Probability: Binary Search Trees and Quicksort

Xiong-Fei Du

Updated August 31, 2017

1 Introduction

The purpose of this document will be to outline some basic application of probability concepts and their applicability to binary search trees and the quicksort algorithm.

2 Binary Search Trees: Tree Sort

From here we will consider binary search trees satisfying the invariant that all elements in the left subtree is less than the value of the root and all elements in the right subtree is greater than or equal to the value of the root.

Consider a set S such that there exists a comparison operator that is reflexive, symmetric, and transitive among its elements. Then we can create a binary search tree that satisfies the above invariant.

Suppose we consider the tree sort algorithm: for each element in S, we insert that element into a binary search tree T, and then we traverse T to retrieve the elements in sorted order. Pseudocode is provided in the appendix.

The big-O runtime of tree sort is hence entirely dependent on in what order the elements are inserted into T. There are $n!$ different ways to insert the elements of S into T, and the number of unique trees T that can be formed is given by the nth Catalan number, where $n = |S|$, which also intuitively explains why the nth Catalan number is at most $n!$.

The big-O runtime of any sorting algorithm is dependent on the number of comparisons made. Efficient sorting algorthms utilize the transitive property of the comparison operator and run in $O(n \log n)$, while inefficient algorithms do not exploit transitivity and run in $O(n^2)$. In inefficient algorithms, each pair of elements in S must be compared with each other, leading to the total number of comparisons being

$$\binom{n}{2} = \sum_{i=0}^{n-1} i = \frac{n(n-1)}{2} \in O(n^2)$$

2.1 Worst Case: $O(n^2)$

First, let us consider the worst-case runtime of tree sort. Suppose that at every iteration, we insert either the minimum element or the maximum element from the uninserted subset of S into T, and so tree sort essentially devolves into out-of-place selection sort. In this case, every element will be compared to every other element already in the tree T, and inserting that element will have a runtime of $O(n)$, which makes the total runtime in the worst case $O(n^2)$, using the formula outline above. This will create a tree of depth n in one long chain, where each node has one child node and one leaf. The total number of ways we can do this is 2^{n-1} since for each n, we can either choose the minimum or the maximum element in the remaining subset, but when we only have one element left, the minimum element and the maximum element is the same. There are a total of $n!$ different permutations of the order we insert the elements of S into T. This implies that the probability that the worst case will happen is $\frac{2^{n-1}}{n!}$. Hence as n increases, the probability that this will happen given that the element inserted is chosen uniformly at random is very, very low.

2.2 Expected Case: $O(n \log n)$

Now let's consider the expected case. Let's denote A as the sorted sequence of all elements in S, and A_i as the ith element in A, starting from index 0. Then if the elements that are chosen to be inserted into T are chosen uniformly at random, then we want to find the probability that two elements A_i and A_j are compared in order to determine the expected runtime. So let's consider an indicator random variable $X(i, j)$, where $X(i, j) = 1$ if elements A_i and A_j are compared, $X(i, j) = 0$ otherwise. Additionally, let's denote $F(i, j)$ as the probability that A_i and A_j are compared. Without loss of generality, we'll say that $i < j$.

In order to determine this probability $F(i, j)$, let's consider the element that we will pick to insert at the root A_k. k is selected uniformly at random from the range $0 \leq k < n$. Let's consider the following cases of k.

1. $i < k < j$. Then A_i and A_j are put into the left subtree and the right subtree respectively, and so A_i and A_j are never compared, since A_i and A_j are both individually compared with A_k. Then $X(i, j) = 0$. This case happens with probability $\frac{j-i-1}{n}$.
2. $k = i$. Then A_j must be compared to A_i since A_i is selected to be the root, so every other element must be compared to A_i. Then $X(i, j) = 1$. This case happens with probability $\frac{1}{n}$.

3. $k = j$. Then we can say the same as what we said about i in the previous case. $X(i, j) = 1$. This case happens with probability $\frac{1}{n}$.

4. $k < i$ or $k > j$. Then both A_i and A_j are inserted into the same subtree. Then we recursively look at the above three cases again. This case happens with probability $\frac{n-j+i-1}{n} = 1 - \frac{j-i+1}{n}$.

