Searching and sorting

- Computational resources: space, time
- Inserting an item into an array
- Search algorithms: linear, binary
- Sorting algorithms:
  insertion, selection, bubble
- Merge problem
- Notation to express time complexity

How many stars?

```
1. for (i = 0; i < n; ++i)
     cout << "*";                 _______
2. for (i = 0; i < n; ++i)
     cout << "*";
   for (j = 0; j < n; ++j)
     cout << "*";                 _______
3. for (i = 0; i < n; ++i)
     for (j = 0; j < n; ++j)
        cout << "*";              _______
4. for (i = 0; i < n; ++i)
     for (j = i; j < n; ++j)
        cout << "*";              ______
```

Searching a collection

- A *collection* is a data item containing a set of items all of the same type
- The search key is one member of an array element
- *Example:* Linear search. To find the name of an employee in array *A* whose ID is *x*:

```
for (int i=0; i < n; ++i)
   if (A[i].ID == x)
       return A[i].name;
```


Searching for the first '1'

```
const int ARR_SZ = 6;
int match = -1;
char A[ARR_SZ] =
{'0','0','1','0','1','1'};
for (int i = 0; i < ARR_SZ; ++i)
  if(A{i] == '1') {
    match = i;
    break;
  }
if match >= 0
  cout << "Found at A[" << match << "]";
else cout << "Not found" << endl;
```

Output:
`Found at A[2]`

- By what proportion does time increase as *ARR_SZ* increases?

Search algorithms

- *Linear*
  - inspects each element of array
  - slow
  - works on any array
  - simple to code
- *Binary*
  - Similar to phonebook lookup
  - fast
  - works only on ordered arrays
  - more complex to code

Finding string length by search

[*leng_str.c*]

```
#define MAX 80

void main(void)
{
  int i=0;
  char input[MAX];
  printf("Enter a string: ");
  scanf("%s",input);
  while (i < MAX && input[i] != '\0')
     ++i;
  printf("Length is %i\n",i);
}
```

the search

Sample I/O:

```
Enter a string: hello
Length is 5
```

Linear-search (A, *key*)

```
i ← 1
while A[ i ] ≠ key and i ≤ size of A
      i ← i + 1
if A[ i ] matches key
      report success at A[ i ]
otherwise
      report failure
```

- Finds element with value *key* in array A
- Takes up to one pass through A

David Keil 1/03 7

Binary phone-book search

Search key: "Simmons"

ABCDEFGHIJKLMNOPQRSTUVWXYZ

First try: "Simmons" is after M

Second try: "Simmons" is before T

Third try: "Simmons" is after P

Fourth try: "Simmons" is after R

1 2 3 4 5

- Each step eliminates half the unsearched data, cuts remaining work in half

David Keil 1/03 8

A driver to test a binary-search function

```
void main()
{
  int arr[] = { 2,3,4,6,7,9,10,12,14,17,18,20},
     key;
  cout << "Enter search key: ";
  cin >> key;
  if (binary_srch(arr,key,sizeof(arr)/sizeof(int)))
    cout << "Found" << endl;
  else
    cout << "Not found" << endl;
}
```

[*binsrch.cpp*]

Sample I/O:

```
Enter search key: 2 Found
Enter search key: 8 Not found
Enter search key: 9 Found
Enter search key: 20 Found
```


Binary search algorithm

Compares middle array element to search key; repeats as necessary for left or right half

Binary-search(*A, key*)
first ← 1
last ← *Size*(*A*)
While *first* ≤ *last*
    middle ← ⌊(*first* + *last*) ÷ 2⌋
    if *A*[*middle*] = *key*   return *true*
    otherwise
        if *A*[*middle*] > *key*   *last* ← *middle* – 1
        otherwise   *first* ← *middle* + 1
Return *false*

C++ code for iterative binary search

```
bool binary_search(int A[],int key,int size)
{
  int first = 0, last = size-1, middle;
  bool found = false;
  // Preconditions: <size> is size of A;A is sorted
  while (first <= last && ! found)
  {
    // Loop invariant:
    // key is in range A[first..last] or not in A
    middle = (first + last) / 2;
    if (A[middle] == key)     found = true;
    else
      if (key < A[middle])    last = middle - 1;
      else                    first = middle + 1;
  }
  return found;
  // Postcondition: <found> tells whether <key> in A
}
```

[*binsrch.cpp*]

Big-O notation

- We measure the time efficiency of an algorithm as a *function* of the size of the data set it works with
- We group together all constant functions, linear functions, quadratic functions, etc., as $O(1)$, $O(n)$, $O(n^2)$
- To say that an algorithm is $O(n)$ means that the algorithm's running time grows roughly in proportion to the size of the data set

Search algorithm complexities

n	Linear search $O(n)$	Binary search $O(\log_2 n)$
10	10	3
100	100	6
1000	1000	10
1M	1M	~20
1G	1G	~30

David Keil 1/03 13

Verifying an array is sorted

- *Fact:* The best search algorithm requires array be in ascending order
- *Problem:* Determine whether the elements of array *A* are in ascending order

2	3	5	4	7	8	11	15

- How many steps for this array?
- How many for *any* 8-element array?
- How many for *any* n-element array?

David Keil 1/03 14

Inserting into a sorted array

```
void insert(float A[],int& size,float
new_item)
// Preconditions:  A not full; A is ascending
// Postconditions:  <A> contains <new_item>,
//                 <A> is still ascending.
{
   int i = size;
   while (new_item < A[i-1] && i > 0)
   {
      A[i] = A[i-1];
      i = i - 1;
   }
   A[i] = new_item;
   ++size;
}
```

*Move elements greater than **new_item** to the right*

*Drop **new_item** in place*

- Complexity? (Hint: takes up to 1 pass through array)

Insertion sort

num_sorted ← 1
Repeat
insert *A*[*num_sorted*+1] at proper location in *A*[1…*num_sorted*]
num_sorted ← *num_sorted* + 1
until *num_sorted* is size of *A*

Complexity: $O(n^2)$, because loop is nested 1 level

Bubble sort intuition

```
Repeat
  for each element
    if it is greater than its successor
      swap them
until none are found out of order
```

- The exit condition helps prove that bubble sort works
- The use of nested loops suggests it may take (n^2) comparisons to sort an n-element array

Bubble-sort

```
> Precondition: A is an array
Repeat
  swapped ← false
  for i ← 1 to Size(A) – 1
    if A[ i ] > A[ i +1]
      swap A[ i ] with A[ i +1]
      swapped ← true
until swapped = false
> Postcondition: A is ascending
```

Most sorting requires swapping array elements

- Clearest coding puts swap operation into a function:
```
int input1 = 2,input2 = 6;
swap(input1,input2);
```
- Function header:
```
swap(int& a, int& b);
```
- Naïve algorithm:
```
a = b;
b = a;
```
- Why naïve? Fix it

David Keil 1/03 19

Code for *swap*

```
void swap(double& a,double& b)
// Exchanges values of a, b
{
  double temp = a;
  a = b;
  b = temp;
}
```

```
void swap(double* pa,double* pb)
// Exchanges values pointed to by a, b
{
  double temp = *pa;
  *pa = *pb;
  *pb = temp;
}
```

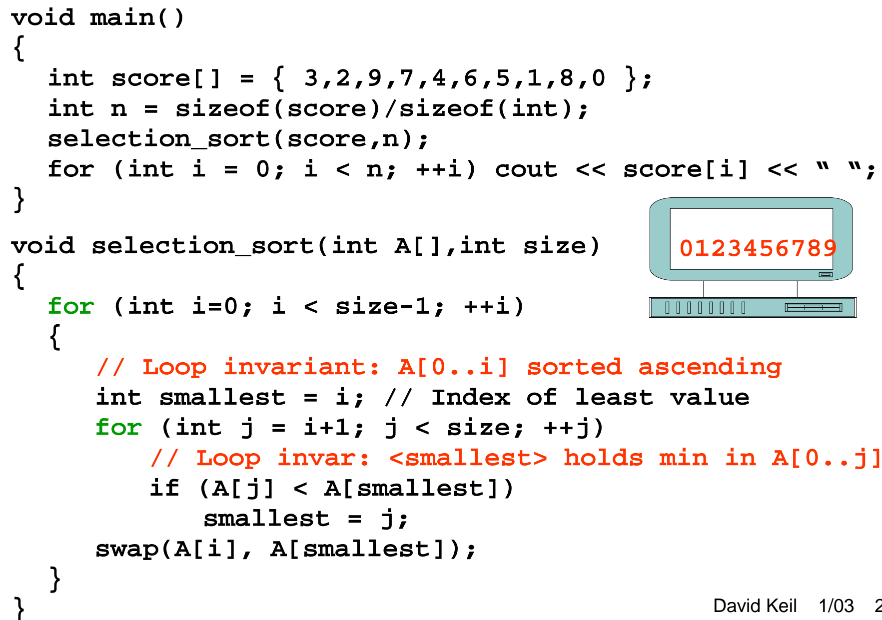
David Keil 1/03 20

Selection sort strategy

- This sort starts with an unsorted array and builds a sorted subarray from its leftmost element
- Algorithm repeatedly finds the smallest element of the *unsorted* subarray, moves it to the right end of the *sorted* segment
- Running time is $O(n^2)$ because each smallest element takes up to n steps to find

