Ch 8. Searching and Sorting Arrays

8.1 and 8.3 only

CS 2308
Spring 2013

Jill Seaman

Definitions of Search and Sort

- Search: find an item in an array, return the index to the item, or -1 if not found.
- Sort: rearrange the items in an array into some order (smallest to biggest, alphabetical order, etc.).
- There are various methods (algorithms) for carrying out these common tasks.
- Which ones are better? Why?

Linear Search

- Very simple method.
- Compare first element to target value, if not found then compare second element to target value . . .
- Repeat until:
  target value is found (return its index) or
  we run out of items (return -1).

Linear Search in C++

```
int searchList (int list[], int numElems, int value) {

  int index=0;          //index to process array
  int position = -1;    //position of value
  bool found = false;   //flag, true when value is found

  while (index < numElems && !found)
  {
    if (list[index] == value)  //found the value!
    {
      found = true;            //set the flag
      position = index;        //record which item
    }
    index++;                   //increment loop index
  }
  return position;
}
```

What if we don't use found?

Program using Linear Search

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

int searchList(int[], int, int);

int main() {
  const int SIZE=5;
  int idNums[SIZE] = {871, 750, 988, 100, 822};
  int results, id;

  cout << “Enter the employee ID to search for: “;
  cin >> id;

  results = searchList(idNums, SIZE, id);

  if (results == -1) {
    cout << “That id number is not registered\n”;
  } else {
    cout << “That id number is found at location “;
    cout << results+1 << endl;
  }

  return 0;
}
```

Evaluating the Algorithm

- Is it efficient? Does it do any unnecessary work?
- We measure efficiency of algorithms in terms of number of main steps required to finish.
- For search algorithms, the main step is comparing an array element to the target value.
- Number of steps depends on:
  - size of input array
  - whether or not value is in array
  - where the value is in the array

Efficiency of Linear Search

	N=50,000	In terms of N
Best Case:	1	1
Average Case:	25,000	N/2
Worst Case:	50,000	N

*N is the number of elements in the array

Note: if we search for items not in the array, the average case will increase.

Binary Search

- Works only for SORTED arrays
- Divide and conquer style algorithm
- Compare target value to middle element in list.
  - if equal, then return its index
  - if less than middle element, search in first half of list (repeat)
  - if greater than middle element, search in last half of list (repeat)
- If current search list is narrowed down to 0 elements, return -1

Binary Search Algorithm

- The algorithm described in pseudocode:

```
while (number of items in list >= 1) and (target not found)

  if (item at middle position is equal to target)
      target is found!
      location = middle position
  else
      if (target < middle item)        (narrow search list)
          list = lower half of list
      else
          list = upper half of list

end while

if target not found, location = -1
```

Binary Search in C++

```
int binarySearch (int array[], int numElems, int value) {

  int first = 0,              //index to first elem
      last = numElems – 1,    //index to last elem
      middle,                 //index of middle elem
      position = -1;          //index of target value
  bool found = false;         //flag

  while (first <= last && !found) {

    middle = (first + last) /2;    //calculate midpoint

    if (array[middle] == value) {
      found = true;
      position = middle;
    } else if (array[middle] > value) {
      last = middle – 1;              //search lower half
    } else {
      first = middle + 1;             //search upper half
    }
  }
  return position;
}
```

What if first + last is odd?
What if first==last?

Binary Search

Example Exam Question!

The target of your search is 42. Given the following list of integers, record the values of first, last, and middle during a binary search. Assume the following numbers are in an array.

1 7 8 14 20 42 55 67 78 101 112 122 170 179 190

Repeat the exercise with a target of 82

first	0	0	4
last	14	6	6
middle	7	3	5

first	0	8	8	8	9
last	14	14	10	8	8
middle	7	11	9	8	

Note: these are the **indexes**, not the values in the array

Program using Binary Search

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

int binarySearch(int[], int, int);

int main() {
  const int SIZE=5;
  int idNums[SIZE] = {100, 750, 822, 871, 988};
  int results, id;

  cout << “Enter the employee ID to search for: “;
  cin >> id;

  results = binarySearch(idNums, SIZE, id);

  if (results == -1) {
    cout << “That id number is not registered\n”;
  } else {
    cout << “That id number is found at location “;
    cout << results+1 << endl;
  }

  return 0;
}
```

How is this program different from the one on slide 5?

