CS 330 Discussion - Randomized Quicksort, Collision Handing

March 31 2017

1 Randomized Quicksort Alternate Analysis

In lecture, we showed that randomized quicksort runs in $O(n\ log\ n)$ time in expectation. However, this statement alone suggests it is possible that it may run in time greater than $O(n\ log\ n)$ with non-trivial probability. In this discussion, we will present an alternative runtime analysis which shows that it runs in time $O(n\ log\ n)$ with high probability.

To use this proof, we will need to use an inequality called the Chernoff bound, which we state here without proof:

Theorem 1. ***Chernoff Bound*** *- Let X be some random variable which can be written as $X = \sum_i X_i$, where each X_i only takes value 0 or 1. Let μ be the expected value of X. Then, $Pr[X < (1-\epsilon)\mu] \leq e^{-\frac{\epsilon^2\mu}{3}}$*

We will use an amortized analysis where we charge 1 to each element s for each subproblem it participates in. In other words, s is charged the number of subproblems it participates in before it is chosen as a pivot.

Let $h(s)$ be the number of subproblems s participates in. Then an upper bound for the runtime of the algorithm is $n \max_s h(s)$.

We will show that $h(s) \in O(logn)$ for all s with high probability. To do this, we need to show that the recursive subsequences chosen are often roughly balanced in length.

Let a "good" subproblem involving s be one where if we partition the sequence S into $S_1, S \setminus S_1$, $\frac{1}{4}|S| \leq |S_1| \leq \frac{3}{4}|S|$ (if this is true for S_1, it is also true for $S \setminus S_1$).

First, we bound the number of good subproblems any element can participate in:

Lemma 1. *Any element s can only participate in $O(log\ n)$ good subproblems.*

Proof. s is initially part of the original problem with n elements. No subproblem is smaller than size 1. Each time s participates in a good subproblem involving sequence S, the next subproblem it participates in is of size at most $\frac{3}{4}|S|$. Thus, if s has participated in $log_{4/3}\ n$ subproblems, the size of the next subproblem it participates in cannot be more than 1. This proves the lemma. □

Let $X_{s,i}$ be an indicator variable of the event that the ith subproblem s participates in is a good subproblem. That is $X_{s,i} = 1$ if the ith subproblem s participates in is good and 0 otherwise. Note that by definition of good subproblems, $X_{s,i}$ is 1 with probability $\frac{1}{2}$.

Let $X_s(N)$ be the number of good subproblems in the first N subproblems s participates in. Note that $X_s(N) = \sum_{i=1}^{N} X_{s,i}$. Then $E[X_s(N)] = \frac{N}{2}$ for any N.

Then, let us apply the Chernoff bound to $X_s(N)$. We get that:

$$Pr[X_s(N) < (1-\epsilon)\frac{N}{2}] \leq e^{-\frac{-\epsilon^2 N}{6}}$$

Since s is done participating in subproblems after it participates in $\Omega(log\ n)$ good subproblems, we want to bound the number of subproblems it takes to see $\Omega(log\ n)$ good subproblems. Then, let us choose $N = 2C\ log\ n$.

$$Pr[X_s(2C\ log\ n) < (1-\epsilon)C\ log\ n] \leq n^{-\frac{-\epsilon^2 C}{3}}$$

If we can show a single element participates in more than $O(log\ n)$ subproblems with probability $\frac{1}{n^2}$, we can then show that no element participates in $\omega(log\ n)$ subproblems with high probability. In order for the right side of the inequality to be $\frac{1}{n^2}$, we choose $\epsilon = \sqrt{\frac{6}{C}}$.

$$Pr[X_s(2C\ log\ n) < (1-\sqrt{\frac{6}{C}})C\ log\ n] \leq n^{-2}$$

.

In order for this to be a meaningful bound, we just choose some $C > 6$.

Now, this statement says that if s has participated in $2C\ log\ n$ subproblems, it has participated in roughly $C\ log\ n$ good subproblems with high probability. If C is large enough then by the above Lemma, it cannot participate in any more subproblems. Thus, s participates in $\omega(log\ n)$ subproblems with probability $\frac{1}{n^2}$.

Then, the probability that some s participates in $\omega(log\ n)$ subproblems is at most $\frac{1}{n}$ by union bound. This means that all s participate in $O(log\ n$ subproblems with probability at least $1 - \frac{1}{n}$, i.e. that randomized quicksort finishes in $O(n\ log\ n)$ time with high probability.

2 Collision Handling

We now move on to discussing hashing collisions. If a perfect hash code existed, no two elements would ever hash to the same value in a hash table (i.e. collide) , and hashing would be an $O(1)$ time operation. However, hashing is not perfect in practice, and collisions do occur, thus some way of dealing with collisions (collision handling) must be implemented.