Find-min(A, first, last)

> Precondition: *A* is an array; *first* and *last* are subscripts into *A*
If *first* ≤ *last*
  smallest ← *A*[*first*]
  for *i* ← *first* to *last*
    > Invariant: *smallest* is min{*A*[*first..last*]}
    If *A*[*i*] < *smallest*
      smallest ← *A*[*i*]
  return *smallest*
else
  error
> Postcondition: *Find-min* has returned smallest element of *A*[*first..last*]



Worst, best, average cases

- Best case for search of n-element unsorted array: match on first try
- Worst case: match on nth try
- Average case: match halfway through $((n + 1) / 2)$
- Average-case analysis uses probability concepts
- Average-case analysis tends to approximate worst-case

David Keil 1/03 24

Constant time is O(1) time

- Any number of steps that *doesn't* depend on array size (n) can be considered *one* step
- Constant time is O(1)
- *Examples:*
  - *swap* time is O(1), though it takes 3 steps
  - Appending an element to an array is O(1)

David Keil 1/03 25

Optimal algorithms for sorts that are based on comparisons

- The best possible performance of a sorting algorithm that works by comparing array elements is $O(n\log n)$
- *Examples:* Shell sort (sub-optimal), Quicksort, Heap sort, Tree sort
- $O(n\log n)$ is as much faster than $O(n^2)$ (Bubble, Selection, Insertion) as the binary search is faster than linear search, i.e., 50K times better for a 1meg array

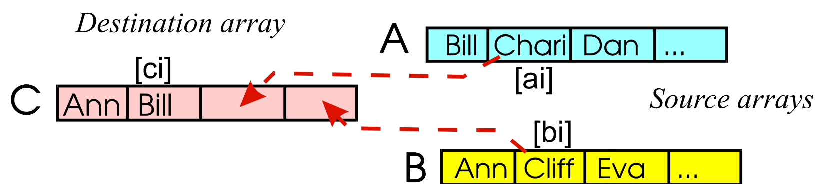
David Keil 1/03 26

Loops and big-O notation

- A simple loop of iterations proportional to n is $O(n)$
- A loop nested to two levels, each with roughly n iterations, is $O(n^2)$
- If we start with n items to look at and cut our remaining work in half at each step, then the job will take $O(\log_2 n)$ such steps.

David Keil 1/03 27

Merging two sorted arrays



Repeat until A and B are exhausted:
Append the lesser of $\{A_{ai}, B_{bi}\}$ to C, incrementing the indexes *ai, bi*, and *ci* as appropriate

- C should be as large as A, B, together
- Complexity: O(size(A) + size(B))

David Keil 1/03 28

Array-merge code

```
void main()
{
  const double A1[] = {1,3,4,6,9,12,13},
    A2[] = {2,5,6,8,10,11,13,15};
  const int sz1 = sizeof(A1) / sizeof(double),
    sz2 = sizeof(A2) / sizeof(double);
  const int sz3 = sz1 + sz2;
  double A3[sz3]; // Merge result (A3 gets merge of A1, A2)
  int i1=0,i2=0,i3=0;
  // Precondition: <A1>, <A2> sorted ascending
  while (i1  < sz1 && i2 < sz2)
    // Loop invariant: <A3>[0..i3] is sorted; i3 = i1 + i2
    A3[i3++] = ((A1[i1] < A2[i2]) ? A1[i1++] : A2[i2++]);
  while (i1 < sz1) A3[i3++] = A1[i1++];
  while (i2 < sz2) A3[i3++] = A2[i2++];
  // Postconditions: A3 contains all elements of A1, A2;
  // A3 is sorted ascending.
  for (int i=0; i < sz3; ++i)
    cout << A3[i] << " ";
}
```

[*merge.cpp*]

Output: `1 2 3 4 5 6 6 8 9 10 11 12 13 13 15`

David Keil 1/03 29

Discussion questions

1. Rewrite the linear-search algorithm to start at the *end* of an array, moving leftward.
2. Could an array be sorted in fewer than *n* comparisons? Why or why not?
3. How can we modify Bubble Sort to sort *descending*?
4. Rewrite Insertion Sort so that it repeatedly finds the highest value in a sequence at the left of an array, and moves the value into a sequence at the right.

David Keil 1/03 30

Challenge problem

- Write an algorithm that finds the *median* (middle-valued) element of an unsorted array of (odd) size n
- Hint: $(n - 1) / 2$ elements are at least as large as the median and $(n - 1) / 2$ elements are no larger
- What is complexity of your algorithm in Big-O notation?