Efficiency of Binary Search

Calculate worst case for N=1024

Items left to search	Comparisons so far
1024	0
512	1
256	2
128	3
64	4
32	5
16	6
8	7
4	8
2	9
1	10

Goal: calculate this value from N

$1024 = 2^{10}$ <==> $\log_2 1024 = 10$

Efficiency of Binary Search

If N is the number of elements in the array, how many comparisons (steps)?

$1024 = 2^{10}$ <==> $\log_2 1024 = 10$

$N = 2^{steps}$ <==> $\log_2 N = steps$

To what power do I raise 2 to get N?

	N=50,000	In terms of N
Best Case:	1	1
Worst Case:	16	$\log_2 N$

Rounded up to next whole number

Is Log_2N better than N?

Is <u>binary search</u> better than <u>linear search</u>?

Is this really a fair comparison?

Compare values of N/2, N, and Log_2 N as N increases:

N	N/2	Log_2N
5	2.5	2.3
50	25	5.6
500	250	9.0
5,000	2,500	12.3
50,000	25,000	15.6

N and N/2 are growing much faster than log N!

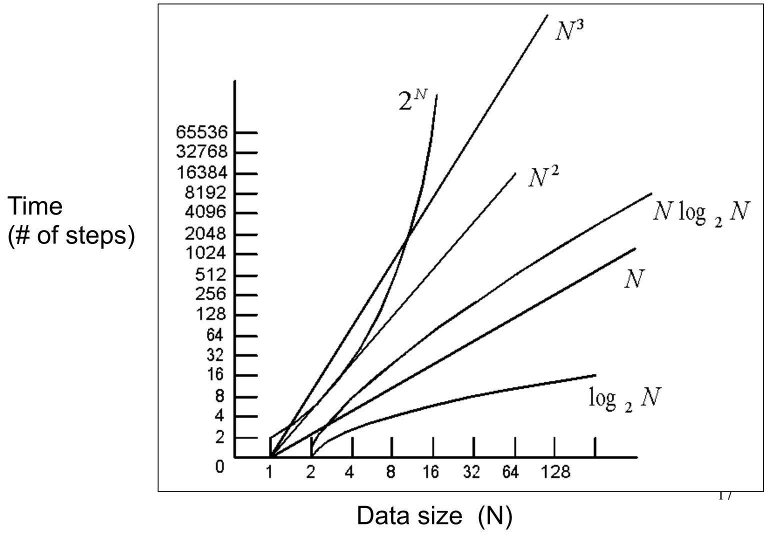
slower growing is more efficient (fewer steps).

Classifications of (math) functions

Constant	f(x)=b	O(1)
Logarithmic	$f(x)=\log_b(x)$	O(log n)
Linear	f(x)=ax+b	O(n)
Linearithmic	$f(x)=x \log_b(x)$	O(n log n)
Quadratic	$f(x)=ax^2+bx+c$	$O(n^2)$
Exponential	$f(x)=b^x$	$O(2^n)$

- Last column is "big Oh notation", used in CS.
- It ignores all but dominant term, constant factors

Comparing growth of functions



Efficiency of Algorithms

- To classify efficiency of an algorithm:
  - Express “time” (using number of main steps or comparisons), as a function of input size
  - Determine which classification the function fits into.
- Nearer to the top of the chart is slower growth, and more efficient (constant is better than logarithmic, etc.)

