Analyzing Quicksort

Scribe by Anusheel Bhushan

February 23, 2003

1 Introduction

In this lecture, we will look at the Quicksort Algorithm more closely.

2 A Variation

Let us first look at the following variation of Quicksort:

Input: $S = \{a_1, a_2 \ldots a_n\}$ to be sorted.

Step 1: Pick a random element a_r.

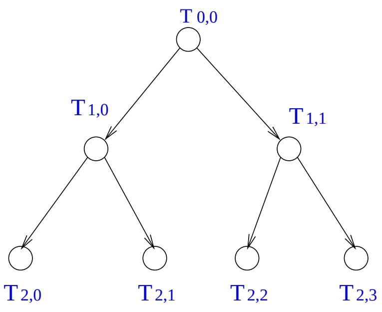
Step 2: Partition the set S about a_r into T_{left} and T_{right}.

If the size of either T_{left} or T_{right} is $> 3|S|/4$ (*not a balanced partition*), then goto Step 1.

Step 3: Recursively call on both partitions.

Clearly, the expected number of trials to get a *balanced splitter* is 2. Also, after every successful partition, we reduce the size of the recursive call by atleast $1/4^{th}$.

If we consider the tree induced by the partitions,



Since each subsequent sub-tree in a path will have atmost $3/4^{th}$ the size of the previous, the height of the tree is atmost $\log_{\frac{4}{3}} n$.

Moreover, the amount of work done at level j = the number of elements in all the sub-trees at this level, which is atmost $O(n)$.

The total running time is thus atmost $O(n \log_{\frac{4}{3}} n)$.

Formally, let $T_{i,j}$ be the running time to partition the sub-tree $\pi_{i,j}$.

$$\begin{aligned} \text{Then, expected running time} &= \sum_j \sum_i E(T_{i,j}). \\ &= \sum_j (\sum_i 2|\pi_{i,j}|). \\ &\leq \sum_j 2n = O(n \log n) \end{aligned}$$

Now consider an algorithm based on the k^{th} Selection Algorithm done earlier. We use the Selection Algorithm to select the median element in $O(n)$ with a failure probability $1 - \frac{1}{n}$. Now, we partition the set about this element and recursively call sort on the partitions.

However, there is a catch in this scheme. As the size of the set(i.e n) goes down, the failure probability increases. Thus the overall failure probability will not be small.

3 Analyzing Quicksort

Now, let us get back to the good old Quicksort.

Input: $S = \{a_1, a_2 \ldots a_n\}$ to be sorted.

Step 1: Pick a random element a_r (called the *splitter*).

Step 2: Partition the set S about a_r into T_{left} and T_{right}.

Step 3: Recursively call Quicksort on both partitions.

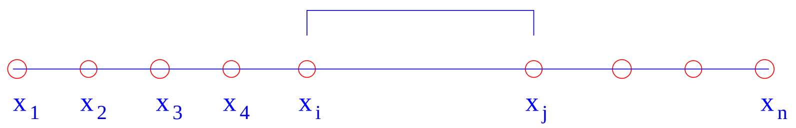
Observation: Each element becomes a splitter once and only once.

Let us generate a random permutation of the input elements and chose the splitters in this order.

We denote the finally sorted elements by $\{x_1, x_2 \ldots x_n\}$

Now, let

$$Z_{i,j} = \begin{cases} 1 & \text{if } x_i \text{ is compared with } x_j \\ 0 & \text{else} \end{cases}$$



Now, x_i and x_j will be compared iff either x_i or x_j becomes a splitter before any other element in between them. Once an intermediate element becomes a splitter, x_i and x_j will go into different partitions.

In other words, the Prob that x_i and x_j are compared

$$= \frac{\text{no. of permutations where either } x_i \text{ or } x_j \text{ appears before the elements between them}}{\text{total no. of permutations}}$$

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

$$\begin{aligned}\text{Then, the running time T} &= \sum_i \sum_j Z_{i,j}. \\ &= \sum_i \sum_{j>i} \frac{2}{j-i+1}. \\ &\leq 2nH_n = O(n \log n)\end{aligned}$$

Note that the random variables $Z_{i,j}$ are not independent. Thus, we can't apply Hoeffding's inequality to bound the expectation. Let us count the comparisons slightly differently.

