Sections 7.1 and 7.2

1) Implement the Insertion Sort algorithm (Algorithm 7.1), run it on your system, and study its best-case, average-case, and worst-case time complexities using several problem instances.

Solution:

Best case: Consider the following input 1, 2, 3, 4, 5 (completely sorted) → The for loop in Alg. 7.1 runs 4 times (n-1), and each time the comparison S[j] > x is evaluated only once → 4 comparisons. For n elements, the best-case needs n-1 comparisons.

Average-case:

Consider the following input 3, 5, 6, 4, 7

for i = 2 → j = 1 ⇒ 3, 5, 6, 4, 7 (1 comparison)

for i = 3 → j = 2 ⇒ 3, 5, 6, 4, 7 (1 comparison)

for i = 4 → j = 3 ⇒ 3, 4, 5, 6, 7 (3 comparisons)

for i = 5 → j = 4 ⇒ 3, 4, 5, 6, 7 (1 comparison)

Total: 6 comparisons, close to the asymptotical one listed in Table 7.1: $A(n) = n^2/4 = 5^2/4 = 6.75$.

Here it would be instructive to use a program to sort several larger arrays (say, with n = 100) of random integers. The pseudocode in Alg. 5.1 can be slightly modified to produce the exact number of comparisons:

```
void insertionsort (int n, keytype S[]) {
        index i , j ;
        keytype x;
        counter = 0;
        for (i = 2; i <= n; i++) {
              x = S[ i ];
              j = i− 1;
              while (j > 0 && S[ j ] > x) {
                 S[ j + 1] = S[ j ];
                 j−−;
                 counter++;
              }
              if (j > 0)              //no comparison if  j == 0
                 counter++;
              S[ j + 1] = x;
        }
        cout << counter;
}
```

When the experiment is repeated many times, the average of the number of comparisons should approach $A(n) = n^2/4 = 10{,}000/4 = 2{,}500$.

Worst-case:
Consider the following input 6, 5, 4, 3, 2 (sorted in reverse order)
for i = 2 → j = 1 ⇒ 5, 6, 4, 3, 2 (1 comparison)
for i = 3 → j = 2 ⇒ 4, 5, 6, 3, 2 (2 comparisons)
for i = 4 → j = 3 ⇒ 3, 4, 5, 6, 2 (3 comparisons)
i = 5 → j = 4 ⇒ 2, 3, 4, 5, 6 (4 comparisons)
Total: 10 comparisons, close to the asymptotical one listed in Table 7.1: $A(n) = n^2/2 = 5^2/2 = 12.5$.

2) Show that the maximum number of comparisons performed by the Insertion Sort algorithm (Algorithm 7.1) is achieved when the keys are input in nonincreasing order.

Solution: On page 289, it is shown that the maximum number of comparisons of Insertion Sort is given by n(n-1)/2. When the keys are in nonincreasing order (i.e., [n, n-1, n-2, …, 2, 1]), then each iteration of the algorithm, starting with i=2, must compare the ith element with each of the elements in the indices less than i (all the way down to index 1). This is because each element at index i will always need to be inserted at the beginning of the array, since the elements to its left, while sorted, are all greater than it by construction of the input. The total number of comparisons done is thus 1 on the first iteration, and increases by 1 on each of the subsequent n-2 iterations, increasing to n-1. This is the sum of the integers from 1 to n-1, which is given by the expression n(n-1)/2, the maximum given on page 289.

3) Show that the worst-case and average-case time complexities for the number of assignments of records performed by the Insertion Sort algorithm (Algorithm 7.1) are given by

$$W(n) = \frac{(n+4)(n-1)}{2} \approx \frac{n^2}{2} \qquad \textit{and} \qquad A(n) = \frac{n(n+7)}{4} - 2 \approx \frac{n^2}{4}$$

Solution: The worst case is when the array is sorted in reverse order, so each insertion is done in the first position of the array. For i = 2, 1 + 1+1 = 3 record assignments are needed, for i = 3, 1 + 2 + 1 = 4, and so on until i = n, where 1 + n – 1 + 1 = n + 1 assignments are needed.
Total: 3 + 4 + … + n + (n+1) = [1 + 2 + 3 + 4 + … + n + (n+1)] – (1 + 2) = (n+1)(n+2)/2 – 3 = (n-1)(n+4)/2 ≈ $n^2/2$.

In the average case:

- Each S[i] has equal probabilities 1/i to be inserted in any of the positions 1 through i.
- Insertion in position k requires: one assignment for x = S[i], (i – k) assignments to shift the elements, and one more for S[k] = x, for a total of (i – j + 2).
- If k = i, no assignments are really needed, since no insertion is performed, but Algorithm 5.1 still performs two useless assignments.
- The average is [2 + 3 + 4 + … + (i – 1) + i + (i+1)] ·1/i =

  [(i – 1)·i/2 – 1 + i +(i +1)] / i = (i – 1)/2 + 2.

- Summing the above from 2 to n, we get A(n) = n(n+7)/4 – 2 ≈ $n^2/4$ for large n.

A quick check can be performed for n = 2: The formula gives A(2) = 2(2+7)/4 – 2 = 2.5. There is a 0.5 chance for the array to be already sorted, in which case two assignments are performed, and a 0.5 chance for the array not to be sorted, in which case there are three assignments. The average is 0.5 x 2 + 0.5 x 3 = 2.5.

4) Show that the worst-case and average-case time complexities for the number of assignments of records performed by the Exchange Sort algorithm (Algorithm 1.3) are given by

$$W(n) = \frac{3n\,(n-1)}{2} \qquad \textit{and} \qquad A(n) = \frac{3n\,(n-1)}{4}$$

Solution: The worst-case occurs when the array is sorted in reverse order, because every comparison results in an exchange. The total number of exchanges is (n-1) + (n-2) + … + 2 + 1 = n(n-1)/2. Each exchange requires three assignments, so we obtain $W(n) = \frac{3n\,(n-1)}{2}$.

In the average case, only half of the comparisons result in exchanges, so we divide the previous expression by 2 to obtain $A(n) = \frac{3n\,(n-1)}{4}$.

5) Compare the best-case time complexities of Exchange Sort (Algorithm 1.3) and Insertion Sort (Algorithm 7.1).

Solution: In either algorithm, the best case is when the array is already sorted.

	Exchange sort	Insertion sort
Best case time complexity of comparisons	$n(n-1)/2 \rightarrow O(n^2)$	$n-1 \rightarrow O(n)$
Best case time complexity of assignment of records	Zero $\rightarrow O(1)$	$2(n-1) \rightarrow O(n)$

6) Is Exchange Sort (Algorithm 1.3) or Insertion Sort (Algorithm 7.1) more appropriate when we need to find in nonincreasing order the k largest (or in nondecreasing order the k smallest) keys in a list of n keys? Justify your answer.

Solution: Exchange Sort is more appropriate in this situation as after k iterations of Exchange Sort, the array will contain the k smallest (largest) elements in the first k indices of the array, since each iteration of the algorithm inserts the next smallest (largest) element into the appropriate spot in the array. This is not the case for Insertion Sort, which could take n iterations to find the smallest (largest) k elements.

