CSC 143 Java

Searching & Sorting

Sequential (Linear) Search

- We've seen the linear search algorithm

```
int indexOf(Object o) {
    int foundIndex = -1;
    for (int k = 0; k < size && foundIndex == -1; k++) {
      if (elements[k].equals(o))
            foundIndex = k;
    }
    return foundIndex;
}
```

O() for list of size n is…

- Can we do better?

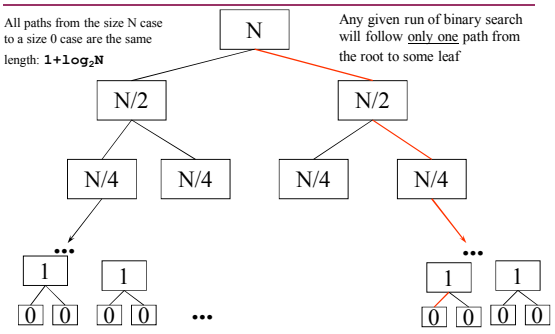
Binary Search

```
/** Return location of word in words, or –1 if not found */
public int find(String word) {
    return bSearch(word, 0, size-1);
}
// Return location of word in words[lo..hi] or –1 if not found
private int bSearch(String word, int lo, int hi) {
    // return –1 if interval lo..hi is empty
    if (lo > hi) { return –1; }
    // search words[lo..hi]
    int mid = (lo + hi) / 2;
    int comp = word.compareTo(words[mid]);
    if (comp == 0) { return mid; }
    else if (comp < 0)  { return bSearch(word, lo, mid-1) ; }
    else /* comp > 0 */ { return  bSearch(word, mid+1, hi) ; }
}
```


Picture the Execution

All paths from the size N case to a size 0 case are the same length: $1+\log_2 N$

Any given run of binary search will follow only one path from the root to some leaf



Performance of Binary Search

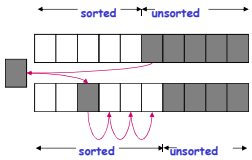
- Analysis
  - Time of each recursive call:
  - Number of recursive calls:
  - Total time:
- Compare to linear search
  - Time to search 10, 100, 1000, 1,000,000 words

    linear

    binary
  - What is incremental cost if size of list is doubled?

Insert into a Sorted List

```
// given existing stuff[0..sz-1], insert word in correct place and increase size
// assumes there is room for one more piece of data
private static void insertData(Object stuff[ ], Object o, int sz) {
    Comparable c = (Comparable)o;  // will throw a ClassCastException
     int pos;
     for( pos = sz; pos > 0 &&  c.compareTo(stuff[pos -1])<0; pos--)
            stuff[pos] = stuff[pos-1];
    stuff[pos] = o;
}
```



Insertion Sort

We can sort a list in place by repeating the insertion operation

```
public static void insertionSort( Object data[ ], int size) {
        for (int k = 1; k < size; k++) {
                insertData(data, data[k], k);
    }
}
```

➢Cost of each insertData operation:

➢Number of times insertData is executed:

➢Total cost:

Can we do better?

Analysis

- Why was binary search so much more effective than sequential search?
  - Answer: binary search divided the search space in half each time; sequential search only reduced the search space by 1 item
- Why is insertion sort $O(n^2)$?
  - Each insert operation only gets 1 more item in place at cost O(n)
  - O(n) insert operations
- Can we do something similar for sorting?

Divide and Conquer Algorithms

- Idea: like binary search,
  - divide the sorting problem into two subproblems;
  - recursively sort each subproblem;
  - combine results
- Want division and combination at the end to be fast
- Want to be able to sort two halves independently
- This is a particular example of an algorithm technique known as **divide and conquer**

Quicksort

- **Idea**
  - **Pick an element of the list: the *pivot***
  - **Place all elements of the list smaller than the pivot in the half of the list to its left; place larger elements to the right**
  - **Recursively sort each of the halves**
- **Invented by C. A. R. Hoare (1962)**

Code for Quicksort

```
/** Sort the data */
public static void quickSort(Object[ ] data, int size ) {
    qsort(data, 0, size-1);
}

// Sort words[lo..hi]
private void qsort(Object [ ] stuff, int lo, int hi) {
    // quit if empty partition
    if (lo >= hi) { return; }

    // partition array and return pivot loc
    int pivotLocation = partition(stuff, lo, hi);
    qsort(stuff, lo, pivotLocation-1);
    qsort(stuff, pivotLocation+1, hi);
}
```


Recursion Analysis

- Base case? Yes.

```
// quit if empty partition
if (lo >= hi) { return; }
```

- Recursive cases? Yes

```
qsort(stuff, lo, pivotLocation-1);
qsort(stuff, pivotLocation+1, hi);
```

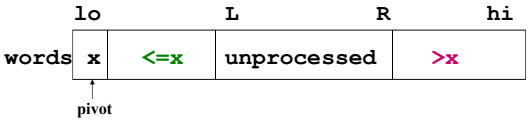
  - Observation: recursive cases are guaranteed to work on a smaller subproblem, so algorithm will terminate.

