CS 361, Lecture 19

Jared Saia
University of New Mexico

Outline

- Lower Bound for Sorting by Comparison
- Bucket Sort
- Dictionary ADT

Administrivia

- Appendix C.1 in the book is an excellent reference for background math on counting
- Appendix C.2 is good background for probability

How Fast Can We Sort?

- Q: What is a lowerbound on the runtime of any sorting algorithm?
- We know that $\Omega(n)$ is a trivial lowerbound
- But all the algorithms we've seen so far are $O(n \log n)$ (or $O(n^2)$), so is $\Omega(n \log n)$ a lowerbound?

Comparison Sorts

- Definition: An sorting algorithm is a *comparison sort* if the sorted order they determine is based only on comparisons between input elements.
- Heapsort, mergesort, quicksort, bubblesort, and insertion sort are all comparison sorts
- We will show that any comparison sort must take $\Omega(n \log n)$

Comparisons

- Assume we have an input sequence $A = (a_1, a_2, \ldots, a_n)$
- In a comparison sort, we only perform tests of the form $a_i < a_j$, $a_i \leq a_j$, $a_i = a_j$, $a_i \geq a_j$, or $a_i > a_j$ to determine the relative order of all elements in A
- We'll assume that all elements are distinct, and so note that the only comparison we need to make is $a_i \leq a_j$.
- This comparison gives us a yes or no answer

Decision Tree Model

- A decision tree is a full binary tree that gives the possible sequences of comparisons made for a particular input array, A
- Each internal node is labelled with the indices of the two elements to be compared
- Each leaf node gives a permutation of A

Decision Tree Model

- The execution of the sorting algorithm corresponds to a path from the root node to a leaf node in the tree.
- We take the left child of the node if the comparison is $\leq$ and we take the right child if the comparison is $>$
- The internal nodes along this path give the comparisons made by the alg, and the leaf node gives the output of the sorting algorithm.

Leaf Nodes

- Any correct sorting algorithm must be able to produce each possible permutation of the input
- Thus there must be at least $n!$ leaf nodes
- The length of the longest path from the root node to a leaf in this tree gives the worst case run time of the algorithm (i.e. the height of the tree gives the worst case runtime)

Height of Decision Tree

- Q: What is the height of a binary tree with at least $n!$ leaf nodes?
- A: If h is the height, we know that $2^h \geq n!$
- Taking log of both sides, we get $h \geq \log(n!)$

Height of Decision Tree

- Q: What is $\log(n!)$?
- A: It is

$$\begin{aligned}
\log(n * (n-1) * \cdots * 1) &= \log n + \log(n-1) + \cdots + \log(1) && (1)\\
&\geq (n/2)\log(n/2) && (2)\\
&\geq (n/2)(\log n - \log 2) && (3)\\
&= \Omega(n \log n) && (4)
\end{aligned}$$

- Thus any decision tree for sorting n elements will have a height of $\Omega(n \log n)$

Take Away

- We've just proven that any comparison-based sorting algorithm takes $\Omega(n \log n)$ time
- This does *not* mean that *all* sorting algorithms take $\Omega(n \log n)$ time
- In fact, there are non comparison-based sorting algorithms which, under certain circumstances, are asymptotically faster.

Bucket Sort

- Bucket sort assumes that the input is drawn from a uniform distribution over the range $[0, 1)$
- Basic idea is to divide the interval $[0, 1)$ into n equal size regions, or buckets
- We expect that a small number of elements in A will fall into each bucket
- To get the output, we can sort the numbers in each bucket and just output the sorted buckets in order

Bucket Sort

```
//PRE: A is the array to be sorted, all elements in A[i] are between
$0$ and $1$ inclusive.
//POST: returns a list which is the elements of A in sorted order
BucketSort(A){
B = new List[]
n = length(A)
for (i=1;i<=n;i++){
  insert A[i] at end of list B[floor(n*A[i])];
}
for (i=0;i<=n-1;i++){
  sort list B[i] with insertion sort;
}
return the concatenated list B[0],B[1],...,B[n-1];
}
```


Bucket Sort

- If the input numbers are distributed uniformly over the range $[0, 1)$, then Bucket sort takes expected time $O(n)$
- Proof of this is given in the book, make sure you understand it.

Dictionary ADT

A dictionary ADT implements the following operations

- *Insert(x)*: puts the item x into the dictionary
- *Delete(x)*: deletes the item x from the dictionary
- *IsIn(x)*: returns true iff the item x is in the dictionary

Dictionary ADT

- Frequently, we think of the items being stored in the dictionary as *keys*
- The keys typically have *records* associated with them which are carried around with the key but not used by the ADT implementation
- Thus we can implement functions like:
  - *Insert(k,r)*: puts the item (k,r) into the dictionary if the key k is not already there, otherwise returns an error
  - *Delete(k)*: deletes the item with key k from the dictionary
  - *Lookup(k)*: returns the item (k,r) if k is in the dictionary, otherwise returns null

Implementing Dictionaries

- The simplest way to implement a dictionary ADT is with a linked list
- Let l be a linked list data structure, assume we have the following operations defined for l
  - head(l): returns a pointer to the head of the list
  - next(p): given a pointer p into the list, returns a pointer to the next element in the list if such exists, null otherwise
  - previous(p): given a pointer p into the list, returns a pointer to the previous element in the list if such exists, null otherwise
  - key(p): given a pointer into the list, returns the key value of that item
  - record(p): given a pointer into the list, returns the record value of that item

In-Class Exercise

Implement a dictionary with a linked list

- Q1: Write the operation Lookup(k) which returns a pointer to the item with key k if it is in the dictionary or null otherwise
- Q2: Write the operation Insert(k,r)
- Q3: Write the operation Delete(k)
- Q4: For a dictionary with n elements, what is the runtime of all of these operations for the linked list data structure?
- Q5: Describe how you would use this dictionary ADT to count the number of occurences of each word in an online book.

Dictionaries

- This linked list implementation of dictionaries is very slow
- Q: Can we do better?
- A: Yes, with hash tables, AVL trees, etc