

[자료구조 및 알고리즘] Homework#1

Sorting Algorithm Test

전자공학과 21611591 김난희

Step1. Make a randomly permuted list of integers in range [0..N]

(Do with $N = 10k$, $k = 2, 3, 4, 5$)

```
169 def randomNum(k): # 0부터 10의 k승까지 10의 k승 숫자를 random으로 뽑음
170     num_list = random.sample(range(0, 10**(k+1)), 10**(k+1)) # 10**k는 pow(10,k)로도 나타낼 수 있음, 10^k 의미함
171     #print(num_list)
172     return num_list # 랜덤한 수가 담긴 list 반환
```

$k = 2, 3, 4, 5$ 를 넣으면 random한 숫자 list를 반환한다.

Step2~4. Measure the elapsed times for each algorithm with this list

(For radix sort, use base of 8, 16, 32)

Repeat step1 and 2 more than ten times

Measure the avg. and std. of elapsed time of each sorting algorithm

```
7 import random # random 메소드 사용을 위함
8 import time # time을 구함
9 import numpy # mean값과 std 값을 구함
10 from multiprocessing import Pool # cpu multiprocessing
11 from math import log # Radix sort에서 사용하기 위함

174 def testFunc(args): # 인자: 함수, data 수의 지수, base(0은 radix sort 외, 0 제외 숫자는 radix sort의 base
175     Sorting_func_call = args[0] # 함수를 인자로 받아옴
176     k = args[1] # N=10^k, data 개수
177     base = args[2] # radix 정렬의 base = 8, 16, 32 ...
178
179     time_list=[] # 계산한 elapsed time의 리스트, 평균과 표준 편차를 계산하기 위함
180
181     if base==0: #insertion, bubble, selection, shell, merge, quick, heap sort실행
182         for i in range(10):
183             start_time = time.time() # 시작 시간
184             num_list=randomNum(k) # random한 숫자 N=10^k, [0..N]
185             Sorting_func_call(num_list) # Sorting function
186             #print(num_list) # sorting 되었는지 확인
187             end_time=(time.time() - start_time) # 실행(elapsed) 시간 = 현재 시간 - 시작 시간
188             #print(end_time)
189             time_list.append(end_time) # 시간을 리스트로 append
190
191         print("input data: 10^%s -> %18s -> elapsed time mean: %s seconds"
192               % (k, Sorting_func_call.__name__, numpy.mean(time_list))) # 10개 test 값의 평균
193         print("input data: 10^%s -> %18s -> elapsed time std: %s seconds"
194               % (k, Sorting_func_call.__name__, numpy.std(time_list))) # 10개 test 값의 표준편차
195
196     else: #radixSort(unsorted list, base = 8, 16, 32 ...)
```

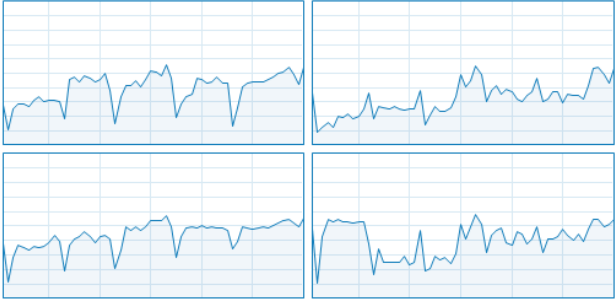
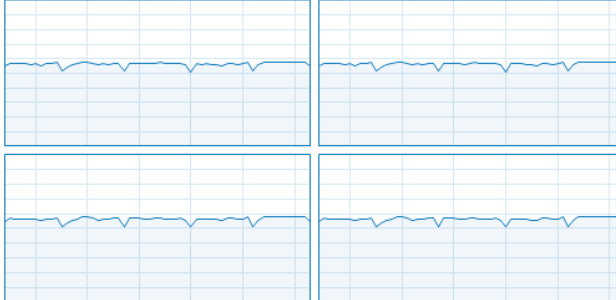
time 모듈을 import하여 time.time함수로 시간을 측정하였다. Sorting 함수를 인자로 받아와서 함수 실행 앞뒤로 시간 함수를 실행하였다. 10번을 측정하여 거기에 대한 mean 값과 std 값을 print 함수로 출력하였다.

10번 테스트한 것의 mean 값과 std 값을 적절히 잘 출력하는지 확인하기 위해서 따로 엑셀로 확인을 해보았다.

<pre> C:\Windows\system32\cmd.exe 0.0034716129302978516 0.0019741058349609375 0.0034723281860351562 0.0014879703521728516 0.0019745826721191406 0.0019843578338623047 0.002256631851196289 0.0011949539184570312 0.001954793930053711 input data: 10^2 -> bubbleSort -> elapsed time mean: 0.0021259307861328123 seconds input data: 10^2 -> bubbleSort -> elapsed time std: 0.0007359838252623744 seconds 계속하려면 아무 키나 누르십시오 . . . input data: 10^2 -> bubbleSort -> elapsed time mean: 0.0021259307861328123 seconds input data: 10^2 -> bubbleSort -> elapsed time std: 0.0007359838252623744 seconds </pre>	<table> <tr><td>elapsed time 1</td><td>0.00148797</td></tr> <tr><td>elapsed time 2</td><td>0.003471613</td></tr> <tr><td>elapsed time 3</td><td>0.001974106</td></tr> <tr><td>elapsed time 4</td><td>0.003472328</td></tr> <tr><td>elapsed time 5</td><td>0.00148797</td></tr> <tr><td>elapsed time 6</td><td>0.001974583</td></tr> <tr><td>elapsed time 7</td><td>0.001984358</td></tr> <tr><td>elapsed time 8</td><td>0.002256632</td></tr> <tr><td>elapsed time 9</td><td>0.001194954</td></tr> <tr><td>elapsed time 10</td><td>0.001954794</td></tr> <tr><td>AVERAGE</td><td>0.002125931</td></tr> <tr><td>STDEVP</td><td>0.000735984</td></tr> </table>	elapsed time 1	0.00148797	elapsed time 2	0.003471613	elapsed time 3	0.001974106	elapsed time 4	0.003472328	elapsed time 5	0.00148797	elapsed time 6	0.001974583	elapsed time 7	0.001984358	elapsed time 8	0.002256632	elapsed time 9	0.001194954	elapsed time 10	0.001954794	AVERAGE	0.002125931	STDEVP	0.000735984
elapsed time 1	0.00148797																								
elapsed time 2	0.003471613																								
elapsed time 3	0.001974106																								
elapsed time 4	0.003472328																								
elapsed time 5	0.00148797																								
elapsed time 6	0.001974583																								
elapsed time 7	0.001984358																								
elapsed time 8	0.002256632																								
elapsed time 9	0.001194954																								
elapsed time 10	0.001954794																								
AVERAGE	0.002125931																								
STDEVP	0.000735984																								
<p><Bubble sorting을 예시로 10번 측정한 것의 mean과 std가 적절한지 확인용으로 출력></p>	<p><엑셀로 확인></p>																								

프로그램이 mean과 std 연산을 잘하는 것을 확인했다.

다음으로 전체 Sorting Method의 mean과 std를 측정해보았다. 측정할 때는 시간이 오래 걸리는 것을 조금이나마 단축하기 위해서 multiprocessing을 이용하였다. 가지고 있는 CPU core 4개를 돌아가며 돌리도록 하였다.

<p>CPU Intel(R) Core(TM) i5-7500 CPU @ 3.40GHz</p> <p>60초 간 이용률(%) 100%</p> 	<p>CPU Intel(R) Core(TM) i5-7500 CPU @ 3.40GHz</p> <p>60초 간 이용률(%) 100%</p> 
<p><Multiprocessing 사용 안함></p>	<p><Multiprocessing 사용함></p>

진행 상황에서 계속적으로 각 Core 사용량을 보았을 때 특별히 다른 점을 찾지는 못했지만, 대체로 Core 4개를 사용한 multiprocessing은 사용량 변화가 크게 일어나지 않고 비슷하게 유지했다.

