CS 561, Randomized Algorithms

Jared Saia
University of New Mexico
Quicksort

- Based on divide and conquer strategy
- Worst case is $\Theta(n^2)$
- Expected running time is $\Theta(n \log n)$
- An In-place sorting algorithm
- Almost always the fastest sorting algorithm
Quicksort

- **Divide:** Pick some element $A[q]$ of the array $A$ and partition $A$ into two arrays $A_1$ and $A_2$ such that every element in $A_1$ is $\leq A[q]$, and every element in $A_2$ is $> A[p]$
- **Conquer:** Recursively sort $A_1$ and $A_2$
- **Combine:** $A_1$ concatenated with $A[q]$ concatenated with $A_2$ is now the sorted version of $A$
The Algorithm

//PRE: A is the array to be sorted, p>=1;
//  r is <= the size of A
//POST: A[p..r] is in sorted order
Quicksort (A,p,r){
  if (p<r){
    q = Partition (A,p,r);
    Quicksort (A,p,q-1);
    Quicksort (A,q+1,r);
  }
}
Partition

//PRE: A[p..r] is the array to be partitioned, p>=1 and r <= size of A, A[r] is the pivot element
//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[r], A’[res] = A[r],
// and all elements in A’[res+1..r] are > A[r]
Partition (A,p,r){
    x = A[r];
    i = p-1;
    for (j=p;j<=r-1;j++){
        if (A[j]<=x){
            i++;
            exchange A[i] and A[j];
        }
    }
    exchange A[i+1] and A[r];
    return i+1;
}
Correctness

Basic idea: The array is partitioned into four regions, $x$ is the pivot

- Region 1: Region that is less than or equal to $x$ (between $p$ and $i$)
- Region 2: Region that is greater than $x$ (between $i + 1$ and $j - 1$)
- Region 3: Unprocessed region (between $j$ and $r - 1$)
- Region 4: Region that contains $x$ only ($r$)

Region 1 and 2 are growing and Region 3 is shrinking
Loop Invariant

At the beginning of each iteration of the for loop, for any index $k$:

1. If $p \leq k \leq i$ then $A[k] \leq x$
2. If $i + 1 \leq k \leq j - 1$ then $A[k] > x$
3. If $k = r$ then $A[k] = x$
Example

- Consider the array (2 6 4 1 5 3)
At-Home Exercise (Soln on p. 147)

• Show this invariant holds before the loop begins (Initialization)
• Show if the invariant holds after the \( i - 1 \)-th iteration, that it will hold after the \( i \)-th iteration (Maintenance)
• Show that if the invariant holds when the loop exits, that the array will be successfully partitioned (Termination)
Analysis

- The function Partition takes $O(n)$ time. Why?
Randomized Quick-Sort

- We’d like to ensure that we get reasonably good splits reasonably quickly
- Q: How do we ensure that we “usually” get good splits? How can we ensure this even for worst case inputs?
- A: We use randomization.
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-Quicksort is a *randomized* algorithm
- The run time is a *random variable*
- We’d like to analyze the *expected* run time of R-Quicksort
- To do this, we first need to learn some basic probability theory.
Probability Definitions

(from Appendix C.3)

- A *random variable* is a variable that takes on one of several values, each with some probability. (Example: if $X$ is the outcome of the roll of a die, $X$ is a random variable)
- The *expected value* of a random variable, $X$ is defined as:

$$ E(X) = \sum_x x \cdot P(X = x) $$

(Example if $X$ is the outcome of the roll of a three sided die,

$$ E(X) = 1 \cdot (1/3) + 2 \cdot (1/3) + 3 \cdot (1/3) $$

$$ = 2 $$
Probability Definitions

- Two events $A$ and $B$ are *mutually exclusive* if $A \cap B$ is the empty set (Example: $A$ is the event that the outcome of a die is 1 and $B$ is the event that the outcome of a die is 2)
- Two random variables $X$ and $Y$ are *independent* if for all $x$ and $y$, $P(X = x \text{ and } Y = y) = P(X = x)P(Y = y)$ (Example: let $X$ be the outcome of the first roll of a die, and $Y$ be the outcome of the second roll of the die. Then $X$ and $Y$ are independent.)
Probability Definitions

- An *Indicator Random Variable* associated with event $A$ is defined as:
  - $I(A) = 1$ if $A$ occurs
  - $I(A) = 0$ if $A$ does not occur
- Example: Let $A$ be the event that the roll of a die comes up 2. Then $I(A)$ is 1 if the die comes up 2 and 0 otherwise.
Linearity of Expectation

- Let $X$ and $Y$ be two random variables
- Then $E(X + Y) = E(X) + E(Y)$
- (Holds even if $X$ and $Y$ are not independent.)

