 6? `LinearSearch(A,8,6) = ?`
- Where is 45? `LinearSearch(A,8,45) = ?`
- Where is 15? `LinearSearch(A,8,15) = ?`

3	12	-5	6	142	21	-17	45
A[0]	A[1]	A[2]	A[3]		...		A[N-1]

COMPLEXITY

```
int LinearSearch(int A[], int size, int target){
}
```

Number of elements to consider in order to give an answer

- Worst case?
- Average?

3	12	-5	6	142	21	-17	45
A[0]	A[1]	A[2]	A[3]		...		A[N-1]

ANALYZING ALGORITHM EFFICIENCY

In an algorithm, each operation has a *computation cost*

Efficiency is an important consideration when programming

ANALYZING ALGORITHM EFFICIENCY

In an algorithm, each operation has a *computation cost*

Efficiency is an important consideration when programming

Analyzing each step informally:

- Count each assignment or calculation as one step
- Loops:
- Conditionals:
- Functions:

ANALYZING ALGORITHM EFFICIENCY

In an algorithm, each operation has a *computation cost*

Efficiency is an important consideration when programming

Analyzing each step informally:

- Count each assignment or calculation as one step
- Loops: Count the maximum number of times it could be executed
- Conditionals:
- Functions:

ANALYZING ALGORITHM EFFICIENCY

In an algorithm, each operation has a *computation cost*

Efficiency is an important consideration when programming

Analyzing each step informally:

- Count each assignment or calculation as one step
- Loops: Count the maximum number of times it could be executed
- Conditionals: Pick the more expensive branch
- Functions:

ANALYZING ALGORITHM EFFICIENCY

In an algorithm, each operation has a *computation cost*

Efficiency is an important consideration when programming

Analyzing each step informally:

- Count each assignment or calculation as one step
- Loops: Count the maximum number of times it could be executed
- Conditionals: Pick the more expensive branch
- Functions: Calling a function is not just one step! You have to trace through the entire code of the method

ANALYZING ALGORITHM EFFICIENCY

In an algorithm, each operation has a *computation cost*

Efficiency is an important consideration when programming

Analyzing each step informally:

- Count each assignment or calculation as one step
- Loops: Count the maximum number of times it could be executed
- Conditionals: Pick the more expensive branch
- Functions: Calling a function is not just one step! You have to trace through the entire code of the method

Rough *upper bound* of how many units of time an algorithm takes

SORTING

The problem: Put things in order

- Usually smallest to largest: *ascending*
- Could also be largest to smallest: *descending*

Many applications:

- ordering hits in web search engine
- preparing lists of output
- merging data from multiple sources
- faster search (e.g. binary search)

Sorting Hat from *Harry Potter*

SORTING

A sorting algorithm arranges the elements of an input list (in our case an array) in a certain order

Sorting helps other algorithms run more efficiently and makes data more readable for humans

The output array must be a permutation (reordering) of the input – no elements lost or added

SORTING: MORE FORMALLY

Given an array A[0], A[1], ... A[n-1]

Reorder entries so that

$A[0] \leq A[1] \leq \ \ldots \ \leq A[n-1]$

Shorthand for these slides: the notation array[i..k] means all of the elements array[i], array[i+1], ... , array[k]

Using this notation, the entire array would be: A[0..n-1]

P.S. This notation is not C++ syntax!

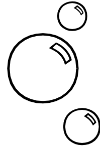
SORTING ALGORITHMS

Sorting has been intensely studied for decades

- Efficiency is important

Many different ways to do it

How would you do it?



We will look at one algorithm called **bubble sort**

THE SORTING PROBLEM

Initial conditions: An unsorted array

Goal: Data sorted in increasing order:

$A[0] \le A[1] \le \ldots \le A[N-1]$

3	12	-5	6	142	21	-17	45

↓ Sort

-17	-5	3	6	12	21	45	142

BUBBLE SORT

Compare the first two elements:

- If they are out of order, swap them

Move down one element: compare the 2nd and 3rd elements:

- If necessary, swap them

Continue until the end of array

Pass through the entire array again

- and again, ... until a full pass with no swaps necessary

BUBBLE SORT EXAMPLE: FIRST PASS

Initial state	3	12	-5	6	142	21	-17	45

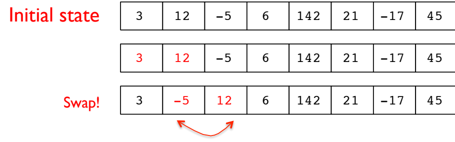
BUBBLE SORT EXAMPLE: FIRST PASS

Initial state	3	12	-5	6	142	21	-17	45
	3	12	-5	6	142	21	-17	45

BUBBLE SORT EXAMPLE: FIRST PASS

Initial state	3	12	-5	6	142	21	-17	45
	3	12	-5	6	142	21	-17	45
	3	12	-5	6	142	21	-17	45

BUBBLE SORT EXAMPLE: FIRST PASS

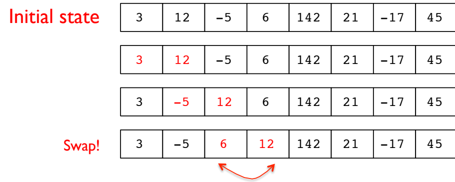


Initial state	3	12	-5	6	142	21	-17	45
	3	12	-5	6	142	21	-17	45
Swap!	3	-5	12	6	142	21	-17	45

BUBBLE SORT EXAMPLE: FIRST PASS

Initial state	3	12	-5	6	142	21	-17	45
	3	12	-5	6	142	21	-17	45
	3	-5	12	6	142	21	-17	45
	3	-5	12	6	142	21	-17	45

BUBBLE SORT EXAMPLE: FIRST PASS



Initial state	3	12	-5	6	142	21	-17	45
	3	12	-5	6	142	21	-17	45
	3	-5	12	6	142	21	-17	45
Swap!	3	-5	6	12	142	21	-17	45

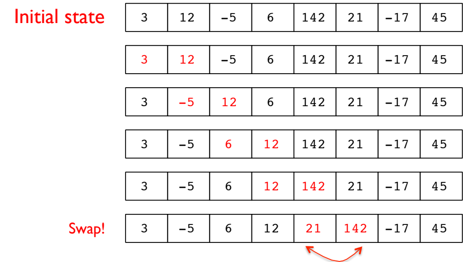
BUBBLE SORT EXAMPLE: FIRST PASS

Initial state	3	12	-5	6	142	21	-17	45
	3	12	-5	6	142	21	-17	45
	3	-5	12	6	142	21	-17	45
	3	-5	6	12	142	21	-17	45
	3	-5	6	12	142	21	-17	45

BUBBLE SORT EXAMPLE: FIRST PASS

Initial state	3	12	-5	6	142	21	-17	45
	3	12	-5	6	142	21	-17	45
	3	-5	12	6	142	21	-17	45
	3	-5	6	12	142	21	-17	45
	3	-5	6	12	142	21	-17	45
	3	-5	6	12	142	21	-17	45

BUBBLE SORT EXAMPLE: FIRST PASS



Initial state	3	12	-5	6	142	21	-17	45
	3	12	-5	6	142	21	-17	45
	3	-5	12	6	142	21	-17	45
	3	-5	6	12	142	21	-17	45
	3	-5	6	12	142	21	-17	45
Swap!	3	-5	6	12	21	142	-17	45

BUBBLE SORT EXAMPLE: FIRST PASS

Initial state	3	12	-5	6	142	21	-17	45
	3	12	-5	6	142	21	-17	45
	3	-5	12	6	142	21	-17	45
	3	-5	6	12	142	21	-17	45
	3	-5	6	12	142	21	-17	45
	3	-5	6	12	21	142	-17	45
	3	-5	6	12	21	142	-17	45

BUBBLE SORT EXAMPLE: FIRST PASS

Initial state	3	12	-5	6	142	21	-17	45
	3	12	-5	6	142	21	-17	45
	3	-5	12	6	142	21	-17	45
	3	-5	6	12	142	21	-17	45
	3	-5	6	12	142	21	-17	45
	3	-5	6	12	21	142	-17	45
Swap!	3	-5	6	12	21	-17	142	45

BUBBLE SORT EXAMPLE: FIRST PASS

Initial state	3	12	-5	6	142	21	-17	45
	3	12	-5	6	142	21	-17	45
	3	-5	12	6	142	21	-17	45
	3	-5	6	12	142	21	-17	45
	3	-5	6	12	142	21	-17	45
	3	-5	6	12	21	142	-17	45
	3	-5	6	12	21	-17	142	45
	3	-5	6	12	21	-17	142	45

