CSE 143

Searching and Sorting

[Chapter 9, pp. 402-432]

Two important problems

- *Search*: finding something in a set of data
- *Sorting*: putting a set of data in order
- Both very common, very useful operations
- Both can be done more efficiently after some thought
- Both have been studied intensively by computer scientists

Review: Linear Search

- Given an array **A** of **N** **ints**, search for an element **x**.

```
// Return index of x if found, or -1 if not
int Find (int A[], int N, int x) {
   for ( int i = 0; i < N; i++ )
      if ( A[i] == x )
         return i;
 return -1;
}
```


How Efficient Is Linear Search?

```
// Return index of x if found, or -1 if not
int Find (int A[], int N, int x) {
  for ( int i = 0; i < N; i++ )
    if ( A[i] == x )
      return i;
 return -1;
}
```

- Problem size: N
- Best case (**x** is **A[0]**): **O(1)**
- Worst case (**x** not present): **O(N)**
- Average case (**x** in middle): **O(N/2) = O(N)**
  - Challenge for math majors: prove this!

Review: Binary Search

- If array is *sorted*, we can search faster
  - Start search in middle of array
  - If **x** is less than middle element, search (recursively) in lower half
  - If **x** is greater than middle element, search (recursively) in upper half
- Why is this faster than linear search?
  - At each step, linear search throws out one element
  - Binary search throws out *half* of remaining elements

Example

Find 26 in the following sorted array:

```
1 3 4 7 9 11 15 19 22 24 26 31 35 50 61
                 ↑
                22 24 26 31 35 50 61
                          ↑
                22 24 26
                    ↑
                      26
                      ↑
```

Binary Search (Recursive)

```
int find(int A[], int size, int x) {
  return findInRange(A, x, 0, size-1);
}

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

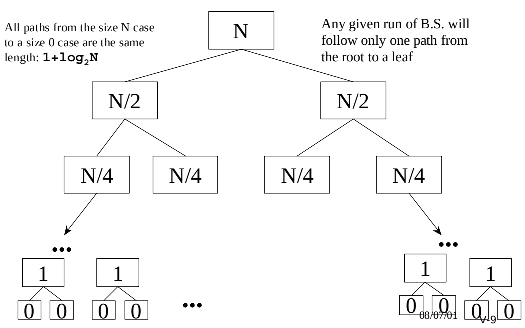
08/07/01 V-7

Analysis (recursive)

- Time per recursive call of binary search is `O(1)`
- How many recursive calls?
  - Each call discards at least half of the remaining input.
  - Recursion ends when input size is `0`
  - How many times can we divide `N` in half? $1+\log_2 N$
- With `O(1)` time per call and `O(log N)` calls, total is `O(1)*O(log N) = O(log N)`
- Doubling size of input only adds a *single* recursive call
  - Very fast for large arrays, especially compared to `O(N)` linear search

08/07/01 V-8

Binary Search Sizes



08/07/01 V-9

Sorting

- Binary search requires a sorted input array
  But how did the array get sorted?
- Many other applications need sorted input array
  - Language dictionaries
  - Telephone books
  - Printing data in organized fashion
    `Web search engine results, for example`
  - Spreadsheets
- Data sets may be very large

08/07/01 V-10

Sorting Algorithms

Many different sorting algorithms, with many different characteristics

- Some work better on small vs. large inputs
- Some preserve relative ordering of "equal" elements (*stable* sorts)
- Some need extra memory, some are in-place
- Some designed to exploit data locality (not jump around in memory/disk)

- Which ones are best?
  - Try to answer using efficiency analysis

08/07/01 V-11

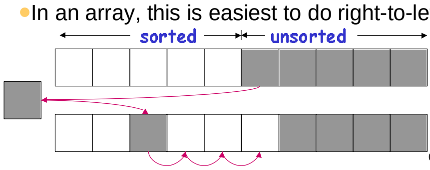
Sorts You May Know – Or Soon Will!

- 142 review
  - Bubble Sort
    `Some think it's a good "intro" sort`
    `Not very efficient`
  - Selection Sort
    `See appendix to this lecture unit`
  - Insertion Sort
    `A lot like Selection Sort`
- Mergesort
- Quicksort
- Radixsort (see appendix)

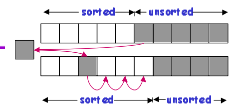
08/07/01 V-12

Insertion Sort

- A bit like sorting a hand full of cards:
  - Pick up 1 card – it's 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
  - ...
- Note: make room for the newly inserted member.
- In an array, this is easiest to do right-to-left



Insertion Sort Code




