Quicksort And Liner Time Sorts - Recitation 6

Quicksort, Countingsort, Radixsort, And Bucketsort.

November 19, 2022

Till now, we have quantified the algorithm performance against the worst-case scenario. And we saw that according to that measure, In the comparisons model, one can not sort in less than $\Theta(n \log n)$ time. In this recitation, we present two new main concepts that, in some instances, achieve better analyses. The first one is the Exception Complexity; By Letting the algorithm behave non-determinately, we might obtain an algorithm that most of the time runs to compute the task fast. Yet we will not success get down the $\Theta(n \log n)$ lower bound, but we will go back to use that concept in the pending of the course. The second concept is to restrict ourselves to dealing only with particular inputs. For example, We will see that if we suppose that the given array contains only integers in a bounded domain, then we can sort it in linear time.

Quicksort.

The quicksort algorithm is a good example of a **non-determistic** algorithm that has a worst-case running time of $\Theta\left(n^2\right)$. Yet its expected running time is $\Theta(n \log n)$. Namely, fix an array of n numbers. The running of Quicksort over that array might be different. Each of them is a different event in probability space, and the algorithm's running time is a random variable defined over that space. Saying that the algorithm has the worst space complexity of $\Theta(n^2)$ means that there exists an event in which it runs $\Theta\left(n^2\right)$ time with non-zero probability. But practically, the interesting question is not the existence of such an event but how likely it would happen. It turns out that the expectation of the running time is $\Theta(n \log n)$.

What is the exact reason that happens? By giving up on the algorithm behavior century, we will turn the task of engineering a bad input impossible.

randomized-partition(A, p, r)

$i \leftarrow$ random (p, r)
$A_r \leftrightarrow A_i$
return Partition (A, p, r)

randomized-quicksort (A, p, r)

if $p < r$ **then**
    $q \leftarrow$ randomized-partition (A, p, r)
    randomized-quicksort $(A, p, q-1)$
    randomized-quicksort $(A, q+1, r)$

Partitioning. To complete the correctness proof of the algorithm (most of it passed in the Lecture), we have to prove that the partition method is indeed rearranging the array such that all the elements contained in the right subarray are greater than all the elements on the left subarray.