BUBBLE SORT EXAMPLE: FIRST PASS

Initial state	3	12	-5	6	142	21	-17	45
	3	12	-5	6	142	21	-17	45
	3	-5	12	6	142	21	-17	45
	3	-5	6	12	142	21	-17	45
	3	-5	6	12	142	21	-17	45
	3	-5	6	12	21	142	-17	45
	3	-5	6	12	21	-17	142	45
Swap!	3	-5	6	12	21	-17	45	142

BUBBLE SORT EXAMPLE: FIRST PASS

Initial state	3	12	-5	6	142	21	-17	45
	3	12	-5	6	142	21	-17	45
	3	-5	12	6	142	21	-17	45
	3	-5	6	12	142	21	-17	45
	3	-5	6	12	142	21	-17	45
	3	-5	6	12	21	142	-17	45
	3	-5	6	12	21	-17	142	45
End of pass 1	3	-5	6	12	21	-17	45	142

BUBBLE SORT EXAMPLE

Initial state	3	12	-5	6	142	21	-17	45
End of pass 1	3	-5	6	12	21	-17	45	142
End of pass 2	-5	3	6	12	-17	21	45	142
End of pass 3	-5	3	6	-17	12	21	45	142
End of pass 4	-5	3	-17	6	12	21	45	142
End of pass 5	-5	-17	3	6	12	21	45	142
End of pass 6	-17	-5	3	6	12	21	45	142

No more swaps

Bubble Sort

Problem Size: 20 · 30 · 40 · 50 Magnification: 1x · 2x · 3x
Algorithm: Insertion · Selection · Bubble · Shell · Merge · Heap · Quick · Quick3

http://sorting-algorithms.com

BUBBLE SORT

Compare the first two elements:

- If they are out of order, swap them

Move down one element: compare the 2nd and 3rd elements:

- If necessary, swap them

Continue until the end of array

Pass through the entire array again

- and again, ... until a full pass with no swaps necessary

WHEN DO WE SWAP?

WHEN DO WE SWAP?

When two adjacent elements are out of order

How do we swap?

WHEN DO WE SWAP?

When two adjacent elements are out of order

How do we swap?

```
int temp;
if (A[count] > A[count+1]) {
      temp = A[count];
      A[count] = A[count+1];
      A[count+1] = temp;
}
```

CODE FOR BUBBLE SORT

```
void BubbleSort1(int A[ ], int size) {
  int temp;
  bool swap = true;

  while (swap) {
      swap = false;

      for (int count = 0; count < size-1; count ++) {

            if (A[count] > A[count+1]) {
                  temp = A[count];
                  A[count] = A[count+1];
                  A[count+1] = temp;
                  swap = true;
            }
      }
  }
}
```

CODE FOR *OPTIMAL* BUBBLE SORT

```
void BubbleSort(int A[ ], int size) {
  int temp;

  for (int i=0; i<size-1; i++) {

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

           if (A[j+1] < A[j]) {
                temp = A[j];
                A[j]= A[j+1];
                A[j+1] = temp;
           }
      }
  }
}
```

1 comparison and 3 assignments each time through

N times through outer loop

and a total of N/2 times through inner loop

LAB PREP: SORT AND SEARCH

1. **Bubble sort**

   The user will enter 10 integers between 0 and 20 in a random order

   Your program will then sort the integers into ascending order using the bubble sort algorithm and print the result.

LAB PREP: SORT AND SEARCH

1. **Bubble sort**

   The user will enter 10 integers between 0 and 20 in a random order

   Your program will then sort the integers into ascending order using the bubble sort algorithm and print the result.

2. **Linear search**

   The user will search for numbers in the array

   The program should report the position in the array where the number was found, if it was found at all

   After each search the program will ask the user if there is another number to search for

LAB PREP: SORT AND SEARCH

You will only have to implement two functions to perform the sorting and the searching

Other code is provided in a template

LAB PREP: VISUALIZATION WITH LEDA

```
void draw_histogram(int A[], int size, int current)
```

Draws a histogram (bar chart) representing the values of the integers in the array A

- The height of each bar corresponds to the value of that array element

Parameters:

- size: number of elements in the array
- current: the index of the array you are currently searching or sorting

The current bar will be drawn in a different color than the others