Advanced Algorithms Assignment I

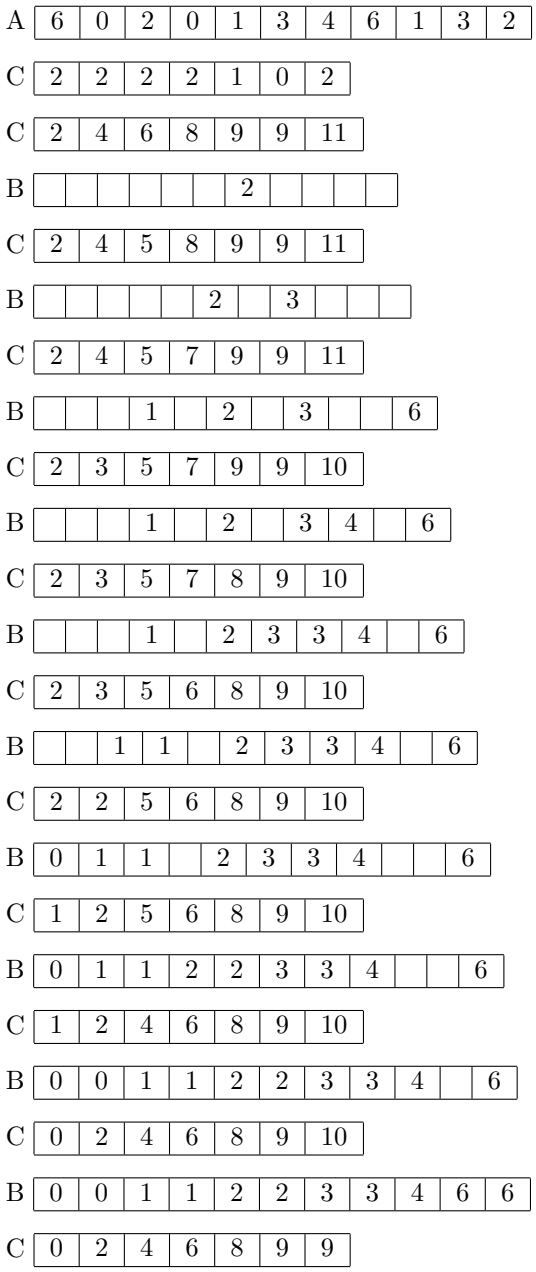
Matthew Bennett

© *Draft date September 18, 2006*

Exercise 8.1-1 What is the smallest possible depth of a leaf in a decision tree for a comparison sort?

We know already that the height of the decision tree must be $h = \Omega(n \lg n)$. That also represents the worst-case. It should be clear that $h = \Theta(n \lg n)$ as well, namely within a constant of 1 (the best case), since the deepest leaf in a complete binary tree should not differ from the depth of the shortest leaf by more than 1. Therefore, the minimum number of comparisons for any comparison sort is $n \lg n - 1$.

Exercise 8.2-1 Using Figure 8.2 as a model, illustrate the operation of COUNTING-SORT on the vector A = $\langle 6, 0, 2, 0, 1, 3, 4, 6, 1, 3, 2 \rangle$



Exercises 8.2-2 Prove that COUNTING-SORT is stable.

```
 1  for i = 0 to k
 2     do C[i] = 0
 3  for j = 1 to length[A]
 4     do C[A[j]] = C[A[j]] + 1
 5  // C[i] now contains the number of elements equal to i.
 6  for i = 1 to k
 7     do C[i] ? C[i] + C[i - 1]
 8  // C[i] now contains the number of elements less than or equal to i.
```

Lemma 1: $\forall i, C[i]$ contains the number of elements less than or equal to i, and must therefore be in increasing order.

This should be apparent from the code lines 1-8, and will not be proved.

```
 9  for j = length[A] downto 1
10     do B[C[A[j]]] = A[j]
11        C[A[j]] = C[A[j]] - 1
```

Loop invariant: For the loop starting at line 09, at each execution of the loop, any element A[j] to be placed in the resultant vector B shall be placed only to the left of any identical-valued element A[j + γ] which has already been placed.
Initialization: No elements have yet been placed when A[n] is considered, so the loop invariant holds.
Maintenance: Assume that on the j iteration, A[j + γ] = α is not the first α-valued element to be placed. Then, it will be true that B[C[A[j]]] is assigned α. We know that this α will not be placed in any of C[A[j + γ]] = C[α], C[α + 1], C[α + 2], ... because of the decrementing action on line 11. The α of iteration j must be placed left of the alpha in the previous iteration, j + γ. Specifically, at the next available left position.
Termination: Assume that on the j iteration, A[j + γ] = α is not the first α-valued element to be placed. Then, it will be true that B[C[A[j]]] is assigned α. We know that this α will not be placed in any of C[A[j + γ]] = C[α], C[α + 1], C[α + 2], ... because of the decrementing action on line 11. The α of iteration j must be placed left of the alpha in the previous iteration, j + γ. Specifically, at the only left position.

Exercises 8.3-1 Using Figure 8.3 as a model, illustrate the operation of RADIX-SORT on the following list of English words: COW, DOG, SEA, RUG, ROW, MOB, BOX, TAB, BAR, EAR, TAR, DIG, BIG, TEA, NOW, FOX

TIME $\longrightarrow$

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	TAB
DIG	COW	ROW	TAR
BIG	ROW	NOW	ROW
TEA	NOW	BOX	RUG
NOW	BOX	FOX	TAR
FOX	FOX	RUG	TEA

Exercise 8.3-2 Which of the following sorting algorithms are stable: insertion sort, merge sort, heapsort, and quicksort?

Insertion sort is stable, because we iterate through the result vector A[1..k] until we find a place to "insert" each element α. The condition for insertion is that if α is to be placed in position A[j], then $A[j-1] \leq \alpha$ and $A[j+1] > \alpha$, so the original ordering is preserved.

Merge sort is stable. When two elements have the same key, the LEFT subarray has precedence over the the right subarray, which maintains the original ordering.

Heapsort is not stable because the max-heapify or min-heapify property disregards any other ordering which is present.

Quicksort is not stable in general, because the partitioning scheme may not be stable. However, versions of quicksort may be stable if they resort to some other stable sort, such as insertion sort, for partitioning. From wikipedia, "Typically, in-place partitions are unstable, while not-in-place partitions are stable."

Give a simple scheme that makes any sorting algorithm stable. How much additional time and space does your scheme entail?

One way to assure stable sorting is to make each element a tuple, with the sorting key as the primary key, but also carry along the original index as a secondary key, which will be unique. In this case, sort by comparison on the primary key unless the primary keys are the same. If that is the case, sort on the secondary keys.

This method requires an extra binary number per element, size $\lg n$ since the original position must be stored. The space complexity undergoes $O(n \times sizeof(datatype)) = O(n) \longrightarrow O(n \lg n)$. This method also requires an extra comparison per element (worst case). The time complexity undergoes no transformation: $O(cf(n)) = O(f(n))$.