CSE 143 Java

Sorting

Reading: Sec. 19.3

Sorting

- Binary search is a huge speedup over sequential search
  - But requires the list be sorted
- Slight Problem: How do we get a sorted list?
  - Maintain the list in sorted order as each word is added
  - Sort the entire list when needed
- Many, many algorithms for sorting have been invented and analyzed
- Our algorithms mostly assume the data is already in an array
  - Other starting points and assumptions are possible

Insert for a Sorted List

- One possibility: ensure the list is always sorted as it is created
- Exercise: Assume that words[0..size-1] is sorted. Place new word in correct location so modified list remains sorted
  - Assume that there is spare capacity for the new word
- Before coding:
  - Draw pictures of an example situation, before and after
  - Write down the postconditions for the operation

```
// given existing list words[0..size-1], insert word in correct place and increase size
void insertWord(String word) {



    size++;
}
```

Picture

- Draw your picture here

Insertion Sort

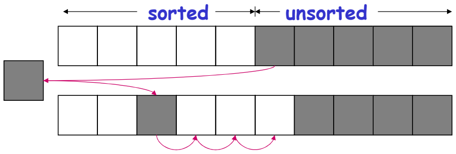
- Once we have insertWord working...
- We can sort a list in place by repeating the insertion operation

```
void insertionSort( ) {
    int finalSize = size;
    size = 1;
    for (int k = 1; k < finalSize; k++) {
        insertWord(words[k]);
    }
}
```

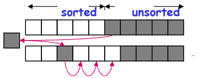
Insertion Sort As A Card Game Operation

- A bit like sorting a hand full of cards dealt one by one:
  - Pick up 1st card – it's sorted, the hand is sorted
  - Pick up 2nd card; *insert* it after or before 1st – both sorted
  - Pick up 3rd card; *insert* it after, between, or before 1st two
  - ...
- Each time:
  - Determine where new card goes
  - Make room for the newly inserted card and place it there

Insertion Sort As Invariant Progression



Insertion Sort



```
// instance variable
int[ ] list;                    // list[0..size-1] is the list to be sorted
int size;
// Sort list[0..size-1]
public void  sort {
   for (int j=1 ; j < size; j++) {
        // pre: 1 <= j && j < size && list[0 ... j-1] is in sorted order
        int temp = list[ j ];
        for (int i = j -1 ; i >= 0 && list[i] > temp ; i-- ) {
            list[i+1] = list[i] ;
         }
         list[i+1] = temp ;
         // post: 1 <= j && j < size && list[0 ... j] in sorted order
     }
}
```

Insertion Sort Trace

- Initial array contents

0	pear
1	orange
2	apple
3	rutabaga
4	aardvark
5	cherry
6	banana
7	kumquat

Insertion Sort Performance

- Cost of each insertWord operation:

- Number of times insertWord 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 per iteration
- 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?

Where are we on the chart?

N	$\log_2 N$	5N	$N \log_2 N$	N^2	2^N
8	3	40	24	64	256
16	4	80	64	256	65536
32	5	160	160	1024	$\sim 10^9$
64	6	320	384	4096	$\sim 10^{19}$
128	7	640	896	16384	$\sim 10^{38}$
256	8	1280	2048	65536	$\sim 10^{76}$
10000	13	50000	10^5	10^8	$\sim 10^{3010}$

Divide and Conquer Sorting

- Idea: emulate binary search in some ways
  1. divide the sorting problem into two subproblems;
  2. recursively sort each subproblem;
  3. combine results
- Want division and combination at the end to be fast
- Want to be able to sort two halves independently
- This algorithm strategy is called *divide and conquer*

Quicksort

- Invented by C. A. R. Hoare (1962)
- 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
- Before looking at any code, see if you can draw pictures based just on the first two steps of the description

Code for QuickSort

```
// Sort words[0..size-1]
void quickSort( ) {
    qsort(0, size-1);
}

// Sort words[lo..hi]
void qsort(int lo, int hi) {
    // quit if empty partition
    if (lo > hi) { return; }
    int pivotLocation = partition(lo, hi);      // partition array and return pivot loc
    qsort(lo, pivotLocation-1);
    qsort(pivotLocation+1, hi);
}
```

Recursion Analysis

- Base case? Yes.

