**2.3.1** Show, in the style of the trace given with partition(), how that method partitions the array E A S Y Q U E S T I O N.

E A S Y Q U E S T I O N

E A **S** Y Q U **E** S T I O N

E A **E** **Y** Q U S S T I O N

E A **E** Y Q U S S T I O N

**2.3.2** Show, in the style of the quicksort trace given in this section, how quicksort sorts the array E A S Y Q U E S T I O N (for the purposes of this exercise, ignore the initial shuffle).

E A S Y Q U E S T I O N

Quicksort(0,11)

Partition (0,11) -> 2

**E A** **E** **Y Q U S S T I O N**

Quicksort(0,1)

Partition (0, 1) -> 1

**A** **E** E Y Q U S S T I O N

Quicksort(0, 0)

Quicksort(2, 1)

Quicksort(3, 11)

Partition(3,11) -> 11

A E E **Q U S S T I O N** **Y**

Quicksort(3, 10)

Partition(3,10) -> 6

A E E **I N O** **Q** **T S S U Y**

Quicksort(3, 5)

Partition(3, 5) ->3

A E E **I N O** Q T S S U Y

Quicksort(3,2)

Quicksort(4,5)

Partition(4,5) ->5

A E E I **O** **N** Q T S S U Y

Quicksort(4,4)

Quicksort(6,5)

Quicksort(7,10)

Partition(7, 10) -> 9

A E E I O N Q **S S** **T** **U** Y

Quicksort(7, 8)

A E E I O N Q **S** **S** T U Y

Quicksort(10,10)

Quicksort(12,11)

Result **A E E I O N Q S S T U Y**

**2.3.3** What is the maximum number of times during the execution of Quick.sort() that the largest item can be exchanged, for an array of length *N* ?

~lnN

In the worst case for Quicksort, the largest item will not be exchanged at all or only once.

So the largest item can only be exchanged at average case where it is exchanged at every partition procedure so, the maximum number will be ~lnN

**2.3.4** Suppose that the initial random shuffle is omitted. Give six arrays of ten elements for which Quick.sort() uses the worst-case number of compares.

0 1 2 3 4 5 6 7 8 9

9 8 7 6 5 4 3 2 1 0

0 9 8 7 6 5 4 3 2 1

9 0 1 2 3 4 5 6 7 8

0 9 1 2 3 4 5 6 7 8

9 0 8 7 6 5 4 3 2 1

**2.3.5** Give a code fragment that sorts an array that is known to consist of items having just two distinct keys.

static void Sort2KeyArrayDutchNationalFlag(int[] a)

{

int lt = 0, i = 1, gt = a.Length - 1;

int piv = a[0];

while (i <= gt)

{

if (a[i] < piv) Exch(a, lt++, i++);

else if (a[i] > piv) Exch(a, i, gt--);

else i++;

}

}

Or

static void Sort2KeyArray(int[] a)

{

int key = a[0];

int n = a.Length;

int i = 0;

while (i < n && a[++i] == key)

;

if (i == n)

return;

if (a[i] < key)

Sort2KeyArray(a, 0, i);

else

Sort2KeyArray(a, i, i + 1);

}

static void Sort2KeyArray(int[] a, int low, int high)

{

int n = a.Length, i = low;

int key = a[low];

for (int j = high; j < n; j++)

if (a[j] < key)

Exch(a, i++, j);

}

**2.3.6** Write a program to compute the exact value of *CN*, and compare the exact value with the approximation 2*N* ln *N*, for *N* = 100, 1,000, and 10,000.

**2.3.7** Find the expected number of subarrays of size 0, 1, and 2 when quicksort is used to sort an array of *N* items with distinct keys. If you are mathematically inclined, do the math; if not, run some experiments to develop hypotheses.