8.3 Sorting Algorithms

- Sort: rearrange the items in an array into ascending or descending order.
- Selection Sort
- Bubble Sort

55 112 78 14 20 179 42 67 190 7 101 1 122 170 8 unsorted

1 7 8 14 20 42 55 67 78 101 112 122 170 179 190 sorted

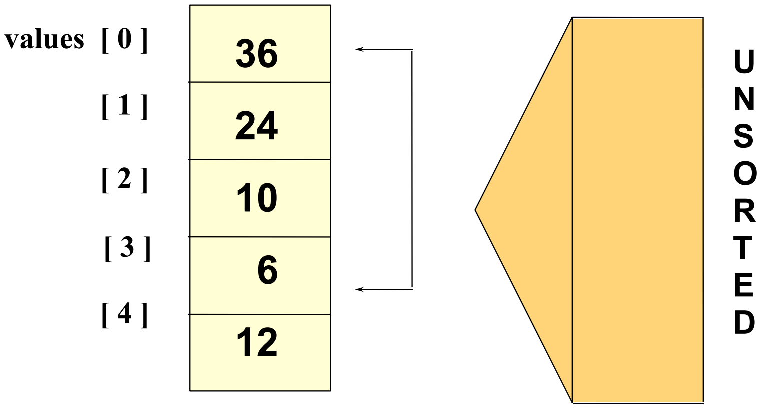
Why is sorting important?

- Searching in a sorted list is much easier than searching in an unsorted list.
- Especially for people
  - dictionary entries
  - phone book
  - card catalog in library
  - bank statement: transactions in date order
- Most of the data displayed by computers is sorted.

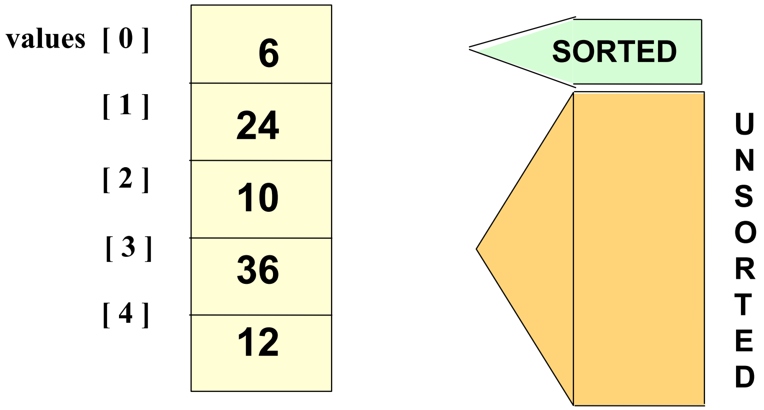
Selection Sort

- There is a pass for each position (0..size-1)
- On each pass, the smallest (minimum) element in the rest of the list is exchanged (swapped) with element at the current position.
- The first part of the list (the part that is already processed) is always sorted
- Each pass increases the size of the sorted portion.

