Today's Outline

- Randomized Quicksort
- Sorting Lowerbound
- Bucket Sort
- Dictionary ADT

R-Partition

//PRE: A[p..r] is the array to be partitioned, p>=1 and r <= size of A
//POST: Let A’ be the array A after the function is run. Then
// A’[p..r] contains the same elements as A[p..r]. Further,
// all elements in A’[p..res-1] are <= A[i], A’[res] = A[i],
// and all elements in A’[res+1..r] are > A[i], where i is
// a random number between $p$ and $r$.

R-Partition (A,p,r){
    i = Random(p,r);
    exchange A[r] and A[i];
    return Partition(A,p,r);
}

Randomized Quicksort

//PRE: A is the array to be sorted, p>=1, and r is <= the size of A
//POST: A[p..r] is in sorted order

R-Quicksort (A,p,r){
    if (p<r){
        q = R-Partition (A,p,r);
        R-Quicksort (A,p,q-1);
        R-Quicksort (A,q+1,r);
    }
}
Analysis

- R-Quick sort is a randomized algorithm
- The run time is a random variable
- We’d like to analyze the expected run time of R-Quick sort
- To do this, we first need to learn some basic probability theory.

Plan of Attack

“If you get hold of the head of a snake, the rest of it is mere rope” - Akan Proverb

- We will analyze the total number of comparisons made by quicksort
- We will let $X$ be the total number of comparisons made by R-Quick sort
- We will write $X$ as the sum of a bunch of indicator random variables
- We will use linearity of expectation to compute the expected value of $X$

Notation

- Let $A$ be the array to be sorted
- Let $z_i$ be the $i$-th smallest element in the array $A$
- Let $Z_{i,j} = \{z_i, z_{i+1}, \ldots, z_j\}$

Indicator Random Variables

- Let $X_{i,j}$ be 1 if $z_i$ is compared with $z_j$ and 0 otherwise
- Note that $X = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{i,j}$
- Further note that
  \[
  E(X) = E(\sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{i,j}) = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} E(X_{i,j})
  \]
Questions

• Q1: So what is $E(X_{i,j})$?
• A1: It is $P(z_i$ is compared to $z_j$) 

• Q2: What is $P(z_i$ is compared to $z_j$)?
• A2: It is: $P($either $z_i$ or $z_j$ are first elem in $Z_{i,j}$ chosen as pivots)

Why?

– If no element in $Z_{i,j}$ has been chosen yet, no two elements in $Z_{i,j}$ have yet been compared, and all of $Z_{i,j}$ is in same list
– If some element in $Z_{i,j}$ other than $z_i$ or $z_j$ is chosen first, $z_i$ and $z_j$ will be split into separate lists (and hence will never be compared)

More Questions

• Q: What is $P($either $z_i$ or $z_j$ are first elem in $Z_{i,j}$ chosen as pivots$)$
• A: $P(z_i$ chosen as first elem in $Z_{i,j}$) + $P(z_j$ chosen as first elem in $Z_{i,j}$) 

Further note that number of elems in $Z_{i,j}$ is $j - i + 1$, so

$$P(z_i$ chosen as first elem in $Z_{i,j}) = \frac{1}{j - i + 1}$$

and

$$P(z_j$ chosen as first elem in $Z_{i,j}) = \frac{1}{j - i + 1}$$

• Hence

$$P(z_i$ or $z_j$ are first elem in $Z_{i,j}$ chosen as pivots$) = \frac{2}{j - i + 1}$$