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

- Recursive cases? Yes

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

  - Each recursive cases work on a smaller subproblem, so algorithm will terminate

A Small Matter of Programming

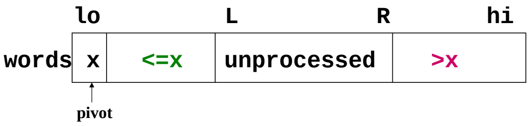
- *Partition* algorithm
  - Pick pivot
  - Rearrange array so all smaller element are to the left, all larger to the right, with pivot in the middle
- *Partition* is not recursive
- Fact of life: *partition* can be tricky to get right
  - Pictures and invariants are your friends here
- How do we pick the pivot?
  - For now, keep it simple – use the first item in the interval
  - Better strategies exist

Partition design

- We need to partition words[lo..hi]
- Pick words[lo] as the pivot
- Picture:

A `Partition` Implementation

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



Partition Algorithm: PseudoCode

The two-fingered method

```
// Partition words[lo..hi]; return location of pivot in range lo..hi
int partition(int lo, int hi) {









}
```

Partition Test

- Check: partition(0,7)

  0 orange
  1 pear
  2 apple
  3 rutabaga
  4 aardvark
  5 cherry
  6 banana
  7 kumquat

Complexity of QuickSort

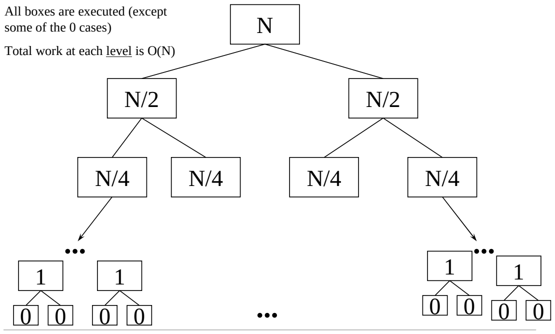
- Each call to Quicksort (ignoring recursive calls):
  - Each call of partition( ) is O(n) where n is size of the *part* of array being sorted

    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 (Ideally)

All boxes are executed (except some of the 0 cases)

Total work at each level is O(N)

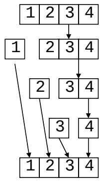


QuickSort Performance (Ideal Case)

- 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?

QuickSort Performance (Worst Case)

- Each partition manages to pick the largest or smallest item in the list as a pivot
  - Sublist sizes on recursive calls:
  - Total depth of recursion: ________________
  - Total work at each level: O(n)
  - Total cost of quicksort: ______________ !

Worst Case vs Average Case

- QuickSort has been shown to work well in the average case (mathematically speaking)
- In practice, Quicksort works well, provided the pivot is picked with some care
- Some strategies for choosing the pivot:
  - 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

QuickSort as an Instance of Divide and Conquer

Generic Divide and Conquer	QuickSort
1. Divide	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
2. Solve subproblems separately (and recursively)	Recursively sort each of the halves
3. Combine subsolutions to get overall solution	Surprise! Nothing to do

---

Another Divide-and-Conquer Sort: Mergesort

- 1. Split array in half
  - just take the first half and the second half of the array, *without* rearranging
- 2. Sort the halves separately
- 3. 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

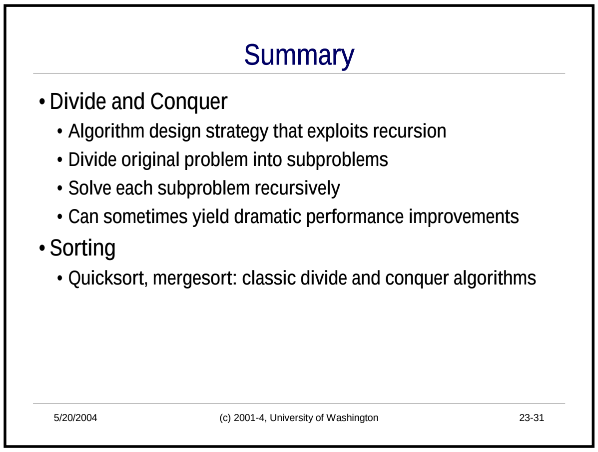
The "merged" array is

1 5 6 12 13 15 20 21 30

  - note: we will need a second array to hold the result

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
  - No problem if you’re sorting disk or tape files
  - Can be a problem if you’re trying to sort large lists in main memory
- 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

- Divide and Conquer
  - Algorithm design strategy that exploits recursion
  - Divide original problem into subproblems
  - Solve each subproblem recursively
  - Can sometimes yield dramatic performance improvements
- Sorting
  - Quicksort, mergesort: classic divide and conquer algorithms

5/20/2004 (c) 2001-4, University of Washington 23-31