7) Rewrite the Insertion Sort algorithm (Algorithm 7.1) as follows. Include an extra array slot S[0] that has a value smaller than any key. This eliminates the need to compare j with 0 at the top of the while loop. Determine the exact time complexity of this version of the algorithm. Is it better or worse than the time complexity of Algorithm 7.1? Which version should be more efficient? Justify your answer.

Solution:

```
void insertion sort (int n, key type S [ ]) {
        index i, j;
        key type x;
        S [0] = MINUS_INFINITY;     //sufficiently small value
        for (i = 2; i < n; i++)
                x = S [i];
                j = i – 1;
                while [S [j] > x] {
                    S [j + 1] = S [ j ];
                    j – – ;
                }
                S [j + 1] = x;
        }
}
```

Worst-case complexity analysis:

- For comparisons: This algorithm performs in each iteration of the for loop an extra array comparison S[0] > x, which is always false. We add (n-1) to the W(n) for insertion sort →(n-1)(n+2)/2.
- The number of assignments is the same.

Hence Algorithm 7.1 is better.

8) An algorithm called Shell Sort is inspired by Insertion Sort's ability to take advantage of the order of the elements in the list. In Shell Sort, the entire list is divided into noncontiguous sublists whose elements are a distance h apart for some number h. Each sublist is then sorted using Insertion Sort. During the next pass, the value of h is reduced, increasing the size of each sublist. Usually the value of each h is chosen to be relatively prime to its previous value. The final pass uses the value 1 forh to sort the list. Write an algorithm for Shell Sort, study its performance, and compare the result with the performance of Insertion Sort.

Solution: Shells's original publication (Shell, D.L., "A High-Speed Sorting Procedure", Communications of the ACM 2 (7): 30–32, 1959) used the sequence of h-values (a.k.a. *gaps*) equal to N/2, N/4, … , 1 (taking floors if necessary). With these gaps, the algorithm is

```
int i, j, h, temp;


for (h = N/2; h > 0; h = h/2)
   for (i = h; i < N ; i++){
          temp = a[i];
          for (j = i; j >= h && temp < a[j-h]; j = j-h)
                 a[j] = a[j-h];
          a[j] = temp;
   }
}
```

Many better sequences have been found, e.g.

- 1, 3, 5, … 2^k-1, …, proven to be $\Omega(n^{3/2})$, and
- 1, 2, 3, 4, 6, …, 2^a3^b, …, proven to be $\Theta(n \cdot \lg^2 n)$.

Because of this, Shell Sort is theoretically important: by choosing different sequences, we can "bridge" the gap between $\Theta(n^2)$ and $\Theta(n \cdot \lg n)$.

Practically, it has the advantage of having a very simple code implementation (inherited from Insertion Sort), and it was found to outperform the $\Theta(n \cdot \lg n)$ algorithms on array sizes up to a few tens of thousands of elements.

Section 7.3

9) Show that the permutation [n,n−1, . . . , 2, 1] has n(n−1)/2 inversions.

Solution: Since, in the permutation [n, n-1, n-2, …, 2, 1], any element at index i is greater than the elements at exactly all indices j>i, each element at position i has n-i inversions. Thus, the total number of inversions is given by the sum of this difference for i=1 to n, which is the sum of the integers from n-1 down to 1, which is equal to n(n-1)/2.

10) Give the transpose of the permutation [2, 5, 1, 6, 3, 4], and find the number of inversions in both permutations. What is the total number of inversions?

Solution: The permutation [2,5,1,6,3,4] has 6 inversions. Its transpose [4,3,6,1,5,2] has 9 inversions. The total number of inversions is thus 15.

11) Show that there are n(n−1)/2 inversions in a permutation of n distinct ordered elements with respect to its transpose.

Solution: A permutation (k1,k2, … , kn) has an inversion (i, j) with respect to its transpose if the order between ki and kj in the permutation is the opposite of the order of the same elements in the transpose.
It is easy to see that all elements change the order in the transpose, so the number of inversions is the number of distinct pairs (i, j). As shown in Section A.7, the number of combinations of n objects taken k at a time is $\frac{n!}{k!(n-k)!}$. Making k = 2 and simplifying, we obtain n(n-1)/2.

12) Show that the total number of inversions in a permutation and its transpose is n(n−1)/2. Use this to find the total number of inversions in the permutation in Exercise 10 and its transpose.

Solution: The proof of Theorem 7.1 states:
--Let r and s be integers between 1 and n such that s >r. Showing that there are n(n−1)/2 such pairs of integers between 1 and n is left as an exercise.--
As shown in Section A.7, the number of combinations of n objects taken k at a time is $\frac{n!}{k!(n-k)!}$. Making k = 2 and simplifying, we obtain n(n-1)/2.

In exercise 10, n is 6, so our formula yields:

$$\frac{n(n-1)}{2} = \frac{6(6-1)}{2}$$

$$= \frac{6\times 5}{2} = 15 \text{ inversions.}$$

Section 7.4

13) Implement the different Mergesort algorithms discussed in Section 2.2 and Section7.4, run them on your system, and study their best-case, average-case, and worst-case performances using several problem instances.

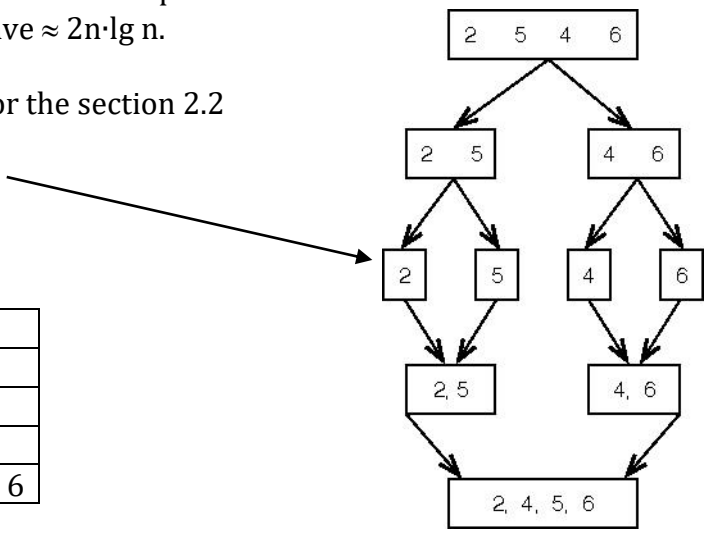
Solution: The implementations and performances will vary.

14) Show that the time complexity for the number of assignments of records for the Mergesort algorithm (Algorithms 2.2 and 2.4) is approximated by T (n) =2n·lg n.

Solution: In Section 2.2, it was shown that the worst-case comparison complexity of Mergesort is ≈ n·lg n, and from Algorithm 2.3 we see that every comparison in Merge is followed by a record assignment, so we have ≈ n·lg n assignments in the Merge part of the algorithm. To this, we add the copy operations in Algorithm 2.2, which are also ≈ n·lg n, because the division tree is lg n levels deep.
Adding the two previous numbers, we have ≈ 2n·lg n.

