Section 2.1

1) Use Binary Search, Recursive (Algorithm 2.1) to search for the integer 120 in the following list (array) of integers. Show the actions step by step.

12 34 37 45 57 82 99 120 134

Solution:

1. Compute the middle index from initial low (1) and high (9) indices: **mid** = 5
2. Check key at position **mid**: 120 > **S**[**mid**] = 57
3. Compute new middle index from new low (**mid**=5) and previous high (9) indices: **mid** = 7
4. Check key at position **mid**: 120 > **S**[**mid**] = 99
5. Compute new middle index from new low (**mid**=5) and previous high (9) indices: **mid** = 8
6. Check key at position **mid**: 120 == S[mid]
7. Key found, return index **mid**=8

2) Suppose that, even unrealistically, we are to search a list of 700 million items using Binary Search, Recursive (Algorithm 2.1). What is the maximum number of comparisons that this algorithm must perform before finding a given item or concluding that it is not in the list?

Solution:

ceil($\log_2$700,000,000) = 30

3) Let us assume that we always perform a successful search. That is, in Algorithm 2.1 the item x can always be found in the list S. Improve Algorithm 2.1 by removing all unnecessary operations.

Solution:

```
index location (index low, index high){
        index mid;
        mid = (low + high)/2;
        if (x = S[mid])
            return mid
        else if (x < s[mid])
            return location (low, mid-1);
        else
        return location (mid+1, high);
    }
```

where

Prob: Determine whether *x* is in the Sorted array '*s*' of size '*n*'

Input: Positive integer *n*, sorted array of key '*s*', indexed from 1 to *n* a key *x*.

Output: Location, the location of *x* is *s*

4) Show that the worst-case time complexity for Binary Search (Algorithm 2.1) is given by

$$W(n) = \lfloor \lg n \rfloor + 1$$

when n is not restricted to being a power of 2.

Solution: As hinted, we first show that the recurrence equation for W (n) is given by

$W(n) = 1 + W\left(\left\lfloor \frac{n}{2} \right\rfloor\right)$ for n > 1, and W (1) = 1. To do this, we consider even and odd values of n separately:

case 1: n is even → n = 2m → the array is split into two equal subarrays of size m = n/2 = $\left\lfloor \frac{n}{2} \right\rfloor$ each, and mid is (low + high)/2, which is the right-most element of the left array. To decide which way to continue the search, we need one comparison, and in the worst case we go right, so we have $\left\lfloor \frac{n}{2} \right\rfloor$ elements for the next level of the recursion, for a total of $\left\lfloor \frac{n}{2} \right\rfloor + 1$.

case 2: n is odd → n = 2m + 1 → mid is (low + high)/2, which is the central element of the array. To decide which way to continue the search, we need one comparison, and either way we have $\left\lfloor \frac{n}{2} \right\rfloor$ for the next level of the recursion, for a total of $\left\lfloor \frac{n}{2} \right\rfloor + 1$.

If n = 1, we only have the comparison (x == S(mid)), so W(1) = 1.

We have proved that W (n) = 1 + W $\left(\left\lfloor \frac{n}{2} \right\rfloor \right)$ for n > 1, and W (1) = 1.

Now we use induction to solve the recurrence equation:

Base case: n = 1 → lg n = 0 → W(n) = 0 + 1 = 1 correct.

Induction step: We assume W (k) = $\lfloor \lg k \rfloor + 1$ for all k ≤ n, and prove it for n+1.

According to the result proven above, W(n+1) = 1 + W $\left(\left\lfloor \frac{n+1}{2} \right\rfloor \right)$. Is $\left\lfloor \frac{n+1}{2} \right\rfloor \le n$? $\left\lfloor \frac{n+1}{2} \right\rfloor \le (n+1)/2$, and $(n+1)/2 \le n$ means $n+1 \le 2n$, or $1 \le n$, which is true. According to the induction hypothesis, W $\left(\left\lfloor \frac{n+1}{2} \right\rfloor \right) = \left\lfloor \lg\left(\left\lfloor \frac{n+1}{2} \right\rfloor \right) \right\rfloor + 1$,

so we have that $W(n+1) = 2 + \left\lfloor \lg\left(\left\lfloor \frac{n+1}{2} \right\rfloor \right) \right\rfloor$.