- More generally, let $X_1, X_2, \ldots, X_n$ be $n$ random variables
- Then

$$E\left(\sum_{i=1}^{n} X_i\right) = \sum_{i=1}^{n} E(X_i)$$
Example

- For $1 \leq i \leq n$, let $X_i$ be the outcome of the $i$-th roll of a three-sided die.
- Then

$$E\left( \sum_{i=1}^{n} X_i \right) = \sum_{i=1}^{n} E(X_i) = 2n$$
Example

- Indicator Random Variables and Linearity of Expectation used together are a very powerful tool
- The “Birthday Paradox” illustrates this point
- To analyze the run time of quicksort, we will also use indicator r.v.’s and linearity of expectation (analysis will be similar to “birthday paradox” problem)
“Birthday Paradox”

- Assume there are $k$ people in a room, and $n$ days in a year
- Assume that each of these $k$ people is born on a day chosen uniformly at random from the $n$ days
- Q: What is the expected number of pairs of individuals that have the same birthday?
- We can use indicator random variables and linearity of expectation to compute this
Analysis

- For all $1 \leq i < j \leq k$, let $X_{i,j}$ be an indicator random variable defined such that:
  - $X_{i,j} = 1$ if person $i$ and person $j$ have the same birthday
  - $X_{i,j} = 0$ otherwise
- Note that for all $i, j$,

\[
E(X_{i,j}) = P(\text{person } i \text{ and } j \text{ have the same birthday}) = \frac{1}{n}
\]
• Let $X$ be a random variable giving the number of pairs of people with the same birthday
• We want $E(X)$
• Then $X = \sum_{(i,j)} X_{i,j}$
• So $E(X) = E(\sum_{(i,j)} X_{i,j})$
Analysis

\[
E(X) = E\left( \sum_{(i,j)} X_{i,j} \right)
\]
\[
= \sum_{(i,j)} E(X_{i,j})
\]
\[
= \sum_{(i,j)} 1/n
\]
\[
= \binom{k}{2} 1/n
\]
\[
= \frac{k(k - 1)}{2n}
\]

The second step follows by Linearity of Expectation.
Reality Check

- Thus, if $k(k - 1) \geq 2n$, expected number of pairs of people with same birthday is at least 1
- Thus if have at least $\sqrt{2n} + 1$ people in the room, can expect to have at least two with same birthday
- For $n = 365$, if $k = 28$, expected number of pairs with same birthday is 1.04
In-Class Exercise

- Assume there are \( k \) people in a room, and \( n \) days in a year
- Assume that each of these \( k \) people is born on a day chosen uniformly at random from the \( n \) days
- Let \( X \) be the number of groups of three people who all have the same birthday. What is \( E(X) \)?
- Let \( X_{i,j,k} \) be an indicator r.v. which is 1 if people \( i, j, \) and \( k \) have the same birthday and 0 otherwise
In-Class Exercise

• Q1: Write the expected value of $X$ as a function of the $X_{i,j,k}$ (use linearity of expectation)
• Q2: What is $E(X_{i,j,k})$?
• Q3: What is the total number of groups of three people out of $k$?
• Q4: What is $E(X)$?
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-Quicksort
- 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

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

$$P(\text{either } z_i \text{ or } z_j \text{ are the first elems in } Z_{i,j} \text{ chosen as pivots})$$

• 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

\[ P(\text{either } z_i \text{ or } z_j \text{ are first elems in } Z_{i,j} \text{ chosen as pivots}) \]

• A: \( P(z_i \text{ chosen as first elem in } Z_{i,j}) + P(z_j \text{ chosen as first elem in } Z_{i,j}) \)

• Further note that number of elems in \( Z_{i,j} \) is \( j - i + 1 \), so

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

and

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

• Hence

\[ P(z_i \text{ or } z_j \text{ are first elems in } Z_{i,j} \text{ chosen as pivots}) = \frac{2}{j - i + 1} \]
Conclusion

\[ E(X_{i,j}) = P(z_i \text{ is compared to } z_j) \quad (1) \]
\[ = \frac{2}{j - i + 1} \quad (2) \]
Putting it together

\[ E(X) = E \left( \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{i,j} \right) \]  

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

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

\[ = \sum_{i=1}^{n-1} \sum_{k=1}^{n-i} \frac{2}{k + 1} \]  

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

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

\[ = O(n \log n) \]
Questions

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

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

Where the last step follows by an integral bound on the sum (p. 1067)
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.
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 \( h \geq \log(n!) \)
Height of Decision Tree

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

$$\log(n \cdot (n-1) \cdot \cdots \cdot 1) = \log n + \log(n-1) + \cdots + \log 1$$

$$\geq \frac{n}{2} \log(n/2)$$

$$\geq \frac{n}{2}(\log n - \log 2)$$

$$= \Omega(n \log n)$$

- 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 \(T(n) = \Theta(n) + \sum_{i=0}^{n-1} O(n_i^2)\)
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)
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

\[ E(n_i^2) = E\left(\left( \sum_{j=1}^{n} X_{ij} \right)^2 \right) \]

\[ = E\left( \sum_{j=1}^{n} \sum_{k=1}^{n} X_{ij} X_{ik} \right) \]

\[ = E\left( \sum_{j=1}^{n} X_{ij}^2 + \sum_{1 \leq j \leq n} \sum_{1 \leq k \leq n, k \neq j} X_{ij} X_{ik} \right) \]

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

- We can evaluate the two summations separately. $X_{ij}$ is 1 with probability $1/n$ and 0 otherwise.
- Thus $E(X^2_{ij}) = 1 \times (1/n) + 0 \times (1 - 1/n) = 1/n$
- 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:

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

\[
= \sum_{j=1}^{n} \left( \frac{1}{n} \right) + \sum_{1 \leq j \leq n} \sum_{1 \leq k \leq n, k \neq j} \left( \frac{1}{n^2} \right)
\]

\[
= n \left( \frac{1}{n} \right) + (n)(n - 1)(\frac{1}{n^2})
\]

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

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

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