**2.3.8** About how many compares will Quick.sort() make when sorting an array of *N* items that are all equal?

By algorithm 2.5 in the book, it will take ~NlgN compares, for

while(true)

{

while(a[++i].CompareTo(piv) **<** 0 ) if(i == high) break;

while(a[--j].CompareTo(piv) **>** 0) if(i==low) break;

…

}

**2.3.9** Explain what happens when Quick.sort() is run on an array having items with just two distinct keys, and then explain what happens when it is run on an array having just three distinct keys.

When N is large, with only 2 or 3 distinct keys the partition method will not fall into worst case unless all its left side or right side are equal. Either way the compares used will be guaranteed to be ~NlgN.

**2.3.10** *Chebyshev’s inequality* says that the probability that a random variable is more

than *k* standard deviations away from the mean is less than 1/*k* 2. For *N* = 1 million, use

Chebyshev’s inequality to bound the probability that the number of compares used by

quicksort is more than 100 billion (.1 *N* 2).

**2.3.11** Suppose that we scan over items with keys equal to the partitioning item’s key instead of stopping the scans when we encounter them. Show that the running time of this version of quicksort is quadratic for all arrays with just a constant number of distinct keys.

**2.3.12** Show, in the style of the trace given with the code, how the entropy-optimal sort

first partitions the array B A B A B A B A C A D A B R A.

**2.3.13** What is the *recursive depth* of quicksort, in the best, worst, and average cases?

This is the size of the stack that the system needs to keep track of the recursive calls. See

Exercise 2.3.20 for a way to guarantee that the recursive depth is logarithmic in the

worst case.

**2.3.14** Prove that when running quicksort on an array with *N* distinct items, the probability

of comparing the *i* th and *j* th largest items is 2/(*j* \_ *i*). Then use this result to

prove Proposition K.

2.3.15 Nuts and bolts. (G. J. E. Rawlins) You have a mixed pile of N nuts and N bolts and need to quickly find the corresponding pairs of nuts and bolts. Each nut matches exactly one bolt, and each bolt matches exactly one nut. By fitting a nut and bolt together, you can see which is bigger, but it is not possible to directly compare two nuts or two bolts. Give an efficient method for solving the problem.

void MatchNutsBolts (Pile<Nut> nuts, Pile<Bolt> bolts, Bag<Pair<Nut, Bolt>> result)

{

If(nuts.Count == 1)

{

result.add(nuts.pickOne(), bolts.pickOne());

return;

}

Nut nut = nuts.pickOne();

Pile<Bolt> smallerBots = new Pile<Bolt>();

Pile<Bolt> largerBots = new Pile<Bolt>();

Bolt matchedBolt;

foreach(Bolt b in bolts)

{

If(b.match(nut) < 0)

smallerBots.add(b);

else if(b.match(nut) >0)

largerBots.add(b);

else

matchedBolt = b;

}

result.add(nut, matchedBolt);

bolts.pickUp(matchedBolt);

Pile<Nut> smallerNuts = new Pile<Nut>();

Pile<Nut> largerNuts = new Pile<Nut>();

foreach(Nut n in nuts)

{

If(n.match(matchedBolt) < 0)

smallerNuts.add(n);

else if(n.match(matchedBolt) >0)

largerNuts.add(n);

}

MatchNutsBolts(smallerNuts, smallerBolts);

MatchNutsBolts(largerNuts, largerBolts);

}

2.3.16 Best case. Write a program that produces a best-case array (with no duplicates) for sort() in Algorithm 2.5: an array of N items with distinct keys having the property that every partition will produce subarrays that differ in size by at most 1 (the same subarray sizes that would happen for an array of N equal keys). (For the purposes of this exercise, ignore the initial shuffle.)

**static** **int**[] produceBestCaseArray(**int** n)

{

**int**[] a = **new** **int**[n];

**for**(**int** i = 0 ; i < n ; i++)

a[i] = i;

*produceBestCaseArray*(a, 0, n-1);

**return** a;

}

**static** **void** produceBestCaseArray(**int**[] a, **int** low, **int** high)

{

**if**(low >= high)

**return**;

**int** mid = low + (high - low) / 2;

*produceBestCaseArray*(a, low, mid - 1);

*produceBestCaseArray*(a, mid + 1, high);

*exch*(a, low, mid);

}

Chose “exch” the only substantial operation as cost model.

