CS310 – Assignment 228 – Karl Ramberg

Quicksort is only as good as its partition function so analyzing different approaches for partitioning is super important.

Let's look at the two provided partition functions, `partition_j()` and `partition_w()`. Both `partition_j()` and `partition_w()` have and input size of n where n is the difference between `hi` and `lo`. When dealing with a `vector` our input size usually corresponds with the `vector` size, but both partition algortithms will only look at values between `a.at(lo)` and `a.at(hi)`.

Let's look at `partition_j()` in detail first.

`partition_j()` has one main `if` statement, but this only takes care of a few special cases, namely a singleton `vector` and a `vector` of size 0. We won't count these special cases in our analysis. This means we should only care what is inside the `if` block.

The `if` comparisons themeselves are 3 operations. Inside the block there are 2 more before we reach a `for` loop. The body of the loop will run $n - 1$ times since we start `index` at `lo + 1` and the header will run an additional time in order to exit.

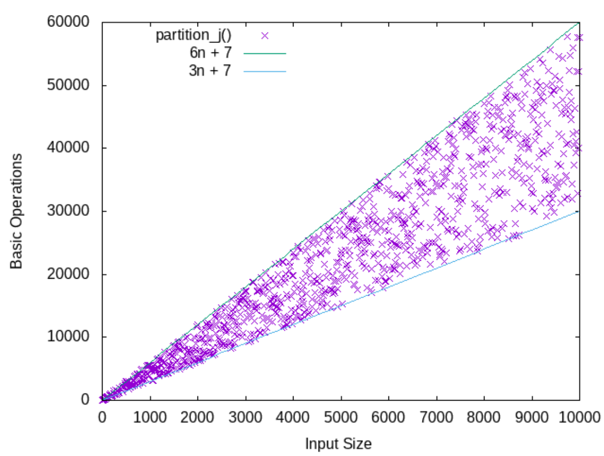
Inside the `for` loop there is a single `if` statement. The loop body will always do that 1 comparison and 3 additional if the comparison was true – 1 arthimatic operation and 2 for a swap.

This means the formal formulations of the number of basic operations as a function of input size is $T(n) = 6n + 7$ for worst case (swap every time) and $T(n) = 3n + 7$ for best case (no swaps needed).

These are both of the same efficiency class so we can conclude

$$T(n) \in \Theta(n)$$

I ran `partition_j()` many times with many different input sizes and scaled standard functions in order to illustrate this. Here are the results graphed with gnuplot...



Now let's look at `partition_w()` in detail.

`partition_w()` has one main `if` structure just like `partition_j()`. We can ignore the first `if` and `else if` because they deal with special cases of a `hi` and `lo` being invalid or only being 1 or 2 apart. However we should count the 5 comparisons needed to find the normal case.

Inside the normal case there are 4 various operations before a `while` loop. The `while` loop runs based on two markers, `left` and `right` the approach eachother from opposite sides of the `vector`. The elements at `left` and `right` will swap places if they are on the wrong side of the pivot.

For the worst case that the `vector` is reversed, the outer `while` will run n times and each inner `while` loop will only run once each time since the next value will alawys be on the wrong side.

For the best case, the outer loop will only run once since the inner loops will cross there markers on the first pass.

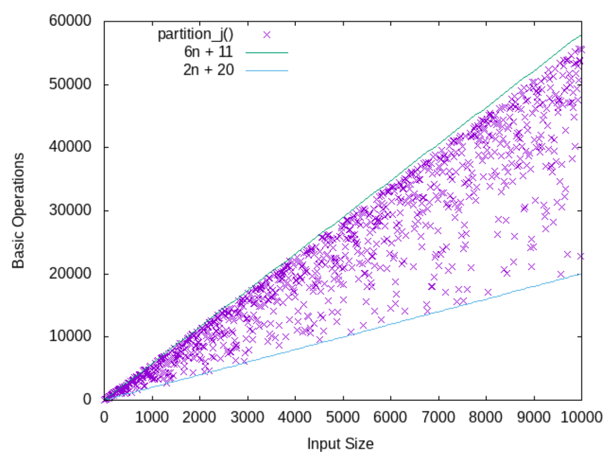
This means that each set of loops, outer and inner, will run more or less based on the input arrangment. However this means that if one runs fewer times the other will run more. This means the algorithm is not more complex simply because there are nested loops.

For the worst case (outer loop run n times) the number of basic operations as a function of input looks like so: $T(n) = 6n+11$. For the best case (inner loops run n times combined) the number of basic operations as a function of input size looks like so: $T(n) = 2n + 20$.

These are both of the same efficiency class so we can conclude

$$T(n) \in \Theta(n)$$

I ran `partition_w()` many times with many different input sizes and scaled standard functions in order to illustrate this. Here are the results graphed with gnuplot...



We can modify both of these paritioning functions to group all the duplicate pivots together. I modified `partitiion_j()` to show this. You can see the modified version at `partition_j_modified.cpp`. I did this by moving any dupliates of the pivot next to the original and swap all pivots to there position when everything else is in order. This added a few operations and a `for` loop for swapping the pivots at the end. This loop will run a maximum of n times when all elements are the same as the pivot. This means out complexity won't change. Here is the efficiency of the modifcations graphed with gnuplot...