



CPE 11040830: Algorithms, Homework #4 Solution

Dr. Khalil Ahmad Yousef

(1) CLR 8.1-3

If the sort runs in linear time for m input permutations, then the height h of those paths of the decision tree consisting of the m corresponding leaves and their ancestors must be linear. Hence, we can use the same argument as in the proof of Theorem 8.1 to show that this is impossible for $m = \frac{n!}{2}$, $\frac{n!}{n}$, or $\frac{n!}{2^n}$. We have $2^h \geq m$, which gives $h \geq \lg m$, and so for all possible m 's given here, $\lg m = \Omega(n \lg n)$, so $h = \Omega(n \lg n)$. In particular,

(i) $\lg\left(\frac{n!}{2}\right) = \lg(n!) - 1 \geq (n \lg n - n \lg e) - 1$

(ii) $\lg\left(\frac{n!}{n}\right) = \lg(n!) - \lg n \geq (n \lg n - n \lg e) - \lg n$

(iii) $\lg\left(\frac{n!}{2^n}\right) = \lg(n!) - n \geq (n \lg n - n \lg e) - n$

(2) CLR 8.2-4

For the preprocessing step, compute the C array as in lines 1 through 7 of COUNTING-SORT. Then for any query NUM-IN-RANGE(a, b), we simply return $C[b] - C[a - 1]$, where $C[0] = 0$. The query requires $O(1)$ time to answer.

(3) CLR 8.3-1

COW	SEA	TAB	BAR
DOG	TEA	BAR	BIG
SEA	MOB	EAR	BOX
RUG	TAB	TAR	COW
ROW	DOG	SEA	DIG
MOB	RUG	TEA	DOG
BOX	DIG	DIG	EAR
TAB	→ BIG	→ BIG	→ FOX
BAR	BAR	MOB	MOB
EAR	EAR	DOG	NOW
TAR	TAR	COW	ROW
DIG	COW	ROW	RUG
BIG	ROW	NOW	SEA
TEA	NOW	BOX	TAB
NOW	BOX	FOX	TAR
FOX	FOX	RUG	TEA



(4) CLR 8.3-3

Basis: If $d = 1$, there is only one digit, so sorting on that digit sorts the array.

Inductive step: Assuming that RADIX-SORT works for $d - 1$ digits, we will show that it works for d digits.

RADIX-SORT sorts separately on each digit, starting from digit 1. Thus RADIX-SORT of d digits, which sort on digits $1, \dots, d$ is equivalent to RADIX-SORT of the low-order $d - 1$ digits followed by a sort on digit d . By our induction hypothesis, the sort of the low-order $d - 1$ digits works, so just before the sort on digit d , the elements are in order according to their low-order $d - 1$ digits.

The sort on digit d will order the elements by their d th digit. Consider two elements, a and b , with d th digits a_d and b_d respectively.

- If $a_d < b_d$, the sort will put a before b , which is correct, since $a < b$ regardless of the low-order digits.
- If $a_d > b_d$, the sort will put a after b , which is correct, since $a > b$ regardless of the low-order digits.
- If $a_d = b_d$, the sort will leave a and b in the same order they were in, because it is stable. But that order is already correct, since the correct order of a and b is determined by the low-order $d - 1$ digits when their d th digits are equal, and the elements are already sorted by their low-order $d - 1$ digits.

If the intermediate sort were not stable, it might rearrange elements whose d th digits were equal — elements that *were* in the right order after the sort on their lower-order digits.

(5) CLR 8.3-4

With n input elements having values between 0 and $n^2 - 1$, we represent the numbers with a two-digit place notation using n as the base(radix). For example, when $n = 1000$, 0 is represented as (000, 000), 999,999 is represented as (999, 999), and 623 is represented as (000, 623), 999 is represented as (000, 999), 1,000 is represented as (001, 000), etc. Then, by using RADIX-SORT to sort the above inputs with $d = 2$ and $k = n$, the running time is $\Theta(dn + dk) = \Theta(2n + 2k) = \Theta(2n + 2n) = \Theta(n)$.