On the other hand, we want to show that $W(n+1) = \lfloor \lg(n+1) \rfloor + 1$, so it remains to show that

$$1 + \left\lfloor \lg\left(\left\lfloor \frac{n+1}{2} \right\rfloor \right) \right\rfloor = \lfloor \lg(n+1) \rfloor \qquad (*)$$

Taking 2 for the base of the logarithm, and letting $n = 2^m - p$, where $0 \leq p < 2^{m-1}$. We have three cases:

- If p = 0, eq. (*) above becomes 1 + m – 1 = m, which is true.
- If p = 1, eq. (*) becomes 1 + m – 1 = m, also true.
- If p > 1, eq. (*) becomes 1 + m – 2 = m-1, also true. Q.e.d.

5) Suppose that, in Algorithm 2.1 (line 4), the splitting function is changed to mid = low;. Explain the new search strategy. Analyze the performance of this strategy and show the results using order notation.

Solution: The algorithm now is:

```
index location (index low, index high) {
    index mid;
    if (low > high)
        return 0;
    else {
            mid = low;
            if (x = = s [mid])
                    return mid
            else if (x < s[mid])
                    return location (low, mid-1);
            else
                    return location (mid+1, high)
    }
}
```

Complexity analysis: The worst case for this algorithm is when x is either the largest array element, or larger than the largest. In either case, n comparisons are needed, since always the branch ***location (mid+1, high)*** is taken. This is a Linear Search → Θ(n).

6) Write an algorithm that searches a sorted list of n items by dividing it into three sublists of almost n=3 items. This algorithm finds the sublist that might contain the given item and divides it into three smaller sublists of almost equal size. The algorithm repeats this process until it finds the item or concludes that the item is not in the list. Analyze your algorithm and give the results using order notation.

Solution:

Input: Positive integer 'n' Sorted array of Keys 'A' indexed from 1 to n a key '*x*'

Output: Location, the Location of *x* in 'A' (0 if *x* is no in 'A')

```
index location3 (int A[ ], index L, index H){
    index m1, m2;
    if (L > H) return 0;
    else {
        m1 = L + (H – L)/3;       //one third
        m2 = L + 2*(H – L)/3;     //two thirds
        if (x == A[m1])           return m1;
        elseif (x < A[m1])        return location3 (A, L, m1 –1);
        elseif (x < A[m2])        return location3 (A, m1 +1, m2 – 1);
        elseif (x == A[m2])       return m2;
        else                      return location3 (A, m2 + 1, H);
    }
}
```

Complexity analysis: Assuming $n = 3^k$, and counting only the operations of addressing the array, the worst case is when the right third is chosen at each call, in which case the recursion is T(n) = 2 + T(n/3). As in Example B.18, this is found to be **Θ(lg n)**, which can be extended to any n with the tools in Section B.4.

7) Use the divide-and-conquer approach to write an algorithm that finds the largest item in a list of n items. Analyze your algorithm, and show the results in order notation.

Solution:

```
int maximum (int low, int high) {
    if (low == high)
            return a[Low];
    int mid = (low + high)/2;              //integer division
    int L1 = maximum (low, mid);
    int L2 = maximum (mid + 1, high);
    if A[L1] > A[L2] then
            return L1
    else
            return L2
}
```

Example: Let A be this array:

7	3	8	13	9	14
1	2	3	4	5	6

Level 1 : $1 \neq 6 \rightarrow$ mid = $(1+6)/2 = 3 \rightarrow$ A[mid] = 8

Recursive calls:

Level 2:

7	3	8
1	2	3
	↑	

13	9	14
4	5	6
	↑	

4/2

10/2 = 5

7	3	8
1	2	↑

13	9	14

max

7	3	8

13	9	14

7 9 6

(0 + 2)/2 = 1

0, 1

7	9

6

2

1/2 0

Complexity analysis: $T(n) = 2T(n/2) + 1$, which yields $\Theta(n)$.

Section 2.2

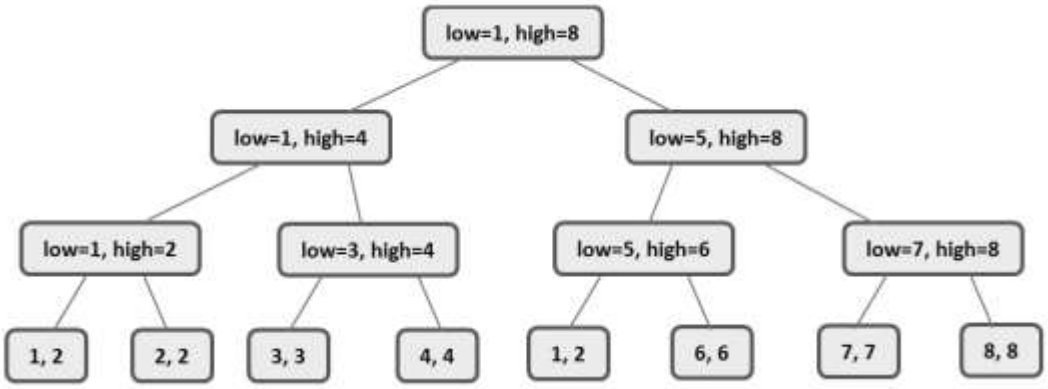
8) Starting with array S[] = { 123 34 89 56 150 12 9 240 } indexed from 1 to 8, sort S[] using Mergesort. Using algorithm mergesort2 on page62:

