CS C341 / IS C361
Data Structures & Algorithms

2/22/2014

SORTING – PERFORMANCE AND APPLICATIONS

Comparative Performance of Algorithms

Lower Bound on Comparison-based Sorting

Sorting without Comparisons

- Bucket Sorting
- Radix Sorting

Why Sort?

Online vs. Offline, Preprocessing.

Dictionary Data Structures

SORTING ALGORITHMS – COMPARISON OF APPROACHES

Feature → Algo.	Insertion Sort	Merge Sort	QuickSort
Ordering Principle	Insertion *preserves* Order	Merging *preserves* Order	Partition *induces* Order
Ranking Operation	Comparison	Comparison	Comparison
Positioning Operation	Shift	Copy	Exchange

Sorting – Performance Comparison

Metric / Algo.	Insertion Sort	Merge Sort	QuickSort
Worst Case Time	O(N*N)	O(NlogN)	O(N*N) w. low prob.
Average Case Time	O(N*N)	O(NlogN)	O(NlogN) w. high prob.
Performance on small lists	Extremely good	Not good	Not good
Space	O(1)	O(N)	O(logN)
Online/Offline	Online	Partly Online	Offline
Memory access	Seq. Read (find) Random Write (insert)	Seq. Read Seq. Write	Random Read Random Write

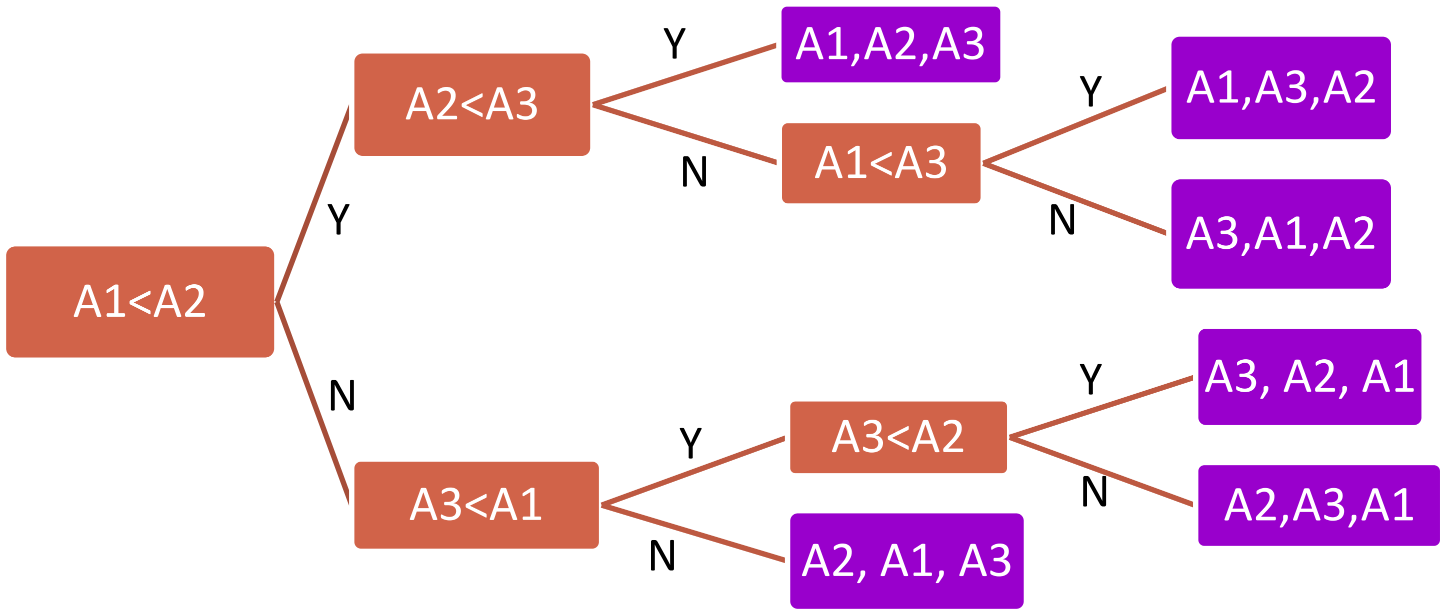
Why?

