Avancerade algoritmer
Uppgift A

Peter Boström
pbos@kth.se

2010-10-01

Assignment

In the lectures you've seen how to sort n word-sized integers on a unit-cost RAM model in O(n log log n) time. In this homework you will study special cases where it's possible to find easier algorithms or better time bounds.

1 Give a short description of the unit-cost RAM model and explain how the word-size w gives an upper bound on the number of integers n in the sorting problem above. (5p)

The unit-cost RAM model gives a simplified computing model where all operations, arithmetic as well as loading/storing are performed in constant time.

A RAM model using a word size w, uses word-sized address pointers as well. Each element to be sorted (or even stored) requires an unique address. A pointer using a word size of w bits, cannot be used to address more than 2^w elements. This introduces a limit to the number of elements that can be addressed, and therefore sorted, to 2^w.

2 If the maximum element m is O(n) you may sort in linear time using a simple algorithm. Describe how. (5p)

Counting sort. (Note that all integers are positive or zero.)

Reserve space for $m + 1$ elements, initialize all to zero. This is assumed to be done in $O(m)$ and therefore $O(n)$ time. This array, *counts*, will be used to count the number of occurances of each element. Increase $counts[i]$ for each element i in the list. All elements are now represented by the *counts* array, and the sum of each element in the *counts* array will be equal to the length of the original array, that is n. Now start writing back the counts of the array. Iterate through each possible value i which goes from 0 to m, and insert it into the array $counts[i]$ times. This is an $O(n) + O(m)$ operation, that is $O(n)$.

3 Give an algorithm that sorts in linear time when the maximum element m is $O(n^k)$, where k is a positive constant. (5p)

Radix sort. Radix sort does a lexicographical sort with a certain base. When sorting text strings this could be done with base 256, if each character is a byte. Each text string can then be sorted according to their MSD, most significant digit, into 256 separate lists. These lists will be in rising order, that is each element in a "smaller list" will be larger than every element in all "larger lists". Then we repeat the step internally for the second most significant digit, and so on. Recursively, when all lists of a level is sorted, they're merged back in. This small step is similar to merge sort. Each step is $O(n)$, and we have repeat the process k times, where k is the number of digits, or characters, in the longest string. The step for picking out a digit can be made $O(1)$, constant and gives $O(n * k)$ in total. k is constant, which gives $O(n)$.

The previous example was with strings that would be of constant length. Our problem is a bit more tricky. The size of our largest key depend on the number of integers to sort. We will then perform a trick that lets us sort these linearly in $O(k)$ steps, even though the maximum element is allowed to get larger for larger lists. This trick is done as follows: Instead of sorting into a constant number of buckets; 256 in the previous example, we will sort into n buckets. If we get a longer list, we use more buckets. And as the largest key will not require more than $O(k)$ sorting steps, this sorting is done in linear time.

In short, radix sort with varying base, base-n digits.

4 Describe an algorithm that sorts in linear time if there are many repeated elements. How many distinct elements can your algorithm handle and how fast is it? (5p)

Create a list with reserved space for k tuples, with two values, 'value' and 'count'. This will be used to store each distinct element and count how many times it's been included in the list.

For each element, find the corresponding tuple for that element's value. If the corresponding tuple exists, increase count, else create a new tuple with count one.

Empty the original list, then sort all tuples by value, and go through them in order. Append 'count' of 'value' elements to the list. As all tuples are sorted, the list will also be sorted.

$O(k)$ tuples will be inserted into the list over time, each insertion being constant. $O(k)$.

This list of tuples will be looked upon n times, and as the list is unsorted, each of them will take $O(k)$ time, this step will take $O(n * k)$.

Sorting the tuple list takes $O(k * log(k))$. As the tuples cannot be more than n, merging back into the list will take $O(n)$ time.

In total, this gives an algorithm with $O(n * k + k * log(k))$ k running time. If we set the number of repeated elements to a constant, say 42, this algorithm can be improved to sort a list of any size with 42 different values in linear time.

By using a different data structure for the tuples which keeps the elements sorted and guarantees $O(log(n))$ insertion time and $O(n * log(n))$ removal we can get an algorithm which runs in $O(n * log(k) + k * log(k)$. Note that as $k \leq n$, this results in an $O(n * log(k))$ algorithm which sorts n elements of k distinct values. This can be done for instance by using a balanced binary tree. Limit the number of values k to a constant, and it still sorts in linear time, but with a better constant.

5 The algorithm discussed in class uses large amounts of memory. How much? By doing the radix sorting phase in more than two steps you may reduce the memory requirements while increasing the running time. Explain how to do this and give a formula describing the time-space tradeoff. (5p)

The radix sorting phase uses $2*2^{W/2}$ buckets, using a common word size of 32 bits, this gives $2*2^{16} = 2*17$ buckets, but only gives 2 radix steps. If we decide to do this in n steps, we get a $n * 2^{W/n}$ buckets, where n must be a power of two, or numbers won't be divisible. So say that we did this in four steps instead, giving $4 * 2^{32/4} = 4 * 2^8 = 2^{10}$ buckets.

To have this work, we first perform a radix step on the W/n MSB, then the second most significant bits, and so on, in n steps. When we perform the merging step, we merge from right to left. The LSB are sorted into the second least significant bit buckets, and so on until everything's been merged back into the MSB buckets..

This means that the splitting radix step with 4 buckets take twice as long, as there are 4 lists to split to, instead of 2. The merging part of radix will however take three times as long, as we only had to perform one merge before, but now how to merge from the last, $4th$, to the second last, $3rd$, to the $2nd$, from here we do that only merge the two-step version had to deal with. So we have two additional steps.