CSE 143

Searching and Sorting

[Chapter 9, pp. 402-432]

Linear Search

- Given an array **A** of **N** **int**s, how efficient is it to 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;
}
```

- 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)**

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

Sorting

- Binary search requires a sorted input array
  - How to make array sorted?
- Many other applications need sorted input array
  - Language dictionaries
  - Telephone books
  - Printing data in organized fashion
  - 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)

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++)
     swap(A[lo], indexOfSmallest(A, lo, N-1));
}
```


Analysis of Selection Sort

- Finding the smallest element:

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

- Loop in selectionSort iterates ?? times:
- How much work does the helper function indexOfSmallest do?

Quicksort

- Idea: Apply divide and conquer approach to sorting
- Plan: To sort an array **`A`** with **`N`** elements:
  - Pick an element **`midval`** of array (the *pivot*)
  - Partition elements so those less than or equal to **`midval`** are left of it, and those greater than **`midval`** are right of it
  - Recursively sort each of those 2 portions

Quicksort Implementation

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

void QuicksortInRange(int A[], int lo,int hi) {
  if ( lo >= hi ) return;
  int midindex = Partition(A, lo, hi);
  QuicksortInRange(A, lo, midindex-1);
  QuicksortInRange(A, midindex+1, hi);
}
```

`Partition` Helper Function

- Let `midval` = `A[lo]`
- Rearrange `A[lo] .. A[hi]` so elements ≤ `midval` are left of `midval` , and the rest are right of `midval`
- Return new index of `midval`

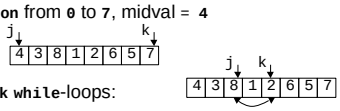
`Partition`

```
int Partition(int A[], int lo, int hi) {
  int midval = A[lo];
  int j = lo, k = hi;

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

  Swap(A[lo], A[k]);
  return k;
}
```


Example of Quicksort

Array: 4 3 8 1 2 6 5 7

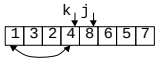
`Partition` from **`0`** to **`7`**, midval = **`4`**



After **`j, k while`**-loops:

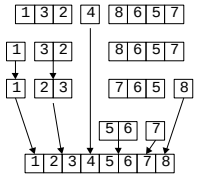
`j` and **`k`** have not crossed, so swap and continue

`j` and **`k`** then cross, so move midpoint value and return **`k=3`**



Example of Quicksort (2)

Now recursively Quicksort the two halves :



Complexity of `Partition`

- Nested loops may look slow, but
  - **`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

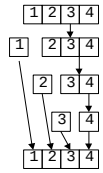
- Each call to Quicksort (ignoring recursive calls):
  - One call to **`Partition = O(N)`**, where **`N`** is size of *part* of array being sorted
  - Some **`O(1)`** work
  - Total **`= O(N)`** for **`N`** the size of array part being sorted
- Including recursive calls:
  - Two recursive calls at each level of recursion, each does "half" the array **`= O(N)`** at each level of recursion
  - How many levels of recursion?

Best Case for Quicksort

- Assume **`Partition`** will split array exactly in half
- Depth of recursion is then $\log_2 N$, just like for binary search
- 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: $10{,}000 \log_2 10{,}000 \approx 132{,}877$

Worst Case for Quicksort

- If we're not lucky, 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

Back to Worst Case

- Can we do better than $O(N^2)$?
  - Depends on how we pick the pivot element **`midval`**
- 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*

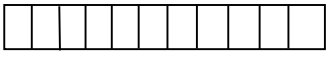
Guaranteed Fast Sorting

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

Even Faster Sorting

- Sort *integers* in the range 1..m
- Use array A of size m
- Store cell with value v in A[v]
- Make pass over the array to produce the values
- Run time O(n + m)
- "Bucket Sort"

```
9, 3, 8, 1, 6
```



Summary

- Searching
  - Linear Search: `O(N)`
  - Binary Search: `O(log N)`, requires sorted data
- Sorting
  - Selection Sort: $O(N^2)$
  - Quicksort: `O(N log N)` average, $O(N^2)$ worst-case

3/10/99 471