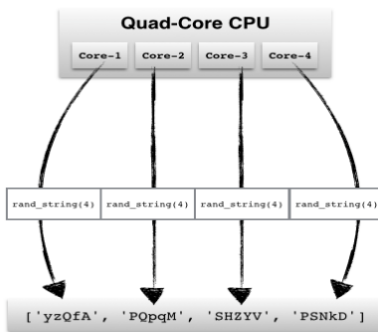
```

210 def main(): # multiprocessing 위주의 구현 # test function은 따로 분리(testFunc(args))
211     pool = Pool(processes=4) # cpu 4 core 사용
212     for k in range(2,6): # input data: 10^2, 10^3, 10^4, 10^5
213         pool.map(testFunc, [[bubbleSort,k,0],[insertionSort,k,0],[selectionSort,k,0], # cpu에게 일을 던져줌
214                             [shellSort,k,0],[mergeSort,k,0],[quickSort,k,0],
215                             [heapSort,k,0],[radixSort,k,8],[radixSort,k,16],[radixSort,k,32]])
216         # pool.map(testFunc,[[bubbleSort,k,0]]) # test 1 sorting method
217     pool.close() # 병렬 처리가 끝났을 때, 프로세스 종료
218     pool.join() # 작업자 프로세스가 종료 될 때까지 기다림
219
220 if __name__ == '__main__':
221     main() # main()문 실행

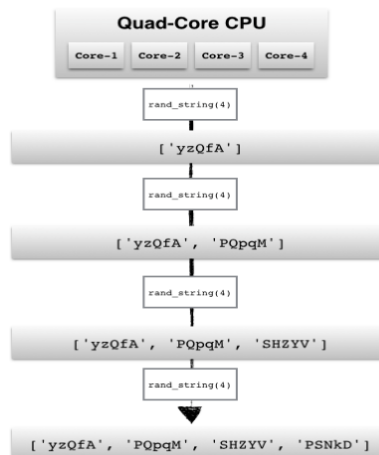
```

MultiProcessing 처리 구조 - 출처

[parallel processing]



[serial processing]



위의 사용한 소스 코드
처럼 병렬 처리를 한 경
우에는,
각 Sorting Method가
Core 4개 각각에 배정된
다. 일이 먼저 끝난 Core
는 다음 일을 가져가도록
되어있다.

또한 Sorting의 elapsed time mean과 std를 얻을 때, 속도가 더욱 단축되었다. 일정 시간이 지났을 때, multiprocessing을 사용하지 않은 경우는 아직 data가 10^4개가 들어간 sorting을 진행 중이었지만, multiprocessing을 사용한 경우는 data가 10^5개를 진행하고 있었다.

다음은 각 Sorting Method를 10번 측정한 것의 average와 standard deviation을 출력한 결과이다.

```

input data: 10^2 -> insertionSort -> elapsed time mean: 0.0007935762405395508 seconds
input data: 10^2 -> insertionSort -> elapsed time std: 0.00024309248234742327 seconds
input data: 10^2 -> bubbleSort -> elapsed time mean: 0.0014879941940307618 seconds
input data: 10^2 -> bubbleSort -> elapsed time std: 0.00022156507939300227 seconds
input data: 10^2 -> selectionSort -> elapsed time mean: 0.0006402730941772461 seconds
input data: 10^2 -> selectionSort -> elapsed time std: 0.00022602089472459368 seconds
input data: 10^2 -> shellSort -> elapsed time mean: 0.0004442453384399414 seconds
input data: 10^2 -> shellSort -> elapsed time std: 0.00014912494355106453 seconds
input data: 10^2 -> mergeSort -> elapsed time mean: 0.0005456209182739258 seconds
input data: 10^2 -> mergeSort -> elapsed time std: 0.00018773906885472474 seconds
input data: 10^2 -> quickSort -> elapsed time mean: 0.00039682388305664065 seconds
input data: 10^2 -> quickSort -> elapsed time std: 0.0001653454143741795 seconds
input data: 10^2 -> heapSort -> elapsed time mean: 0.001636838912963867 seconds
input data: 10^2 -> heapSort -> elapsed time std: 0.002285279419444204 seconds
input data: 10^2 -> radixSort(base 16) -> elapsed time mean: 0.00046901702880859376 seconds
input data: 10^2 -> radixSort(base 16) -> elapsed time std: 0.00011015128262026332 seconds
input data: 10^2 -> radixSort(base 8) -> elapsed time mean: 0.0014880180358886718 seconds
input data: 10^2 -> radixSort(base 8) -> elapsed time std: 0.0021571526990589733 seconds
input data: 10^2 -> radixSort(base 32) -> elapsed time mean: 0.0004826784133911133 seconds
input data: 10^2 -> radixSort(base 32) -> elapsed time std: 0.00024111515616524848 seconds
input data: 10^3 -> shellSort -> elapsed time mean: 0.009275174140930176 seconds
input data: 10^3 -> shellSort -> elapsed time std: 0.005309889087606218 seconds
input data: 10^3 -> mergeSort -> elapsed time mean: 0.014830374717712402 seconds
input data: 10^3 -> mergeSort -> elapsed time std: 0.01566547855913338 seconds
input data: 10^3 -> quickSort -> elapsed time mean: 0.00932481288909912 seconds
input data: 10^3 -> quickSort -> elapsed time std: 0.009382911354997015 seconds
input data: 10^3 -> heapSort -> elapsed time mean: 0.018454265594482423 seconds
input data: 10^3 -> heapSort -> elapsed time std: 0.012652888788071215 seconds
input data: 10^3 -> selectionSort -> elapsed time mean: 0.09190638065338134 seconds
input data: 10^3 -> selectionSort -> elapsed time std: 0.02321721505522194 seconds
input data: 10^3 -> radixSort(base 8) -> elapsed time mean: 0.017508769035339357 seconds
input data: 10^3 -> radixSort(base 8) -> elapsed time std: 0.00951155835154665 seconds
input data: 10^3 -> radixSort(base 16) -> elapsed time mean: 0.00981760025024414 seconds
input data: 10^3 -> radixSort(base 16) -> elapsed time std: 0.00517185222841689 seconds
input data: 10^3 -> radixSort(base 32) -> elapsed time mean: 0.005555152893066406 seconds
input data: 10^3 -> radixSort(base 32) -> elapsed time std: 0.0021254738245849608 seconds
input data: 10^3 -> insertionSort -> elapsed time mean: 0.1304973840713501 seconds
input data: 10^3 -> insertionSort -> elapsed time std: 0.026403618504283275 seconds
input data: 10^3 -> bubbleSort -> elapsed time mean: 0.1967132568359375 seconds
input data: 10^3 -> bubbleSort -> elapsed time std: 0.05360966503478645 seconds
input data: 10^4 -> shellSort -> elapsed time mean: 0.13029897212982178 seconds
input data: 10^4 -> shellSort -> elapsed time std: 0.02259656334477429 seconds
input data: 10^4 -> mergeSort -> elapsed time mean: 0.14939496517181397 seconds
input data: 10^4 -> mergeSort -> elapsed time std: 0.01924054752730427 seconds
input data: 10^4 -> quickSort -> elapsed time mean: 0.09602546691894531 seconds
input data: 10^4 -> quickSort -> elapsed time std: 0.01483661212828955 seconds
input data: 10^4 -> heapSort -> elapsed time mean: 0.20350849628448486 seconds
input data: 10^4 -> heapSort -> elapsed time std: 0.013512467589823223 seconds
input data: 10^4 -> radixSort(base 8) -> elapsed time mean: 0.19100112915039064 seconds
input data: 10^4 -> radixSort(base 8) -> elapsed time std: 0.01093887107326599 seconds
input data: 10^4 -> radixSort(base 16) -> elapsed time mean: 0.1438894748687744 seconds
input data: 10^4 -> radixSort(base 16) -> elapsed time std: 0.00858632783000334 seconds
input data: 10^4 -> radixSort(base 32) -> elapsed time mean: 0.10976464748382568 seconds
input data: 10^4 -> radixSort(base 32) -> elapsed time std: 0.015451816418939586 seconds
input data: 10^4 -> selectionSort -> elapsed time mean: 8.339485263824463 seconds
input data: 10^4 -> selectionSort -> elapsed time std: 0.5824785488749183 seconds
input data: 10^4 -> insertionSort -> elapsed time mean: 12.214381170272826 seconds
input data: 10^4 -> insertionSort -> elapsed time std: 1.203583735362794 seconds
input data: 10^4 -> bubbleSort -> elapsed time mean: 20.5910649061203 seconds
input data: 10^4 -> bubbleSort -> elapsed time std: 3.5274354624692994 seconds
input data: 10^5 -> shellSort -> elapsed time mean: 2.9834856748580934 seconds
input data: 10^5 -> shellSort -> elapsed time std: 0.10225072500843962 seconds
input data: 10^5 -> mergeSort -> elapsed time mean: 1.9725894212722779 seconds
input data: 10^5 -> mergeSort -> elapsed time std: 0.08736559940187619 seconds
input data: 10^5 -> quickSort -> elapsed time mean: 1.1794333696365356 seconds
input data: 10^5 -> quickSort -> elapsed time std: 0.06799339843180044 seconds
input data: 10^5 -> heapSort -> elapsed time mean: 2.849367594718933 seconds
input data: 10^5 -> heapSort -> elapsed time std: 0.2379597821835177 seconds
input data: 10^5 -> radixSort(base 8) -> elapsed time mean: 2.6800334215164185 seconds
input data: 10^5 -> radixSort(base 8) -> elapsed time std: 0.10787553794583508 seconds
input data: 10^5 -> radixSort(base 16) -> elapsed time mean: 2.279817008972168 seconds
input data: 10^5 -> radixSort(base 16) -> elapsed time std: 0.12699628460139825 seconds
input data: 10^5 -> radixSort(base 32) -> elapsed time mean: 1.8099513053894043 seconds
input data: 10^5 -> radixSort(base 32) -> elapsed time std: 0.1095635950220972 seconds
input data: 10^5 -> selectionSort -> elapsed time mean: 509.5974875688553 seconds
input data: 10^5 -> selectionSort -> elapsed time std: 150.72864978426145 seconds
input data: 10^5 -> insertionSort -> elapsed time mean: 586.2726405620575 seconds
input data: 10^5 -> insertionSort -> elapsed time std: 160.02232523930644 seconds
input data: 10^5 -> bubbleSort -> elapsed time mean: 1045.4703704357148 seconds
input data: 10^5 -> bubbleSort -> elapsed time std: 223.87833715975827 seconds