Sorting – Lower Bound

- Is this the best we can do for Sorting?
  - Algorithm Complexity vs. Problem Complexity
- Sorting
  - Can be solved in polynomial time – in particular in O(NlogN) time (worst case)
    - Witness: Merge Sort
  - Is there a lower bound (on time complexity) for sorting?
    - i.e. is there a (lower) limit for the time taken for sorting a list of N elements using any sorting algorithm?
  - If we assume that values must be compared, then yes, there is a bound.
    - Lower Bound on number of comparisons required – in the worst case - for sorting: $\Omega(N\log N)$

Sorting – Lower Bound [2]

Decision Tree for Sorting (a list of 3 unique values)



Internal Nodes (Decision Nodes)

External Nodes (Results)

How many decisions are necessary?

N! permutations

SORTING – LOWER BOUND [3]

- Minimum number of decisions necessary
  - is the same as the minimum depth of a (binary) tree with N! nodes.
- Depth of a tree is the length of the longest path from root to an external node.
  - Number of nodes can grow geometrically – in the best case - at every level i.e. # nodes at each level is 1, 2, 4, …
  - Thus if the total number of nodes in the tree is M the longest path would be at least log(M)
- In our case, the depth is log(N!)
  - And hence the minimum number of decisions is log(N!)

Sorting – Lower Bound [4]

- Two ways to simplify log(N!):
  - Stirling's approximation: $n! = (\sqrt{(2\pi n)})(n/e)^n(1+\Theta(1/n))$

    i.e. $n! > (n/e)^n$
    - Then, $\log(n!) > n\log(n) - n\log(e)$
  - By definition, $n! = \prod_{j=1 \text{ to } n} j$
    - $\log(n!) = \Sigma_{j=1 \text{ to } n} \log(j) = \Sigma_{j=2 \text{ to } n} \log(j)$
    - Because log is a monotonically increasing function

    $\Sigma_{j=2 \text{ to } n} \log(j) >= \int_{x=1 \text{ to } n} \log(x)\, dx = n \log(n) - n - 1$
- Either way, the minimum number of comparisons required for sorting N values is $\Omega(N\log N)$

SORTING - ALTERNATIVES

- Can we sort without comparison?
  - Yes, if keys carry inherent information that is useful for positioning
    - This is true when the range of keys are known
  - Refer to exercises on Quicksort:
    - QuickSort for sorting lists with a constant number of unique keys.
    - Radix Exchange Sort

Sorting By Distribution

- Bucket Sorting (or Bin Sorting)
  - If the range of keys is known & finite,
  - then one can distribute the values into different "buckets" for different keys.
- If keys are unique integers then the distribution results in a $\Theta(N)$ time, $\Theta(K)$ space sorting algorithm
  - N is the number of elements to be sorted.
  - K is the range size and K is $O(N)$

```
bucketSort(int A[], int size, int lowKey, int hiKey) {
   allocate array Temp[0 .. (hiKey – lowKey)] ;
   for j = 0 to size-1  { Temp[A[j].key - lowKey] = A[j]; }
 }
     // copy back if the original array must be used
     // Caveat: Given K>N some buckets would be empty.
```

BUCKET SORTING

- Extensions:
  - What if keys are not unique?
    - Then buckets may not be unit sized
    - Solution: Implement buckets as linked lists
  - What if keys are not integers?
    - Is there a 1-1 mapping between key type and a small range of integers?
      - E.g. Points in 2-d space
      - E.g. Strings of characters such that
        - length < m for some constant m and alphabet is finite.
- Variations:
  - Count Sorting or Frequency Sorting
  - Interval Sorting

