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.

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. To complete the correctness proof of the algorithm (that most of it has passed in the Lecture). We have to prove that the partition method is indeeded rearrange the array such that all the elements contained in right subarray are greater than all the elements on the left subarray.

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 $p \le k \le i$, then $A_k \le x$ (the tan region of Figure 7.2);
- if $i + 1 \le k \le j$–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.

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: 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 $A_j \le x$: the loop increments i, swaps A_i and A_j, and then increments j. Because of the swap, we now have that $A_i \le x$, 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 $A_{j:r-1}$ 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	1	5	3	2	-1
-1	5	1	5	3	2	1
-1	1	1	5	3	2	5
-1	1	1	5	3	2	5
-1	1	1	5	3	2	5
-1	1	1	5	3	2	5
-1	1	1	5	3	2	5
-1	1	1	5	3	2	5
-1	1	1	5	3	2	5
-1	1	1	5	2	3	5
-1	1	1	5	2	3	5
-1	1	1	5	2	3	5
-1	1	1	2	5	3	5
-1	1	1	2	3	5	5
-1	1	2	1	3	5	5
-1	1	2	1	3	5	5
-1	1	1	2	3	5	5

3	4	2	5	6	2	1	2	5	7	10	6
3	4	2	5	6	2	1	2	5	7	10	6
3	4	2	5	6	2	1	2	5	7	10	6
3	4	2	5	6	2	1	2	5	7	10	6
3	4	2	5	6	2	1	2	5	7	10	6
3	4	2	5	6	2	1	2	5	7	10	6
3	4	2	5	6	2	1	2	5	7	10	6
3	4	2	5	6	2	1	2	5	7	10	6
3	4	2	5	6	2	1	2	5	7	10	6
3	4	2	5	6	2	1	2	5	7	10	6
3	4	2	5	6	2	1	2	5	6	10	7
3	4	2	5	5	2	1	2	6	6	10	7
3	4	2	5	5	2	1	2	6	6	10	7
3	4	2	5	5	2	1	2	6	6	10	7
3	4	2	5	5	2	1	2	6	6	10	7
3	4	2	5	5	2	1	2	6	6	10	7
3	4	2	5	5	2	1	2	6	6	10	7
3	4	2	5	5	2	1	2	6	6	10	7
3	4	2	5	5	2	1	2	6	6	10	7
3	4	2	5	5	2	1	2	6	6	10	7
3	4	2	5	5	2	1	2	6	6	10	7
3	4	2	2	5	2	1	5	6	6	10	7

0.2 Linear Time Sorts

8.2 Counting sort Counting sort assumes that each of the n input elements is an integer in the range 0 to k, for some integer 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. The COUNTING-SORT procedure on the facing page takes as input an array A[1 : n], the size n of this array, and the limit k on the nonnegative integer values in A. It returns its sorted output in the array B[1 : n] and uses an array C [0 : k] for temporary working storage.

Counting-sort(A, n, k)

```
1  let B and C be new arrays at size n and k for i ∈ [0, k] do
2      C_i ← 0
3  for j ← [1, n] do
4      C_Aj ← C_Aj + 1
5  // C_i now contains the number of elements equal to i. for i ∈ [1, k] do
6      C_i ← C_i + C_i−1
7  // C_i now contains the number of elements less than or equal to i. // Copy A to
       B, starting from the end of A. for i ∈ [n, 1] do
8      B_C_Aj ← A_j
9      C_Aj ← C_Aj−1
10     // to handle duplicate values
11 return B
```

Counting Sort Counting sort can beat the lower bound of $\Omega(n \log n)$ proved in Section 8.1 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. The $\Omega(n \log n)$ lower bound for sorting does not apply when we depart from the comparison sort model.

An important property of counting sort is that it is stable: elements with the same value appear in the output array in the same order as they do in the input array. That is, it breaks ties between two elements by the rule that whichever element appears first in the input array appears first in the output array. Normally, the property of stability is important only when satellite data are carried around with the element being sorted. 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.

For decimal digits, each column uses only 10 places. (The other two places are reserved for encoding nonnumeric characters.) A d-digit number occupies a field of d columns. Since the card sorter can look at only one column at a time, the problem of sorting n cards on a d-digit number requires a sorting algorithm. Intuitively, you might sort numbers on their most significant (leftmost) digit, sort each of the resulting bins recursively, and then combine the decks in order. Unfortunately, since the cards in 9 of the 10 bins must be put aside to sort each of the bins, this procedure generates many intermediate piles of cards that you would have to keep track of. (See Exercise 8.3-6.) Radix sort solves the problem of card sorting—counterintuitively—by

sorting on the least significant digit first. The algorithm then combines the cards into a single deck, with the cards in the 0 bin preceding the cards in the 1 bin preceding the cards in the 2 bin, and so on. Then it sorts the entire deck again on the second-least significant digit and recombines the deck in a like manner. The process continues until the cards have been sorted on all d digits. Remarkably, at that point the cards are fully sorted on the d-digit number. Thus, only d passes through the deck are required to sort. Figure 8.3 shows how radix sort operates on a "deck" of seven 3-digit numbers. In order for radix sort to work correctly, the digit sorts must be stable. The sort performed by a card sorter is stable, but the operator must be careful not to change the order of the cards as they come out of a bin, even though all the cards in a bin have the same digit in the chosen column. In a typical computer, which is a sequential random-access machine, we sometimes use radix sort to sort records of information that are keyed by multiple fields. For example, we might wish to sort dates by three keys: year, month, and day. We could run a sorting algorithm with a comparison function that, given two dates, compares years, and if there is a tie, compares months, and if another tie occurs, compares days. Alternatively, we could sort the information three times with a stable sort: first on day (the "least significant" part), next on month, and finally on year. The code for radix sort is straightforward. The RADIX-SORT procedure assumes that each element in array $A_{1:n}$ has d digits, where digit 1 is the lowest-order digit and digit d is the highest-order digit.