```
void  insert(int list[], int n) {
    int i;
    for (int j=1 ; j < n; ++j) {
        // pre: 1<=j && j<n && list[0 ... j-1] in sorted order
        int temp = list[j];
        for ( i = j-1 ; i >= 0 && list[i] > temp ; --i ) {
            list[i+1] = list[i] ;
        }
        list[i+1] = temp ;
        // post: 1<=j && j<n && list[0 ... j] in sorted order
    }
}
```


Insertion Sort Analysis

- Outer loop – n times
- Inner loop – at most n times
- Overall – $O(n^2)$ in worst case
- ("Average" is about $n^2/4$ comparisons.)

- In practice, insertion sort is the fastest of the simple quadratic methods
- 2x - 4x faster than bubble or selection sorts, and no harder to code
- Among fastest methods overall for n < 20 or so
- Among the fastest overall if the array is "almost sorted"

Comparing Sorts

- Insertion Sort: $O(N^2)$ in average case
  - For each of the N elements of the array, you inspect and move up to N-1 remaining elements to do the insertion
- Selection Sort: also $O(N^2)$
- Bubble Sort: also $O(N^2)$
  - For each of the N elements, you "bubble" through the remaining (up to N) elements
- All are referred to as "quadratic" sorts (Why?)

Is $O(N^2)$ the Best Possible?

- Asymptotic average case complexity is not always the whole story
- Examples:
  - Bubble Sort is usually slowest in practice because it does lots of swaps
  - Insertion Sort is almost O(N) if the array is "almost" sorted already
- If you know something about the data for a particular application, you may be able to tailor the algorithm
- At the end of the day, still $O(N^2)$

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**	**$\sim 10^{3010}$**

Can We Sort Faster Than O(N²)?

- Why was binary search so good?
  - Answer: at each stage, we divided the problem in two parts, each only half as big as the original
- With Selection Sort, at each stage the new problem was only 1 smaller than the original
  - Same was true of the other quadratic sort algorithms
- How could we treat sorting like we do searching?
  - I.e., somehow making the problem *much smaller* at each stage instead of just a *little smaller*

08/07/01 V-19

An Approach

- Try a "Divide and Conquer" approach
- Divide the array into two parts, in some sensible way
  - Hopefully doing this dividing up can be done efficiently
- Arrange it so we can
  - 1. sort the two halves separately
    `This would give us the "much smaller" property`
  - 2. recombine the two halves easily
    `This would keep the amount of work reasonable`

08/07/01 V-20

Strategy: Use Recursion!

- Base case
  - an array of size 1 is already sorted!
- Recursive case
  - split array in half
  - use a recursive call to sort each half
  - combine the sorted halves into a sorted array
- Two ways to do the splitting/combining
  - mergesort
  - quicksort

08/07/01 V-21

Quicksort

- Discovered by Anthony Hoare (1962)
- Split in half ("Partition")
  - Pick an element **`midval`** of array (the *pivot*)
  - Partition array into two portions, so that
    1. all elements less than or equal to **midval** are left of it, and
    2. all elements those greater than **midval** are right of it
  - (Recursively) sort each of those 2 portions
- Combining halves
  - No work -- already in order!

08/07/01 V-22

Partitioning Example

- Before partition:
  - **5 10 3 0 12 15 2 -4 8**
- Suppose we choose 5 as the "pivot"
- After the partition:
  - What values are to the left of the pivot?
  - What values are to the right of the pivot?
  - What about the exact order of the partitioned array? Does it matter?
  - Is the array now sorted? Is it "closer" to being sorted?
  - What is the next step...

08/07/01 V-23

Quicksort Code

