CSE 143 SUMMER 1998

Searching and Sorting
[Sections 12.4, 12.7-12.8]

Searching and Sorting

- Two very common problems in Computer Science
- Searching
  - Given a collection and an element, find the element in the collection
  - Useful for both Table and Set ADTs
  - Elements must be comparable using ==
- Sorting
  - Given a collection, rearrange elements into some order
  - Elements must be comparable using <= (or <)

Searching Arrays

- We can search any sequential collection using the supplied navigation methods
- But we'll focus on searching arrays
  - Direct access makes searching potentially faster
  - Indices allow us to return location of element in array

```
int search( int data[], int size, int item ) {
    // Return the index of item in the array
    // or -1 if the item isn't in the array
}
```

- Size of problem is length of the array

Linear Search

- Look at every array element in order

```
// Linear search
int search( int data[], int size, int item ) {
    for( int idx = 0; idx < size; idx++ ) {
        if( data[ idx ] == item ) {
            return idx;
        }
    }

    return -1;
}
```

- What's the complexity of linear searching?

Searching a Sorted Array

- If the array is already in sorted order, we can do a lot better
- Idea: for any chunk of the array, the item we're searching for is either <, == or > the element at the middle of the chunk
  - If ==, we're done
  - If <, we don't have to look at anything right of the middle
  - If >, we don't have to look at anything left of the middle

Binary Search

- Jump to the middle and discard the half we don't care about

```
// Binary search
int search( int data[], int size, int item ) {
    return findInRange( data, item, 0, size - 1 );
}

int findInRange( int data[], int item, int lo, int hi ) {
    if( lo > hi ) return -1;
    int mid = (lo+hi) / 2;
    if( item == data[ mid ] ) {
        return mid;
    } else if( item < data[ mid ] )
        return findInRange( data, item, lo, mid - 1 );
    } else {
        return findInRange( data, item, mid + 1, hi );
    }
}
```

Performance Analysis

- How much work is done at each recursive call?
- How many recursive calls are there?
- On a sorted array, would you rather use linear search or binary search?

Other Searching Methods

- Interpolation Search
  - If the array contains numeric data, can "guess" how far into array to jump instead of middle
  - Works well on uniformly distributed data
  - Worst case is $O(n)$, average case is better than $O(\log_2 n)$
- Hashing (maybe later)
  - Relies on constructing a "hash function"
  - Average case is $O(1)$!!

Sorting Arrays

- Given an array, move elements around until the array is sorted according to some <= relation
  - If `idx <= jdx`, then `data[ idx ] <= data[ jdx ]`
- Size of problem is length of array
- Two facets to amount of work done
  - Number of element comparisons
  - Number of data movements

Selection Sort

```
void selectionSort( int* data, int size )
{
    if( size > 1 ) {
        int mi = 0;
        for( int idx = 1; idx < size; ++idx ) {
            if( data[ idx ] < data[ mi ] ) {
                mi = idx;
            }
        }
        swap( data[ 0 ], data[ mi ] );
        selectionSort( data + 1, size - 1 );
    }
}
```

- How many recursive calls are there?
- How much work done at each recursive call?

Insertion Sort

- Idea: build a new list that's always sorted and return that
- Better for linked lists
- Can be faster than selection sort
- Can use as basis for new ADT: SortedList

```
IntList insertionSort( IntList list )
{
    IntList result;
    list.start();
    while( !list.isEmpty() ) {
        int item = list.getData();
        list.deleteItem();
        insert( result, item );
    }
    return result;
}

void insert( IntList& list, int item )
{
    for( list.start(); !list.atEnd();
              list.advance() ) {
        if( item < list.getData() ) {
            break;
        }
    }
    list.insertBefore( item );
}
```


Fast Sorting Algorithms

- Even $O(n^2)$ can get expensive when n is really big
- There are sorting algorithms that run closer to $O(n \log n)$
- These algorithms tend to be elegantly recursive: break the problem down, sort subproblems, reassemble (**divide and conquer**)
  - Quicksort
  - Mergesort

Mergesort

- Idea:
  - Break the list down into two roughly equal-sized sublists
  - Recursively sort each sublist
  - "Merge" the sublists together to get solution

How To Merge

- Start with two **sorted** lists of integers, which act like two queues
- Until both lists are empty, remove the front element which is smaller and add to result list
- Result will be sorted!

```
IntList merge( IntList& l1, IntList& l2 ) {
  IntList result;
  l1.start();
  l2.start();
  while( true ) {
    if( l1.isEmpty() && l2.isEmpty() ) {
      return result;
    } else if( l1.isEmpty() ) {
      result.insertAfter( l2.getData() );
      l2.deleteItem();
    } else if( l2.isEmpty() ) {
      result.insertAfter( l1.getData() );
      l1.deleteItem();
    } else {
      if( l1.getData() < l2.getData() ) {
        result.insertAfter( l1.getData() );
        l1.deleteItem();
      } else {
        result.insertAfter( l2.getData() );
        l2.deleteItem();
      }
    }
  }
}
```