1. Begin with low = 1, high = 8; mid = (1+8)/2 = 4 (recursion level 0)
2. Recur with low = 1, high = 4; mid = (1+4)/2 = 2 (recursion level 1)
3. Recur with low = 1, high = 2; mid = (1+2)/2 = 1 (recursion level 2)
4. Both recursive calls to recursion level 3 with (low = 1, high = 1), and (low = 2, high = 2) return since low==high
5. Call merge2 with low = 1, mid = 1, high = 2
6. merge2 produces partially sorted array S[] = { 34 123 89 56 150 12 9 240 }
7. Return from recursion level 2
8. Recur with low = 3, high = 4; mid = (3+4)/2 = 3 (recursion level 2)

9. Both recursive calls to recursion level 3 with (low = 3, high = 3), and (low = 4, high = 4) return since low==high
10. Call merge2 with low = 3, mid = 3, high = 4
11. merge2 produces partially sorted array S[] = { 34 123 56 89 150 12 9 240 }
12. Return from recursion level 2
13. Call merge2 with low = 1, mid = 2, high = 4
14. merge2 produces partially sorted array S[] = { 34 56 89 123 150 12 9 240 }
15. Return from recursion level 1
16. Recur with low = 5, high = 8; mid = (5+8)/2 = 6 (recursion level 1)
17. Recur with low = 5, high = 6; mid = (1+2)/2 = 5 (recursion level 2)
18. Both recursive calls to recursion level 3 with (low = 5, high = 5), and (low = 6, high = 6) return since low==high
19. Call merge2 with low = 5, mid = 5, high = 6
20. merge2 produces partially sorted array S[] = { 34 56 89 123 12 150 9 240 }
21. Return from recursion level 2
22. Recur with low = 7, high = 8; mid = (1+2)/2 = 7 (recursion level 2)
23. Both recursive calls to recursion level 3 with (low = 7, high = 7), and (low = 8, high = 8) return since low==high
24. Call merge2 with low = 7, mid = 7, high = 8
25. merge2 produces partially sorted array S[] = { 34 56 89 123 12 150 9 240 }
26. Return from recursion level 2
27. Call merge2 with low = 5, mid = 6, high = 8
28. merge2 produces partially sorted array S[] = { 34 56 89 123 9 12 150 240 }
29. Return from recursion level 1
30. Call merge2 with low = 1, mid = 4, high = 8
31. merge2 produces sorted array S[] = { 9 12 34 56 89 123 150 240 }

9) Give the tree of recursive calls in Exercise 8.

Solution:



10) Write for the following problem a recursive algorithm whose worst-case time complexity is not worse than $\Theta(n \cdot \lg n)$. Given a list of n distinct positive integers, partition the list into two sublists, each of size n/2, such that the difference between the sums of the integers in the two sublists is maximized. You may assume that n is a multiple of 2.

Solution: The difference is maximized if one list contains all the elements less than the median, and the other list contains all the elements larger.

A simple solution is to sort the list using a $\Theta(n \cdot \lg n)$ algorithm (e.g. Mergesort, Quicksort), and then split it into two halves.

One could object that sorting is overkill for this problem: we don't need the two halves to be sorted, after all. If we could find the median in another, easier way, we could then partition in $\Theta(n)$ by using the Partition algorithm from Quicksort (Algorithm 2.7). Indeed, the Median of Medians algorithm allows us to find the k-th largest element of an unsorted sequence in **linear** time $\Theta(n)$. The algorithm was invented by Blum, Floyd, Pratt, Rivest, and Tarjan in their 1973 paper "Time Bounds for Selection", and thus is sometimes referred to as the BFPRT algorithm.

The original paper and many good explanations of BFPRT, including pseudocode, are freely available on the Web.

11) void nonrecursiveMergesort(int n, keytype S[]) {

```
    index i, j;
    for(i=1; i<=n; i = i*2)
        for(j=1; j<=n; j = j + 2*i)
            merge2(j, j+k, j+2*k, S[]);
}
```

The merge2 method is specified on page 63. We make the small modification of passing S[] as a parameter.