```
// sort A[0..N-1]
void quicksort(int A[], int N) {
  qsort(A, 0, N-1);
}
// sort A[lo..hi]
void qsort(int A[], int lo, int hi) {
  if ( lo >= hi ) return;
  int mid = partition(A, lo, hi);
  qsort(A, lo, mid-1);
  qsort(A, mid+1, hi);
}
```

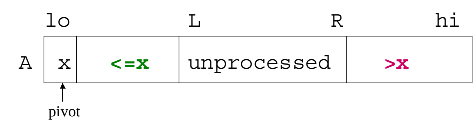
08/07/01 V-24

`Partition` Helper Function

- Partition will have to choose a pivot (midval)
  - Simple implementation: pivot on first element of array
- At the end, have to return new index of `midval`
  - We don't know in advance where it will end up!
- Have to rearrange `A[lo] .. A[hi]` so elements ≤ `midval` are left of `midval` , and the rest are right of `midval`
  - *this can be tricky code*

A `Partition` Implementation

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



- For simplicity, handle only one case per iteration
  - `This can be tuned to be more efficient, but not needed for our purposes.`

Partition

```
// Partition A[lo..hi]; return location of pivot
// Precondition: lo < hi
int partition(int A[],int lo,int hi){
  assert(lo < hi);
  int L = lo+1, R = hi;
  while (L <= R) {
    if      (A[L] <= A[lo]) L++;
    else if (A[R] >  A[lo]) R--;
    else { // A[L] > pivot && A[R] <= pivot
      swap(A[L],A[R]);
      L++; R--;
    }
  }
  // put pivot element in middle & return location
  swap(A[lo],A[L-1]);
  return L-1;
}
```


Example of Quicksort

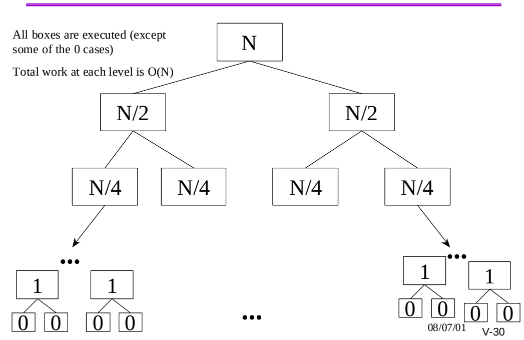
6 4 2 9 5 8 1 7

Complexity of Quicksort

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

    `Note: This n is smaller than the N of the original problem`
  - Some `O(1)` work
  - Total = `O(n)` for **`n`** 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)



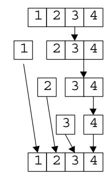
Best Case for Quicksort

- Assume **partition** will split array exactly in half
- Depth of recursion is then $\log_2 N$
- Total work is **O(N)*O(log N)=O(N log N)**, much better than $O(N^2)$ for selection sort
- Example: Sorting 10,000 items:
  - Selection sort: $10{,}000^2$ = 100,000,000
  - **Quicksort**: 10,000 $\log_2$ 10,000 ≈ **132,877**

08/07/01 V-31

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?

08/07/01 V-32

Average Case for Quicksort

- How to perform average-case analysis?
  - Assume data values are in random order
- What probability that `A[lo]` is the least element in `A`?
  - If data is random, it is **1/N**
- Expected time turns out to be `O(N log N)`, like best case

08/07/01 V-33

Back to Worst Case

- Can we do better than $O(N^2)$?
  - Depends on how we pick the pivot element **midval**
  - Lots of tricks have been tried
- One such trick:
  - pick **midval** *randomly* among `A[lo]`, `A[lo+1]`, …, `A[hi-1]`, `A[hi]`
  - Expected time turns out to be `O(N log N)`, *independent of input*

08/07/01 V-34

Divide & Conquer Revisited

- Quicksort illustrates "Divide and Conquer" approach:
  - 1. Divide the array into two parts, in some sensible way
    `Quicksort: "Partition"`
  - 2. Sort the two parts separately (recursively)
  - 3. Recombine the two halves easily
    `Quicksort: nothing to do at this step`
- Mergesort takes similar steps
  - 1. Divide the array
  - 2. Sort the parts recursively
  - 3. Recombine the parts

08/07/01 V-35

Mergesort

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

08/07/01 V-36

Mergesort Code