```

계속하려면 아무 키나 누르십시오 . . .

input data: 10^2 -> insertionSort -> elapsed time mean: 0.0007935762405395508 seconds
input data: 10^2 -> insertionSort -> elapsed time std: 0.00024309248234742327 seconds
input data: 10^2 -> bubbleSort -> elapsed time mean: 0.0014879941940307618 seconds
input data: 10^2 -> bubbleSort -> elapsed time std: 0.00022156507939300227 seconds
input data: 10^2 -> selectionSort -> elapsed time mean: 0.0006402730941772461 seconds
input data: 10^2 -> selectionSort -> elapsed time std: 0.00022602089472459368 seconds
input data: 10^2 -> shellSort -> elapsed time mean: 0.0004442453384399414 seconds
input data: 10^2 -> shellSort -> elapsed time std: 0.00014912494355106453 seconds
input data: 10^2 -> mergeSort -> elapsed time mean: 0.0005456209182739258 seconds
input data: 10^2 -> mergeSort -> elapsed time std: 0.00018773906885472474 seconds
input data: 10^2 -> quickSort -> elapsed time mean: 0.00039682388305664065 seconds
input data: 10^2 -> quickSort -> elapsed time std: 0.0001653454143741795 seconds
input data: 10^2 -> heapSort -> elapsed time mean: 0.001636838912963867 seconds
input data: 10^2 -> heapSort -> elapsed time std: 0.002285279419444204 seconds
input data: 10^2 -> radixSort(base 16) -> elapsed time mean: 0.00046901702880859376 seconds
input data: 10^2 -> radixSort(base 16) -> elapsed time std: 0.00011015128262026332 seconds
input data: 10^2 -> radixSort(base 8) -> elapsed time mean: 0.0014880180358886718 seconds
input data: 10^2 -> radixSort(base 8) -> elapsed time std: 0.0021571526990589733 seconds
input data: 10^2 -> radixSort(base 32) -> elapsed time mean: 0.0004826784133911133 seconds
input data: 10^2 -> radixSort(base 32) -> elapsed time std: 0.00024111515616524848 seconds
input data: 10^3 -> shellSort -> elapsed time mean: 0.009275174140930176 seconds
input data: 10^3 -> shellSort -> elapsed time std: 0.005309889087606218 seconds
input data: 10^3 -> mergeSort -> elapsed time mean: 0.014830374717712402 seconds
input data: 10^3 -> mergeSort -> elapsed time std: 0.015665478559913338 seconds
input data: 10^3 -> quickSort -> elapsed time mean: 0.00932481288909912 seconds
input data: 10^3 -> quickSort -> elapsed time std: 0.009382911354997015 seconds
input data: 10^3 -> heapSort -> elapsed time mean: 0.018454265594482423 seconds
input data: 10^3 -> heapSort -> elapsed time std: 0.012652888788071215 seconds
input data: 10^3 -> selectionSort -> elapsed time mean: 0.09190638065338134 seconds
input data: 10^3 -> selectionSort -> elapsed time std: 0.02321721505522194 seconds
input data: 10^3 -> radixSort(base 8) -> elapsed time mean: 0.017508769035339357 seconds
input data: 10^3 -> radixSort(base 8) -> elapsed time std: 0.00951155835154665 seconds
input data: 10^3 -> radixSort(base 16) -> elapsed time mean: 0.00981760025024414 seconds
input data: 10^3 -> radixSort(base 16) -> elapsed time std: 0.00517185222841689 seconds
input data: 10^3 -> radixSort(base 32) -> elapsed time mean: 0.005555152893066406 seconds
input data: 10^3 -> radixSort(base 32) -> elapsed time std: 0.0021254738245849608 seconds
input data: 10^3 -> insertionSort -> elapsed time mean: 0.1304973840713501 seconds
input data: 10^3 -> insertionSort -> elapsed time std: 0.026403618504283275 seconds
input data: 10^3 -> bubbleSort -> elapsed time mean: 0.1967132568359375 seconds
input data: 10^3 -> bubbleSort -> elapsed time std: 0.05360966503478645 seconds
input data: 10^4 -> shellSort -> elapsed time mean: 0.13029897212982178 seconds
input data: 10^4 -> shellSort -> elapsed time std: 0.02259656334477429 seconds
input data: 10^4 -> mergeSort -> elapsed time mean: 0.14939496517181397 seconds
input data: 10^4 -> mergeSort -> elapsed time std: 0.01924054752730427 seconds

```
input data: 10^4 -> quickSort -> elapsed time mean: 0.09602546691894531 seconds
input data: 10^4 -> quickSort -> elapsed time std: 0.01483661212828955 seconds
input data: 10^4 -> heapSort -> elapsed time mean: 0.20350849628448486 seconds
input data: 10^4 -> heapSort -> elapsed time std: 0.013512467589823223 seconds
input data: 10^4 -> radixSort(base 8) -> elapsed time mean: 0.19100112915039064 seconds
input data: 10^4 -> radixSort(base 8) -> elapsed time std: 0.01093887107326599 seconds
input data: 10^4 -> radixSort(base 16) -> elapsed time mean: 0.1438894748687744 seconds
input data: 10^4 -> radixSort(base 16) -> elapsed time std: 0.00858632783000334 seconds
input data: 10^4 -> radixSort(base 32) -> elapsed time mean: 0.10976464748382568 seconds
input data: 10^4 -> radixSort(base 32) -> elapsed time std: 0.015451816418939586 seconds
input data: 10^4 -> selectionSort -> elapsed time mean: 8.339485263824463 seconds
input data: 10^4 -> selectionSort -> elapsed time std: 0.5824785488749183 seconds
input data: 10^4 -> insertionSort -> elapsed time mean: 12.214381170272826 seconds
input data: 10^4 -> insertionSort -> elapsed time std: 1.203583735362794 seconds
input data: 10^4 -> bubbleSort -> elapsed time mean: 20.5910649061203 seconds
input data: 10^4 -> bubbleSort -> elapsed time std: 3.5274354624692994 seconds
input data: 10^5 -> shellSort -> elapsed time mean: 2.9834856748580934 seconds
input data: 10^5 -> shellSort -> elapsed time std: 0.10225072500843962 seconds
input data: 10^5 -> mergeSort -> elapsed time mean: 1.9725894212722779 seconds
input data: 10^5 -> mergeSort -> elapsed time std: 0.08736559940187619 seconds
input data: 10^5 -> quickSort -> elapsed time mean: 1.1794333696365356 seconds
input data: 10^5 -> quickSort -> elapsed time std: 0.06799339843180044 seconds
input data: 10^5 -> heapSort -> elapsed time mean: 2.849367594718933 seconds
input data: 10^5 -> heapSort -> elapsed time std: 0.2379597821835177 seconds
input data: 10^5 -> radixSort(base 8) -> elapsed time mean: 2.6800334215164185 seconds
input data: 10^5 -> radixSort(base 8) -> elapsed time std: 0.10787553794583508 seconds
input data: 10^5 -> radixSort(base 16) -> elapsed time mean: 2.279817008972168 seconds
input data: 10^5 -> radixSort(base 16) -> elapsed time std: 0.12699628460139825 seconds
input data: 10^5 -> radixSort(base 32) -> elapsed time mean: 1.8099513053894043 seconds
input data: 10^5 -> radixSort(base 32) -> elapsed time std: 0.1095635950220972 seconds
input data: 10^5 -> selectionSort -> elapsed time mean: 509.5974875688553 seconds
input data: 10^5 -> selectionSort -> elapsed time std: 150.72864978426145 seconds
input data: 10^5 -> insertionSort -> elapsed time mean: 586.2726405620575 seconds
input data: 10^5 -> insertionSort -> elapsed time std: 160.02232523930644 seconds
input data: 10^5 -> bubbleSort -> elapsed time mean: 1045.4703704357148 seconds
input data: 10^5 -> bubbleSort -> elapsed time std: 223.87833715975827 seconds
```

Step5. Make a table(mean, std of sorting algorithms)

Table 1. **Mean** of Sorting Algorithms(단위: seconds)

Sort	100	1K	10K	100K
Insertion	0.00079	0.13050	12.21438	586.27264
Bubble	0.00149	0.19671	20.59106	1045.47037
Selection	0.00064	0.09191	8.33949	509.59749
Shell	0.00044	0.00928	0.13030	2.98349
Merge	0.00055	0.01483	0.14939	1.97259
Quick	0.00040	0.00932	0.09603	1.17943
Heap	0.00164	0.01845	0.20351	2.84937
Radix/8	0.00149	0.01751	0.19100	2.68003
Radix/16	0.00047	0.00982	0.14389	2.27982
Radix/32	0.00048	0.00556	0.10976	1.80995

