Quicksort And Liner Time Sorts - Recitation 6

Quicksort, Countingsort, Radixsort And Bucketsort.

November 19, 2022

Till now we have quantiffied the algorithm permformance against the worst case scenerio. And we saw that accorrding to that masure, In the comparisons model, one can not sorting in less then $\Theta(n \log n)$ time. In this recation we present a two new main concepts that, in certain cases, achive batter anlysies. The first one is the Excpection Complexitiy, By Letting the algorithm to behave undetrmistcly, we might obtain an algortihm that most of the time run compute the task fast. Yet we will not secussed get down the $\Theta(n \log n)$ lower bound, but we will back to use that concept in the pending of the course. The seconed concept is to ristrict ourselves to deal only in particular type of inputs. For example We will see that if we suppose that the a given array contains only integer in bounded domain then we can sort it in linear time.

0.1 Quicksort.

The quicksort algorithm is a good example for a **non-determistic** algorithm that has a worst-case running time of $\Theta(n^2)$. Yet its expected running time is $\Theta(n \log n)$. Namley fix an a array of n numbers, the ruunings of Quicksort over that array might be diffrent, each of them is an diffrent event in probability space, and the rununing time of the algorithm is a random variabile defiend over that space. Saying that the algorithm has worst space complexitiy of $\Theta(n^2)$ means that there exist event in which it runs $\Theta(n^2)$ time with non-zero probability. But parcticaly the interesting question is not the existance of such event but how likily that it happend. It turns out that expection of the runnning time is acctuly $\Theta(n \log n)$.

What is the exactly reason that happens? By giving up on the algoirthm behaveoir centertiy we are going to turn the task of eneginering bad input impossible.

Our study of quicksort is broken into four sections. Section 7.1 describes the algorithm and an important subroutine used by quicksort for partitioning. Because the behavior of quicksort is complex, we'll start with an intuitive discussion of its performance in Section 7.2 and analyze it precisely at the end of the chapter. Section 7.3 presents a randomized version of quicksort. When all elements are distinct,1 this randomized algorithm has a good expected running time and no particular input elicits its worst-case behavior. (See Problem 7-2 for the case in which elements may be equal.) Section 7.4 analyzes the randomized algorithm, showing that it runs in $\Theta(n^2)$ time in the worst case and, assuming distinct elements, in expected $O(n \log n)$ time.

randomized-partition(A, p, r)

```
1 i ← random (p, r)
2 A_r ↔ A_i
3 return Partition (A, p, r)
```

randomized-quicksort (A, p, r)

```
1 if p < r then
2     q ← randomized-partition (A, p, r)
3     randomized-quicksort (A, p, q − 1)
4     randomized-quicksort (A, q + 1, r)
```

¡++¿

Partitioning the array The key to the algorithm is the PARTITION procedure on the next page, which rearranges the subarray A[p : r] in place, returning the index of the dividing point between the two sides of the partition. Figure 7.1 shows how PARTITION works on an 8-element array. PARTITION always selects the element x = A[r] as the pivot. As the procedure runs, each element falls into exactly one of four regions, some of which may be empty. At the start of each iteration of the for loop in lines 3–6, the regions satisfy certain properties, shown in Figure 7.2. We state these properties as a loop invariant:

Partition(A, p, r)

```
1 x ← A_r
2 i ← p − 1
3 for j ∈ [p, r − 1] do
4     if A_j ≤ x then
5         i ← i + 1
6         A_i ↔ A_j
7 A_{i+1} ↔ A_r
8 return i + 1
```

At the beginning of each iteration of the loop of lines 3–6, for any array index k, the following conditions hold: • if p k i, then A[k] x (the tan region of Figure 7.2); • if i + 1 k j – 1, then A[k] ¿ x (the blue region); • if k = r, then A[k] = x (the yellow region). We need to show that this loop invariant is true prior to the first iteration, that each iteration of the loop maintains the invariant, that the loop terminates, and that correctness follows from the invariant when the loop terminates. Initialization: Prior to the first iteration of the loop, we have i = p – 1 and j = p. Because no values lie between p and i and no values lie between i + 1 and j – 1, the first two conditions of the loop invariant are trivially satisfied. The assignment in line 1 satisfies the third condition. Maintenance: As Figure 7.3 shows, we consider two cases, depending on the outcome of the test in line 4. Figure 7.3(a) shows what happens when A[j] ¿ x: the only action in the loop is to increment j. After j has been incremented, the second condition holds for A[j – 1] and all other entries remain unchanged. Figure 7.3(b) shows what happens when A[j] x: the loop increments i, swaps A[i] and A[j], and then increments j. Because of the swap, we now have that A[i] x, and condition 1 is satisfied. Similarly, we also have that A[j – 1] ¿ x, since the item that was swapped into A[j – 1] is, by the loop invariant, greater than x. Termination: Since the loop makes exactly r – p iterations, it terminates, whereupon j = r. At that point, the unexamined subarray A[j : r – 1] is empty, and every entry in the array belongs to one of the other three sets described by the invariant. Thus, the values in the array have been partitioned into three sets: those less than or equal to x (the low side), those greater than x (the high side), and a singleton set containing x (the pivot).