Problem 8.2

Sorting in place in linear time

Suppose that we have an array of n data records and that the key of each record has the value 0 or 1. An algorithm for sorting such a set of records might posses some subset of the following three desirable characteristics:

1. The algorithm runs in $\mathcal{O}(n)$ time
2. The algorithm is stable.
3. The algorithm sorts in place, using no more than a constant amount of storage space in addition to the original array.

Do the following:

1. Give an algorithm that satisfies criteria 1 and 2 above
2. Give an algorithm that satisfies criteria 1 and 3 above
3. Give an algorithm that satisfies criteria 2 and 3 above
4. Can you use any of your algorithms from parts (a)-(c) as the sorting method used in line 2 of `RADIX-SORT`, so that `RADIX-SORT` sorts n records with b-bit keys in $\mathcal{O}(bn)$ time? Explain how or why not.
5. Suppose that the n records have keys in the range 1 to k. Show how to modify counting sort so that it sorts the records in place in $\mathcal{O}(n+k)$ time. You may use $\mathcal{O}(k)$ storage outside the input array. Is your algorithm stable? (*Hint*: How would you do it for $k=3$?)

Algorithms

1. This can be done with counting sort. We need two variables to track the numbers/indices of ones and zeroes and $\Theta(n)$ space to make a copy.
2. This can be done with approach similar to Hoare partition in problem 7.1 (/07/problems/01.html)
3. I can't think of a stable in-place algorithm so bubble-sort will do

Usage in radix sort

Only the first one (the counting sort variant) can be used. The second is not stable, which is a requirement for radix sort, and the third takes $\Theta(n^2)$ time, which will turn the compound sorting algorithm $\Theta(bn^2)$.

In place counting sort

We build an array of counts as in `COUNTING-SORT`, but we perform the sorting differently. We start with `i = 0` and then.

```
while i ≤ A.length
    if A[i] is correctly placed
        i = i + 1
    else
        put A[i] in place, exchanging with the element there
```

On each step we're either (1) incrementing `i` or (2) putting an element in its place. The algorithm terminates because eventually we run out of misplaced elements and have to increment `i`.

There are some details about checking whether `A[i]` is correctly placed that are in the C code.

C code

```c
#include <stdbool.h>

typedef struct {
    int key;
    int value;
} item;

static item tmp;

#define EXCHANGE(a, b) tmp = a; a = b; b = tmp;

void stable_linear_sort(item *A, int size) {
    int zero = 0,
        one  = 0;
    item copy[size];

    for (int i = 0; i < size; i++) {
        if (A[i].key == 0) {
            one++;
        }
    }

    for (int i = 0; i < size; i++) {
        if (A[i].key == 0) {
            copy[zero] = A[i];
            zero++;
        } else {
            copy[one] = A[i];
            one++;
        }
    }

    for (int i = 0; i < size; i++) {
        A[i] = copy[i];
    }
}

void linear_in_place_sort(item *A, int size) {
    int left = -1,
        right = size;

    while (true) {
        do { left++;  } while (A[left].key  == 0);
        do { right--; } while (A[right].key == 1);

        if (left > right) {
            return;
        }

        EXCHANGE(A[left], A[right]);
    }
}

void stable_in_place_sort(item *A, int size) {
    for (int i = size; i > 0; i--) {
        for (int j = 0; j < i; j++) {
            if (A[j].key > A[j + 1].key) {
                EXCHANGE(A[j], A[j+1]);
            }
        }
    }
}

void in_place_counting_sort(item *A, int size, int range) {
    int counts[range + 1];
    int positions[range + 1];
```

```
    for (int i = 0; i <= range; i++) {
        counts[i] = 0;
    }

    for (int i = 0; i < size; i++) {
        counts[A[i].key]++;
    }

    for (int i = 2; i <= range; i++) {
        counts[i] += counts[i-1];
    }

    for (int i = 0; i <= range; i++) {
        positions[i] = counts[i];
    }

    int i = 0;
    while (i < size) {
        int key = A[i].key;
        bool placed = (positions[key - 1] <= i && i < positions[key]);

        if (placed) {
            i++;
        } else {
            EXCHANGE(A[i], A[counts[key] - 1]);
            counts[key]--;
        }
    }
}
```