For radix sort, use base of 8, 16, 32

Table 2. **Standard deviation** of Sorting Algorithms(단위: seconds)

Sort	100	1K	10K	100K
Insertion	0.00024	0.02640	1.20358	160.02233
Bubble	0.00022	0.05361	3.52744	223.87834
Selection	0.00023	0.023217	0.58248	150.72865
Shell	0.00015	0.00531	0.02260	0.10225
Merge	0.00019	0.01567	0.01924	0.08737
Quick	0.00017	0.00938	0.01484	0.06799
Heap	0.00229	0.01265	0.01351	0.23796
Radix/8	0.00216	0.00951	0.01094	0.10788
Radix/16	0.00011	0.00517	0.00859	0.12700
Radix/32	0.00024	0.00213	0.01545	0.10956

For radix sort, use base of 8, 16, 32

Step6. Analyze and explain the results

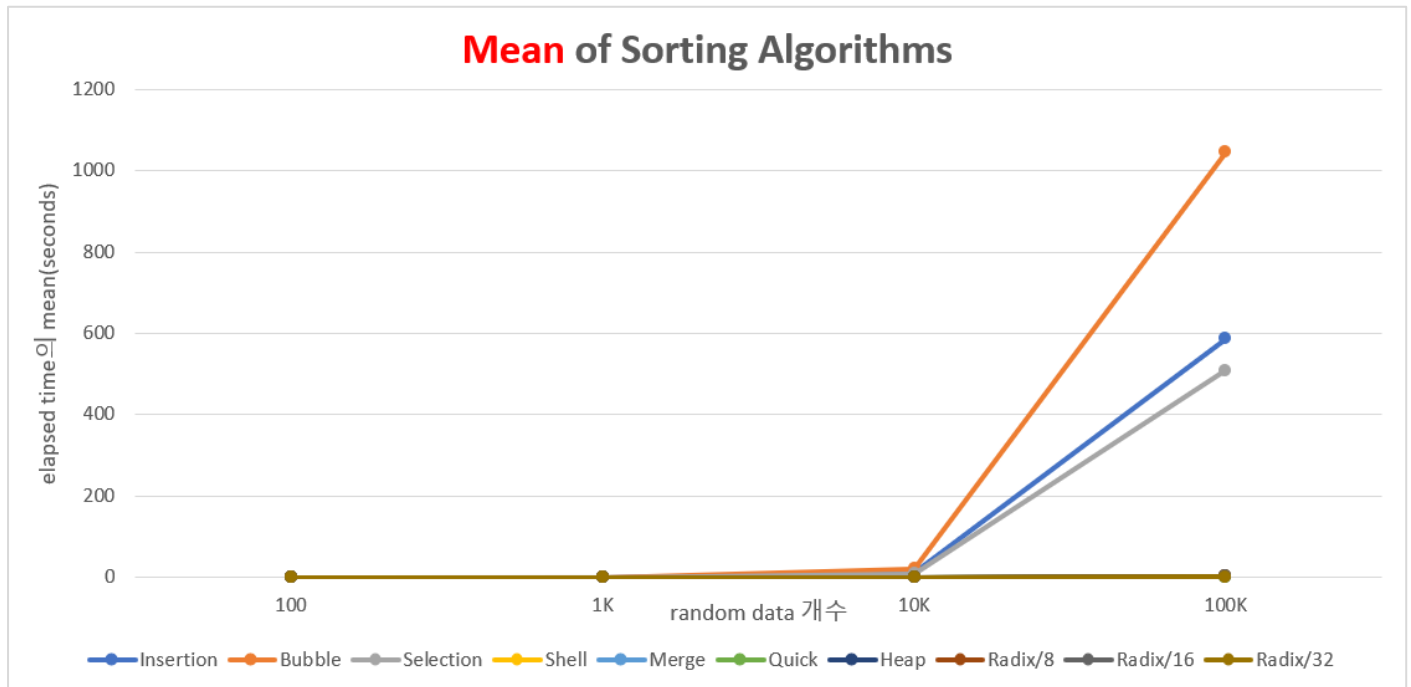


그림 1. 모든 Sorting Method에서 각각의 mean(10번 측정한 elapsed time의 평균)

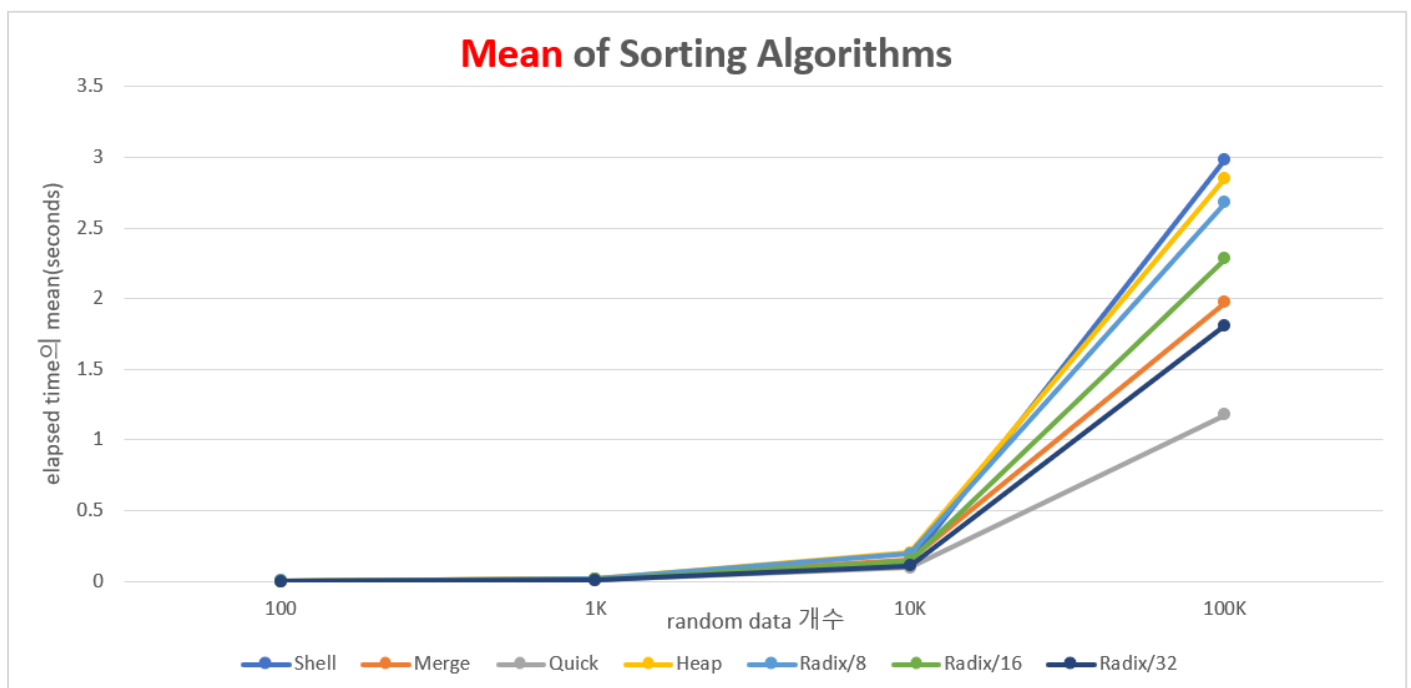


그림 2. $O(n^2)$ 을 제외한 Sorting Method에서 각각의 mean(10번 측정한 elapsed time의 평균)