Conclusion

$E(X_{i,j}) = P(z_i$ is compared to $z_j)$

$$E(X_{i,j}) = P(z_i$ is compared to $z_j) = \frac{2}{j - i + 1}$$

Putting it together

$$E(X) = E(\sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{i,j})$$

$$= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} E(X_{i,j})$$

$$= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j - i + 1}$$

$$< \sum_{i=1}^{n-1} \sum_{k=1}^{n-i} 2 \frac{1}{k+1}$$

$$< \sum_{i=1}^{n-1} \sum_{k=1}^{n} \frac{2}{k}$$

$$= \sum_{i=1}^{n-1} O(\log n)$$

$$= O(n \log n)$$
Questions

• Q: Why is $\sum_{k=1}^{n} \frac{2}{k} = O(\log n)$?
• A:

\[
\sum_{k=1}^{n} \frac{2}{k} = 2 \sum_{k=1}^{n} \frac{1}{k} \leq 2(\ln n + 1) \quad (11)
\]

Where the last step follows by an integral bound on the sum (p. 1067)

Take Away

• The expected number of comparisons for r-quicksort is $O(n \log n)$
• Competitive with mergesort and heapsort
• Randomized version is “better” than deterministic version

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).
Example

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

```
          a1 <= a2?
         /   \
        yes   no
     (a1,a2) (a2,a1)
```

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.

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

- Claim: If the input numbers are distributed uniformly over the range $[0,1)$, then Bucket sort takes expected time $O(n)$.
- 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 $T(n) = \Theta(n) + \sum_{i=0}^{n-1} O(n_i^2)$. 

• We know \( T(n) = \Theta(n) + \sum_{i=0}^{n-1} O(n_i^2) \)
• Taking expectation of both sides, we have
\[
E(T(n)) = E(\Theta(n) + \sum_{i=0}^{n-1} O(n_i^2))
\]
\[
= \Theta(n) + \sum_{i=0}^{n-1} E(O(n_i^2))
\]
• 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 book)

We can evaluate the two summations separately. \( X_{ij} \) is 1 with probability \( 1/n \) and 0 otherwise
• Thus \( E(X_{ij}^2) = 1 \times (1/n) + 0 \times (1 - 1/n) = 1/n \)
• 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
\[
E(X_{ij}X_{ik}) = E(X_{ij})E(X_{ik})
\]
\[
= (1/n)(1/n)
\]
\[
= (1/n^2)
\]
Analysis

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

\[ E(n^2_i) = \sum_{j=1}^{n} (1/n) + \sum_{1\leq j \leq n} \sum_{1\leq k \leq n, k \neq j} (1/n^2) \]

\[ = n(1/n) + (n)(n - 1)(1/n^2) \]

\[ = 1 + (n - 1)/n \]

\[ = 2 - (1/n) \]

Analysis

• Recall that \( E(T(n)) = \Theta(n) + \sum_{i=0}^{n-1} (O(E(n^2_i))) \)
• We can now plug in the equation \( E(n^2_i) = 2 - (1/n) \) 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

Dictionary ADT

A dictionary ADT implements the following operations

• \textit{Insert}(x): puts the item x into the dictionary
• \textit{Delete}(x): deletes the item x from the dictionary
• \textit{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:
  - \textit{Insert}(k,r): puts the item \((k,r)\) into the dictionary if the key k is not already there, otherwise returns an error
  - \textit{Delete}(k): deletes the item with key k from the dictionary
  - \textit{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 \( l \)
  - \( \text{head}(l) \): returns a pointer to the head of the list
  - \( \text{next}(p) \): given a pointer \( p \) into the list, returns a pointer to the next element in the list if such exists, null otherwise
  - \( \text{previous}(p) \): given a pointer \( p \) into the list, returns a pointer to the previous element in the list if such exists, null otherwise
  - \( \text{key}(p) \): given a pointer into the list, returns the key value of that item
  - \( \text{record}(p) \): given a pointer into the list, returns the record value of that item

At-Home Exercise

Implement a dictionary with a linked list

- Q1: Write the operation \( \text{Lookup}(k) \) which returns a pointer to the item with key \( k \) if it is in the dictionary or null otherwise
- Q2: Write the operation \( \text{Insert}(k,r) \)
- Q3: Write the operation \( \text{Delete}(k) \)
- 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

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

• 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 $k$

Direct Address Functions

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

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

Direct Addressing Problem

• 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