12) Show that the recurrence equation for the worst-case time complexity for Mergesort (Algorithms 2.2 and 2.4) is given by

$$W(n) = W\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + W\left(\left\lceil \frac{n}{2} \right\rceil\right) + n - 1$$

when n is not restricted to being a power of 2.

Solution: In Algorithm 2.4, we have two cases:

A. If n = (high – low + 1) is even, low + high is odd, so (low + high)/2 = (low + high)/2.0 – 0.5.

a. The array [low, mid] has length (low + high)/2.0 – 0.5 – low + 1 = (high-low)/2.0 + 0.5 = (high – low + 1)/2.0 = (high – low + 1)/2 = n/2 = $\left\lfloor \frac{n}{2} \right\rfloor$

b. Similarly, the array [mid+1, high] has length $\left\lceil \frac{n}{2} \right\rceil$

B. If n = (high – low + 1) is odd, low + high is even, so (low + high)/2 = (low + high)/2.0 .

a. The array [low, mid] has length (low + high)/2.0 – low + 1 = (high-low)/2.0 + 1 = (high – low + 1)/2.0 – 0.5 = $\left\lfloor \frac{n}{2} \right\rfloor$

b. Similarly, the array [mid+1, high] has length $\left\lceil \frac{n}{2} \right\rceil$

The same formula was proved for both cases, q.e.d.

13) Write an algorithm that sorts a list of n items by dividing it into three sublists of about n=3 items, sorting each sublist recursively and merging the three sorted sublists. Analyze your algorithm, and give the results under order notation.

Solution:

Input : Positive integer n, array of keys ***S***, indexed from 1 to n

Output :The array S containing the keys in nondecreasing order.

```
void mergesort3 (int n, keytype S[ ]) {
    if (n>1) {
        int h = floor(n/3), i = 2*h;
        int  m = i - h , t = n – i;
        int U[1…h], V[1…m], W[1…t]
        Copy S[1] through S[h]        to  U[1] through U[m]
        Copy S[h+1] through S[i]      to  V[1] through V[t]
        Copy S[i+1] through S[n]      to  W[1] through W[t]
        mergesort3 (h, U);
        mergesort3 (m, V);
        mergesort3 (t, W);
        merge (h, m, t, U, V, W, S);
    }
}

void merge3 (int h, int m, int t, const int U[], const int V[], const int W[], int S[]) {
    //Similar to Algorithm 2.3
}
```

Section 2.3

14) Given the recurrence relation

$$T(n) = 7T\left(\frac{n}{5}\right) + 10n \qquad \text{for } n > 1$$
$$T(1) = 1$$

find T(625).

Solution:

$T(625) = 7T(\frac{625}{5}) + 10(625) = 7T(125) + 6{,}250$

$T(125) = 7T(\frac{125}{5}) + 10(125) = 7T(25) + 1{,}250$

$T(25) = 7T(\frac{25}{5}) + 10(25) = 7T(5) + 250$

$T(5) = 7T(\frac{5}{5}) + 10(5) = 7T(1) + 50$, and $T(1) = 1$.

Substituting back, we have: T(5) = 57, T(25) = 649, T(125) = 5,793, T(625) = **46,801**

15)

Consider algorithm *solve* given below. This algorithm solves problem P by finding the output (solution) O corresponding to any input I.

```
void solve (input I, output& O)
{
  if (size(I) == 1)
     find solution O directly;
  else{
     partition I into 5 inputs I1, I2, I3, I4, I5, where
     size (Ij) = size(I)/3 for j = 1,..., 5;
     for (j = 1; j <= 5; j++)
        solve(Ij,Oj);
     combine O1, O2, O3, O4, O5 to get O for P with input I;
  }
}
```

Assume g(n) basic operations for partitioning and combining, and none for an instance of size 1.

(a) Write a recurrence equation T(n) for the number of basic operations needed to solve P when the input size is n.

Solution: **$T(n) = 5T(n/3) + g(n)$**

(b) What is the solution to this recurrence equation if $g(n) \in \Theta(n)$? (Proof is not required.)

Solution: We assume a linear function $g(n) = a \cdot n + b$, and n a power of 3: $n = 3^k$. The recursion is

$$T(3^k) = 5T(3^{k-1}) + a \cdot 3^k + b$$

We denote $T(3^k) = x_k$, and the recursion becomes

$$x_k - 5x_{k-1} = a \cdot 3^k + b$$

Which is solved as a non-homogeneous recursion as explained in Theorem B.3:

The solution to the homogeneous part is $r_5 = 5$, and the two non-homogeneous terms each contribute a root: $r_1 = 1$, and $r_3 = 3$. The general form of the solution is:

$$\mathbf{x_k = C_1 1^k + C_3 3^k + C_5 5^k}$$

If need be, the constants are found by making use of the first 3 terms of the recursion: $x_0 = 0$, $x_1 = 3a + b$, $x_2 = 24a + 6b$. We solve the linear system of 3 eqns. with 3 unknowns to find $C_3 = -3a/2$, etc.