(6) CLR 8.4-2

The worst case occurs when every element falls into the same bucket. When this occurs, INSERTION-SORT in the worst case takes $\Theta(n^2)$. So the worst case running time is determined by the worst case running time of the sorting algorithm used. If we use MERGE-SORT instead of INSERTION-SORT, then the worst case running time is $\Theta(n \lg n)$, while we still preserve the linear expected running time because $E[n_i \lg n_i] \leq E[n_i^2] = \Theta(1)$.

(7) CLR 8-3



(a) The first thing to note is that numbers with fewer digits are smaller than numbers with more digits. So the numbers first need to be sorted on the basis of how many digits they have. Say that the maximum number of digits any number has is D . Then we first sort the numbers into D sub-arrays, such that the first sub-array consists of all numbers with one digit, the second one with two digits ... and so on. The procedure SORT-BY-DIGITS given below performs this task in $O(n)$

SORT-BY-DIGITS (A, B, C, D)

1. **for** $i \leftarrow 0$ to D
2. **do** $C[i] \leftarrow 0$
3. **for** $j \leftarrow 1$ to $length[A]$
4. **do** $C[digits[A[j]]] \leftarrow C[digits[A[j]]] + 1$
5. **for** $i \leftarrow 1$ to D
6. **do** $C[i] \leftarrow C[i] + C[i - 1]$
7. Save a copy of the C array in $O(D)$ time
8. **for** $j \leftarrow length[A]$ downto 1
9. **do** $B[C[digits[A[j]]]] \leftarrow A[j]$
10. $C[digits[A[j]]] \leftarrow C[digits[A[j]]] - 1$

SORT-BY-DIGITS takes $O(n)$ because each of the for loops either iterates n times or D times and even if each number has 1 digit D is less than n . We assume that the contents of the array C were saved before the for-loop of line 8-9, this is acceptable as it will take $O(D)$ time. Once this is done, we can run radix sort on each of the subarrays. There are D subarrays, let n_k denote the number of elements in the k 'th subarray and k be the number of digits of each element in the k 'th sub-array. The following procedure sorts all sub-arrays in $O(n)$ time.

SORT-SUBARRAYS (A, B, C, D)

1. **for** $k \leftarrow 0$ to $D - 1$
2. **RADIX-SORT**($B[C[k]..C[k + 1]]$, $k + 1$)

Each invocation of RADIX-SORT takes $O(n_k k)$ time, as there are n_k numbers in the k 'th subarray and each has k digits. The total running time is given as:

$$T(n) = \sum_{k=1}^D n_k k$$

Because the sum of all n_k products is the number of digits in the entire array, we have:

$$T(n) = O(n)$$



(b) We are given a stream of n characters, that makes up upto n strings. The strings are to be sorted in alphabetic order, and a short string of k characters is to be ordered before a longer string that has the same first k characters.

We use a counting sort to sort the words based on their first letter. Then, for each initial letter, we recursively sort the words with that first letter using the sort algorithm we are designing here, but with the first letter of each word removed. If any one of these entries is just one letter long then we do not include it in the recursion but place it at the beginning of the results (when the recursion returns, place the first letter back on the front of each word). The base case of the recursion is when the set of words to sort is empty.

Analysis: Each counting sort call is $O(k + 26) = O(k)$ when k words are sorted. Let $T(n)$ be the worst-case cost for sorting total length n . For each letter a , let n_a be the total length of the words starting with letter a , and c_a be the number of words starting with a . The sort requires a recursive call of cost $T(n_a - c_a)$. Also, there is a divide and recombine cost for each subproblem of size $O(c_a)$. Also, there is a counting sort call at cost $O(\text{Sum}_a c_a)$. These last two kinds of cost can be combined as one charge or $O(\text{Sum}_a c_a)$.

The recurrence for the sort is thus $T(n) = \text{Sum}_a T(n_a - c_a) + k(\text{Sum}_a c_a)$, for some k .