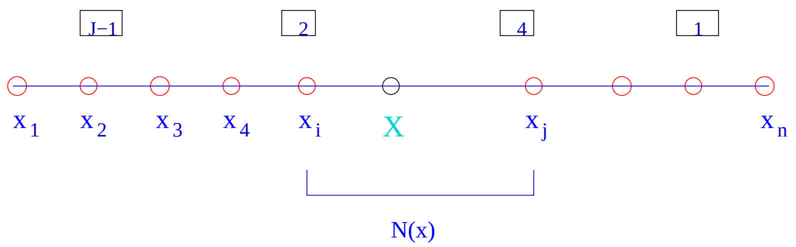
4 Backward Analysis

Let

$$Z_{i,j} = \begin{cases} 1 & \text{if } x_i \text{ is compared with the } j^{th} \text{ splitter} \\ 0 & \text{else} \end{cases}$$

Let us compute the probability of some element X being compared with the j^{th} splitter. We already have a random permutation of the initial elements to decide the order of choosing the splitters. Now, let us view quicksort as an iterative procedure wherein we add the splitters one by one.

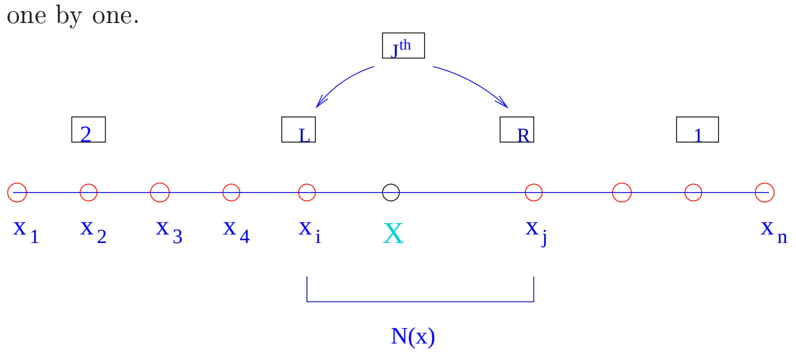
Observation: X may be compared with the j^{th} splitter only in the j^{th} step. The numbers in the box represent the order in which the splitters are chosen.



For example, in the above figure, X lies between the 2^{nd} and the 4^{th} splitter. Thus, when we add the j^{th} splitter, it will be compared with X only if it lies in the interval $N(X)$. Thus, the probability that X is compared to the j^{th} splitter will be $\frac{N(X)}{N}$ which is again a random variable.

To avoid all this, we use the powerful technique of *Backward analysis.*

Let us view the iteration in reverse, wherein we take out the splitter elements one by one.



We want the probability that X is compared with the j^{th} splitter.

$$
\begin{aligned}
&= \text{Prob that the interval N(X) containing X is modified on removing the } j^{th} \text{ splitter} \\
&= \text{Prob that the } j^{th} \text{ splitter is either } L \text{ or } R.
\end{aligned}
$$

Since, out of the remaining j splitters, each splitter is equally likely to be the j^{th} splitter, this Prob is $\frac{2}{j}$.

$$
\begin{aligned}
\text{Then, expected running time} &= \sum_i \sum_j E(Z_{i,j}). \\
&= \sum_i \sum_j \frac{2}{j}. \\
&= \sum_i H_n = nH_n = O(nlogn)
\end{aligned}
$$

Let us analyze the technique of backward analysis more closely.
We are adding elements $\{1, 2, \ldots n\}$ in a random permutation and we update a data structure D^j, which is the state of the data structure after j insertions.

We want to calculate the cost of updating D^j to D^{j+1} as the sum of expected costs incurred by each element (ie by charging it to the individual elements).