Example: Consider the following input for the section 2.2 Algorithm: 2, 5, 4, 6, where n = 4.
To get to the singleton list in the middle, we did 8 assignments.
Then we performed this Merge:

U	V	S
2, 5	4, 6	2
2, 5	4, 6	2, 4
2, 5	4, 6	2, 4, 5
2, 5	4, 6	2, 4, 5, 6



The Merge required another 8 assignments, for a total of 16.

The formula 2n·lgn yields $2x4x\log_2 4 = 2x4x2 = 16$, the same number!

15) Write an in-place, linear-time algorithm that takes as input the linked list constructed by the Mergesort 4 algorithm (Algorithm 7.4) and stores the records in the contiguous array slots in nondecreasing order according to the values of their keys.

Solution: With the notation in Algorithm 7.4, the merged list is identified by the index of its head, called *mergedlist*. The whole list is stored in the array S, whose elements have type *struct node*, i.e. they are pairs (key, link). We need the local variable *next_index*. The pseudocode is:

```
void contiguous(S, n, mergedlist) {
     next_index = mergedlist;
     for (int i =1; i<=n; i++) {
          swap S[i] with S[next_index];
          next_index = S[i].link;
     }
}
```

Notes:

- The *link* members are not updated, since they are redundant once the array is physically sorted.
- Swapping requires a temporary variable of type *struct node*.
- The every-time complexity is $\Theta(n)$, due to the existence of only one for loop.

16) Use the divide-and-conquer approach to write a nonrecursive Mergesort algorithm. Analyze your algorithm, and show the results using order notation. Note that it will be necessary to explicitly maintain a stack in your algorithm.

Solution: The non-Recursive Mergesort Algorithm is modeled after Algorithm 2.1, but instead of using recursive calls, the stack details are manipulated explicitely. The array S is global.

```
struct Stack {
      int lo, hi;
      bool new;
};

void non_rec_Mergesort (int n) {    //n array elements
   struct Stack S[100];
   int top, low, hig, mid;
   top = 0;

   S[top].lo = 1;              //first array index and
   S[top].hi = n;              //last array index are being
   S[top].new = true;
```

```
    top++;                          //push on the stack

    while (top > 0) {                   //stack is not empty
       low = s[top].lo;
       hig = s[top].hig;
       mid = floor((lo + hi)/2);
       if (S[top].new) {                //first time processing subarray
         if (low >= hig) {              //boundary case: 0 or 1 elements
              top--;                    //pull off stack
              continue;                 //go to next top of stack
         }
         else {                         //non-trivial subarray
              S[top].new = false;
              top++;                    //push left subarray
              S[top].lo = low;
              S[top].hi = mid;
              S[top].new = true;
              top++;                    //push right subarray
              S[top].lo = mid+1;
              S[top].hi = hig;
              S[top].new = true;
         }
       }
       else {                           //not new subarray, must merge
         merge2(low, mid, hig);         //See Algorithm 2.3
         top--;                         //pull off stack once merged
       }
    }// end of while
}
```

The operation is identical to that of the recursive Mergesort from Algorithm 2.2, so it has the same complexity: $\Theta(n \cdot \lg n)$ in the worst and average cases.

17) Implement the nonrecursive Mergesort algorithm of Exercise 16, run it on your system using the problem instances of Exercise 13, and compare the results to the results of the recursive versions of Mergesort in Exercise 13.

Solution: Implementations and comparisons will vary.

18) Write a version of mergesort3 (Algorithm 7.3), and a corresponding version of merge3, that reverses the roles of two arrays S and U in each pass through the repeat loop.

Solution:

```
void mergesort3_reverse_SU (int n, keytype S[]) {
        int m;
        index low , mid, high , size ;
        keytype U[n];                   //U is now local to mergesort3
        bool S_to_U = true;             //keeps track of roles

        m = pow(2, ceil(log(n));
        size = 1;                       //start with singletons
        repeat (lg m times) {
                for (low = 1; low <= m− 2∗size + 1; low = low + 2∗size) {
                        mid = low + size − 1;
                        high = minimum(low + 2∗size − 1, n);
                        merge3(low , mid, high , S_to_U);
                        S_to_U = !S_to_U;       //reverse roles of S and U
                }
                size = 2∗ size ;
}
```

The function merge3 is similar to merge2 from Algorithm 2.5, but it makes a decision based on S_to_U: if true, it works exactly as merge2, if not, the roles of S and U in the code (comparisons, assignments) are reversed.

Section 7.5

19) Implement the Quicksort algorithm (Algorithm 2.6) discussed in Section 2.4, run it on your system, and study its best-case, average-case, and worst-case performances using several problem instances.

Solution:

```
#include<stdio.h>
      void quicksort(int [10], int, int);
      int main(){
       int x[20], size, i;
       printf (“Enter size of the an(array: ”);
       scanf(“%d”,&size);
       printf(“Enter %d elements: “,size);
       for(i=0;i<size; i++)
        scanf(“%d”, &x[i]);
```

```
 quicksort(x,0,size-1);
 printf("Sorted elements: ");
 for(i=0;i<size; i++)
  printf("%d", x[i]);
 return 0;
}
void quicksort(int x[10], int first, int last) {
   int pivot, j, temp, i;
   if(first<last){
    pivot=first;
    i=first;
    j=last;

    while(i<j){
     while(x[i]<=x[pivot]&&i<last)
      i++;
     while(x[j]>x[pivot])
      j--;
     if(i<j){
      temp=x[i];
      x[i]=x[j];
      x[j]=temp;
     }
   }
   temp=x [pivot];
   x[pivot]=x[j];
   x[j]=temp;
   quicksort(x, first, j -1);
   quicksort(x, j+l, last); }
}
```

Worst Case Analysis:

$T(n) = n + T(n-l)$

$T(n) = \Theta(n2)$

Best Case and Average Case:

$T(n) = n+2T(n/2)$

$T(n) = \Theta (n \log n)$

Analysis

n	**n^2**	**n log n**
10	100	34
50	2500	288
100	10000	670
200	22500	1095

20) Show that the time complexity for the average number of exchanges performed by the Quicksort algorithm is approximated by 0.69(n+1)lg n.

Solution: At the end of Section 2.4, the average number of comparisons in Quicksort is proven to be A(n) ≈ 1.38(n+1)lg n. Since the exchanges are performed only in the *partition* function (Algorithm 2.7), we reproduce it here:

```
void partition (index low, index high, index& pivotpoint) {
     index i , j ;
     keytype pivotitem ;
     pivotitem = S[low ];
     j = low;
     for (i = low + 1; i<= high; i++)
          if (S[ i ] < pivotitem) {
               j++;
               exchange S[ i ] and S[ j ];
          }
     pivotpoint = j ;
     exchange S[low] and S[pivotpoint];
}
```

At each comparison (if), in the average case there are equal probabilities for the result to be true or false, so on average there are half as many exchanges: 1.386(n+1)lg n / 2 = 0.693(n+1)lg n.
Note that the bulk of the exchanges are performed inside the for loop. The exchange on the last line is only performed once per *partition* call, and on average there are ≈(n-1) such calls, so for asymptotic analysis we can neglect the linear term in comparison to n lg n.