Here, we know that $\text{Sum}_a n_a + c_a = n$, because every letter is either the first letter of a distinct word or one of the remaining letters passed on, and $\text{Sum}_a c_a \geq 1$, because there is at least one word. We show by substitution that $T(n)$ is $\leq dn$ for some constant d , for all $n \geq 1$.

In the base case, $n=1$, and the cost is $k(1)$, so we can take $d \geq k$.

In the recursive case, we have $T(n) \leq \text{Sum}_a d(n_a - c_a) + k(\text{Sum}_a c_a) \leq \text{Sum}_a d(n_a) [\text{using } d \geq k] \leq dn$, as desired.

(8) CLR 11.1-1

To find the maximum element of the set S , it requires searching the entire table T in the worst case. Note that NIL is returned if there are no elements in the table T ; otherwise, the index to the largest element is returned.

GET-TABLE-MAX(T)

1. $\text{maxIndex} = \text{length}(T) - 1$
2. **while** $\text{maxIndex} \geq 0$
3. **do if** $T[\text{maxIndex}] = \text{NIL}$
4. **then** $\text{maxIndex} \leftarrow \text{maxIndex} - 1$
5. **else return** maxIndex
6. **return** NIL

This procedure requires $\Theta(m)$ time in the worst case, where m is the table size.



(9) CLR 11.2-1

Let X_{ij} be an indicator random variable, such that $1 \leq i < j \leq n$ such that $X_{ij} = I\{h(i) = h(j)\}$. Note that the probability that two keys hash to the same slot is simply $\sum_{i=1}^m \frac{1}{m^2} = \frac{1}{m}$ based on simple uniform hashing and the fact that they can hash to any of the m slots in the table. Thus by Lemma 5.1, $E[X_{ij}] = \frac{1}{m}$.

Let X be the random variable that counts the number of pairs of keys that hash to the same slot (i.e., the number of collisions). Then $X = \sum_{i=1}^n \sum_{j=i+1}^n X_{ij}$.

Hence,

$$\begin{aligned}
 E[X] &= E\left[\sum_{i=1}^n \sum_{j=i+1}^n X_{ij}\right] \\
 &= \sum_{i=1}^n \sum_{j=i+1}^n E[X_{ij}] \\
 &= \sum_{i=1}^n \sum_{j=i+1}^n \frac{1}{m} \\
 &= \sum_{i=1}^n \frac{n-i}{m} \\
 &= \sum_{k=0}^{n-1} \frac{k}{m} \\
 &= \frac{1}{m} \sum_{k=1}^{n-1} k \\
 &= \frac{1}{m} \frac{n(n-1)}{2} \\
 &= \frac{n(n-1)}{2m}
 \end{aligned}$$

Hence, the expected number of collisions is $\frac{n(n-1)}{2m}$.

(10) CLR 11.4-2

HASH-DELETE(T, i)

1. $T[i] \leftarrow \text{DELETED}$

HASH-INSERT(T, k)

1. $i \leftarrow 0$

2. **repeat** $j \leftarrow h(k, i)$

3. **if** $T[j] = \text{NIL}$ **or** $T[j] = \text{DELETED}$



```
4.         then  $T[j] \leftarrow k$ 
5.         return  $j$ 
6.         else  $i \leftarrow i + 1$ 
7. until  $i = m$ 
8. error "Hash Table Overflow"
```

HASH-SEARCH does not need to be changed, since we want the function to keep searching when it finds a DELETED mark.

(11) CLR 11.4-4

For $\alpha = \frac{3}{4}$:

Expected probes in unsuccessful search : $\frac{1}{1-\alpha} = 4$.

Expected probes in successful search : $\frac{1}{\alpha} \ln\left(\frac{1}{1-\alpha}\right) = \frac{4}{3} \ln(4) \approx 1.85$

For $\alpha = \frac{7}{8}$:

Expected probes in unsuccessful search : $\frac{1}{1-\alpha} = 8$.

Expected probes in successful search : $\frac{1}{\alpha} \ln\left(\frac{1}{1-\alpha}\right) = \frac{8}{7} \ln(8) \approx 2.38$