(c) Assuming that $g(n) = n^2$, solve the recurrence equation exactly for n =27.

Solution: By direct substitution, we find T(1) = 0, T(3) = 9, t(9) = 126, **$T(27) = 1359$**.

(d) Find the general solution for n a power of 3.

Solution: We still assume $g(n) = n^2$, as above. The recursion is $T(n) = 5T(n/3) + n^2$, so we substitute $n = 3^k$, and rename $T(3^k) = x_k$ to obtain a new recursion $x_k - 5x_{k-1} = 9^k$. This is solved as a non-homogeneous recursion (Theorem B.3), with the solution $x_k = C_5 5^k + C_9 9^k$. The constants are determined by solving the linear system $x_0 = 0$, $x_1 = 9$, and we have $C_5 = -9/4$, $C_9 = 9/4$.

The final solution is **$T(3^k) = x_k = -9/4 \cdot 5^k + 9/4 \cdot 9^k$**

16) Suppose that, in a divide-and-conquer algorithm, we always divide an instance of size n of a problem into 10 sub-instances of size n=3, and the dividing and combining steps take a time in $\Theta(n^2)$. Write a recurrence equation for the running time T (n), and solve the equation for T (n).

Solution: $T(n) = 10T(n/3) + Cn^2$, where C is a constant. Assuming that n is a power of 3, we have $n = 3^k$.

$T(3^k) = 10T(3^{k-1}) + C9^k$, and we change the recursion by denoting $x_k = T(3^k) \rightarrow x_k = 10x_{k-1} + C9^k$. This is a non-homogeneous recursion, solved as in Theorem B.3: there are two roots: 9 and 10, so the solution has the form $C_9 9^k + C_{10} 10^k$. The constants are determined by plugging in the first two values of k: k = 0 and k = 1. We assume $x_0 = 0$, meaning that the instance n=1 needs no operations to solve. This means $x_1 = 9C$. Solving for C_9 and C_{10} we have $C_9 = -9C$, $C_{10} = 9C$. The solution is $x_k = 9C(10^k - 9^k)$.

17) Divide-and-conquer algorithm for Towers of Hanoi with n disks.

Solution:

```
void ToH(int n, char x, char y, char z) {
if (n > 0) {
    ToH (n−1, x, y, z);
    cout  << x  << " moves to " <<  y << endl;
    ToH (n−1, z, y, x);
  }
}
```

(a) Show for your algorithm that S (n) = 2^{n-1}. (Here S (n) denotes the number of steps (moves), given an input of n disks.)

Solution: The recursion equation is T(n) = 2T(n-1) + 1 for n > 1, T(1) = 1. By direct substitution, we have T(n) = 2·(2·(…) + 1) + 1 = 2^{n-1} +2^{n-2} + … + 2 + 1, which, using the sum of the geometric series from Example A.4, is 2^n – 1.

(b) Prove that any other algorithm takes at least as many moves as given in part (a).

Solution: In order to move the largest disk (call it Dn), any other algorithm A' must first place disks 1 through n-1 on the intermediate peg, then perform at least one move to place Dn on the destination peg, and then move disks 1 through n-1 on the destination peg, so we have that T'(n) ≥ 2T'(n-1) + 1, for n > 1.

The boundary condition is T'(1) > 1.

If we now write the sum from the solution to part (a), it's clear that T'(n) ≥ T(n).

18) When a divide-and-conquer algorithm divides an instance of size n of a problem into sub-instances each of size n=c, the recurrence relation is typically given by

$$T(n) = aT\left(\frac{n}{c}\right) + g(n) \qquad \text{for } n > 1$$
$$T(1) = d$$

where g (n) is the cost of the dividing and combining processes, and d is a constant.

Let n = c^k.

(a) Show that

$$T\left(c^k\right) = d \times a^k + \sum_{j=1}^{k} \left[a^{k-j} \times g\left(c^j\right)\right].$$

Solution: Induction on k:

Base case: When k = 0, we have T(1) = dx1 + 0 = d → correct.