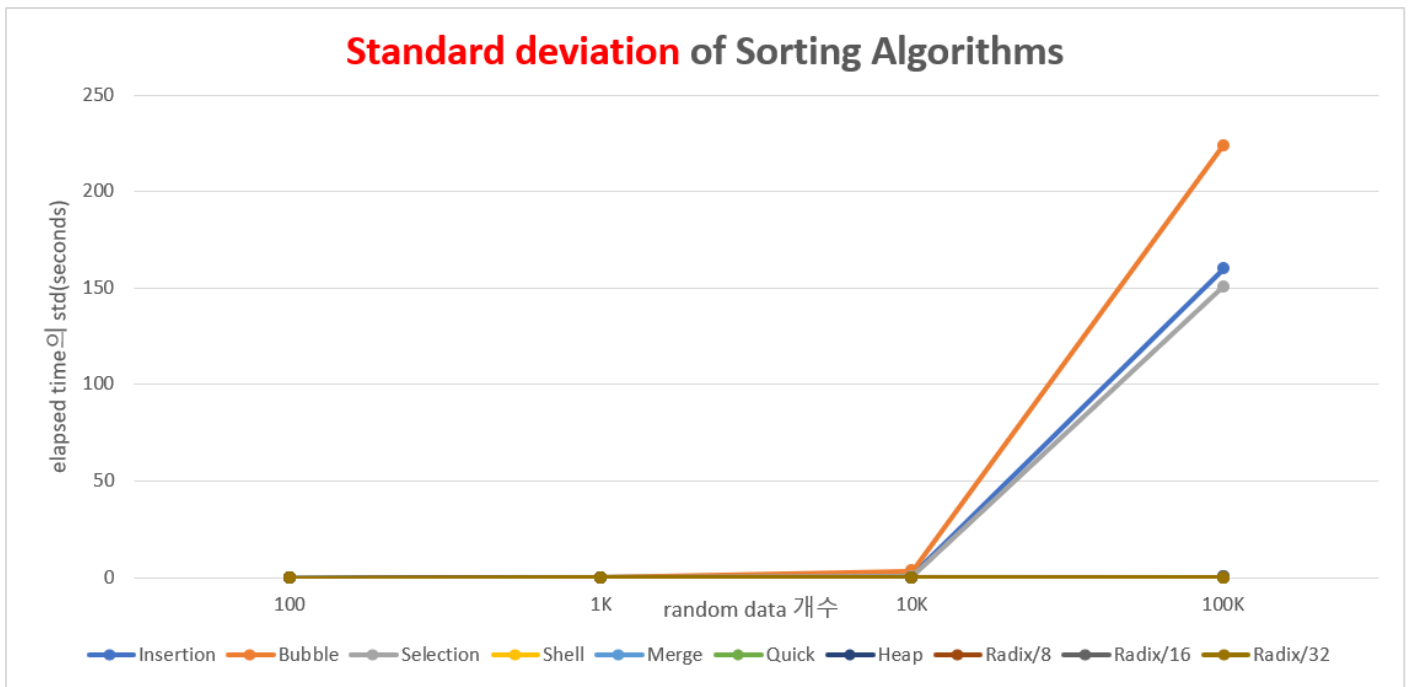


그림 3. 모든 Sorting Method에서 각각의 std(10번 측정한 elapsed time의 표준편차)

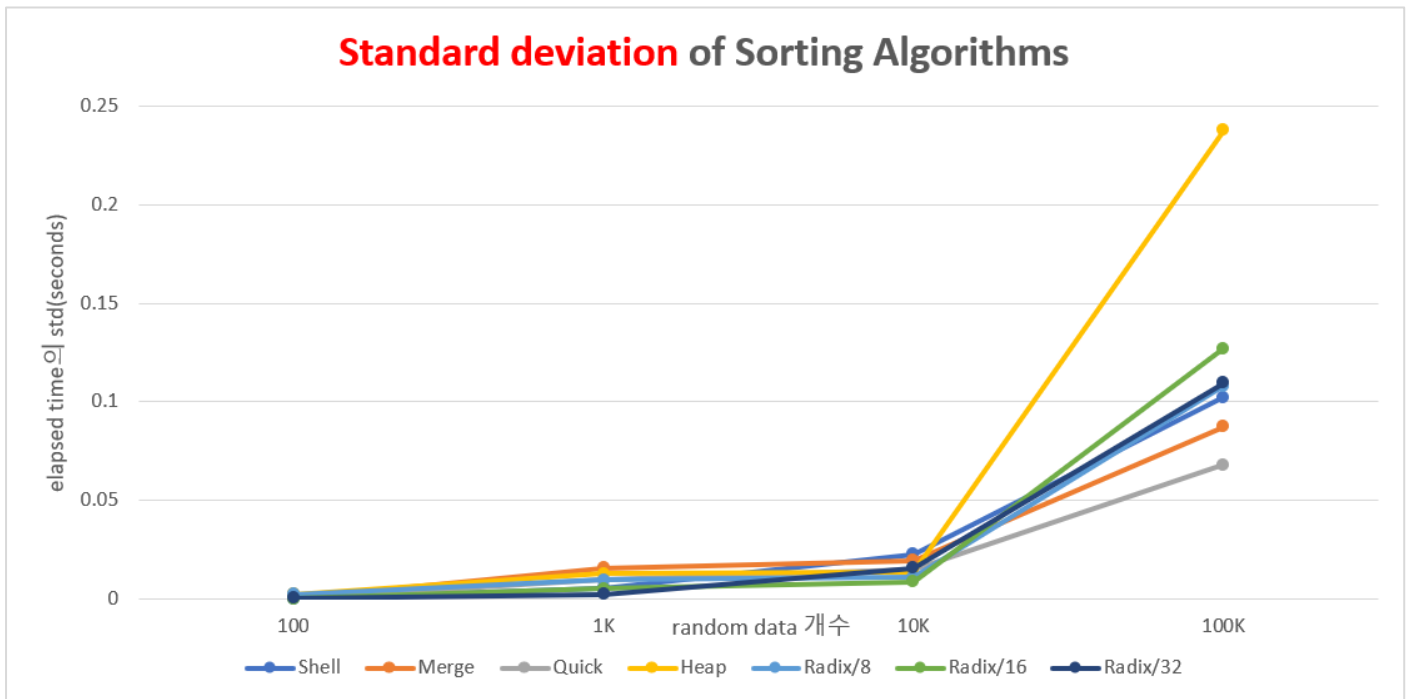


그림 4. $O(n^2)$ 을 제외한 Sorting Method에서 각각의 std(10번 측정한 elapsed time의 표준편차)

Table 3. 각 Sorting 알고리즘의 특성

Sort	algorithm	complexity	stability
Bubble	Exchanging two neighbor items	$O(n^2)$	stable
Selection	Finding the maximum of the remained items	$O(n^2)$	stable

Insertion	Keeping the previous list sorted	$O(n^2)$	stable
Shell	Sorting sub-lists by gab	$\sim O(n^{1.5})$	stable
Merge	Splitting list and merging sub-lists recursively (additional memory)	$O(n\log(n))$	stable
Quick	Splitting list according to pivot value and swapping recursively	Avg.: $O(n\log(n))$ Worst case: $O(n^2)$	Not stable
Heap	Utilizing max-heap (additional memory)	$O(n\log(n))$	Not stable
Radix	Hashing from LSB to MSB (additional memory)	$\sim O(c * n)$ $O(n\log(n))$	Stable (LSB)

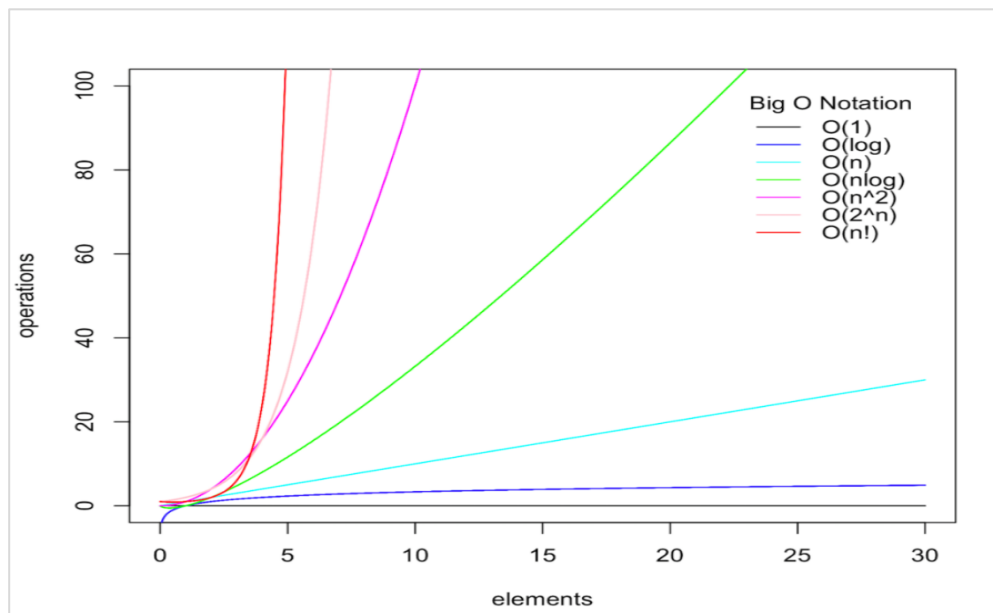


그림 5. Big O Notation: common function

complexity와 elapsed time은 관계가 있다. complexity가 큰 것이 무조건 느린 것은 아니다. complexity는 데이터가 증가했을 때, 얼마만큼 느려지는가에 의미가 있다.

데이터가 충분히 커졌을 때, sorting 알고리즘 간의 차이가 확연히 나타나기 때문에, 충분히 큰 데이터 수 1k와 10k를 가지고 비교를 해보았다.

<Bubble, Selection, Insertion sort>

우선 complexity가 $O(n^2)$ 인 bubble sort에 데이터 1k를 넣었을 때는 10번의 테스트를 평균 낸 elapsed time은 0.19671 seconds, 10k를 넣었을 때 20.59106 seconds이다. 데이터가 10배 증가할 때 n^2 인 100배만큼 증가한 elapsed time의 평균임을 볼 수 있다.

$$\text{Bubble sort}(O(n^2)): 0.19671\text{seconds}(1k \text{ data}) \times 10^2 = 19.671\text{seconds} \\ \approx 20.59106\text{seconds}(10k \text{ data})$$

Insertion, selection sort도 마찬가지이다.

$$\text{Insertion sort}(O(n^2)): 0.13050\text{seconds}(1k \text{ data}) \times 10^2 = 13.050\text{seconds} \\ \approx 12.21438\text{seconds}(10k \text{ data}) \\ \text{Selection sort}(O(n^2)): 0.09191\text{seconds}(1k \text{ data}) \times 10^2 = 9.191\text{seconds} \\ \approx 8.33949\text{seconds}(10k \text{ data})$$

Mean of Sorting Algorithms의 그래프(그림 1)를 보면, 데이터의 크기가 커질 수록 elapsed time이 가파르게 증가한다. bubble sort가 기울기가 가장 크다. 다음으로 insertion sort, selection sort 순으로 가파르게 증가한다. bubble sort는 수합이 빈번한 반면, selection sort는 비교만 하고 가장 큰 수와 한번씩만 수합하기 때문에 같은 n^2 의 complexity를 가져도 실행 시간이 짧다. insertion sort를 linked list로 구현했더라면, 시간이 많이 걸리지 않았을 것이다. array based로 구현했기에 복사를 하면서 옮겨야 하므로, 그만큼 시간이 더 걸린다.