Writing Mergesort

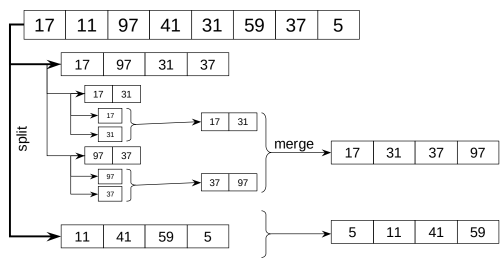
```
void split( IntList& from, IntList& t1, IntList& t2 ) {
    from.start();
    while( true ) {
        if( from.isEmpty() ) break;
        t1.insertAfter( from.getData() );
        from.deleteItem();

        if( from.isEmpty() ) break;
        t2.insertAfter( from.getData() );
        from.deleteItem();
    }
}

IntList mergesort( IntList& list ) {
    if( list.getSize() <= 1 ) {
        return list;
    } else {
        IntList sub1, sub2;
        split( list, sub1, sub2 );
        sub1 = mergesort( sub1 );
        sub2 = mergesort( sub2 );
        return merge( sub1, sub2 );
    }
}
```

- Break the list down into two roughly equal-sized sublists
- Recursively sort each sublist
- "Merge" the sublists together to get solution

Example of Mergesort



Analysis of Mergesort

- Initial questions:
  - What's the complexity of `merge()`?
  - What's the complexity of `split()`?
- Complete analysis is rather difficult
  - Each "level" of recursion involves 2^k calls to mergesort, each of size $n/2^k$
  - So we're doing a total of $O(n)$ work at each level
  - There are $O(\log n)$ levels, so total complexity is $O(n \log n)$
- But mergesort is not used very often!

Quicksort

- The most widely-accepted fast sorting algorithm
  - "Easy" to implement
  - Good performance
  - Operates "in place": no need to create temporary data structures
- Idea: given an array of integers…
  - Choose an element of the array to act as the **pivot**
  - Move everything <=pivot to the left, everything >pivot to the right (**partition** the array)
  - Recursively sort left and right halves

Writing Quicksort

```
void quicksort( int data[], int size )
{
    quicksortRange( data, 0, size - 1 );
}

void quicksortRange( int data[], int lo, int hi )
{
    if( lo >= hi ) {
        return;
    }

    int midindex = partition( data, lo, hi );
    quicksortRange( data, lo, midindex - 1 );
    quicksortRange( data, midindex + 1, hi );
}
```

- Partitioning is the hard part…

How to Partition

- (Arbitrarily) choose the first element of the range as the pivot
- Swap array elements around until everything <= pivot is left of pivot, everything >pivot is right of pivot
- Return the final resting place of the pivot element

37	11	97	41	31	59	43	5

⇩

11	31	5	37	97	41	59	43

Implementing Partitioning

```
void partition( int data[], int lo, int hi )
{
    int midval = data[ lo ];
    int j = lo
    int k = hi;

    while( j < k ) {
        while( (j <= hi) && (data[j] <= midval) ) ++j;
        while( (k >= lo) && (data[k] > midval) ) --k;
        if( j < k ) {
            swap( data[j], data[k] );
        }
    }

    swap( data[ lo ], data[ k ] );
    return k;
}
```


Complexity of Partition

- Don't be fooled by the nested while loops
  - `j` starts at `lo` and only increases
  - `k` starts at `hi` and only decreases
  - `j` is incremented at least once for each iteration of outer loop
  - Stops when `j` and `k` cross
- At most `hi-lo+1` iterations of inner loops, so $O(n)$ time

Complexity of Quicksort

- Similar to mergesort
  - Linear work for each recursive call
  - At every level of recursion, roughly 2^k calls on arrays of size $n/2^k$
  - So $O(n)$ work at each level of recursion
  - About $O(\log n)$ levels, so total complexity is $O(n \log n)$
- But there's a big assumption here!
  - This analysis depends on how evenly partition divides arrays

Best Case For Quicksort

- The best case is when partition breaks every array exactly in half
- Then our assumptions are valid: every level does $O(n)$, $O(\log n)$ levels, so $O(n \log n)$ total

Worst Case For Quicksort

- Worst case is when partition breaks array into subarrays of size 1 and n-1
- In this case, there are n levels of recursion, and we still do $O(n)$ work at each level, so $O(n^2)$ total!
- This can happen when the chosen pivot is the smallest or largest element of the array
- We want to avoid that!

Average Case For Quicksort

- It turns out that the average case is still pretty good
- For "most" arrays, quicksort will run in $O(n \log n)$ time
- Average case analysis based on probability that pivot is bad in a random array

Building A Better Quicksort

- Don't have to choose **`data[lo]`** as pivot every time
  - Can do some small (linear) amount of work to choose a better pivot
- Choosing a pivot at random yields $O(n \log n)$!!
- Other techniques involve computing the median of some elements in the array and work fairly well in practice
- Quicksort is usually the best choice for sorting

Summary

- Searching: find an element in a collection
  - Linear search: $O(n)$
  - Binary search: $O(\log n)$
- Sorting: rearrange a collection according to some ordering relation
  - Selection sort, insertion sort: $O(n^2)$
  - Mergesort: $O(n \log n)$ but inefficient
  - Quicksort: $O(n \log n)$ average, $O(n^2)$ worst, but a good choice in practice