Induction step: $T(c^{k+1}) = aT(c^k) + g(c^{k+1})$, and we substitute the formula for $T(c^k)$:

$T(c^{k+1}) = a\left(d \times a^k + \sum_{j=1}^{k} a^{k-j} \times g(c^j) \right) + g(c^{k+1}) =$

$= d \times a^{k+1} + \sum_{j=1}^{k} a^{k+1-j} \times g(c^j) + a^{k+1-(k+1)} \cdot g(c^{k+1}) =$

$= d \times a^{k+1} + \sum_{j=1}^{k+1} a^{k+!-j} \times g(c^j)$ Q.e.d.

(b) Solve the recurrence relation given that $g(n) \in \Theta(n)$.

Solution: Assume $g(n) = E \cdot n$. Denoting $T(c^k)$ by x_k, we have the new recursion

$$x_k = ax_{k-1} + Ec^k ,$$

which is solved as a non-homogeneous recursion (Theorem B.3). The roots are a and c, and we consider two cases:

- If $a \neq c$, the solution has the form $x_k = C_1 a^k + C_2 c^k$, and the constants are determined by using the eqns. $x_0 = d$ and $x_1 = ad + Ec$. We find $C_1 = d + cE/(a-c)$, $C_2 = -cE/(a-c)$.
- If $a = c$, we can either apply the non-homogeneous case with the double root a, or expand the recursion directly to find $x_k = (d+kE)a^k$.

Section 2.4

19) Use Quicksort (Algorithm 2.6) to sort the following list. Show the actions step by step.

123 34 189 56 150 12 9 240

Solution:

i	j	S[0]	S[1]	S[2]	S[3]	S[4]	S[5]	S[6]	S[7]
–	–	123	34	189	56	150	12	9	240
1	7	123	34	189	56	150	12	9	240
2	6	123	34	9	56	150	12	189	240
3	5	123	34	9	56	150	12	189	240
5	4	123	34	9	56	150	12	189	240
–	–	123	34	9	56	150	12	189	240

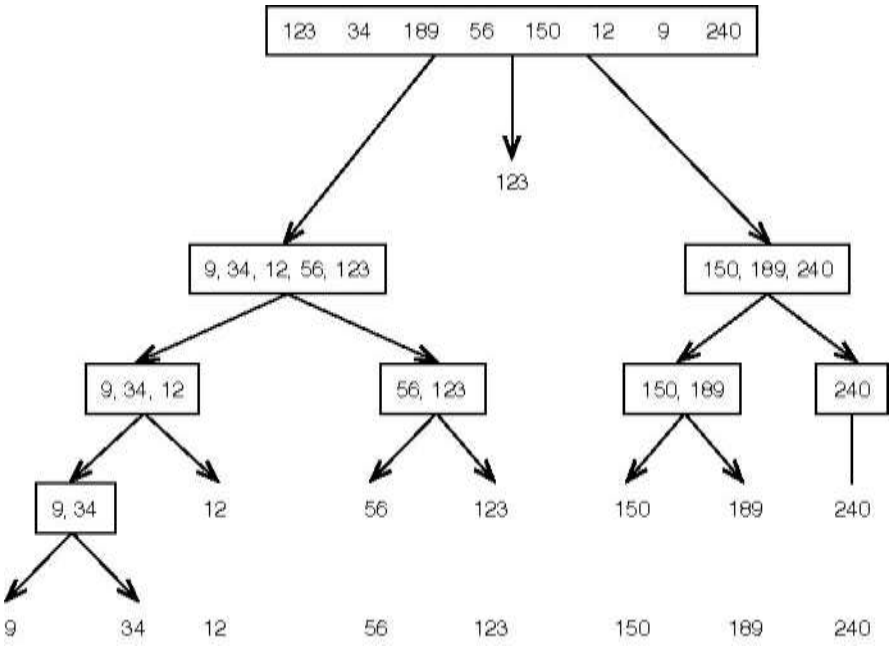
I	j	S[0]	S[1]	S[2]	S[3]	S[4]
0	4	123	34	9	56	12
1	2	12	34	9	56	123
2	1	9	34	12	56	123

I	J	S[5]	S[6]	S[7]
5	7	150	189	240
6	6	150	189	240

9	34	12	56	123	66	150	189	240

20) Give the tree of recursive calls in Exercise 19.

Solution:



21) Show that if $W(n) \leq \frac{(p-1)(p-2)}{2} + \frac{(n-p)(n-p-1)}{2} + n - 1,$ then

$$W(n) \leq \frac{n(n-1)}{2} \qquad \text{for } 1 \leq p \leq n.$$

This result is used in the discussion of the worst-case time complexity analysis of Algorithm 2.6 (Quicksort).