A Small Matter of Programming

- Partition function
  - Pick pivot
  - Rearrange array so all smaller elements are to the left, all larger to the right, with pivot in the middle
  - Partition is not recursive
- How do we pick the pivot?
  - For now, keep it simple – use the first item in the interval
  - Better strategies exist

A `Partition` Implementation

- Use first element of array section as the pivot
- Invariant:



Partition

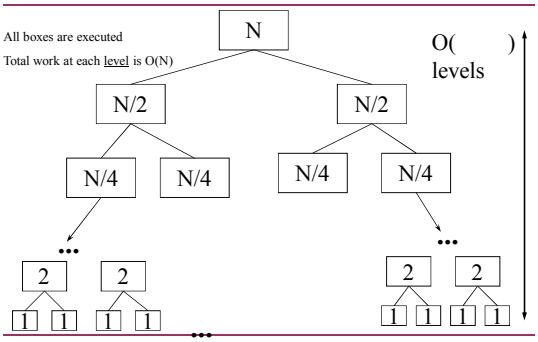
```
//Partition words[lo..hi]; return location of pivot
// Precondition: lo <= hi

private int partition(Object[ ] stuff, int lo, int hi){
  if(lo > hi) throw new IllegalArgumentException();
  int L = lo+1, R = hi;
  Comparable pivot = (Comparable) stuff[lo];
  while (L <= R) {
          if (pivot.compareTo(stuff[ L]) > 0)   L++;
          else if (pivot.compareTo(stuff[R]) < 0)   R--;
          else {
                  swap(stuff, L, R);
                  L++; R--;
          }
  }
  // put pivot element in middle & return location
  swap(stuff, lo,L-1);
  return L-1;
}
```


Quicksort Performance

- Each call to Quicksort (ignoring recursive calls):
  - Call to partition( ) is O(n) where n is size of the *part* of array being sorted

    Note: This n is smaller than the N of the original problem
  - Some O(1) work
  - Total = O(n) (n is the size of array part being sorted)
- Including recursive calls:
  - Two recursive calls at each level of recursion, each partitions "half" the array at a cost of O(n/2)
  - How many levels of recursion?

QuickSort Recursion

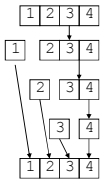


Quicksort Performance (Ideal Case)

- Each partition divides the list parts in half
  - Sublist sizes on recursive calls: n, n/2, n/4, n/8….
  - Total depth of recursion: _____
  - Total work at each level: O(N)
  - Total cost of quicksort: _____ !
- For a list of 10,000 items
  - Insertion sort: $O(n^2)$: 100,000,000
  - Quicksort: O(n log n): 10,000 $\log_2$ 10,000 = 132,877

Worst Case for Quicksort

- If we're very unlucky, then each pass through partition removes only a *single* element.



- In this case, we have N levels of recursion rather than $\log_2 N$. What's the total complexity?

Worst Case vs Average Case

- In practice, Quicksort works well, provided the pivot is picked with some care. Some strategies:
  - Compare a small number of list items (3-5) and pick the median for the pivot
  - Pick a pivot element randomly in the range lo..hi

Mergesort

- Split in half
  - just take the first half and the second half of the array, without rearranging
  - sort the halves separately
- Combining the sorted halves ("merge")
  - repeatedly pick the least element from each array
  - compare, and put the smaller in the resulting array
  - example: if the two arrays are

    1 12 15 20

    5 6 13 21 30

    The "merged" array is

    1 5 6 12 13 15 20 21 30
  - note: we will need a temporary result array

Mergesort Complexity

- Time complexity of merge() = O(____)
  - N is size of the part of the array being sorted
- Recursive calls:
  - Two recursive calls at each level of recursion, each does "half" the array at a cost of **`O(N/2)`**
  - How many levels of recursion?

Quicksort vs MergeSort

- Mergesort always has subproblems of size n/2
  - Which means guaranteed O(n log n)
- But mergesort requires an extra array for the result
- In practice, quicksort is the most commonly used general-purpose sort
  - Pretty easy to pick pivots well, so expected time is O(n log n)
  - Doesn't require extra space for a copy of the data

Summary

- ✓ Searching
  - Linear Search: `O(N)`
  - Binary Search: `O(log N)`, needs sorted data
- ✓ Sorting
  - Insertion Sort: $O(N^2)$

    Other quadratic sorts: Selection, Bubble
  - Quicksort: average: `O(N log N)`, worst-case: $O(N^2)$
  - Mergesort: `O(N log N)` guaranteed, but needs extra space