21) Write a non-recursive Quicksort algorithm. Analyze your algorithm, and show the results using order notation. Note that it will be necessary to explicitly maintain a stack in your algorithm.

Solution:

```
void quicksort (item * a, int start, int stop) {
   int i, s = 0, stack [64];
   stack [s++] = start; stack [s++] = stop;
   while (s > 0) {
          stop = stack [--s]; start = stack [--s];
          if (start >= stop) continue;
          i = partition (a, start, stop);
          if (i-start > stop-i) {
                stack [s++] = start; stack [s++] = i - 1;
                stack [s++] = i + 1; stack [s++] = stop;
```

```
            }
            else {
                    stack [s++] = i + 1; stack [s++] = stop;
                    stack [s++] = start; stack [s++] = i -1;
            }
    }
}
```

Complexity analysis: All the comparisons and record assignments are done in *partition*, which has not changed, so these complexities are the same as in the recursive algorithm: worst-case: $\Theta(n^2)$, best-case: Θ (n log n)

22) Implement the non-recursive Quicksort algorithm of Exercise21, run it on your system using the same problem instances you used in Exercise19, and compare the results to the results of the recursive version of Quicksort in Exercise19.

Solution: Implementation and performance will vary, but, all other things being equal, the non-recursive algorithm performs a little faster than the recursive one because it lacks the overhead incurred by the recursive function calls.

23) Show that with the new partition procedure, the time complexity for the number of assignments of records performed by Quicksort is given by $A(n) \approx 0.69(n+1)\lg n$. Show further that the average-case time complexity for the number of comparisons of keys is about the same as before.

Solution:
For the average–case analysis, we assume that all possible orderings are equal likely. For simplicity, we take the keys to be just the integers from 1 to n.

Average nr. of assignments of records
When we select the pivot in the function *partition*, it is equally likely to be any one of the n keys. Denote by p whatever key is selected as pivot. After the partition, key p is guaranteed to be in position p, since the keys 1, …p-1 are all to its left (and p+1 , … , n are all to its right). The number of exchanges inside the loop is therefore equal to the number N(p) of keys larger than p that were initially positioned to the left of index p. (Compare this with the initial *partition* in Algorithm 2.7, where all p-2 numbers between *low* and *p* need to be exchanged in!) What is the average E[N(p)] of N(p)? Since the permutation is random, we can assume that the (n-p) numbers larger than p are uniformly distributed, so the number of them in the interval [2 … p] is proportional to the length of this interval: E[N(p)] =(p-1)(n-p)/(n-1). The "minus ones" are there because we already know that p is in position 1, so we don't count this position.

Using the 3-assignment exchange technique, this means 3E[N(p)] assignments.

Outside the loop, we have one assignment for **`pivotitem = S[low]`**, and three for the exchange on the last line, for a total of 4.

Summing up, there are 3E[N(p)] + 4 assignments in the first call to *partition* if the pivot (first element) is p.

The solution continues similarly to the average-case analysis at the end of Section 2.4:

Denote by A(n) the average number of assignments done by quicksort on a list of length n. Since each p is equally likely, we have

$$\text{A(n)} = \sum_{p=1}^{n} \frac{1}{n}\left[A(p-1) + E[N(p)] + 4 + A(n-p)\right] =$$

$$\frac{1}{n}\left[3\sum_{p=1}^{n} E[N(p)] \quad + 4n + \quad \sum_{p=1}^{n}[A(p-1) + A(n-p)]\right] =$$

$$4 + \frac{3}{n}\sum_{p=1}^{n}\frac{(p-1)(n-p)}{n-1} + \frac{2}{n}\sum_{p=1}^{n} A(p-1)$$

With the tools in Section A.3, the first sum can be shown to be $\frac{n(n-2)}{6}$, so we have:

$$\text{A(n)} = 4 + \frac{n-2}{2} + \frac{2}{n}\sum_{p=1}^{n} A(p-1)$$

$$\text{A(n)} = \frac{n+6}{2} + \frac{2}{n}\sum_{p=1}^{n} A(p-1) \qquad (*)$$

Multiply (*) by n, then write a second relation where n is substituted by (n-1), and then subtract the latter from the former to obtain

$$\frac{A(n)}{n+1} = \frac{A(n-1)}{n} + \frac{n+2.5}{n(n+1)}$$

Denote A(n)/(n+1) by x_n and solve the recursion for x_n by direct substitution, as in Example B.22, to obtain $x_n \approx \ln n$, and, accordingly, $A(n) \approx (n+1)\ln n$.
Changing the base of the logarithm to binary, we use $\ln n = \ln 2 \cdot \lg n \approx 0.69 \lg n$, so

$$A(n) \approx 0.69(n+1)\lg n$$

Average nr. of comparison of keys
i and j start, respectively, from 1 and n, and take steps moving towards each other until they meet (or overlap by 1 position). Each move, either i++ or j--, is always done as a result of a key comparison, so the number of key comparisons is <u>every time</u> about n. Therefore the recursion at the end of Section 2.4 is essentially the same, and has the same approximate complexity: 1.38(n+1)lg n.

24) Give two instances for which Quicksort algorithm is the most appropriate choice.

Solution: The quadratic worst-case complexity of Quicksort (obtained in Section 2.4) is not often a practical issue, because, even in applications where the array is already sorted (or almost sorted) in non-decreasing order, randomized Quicksort can be applied.

The only competition from $\Theta(n^2)$ algorithms could come in applications where the extra memory space is an issue, since those algorithms (e.g. insertion sort) are able to sort in place, whereas the best Quicksort still needs lg n extra locations on the stack. However, in order for lg n to be significant, n itself must be large, so $\Theta(n^2)$ will make the running time prohibitive. Shellsort with a gap sequence that yields $\Theta(n^{1.5})$ is the only practical competition, for n up to $\approx$ 10,000 records.

When comparing Quicksort with Mergesort, one significant difference in performance comes from the extra memory space:

- The best Quicksort needs only lg n extra records, whereas the best Mergesort needs n.
- True, if the records are large, Mergesort can be implemented with a linked list version, which is in-place, but it is more complex to code and still has overhead for the links. However, if the data to be sorted is already stored as a linked-list, Mergesort becomes more promising.
- Mergesort is better when access to the individual records is slow, which happens when the array is too large for RAM, and therefore stored in external memory (HDD, tape); for this case, so-called "external sorting" algorithms exist base on Mergesort. Quicksort is not as easily applicable to external sorting, because the final position of the pivot is not easy to determine before partitioning. There seems to be a loose consensus that Quicksort is better for RAM-based sorting, whereas Mergesort is better for external sorting.