In case of Quicksort, we want to find out the Prob that the j^{th} splitter falls in the same partition as x_i, induced by the previous $j-1$ splitters. Let $\{y_i, y_2, \ldots y_j\}$ be a "fixed" set of elements and x_i is in the interval $[y_{j_1}, y_{j_2}]$.

$$
\begin{aligned}
Pr[Z_{i,j} = 1/_{\{y_i, y_2, \ldots y_j\}}] &= \text{Pr}[y_{j_1} \text{ or } y_{j_2} \text{ was the last element added}] \\
&= \frac{2}{j} \\
&\quad \text{(as in a random permutation of } y_1 \ldots y_j \text{ each of} \\
&\quad \text{which is equally likely to be the last element added)}
\end{aligned}
$$

Then,

$$
\begin{aligned}
Pr[Z_{i,j} = 1] &= \sum_{y_i, y_2, \ldots y_j \in \{1 \ldots n\}} Pr[Z_{i,j} = 1/_{\{y_i, y_2, \ldots y_j\}}] \; Pr[\{y_i, y_2, \ldots y_j\} \text{ are the first } j \text{ elts.}] \\
&= \frac{2}{j} \sum_{\text{all choices}} Pr[\{y_i, y_2, \ldots y_j\} \text{ are the first } j \text{ elts.}] = \frac{2}{j}
\end{aligned}
$$

Since the $Z_{i,j}$'s are all independent, we can now apply Chernoff-Hoeffding inequalities.

5 TREAPS

We describe a randomized binary search tree with the heap property.

EX : How many binary search trees can we make (for a given sorted sequence).

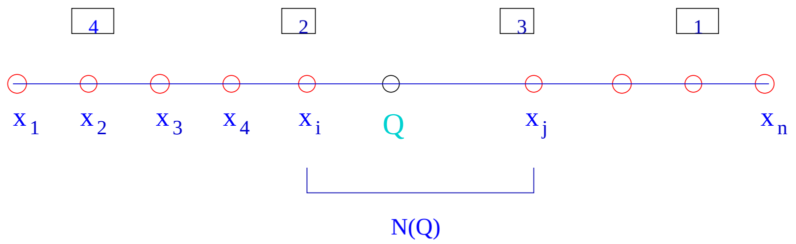
We are given a set of elements along with their associated priorities $\in \{0, 1\}$. We would like to maintain the property that the priorities should be non-decreasing along any root $\rightarrow$ leaf path.

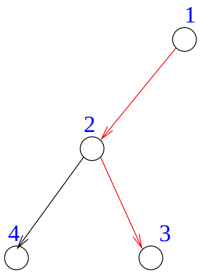
Observation : If we sort the priorities and then insert like a binary tree, we'll get a unique binary tree.

The priorities are independently and uniformly distributed random variables between 0 and 1. We generate the priorities online, ie we generate the priority for an element whenever we want to insert it. Since the final tree is unique, the tree we get is same as the tree built using all the priorities initially.

Let us fix a query element Q. We would like that the search path of Q is "not too long". We pretend to insert the n elements at random (following the random priorities).

We denote the elements obtained finally by an inorder traversal by $\{x_1, x_2 \ldots x_n\}$. The numbers in the square box represent the order in which the elements are added.





When we add the 4^{th} element x_2, it doesn't affect the search time of Q since it lies in a different subtree from the search path (shown in red). Thus, the search path of Q is determined by the number of times the bounding interval of Q is modified. Thus we can resort to an analysis similar to to that used for quicksort.

The expected number of times the left/right boundary of the interval containing Q changes as we insert the tree elements in a random order

$$= \sum_j \frac{2}{j} = 2H_n = O(\log \text{n}) \qquad \text{(similar to the quicksort analysis)}$$

EX : How to actually do insertions and deletions in the treap?

Hint : To insert, we start with a priority of $-\infty$, and propagate the element upwards using swaps and rotations.

EX : Show that the expected number of rotations to insert an element is 2.

The above analysis was for a single Q. If we have a set of m keys, we would like to find the search time for searching any key. For n elements, there are $n+1$ intervals having the same search path. Thus, the search time for any key is atmost n times that for a single key.