<Shell sort>

shell sort는 gap size에 따라 complexity가 달라지기는 하지만, 평균적으로 $O(n^{1.5})$ 를 따른다. shell sort는 삽입 정렬의 개선한 정렬로, complexity가 최대 $O(n^2)$ 을 갖는다. 현재 알려진 바로는 $O(n \log^2(n))$ 이 최선이라고 한다. 이 사이에서 complexity가 나타날 것이다.

1k일 때 0.00928 seconds로, 10k일 때는 0.13030 seconds로 데이터가 10배 증가할 때 $n^{1.1473}$ 인 $10^{1.1473}$ 만큼 elapsed time의 평균이 증가한 것을 볼 수 있다. 10k가 0.13030 seconds에서 100k는 2.98349 seconds로 데이터가 10배 증가할 때 $n^{1.35978}$ 인 $10^{1.35978}$ 만큼 elapsed time의 평균이 증가한 것을 볼 수 있다.

$$\text{shell sort}(O(n^2)): 0.00928\text{seconds}(1k \text{ data}) \times 10^{1.1473} = 0.13030\text{seconds}(10k \text{ data}) \\ 0.13030\text{seconds}(10k \text{ data}) \times 10^{1.35978} = 2.98349\text{seconds}(100k \text{ data})$$

만약 최선인 $O(n \log^2(n))$ 으로 계산을 해본다면, 1k일 때 0.00928 seconds는 데이터가 10배 증가한다면, 10k일 때 0.0928 seconds가 나온다. 실제 0.13030seconds보다는 빠르지만, 근사한 값인 것을 알 수 있다.

$$\text{shell sort 최선}(O(n \log^2(n))): 0.00928\text{seconds}(1k \text{ data}) \times 10 \log^2 10 = 0.0928\text{seconds} \\ \leq 0.13030\text{seconds}(10k \text{ data})$$

데이터를 더 크게 하여 실험해서, shell sort의 complexity가 평균적으로 $O(n^{1.5})$ 을 따른다는 것을 확인해보면 좋을 것 같다.

<Merge, Quick, Heap, Radix>

complexity가 보통 $n \log n$ 이라고 볼 수 있는 나머지 4가지 Sorting Method를 본다.

merge sort는 item을 절반으로 계속 나눈 후, 하나씩만 남으면 다시 sorting하면서 합친다. 그러므로 $n \log_2^n$ 이라 볼 수 있다. 나머지 sorting 방법들도 같은 $n \log n$ 이지만 정확하게는 밑이 2가 아니다. Quick sort는 pivot을 어떻게 정하냐, sort할 array가 이미 sorting되어있느냐에 따라 complexity가 달라진다. Radix sort는 밑이 base에 따라 달라진다. 그러므로 각각의 sorting method들은 같은 $n \log n$ 의 complexity를 가져도 elapsed time에 대해서는 차이가 있다.

merge sort는 밑을 2로 가지지만, 보통은 밑을 2로 가지든, 10으로 가지든 값이 비슷하므로 보통의 log를 씌워 계산을 한다. 1k의 데이터는 0.01483 seconds로 10k일 때 0.14939 seconds로 1k 데이터의 0.01483 seconds에서 10배 증가한 $10 \log 10$ 을 곱하면 10k의 0.14939 seconds와 근사한 값이 나온다.

$$\begin{aligned} \text{merge}(O(n \log n)): 0.01483 \text{seconds}(1k \text{ data}) \times 10 \log 10 &= 0.1483 \text{seconds} \\ &\approx 0.14939 \text{seconds}(10k \text{ data}) \end{aligned}$$

Quick sort도 마찬가지이다. 1k 데이터는 0.00932 seconds에서 10k 데이터는 0.09603 seconds가 나온다. 데이터가 10배 증가했으므로, $n \log n$ 의 값인 $10 \log 10$ 을 곱하면 0.09603 seconds에 근사한 값이 나온다.

$$\begin{aligned} \text{Quick sort}(O(n \log n)): 0.00932 \text{seconds}(1k \text{ data}) \times 10 \log 10 &= 0.0932 \text{seconds} \\ &\approx 0.09603 \text{seconds}(10k \text{ data}) \end{aligned}$$

Heap sort도 계산해보면, 1k 데이터는 0.01845 seconds에서 10k 데이터는 0.20351 seconds이다. 데이터가 10배 증가했으므로, $n \log n$ 의 값인 $10 \log 10$ 을 곱하면 0.20351 seconds에 근사한 값이 나온다.

$$\begin{aligned} \text{Heap sort}(O(n \log n)): 0.01845 \text{seconds}(1k \text{ data}) \times 10 \log 10 &= 0.1845 \text{seconds} \\ &\approx 0.20351 \text{seconds}(10k \text{ data}) \end{aligned}$$

Radix sort도 계산해보면, base가 8일 때, 1k 데이터는 0.01751 seconds에서 10k 데이터는 0.19100 seconds로 데이터가 10배 증가했으므로, $n \log_{base}^n$ 의 값인 $10 \log_8^{10}$ 을 곱하면 0.19100 seconds에 근사한 값이 나온다.

$$\begin{aligned} \text{Radix sort}(O(n \log_{base}^n)), (\text{base}: 8) : 0.01751 \text{seconds}(1k \text{ data}) \times 10 \log_8^{10} &= 0.19389 \text{seconds} \\ &\approx 0.19100 \text{seconds}(10k \text{ data}) \end{aligned}$$

base가 16일 때, 32일때를 계산해보면, 실험 값과 이론 값이 차이가 나지만, base가 증가할 수록 전체적으로 seconds가 줄어드는 것을 보았을 때, 알맞게 나온 것이라 볼 수 있다.

Radix sort($O(n \log_{base}^n)$), (base: 16) : 0.00982seconds(1k data) $\times 10 \log_{16}^{10} = 0.08155seconds$
(no \approx) 0.14389seconds(10k data)

Radix sort($O(n \log_{base}^n)$), (base: 32) : 0.00556seconds(1k data) $\times 10 \log_{32}^{10} = 0.03694seconds$
(no \approx) 0.10976seconds(10k data)

<표준편차>

편차는 '평균으로부터 얼마나 차이나는 값을 얻었는 가'이다.

전체적으로 데이터 크기가 커질수록 확연히 표준편차의 크기도 커졌다.

100k일 때 그래프(그림 3, 4)를 보면 다음의 순서로 표준편차가 큰 것을 볼 수 있다.

Bubble >> Insertion > Selection
>> Heap >> Radix/16 > Radix/32 > Radix/8 > Shell > Merge > Quick

데이터의 크기가 일정할 때, sorting 알고리즘 간의 실행시간 표준편차가 큰 것은 보통 elements의 크기, 배열 상태에 달려있다. data의 상태가 random, reversed, sorted에 따라 비교를 해보았을 때, '각 sorting 알고리즘의 편차가 어떤가'가 의미가 있는 자료라고 생각한다. 현재 자료는 'random한 데이터를 넣었을 때 각 실행시간 간의 편차가 얼마나 있는가'에만 집중하면 된다. 이 상황에서는 일반적으로 complexity가 높은 정렬이 std가 높다. 또한, random한 데이터를 넣었을 경우, random한 내부에서도 어떠한 경우에도 큰 탈이 없는 정렬 기법이 std가 낮게 나올 것이다. 즉 reversed data를 넣어 elapsed time이 크게 나오더라도, random data를 넣어 알고리즘 방법에 상관없이 elapsed time이 대체로 작은 것들은 std가 낮게 나오는 것이다. shell sort는 다른 몇몇 알고리즘보다 elapsed time이 높게 나온 반면, std는 상대적으로 낮게 나왔다. 이는 data의 개수가 일정할 때는 시작할 때 결정하는 gap size도 일정하기 때문이다.