Solution:

$$w(n) \leq \frac{(p-1)(p-2)}{2} + \frac{(n-p)(n-p-1)}{2} + (n-1)$$

$$\leq \frac{(n-1)(n-2)}{2} + 0 + \frac{2(n-1)}{2} = \frac{n^2 - 3n + \cancel{2} + 2n \cancel{-2}}{2} = \frac{n^2 - n}{2} = \frac{n(n-1)}{2}$$

Second solution:

$$w(n) \leq \frac{(n-1)(n-2)}{2} + \frac{(n-p)(n-p-1)}{2} + \frac{2(n-1)}{2} =$$

$$= \frac{n2 - 3n + \cancel{2} + n^2 - np - n - pn + p^2 + p + 2n - \cancel{2}}{2}$$

$= 2n^2 - 2np - 2n + p^2\ p/2$

$= 2n^2 - 2n\ (p + 1) + p\ (p + 1)/2$

$= 2n^2 + (p - 2n)\ (p + 1)/2$

$= 2n^2 - n\ (n + 1)/2$

$= 2n^2 - n^2 - n = n^2 - n/2$

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

22) Verify the following identity

$$\sum_{p=1}^{n} [A(p-1) + A(n-p)] = 2 \sum_{p=1}^{n} A(p-1).$$

This result is used in the discussion of the average-case time complexity analysis of Algorithm 2.6 (Quicksort).

Solution: Induction on n.
Base case: n = 1 → A(0) + A(0) = 2A(0) → correct.
Induction step: We split the sum for n+1 so that we isolate a sum for n:

$$\sum_{p=1}^{n+1}[A(p-1)+A(n-(p-1))] \quad = \quad \sum_{p=1}^{n} A(p-1) + A(n) + A(n) + \sum_{p=1}^{n} A(n-p) \quad =$$

$$= \sum_{p=1}^{n}[A(p-1)+A(n-p)] + 2A(n)$$

By the induction hypothesis, the first sum is $2\sum_{p=1}^{n} A(p-1)$, and, when we add 2A(n), the sum is complete up to n+1.

23) Write a non-recursive algorithm for Quicksort (Algorithm 2.6). Analyze your algorithm, and give the results using order notation.

Solution: Non-Recursive Quick sort Algorithm by using a stack.

```
struct Stack {
      int low, high:
};
void quick-sort (int a[], int n) {  //n elements in the array
   struct Stack S[100];
   int top, i, j, pivot, lo, hi;
   top = 0;
   S[top].low = 0;             //first array index
   S[top].high = n - 1;        //last array index
   top++;                      //pushing
   while (top > 0) {           //stack is not empty
      lo = s[top].low;
      hi = s[top].high;
      top--;                   //pulling sub-problem off stack
      if (lo >= hi) continue; //empty problem
      pivot = a[lo];           //pivot is leftmost element
      i = lo;
      j = hi;
      while (i < j) {
        while (i < hi && a[i] <= k)
             i++;
        while (j < lo && a[j] >= k)
```

```
            j--;
        if (i < j)
            swap(i,j);
      }
      if (lo! = j)
        swap(lo, j);          //place pivot in the "middle"
      }
      S[top].low = j + 1;
      S[top].high = hi;
      top++;                  //pushing right sub-array
      S[top].low = lo;
      S[top].high = j - 1;
      top++;                  //pushing left sub-array
   }// end of while
}
```

The operation is identical to that of the recursive Quicksort, so it has the same complexity: $\Theta(n^2)$ in the worst case and $\Theta(n \cdot \lg n)$ on average.

24) Assuming that Quicksort uses the first item in the list as the pivot item:

(a) Give a list of n items (for example, an array of 10 integers) representing the worst-case scenario.

(b) Give a list of n items (for example, an array of 10 integers) representing the best-case scenario.

Solution:

(a) { 1 2 3 4 5 6 7 8 9 10 }

(b) { 6 3 8 2 7 5 1 9 4 10 }

Section 2.5

25) Show that the number of additions performed by Algorithm 1.4 (MatrixMult)

can be reduced to $n^3 - n^2$ after a slight modification of this algorithm.

Solution:

The loops Algorithm 1.4 are modified thus:

```
for (i=1; i<=n; i++)
      for (j=1; j<=n; j++)
        C[i][j] = C[i][1] * C[1][j];
        for (k=2; k<=n; k++)
             C[i][j] = C[i][j] + C[i][k] * C[k][j];
```

This way, one less addition is performed per innermost for loop, and, since that loop executes n^2 times, the total number of additions is n^2 less: $n^3 - n^2$.