Then considering these cases, we can use the infinite geometric sum formula $S = \frac{a}{1-r}$ to compute that the total probability that A_i and A_j are compared is $F(i, j) = \frac{2}{j-i+1}$. This analysis also works if we simply ignore case 4 and consider the outcome space to be the first three cases, which have a total of $j - i + 1$ total indices, of which we only compare A_i and A_j if 2 of the indices are chosen, which implies $F(i, j) = \frac{2}{j-i+1}$. Using this, we can say that

$$E(X(i,j)) = \sum_{x \in \{0,1\}} xP(X = x) = F(i,j) = \frac{2}{j - i + 1}$$

Now that we've established this, let's calculate the expected number of comparisons we would make, which we will denote as Y. By how we've defined our random variables, we can say that

$$Y = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} X(i,j)$$

and hence

$$\begin{aligned} E(Y) &= E\left(\sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} X(i,j)\right) \\ &= \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} E(X(i,j)) \\ &= \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} \frac{2}{j-i+1} \\ &= \sum_{i=0}^{n-2} \sum_{j=2}^{n-i} \frac{2}{j} \\ &= 2\sum_{i=0}^{n-2} \sum_{j=2}^{n-i} \frac{1}{j} \end{aligned}$$

Now consider the upper limit of this expression.

$$
\begin{aligned}
E(Y) &= 2\sum_{i=0}^{n-2}\sum_{j=2}^{n-i}\frac{1}{j} \\
&\leq 2\sum_{i=0}^{n-2}\sum_{j=2}^{n}\frac{1}{j} \\
&= 2(n-1)\sum_{j=2}^{n}\frac{1}{j}
\end{aligned}
$$

Now consider the expression $\sum\limits_{j=2}^{n}\frac{1}{j}$, which is equivalent to $\frac{1}{2}+\frac{1}{3}+\frac{1}{4}+...+\frac{1}{n}$. This expression is just $H_n - 1$, where H_n is the nth Harmonic number. So now we have

$$E(Y) \leq 2(n-1)(H_n - 1)$$

We can also note that the expression $\sum\limits_{j=2}^{n}\frac{1}{j} \approx \int\limits_{2}^{n}\frac{1}{x}dx = \ln n - \ln 2 \leq \ln n$. Mathematically, we know that $\lim\limits_{x\to\infty}(H_n - \ln n) = \gamma$, where γ is the Euler-Mascheroni constant 0.57721566..., so we can say that $H_n \in O(\log n)$. Therefore, we can now say that

$$E(Y) \leq 2n\ln n$$

Or in big-O notation:

$$E(Y) \in O(n\log n)$$

Hence, we can conclude that the big-O bound of the expected runtime of tree sort is $O(n\log n)$.

3 Quicksort: In-place tree sort

Consider the quicksort algorithm, where we randomly select a pivot from a sequence S, recursively sort all elements smaller than the pivot and all elements greater than or equal to the pivot, and then join them together. The pseudocode for this algorithm is given in the appendix.

Conceptually, we can think of quicksort as simply in-place tree sort. Selecting and removing a pivot from a sequence is the same as selecting an element to be inserted into the tree as a root. Hence, the big-O bounds of quicksort is the same as that of tree sort, and the probability that quicksort will devolve into selection sort is the same probability that a tree will end up as a long chain. To summarize, quicksort has worst case runtime $O(n^2)$ and expected case runtime $O(n \log n)$. The probability that quicksort on n elements will devolve into selection sort is $\frac{2^{n-1}}{n!}$.

4 Appendix

4.1 Tree Sort: Pseudocode

The following pseudocode is for a version of out-of-place quicksort. Assume that there exists a function `join` that takes in a sequence, an element, and a sequence and forms a new sequence with the element sandwiched in the middle of the two sequences.

```
datatype bst = Leaf | Node(bst, elem, bst)

initbst() =
    return Leaf

insert(T, x) =
    if T = Leaf
        return Node(Leaf, x, Leaf)
    else
        let T = Node(L, y, R)
        if x < y
            return Node(insert(L, x), y, R)
        else
            return Node(L, y, insert(y, R))

traverse(T) =
    if T = Leaf
        return empty
    else
        let T = Node(L, x, R)
        return join(traverse(L), x, traverse(R))

treesort(A) =
    T = init_bst()
    for x in A
        insert(T, x)
    return traverse(T)
```

4.2 Quicksort: Pseudocode

The following pseudocode implements in-place quicksort. Note that there is no return statement, since the sequence is destructively modified in place.

```
quicksort(A, lo, hi) =
    if (hi - lo) > 1
        i = random_choose(lo, hi)
        p = partition(A, lo, i, hi)
        quicksort(A, lo, p)
        quicksort(A, p + 1, hi)

partition(A, lo, p, hi) =
    hi--
    upper = hi
    pivot = A[p]
    A[p] = A[upper]
    while (lo < hi)
        if A[lo] < pivot
            lo++
        else
            hi--
            temp = A[lo]
            A[lo] = A[hi]
            A[hi] = temp
    A[upper] = A[lo]
    A[lo] = pivot
    return lo
```

The following pseudocode implements out-of-place quicksort, which is much easier to conceptualize. Assume that there exists a `random_remove` function that takes in a sequence and removes an element while destructively modifying that sequence.

```
quicksort(A) =
    if length(A) = 0
        return A
    else
        pivot = random_remove(A)
        L = <x for x in A : x < pivot>
        R = <x for x in A : x >= pivot>
        return join(quicksort(L), pivot, quicksort(R))
```