CS512 LECTURE NOTES - LECTURE 8

0.1 Solving the recurrence relation

We will now show that $T(n) \leq cn$ for some $c > 0$, i.e. $T(n) \in O(n)$

We will show this by assuming it to be true, and then find the value of c.

Assume $T(n) \in O(n)$, then $T(n) \leq cn$ for some $c > 0$.

If we put this in the recurrence relation we have:

$$T(n) \leq c(\frac{n}{5}) + c(\frac{7}{10}n) + an \leq cn$$

Multiplying times 10 and solving for c we get

$$\begin{aligned} c(\frac{n}{5}) + c(\frac{7}{10}n) + an &\leq cn \\ 2cn + 7cn + 10an &\leq 10cn \\ 10cn - 2cn - 7cn &\geq 10an \\ c &\geq 10a \end{aligned}$$

So $T(n) \leq cn$ if $c \geq 10a$.

Therefore $T(n) \in O(n)$ and the **Selection** algorithm runs in linear time (worst case).

1 Sorting without comparisons

1.1 Counting sort

Suppose that we have to sort the following list:

1	2	1	4	2	1	4	3	2	3	4	2

We can use the values stored in the array as indices to a counter where we just keep a running count of the number of times that each element (key) has appeared in the original array:

key	times
1	3
2	4
3	2
4	3

Which means that the element with key=1 appears 3 times, the element with key=2 appears 4 times, etc.

Notice that we cannot just write the sorted array as:

1 1 1 2 2 2 3 3 4 4 4

because those numbers represent keys and each element of the array might contain more data. They could represent students, and thy might have a name, address, gpa, etc. The number used for sorting could be the year (1=freshman, 2=sophomore, 3=junior, 4=senior).

In order to effectively sort the elements we will find out the location that each element should have in the final sorted array: Since there are 3 elements with key=1, the last element with key=1 must be in position 3, and since there are 4 elements with key=2, the position of the last element with key=2 must be 3+4=7. So we can easily compute the position of the last element of each key:

key	times	position of last element
1	3	3
2	4	3+4=7
3	2	7+2=9
4	3	9+3=12

We can now scan the original array in reverse order and move the corresponding element to the sorted final list in its proper location, decreasing the position of last for each element that we move.

Notice that this strategy works only if the keys to be used are in the range from 1 to k for a fixed k

The algorithm is as follows:

Algorithm CountingSort
Input: $a[1], a[2], \ldots, a[n]$
  where $1 \leq a[i] \leq k \ \forall i = 1 \ldots n$
for $i = 1$ to k
  $c[i] = 0$
for $i = 1$ to n
  $c[a[i]] + +$
for $j = 2$ to k
  $c[j] = c[j-1] + c[j]$
for $i = n$ downto 1
  $b[c[a[i]]] = a[i]$
  $c[a[i]] - -$

Analysis

Notice that NO comparisons are made in order to sort the list. We will count assignments, additions and subtractions. If we assume that those operations can be done in constant time then the time complexity of the algorithm is:

$$\Theta(n + k)$$

In the case where $k \in \Theta(n)$ then the algorithm works in linear time.

Stability

Notice that in the case of duplicate keys, the order in which the elements are placed on the final sorted list is the same as the order in which the elements appear in the original list. This property is called *Stability*.

Counting sort is stable.

1.2 RadixSort

Suppose that we have a list of 100 employees and we want to sort them by SSN (XXX-XX-XXXX), each SSN has 9 digits, i.e. if we wanted to use counting sort we would have to set $k = 10^9$, and even if we only had 100 employees we could still need to do at least 10^9 operations to sort them!

To avoid this problem, the idea would be to sort the list one digit at a time, starting from the least significant digit.

The algorithm is as follows:

Algorithm RadixSort
Input: $a[1], a[2], \ldots, a[n]$
for $i = 1$ to d
  Use counting sort to sort a by digit i

The correctness of this algorithm is easy to prove:

Suppose that two elements x and y are different. Since they are different, let p be the most significant digit where there is a difference. Clearly the algorithm will correctly place x before y when sorting by digit p. Notice that after that phase , since counting sort is stable, the remaining "more" significant digits will be the same, and the order of x and y will not change. Therefore, the entire list will be sorted after the last phase.

Example:

Original	digit 1	digit 2	digit 3
25$\underline{3}$	4$\underline{2}$2	$\underline{5}$14	138
62$\underline{4}$	2$\underline{5}$3	$\underline{9}$15	176
32$\underline{8}$	6$\underline{2}$4	$\underline{4}$22	253
17$\underline{6}$	5$\underline{1}$4	$\underline{6}$24	328
51$\underline{4}$	7$\underline{2}$5	$\underline{7}$25	422
72$\underline{5}$	9$\underline{1}$5	$\underline{3}$28	514
42$\underline{2}$	1$\underline{7}$6	$\underline{1}$38	624
13$\underline{8}$	3$\underline{2}$8	$\underline{2}$53	725
91$\underline{5}$	1$\underline{3}$8	$\underline{1}$76	915

Analysis

The time complexity

$$W(n,k) \in \Theta(d(n+k))$$

Since we know that the time complexity of CountingSort is $\Theta(n+k)$

In the case of binary numbers: Assume that we have n numbers, each one with a fixed number s of bits, and we divide the numbers into d digits with $r = \frac{s}{d}$ bits each. The total number of values with r bits is equal to $k = 2^r$, so we have that

$$W(n) \in \Theta(\frac{s}{r}(n+2^r)$$

Can we sort in linear time?

If we let the radix $r = \lg n$ then we have that

$$W(n) \in \Theta(\frac{s}{\lg n}(n+n)) \in O(n)$$

(remember that s is constant)