radix-sort(A, n, d)

```
1 for i ∈ [1, d] do
2     use a stable sort to sort array A on digit i
```

Although the pseudocode for RADIX-SORT does not specify which stable sort to use, COUNTING-SORT is commonly used. If you use COUNTING-SORT as the stable sort, you can make RADIX-SORT a little more efficient by revising COUNTING-SORT to take a pointer to the output array as a parameter, having RADIX-SORT preallocate this array, and alternating input and output between the two arrays in successive iterations of the for loop in RADIX-SORT. Lemma 8.3 Given n d-digit numbers in which each digit can take on up to k possible values, RADIX-SORT correctly sorts these numbers in $\Theta(d(n+k))$ time if the stable sort it uses takes $\Theta(n+k)$ time. Proof The correctness of radix sort follows by induction on the column being sorted (see Exercise 8.3-3). 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 $\Theta(d(n+k))$. When d is constant and k = O(n), we can make radix sort run in linear time. More generally, we have some flexibility in how to break each key into digits. Lemma 8.4 Given n b-bit numbers and any positive integer r b, RADIX-SORT correctly sorts these numbers in $\Theta((b/r)(n+2r))$ time if the stable sort it uses takes (n + k) time for inputs in the range 0 to k. Proof For a value r b, view each key as having d = b/r digits of r bits each. Each digit is an integer in the range 0 to 2r – 1, so that we can use counting sort with k = 2r – 1. (For example, we can view a 32-bit word as having four 8-bit digits, so that b = 32, r = 8, k = 2r – 1 = 255, and d = b/r = 4.) Each pass of counting sort takes $\Theta(n+k) = \Theta(n+2r)$ time and there are d passes, for a total running time of $\Theta(d(n+2r)) = \Theta((b/r)(n+2r))$.

Given n and b, what value of r b minimizes the expression (b/r)(n + 2r)? As r decreases, the factor b/r increases, but as r increases so does 2r. The answer depends on whether b ¡ lg n. If b ¡ lg n, then r b implies (n + 2r) = (n). Thus, choosing r = b yields a running time of (b/b)(n + 2b) = (n), which is asymptotically optimal. If b lg n, then choosing r = lg n gives the best running time to within a constant factor, which we can see as follows.1 Choosing r = lg n yields a running time of (bn/lg n). As r increases above lg n, the 2r term in the numerator increases faster than the r term in the denominator, and so increasing r above lg n yields a running time of (bn / lg n). If instead r were to decrease below lg n, then the b/r term increases and the n + 2r term remains at (n). Is radix sort preferable to a comparison-based sorting algorithm, such as quicksort? If b = O(lg n), as is often the case, and r lg n, then radix sort's running time is (n), which appears to be better than quicksort's expected running time of (n lg n). The constant factors hidden in the -notation differ, however. Although radix sort may make fewer passes than quicksort over the n keys, each pass of radix sort may take significantly longer. Which sorting algorithm to prefer depends on the characteristics of

the implementations, of the underlying machine (e.g., quicksort often uses hardware caches more effectively than radix sort), and of the input data. Moreover, the version of radix sort that uses counting sort as the intermediate stable sort does not sort in place, which many of the (n lg n)-time comparison sorts do. Thus, when primary memory storage is at a premium, an in-place algorithm such as quicksort could be the better choice.

Bucket sort. Bucket sort assumes that the input is drawn from a uniform distribution and has an average-case running time of O(n). Like counting sort, bucket sort is fast because it assumes something about the input. Whereas counting sort assumes that the input consists of integers in a small range, bucket sort assumes that the input is generated by a random process that distributes elements uniformly and independently over the interval [0, 1). (See Section C.2 for a definition of a uniform distribution.) 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. The BUCKET-SORT procedure on the next page assumes that the input is an array A[1 : n] and that each element A[i] in the array satisfies 0 A[i] ¡ 1. The code requires an auxiliary array B[0 : n – 1] of linked lists (buckets) and assumes that there is a mechanism for maintaining such lists. (Section 10.2 describes how to implement basic operations on linked lists.) Figure 8.4 shows the operation of bucket sort on an input array of 10 numbers.

bucket-sort(A, n)

```
1 let B[0 : n – 1] be a new array for i ← [0, n–1] do
2   make B_i an empty list
3 for i ← [1, n] do
4   insert A_i into list B_⌊nA_i⌋]
5 for i ← [0, n–1] do
6   sort list B_i
7 concatenate the lists B_0, B1, .., B_n–1 together and
8 return the concatenated lists
```