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`** **`int`**s, 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);
}
```

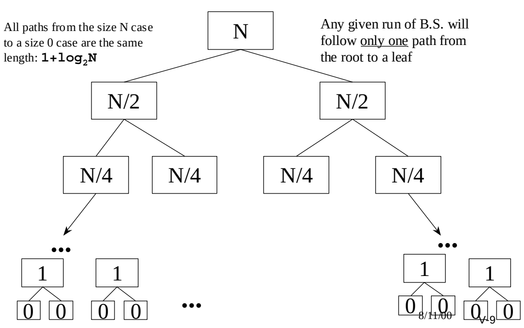

Analysis (recursive)

- 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

Binary Search Sizes

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

Any given run of B.S. will follow only one path from the root to a leaf



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

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)

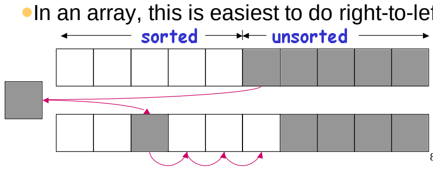
- Which ones are best?
  - Efficiency analysis is one way to compare

Sorts You May Know

- 142 review
  - Bubble Sort
    Some think it's a good "intro" sort
    Actually it is not a great choice
  - Selection Sort
    Covered in CSE142 textbook
  - Insertion Sort
    Will discuss shortly
- Mergesort
- Quicksort
- Radixsort

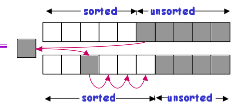
Insertion Sort

- 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



8/11/00 V-13

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
    }
}
```

8/11/00 V-14

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 easier to code (certainly no harder)
- Among fastest methods overall for n < 20 or so

8/11/00 V-15

Comparing Sorts

- Selection Sort: O(N^2)
- Bubble Sort: also O(N^2)
  - For each of the N elements, you "bubble" through the remaining (up to N) elements
- Insertion Sort: also O(N^2) in average case
  - But in practice usually faster than selection and bubble
- All are referred to as "quadratic" sorts

8/11/00 V-16

Can We Sort Faster Than O(N^2)?

- 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 Insertion 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*

8/11/00 V-17

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`

8/11/00 V-18

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
  - quicksort
  - mergesort

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
  - Nothing to do – they are already in place!

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

Quicksort

```
// 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);
}
```


`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 is tricky code*

A `Partition` Implementation

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

	lo		L	R	hi
A	x	<=x	unprocessed	>x	

pivot

- 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;
}
```

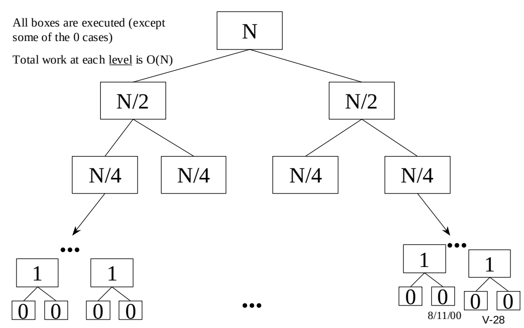

Example of Quicksort

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)

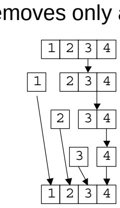


Best Case for Quicksort

- In the ideal case, **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 insertion sort
- Example: Sorting 10,000 items:
  - Selection sort: $10{,}000^2$ = 100,000,000
  - **Quicksort**: 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?

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

8/11/00 V-31

Back to Worst Case

- Can we do better than **`O(N²)`**?
  - 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*

8/11/00 V-32

Divide & Conquer Reviseted

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

8/11/00 V-33

Mergesort

- Split into two parts
  - No partition: 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 output array
  - Example: if the two arrays are

```
  1      12      15      20
  5       6      13      21      30
The "merged" output array is
  1  5  6  12  13  15  20  21  30
```

  - note: we will need a temporary result array

8/11/00 V-34

Mergesort Code