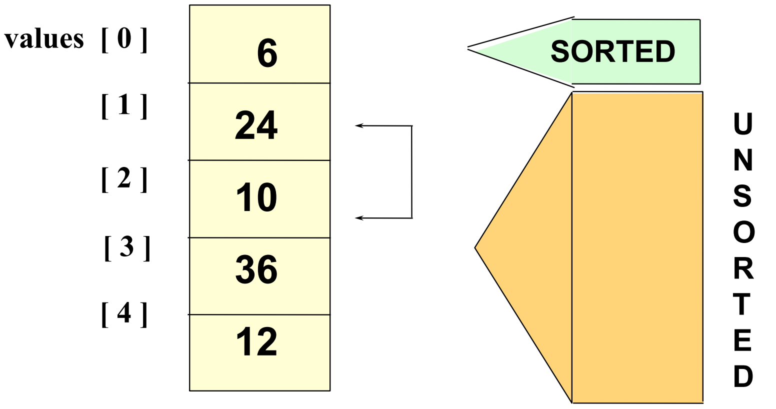
Selection Sort: Pass One



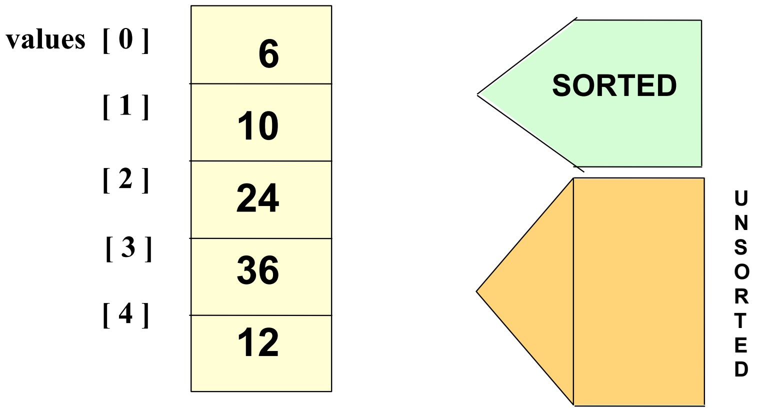
Selection Sort: End Pass One



Selection Sort: Pass Two

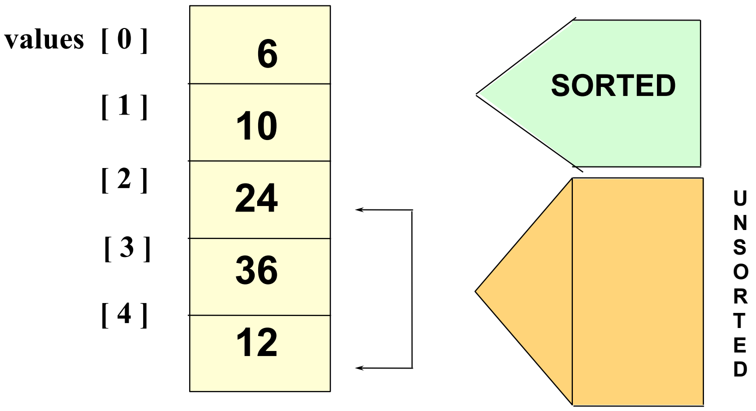


Selection Sort: End Pass Two



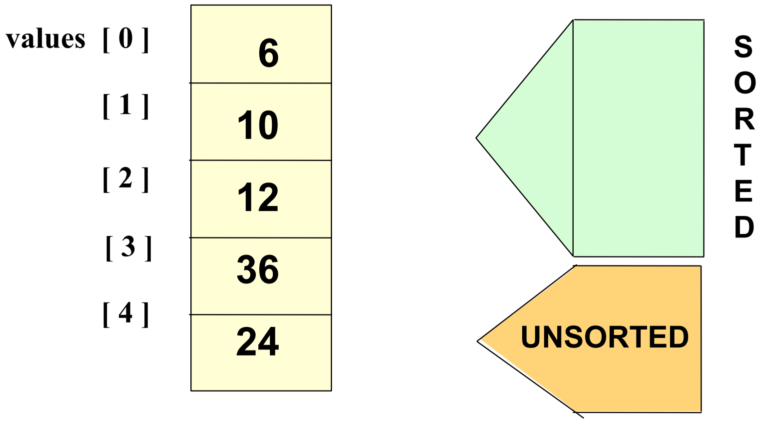
values [0]	6
[1]	10
[2]	24
[3]	36
[4]	12