만약 random, reversed, sorted에 따라 편차를 계산해보았다면, quick, insertion, bubble sorting 방법이 편차가 컸을 것이다.

quick는 sorted이거나 reversed data(데이터가 역순)일 때 random data보다 편차가 훨씬 클 것이다. pivot이 정확히 median 값이면 좋지만, 맨 앞의 element를 골랐을 때(현재 사용한 알고리즘은 맨 앞의 elements를 pivot으로 한다), 가장 큰 값이거나 가장 작은 값을 고르면 한 stage마다 item 1개만 고정하기 때문이다. 그래서 모두 n번의 비교와 n번의 stage를 거쳐 $O(n^2)$ 이 된다. 실험에서 quick sort의 std가 다른 알고리즘보다 상대적으로 낮게 나온 것은, random한 데이터를 넣었기 때문에 그렇다.

insertion sort는 데이터가 역순이라면 하나씩 밀어내는 과정을 다 거쳐야한다. 뒤로 미는 작업 때문에 편차가 크게 나올 것이다.

bubble sort는 이미 sorted되어있다면 비교만 하고, 수합은 넘어가도 되기 때문이다. 수합을 하면서 reversed data > random data > sorted data 순으로 elapsed time이 발생했을 것이다.

*** 전체 소스 코드***

```
"""
자료구조 및 알고리즘
Homework#1
21611591 김난희
"""

import random                # random 메소드 사용을 위함
import time                  # time을 구함
import numpy                 # mean값과 std 값을 구함
from multiprocessing import Pool # cpu multiprocessing
from math import log         # Radix sort에서 사용하기 위함


def swap(x, i, j):           # swap function
    x[i], x[j] = x[j], x[i]  # 원소 교환


# * * * * * Bubble Sort (버블정렬) * * * * * # 0(N^2)
def bubbleSort(x):           # original bubble sort: short bubble이 아닌 원본 버전
    for size in reversed(range(len(x))): # 크기를 하나씩 줄여가며 반복
        for i in range(size):    # 리스트의 크기만큼 반복
            if x[i] > x[i + 1]:    # 현재의 index 값이 다음 index의 값보다 크면 참 (비교: N-1, N-2, ..., 1)
                swap(x, i, i + 1) # 위치 교환


# * * * * * Selection Sort (선택정렬) * * * * * # 0(N^2)
def selectionSort(x):
    for size in reversed(range(len(x))): # max 수합하면 하나 제외하고 찾는 for문
        max_i = 0                        # max 초기값 0
        for i in range(1, 1+size):      # max 찾는 for문 # max 수합 후 범위에서 제외하고 다시 범위를 줌
            if x[i] > x[max_i]:          # 현재 index 값이 max로 되어있는 값보다 크다면 (비교: N-1번)
                max_i = i               # max에는 현재 index를 넣어줌
        swap(x, max_i, size)           # max를 수합하여 이미 정렬된 것중 제일 앞에 수와 수합 (수합: 1번)


# * * * * * Insertion Sort (삽입정렬) * * * * * # 0(N^2)
def insertionSort(x):             # 통계적으로 bubble이나 selection보다
    # 비교연산이 적음
    for size in range(1, len(x)): # 리스트 크기만큼 반복
        val = x[size]             # 처음 index 부터 시작함 # 현재 읽고 있는 값
        i = size                  # 현재 읽고 있는 index
        while i > 0 and x[i-1] > val: # 첫 번째 index 다음 부터 시작 # 현재 값이 이전 값보다 크면
            x[i] = x[i-1]         # 현재 값에 이전 값을 넣고
            i -= 1                # index를 하나 감소 시켜서 # 또다시 이전 값과 비교하여 while문
    반복
    x[i] = val                    # 크기가 알맞은 위치에 삽입


# * * * * * Shell Sort (셸 정렬) * * * * * # average 0(N^1.5)
def InsertionSort(x, start, gap): # 삽입정렬 구현
    for target in range(start+gap, len(x), gap): # (시작인덱스+차이, 리스트 크기만큼 반복, 차이까지)
        val = x[target]                # 리스트의 값
        i = target                     # 인덱스 저장
        while i > start:                # 증감 값 보다 인덱스가 크다면 반복
            if x[i-gap] > val:          # 리스트의 비교 인덱스 값 보다 크다면
                x[i] = x[i-gap]        # 해당 인덱스 값 할당
            else:                      # 리스트의 비교 인덱스 값 보다 작다면
                break                  # 반복 중지
```

```

        i -= gap                # 중간 값만큼 빼주기
        x[i] = val              # 해당 값 삽입

def shellSort(x):               # 본 shell sort
    gap = len(x) // 2          # 리스트를 2로 나눈 몫 (중간 값) 취함
    while gap > 0:
        for start in range(gap): # 중간 값의 크기만큼 반복
            InsertionSort(x, start, gap) # 삽입정렬 메소드 호출 (리스트, 증감 값, 중간 값)
        gap = gap // 2          # 리스트를 2로 나눈 몫 (중간 값) 취함 (반으로 줄어나간다.)

# * * * * * Merge Sort (병합 정렬) * * * * * # 0(nlog(n))
def mergeSort(x):
    if len(x) > 1:              # 배열의 길이가 1보다 클 경우 재귀함수 호출 반복
        mid = len(x) // 2       # 2로 나눈 몫 (중간 값) 취함
        lx, rx = x[:mid], x[mid:] # 중간 기준으로 왼쪽(lx), 오른쪽(rx) split
        mergeSort(lx)           # left sublist의 값을 기준으로 병합 정렬 재귀 호출
        mergeSort(rx)           # right sublist의 값을 기준으로 병합정렬 재귀 호출

        li, ri, i = 0, 0, 0     # left sublist, right sublist, merged list
        while li < len(lx) and ri < len(rx): # 두 sublist 길이만큼 돌린다. # sublist의 정렬이 끝날 때 까지
            if lx[li] < rx[ri]:   # left sublist의 현재 값이 right sublist의 현재 값보다 작다면
                x[i] = lx[li]     # left sublist의 현재 값을 merged list에 넣음
                li += 1           # left sublist의 index 증가(다음 값 검사를 위해)
            else:                # left sublist의 현재 값이 right sublist의 현재 값보다 크다면
                x[i] = rx[ri]     # right sublist의 현재 값을 merged list에 넣음
                ri += 1           # right sublist의 index 증가(다음 값 검사를 위해)
            i += 1               # merged list에 값을 넣은 후 다음 값을 넣기 위해 index 증가
        x[i:] = lx[li:] if li != len(lx) else rx[ri:] # 두 sublist 중 모든 원소가 merged list에 들어가서 더
        # 남은 원소가 있는 sublist를 다시 x에 넣고 merge
    sort함

# * * * * * Quick Sort (퀵 정렬) * * * * * # Average: 0(nlog(n), Worst(이미 정렬된
list): 0(n^2)
def pivotFirst(x, lmark, rmark): # left, right mark를 둔 method
    pivot_val = x[lmark]         # left mark(첫 index부터 시작)의 값을 pivot
    value으로
    pivot_idx = lmark            # left mark(첫 index부터 시작)의 index를 pivot
    index로
    while lmark <= rmark:        # left와 right mark가 cross 되기 전까지
        while lmark <= rmark and x[lmark] <= pivot_val: # pivot value보다 작으면 넘어감 # pivot value보다
        큰 수 찾음
            lmark += 1           # index를 증가시켜 넘어감(오른쪽으로)
        while lmark <= rmark and x[rmark] >= pivot_val: # pivot value보다 크면 넘어감 # pivot value보다
        작은 수 찾음
            rmark -= 1           # index를 감소시켜 넘어감(왼쪽으로)
        if lmark <= rmark:      # left가 pivot보다 큰 수 찾고, right가 pivot보다
        작은 수 찾고, no cross 이면
            swap(x, lmark, rmark) # left와 right mark의 값 수합
            lmark += 1           # index 다음으로 넘어가며 검사
            rmark -= 1
        swap(x, pivot_idx, rmark) # left와 right mark가 cross한 상황에서 pivot과
        right mark 수합
    return rmark                 # 수합한 right mark index 위치 반환

def quickSort(x, pivotMethod=pivotFirst): # left, right mark를 둔 method를 사용하여 퀵 정렬
    def _qsort(x, first, last):
        if first < last:
            splitpoint = pivotMethod(x, first, last) # 위의 pivotFirst 함수를 통해 split point(수합한
            right mark 위치)를 정함