En = 2E(n/2) + 1

Let’s say n = 2^m

E(2^m) = 2 \* E(2^(m-1)) + 1

= 4 \* E(2^(m-2)) + 1 + 2

…

= 2^m \* E(1) + 1 + 2 + … + 2^(m-1)

= 2^m – 1 = n - 1

E(n) = n – 1;

2.3.17 Sentinels. Modify the code in Algorithm 2.5 to remove both bounds checks in the inner while loops. The test against the left end of the subarray is redundant since the partitioning item acts as a sentinel (v is never less than a[lo]). To enable removal of the other test, put an item whose key is the largest in the whole array into a[length-1] just after the shuffle. This item will never move (except possibly to be swapped with an item having the same key) and will serve as a sentinel in all subarrays involving the end of the array. Note : When sorting interior subarrays, the leftmost entry in the subarray to the right serves as a sentinel for the right end of the subarray.

static void QuickSortWithSentinel(double[] a)

{

int max = 0;

for (int i = 0; i < a.Length; i++)

if (a[i] > a[max])

max = i;

Exch(a, a.Length - 1, max);

QuickSortWithSentinel(a, 0, a.Length - 1);

}

static void QuickSortWithSentinel(double[] a, int low, int high)

{

if (low < high)

{

int p = PartitionWithSentinel(a, low, high);

QuickSortWithSentinel(a, low, p - 1);

QuickSortWithSentinel(a, p + 1, high);

}

}

static int PartitionWithSentinel(double[] a, int low, int high)

{

int i = low, j = high + 1;

double piv = a[low];

while (true)

{

while (a[++i] < piv) ;

while (a[--j] > piv) ;

if (i >= j)

break;

Exch(a, i, j);

}

Exch(a, low, j);

return j;

}

2.3.18 Median-of-3 partitioning. Add median-of-3 partitioning to quicksort, as described in the text (see page 296). Run doubling tests to determine the effectiveness of the change.

static void SwapTheMedianOfThree<T>(T[] a, int low, int high) where T : IComparable<T>

{

if (high - low >= 1)

{

int mid = low + (high - low) / 2;

MedianOfThree(ref a[low], ref a[mid], ref a[high]);

}

}

static void MedianOfThree<T>(ref T low, ref T mid, ref T high) where T : IComparable<T>

{

if (low.CompareTo(high) > 0)

{

if (mid.CompareTo(high) > 0)

{

if (mid.CompareTo(low) <= 0) //low >= mid > high

{

T t = high;

high = low;

low = mid;

mid = t;

}

}

else

Swap(ref low, ref high); //low >= high >= mid

}

else

{

if (mid.CompareTo(high) > 0) //mid > high >= low

{

T t = low;

low = high;

high = mid;

mid = t;

}

else if (mid.CompareTo(low) > 0) //high >= mid > low

Swap(ref low, ref mid);

}

}

static void QuickSortMedianOfThree<T>(T[] a, int low, int high) where T : IComparable<T>

{

SwapTheMedianOfThree(a, low, high);

if (low < high)

{

int p = PartitionWithSentinel(a, low, high);

QuickSortMedianOfThree(a, low, p - 1);

QuickSortMedianOfThree(a, p + 1, high);

}

}

2.3.19 Median-of-5 partitioning. Implement a quicksort based on partitioning on the median of a random sample of five items from the subarray. Put the items of the sample at the appropriate ends of the array so that only the median participates in partitioning. Run doubling tests to determine the effectiveness of the change, in comparison both to the standard algorithm and to median-of-3 partitioning (see the previous exercise). Extra credit: Devise a median-of-5 algorithm that uses fewer than seven compares on any input.

static void QuickSortMedianOfFive<T>(T[] a, int low, int high) where T : IComparable<T>

{

SwapTheMedianOfFive(a, low, high);

if (low < high)

