CS 361, Lecture 12

Jared Saia
University of New Mexico

Outline

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

How Fast Can We Sort?

- Q: What is a lower bound on the runtime of any sorting algorithm?
- We know that $\Omega(n)$ is a trivial lower bound
- 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 lower bound?

Administrivia

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

- Definition: A 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.

### 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 $A$.

### 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

- 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.
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)

Example
Consider the problem of sorting an array of size two: $A = (a_1, a_2)$
Following is a decision tree for this problem.

In-Class Exercise
Give a decision tree for sorting an array of size three: $A = (a_1, a_2, a_3)$
What is the height? What is the number of leaf nodes?

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

\[
\log(n \cdot (n-1) \cdot \cdots \cdot 1) = \log n + \log(n-1) + \cdots + \log 1 \\
\geq (n/2) \log(n/2) \\
\geq (n/2)(\log n - \log 2) \\
= \Omega(n \log n)
\]

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

```c
//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

- Claim: If the input numbers are distributed uniformly over the range $[0, 1)$, then Bucket sort takes expected time $O(n^2)$
- Let $T(n)$ be the run time of bucket sort on a list of size $n$
- Let $n_i$ be the random variable giving the number of elements in bucket $B[i]$
- Then $T(n) = \Theta(n) + \sum_{i=0}^{n-1} O(n_i^2)$

Analysis

- We claim that $E(n_i^2) = 2 - 1/n$
- To prove this, we define indicator random variables: $X_{ij} = 1$ if $A[j]$ falls in bucket $i$ and 0 otherwise (defined for all $i$, $0 \leq i \leq n - 1$ and $j$, $1 \leq j \leq n$)
- Thus, $n_i = \sum_{j=1}^{n} X_{ij}$
- We can now compute $E(n_i^2)$ by expanding the square and regrouping terms

Analysis

- We know $T(n) = \Theta(n) + \sum_{i=0}^{n-1} O(n_i^2)$
- Taking expectation of both sides, we have
  \[
  E(T(n)) = E(\Theta(n) + \sum_{i=0}^{n-1} O(n_i^2))
  = \Theta(n) + \sum_{i=0}^{n-1} E(O(n_i^2))
  = \Theta(n) + \sum_{i=0}^{n-1} (O(E(n_i^2)))
  \]
- The second step follows by linearity of expectation
- The last step holds since for any constant $a$ and random variable $X$, $E(aX) = aE(X)$ (see Equation C.21 in the text)
We can evaluate the two summations separately. $X_{ij}$ is 1 with probability $1/n$ and 0 otherwise.

Thus $E(X_{ij}^2) = 1 \cdot (1/n) + 0 \cdot (1 - 1/n) = 1/n$.

Where $k \neq j$, the random variables $X_{ij}$ and $X_{ik}$ are independent.

For any two independent random variables $X$ and $Y$, $E(XY) = E(X)E(Y)$ (see C.3 in the book for a proof of this).

Thus we have that

$$E(X_{ij}X_{ik}) = E(X_{ij})E(X_{ik})$$

$$= (1/n)(1/n)$$

$$= (1/n^2)$$

Recall that $E(T(n)) = \Theta(n) + \sum_{i=0}^{n-1} O(E(n_i^2))$.

We can now plug in the equation $E(n_i^2) = 2 - (1/n)$ to get:

$$E(T(n)) = \Theta(n) + \sum_{i=0}^{n-1} 2 - (1/n)$$

$$= \Theta(n) + \Theta(n)$$

$$= \Theta(n)$$

Thus the entire bucket sort algorithm runs in expected linear time.

Substituting these two expected values back into our main equation, we get:

$$E(n_i^2) = \sum_{j=1}^{n} E(X_{ij}^2) + \sum_{1 \leq j \leq n} \sum_{1 \leq k \leq n, k \neq j} E(X_{ij}X_{ik})$$

$$= \sum_{j=1}^{n} (1/n) + \sum_{1 \leq j \leq n} \sum_{1 \leq k \leq n, k \neq j} (1/n^2)$$

$$= n(1/n) + (n)(n - 1)(1/n^2)$$

$$= 1 + (n - 1)/n$$

$$= 2 - (1/n)$$

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
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 occurrences 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
Hash Tables implement the Dictionary ADT, namely:

- Insert(x) - $O(1)$ expected time, $\Theta(n)$ worst case
- Lookup(x) - $O(1)$ expected time, $\Theta(n)$ worst case
- Delete(x) - $O(1)$ expected time, $\Theta(n)$ worst case

Direct Addressing Functions

```
DA-Search(T, k) { return T[k]; }
DA-Insert(T, x) { T[key(x)] = x; }
DA-Delete(T, x) { T[key(x)] = NIL; }
```

Each of these operations takes $O(1)$ time.

Direct Addressing Problem

- Suppose universe of keys is $U = \{0, 1, \ldots, m - 1\}$, where $m$ is not too large
- Assume no two elements have the same key
- We use an array $T[0..m - 1]$ to store the keys
- Slot $k$ contains the elem with key $k$

- If universe $U$ is large, storing the array $T$ may be impractical
- Also much space can be wasted in $T$ if number of objects stored is small
- Q: Can we do better?
- A: Yes we can trade time for space