```

```

        _qsort(x, first, splitpoint-1)                # 재귀 호출 (리스트, 시작 인덱스, 수합한 수 index-1)
왼쪽 절반
        _qsort(x, splitpoint+1, last)                # 재귀 호출 (리스트, 수합한 수 index+1, 종료인덱스)
오른쪽 절반
        _qsort(x, 0, len(x)-1)                      # _qsort 함수 호출

# * * * * * Radix Sort (기수 정렬) * * * * * # 0(nlog(n)), ideal: 0(n)
def get_digit(number, d, base):                    ## 현재 자릿수(d)와 진법(base)에 맞는 숫자 변환
    return (number // base ** d) % base            # //는 몫 연산자(그 다음 연산) **는 지수 연산자(우선
연산)
                                                    # ex) number = 102, d = 0, base = 10 -> 첫 번째 자리수에
해당하는 2 찾음
def counting_sort_with_digit(A, d, base):          ## 자릿수 기준으로 counting sort # A: input array # d :
현재 자릿수 # ex) 102, d = 0 : 2
    k = base - 1                                    # k: maximum value of A # ex) 10진수의 최대값 = 9
    B = [-1] * len(A)                              # B: output array # init with -1
    C = [0] * (k + 1)                              # C: count array # init with zeros

    for a in A:                                     ## 현재 자릿수를 기준으로 빈도수 세기
        C[get_digit(a, d, base)] += 1              # 숫자를 하나씩 카운팅
    for i in range(k):                             ## C 업데이트
        C[i + 1] += C[i]                          # 각 요소값에 직전 요소값을 더함
    for j in reversed(range(len(A))):              ## 현재 자릿수를 기준으로 정렬
        B[C[get_digit(A[j], d, base)] - 1] = A[j] # A 요소값의 역순으로 B를 채워 넣음
        C[get_digit(A[j], d, base)] -= 1          # 자리를 채운 후 해당하는 값에서 1 소모
    return B

def radixSort(list, base=10):
    digit = int(log(max(list), base) + 1) # 입력된 리스트 가운데 최대값의 자릿수(digit) 확인
    for d in range(digit):                       # 자릿수 별로 counting sort
        list = counting_sort_with_digit(list, d, base)
    return list

# * * * * * Heap Sort (힙 정렬) * * * * * # 0(nlog(n))
def heapify(x, index, heap_size):                # heap 구조를 유지하는 함수 # heap 성질
만족하도록
    largest = index                                # 배열의 중간부터 시작
    left_index = 2 * index + 1                    # 왼쪽 자식노드의 index를 left_index
    right_index = 2 * index + 2                   # 오른쪽 자식노드의 index를 right_index
    if left_index < heap_size and x[left_index] > x[largest]: # 왼쪽 자식 노드 값 > 부모 노드 값
        largest = left_index                     # 왼쪽 자식 노드 index를 largest에 저장
    if right_index < heap_size and x[right_index] > x[largest]: # 오른쪽 자식 노드 값 > 부모 노드 값
        largest = right_index                   # 오른쪽 자식 노드 index를 largest에 저장
    if largest != index:                          # 본래 부모 노드 index와 달라졌다면
        swap(x, largest, index)                 # 위치를 바꿈
        heapify(x, largest, heap_size)          # 바꾼 자식 노드에서 다시 힙 성질
유지하는지 검사

def heapSort(x):
    n = len(x)

    ### BUILD-MAX-HEAP: 주어진 원소들로 최대 힙 구성
    # 최초 힙 구성시 배열의 중간부터 시작하면
    # 이진트리 성질에 의해 모든 요소값을 서로 한번씩 비교할 수 있게 됨 : 0(n)
    for i in range(n // 2 - 1, -1, -1): # 위에서 아래로 heapify 수행 # index : (n을 2로 나눈 몫-1)~0
        heapify(x, i, n)

    ### Recurrent: max heap의 root(최댓값)과 마지막 요소 교환 + 새 root 노드에 대한 max heap 구성
    # 한번 힙이 구성되면 개별 노드는 최악의 경우에도 트리의 높이(log(n)) 만큼의 자리 이동을 하게 됨
    # 이런 노드들이 n개 있으므로 : 0(nlog(n))

```



```

for i in range(n - 1, 0, -1):
    swap(x, 0, i)
    heapify(x, 0, i)
return x

def randomNum(k):
    # 0부터 10의 k승까지 10의 k승 숫자를
    random으로 뽑음
    num_list = random.sample(range(0, 10**(k+1)), 10**(k+1))
    # 10**k는 pow(10,k)로도 나타낼 수 있음,
    10^k 의미함
    #print(num_list)
    return num_list
    # 랜덤한 수가 담긴 list 반환

def testFunc(args):
    # 인자: 함수, data 수의 지수, base(0은 radix sort 외, 0 제외 숫자는 radix
    sort의 base)
    Sorting_func_call = args[0] # 함수를 인자로 받아옴
    k = args[1] # N=10^k, data 개수
    base = args[2] # radix 정렬의 base = 8, 16, 32 ...

    time_list=[] # 계산한 elapsed time의 리스트, 평균과 표준 편차를 계산하기 위함

    if base==0: #insertion, bubble, selection, shell, merge, quick, heap sort실행
        for i in range(10):
            start_time = time.time() # 시작 시간
            num_list=randomNum(k) # random한 숫자 N=10^k, [0..N]
            Sorting_func_call(num_list) # Sorting function
            #print(num_list) # sorting 되었는지 확인
            end_time=(time.time() - start_time) # 실행(elapsed) 시간 = 현재 시간 - 시작 시간
            #print(end_time)
            time_list.append(end_time) # 시간을 리스트로 append

        print("input data: 10^%s -> %18s -> elapsed time mean: %s seconds"
              % (k, Sorting_func_call.__name__, numpy.mean(time_list))) # 10개 test 값의 평균
        print("input data: 10^%s -> %18s -> elapsed time std: %s seconds"
              % (k, Sorting_func_call.__name__, numpy.std(time_list))) # 10개 test 값의 표준편차

    else: #radixSort(unsorted list, base = 8, 16, 32 ...)
        for i in range(10):
            start_time = time.time() #시작 시간
            num_list=randomNum(k) # random한 숫자 N=10^k, [0..N]
            num_list=Sorting_func_call(num_list,base) # Sorting function
            #print(num_list) # sorting 되었는지 확인
            end_time=(time.time() - start_time) # 실행(elapsed) 시간 = 현재 시간 - 시작 시간
            time_list.append(end_time) # 시간을 리스트로 append

        print("input data: 10^%s -> %8s(base %2s) -> elapsed time mean: %s seconds"
              % (k, Sorting_func_call.__name__, base, numpy.mean(time_list))) # 10개 test 값의 평균
        print("input data: 10^%s -> %8s(base %2s) -> elapsed time std: %s seconds"
              % (k, Sorting_func_call.__name__, base, numpy.std(time_list))) # 10개 test 값의 표준편차

def main():
    # multiprocessing 위주의 구현 # test function은 따로 분리(testFunc(args))
    pool = Pool(processes=4) # cpu 4 core 사용
    for k in range(2,6): # input data: 10^2, 10^3, 10^4, 10^5
        pool.map(testFunc,[[bubbleSort,k,0],[insertionSort,k,0],[selectionSort,k,0],
        # cpu에게 일을
        # shellSort,k,0],[mergeSort,k,0],[quickSort,k,0],
        # heapSort,k,0],[radixSort,k,8],[radixSort,k,16],[radixSort,k,32]])
        # pool.map(testFunc,[[bubbleSort,k,0]]) # test 1 sorting method
    pool.close() # 병렬 처리가 끝났을 때, 프로세스 종료
    pool.join() # 작업자 프로세스가 종료될 때까지 기다림
if __name__ == '__main__':
    main() # main()문 실행

```

*** 수업 자료 외 참고 자료 ***

-python

-삽입, 버블, 선택, 퀵, 병합, 쉘, 기수 정렬 및 내장 정렬 메소드

<https://seongjaemoon.github.io/python/2017/12/16/pythonSort.html>

-버블, 선택, 삽입, 쉘, 병합, 퀵 정렬 및 데이터를 사용한 실험

<http://ejklike.github.io/2017/03/04/sorting-algorithms-with-python.html>

-힙 정렬

<https://ratsgo.github.io/data%20structure&algorithm/2017/09/27/heapsort>

-radix(기수) 정렬

<https://ratsgo.github.io/data%20structure&algorithm/2017/10/16/countingsort/>

-MultiProcessing(Pool)을 사용한 처리

<https://beomi.github.io/2017/07/05/HowToMakeWebCrawler-with-Multiprocess/>

<https://niceman.tistory.com/145>

<https://stackoverflow.com/questions/38271547/when-should-we-call-multiprocessing-pool-join>

-주석처리 방법

<https://mainia.tistory.com/5598>

-time 함수 더 자세하게

<https://python.bakyeono.net/chapter-11-3.html>

-함수를 변수로 넘겨주기

<https://brownbears.tistory.com/107>

-평균, 분산, 표준편차 구하기

<https://m.blog.naver.com/PostView.nhn?blogId=zerosum99&logNo=120194362128&proxyReferer=https%3A%2F%2Fwww.google.com%2F>

-변수 출력시에 공간 채우기

<https://hashcode.co.kr/questions/227/%EC%8A%A4ED%8A%B8%EB%A7%81-%EC%95%9E%EC%97%90-0-%EC%B1%84%EC%9A%B0%EA%B8%B0>

-연산자 **

<https://jythonbook-ko.readthedocs.io/en/latest/OpsExpressPF.html>

#-그래프로 그리기

#<https://ordo.tistory.com/68>

#<https://wikidocs.net/4760>

-C++

-난수 생성, 선택, 삽입, 버블, 병합, 퀵 정렬

<https://hsp1116.tistory.com/34>

-난수 생성, 선택, 삽입, 버블, 쉘, 병합, 퀵 정렬

<https://m.blog.naver.com/bieemiho92/220359947709>

-버블, 선택, 삽입, 쉘, 퀵, 힙, 병합, 기수 정렬

<https://yaraba.tistory.com/79>

-시간측정: time 함수

<https://hijuworld.tistory.com/1>

-thread 관련 개념 및 예제

<https://modoocode.com/269>