{

int p = PartitionWithSentinel(a, low, high);

QuickSortMedianOfFive(a, low, p - 1);

QuickSortMedianOfFive(a, p + 1, high);

}

}

static void SwapTheMedianOfFive<T>(T[] a, int low, int high) where T : IComparable<T>

{

if (high - low < 4)

SwapTheMedianOfThree(a, low, high);

else

{

int n = high - low + 1;

int step = n / 5;

MedianOfFive(ref a[low], ref a[low + step], ref a[low + 2 \* step], ref a[low + 3 \* step], ref a[high]);

}

}

static void MedianOfFive<T>(ref T a, ref T b, ref T c, ref T d, ref T e) where T : IComparable<T>

{

if (a.CompareTo(b) > 0) //make it (a b)

Swap(ref a, ref b);

if (c.CompareTo(d) > 0) // make it (c d)

Swap(ref c, ref d);

if (a.CompareTo(c) < 0) //if a < c we want to throw c anyway so we need to swap.

{

Swap(ref a, ref c);

Swap(ref b, ref d);

}

//till now, it's (a b) vs. (d).

if (d.CompareTo(e) > 0) //let d less than e anyway

Swap(ref d, ref e);

//till now, it's (a b) vs. (d e)

if (d.CompareTo(a) > 0)

{

Swap(ref a, ref d);

Swap(ref b, ref e);

}

// till now, it's (a b) vs. (e)

if (a.CompareTo(e) > 0)

{

Swap(ref a, ref e); // now a is median

Swap(ref b, ref e); // and e is largest

}

else if (b.CompareTo(e) > 0) // or else a has been the median, we make e the largest anyway if b > e.

Swap(ref b, ref e);

}

**2.3.20** *Nonrecursive quicksort.* Implement a nonrecursive version of quicksort based

on a main loop where a subarray is popped from a stack to be partitioned, and the resulting

subarrays are pushed onto the stack. *Note* : Push the larger of the subarrays onto the stack first, which guarantees that the stack will have at most lg *N* entries.

**2.3.21** *Lower bound for sorting with equal keys.* Complete the first part of the proof

of Proposition M by following the logic in the proof of Proposition I and using the

observation that there are *N*! */ f*1!*f*2! . . . *fk*! different ways to arrange keys with *k* different

values, where the *i* th value appears with frequency *fi* (= *Npi* , in the notation of Proposition

M), with *f*1+. . . +*fk* = *N*.

**2.3.22** *Fast 3-way partitioning.* ( J. Bentley and D. McIlroy) Implement an entropyoptimal

sort based on keeping item's with equal keys at both the left and right ends of the subarray. Maintain indices p and q such that a[lo..p-1] and a[q+1..hi] are all equal to a[lo], an index i such that a[p..i-1] are all less than a[lo], and an index j such that a[j+1..q] are all greater than

a[lo]. Add to the inner partitioning loop code to swap a[i] with a[p] (and increment p) if it is equal to v and to swap a[j] with a[q] (and decrement q) if it is equal to v before the usual comparisons of a[i] and a[j] with v. After the partitioning loop has terminated, add code to swap the items with equal keys into position. *Note* : This code complements the code given in the text, in the sense that it does extra swaps for keys equal to the partitioning item’s key, while the code in the text does extra swaps for keys that are *not* equal to the partitioning item’s key.

**2.3.23** *Java system sort.* Add to your implementation from Exercise 2.3.22 code to use the *Tukey ninther* to compute the partitioning item—choose three sets of three items, take the median of each, then use the median of the three medians as the partitioning item. Also, add a cutoff to insertion sort for small subarrays.

**2.3.24** *Samplesort.* ( W. Frazer and A. McKellar) Implement a quicksort based on using

a sample of size 2*k* \_ 1. First, sort the sample, then arrange to have the recursive

routine partition on the median of the sample and to move the two halves of the rest of

the sample to each subarray, such that they can be used in the subarrays, without having

to be sorted again. This algorithm is called *samplesort*.