Real-life examples of Quicksort being the "best" implementation:

- The *qsort* library function for the C/C++ language was initially based on Quicksort (the Berkeley 1983 implementation), although the standards do not mandate it. Most of today's implementations still use it, with various improvements. One notable exception is the GNU library qsort, which is based on Mergesort instead.
- Parallel version of Quicksort on a cache-coherent shared address space multiprocessor outperforms best previous algorithm: P. Tsigas, Y. Zhang, "A Simple, Fast Parallel Implementation of Quicksort and its Performance Evaluation on SUN Enterprise 10000", Parallel, Distributed and Network-Based Processing, 2003. Proceedings of the Eleventh Euromicro Conference on, 2003, pp. 372 –381.
- Parallel multi-core graphics processors (GPU): D. Cederman, P. Tsigasy, "A Practical Quicksort Algorithm", Technical Report no. 2008-01, Goteborg University.

25) Another way to sort a list by exchanging out-of-order keys is called Bubble Sort. Bubble Sort scans adjacent pairs of records and exchanges those found to have out-of-order keys. After the first time through the list, the record with the largest key (or the smallest key) is moved to its proper position. This process is done repeatedly on the remaining, unsorted part of the list until the list is completely sorted. Write the Bubble Sort algorithm. Analyze your algorithm, and show the results using order notation. Compare the performance of the Bubble Sort algorithm to those of Insertion Sort, Exchange Sort, and Selection Sort.

Solution:

```
void bubbleSort(int n, keytype& S[]) {
index i,j;
for(i=n; i>0; i--) {
        for(j=1; j<=n; j++) {
                if(S[j] > S[j+1]) {
                        keytype temp = S[j];
                        S[j] = S[j+1];
                        S[j+1] = temp;
                }
        }
}
}
```

The first iteration performs n-1 comparisons, and each subsequent iteration performs one less comparison than the previous. The total number of comparisons is thus given by the sum of the integers from 1 to n-1, which is equal to n(n-1)/2. The complexity is thus $\Theta(n^2)$. The number of comparisons is always the same regardless of the input.

Section 7.6

26) Write an algorithm that checks if an essentially complete binary tree is a

heap. Analyze your algorithm and show the results using order notation.

Solution:

```
boolean isHeap(node n) {
```

```
        if(n.left==null && n.right==null) return true;
        if(n.right==null) {
                if(n.left > n) return false;
                return true;
        }
        if(n.left > n || n.right < n) return false;
        return isHeap(n.left) && isHeap(n.right);
    }
```

Initial call uses root of tree as input parameter. Assumes nodes have member variables left, right, and key, representing the left and right children of the node and its associated key, respectively. The left and right variables of a node may be null, indicating that the node has no such child. Note that a node in an essentially complete binary tree cannot have a null left child unless the right child is also null.

This algorithm performs a pre-order traversal of the tree, in which each node is visited exactly once. Thus, the time complexity is O(n), where n is the number of nodes in the tree.

27) Show that there are 2^j nodes with depth j for j < d in a heap having n (a power of 2) nodes. Here d is the depth of the heap.

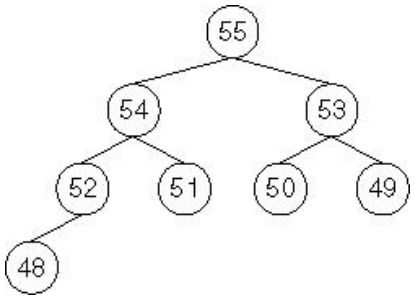
Solution: Induction on the depth d:

Base case: When d = 1, we only have j=0: $2^0 = 1$, and, indeed, there is only one node – the root – on level 0.

Induction step: Assuming the property holds for d, we prove it for d+1. For all j up to d-1, we have 2^j because of the assumption. For j = d, we use the fact that the heap is an essentially complete binary tree, meaning that the tree down to level d is complete; this means that each node on level d-1 has 2 children, so the number of those children is $2 \cdot 2^{d-1} = 2^d$. Q.e.d.

Example: In this heap we have:

$2^0 = 1$ node at depth 0 → 55
$2^1 = 2$ nodes at depth 1 → 54, 53
$2^2 = 4$ nodes at depth 2 → 52, 51, 50, 49



28) Show that a heap with n nodes has $\lceil n/2 \rceil$ leaves.

Solution: As in Exercise 27, let d be the depth of the heap. The number of nodes can be written as $n = 2^0 + 2^1 + 2^{d-1} + N(d)$, where N(d) is the number of nodes on the last level, d.

There are two cases:

- N(d) is odd →
- N(d) is even →

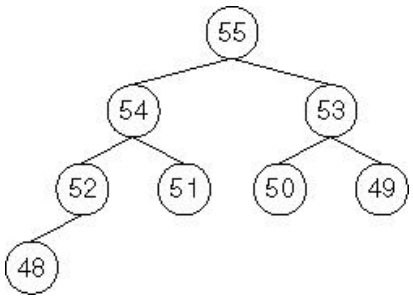
Base case: When n = 1, the only leaf is the root, and $\lceil n/2 \rceil = \lceil 1/2 \rceil = 1$.

Induction step: We assume the property to hold for n, and prove it for n+1. We distinguish between two cases:

- If $n = 2^k - 1$, we use the fact that a complete tree always has a number of nodes of the form $2^k - 1$. Because the heap is an essentially complete tree, then $\lceil n/2 \rceil = 2^{k-1}$. Because a complete tree always has a number of nodes of the form $2^k - 1$, we conclude that

Example: In this heap we have:

n = 8 nodes, and 4 leaves.
The number of leaves is equal to $\lceil n/2 \rceil = \lceil 8/2 \rceil = 4$

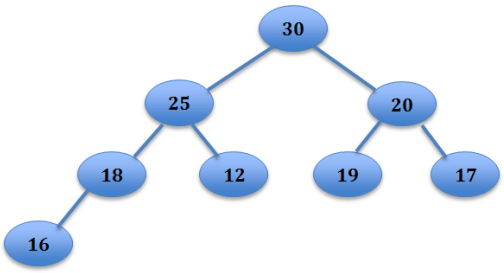


29) Implement the Heapsort algorithm (Algorithm 7.5), run it on your system, and study its best-case, average-case, and worst-case performances using several problem instances.

Solution: Implementation and performance measures will vary.

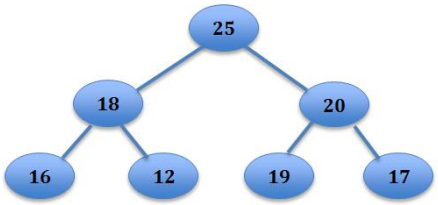
30) Show that there is a case for Heapsort in which we get the worst-case time complexity of $W(n) \approx 2n\lg n \in \Theta(n \lg n)$.

Solution: Consider this heap with n = 8:
$2n \lg n = 2 \times 8 \times 3 = 48$ comparisons.



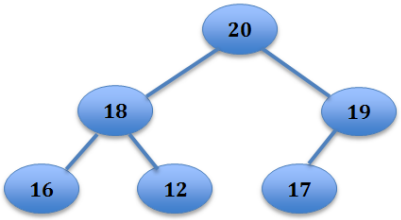
The first call to function *root* causes 30 to be extracted, and performs 2 swaps = 4 comparisons,

producing this heap:

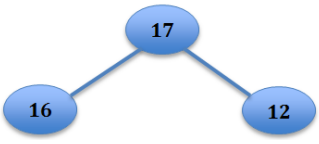


Extract 25 and perform 4 comparisons,

producing this heap:



Extract 20, 19 and 18, with 4 comparisons for each,



producing this heap:

Extracting 17 costs 2 comparisons, and then 16 and 12 need no comparisons.

