Quicksort Analysis Report

3EA3 Software Specifications and Correctness

Author: Paul Warnick – 001300963

Instructor: Musa Al-hassy

History

Quicksort is a sorting algorithm developed in 1959 by Tony Hoare. Since its publication in 1961, the algorithm has been widely used due to its exceptional efficiency and speed. With proper implementation, Quicksort can be about two or three times faster than other popular sorting algorithms (including merge sort and heapsort).

High-level Overview

As with all sorting algorithms, the goal of Quicksort is to take an input array and return a sorted[1] version of that array. The approach taken by Hoare was to:

1. Select a pivot element (say the first element in the array)
2. Partition the array into 3 separate parts based on that pivot
    a. The first part includes all elements in the array less than the pivot
    b. The second part is the pivot itself
    c. The third part contains all elements greater than or equal to the pivot
3. Recursively apply the quicksort algorithm to the first and third parts

In order to achieve a properly partitioned array we must have a separate algorithm for the partitioning step itself. Also, it's worth noting that after the array has been partitioned, the selected pivot element is now in the correct position with respect to the sorted array.

Partitioning

A simple way to understand how partitioning[2] works is to imagine all array elements are cards with one blank side and the element itself on the other side. These cards are laid out in front of you on a table in the order the array is currently in. Picking the pivot to be the left most card we can flip all *other* cards face down. Now we open each card from left to right comparing the current card to the pivot as we go. If the flipped card is less than the pivot, we swap that card with the leftmost open card (not including the pivot) and then flip that leftmost card that was just swapped so that it's now face down. If the current card is not less than the pivot then we skip it and leave it face up. Once all the cards have been inspected and rearranged, swap the right most face down card with the pivot and flip all cards face up. This will result in the array being properly partitioned (example on next page).

[1] For the purpose of this report, I'll be considering "sorted" as an ascending order list of positive numbers potentially including zero

[2] This is one of many algorithms that can be used to partition and array during Quicksort

An example of partitioning using the above method. Underline represents a card that is face down and **bold** represents the pivot:

3 4 5 1 2
3 4 5 1 2
3 4 5 1 2
3 4 5 1 2
3 4 5 1 2
3 1 5 4 2
3 1 5 4 2
3 1 5 4 2
3 1 2 4 5
3 1 2 4 5
2 1 **3** 4 5
2 1 **3** 4 5

Common Pitfalls[3]

Partitioning in Place

Novice programmers may choose to implement partitioning by using a separate array to store all values less than or equal to the partition, then the partition itself, followed by all values greater than the partition. Although this may seem easy to implement, the extra cost of creating, modifying and copying the partitioned array back to the original array can drastically slow down performance.

Staying in Bounds

Often developers may forget that the partitioning value might be the smallest or biggest value in the entire array. Certain checks must be put in place to ensure pointers do not run of the left or right ends of the array.

Preserving Randomness

Quicksort has an issue where it is relatively slow if specific pivots are picked. For example, if the pivot is always chosen to be the first element in the array and the array is in reverse order. The most ideal pivot would be the median of all values in the array, but in order to find this we have to sort the array, which eliminates the need for the actual algorithm. A good way to avoid this problem is to randomly select the pivot each time. This can be done by randomizing the input array before selecting the pivot. Programmers must be cautious not to shuffle the already partitioned arrays.

Variants

Cut-off to Insertion Sort

Although overall Quicksort is significantly faster than Insertion sort, the latter happens to be faster for tiny arrays. With this knowledge we can increase the performance of Quicksort by switching

[3] A summary of the suggestions brought forth in the text: *Algorithms, 4th Edition, Robert Sedgewick & Kevin Wayne* page 292

over to Insertion sort once (sub) array length reaches a certain size. According to Sedgewick & Wayne the length can be anywhere between 5 and 15 to increase average performance.

Median-of-Three Partitioning

Another way Quicksort can be modified to increase performance is to set the pivot to the median of a small sample of items during partitioning. This way a decent partitioning value is chosen at the cost of calculating the median of the sample. A sample size of 3 creates the best improvement.