```
// Sort A[0..N-1] into ascending order
void mergesort(int A[], int N) {
   mergesort_help(A, 0, N-1);
}
// Sort A[lo..hi] into ascending order
void mergesort_help(int A[],int lo,int hi) {
    if (lo < hi) {
        int mid = (lo + hi) / 2;
        mergesort_help(A, lo, mid);
        mergesort_help(A, mid + 1, hi);
        merge(A, lo, mid, hi);
    }
}
```

08/07/01 V-37

Merge Code

```
// merge sequences A[lo..mid] & A[mid+1..hi],
// leaving merged result in A[lo..hi]
void merge(int A[], int lo, int mid, int hi){
    int left = lo; int right = mid + 1;
    int tempArray[MAX_SIZE];
    for (int i = 0; i <= hi-lo; ++i) {
        assert (left <= mid || right <= hi);
        assert (left <= right && left <= mid+1 && right <= hi+1);
        if (right == hi+1
                  || (left <= mid) && (A[left] < A[right]))
            tempArray[i] = A[left++];
        else
            tempArray[i] = A[right++];
 }
    for (int j = 0; j <= hi-lo; ++j)
        A[lo + j] = tempArray[j];
}
```

08/07/01 V-38

Mergesort Example

8 4 2 9 5 6 1 7

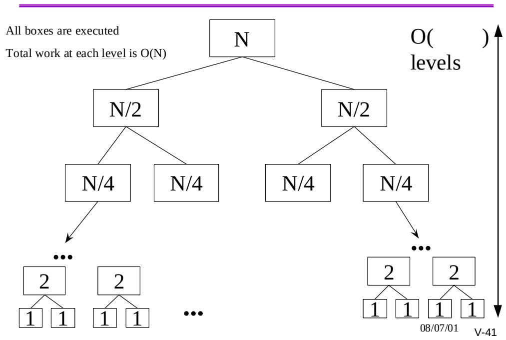
08/07/01 V-39

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?

08/07/01 V-40

Mergesort Recursion



08/07/01 V-41

Mergesort Space Complexity

- "Efficiency" refers to use of resources
  - Very often *time* is the resource
  - Could also be *space* (memory)
- Mergesort needs a temporary array at each call
  - Total temp. space is N at each level
  - Space complexity of O(N*logN)
- Compare with Quicksort, Selection Sort,etc:
  - None of them required a temp array
  - All were "in-place" sorts: space complexity O(N)

08/07/01 V-42

External Sorting

- *Random Factoid: Merging is the usual basis for sorting large data files*
  - Sometimes called "external" sorting
- Big files won't fit into memory all at once
- Pieces of the file are brought into memory, sorted internally, written out to sorted "runs" (subfiles) and then merged.
- Goes all the way back to early computers
  - Main memories and disks were extremely small
  - Large data files were stored on tape, which had (and still have) extremely high storage capacities

Guaranteed Fast Sorting

- There are other sorting algorithms which are always `O(N log N)`, even in worst case
  - Examples: Mergesort, Balanced Binary Search Trees, Heapsort
  - There are even **O(N)** algorithms: Radix, Bucket sort (see appendix to this lecture)
- Why not always use something other than Quicksort?
  - Others may be hard to implement, may require extra memory, have limitations
  - Hidden constants: a well-written quicksort will nearly always beat other algorithms

Summary

- Searching
  - Linear Search: `O(N)`
  - Binary Search: `O(log N)`, needs sorted data
- Sorting
  - Quadratics Sorts: $O(N^2)$
    `Selection, Insertion, Bubble`
  - Mergesort: `O(N log N)`
  - Quicksort: average: `O(N log N)`, worst-case: $O(N^2)$
  - Bucket, Radix (see appendix)
  - Many others (CSE373, CSE326)

Appendix

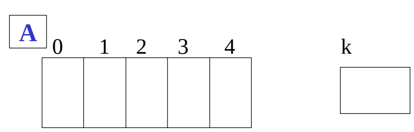
Selection Sort, Bucket Sort, and Radix Sort

Selection Sort

- Simple -- what you might do by hand
- Idea: Make repeated passes through the array, picking the smallest, then second smallest, etc., and move each to the front of the array

