Applying a permutation to a vector, part 4: What is the computational complexity of the apply_permutation function?

devblogs.microsoft.com/oldnewthing/20170109-00

January 9, 2017

Raymond Chen

One question left unanswered was the computational complexity of the `apply_permutation` function.

Here it is again:

```
template<typename T>
void
apply_permutation(
    std::vector<T>& v,
    std::vector<int>& indices)
{
 using std::swap; // to permit Koenig lookup
 for (size_t i = 0; i < indices.size(); i++) {
  auto current = i;
  while (i != indices[current]) {
   auto next = indices[current];
   swap(v[current], v[next]);
   indices[current] = current;
   current = next;
  }
  indices[current] = current;
 }
}
```

The outer `for` loop runs N times; that's easy to see. The maximum number of iterations of the inner `while` loop is a bit less obvious, but if you understood the discussion, you'll see that it runs at most N times because that's the maximum length of a cycle in the permutation. (Of course, this analysis requires that the `indices` be a permutation of $0 \ldots N - 1$.)

Therefore, a naïve analysis would conclude that this has worst-case running time of $O(N^2)$ because the outer `for` loop runs N times, and the inner `while` loop also runs N times in the worst case.

But it's not actually that bad. The complexity is only *O*(*N*), because not all of the worst-case scenarios can exist simultaneously.

One way to notice this is to observe that each item moves only once, namely to its final position. Once an item is in the correct position, it never moves again. In terms of indices, observe that each swap corresponds to an assignment `indices[current] = current`. Therefore, each entry in the index array gets set to its final value. And the `while` loop doesn't iterate at all if `indices[current] == current`, so when we revisit an element that has already moved to its final location, we do nothing.

Since each item moves at most only once, and there are *N* items, then the total number of iterations of the inner `while` loop is at most *N*.

Another way of looking at this is that the `while` loop walks through every cycle. But mathematics tell us that permutations decompose into disjoint cycles, so the number of elements involved in the cycles cannot exceed the total number of elements.

Either way, the conclusion is that there are most *N* iterations of the inner `while` loop in total. Combine this with the fixed overhead of *N* iterations of the `for` loop, and you see that the total running time complexity is *O*(*N*).

(We can determine via inspection that the algorithm consumes constant additional space.)

Raymond Chen

Follow