There are multiple ways to handle collisions. Here, we discuss their benefits and tradeoffs. For all collision handling methods, the process is the same as lookups - for handling a collision, we move to new locations to store the value

until some empty location to store it is found. For a lookup of some value, we start with the position the value would be if no collision occurred, and search use the same sequence of locations the collision handling method uses until the element is found or we encounter an empty location. Thus, all collision handling methods are analagous to table lookup methods for themselves.

2.1 Separate chaining

Separate chaining refers to collision handling methods where each bucket remains independent, but can hold multiple values. To do this, rather than have each bucket hold a single value, we'll have each bucket hold a data structure which can hold multiple elements should a collision occur. Separate chaining is effective when said data structure runs efficiently for the number of expected collisions per bucket - e.g. if we expect a low number of collisions per bucket, a data structure which maybe does not scale well but operates very quickly on an architectural level is desirable. For a large number of collisions, a data structure which scales well is desirable, even if it runs poorly for a few elements. In addition, on an architectural level, it might involve referring to data structures which are not near the hashtable in memory, which could be slow to access.

2.1.1 Separate chaining with linked lists

In separate chaining with linked lists, each bucket contains a pointer to the head of a linked list of values, or alternative is the head of such a linked list. Then, if a collision occurs, we append the value to the end of the linked list in the bucket. This implementation is very simply to use, and often taught and implemented for that reason. However, if a large number of collisions occur, it is slowed very easily.

2.1.2 Separate chaining with self-balancing trees

Separate chaining with self-balancing trees is similar to separate chaining with linked lists, except we use a self-balancing ordered tree instead of a linked list. The benefit is that the runtime of an add or lookup scales logarithmically with the number of collisions, rather than linearly. However, it requires more memory to be implemented.

2.2 Open addressing

Open addressing could be called the opposite of separate chaining. In separate chaining, buckets have independence and store multiple values. In open addressing, buckets store single values and one bucket having many values hashed to it may affect other buckets. In open addressing, if a collision occurs in one bucket, rather than have each bucket hold a variable amount of space we can search for empty space in, we instead search through other buckets in a deterministic fashion and place our colliding value into the first empty bucket we find. Open

addressing is very easy to implement and on an architectural level tends to perform better than separate chaining because all the values are stored in memory locations which are near each other. However, it may slow down considerably if many of the buckets in the hashtable already have values in them, and is, and also will do poorly if the hash code does not actually hash elements uniformly.

2.2.1 Linear probing

In linear probing, when a collision occurs, we move on to a bucket a fixed number of spaces away from the original bucket (this is usually 1 for simplicity). If this bucket also already has a value, we repeat this process until we find an empty bucket. Linear probing has all the benefits of open addressing, but struggles if a hash code not only maps values to the same bucket frequently, but also tends to hash elements to nearby buckets. In this case, the collisions of buckets nearby each other compound their runtimes in the case of a collision. For example, if we often hash to buckets 100 and 101, collisions in these buckets would place colliding values into the same set of buckets, meaning many collisions in bucket 100 would slow down collisions and lookups for bucket 101, even if it has few values hashing to it. To avoid this, we can instead do quadratic probing

2.2.2 Quadratic probing

In linear probing, if k values collided in bucket H, then the sequence of buckets we used was $H+1, H+2, H+3...H+k$. In quadratic probing, rather than use a constant increment and linear offsets, we'll use quadratic offsets. That is, if k values collide in bucket H, they would go in buckets $H+1, H+4, ...H+k^2$. This adds basically nothing in terms of extra computation, and greatly avoids the issues of linear probing because the intersection between the set of buckets two buckets will use in the case of a collision is small for any pair of buckets.

2.2.3 Double hashing

In double hashing, we have two hash functions h_1, h_2, the second of which takes non-zero values, and whose values ideally are not correlated. When we try to add a value s to the hash table, we'll first try to add it to bucket $h_1(s)$. If this bucket is already in use, we'll try $h_1(s) + h_2(s)$ instead. If this also fails, we can then choose to start storing multiple values in one bucket using separate chaining, or use more than 2 hash functions. Double hashing is effective in that there's even less overlap in the space used by any pair of buckets, but requires the computation of an extra hash function or functions, as opposed to simpler calculations like in linear or quadratic probing.

2.2.4 2-choice hashing

2-choice hasing is a combination of the ideologies behind separate chaining and open addressing. In 2-choice hashing, we have two hash functions. For each value we hash, each hash function suggests a bucket to place the value in, and

we choose whichever suggested bucket has less values in it. This is useful for keeping bucket sizes relatively even, but suffers from extra computational load per collision.