Total: 22, which is quite different from 48!

However, asymptotic analysis only deals with the limit when $n \to \infty$. The point of the example is to show that, for general $n = 2^k$, we have the following tallies of comparisons:

- For about half the nodes 2^{k-1}, k swaps or 2k comparisons are performed $\rightarrow k2^k$.
- For half of the remaining half 2^{k-2}, k-1 swaps or 2(k-1) comparisons $\rightarrow (k-1)2^{k-1}$.
- … …

- Finally, when only 2 nodes are left, zero comparisons are needed.

Total: According to Example A.5, the sum is $(k-1)2^{k+1} \approx 2n\lg n$.

31) Show that the worst-case time complexity of the number of exchanges of records for Heapsort is approximated by $W(n) \approx n\lg n$.

Solution: From the *while* loop in the function *siftdown* (p.309), we see that one iteration of the loop contains two comparisons and one exchange. Since the worst case for comparisons is the same as the worst case for exchanges (i.e. after each extraction the root has to sift all the way down), the ratio comparisons/exchanges is 2/1. Using the result from the previous exercise, we have $2n\lg n/2 = n\lg n$.

32) Modify Heapsort so that it stops after it finds the k largest keys in non-increasing order. Analyze your algorithm, and show the results using order notation.

Solution:

```
void heapsort (int n, int k, heap & H) {
        remove keys (n, k, H, S);       //removes the largest k keys
}

void removekeys (int n, int k, heap & H, keytype s []) {
        index i;
        if (k > n) {
            cout << "Not enough elements!" <<endl;
            return;
        }
        for (i = n; i  >= n-k+1; i--)
            s[i] = root (H);
}
```

The other functions are identical.

Complexity analysis: The *for* loop in *removekeys* executes *k* times, and each time it calls *root*, which, as before, performs $\approx 2\lg n$ comparisons. The entire loop therefore has $\approx 2k\lg n$ comparisons, so the algorithm is in $\Theta(k\cdot\lg n)$.

Section 7.7

33) List all the advantages and disadvantages of all the sorting algorithms discussed in this chapter based on the comparisons of keys and the assignments of records.

Solution:

Among the quadratic algorithms (Table 7.1), Exchange Sort loses out (and is almost never used in practice), since all its performance measures are worse than those of the other two. Between Insertion and Selection Sort, Insertion is faster in terms of comparisons, but Selection is much faster (linear vs. quadratic) in terms of assignments.

Insertion sort can be made a lot more efficient (down to $n\lg^2 n$) in both comparisons and assignments by using diminishing gap sequences (Shellsort), as explained in Exercise 7.8, thereby becoming much faster than Selection in comparisons, and only slightly slower in assignments.

Among the $\Theta(n \lg n)$ algorithms (Table 7.2), the order for both assignments and comparisons is the same: Mergesort is best, followed by Quicksort (with Hoare's partition - Exercise 23), followed by Heapsort. However, the order in terms of extra space is exactly the opposite, which is why all three algorithms are used in practice.

34) Run the implementations of all the sorting algorithms discussed in this chapter on your system using several problem instances. Use the results, and the information provided in Exercise33, to give a detailed comparison of these sorting algorithms.

Solution: Implementations and results will vary.

35) Among Selection Sort, Insertion Sort, Mergesort, Quicksort, and Heapsort, which algorithm would you choose in each list-sorting situation below? Justify your answers.

(a) The list has several hundred records. The records are quite long, but the keys are very short.

Solution: For long records, the complexity of assignments is of primary concern, with Selection and Merge as prime candidates. Since n is small, the quadratic algorithm will win out because of its simplicity of implementation, since speed is not an issue.

(b) The list has about 45,000 records. It is necessary that the sort be completed reasonably quickly in all cases. There is barely enough memory to hold the 45,000 records.

Solution: For this n, quadratic algorithms are most likely too slow. The memory restriction recommends Heapsort, because it's in-place.

(c) The list has about 45,000 records, but it starts off only slightly out of order.

Solution: Again, we consider only $\Theta(n \lg n)$ algorithms for speed. With the list only slightly out of order, Quicksort will "go quadratic", so it's not recommended (unless we choose a randomized version). Since extra space is not an issue, Mergesort wins over Heapsort b/c it's twice as fast in comparisons and can be also made better in assignments by choosing the linked list variation.

(d) The list has about 25,000 records. It is desirable to complete the sort as quickly as possible on the average, but it is not critical that the sort be completed quickly in every single case.

Solution: With 25,000 records, Insert Sort (fastest quadratic) requires 156,250,000 comparisons, whereas Mergesort only requires 365,241, so Mergesort wins. The statement of this case seems to suggest that Quicksort is also acceptable, although its nr. of comparisons is slightly (1.38 times) larger.

36) Give at least two instances for each of the sorting algorithms (based on the comparisons of keys) discussed in this chapter for which the algorithm is the most appropriate choice.

Solution:

- Insertion Sort: Relatively small number ($n \approx 1{,}000$) of keys, when only average performance is important e.g. repeatedly sorting 10-minute interval weather data collected in one day from a number of weather stations, or sorting someone's tweets by length in real-time (it is also an "online" algorithm, i.e. it can sort partial data as it receives it).
- Selection sort is preferred to Insertion Sort in real-time applications where it is important for the run-time to be identical irrespective of the order of the elements, e.g. in an embedded system that needs to perform some critical task periodically, or when there is little extra memory space available.
- Mergesort is used when there are many ($n>10{,}000$) keys with long records, when the records are stored externally, e.g. for sorting databases located on tape or network storage (NFS, SAN), or when the records are stored in caches, b/c it accesses blocks of contiguous data (external sorting). Mergesort is now used by several scripting languages as internal sorting algorithm instead of Quicksort, e.g. Python uses Timsort, which is a combination of Mergesort and Insertion Sort.
- Quicksort is used for in-RAM sorting, when we need a good compromise between the complexity of comparisons and that of assignments; it is the basis of most qsort library implementations for C/C++, and of efficient GPU sorting algorithms.

- Heapsort is the slowest of the Θ(nlg n) algorithms, but it has the great merit of sorting in-place, so it's used in embedded systems when the number of keys is large (n>10,000), e.g. in image-processing. One great application is the priority queue (introduced for Huffman's algorithm in Ch.4), because it supports insertion and extraction efficiently (Θ(lg n)). Priority queues are used in their own right in many other algorithms, but also in operating systems (e.g. queue of ready processes), event-driven simulators (queue of events sorted by their time of occurrence), and Internet routers (queues of packets sorted by class of service).

Section 7.8

37) Write a linear-time sorting algorithm that sorts a permutation of integers 1 through n, inclusive. (Hint: Use an n-element array.)

Solution:

Inputs: positive integer n, array of integers S containing a permutation of the integers from 1 to n, inclusive.