Selection Sort: Pass Three



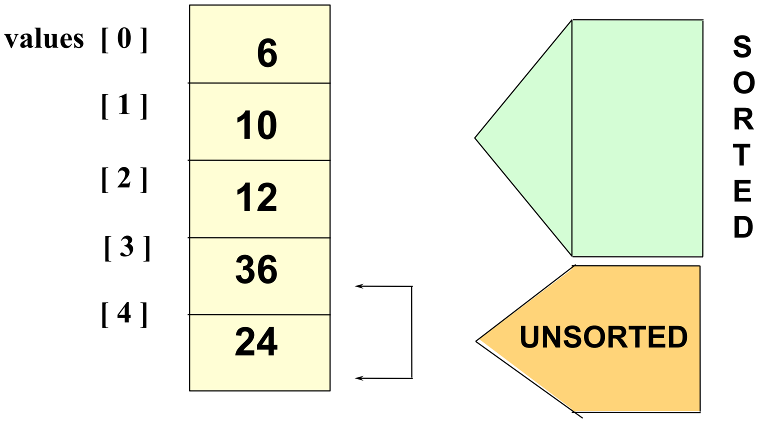
values [0]	6
[1]	10
[2]	24
[3]	36
[4]	12

Selection Sort: End Pass Three



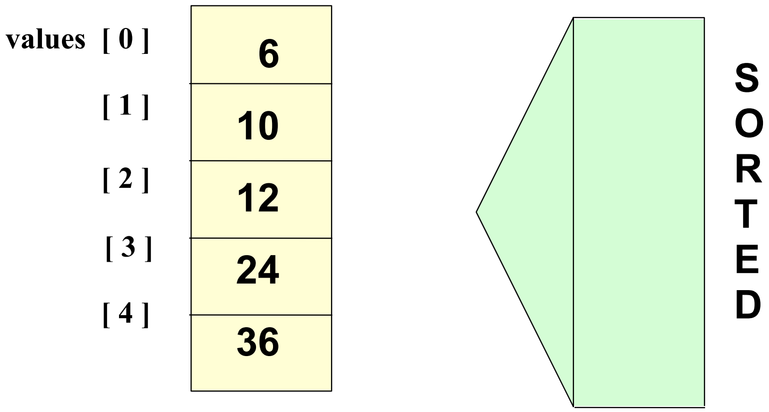
values [0]	6
[1]	10
[2]	12
[3]	36
[4]	24

Selection Sort: Pass Four



values [0]	6
[1]	10
[2]	12
[3]	36
[4]	24

Selection Sort: End Pass Four



Selection Sort in C++

```
// Returns the index of the smallest element, starting at start
int findIndexOfMin (int array[], int size, int start) {
   int minIndex = start;
   for (int i = start+1; i < size; i++) {
      if (array[i] < array[minIndex]) {
         minIndex = i;
      }
   }
   return minIndex;
}

// Sorts an array, using findIndexOfMin
void selectionSort (int array[], int size) {
   int temp;
   int minIndex;
   for (int index = 0; index < (size -1); index++) {
      minIndex = findIndexOfMin(array, size, index);
      //swap
      temp = array[minIndex];
      array[minIndex] = array[index];
      array[index] = temp;
   }
}
```

Note: saving the index

We need to find the **index** of the minimum value so that we can do the swap

---

Program using Selection Sort

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

int findIndexOfMin (int [], int, int);
void selectionSort(int [], int);
void showArray(int [], int);

int main() {
   int values[6] = {7, 2, 3, 8, 9, 1};

   cout << “The unsorted values are: \n”;
   showArray (values, 6);

   selectionSort (values, 6);

   cout << “The sorted values are: \n”;
   showArray(values, 6);
}

void showArray (int array[], int size) {
   for (int i=0; i<size; i++)
      cout << array[i] << “ “ ;
   cout << endl;
}
```

Output:

```
The unsorted values are:
7 2 3 8 9 1
The sorted values are:
1 2 3 7 8 9
```

Efficiency of Selection Sort

- N is the number of elements in the list
- Outer loop (in selectionSort) executes N-1 times
- Inner loop (in minIndex) executes N-1, then N-2, then N-3, ... then once.
- Total number of comparisons (in inner loop):

$(N-1) + (N-2) + \ldots + 2 + 1 =$ sum of 1 to N-1

Note: $N + (N-1) + (N-2) + \ldots + 2 + 1 = N(N+1)/2$

Subtract N from each side:

$$\begin{aligned}(N-1) + (N-2) + \ldots + 2 + 1 &= N(N+1)/2 - N \\ &= (N^2+N-2N)/2 \\ &= N^2/2 - N/2\end{aligned}$$

$\boxed{O(N^2)}$

The Bubble Sort

- On each pass:
  - Compare first two elements. If the first is bigger, they exchange places (swap).
  - Compare second and third elements. If second is bigger, exchange them.
  - Repeat until last two elements of the list are compared.
- Repeat this process until a pass completes with no exchanges

---

Bubble sort

Example

7 2 3 8 9 1	7 > 2, swap
2 7 3 8 9 1	7 > 3, swap
2 3 7 8 9 1	!(7 > 8), no swap
2 3 7 8 9 1	!(8 > 9), no swap
2 3 7 8 9 1	9 > 1, swap
2 3 7 8 1 9	finished pass 1, did 3 swaps

Note: largest element is in last position

Bubble sort

Example

- 2 3 7 8 1 9 — 2<3<7<8, no swap, !(8<1), swap
- 2 3 7 1 8 9 — (8<9) no swap
- finished pass 2, did one swap

2 largest elements in last 2 positions

- 2 3 7 1 8 9 — 2<3<7, no swap, !(7<1), swap
- 2 3 1 7 8 9 — 7<8<9, no swap
- finished pass 3, did one swap

3 largest elements in last 3 positions

Bubble sort

Example

- 2 3 1 7 8 9 — 2<3, !(3<1) swap, 3<7<8<9
- 2 1 3 7 8 9
- finished pass 4, did one swap
- 2 1 3 7 8 9 — !(2<1) swap, 2<3<7<8<9
- 1 2 3 7 8 9
- finished pass 5, did one swap
- 1 2 3 7 8 9 — 1<2<3<7<8<9, no swaps
- finished pass 6, no swaps, list is sorted!

Bubble sort

how does it work?

- At the end of the first pass, the largest element is moved to the end (it’s bigger than all its neighbors)
- At the end of the second pass, the second largest element is moved to just before the last element.
- The back end (tail) of the list remains sorted.
- Each pass increases the size of the sorted portion.
- No exchanges implies each element is smaller than its next neighbor (so the list is sorted).

Bubble Sort in C++

```
void bubbleSort (int array[], int size) {

   bool swap;
   int temp;

   do {

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

         if (array [i] > array[i+1]) {

            temp = array[i];
            array[i] = array[i+1];
            array[i+1] = temp;
            swap = true;
         }
      }
   } while (swap);
}
```

Program using bubble sort

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

void bubbleSort(int [], int);
void showArray(int [], int);

int main() {
   int values[6] = {7, 2, 3, 8, 9, 1};

   cout << “The unsorted values are: \n”;
   showArray (values, 6);

   bubbleSort (values, 6);

   cout << “The sorted values are: \n”;
   showArray(values, 6);
}

void showArray (int array[], int size) {
   for (int i=0; i<size; i++)
      cout << array[i] << “ “ ;
   cout << endl;
}
```

Output:

```
The unsorted values are:
7 2 3 8 9 1
The sorted values are:
1 2 3 7 8 9
```

Efficiency of Bubble Sort

- Each pass makes N-1 comparisons
- There will be at most N passes
- So worst case it’s: $(N-1)*N = N^2 - N$ $\boxed{O(N^2)}$
- If you change the algorithm to look at only the unsorted part of the array in each pass, it’s exactly like the selection sort:

$(N-1) + (N-2) + \ldots + 2 + 1 = N^2/2 - N/2$ $\boxed{\textbf{still } O(N^2)}$