Sorting, Sets, and Selection

Divide and Conquer Algorithms

1. Divide large problem into several similar, but smaller sub-problems
2. Solve each sub-problem (recursively)
3. Combine results to solve original problem

Johns Hopkins Department of Computer Science
Course 600.226: Data Structures, Professor: Jonathan Cohen

Merge-sort

Input: unsorted sequence

Output: sorted sequence

1. If input size is 1, return
2. Split sequence of size n into two sequences of size n/2 according to position
3. Recursively call merge sort on sub-sequences
4. Merge two sorted sequences into one sorted sequence

Johns Hopkins Department of Computer Science
Course 600.226: Data Structures, Professor: Jonathan Cohen

Merging Sorted Sequences

While neither sequence is empty
- Compare first element in each sequence
- Remove smallest and insert into output

Insert all remaining elements into output

O(n) running time
- n is sum of two sequence lengths

Johns Hopkins Department of Computer Science
Course 600.226: Data Structures, Professor: Jonathan Cohen

Analyzing Merge Sort

Number of levels in recursion tree is O(logn)

Each element appears in one sequence per level

Total work done is linear at each call (i.e. O(1) work per element

Therefore, total work is n*O(logn) = O(nlogn)

Johns Hopkins Department of Computer Science
Course 600.226: Data Structures, Professor: Jonathan Cohen

Recurrence Relations

Express total running time as a recursive function

Converting to closed form solution gives running time
- see “Master Method” in appendix A

Johns Hopkins Department of Computer Science
Course 600.226: Data Structures, Professor: Jonathan Cohen

Recurrence relation for merge-sort

T(n) = a n<=1
2T(n/2) + cn n>1

T(n) = 2*(2T(n/4) + c(n/2)) + cn
= 4T(n/4) +2cn
= $2^i T(n/2^i)$ + icn

Recursion stops when $n=2^i$ (i=logn)

T(n) = $2^{\log n}$T(1) + c*nlogn
= a*n + c*nlogn
= O(nlogn)

Quick-sort

Input: unsorted sequence

Output: sorted sequence

1. If input size is 1, return
2. Choose *pivot* element (perhaps last element)
3. Create sub-sequences L, E, and G
   - less than, equal to, or greater than pivot element
3. Recursively call quick-sort on L and G
4. Merge 3 sorted sequences into one sorted sequence
   - Trivial concatenation

Johns Hopkins Department of Computer Science
Course 600.226: Data Structures, Professor: Jonathan Cohen

Worst-case Analysis

Possibly choose “bad” pivot at every call
- L or G has size 0 (or very small)
- G or L has size n-1

Recursion has depth n
- $O(n)$ work at each recursion level

Total work is $O(n^2)$

Johns Hopkins Department of Computer Science
Course 600.226: Data Structures, Professor: Jonathan Cohen

Randomized Quick-sort

Choose a random element as pivot at each step

Define “good” pivot as one which has neither partition less than n/4 or greater than 3/4 n
- 50% chance of picking good partition
- Expect recursion height to be 2 times the height resulting from picking all good partitions

If all pivots are good, find recursion depth, d
$n*(3/4)^d = 1$ ® $n = (4/3)^d$ ® $d = \log_{4/3} n$

Expect depth is 2 *$\log_{4/3} n$

$O(n)$ work per level: $O(n\log n)$ total expected work

Johns Hopkins Department of Computer Science
Course 600.226: Data Structures, Professor: Jonathan Cohen

Lower Bound on Comparison-based Sorting

Heap-sort, merge-sort, quick-sort all $O(n\log n)$

Is it possible to do better?

Prove a “lower bound” on certain types of sorting
- sorts based on comparing two elements

Johns Hopkins Department of Computer Science
Course 600.226: Data Structures, Professor: Jonathan Cohen

Comparison Sort Decision Tree

Each internal node is comparison operation
- branch one way for true, the other for false

Each external nodes is a unique permutation of input
- number of permutations is n! = n(n-1)(n-2)...(2)(1)
- height of decision tree is
  $\log(n!) >= \log(n/2)^{n/2} = n/2 * \log(n/2)$ ® W$(n\log n)$
- Sort is traversing path from root to leaf = W$(n\log n)$

Johns Hopkins Department of Computer Science
Course 600.226: Data Structures, Professor: Jonathan Cohen

Bucket-Sort

Sort certain inputs without comparing elements
- Assume elements have integer keys in range [0,N-1]
- Create bucket (sequence) for each possible key
- Drop each element into proper bucket
- Merge buckets in correct order

O(n + N) : number of elements plus number of buckets

Works well if N is $o(n\log n)$

Radix-Sort

Multi-pass bucket-sort keys with d components
- Sort by key in lexicographical (dictionary) order

First sort over last key, then next to last, etc.

Uses N buckets instead of N^d buckets

Running time $O(d(n+N))$

Only efficient if d is $O(\log n)$
- (especially if there are duplicate keys)

Comparing various sorts

Insertion sort: O(n+k)
- Good for small lists and nearly sorted lists

Merge-sort: O(nlogn)
- Time efficient, but hard to run "in place"
- Good for external memory sorting

Quick-sort (randomized): expected $O(n\log n)$
- very fast in practice, but occasionally $O(n^2)$

Heap-sort: O(nlogn)
- Always pretty fast

Bucket/radix sort: good if d*(n+N) is o(nlogn)

Selection

Find the kth greatest item in a sequence
- Can we do it faster than sorting?
  - —Clearly yes for k=1 or k=n
    - »Also in time k*n for some constant k
  - —Not so clear for k = n/2

Decrease and Conquer

Like divide and conquer, but for searching
- Hopefully do not need to search all subgroups
- E.g. binary search is decrease and conquer

Randomized Quick Select

If sequence length is 1, return the element

As in quick *sort*, pick a random pivot

Partition sequence into <, =, > subsequences
- If "=" contains kth element, return pivot
- Recurse into subsequence (< or >) containing kth element

Analysis of Quick Select

"Good pivot"

- Partitions into subsequences of size < 3/4 n
- 50% of elements are good pivots
- Expected number of elements to try is 2

T(n) <= T(3/4 n) + 2bn

<= T($(3/4)^2$ n) + 2bn(1 + 3/4)

<= 2bn

= $O(n)$ expected time

Set ADT

Set: container of distinct elements

- No duplicates
- No explicit ordering or keys necessary

Operations

- Union
  - AÈB: all elements in either A or B
- Intersection
  - AÇB: all elements in both A and B
- Difference
  - A-B: all elements in A but not B

Implementation Difficulty

Performing methods requires finding duplicates and applying method-specific logic

- Finding duplicates is hard without some sort of order
- Impose order by defining comparator for members
  - —Almost any type of comparator will do as long as it is consistent (i.e. identifies duplicates, and a<b implies b>a)

Implementing Sets as Sorted Sequences

Each set sorted according to the comparator

Operations may be perform as variants of merge operation (similar to merge sort)

- Union: insert all elements into output set, but duplicates only once
- Intersection: insert only duplicates (but each only once)
- Difference: insert all elements from set A unless duplicated in set B

Analysis of Set ADT

Each operation involves only a single pass of the merge algorithm

Worst case time: $O(n)$

Insert may be done in $O(n)$ via Union

Remove may be done in $O(n)$ via Difference