Stability of Sorting

- A sorting algorithm is stable
  - if for any two items in the list before sorting, say Ai and Aj such that i<j and key(Ai) = key(Aj),
  - Ai precedes Aj in the sorted list as well.
- Why is Stability important?
- Which of the algorithms discussed so far is/are stable?
  - OR under what conditions these algorithms can be made stable?
    - Insertion Sort
    - Merge Sort
    - Quick Sort
    - Bucket Sort (or Bin Sort)

Ordering by Multiple Keys

- In our examples, we assumed that sorting is done on the basis of a single key (that is part of the record).
  - This may not affect our comparison-based sorting algorithms:
    - comparison can be encapsulated (i.e. a multi-key comparison algorithm is used).
- Multi-Key Ordering (i.e. where keys are tuples)
  - e.g. Sorting cards by suit and face
  - e.g. Sorting student records by Group and CGPA
- Multi-key ordering is referred to as lexicographic ordering (i.e. dictionary ordering) - if in particular, the keys are all of the same type
  - E.g. Sort a list of points (say, in 2-D space) by the (x and y) co-ordinates.

Ordering by Multiple Keys

- Alternative to multi-key comparison-based sorting:
  - Sort first by the first key, then by the second key and so on.
    - Will this work?
      - Only if a subsequent sorting is restricted to equal key values of the previous sorts.
    - This results in multiple levels of bucketing in bucket sorting
  - What if the order of sorts is reversed?
    - i.e. sort by the last key, then by the penultimate key, and so on until the first key.
      - e.g. if you want to sort in the order (Group, CGPA) first sort by CGPA then sort the whole list by Group
    - Will this work?
      - Yes, if the algorithm(s) are stable.

Radix Sorting

- Solution for Multi-key Sorting:
  - Use (stable) Bucket sort repeatedly – from last key to first key.
- This is referred to as Radix Sort – when the digits (i.e. radix) of a number can be considered keys.
  - E.g. Sorting a list of 5 digit numbers can be achieved by 5 Bucket Sorts – using each digit as a key at a time – from last digit to the first.

Sorting vs. Searching - Preprocessing

- Why do we Sort?
  - When is it useful?
    - Efficient Searching, Enumeration in order or groups
- Online vs. Offline
  - Retrieval has to be done online whereas storage may be done offline
    - If Sorting can be done offline, online work (search) is reduced.
  - What can be done offline?

PREPROCESSING

- Preprocessing (offline) vs. Querying (online)
  - Time complexity (Preprocessing complexity and Querying Complexity)
    - E.g. Sorting is Preprocessing; searching (or finding) is querying.
    - E.g. Considering the problem of *factoring integers*
      - Preprocessing: Computing a (long) sequence of prime numbers
      - Querying: Testing whether the numbers in the sequence are factors of a given number.
  - Assumption:
    - Cost of pre-processing is amortized over many queries.

ADT Dictionary

- A dictionary ADT has the following operations:
  - Element find(Key, Dictionary)
  - Dictionary add(Element, Dictionary)
  - Dictionary delete(Key, Dictionary)
- *find* is the most common operation

- A dictionary may be ordered or unordered
  - If a dictionary must be ordered then sorting (as preprocessing) is required.

Sorting as Solution to Dictionary

- Comparison-Sorting
  - Results in a sorted list
    - *find* costs O(logN) time
    - *add* and *delete* can be implemented in O(logN) time but not with arrays (or linked lists).
- Bucketing [w. unique keys]
  - Results in a direct-address table:
    - *find* costs O(1) time
    - *add* costs O(1) time
    - *delete* costs O(1) time
  - Assumption: *null* element defined
  - Caveats:
    - Size of table vs. number of elements
    - Keys may not be unique

SETS

- A special case of an (unordered) dictionary is a set:
  - ~~Element find(Key, Dictionary)~~ boolean member(key, Dictionary)
  - Dictionary add(Element, Dictionary)
  - Dictionary delete(Key, Dictionary)
- Can we simplify the table?
  - [Hint: member returns a boolean]
  - What is the total memory requirement?