Solutions to Homework 5

22C:044 Algorithms, Fall 2000

1(a) Let n be the number of elements in the input array. On the first call to `Random` a random number is selected among n possible numbers. On the second call, one number among remaining $n-1$ numbers is selected, and so on. All in all, the program has

$$n! = n(n-1)(n-2)\dots$$

possible choices to behave. This number is the same as the total number of different permutations of the array. Each permutation is a possible outcome of the algorithm, therefore each permutation is produced by a unique sequence of random choices, and has probability $1/n!$.

1(b)-(d) See the enclosed program file.

1(e) Here are the outputs of the algorithms:

```
Old QuickSort :  50644.5, 194705.1, 442027.8, 782592.9, 1217313.1,
1744565.6, 2391603.8, 3092663.2, 3910503.2, 4874686.0, 5916152.8,
6927976.0, 8226759.2, 9502844.8, 10903734.4, 12381534.4, 14057915.2,
15735019.2, 17440283.2, 19458699.2
New QuickSort :  4000.1, 8042.0, 12124.0, 16092.8, 20118.2,
24158.3, 28165.1, 32218.8, 36288.6, 40296.0, 44292.8, 48237.3,
52434.0, 56493.9, 60394.8, 64555.5, 68521.1, 72540.9, 76672.3,
80618.7
```

According to theory, the complexity of the original quicksort algorithm should be $\Theta(n^2)$ because the "almost sorted" arrays are worst-case instances to quicksort: the partition is always unbalanced with at most c elements in the lower part. Indeed, the ratios of the numbers of comparisons above divided by n^2 are

```
0.202578, 0.194705, 0.196457, 0.195648, 0.194770, 0.193841,
0.195233, 0.193291, 0.193111, 0.194987, 0.195575, 0.192444,
0.194716, 0.193936, 0.193844, 0.193461, 0.194573, 0.194259,
```

`0.193244, 0.194587`
that is, they are near constant 0.2. In other words, it seems that $T(n) \approx 0.2n^2$.

The complexity of the new algorithm should be $\Theta(n)$. The ratios of the numbers of comparisons to n are `8.00020, 8.04200, 8.08267, 8.04640, 8.04728, 8.05277, 8.04717, 8.05470, 8.06413, 8.05920, 8.05324, 8.03955, 8.06677, 8.07056, 8.05264, 8.06944, 8.06131, 8.06010, 8.07077, 8.06187`
These are also near constant at 8, so $T(n) \approx 8n$.

2(a) The algorithm maintains a heap of k elements, containing the largest unprocessed element from each input array. The root of the heap is always the largest unprocessed element in all the arrays. It is removed from the heap and moved to the merged list. It is replaced in the heap by the next largest element from the same input list that contained the old root. Then the root is heapified, and the process is repeated.

2(b) Let us start a new run at every element that is smaller than the previous element in the array. Finding runs this way takes $\Theta(n)$ time. The runs are sorted arrays, so using the result from (a) the runs can be merged in $\Theta(n \log k)$ time.

3 If we divide the array into blocks of three elements then the array is guaranteed to contain at least

$$2 \times \lfloor (\lceil n/3 \rceil - 1)/2 - 1 \rfloor \geq 2((n/3 - 1)/2 - 2) = n/3 - 5$$

elements that are smaller than the median of the medians of the 3-blocks. So there are at most $2n/3 + 5$ elements in both parts after the partitioning. We obtain the recurrence

$$T(n) \leq T(n/3 + 1) + T(2n/3 + 5) + \Theta(n)$$

for the running time of the selection algorithm. This recurrence does not give the linear time complexity we obtained when 5-blocks were used. On the other hand, this recurrence does not prove the time complexity would not be linear, either. Solving the recurrence (which was not required in the exercise) gives $T(n) = O(n \log n)$. This does not mean that $T(n)$ can not be also $\Theta(n)$...