```
void selectionSort (int A[ ], int N) {
  for (int lo=0; lo<N-1; lo++) {
    int k = indexOfSmallest(A, lo, N-1);
    swap(A[lo], A[k]);
  }
}
```


Example



Analysis of IndexOfSmallest

- Finding the smallest element:

```
int indexOfSmallest(int A[ ], int lo, int hi) {
  int smallIndex = lo;
  for (int i=lo+1; i<=hi; i++)
    if (A[i] < A[smallIndex])
      smallIndex = i;
  return smallIndex;
}
```

- How much work does indexOfSmallest do?

Analysis of Selection Sort

- Loop in selectionSort iterates ___ times
- How much work is done each time...
  - by indexOfSmallest
  - by swap
  - by other statements
- Full formula:
- Asymptotic complexity:

Shortcut Analysis

- Go through outer loop about N times
- Each time, the amount of work done is no worse than about N+c
- So overall, we do about N*(N+c) steps, or $O(N^2)$

Guaranteed Fast Sorting

- There are other sorting algorithms which are always `O(N log N)`, even in worst case
  - Examples: Mergesort, Balanced Binary Search Trees, Heapsort
- Why not always use something other than Quicksort?
  - Others may be hard to implement, may require extra memory
  - Hidden constants: a well-written quicksort will nearly always beat other algorithms

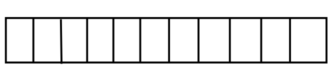
"Bucket Sort:" Even Faster Sorting

- Sort n *integers* from the range 1..m

1. Use temporary array T of size m
   initialized to some sentinel value
2. If v occurs in the data, "mark" T[v]
3. Make pass over T to "condense" the values

- Run time O(n + m)
- Example (n = 5, m = 6)

```
Data: 9, 3, 8, 1, 6
```

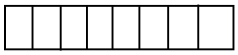
T


Reasons Not to Always Use Bucket Sort

- Integers might be from a large range
  - Social Security Numbers: requires an array T[999999999] no matter how few data points
  - Large arrays will either be disallowed by the compiler, or written to disk (causing extreme slowdown)
- You may not know m in advance
- Might be no reasonable sentinel value
  - If any positive or negative integer is possible
- Sort key might not be an integer
  - Salary, date, name, etc.

Radix Sort: Another Fast Sort

- Imagine you only had to sort numbers from 0 to 9
- First, figure out how many of each number
  - array: 4 6 2 7 9 7 4 4
  - occurrances? 0 1 2 3 4 5 6 7 8 9
- Next, calculate starting index for each number
  - indices? 0 1 2 3 4 5 6 7 8 9
- Last, put numbers into correct position



- Run time O(n)
- So far, this is identical to bucket sort...

08/07/01 V-55

Larger numbers

- What about 2 and 3-digit numbers?
- Sort low digits first, then high digits
  - original: 45 92 33 60 29 55 14
  - first pass:
  - final pass:
- Complexity
  - # of passes? work per pass? overall?
- Problems
  - You may not know # of digits in advance
  - Sort key might not be an integer

    Salary, date, name, etc.

08/07/01 V-56

Summary

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

    Other quadratic sorts: Insertion, Bubble
  - Mergesort: O(N log N)
  - Quicksort: O(N log N) average, $O(N^2)$ worst-case
  - Bucketsort: O(N) [but what about space??]
  - Radixsort: O(N * D)

08/07/01 V-57