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 | 0.00010089874267578125 | 0.001479196548461914 | 0.01720147132873535 | 0.00012660026550292970 | 0.0016021251678466797 | 0.01638960838317871 | 0.00014581680297851563 | 0.0019659519195556640 | 0.016536808013916014 | 0.00010671615600585938 | 0.0011853694915771485 | 0.013599777221679687 |
| Ver2 | 0.00013709068298339844 | 0.0023947238922119142 | 0.023316049575805665 | 0.00021605491638183595 | 0.0021514415740966795 | 0.02028684616088867 | 0.00023670196533203124 | 0.0023258686065673827 | 0.022264480590820312 | 0.0001617431640625 | 0.0014849662780761718 | 0.018249702453613282 |
| Ver3 | 0.00015664100646972656 | 0.0024740219116210936 | 0.025769138336181642 | 0.00019807815551757811 | 0.0032594680786132814 | 0.024078893661499023 | 0.00028324127197265625 | 0.0027259826660156250 | 0.025165367126464843 | 0.00018496513366699218 | 0.0017853736877441405 | 0.02048783302307129 |
| Ver4 | 0.00010972023010253906 | 0.0013367652893066407 | 0.01693096160888672 | 0.00013608932495117186 | 0.002069568634033203 | 0.0182039737701416 | 0.00020012855529785156 | 0.00202178955078125 | 0.017283296585083006 | 0.00012450218200683595 | 0.0012342453002929688 | 0.016356658935546876 |

Comments:

1. Change with respect to different versions (keeping InpType and n fixed):

-Ver1 performs best, then Ver4, then Ver2. Ver3 performs the worst.
This conclusion is consistent with the theory because of several reasons:

- Since the input is already created with randomization, the pivoting strategy does not affect the optimal partitioning of the array when calling QuickSort(L) with left and right partitions. So, runtime is affected by some other operations that differ from version to version. For example, pivot choosing strategy can cause this difference. Ver1 and Ver4 has fastest strategies, because Ver1 chooses pivot directly and Ver4 with very small complexity (sorting 3 elements and chooses median). Ver2 has the slowest strategy because it chooses a random number, this operation is slower. Thus, Ver2 is slower than Ver1 and Ver4.
- The reason that Ver3 performs the worst originates from its additional complexity that comes from shuffling the input array.

2. Change with respect to different data size, n (keeping InpType and Ver fixed):

- n = 10000 is the slowest and n = 100 is the fastest in terms of runtime.
This conclusion is consistent with the theory:

- Since all versions have complexity Theta(n*logn) in average case, their execution time increases with n regardless of InpType.

3. Change with respect to different InpType (keeping n and Ver fixed):

- InpType4 performs the best. InpType1 performs the worst.
This conclusion means that if we increase the frequency of elements (keeping n fixed), our execution time decreases. This is consistent with the theory:

- In the algorithm it can be seen that, regardless of the version, the increasing frequency of elements causes less iterations in the Rearrange(L) function’s while loops. That is the number of recursive calls remains approximately the same with a decreased cost caused by the decreasing number of operations in the Rearrange(L) function.

## 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 | 0.0008778572082519531 | 0.03826737403869629 | 3.702209234237671 | 0.0003657341003417969 | 0.024228811264038086 | 2.5919084548950195 | 0.00036287307739257 81 | 0.018218994140625 | 1.7447361946105957 | 0.0001366138458251953 | 0.0017762184143066406 | 0.01728653907775879 |
| Ver2 | 0.00050711631774902 34 | 0.0018534660339355469 | 0.021548032760620117 | 0.00018668174743652344 | 0.0017640590667724 61 | 0.027195215225219727 | 0.00028228759765625 | 0.0028414726257324 22 | 0.02664470672607422 | 0.00021982192993164062 | 0.0013158321380615234 | 0.018418550491333008 |
| Ver3 | 0.00047898292541503906 | 0.0023000240325927734 | 0.02660536766052246 | 0.00019807815551757811 | 0.0020051002502441406 | 0.03132152557373047 | 0.00033164024353027344 | 0.0030701160430908203 | 0.024943828582763672 | 0.0006823539733886719 | 0.0016193389892578125 | 0.021564245223999023 |
| Ver4 | 0.00031042098999023 44 | 0.0015590190887451172 | 0.018321752548217773 | 0.00017547607421875 | 0.0013260841369628906 | 0.02269721031188965 | 0.00022864341735839844 | 0.0013024806976318 36 | 0.022061586380004883 | 0.0009713172912597656 | 0.00180053710937 5 | 0.015014171600341797 |

Comments:

1. Change with respect to different versions (keeping InpType and n fixed):

- Ver4 is the best, Ver1 is the worst in terms of execution times. Ver2 and Ver3 has similar execution times to Ver4. This is consistent with the theory as follows:

  - The main difference comes from the total number of recursive calls of QuickSort(L). The pivoting strategy and the input format cause this difference. In Ver1 we call QuickSort(L) with the worst possible partitioning strategy that leads to highest complexity (Partitioning caused by pivoting strategy). In Ver2 and Ver3 we call QuickSort(L) with the nearly optimal partitioning strategy through choosing the pivot randomly and shuffling the array, respectively. In Ver4 we achieved best possible partitioning because input is already sorted and we always divide from the middle with our pivoting strategy of "median of three". This leads to the best performance.

2. Change with respect to different data size, n (keeping InpType and Ver fixed):

- n = 10000 is the slowest and n = 100 is the fastest in terms of runtime.
  This conclusion is consistent with the theory:
  - Since Ver1 has Theta(n**2) complexity, its execution time increases with n regardless of InpType.
  - In all other versions expected complexities are Theta(n*logn). So, their execution time increases with n regardless of InpType.

3. Change with respect to different InpType (keeping n and Ver fixed):

- InpType4 performs the best. InpType1 performs the worst.
  This conclusion means that if we increase the frequency of elements (keeping n fixed), our execution time decreases. This is consistent with the theory:
  - In the algorithm it can be seen that, regardless of the version, the increasing frequency of elements causes less iterations in the Rearrange(L) function's while loops. That is the number of recursive calls remains approximately the same with a decreased cost caused by the decreasing number of operations in the Rearrange(L) function.