# CS 677 Homework
# Assignment 04

Hai Nguyen

Semtember 26, 2018

1. Implement in C/C++ an algorithm to arrange elements of a given array of n real numbers so that all its negative elements precede all its positive elements. Show how your algorithm works on the following input: A = [4 3 -2 0 2 9 -1 10 0 5 23 -4].

   *Algorithm:* The algorithm will divide the array in 3 areas and there locations are in order: negative range, zero range and positive range. The algorithm starts with the zero range and this range will be widen to the left and right depending on the element it is examining. The algorithm has no need for a temporary array (memory space O(1)) and has the complexity O(n).

   *Implement in C++:*

---

```
#include <stdio.h>

void swap(int *xp, int *yp)
{
    int temp = *xp;
    *xp = *yp;
    *yp = temp;
}

void printArray(int arr[], int n)
{
    for (int i = 0; i < n; i++)
        printf("%d ", arr[i]);
    printf("\n");
}

void rearrange(int arr[], int n)
{
    int negIdx  = 0;
    int zeroIdx = 0;
    int posIdx = n - 1;

    while (zeroIdx <= posIdx) {
        if (arr[zeroIdx] < 0)
        {
            swap(&arr[negIdx], &arr[zeroIdx]);
            negIdx++;
            zeroIdx++;

            printf("Negative Pass: ");
            printArray(arr, n);
        }

        else if (arr[zeroIdx] == 0)
        {
            zeroIdx++;

            printf("Zero Pass:     ");
            printArray(arr, n);
        }
```

```
        else
        {
            swap(&arr[posIdx], &arr[zeroIdx]);
            posIdx--;

            printf("Positive Pass: ");
            printArray(arr, n);
        }
    }
}


int main()
{
    int arr[] = { 4, 3, -2, 0, 2, 9, -1, 10, 0, 5, 23, -4 };
    int n = sizeof(arr) / sizeof(arr[0]);
    rearrange(arr, n);
    printf("\nSorted array: ");
    printArray(arr, n);
    return 0;
}
```

*Ouputs:*

```
Positive Pass: -4 3 -2 0 2 9 -1 10 0 5 23 4
Negative Pass: -4 3 -2 0 2 9 -1 10 0 5 23 4
Positive Pass: -4 23 -2 0 2 9 -1 10 0 5 3 4
Positive Pass: -4 5 -2 0 2 9 -1 10 0 23 3 4
Positive Pass: -4 0 -2 0 2 9 -1 10 5 23 3 4
Zero Pass:     -4 0 -2 0 2 9 -1 10 5 23 3 4
Negative Pass: -4 -2 0 0 2 9 -1 10 5 23 3 4
Zero Pass:     -4 -2 0 0 2 9 -1 10 5 23 3 4
Positive Pass: -4 -2 0 0 10 9 -1 2 5 23 3 4
Positive Pass: -4 -2 0 0 -1 9 10 2 5 23 3 4
Negative Pass: -4 -2 -1 0 0 9 10 2 5 23 3 4
Positive Pass: -4 -2 -1 0 0 9 10 2 5 23 3 4

Sorted array: -4 -2 -1 0 0 9 10 2 5 23 3 4
```

2. Answer the following question: is Quicksort a stable sorting algorithm? If yes, give a justification. If not, provide a counterexample.

   Answer: *Quicksort is unstable*

   Counter example:

   Given an input array: A = [0 2a 4 2b]. The input array have two equal keys - 2 elements of 2. They are differed by their labels 2a and 2b. If the sorting algorithm is stable then in the sorted output, the orders of equal keys will be maintained so 2a will precede 2b. Also when applying Quicksort as shown in Figure 1, we assume that we always take the final element in the array to be the pivot.

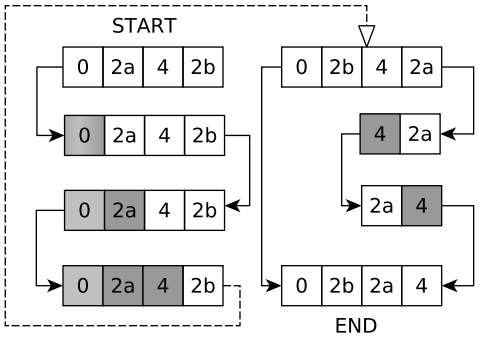


Figure 1: Counter example to prove that Quicksort is unstable.