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
```

claim. At the beginning of each iteration of the loop of lines 3–6, for any array index k, the following conditions hold:

- if $p \leq k \leq i$, then $A_k \leq x$.
- if $i + 1 \leq k \leq j–1$, then $A_k > x$.
- if $k = r$, then $A_k = x$.

Proof.

1. 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.

2. Maintenance: we consider two cases, depending on the outcome of the test in line 4, 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. When $A_j \leq x$: the loop increments i, swaps A_i and A_j, and then increments j. Because of the swap, we now have that $A_i \leq x$, 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.

3. Termination: Since the loop makes exactly $r–p$ iterations, it terminates, whereupon$j = r$. At that point, the unexamined subarray $A_j, A_{j+1}..A_{r–1}$ 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).

1	5	8	5	3	2	1	
1	5	8	5	3	2	1	quicksort$(A, 0, 6)$
1	5	8	5	3	2	1	
1	5	8	5	3	2	1	
1	1	8	5	3	2	5	
1	1	8	5	3	2	5	quicksort$(A, 2, 6)$
1	1	8	5	5	2	3	
1	1	2	5	5	8	3	

1	1	2	3	5	8	5	
1	1	2	3	5	8	5	quicksort$(A, 4, 6)$
1	1	2	3	5	5	8	
1	1	2	3	5	5	8	
1	1	2	3	5	5	8	
1	1	2	3	5	5	8	quicksort$(A, 4, 5)$
1	1	2	3	5	5	8	
1	1	2	3	5	5	8	

Linear Time Sorts

Counting sort. Counting sort assumes that each of the n input elements is an integer at the size at most k. It runs in $\Theta(n+k)$ time, so that when $k = O(n)$, counting sort runs in $\Theta(n)$ time. Counting sort first determines, for each input element x, the number of elements less than or equal to x. It then uses this information to place element x directly into its position in the output array. For example, if 17 elements are less than or equal to x, then x belongs in output position 17. We must modify this scheme slightly to handle the situation in which several elements have the **same value**, since we do not want them all to end up in the same position.

Counting-sort(A, n, k)

let B and C be new arrays at size n and k
for $i \in [0, k]$ **do**
    $C_i \leftarrow 0$
for $j \leftarrow [1, n]$ **do**
    $C_{Aj} \leftarrow C_{A_j} + 1$
for $i \in [1, k]$ **do**
    $C_i \leftarrow C_i + C_{i-1}$
for $i \in [n, 1]$ **do**
    $B_{C_{A_j}} \leftarrow A_j$
    $C_{A_j} \leftarrow C_{Aj}{-}1$ // to handle duplicate values
return B

Notice that the Counting sort can beat the lower bound of $\Omega(n \log n)$ only because it is not a comparison sort. In fact, no comparisons between input elements occur anywhere in the code. Instead, counting sort uses the actual values of the elements to index into an array.

An important property of the counting sort is that it is **stable**.

Stable Sort.

We will say that a sorting algorithm is stable if elements with the same value appear in the output array in the same order as they do in the input array.

Counting sort's stability is important for another reason: counting sort is often used as a subroutine in radix sort. As we shall see in the next section, in order for radix sort to work correctly, counting sort must be stable.

Radix sort Radix sort is the algorithm used by the card-sorting machines you now find only in computer museums. The cards have 80 columns, and in each column, a machine can punch a hole in one of 12 places. The sorter can be mechanically "programmed" to examine a given column of each card in a deck and distribute the card into one of 12 bins depending on which place has been punched. An operator can then gather the cards bin by bin, so that cards with the first place punched are on top of cards with the second place punched, and so on.

The Radix-sort procedure assumes that each element in the array A has d digits, where digit 1 is the lowest-order digit and digit d is the highest-order digit.

radix-sort(A, n, d)

for $i \in [1, d]$ **do**
    use a stable sort to sort array A on digit i

Correctness Proof. By induction on the column being sorted.

- Base. Where $d = 1$, the correctness follows immediately from the correctness of our base sort subroutine.
- Induction Assumption. Assume that Radix-sort is correct for any array of numbers containing at most $d - 1$ digits.
- Step. Let A' be the algorithm output. Consider $x, y \in A$. Assume without losing generality that $x > y$. Denote by x_d, y_d their d-digit and by $x_{/d}, y_{/d}$ the numbers obtained by taking only the first $d - 1$ digits of x, y. Separate in two cases:
  - If $x_d > y_d$ then a scenario in which x appear prior to y is imply contradiction to the correctness of our subroutine.
  - So consider the case in which $x_d = y_d$. In that case, it must hold that $x_{/d} > y_{/d}$. Then the appearance of x prior to y either contradicts the assumption that the base algorithm we have used is stable or that x appears before y at the end of the $d - 1$ iteration. Which contradicts the induction assumption.

The analysis of the running time depends on the stable sort used as the intermediate sorting algorithm. When each digit lies in the range 0 to k–1 (so that it can take on k possible values), and k is not too large, counting sort is the obvious choice. Each pass over n d-digit numbers then takes $\Theta(n + k)$ time. There are d passes, and so the total time for radix sort is $\Theta\left(d(n + k)\right)$.

Bucket sort. Bucket sort divides the interval [0, 1) into n equal-sized subintervals, or buckets, and then distributes the n input numbers into the buckets. Since the inputs are uniformly and independently distributed over [0, 1), we do not expect many numbers to fall into each bucket. To produce the output, we simply sort the numbers in each bucket and then go through the buckets in order, listing the elements in each.

bucket-sort(A, n)

let B[0 : n − 1] be a new array **for** $i \leftarrow [0, n{-}1]$ **do**
    make B_i an empty list
for $i \leftarrow [1, n]$ **do**
    insert A_i into list $B_{\lfloor nA_i \rfloor}$
for $i \leftarrow [0, n{-}1]$ **do**
    sort list B_i
concatenate the lists $B_0, B1, .., B_{n-1}$ together and
return the concatenated lists