Average Case Analysis

	InpType1			InpType2			InpType3			InpType4		
	n=100	*n*=1000	*n*=10000	*n*=100	*n*=1000	*n*=10000	*n*=100	*n*=1000	*n*=10000	*n*=100	*n*=1000	*n*=10000
Ver1	0.0005322456359863281	0.00561990737915039	0.07174201011657715	0.0005985260009765625	0.005542564392089844	0.06271123886108398	0.00034461021423339846	0.00578603744506836	0.06531810760498047	0.0004232883453369141	0.006262350082397461	0.06993842124938965
Ver2	0.0006427288055419922	0.00882267951965332	0.10002322196960449	0.000670480728149414	0.010009765625	0.09353246688842773	0.0008070468902587891	0.008229494094848633	0.0997544765472412	0.0009988307952880859	0.00840167999267578	0.09537644386291504
Ver3	0.0003589153289794922	0.006040620803833008	0.07883777618408203	0.00036182403564453123	0.006978750228881836	0.06586785316467285	0.0004639625549316406	0.005640888214111328	0.07219476699829101	0.00037326812744140623	0.005777692794799805	0.07312402725219727
Ver4	0.0005235671997070312	0.010665607452392579	0.07626562118530274	0.0004498481750488281	0.007470846176147461	0.07334842681884765	0.0004956722259521484	0.006996440887451172	0.0755739688873291	0.0004596233367919922	0.0073546886444091795	0.07684264183044434

Comments:

In our program, we have 4 different versions of quicksort algorithm, each using a different method of implementing quick sort. These methods are:
Version1: Traditional quick sort algorithm is used. Pivot is chosen as the first element in the list in each iteration.
Version2: Pivot is chosen as a random element of the list in each iteration. After this, quick sort is implemented as usual.
Version3: Lists that are handed off to be quick sorted are randomly permuted before each iteration. Afterward, the pivot is chosen as the first element in the lists.
Version4: Version1's deterministic algorithm is implemented but the pivot is chosen according to the median of three method. Its merits and advantages will be explained further down in the comments.

After running our code successfully and inspecting and comparing the different execution times for each scenario, we have found the following results:

We have observed that for both average and worst case inputs that when the input size is small(100) rather than large(10000), the execution times between the different versions of quick sort and input types were minuscule. We know that for the traditional version of quick sort(Version1), the worst case complexity is $O(n^2)$ and average case complexity is $O(n * \log n)$. Different methods used in the other versions(Version2, Version3, and Version4) are a means to improve the worst case complexity and approach it to $O(n * \log n)$. Hence, they don't provide much improvement in the average run time of the algorithms. So, the complexities of quick sort version2, version3, and version4 remain as $O(n * \log n)$. This fact can easily be observed in the execution times of different scenarios for the average case. Changing all the parameters of our program except for input size which would naturally impact the execution time(input type and quick sort version), didn't provide much difference in the execution time which was overall quite small.

As the input lists are generated randomly and are sent to the program unsorted(for average case), methods used in version2, version3 and version4 for randomizing input data don't have much effect on the run time of the algorithms, unlike worst case where the input lists are sorted, hence adding randomness has a great effect on the runtime of the algorithm. Although version3 seems to outperform the other versions in most cases, runtimes are too close together to make a distinct assumption which is expected since their complexities are the same for the average case(O($n * \log n$)). Even when the input size is 10000, although naturally higher, we don't see such a big leap in the execution time of the algorithms.

Worst Case Analysis

	InpType1			InpType2			InpType3			InpType4		
	n=100	n=1000	n=10000	n=100	n=1000	n=10000	n=100	n=1000	n=10000	n=100	n=1000	n=10000
Ver1	0.001077413558959961	0.11056232452392578	10.481112957000732	0.0008950233459472656	0.10329651832580566	9.02038311958313	0.0007624626159667969	0.08428096771240234	6.534164905548096	0.0003724098205566406	0.00711512565612793	0.06500768661499023
Ver2	0.0006284713745117188	0.008541107177734375	0.0921182632446289	0.0007007122039794922	0.008261919021606445	0.09250974655151367	0.0005645751953125	0.008589744567871094	0.08280372619628906	0.0006670951843261719	0.009233951568603516	0.09041976928710938
Ver3	0.0005009174346923828	0.005735635757446289	0.06955456733703613	0.0003783702850341797	0.0068204402923583984	0.0676259994506836	0.00038170814514160156	0.005614519119262695	0.0677499771118164	0.0007114410400390625	0.0054280757904052734	0.07796645164489746
Ver4	0.0003228187561035156	0.00524449348449707	0.0536952018737793	0.000400543212890625	0.005234241485595703	0.05366873741149902	0.0003638267517089844	0.005038022994995117	0.052274465560913086	0.00044918060302734375	0.007045745849609375	0.07697486877441406

Comments:

We know that for the traditional version of quick sort(Version1), the worst case complexity is O(n^2). Different methods used in the other versions(Version2, Version3, and Version4) are a means to improve the worst case complexity and approach it to O($n * \log n$) which is the best/average case.
Worst case input lists, unlike average case input lists, are already sorted before they are placed in the algorithm to be quick sorted. This makes the randomization used in version2, version3, and version4 of quicksort have a massive effect on the runtime as we know that already sorted lists are the worst type of input for quick sort algorithms.
By observing the execution times of different scenarios, unlike the average case, we could see big changes between the different scenarios. Similar to the average case, when the input size was small(100) rather than big(10000), the execution times were smaller and didn't vary as much. However, in all scenarios, the traditional version of quick sort(version1) performed the worst and had the highest execution time. This is because, in the traditional version of quick sort(version1), the first element in the list is always chosen to be the pivot, hence in each iteration, the next list to be quick sorted always has a maximum length.
In version 2, by choosing a random pivot, this problem is eliminated by placing the pivot in its proper position in the list hence in the next iterations, the list will likely not be as big.

In version3, although the pivot is chosen to be the first element in each iteration like the traditional quick sort algortihm(version1), as the list is permutated prior to being sorted, it is no longer the worst case type of input and performs like the average case.
In version 4, the median of three method is used. In this method, median of the first, middle and last element of the list is chosen as the pivot. These elements are sorted before being placed back in the list. This strategy also eliminates the problem of maximum list size in the next calls of quick sort method and sorting of these three elements provides better picks of pivot in the next runs of the algorithm while keeping the list relatively sorted.
The comparatively poor execution of version1 can be seen by observing the runtimes. For size 100, this difference was not as big but still apparent. As the input sizes got larger, the execution time difference between version1 and the others got bigger and for input size 10000, the difference was massive.
Between version2, version3, and version4, although there isn't much difference since they all approach complexity O($n * \log n$), version4(median of three method) mostly performed the best for the reasons presented above. After version4, version3 performed mostly better than version2.
Another difference we can observe between execution times comes from the change in input types. As we go from input type 1 to 4, the range of values that the list elements can take decreases. In fact for input type 4, all of the list elements are 1 independent of size. Hence, as we go from input type 1 to 4, the odds of having more than 1 of the same element in a list increases. When two elements are compared, the possibility of them being equal also increases. Hence, the number of elements to change the positions also decrease. This can result in a quicker run of the algorithm. This notion can easily be observed in the execution times of input 4. Even for quicksort version1 and size 10000, the list is sorted very fast.