```
void mergesort(int A[], int N) {
   mergesort_help(A, 0, N-1);
}

void mergesort_help(int A[],int lo,int hi) {
    if (hi - lo >= 1) {
        int mid = (lo + hi) / 2;
        mergesort_help(A, lo, mid);
        mergesort_help(A, mid + 1, hi);
        merge(A, lo, mid, hi);
    }
}
```

8/11/00 V-35

Merge Code

```
 //merge the two arrays A[lo..mid] and A[mid+1..hi]
void merge(int A[], int lo, int mid, int hi){
     int * tempArray = new int [hi-lo+1];
     assert (tempArray != NULL);
     int fir = lo; int sec = mid + 1;
     for (int i = 0; i <= hi-lo; ++i) {
      if (sec == hi+1 ||
                (fir<=mid && A[fir] < A[sec]))
         tempArray[i] = A[fir++];
      else
         tempArray[i] = A[sec++];
        }
     for (int n = 0; n <= hi-lo; ++n) {
      A[lo + n] = tempArray[n];
        }
     delete [] tempArray;
}
//What are the crucial preconditions??
```

8/11/00 V-36

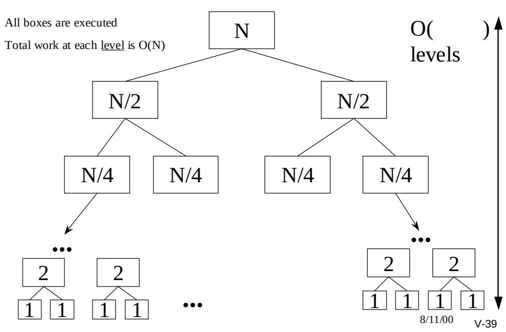
Mergesort Example

8 4 2 9 5 6 1 7

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?

Mergesort Recursion



Mergesort Space Complexity

- Mergesort (actually Merge) needs a temporary array at each call
- Compare with Quicksort, Insertion Sort,etc:
  - None of them required a temp array
  - All were "in-place" sorts
  - Merge's copying back to the original array also increases the run-time

Guaranteed Fast Sorting

- There are other sorting algorithms which are always `O(N log N)`, even in worst case
  - Examples: Balanced Binary Search Trees, Heapsort
  - Are also O(N) algorithms: Bucket and Radix Sort
    `Are not always applicable and have other drawbacks`
- Why not always use Quicksort?
  - Others may be hard to implement, may require extra memory
  - Hidden constants: a well-written quicksort will nearly always beat other algorithms
  - Only merge-based sorts work well with external (disk) data
  - Data considerations. E.g. Insertion sort is very fast when array is almost sorted already
  - Other properties, such as preserving order of duplicate keys ("stability").

Summary

- Searching
  - Linear Search: `O(N)`
  - Binary Search: `O(log N)`, needs sorted data
- Sorting
  - Insert, Selection Sort: O(N^2)
    `Other quadratic sorts: Bubble`
  - Mergesort: `O(N log N)`
  - Quicksort: average: `O(N log N)`, worst-case: O(N^2)

Appendix

Selection Sort, Bucket Sort, and Radix Sort

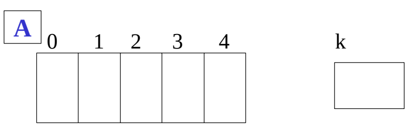
8/11/00 V-43

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]);
   }
}
```

8/11/00 V-44



8/11/00 V-45

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?

8/11/00 V-46

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:

8/11/00 V-47

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 $O(N^2)$

8/11/00 V-48

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...

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.`

Summary

- Searching
  - Linear Search: `O(N)`
  - Binary Search: `O(log N)`, needs sorted data
- Sorting
  - Selection Sort: `O(N²)`
    `Other quadratic sorts: Insertion, Bubble`
  - Mergesort: `O(N log N)`
  - Quicksort: `O(N log N)` average, `O(N²)` worst-case
  - Bucketsort: O(N) [but what about space??]
  - Radixsort: `O(N * D)`