26) In Example 2.4, we gave Strassen's product of two 2×2 matrices. Verify the correctness of this product.

Solution: We verify c_{11}, which from regular matrix multiplication is $a_{11}b_{11} + a_{12}b_{21}$.

We substitute in Strassen's formula: $c_{11} = m_1 + m_4 - m_5 + m_7 =$

$= [a_{11} + a_{22}] [b_{11} + b_{22}] + a_{22} [b_{21} - b_{11}] - [a_{11} + a_{12}] b_{22} + [a_{12} - a_{22}] [b_{21} + b_{22}] =$

$= \mathbf{a_{11}b_{11}} + a_{11}b_{22} + a_{22}b_{11} + a_{22}b_{22} + a_{22}b_{21} - a_{22}b_{11} - a_{11}b_{22} - a_{12}b_{22} + \mathbf{a_{12}b_{21}} + a_{12}b_{22} - a_{22}b_{21} - - a_{22}b_{22}$

All term simplify our except for $a_{11}b_{11} + a_{12}b_{21}$, which is the correct expression.

The positions c_{12}, c_{21}, c_{22} are verified similarly by substitution.

27) When n=64, n^3 = 262,144

28) When n=64, $n^{\log_2 7}$ = 117,649

29) Write a recurrence equation for the modified Strassen's algorithm developed by Shmuel Winograd that uses 15 additions/subtractions instead of 18. Solve the

recurrence equation, and verify your answer using the time complexity shown at the end of Section 2.5.

Solution: The recursion is the one on p.75, with 15 instead of 18:

$$\begin{cases} T(n) = 7T(n/2) + 15(n/2)^2, & \text{if } n>1 \text{ (with } n = 2^k) \\ T(1) = 0 \end{cases}$$

The solution follows closely that in Example B.20:

By substituting x_k instead of $T(2^k)$, we obtain the recursion $x_k = 7x_{k-1} + 15\cdot 4^{k-1}$, which is non-homogeneous, with roots 7 and 4. The solution has the form $x_k = c_1 7^k + c_2 4^k$.

We find the constants by plugging in the values for x_0 $T(1) = 0$, and $x_1 = T(2) = 15$. We obtain $c_1 = 5$, $c_2 = -5$, so $x_k = 5\cdot 7^k - c_2\cdot 4^k$.

Going back to the variable n, we have $4^k = n^2$, and $7^k = (2^{\log_2(7)})^k = (2^k)^{\log_2(7)} = n^{2.8073\ldots}$, so

$T(n) = 5n^{2.8073} - 5n^2$, which corresponds to the formula on p.76 of the text.

Section 2.6

30) Use Algorithm 2.10 (Large Integer Multiplication 2) to find the product of 1253 and 23,103.

Solution:

u = 1253, v = 23103, and we calculate u × v.

No. of digits in u = 4 No. of digits in v = 5 We choose *threshold* = 2 digits.

Level 1: n = max (4, 5) = 5 m = floor(n/2) = 5/2 = 2 → $10^m = 100$

x = 12 y = 53 w = 231 z = 3

r = prod2(65, 234) → recursive call → 15,210 (see Level 2(a) below)

p = prod2(12, 231) → recursive call → 2,772 (see Level 2(b) below)

q = prod2(53, 3) → calculated directly → 53 x 3 = 159

return 2,772 x 10,000 + (15,210 – 2,772 – 159) x 100 + 159 = **28,948,059**

Level 2: (a) prod2(65, 234) → n = 3 > *threshold*, m = floor(n/2) = 1 → $10^m = 10$

x = 6, y = 5, w = 23, z = 4

r = prod2 (11, 27) → directly 11 x 18 = 297

p = prod2 (6, 23) → directly 6 x 23 = 138

q = prod2 (5, 4) → directly 5 x 4 = 20

return 138 x 100 + (297 – 138 – 20) x 10 + 20 = 15,210

(b) prod2(12, 231) → n = 3 > *threshold*, m = floor(n/2) = 1 → $10^m = 10$

x = 1, y = 2, w = 23, z = 1

r = prod2(3, 24) → directly 3 x 24 = 72

p = prod2(1, 23) → directly 1 x 23 = 23

q = prod2(2, 1) → directly 2 x 1 = 2

return 23 x 100 + (72 – 23 – 2) x 10 + 2 = 2,772

31) How many multiplications are needed to find the product of the two integers in Exercise 30?

Solution: We count 7 multiplications in the solution of Ex.30 (only the ones labeled *directly*, not counting the multiplications with powers of 10, which are linear time – see Ex. 32)

32) