SEARCHING AND SORTING

ISE 208
(Intermediate Programming)

SUNY at Stony Brook

SEARCHING AND SORTING

- Searching and sorting are fundamental operations in computer programming
- Searching and sorting go hand in hand
  - Sorted data is much easier to search

LINEAR SEARCH

- Examine each element in turn to see if it's the one you're looking for
- On average, you have to examine half of the data set to find what you're looking for
- Works even if the list is unsorted
- Takes time proportional to O(n) (linear)
  - The time needed increases at the same rate as the size of the data set increases

LINEAR SEARCH CODE

```
public boolean search (int [] list, int value)
{
    for (int i = 0; i < list.length; i++)
    {
        if (list[i] == value)
            return true; // we found our value
    }

    return false;
}
```

BINARY SEARCH

- Method: choose an element at random (usually the middle) and decide whether to search the left or right half
- At each decision point, the space to be searched is cut in half
- Requires sorted data to work properly
- Very efficient: only needs $\log_2(n)$ comparisons
  - Twice as much data only needs one more step

PSEUDOCODE

If (range contains only one element):

Look for desired value

Else:

1. Get midpoint of range
2. Determine which half of the range is likely to contain the desired value
3. Repeat the binary search on just that half of the range

BINARY SEARCH EXAMPLE

- Ex. Find 29
- Start: [10, 13, 14, 29, 37]
- Examine 14: [10, 13] [14] [29, 37]
- Find 29: [10, 13, 14] [29] [37]

BINARY SEARCH CODE

```
// Iterative binary search algorithm
// Search list from indices first-last

public int binSearch (int [] list, int value)
{
    int first = 0;
    last = list.length - 1;
    int position = -1;
    boolean found = false;
```

```
while (!found && first <= last)
{
    int middle = (first + last) / 2;

    if (list[middle] == value)
    {
        found = true;
        position = middle;
    }
```

```
    else if (list[middle] > value) // search left
    {
        last = middle - 1;
    }
    else   // search right half
    {
        first = middle + 1;
    }

    // Return item position or -1 if not found
    return position;
}
```

SORTING TECHNIQUES

- Many sorting techniques exist: bubble sort, insertion sort, selection sort, mergesort, quicksort, shell sort, radix sort, etc.
- These techniques differ in their efficiency
  - Different sorting techniques take different amounts of time (and memory/disk space) to sort the same data
  - Some sorting algorithms are better (faster) than others for larger data sets

BUBBLE SORT

- Method: compare pairs of adjacent items, and swap them if they are "out of order"
  - Elements "bubble" to their proper places
  - At the end of each pass, the largest remaining element is in its proper place
- This is trivial to implement, but very inefficient
  - Each pass may only sort a single value
  - For N values, we need (N-1) passes

PSEUDOCODE

- Let N be the number of elements in the data set
- Repeat N-1 times:
  - Repeat N-1 times:
    - if A[x] > A[x+1], swap them

BUBBLE SORT EXAMPLE

Start: [29, 10, 14, 37, 13]

After 1st pass: [10, 14, 29, 13, 37]

After 2nd pass: [10, 14, 13, 29, 37]

After 3rd pass: [10, 13, 14, 29, 37]

End: [10, 13, 14, 29, 37]

BUBBLE SORT CODE

```
public void bubbleSort (int [] list)
{
    for (int i = 0; i < list.length-1; i++)
    {
        for (int k = 0; k < list.length-1; k++)
        {
            if (list[k] > list[k+1])
                // Swap list[k] and list[k+1]
        }
    }
}
```

INSERTION SORT

- Method: select one element at a time and insert it into its proper sorted position
- Begin by dividing the array into two regions: sorted and unsorted
  - For each pass, move the first unsorted item into its proper position in the sorted region
- Slightly more efficient than bubble sort, since it swaps fewer elements per round

PSEUDOCODE

1. A[0] is sorted; A[1]-A[N-1] are unsorted
2. Repeat N times:
    1. nextItem = first unsorted element
    2. Shift sorted elements > nextItem over one position (A[x] = A[x-1])
    3. Insert nextItem into correct position