Lightly shaded array elements are all in the first partition with values no greater than the pivot. Heavily shaded elements are in the second partition with values greater than the pivot. The unshaded elements have not yet been put in one of the first two partitions, and the final white element is the pivot. After we finish partitioning, in the sorted output array, we can see that the order between two equal keys (2a and 2b) are reversed, so Quicksort is not a stable algorithm.

3. (a) Exercise 7.1-1 (page 173) Illustrate the operation of PARTITION on the array A = [13, 19, 9, 5, 12, 8, 7, 4, 21, 2, 6, 11]

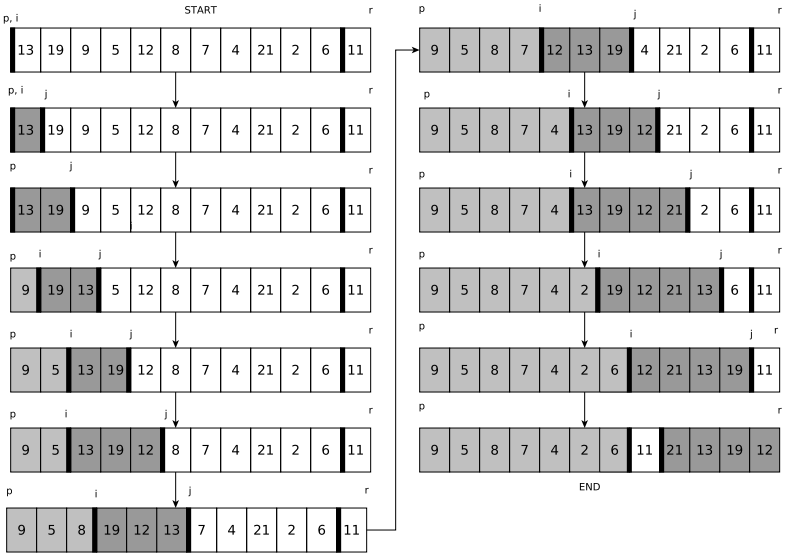


Figure 2: Quicksort partitioning process.

Figure 2 illustrates how partitioning is performed on the above array. Lightly shaded array elements are all in the first partition with values no greater than the pivot. Heavily shaded elements are in the second partition with values greater than the pivot. The unshaded elements

have not yet been put in one of the first two partitions, and the final white element is the pivot.

(b) Give an argument to show that RANDOMIZED-SELECT never makes a recursive call to a 0-length array.

**ALGORITHM**: RANDOMIZED-SELECT(A, p, r, i)
**if** p = r
  **then return** A[p]
q ← RANDOMIZED-PARTITION(A, p, r)
k ← q - p + 1
**if** i = k
  **then return** A[q]
**elseif** i < k
  **then return** RANDOMIZED-SELECT(A, p, q - 1, i)
**else return** RANDOMIZED-SELECT(A, q + 1, r, i - k)

From the algorithm, RANDOMIZED-SELECT makes a recursive call to a 0-length array when:

- Line 8: p > q - 1 → i < k = q - p + 1 ≤ 1 → i = 0 (not possible - there is no such 0-th order statistic)
- Line 9: q + 1 > r but because q ≤ r → q = r. However, in this case i > k = q - p + 1 = r - p + 1 (number of elements in A) → i is not in A (not possible)

So the algorithm will never recurse on 0-length arrays.

4. Exercise 9.3-5 (page 223). Suppose that you have a "black-box" worst-case linear-time median subroutine. Give a simple, linear-time algorithm that solves the selection problem for an arbitrary order statistic.