Outputs: The array S sorted in nondecreasing order.

```
void  nSort(int n, int& S[]) {
    index i;
    int A[1…n];
    for(i=1; i<n; i++)
            A[S[i]] = S[i];
    S = A;
}
```

38) Does the linear-time performance of your algorithm in Exercise 37 violate the lower bound for sorting only by comparisons of keys? Justify your answer.

Solution: The lower bound for sorting based on comparison of keys is not violated by the algorithm of Exercise 37 because there are no comparisons performed by the algorithm. Strong assumptions are made about the input that allow for linear time complexity.

39) Prove the general case of Lemma 7.9 when the number of leaves m is not a power of 2.

Solution: Lemma: 7.9: For any 2 tree that has m leaves and whose EPL equals min EPL(m), the depth d is given by $d=\lceil \lg m \rceil$. The proof is modeled after the one for Lemma 7.9:

If m is not a power of 2, then, for some positive integers k and z, we have

$$m = 2^k + z, \quad \text{with } z < 2^k$$

Let d be the depth of a minimizing tree. By Lemma 7.8, there are $r= 2^d - m = 2^d - 2^k - z$ leaves on the level d – 1, and, since $r \geq 0$, we must have $d \geq k + 1$.

We show that assuming d > k+1 leads to a contradiction. If d > k+1, then $d \geq k+2$, and, by Lemma 7.8, the number of leaves on level d is

$$2m - 2^d = 2(2^k + z) - 2^d = 2^{k+1} + 2z - 2^d < 2^{k+1} + 2\cdot 2^k - 2^d = 2^{k+2} - 2^d \leq 2^{k+2} - 2^{k+2} = 0.$$

This means that the number of leaves on level d is negative – contradiction.

Therefore, d = k + 1.

On the other hand $=\lceil \lg m \rceil = \lceil \lg(2^k + z) \rceil = k + 1$, since $z < 2^k$.

We established that $d = k + 1 = \lceil \lg m \rceil$. Q.e.d.

Section 7.9

40) Implement the Radix Sort algorithm (Algorithm 7.6), run it on your system, and study its best-case, average-case, and worst-case performances using several problem instances.

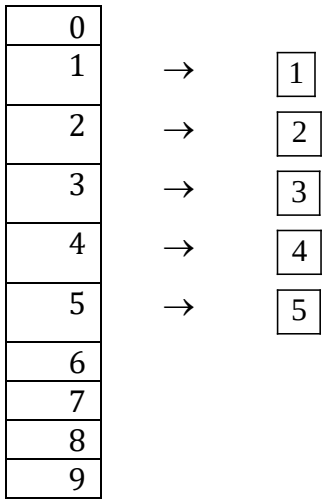
Solution: Implementations and performances will vary.

Best Case performance occurs when we have the smallest number of digits, i.e. the numbers do not have (big) gaps. Ideally, we could have no gaps, meaning consecutive numbers. Here is an example with the keys being one-digit numbers:

1	5	4	2	3

First pass

1
5
4
2
3



items are moved back to the master list:

1	2	3	4	5

Worst Case performance occurs when we have a mix of numbers, of which some are very small, e.g. one-digit, and a few (just one?) very large, e.g. 100 digits. The length of the *for* loop in the function *radixsort*, *numdigits* will have to be 100 to accommodate the few large numbers, but most of the time only the zeroes in the small numbers will be distributed.

For the average case, the concrete class of applications should be considered. If we have a given finite precision (e.g. integers on 32 bits), and the numbers are uniformly distributed, simply use the library random number generator to generate lists for sorting. If, on the other hand, the number of digits is variable within large bounds (e.g. strings, or "infinite-precision" integers), then the meaning of "average" has to be clarified first for the given application.

41) Show that when all the keys are distinct the best-case time complexity of Radix Sort (Algorithm 7.6) is in Θ(nlgn).

Solution: When all keys are distinct, the best case occurs when the keys consist of the first n integers (i.e., 1 through n), since in this case n is the smallest possible maximum value of the set of keys. Since the complexity of Radix Sort is Θ (n×numdigits), and the number of digits, it takes to represent the number n is $\lceil \log_B n \rceil$, the best-case complexity in this case is thus $\Theta(n\log_B n)$. The base does not matter in order notation, since $\log_B n = (\log_2 n)/ (\log_2 B)$, and $\log_2 B$ is just a constant factor, therefore the best-case complexity is also in Θ(nlg n).

42) In the process of rebuilding the master list, the Radix Sort algorithm (Algorithm 7.6) wastes a lot of time examining empty sublists when the number of piles (radix) is large. Is it possible to check only the sublists that are not empty?

Solution: We see in the function *coalesce* that the sublists are implemented as an array of lists named *list*:

```
for (j = 0; j <= 9; j++)
    link the nodes in list[ j ] to the end of masterlist ;
```

With this data structure, it's impossible to keep track of the empty sublists.

The problem can be fixed at the price of using a more complicated data structure, e.g. adding to the array element *list* a boolean field that keep track of whether the list stored at that position is empty or not. The for loop in *coalesce* will now read:

```
for (j = 0; j <= 9; j++)
    if (list[j].notEmpty)
      link the nodes in list[ j ].actualList to the end of masterlist ;
```

The *notEmpty* field has to be always initialized with *false*, and it is turned to *true* in the function *distribute*:

```
while (p != NULL) {
        j = value of ith digit (from the right) in p–> key;
        link p to the end of list[ j ].actualList;
        list[j].notEmpty = true;
        p = p–> link ;
}
```

Additional Exercises

43) Write an algorithm that sorts a list of n elements in nonincreasing order by finding the largest and smallest elements and exchanges those elements with the elements in the first and last positions. Then the size of the list is reduced by 2, excluding the two elements that are already in the proper positions, and the process is repeated on the remaining part of the list until the entire list is sorted. Analyze your algorithm and show the results using order notation.

Solution: Finding the minimum and maximum elements of an unsorted array of n elements can be done in the worst case in 2(n-1) comparisons, by initializing a pair of variables *max* and *min* with a[1] and then comparing them in a loop with all the remaining n-1 elements. The recursion is T(n) = 2(n-1) + T(n-2), easily solve by repeated substitution.

For even n, we have T(n) = 2(n-1) + 2(n-3) + … + 2(3) + T(2), and T(2) =1, because sorting two elements only requires one comparison. Using the summation techniques from Appendix A.3, we obtain $T(n) \approx (3/2)n^2$.

For odd n, we obtain similarly $T(n) \approx (3/2)n^2$, so for all n we have $T(n) \approx (3/2)n^2$), which is in $\Theta(n^2)$.

The algorithm sorts in place, needing only 2 extra variables = $\Theta(1)$ extra space.

44) Implement the Quicksort algorithm using different strategies for choosing a pivot item, run it on your system, and study its best-case, average-case, and worst-case performances for different strategies using several problem instances.