Implementation w/ ACSL Annotations

An implementation of Quicksort in C with added ACSL annotations for checking with Frama-C:

```
10  // A utility function to swap two elements
11  /*
12      Valid: Checks if *a and *b can hold ints in memory
13      Assigns: Makes sure only pointers a and b are modified
14      Ensures: A post condition to check to see if the pointers have been swapped
15      Old: Refers to the state before the function has been run
16  */
17  /*@
18      requires \valid(a) && \valid(b);
19      assigns *a, *b;
20      ensures *a == \old(*b) && *b == \old(*a);
21  */
22  void swap(int* a, int* b)
23  {
24      int t = *a;
25      *a = *b;
26      *b = t;
27  }
```

```c
/*
    This function takes last element as pivot, places
    the pivot element at its correct position in sorted
    array, and places all smaller (smaller than pivot)
    to left of pivot and all greater elements to right
    of pivot
*/
/*
    Precondition:
        - low cannot be less than 0
        - At this stage in the execution low must be less than high
        - high cannot be greater than the length of the input array
    Assigns: Prevents any unwanted value modifications
    Loop Invariants:
        - i can never be less then low - 1 (initally it starts off equal to low - 1)
        - i is never greater then j (j is starts off greater and is incremented more frequently)
        - j can be at most high (the loop should not execute if j > high)
    Postcondition:
        - Ensures the final position of the current pivot is between low and high
        - All values to the left of the pivot index must be less than or equal to the pivot
        - All values to the right of the pivot index must be greater than the pivot
        - The pivot has been swapped to the correct location
*/
/*@
    requires 0 <= low && low < high && high <= \length(arr);
    assigns arr, pivot, i, j;
    ensures low <= \result && \result < high;
    ensures \forall integer i; 0 <= i <= \result ==> arr[i] <= arr[\result];
    ensures \forall integer i; \result < i < \length(arr) ==> arr[\result] < arr[i];
    ensures arr[\result] == arr[pivot];
*/
int partition (int arr[], int low, int high)
{
    int pivot = arr[high]; // pivot
    int i = (low - 1); // Index of smaller element

    for (int j = low; j <= high - 1; j++)
    {
        //@ loop invariant low - 1 <= i && i <= j && j <= high;
        // If current element is smaller than or equal to pivot
        if (arr[j] <= pivot)
        {
            i++; // increment index of smaller element
            swap(&arr[i], &arr[j]);
        }
    }
    swap(&arr[i + 1], &arr[high]);
    return (i + 1);
}
```

```
79  /*
80      The main function that implements QuickSort
81      arr[] --> Array to be sorted
82      low   --> Starting index
83      high  --> Ending index
84  */
85  /*
86      Precondition:
87          - low cannot be less than 0
88          - high cannot be greater than the length of the input array
89      Assigns: Only the array, low, high and partitioning index values can be modified
90      Postcondition: The array is sorted in ascending order
91  */
92  /*@
93      requires 0 <= low && high <= \length(arr);
94      assigns arr, low, high, pi;
95      ensures \forall integer i, j; 0 <= i < j < \length(arr) ==> arr[i] <= arr[j];
96  */
97  void quickSort(int arr[], int low, int high)
98  {
99      if (low < high)
100     {
101         // pi is partitioning index, arr[p] is now at right place
102         int pi = partition(arr, low, high);
103
104         // Separately sort elements before partition and after partition
105         quickSort(arr, low, pi - 1);
106         quickSort(arr, pi + 1, high);
107     }
108 }

110 // Function to print an array
111 /*
112     Forall: Checks to ensure each subsequent value is greater than or equal to the previous
113     Assigns: Makes sure the array is not actually modified by the below function
114 */
115 /*@
116     requires \forall integer i, j; 0 <= i < j < size ==> arr[i] <= arr[j];
117     assigns \nothing;
118 */
119 void printArray(int arr[], int size)
120 {
121     int i;
122     for (i=0; i < size; i++)
123         printf("%d ", arr[i]);
124 }
```

Examples and Tests

In order to test the algorithm I pass in the following array:

```
{3, 1, 4, 1, 5, 9, 2, 6, 5, 3}
```

```
126  // Driver program to test above functions
127  int main()
128  {
129      int arr[] = {3, 1, 4, 1, 5, 9, 2, 6, 5, 3};
130      int n = sizeof(arr)/sizeof(arr[0]);
131      quickSort(arr, 0, n-1);
132      printf("Sorted array: ");
133      printArray(arr, n);
134      return 0;
135  }
136
```

Upon execution of the program a sorted array is output:

```
$ ./quicksort.exe
Sorted array: 1 1 2 3 3 4 5 5 6 9
```