Algorithm to solve the selection problem for an i-order statistic in an input array A[1..n]:

**ALGORITHM** Select(A, i)

Median m ← Black-box median select for A [Time O(n)]

*(We will get A1[1..n/2-1] and A2[n/2..n] by partitioning around the median m)*

**if** i = n/2

  **return** A[n/2] [Time c1]

**elseif** i < n/2

  **return** Select(A1, i) [Time T(n/2)]

**else return** Select(A2, n/2-i) [Time T(n/2)]

Total time:

$$\begin{aligned} T(n) &\leq cn + T(n/2) \\ &\leq cn + T(n/2) \\ &\leq cn(1 + \frac{1}{2} + ... + \frac{1}{2^m}) + T(1) \\ &\leq cn\frac{1}{1 - \frac{1}{2}} \\ &\leq O(n) \end{aligned}$$

5. Exercise 9.2-4 (page 220). Suppose we use RANDOMIZED-SELECT to select the minimum element of the array A = [3, 2, 9, 0, 7, 5, 4, 8, 6, 1]. Describe a sequence of partitions that results in a worst-case performance of RANDOMIZED-SELECT.

   The partitioning process is shown in Figure 3. The worst case scenario is when RANDOMIZED-PARTITION keeps picking the maximum value in the array (number in shaded area) to be the pivot for partitioning. This results in the maximal unbalance between two partitioned arrays when one array is empty and (n - 1) remaining elements (including the minimum and excluding the pivot) are in the other array. Because all remaining elements are less than the pivot, the order of elements are kept after partitioning.

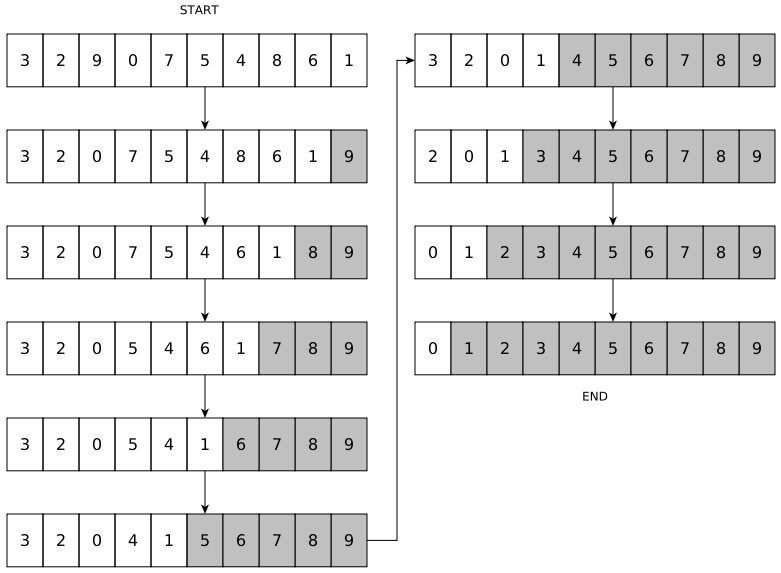


Figure 3: Worst case RANDOMIZED-SELECTION.

6. Exercise 9.3-3 (page 223). Show how Quicksort can be made to run in O(nlgn) time in the worst case, assuming that all elements are distinct.

   Using the Select algorithm mentioned in lecture, we can find the median of an array of distinct elements in linear time even in worst case. After that, we partition around the median value to get two arrays with equal number of elements. Then, we call Quicksort in these two arrays.

   **ALGORITHM** Median-Quicksort($A$)

   Choose pivot to be the median of A ← Median select [Time O(n)]

   *(We will get A1[1..n/2-1] and A2[n/2..n] by partitioning around the median)*

   Median-Quicksort(A1) [Time T(n/2)]

   Median-Quickort(A2) [Time T(n/2)]

From the above algorithm, we have the recurrence relationship:

$$\begin{aligned} T(n) &= 2T(n/2) + O(n) \\ &= 2^{lgn}T(1) + (lgn - 1)O(n) \\ &= nT(1) + lgnO(n) \\ &= O(nlgn) \end{aligned}$$

So by using the selection algorithm to select the median in linear time even in worst case, we can use that strategy to replace the normal partition process to make Quicksort to run in O(nlgn) in the worst case.