Solution: Different ways to select the pivot:

- First element (Program was written in Exercise 19): We have the worst case when the array is already sorted (either ascending or descending) → $\Theta(n^2)$. We say that the algorithm "goes quadratic".
- Last element: We also have the worst case above.
- Median-of-three elements: Calculate the "middle value", a.k.a. the median of the first, middle and last element of the partition. Use median as the pivot. This counters both the sorted-ascending and sorted-descending quadratic cases, but there are still rare cases of "natural" data on which Quicksort goes quadratic.
- Random element: Randomly pick an element as a pivot. Although the worst-case is still $\Theta(n^2)$ as above, it is very unlikely to obtain it!
- If we always picked a pivot value that exactly cuts the array in half, a.k.a. the array median, then, half of the values in the array are larger and half are smaller, making Quicksort optimal. Unfortunately, the only sure way to find the true median is to examine the entire array. There are tradeoff methods that estimate the median that are less expensive.

Ultimately, there is no foolproof way to avoid Quicksort going quadratic. In practice, we can simply detect the quadratic condition and switch to a different algorithm if this happens, as done in Introsort. According to Wikipedia, "[Introsort] begins with quicksort and switches to heapsort when the recursion depth exceeds a level based on (the logarithm of) the number of elements being sorted. This combines the good parts of both algorithms, with practical performance comparable to quicksort on typical data sets and worst-case O(nlog n) runtime due to the heapsort. "

45) Study the idea of designing a sorting algorithm based on a ternary heap. A ternary heap is like an ordinary heap except that each internal node has three children.

Solution: The number of levels d in a ternary heap is $\Theta(\log_3 n) = \Theta(\lg n)$, although the multiplicative constant is smaller: $d \approx (\lg n)/(\lg 3) \approx 0.63 \lg n$, so, for the same number of elements, a ternary heap is only about 63% as high as its binary counterpart.

Insertions in a ternary heap are done exactly as in the binary heap, they still require only one comparison per exchange (between child and parent), and so they are $O(\log_3 n)$. This still leaves the number of comparisons in *makeheap* linear, O(n), although the multiplicative constant is this time larger: 3 instead of 2.

Removing the root key is almost the same, except that determining the largest child for *siftdown* requires two comparisons instead of one, so for every exchange we need 3 comparisons instead of 2. We can obtain an upper bound similar to the one on p.314: $3n\log_3 n$, which is also an upper bound for the entire sorting algorithm.

Based on these upper bounds, ternary heaps should be asymptotically $(2n\log_2 n - 3n\log_3 n)/3n\log_3 n \approx 0.0566$, or 5.66% faster than binary heaps. Practical experiments with long arrays (n = 100,000 – 1,000,000) indicate an even larger speedup, in the range of 8-10%.

46) Suppose we are to find the k smallest elements in a list of n elements, and we are not interested in their relative order. Can a linear-time algorithm be found when k is a constant? Justify your answer.

Solution: This problem can be solved by running Exchange Sort or Selection Sort for k iterations. After k iterations, the list is guaranteed to contain the k smallest elements in the first k indices of the list. Each iteration takes O(n) time, and so the whole process takes O(kn) time, which is linear in n when k is a constant.

47) Suppose we have a very large list stored in external memory that needs to be sorted. Assuming that this list is too large for internal memory, what major factor(s) should be considered in designing an external sorting algorithm?

Solution: The CPU can access in-RAM data very fast (e.g. tens of nanoseconds), but transferring data between RAM and the external memory (disk, tape) is about a million

times slower (e.g. tens of milliseconds or worse). For this reason, we should minimize the number of accesses (reads and writes) to the external memory. The general plan is to:

- Load from the external memory chunks of data that fit in the RAM.
- Sort each chunk in the RAM with an "internal" sorting algorithm that has "locality of reference", herein abbreviated LoL. An algorithm having LoL uses only data items in the local chunk.
    - Mergesort has very good LoL, since it always uses only a well-defined half of the array. It is the main algorithm used in external sorting.
    - Heapsort, in contrast, has poor LoL, because the sifting requires exchanges between parent-child pairs that can be stored in vastly different positions in the physical array.
    - Quicksort also has good LoL: the function *partition* (algorithm 2.7) compares elements in linear, orderly fashion, by moving through the array, so, as long as the indices i and j are staying reasonably close to each other, they will remain in the same chunk.
- Merge smaller chunks together in RAM (2, 3, or k at a time) and move the merged parts back to external memory. Here Quicksort has the upper hand: unlike Mergesort, merging the chunks is trivial!

48) Classify the sorting algorithms discussed in this chapter based on the ideas behind the algorithms. For example, Heapsort and Selection Sort find the largest (or smallest) key and exchange it with the last (or first) element according to the desired order.

Solution:

- Insertion Sort finds for each new element the final place among the already sorted ones, and inserts it there, after shifting other elements to make room for it.
- Quicksort is more subtle, in that it does not require the other elements to be sorted, but still manages to find the final position of each new element (the pivot). The other important idea is divide-and-conquer.
- Mergesort also uses divide-and-conquer, but without actually finding the final place of any element until the final merge. Unlike Quicksort, this relaxation "buys" the ability to always split the array into deterministic halves, which is important in parallel environments and for external sorting.

49) A stable sorting algorithm is one that preserves the original order of equal keys. Which of the sorting algorithms discussed in this chapter are stable? Which are unstable? Justify your answer.

Solution:

Stable:

- Insertion Sort: In algorithm 7.1, if 42A comes before 42B in the original array, then 42A will be inserted first, and, when the turn comes for 42B, only elements strictly larger than 42 are shifted down, so 42B will end up under (after) 42A.

Unstable:

- Heapsort: Example: Consider the initial array [42A, 42B, 42C], in which the keys used for sorting are all 42, and the labels A, B, C will help us to distinguish them. The resulting heap has 42A in the root, with 42B as left child and 42C as right child, so they will be extracted in the reverse order [42A] -> [42C, 42A] -> [42B, 42C, 42A], so the initial order was not preserved.
- Selection Sort: Example: Consider the initial array [42A, 42B, 1]; after the first iteration of the outer loop, the order is [1, 42B, 42A], therefore the relative order of 42A and 42B has changed.
- Mergesort: The pseudocode from Algorithms 2.3 and 2.5 actually inverts the positions of identical elements.
- Quicksort: Consider the initial array [10, 42A, 42B, 1]; during the first partition (we use Hoare's partition from Exercise 23), 42A swaps with 1, so, after swapping the pivot 10 with 1, the order is [1, 10, 42B, 42A], therefore the relative order of 42A and 42B has changed.

50) Which of the sorting algorithms identified as unstable in Exercise 49 can easily be changed to stable sorting algorithms?

Solution:

- Mergesort: In the pseudocode from Algorithms 2.3 and 2.5, we replace one < with <= thus:

```
while (i<= h && j<= m) {
        if (U[ i ] <= V[ j ]) {          //this is the only change!
                S[k] = U[ i ];
                i++;
        }
        else {
                S[k] = V[ j ];
                j++;
        }
        k++;
}
```

- Quicksort can be made stable if we insert the elements during the partitioning at the right end of the sublists, rather than swap them (the swap can change the relative order, as shown in the solution of Exercise 49). Insertions in an array are, however, too expensive, and they will ruin the (nlg n) complexity; the only way to preserve the low complexity is to have the data in a linked list. It can also be made "almost stable" if we are willing to spend more time choosing a better pivot.