INSERTION SORT EXAMPLE

- Start: [29][10, 14, 37, 13]
- Move 10: [10, 29][14, 37, 13]
- Move 14: [10, 14, 29][37, 13]
- Move 37: [10, 14, 29, 37][13]
- Move 13: [10, 13, 14, 29, 37][ ]
- End: [10, 13, 14, 29, 37]

INSERTION SORT CODE

```
public void insertionSort(int [] list)
{
    for (int unsorted = 1; unsorted < list.length; unsorted++)
    {
        int nextItem = list[unsorted];
        int loc;

        // Shift larger sorted elements to the right
        for (loc = unsorted; (loc > 0) && (list[loc-1] > nextItem); loc--)
        {
            list[loc] = list[loc-1];
        }

        // Insert nextItem into sorted position
        list[loc] = nextItem;
    }
}
```

SELECTION SORT

- Repeatedly search for the largest unsorted item, and put it into its sorted position
- Again, we divide the data set into unsorted and sorted regions (the sorted region goes at the end)
- On each pass, swap the largest unsorted item with the last unsorted element
  - This is more efficient than bubble and insertion sort; it only needs one exchange per round

PSEUDOCODE

1. Repeat N-1 times:
    1. Find the largest (unsorted) element
    2. Swap A[last] with A[largest]
    3. Mark the unsorted region as being one element smaller

SELECTION SORT EXAMPLE

Start: [29, 10, 14, 37, 13] [ ]

After 1st pass: [29, 10, 14, 13] [37]

After 2nd pass: [13, 10, 14] [29, 37]

After 3rd pass: [13, 10] [14, 29, 37]

After 4th pass: [10] [13, 14, 29, 37]

End: [ ] [10, 13, 14, 29, 37]

Blue values are unsorted; black values are sorted

SELECTION SORT CODE

```
private int indexOfLargest(int [] A, int size)
{
    int currIndex, largestSoFar = 0;

    for (currIndex = 1; currIndex < size; currIndex++)
    {
        if (A[currIndex] > A[largestSoFar])
            largestSoFar = currIndex;
    }

    return largestSoFar;
}
```

SELECTION SORT CODE (2)

```
public void selSort (int [] A)
{
    for (last = A.length-1; last >= 1; last--)
    {
        int L = indexOfLargest(A, last+1);
        // Swap A[L] and A[last]
    }
}
```

RADIX SORT

- Method: Form groups (based on digits in the same place), then combine those groups
  - i.e., all items with 3 in the tens place
- This requires *d* iterations, where *d* is the number of digits in the largest element
- Worst-case running time: O(*d*n)

PSEUDOCODE

for (J = d down to 1):

1. Initialize 10 groups to empty
2. for (I = 0 through N-1):
    1. Place A[I] at the end of group K
    2. Increment Kth counter
3. Replace A with group 0 + group 1 + etc.

AN ILLUSTRATION

Start: 0123, 2154, 0222, 0004, 0283, 1560, 1061, 2150

Pass 1: 1560, 2150, 1061, 0222, 0123, 0283, 2154, 0004

Pass 2: 0004, 0222, 0123, 2150, 2154, 1560, 1061, 0283

Pass 3: 0004, 1061, 0123, 2150, 2154, 0222, 0283, 1560

Pass 4: 0004, 0123, 0222, 0283, 1061, 1560, 2150, 2154

EXTRACTING DIGITS

- How do we extract the dth digit of an integer?
- Use a combination of / and %
  - Ones digit: n % 10
  - Tens digit: (n / 10) % 10
  - Hundreds digit: (n / 100) % 10

EXTRACTION CODE

```
// Extracts the dth digit from val
// ones digit = 1, tens digit = 2, etc.
private int extract (int val, int d)
{
    int div = 1;

    for (int i = 1; i < d; i++)
        div *= 10;

    return (val / div) % 10;
}
```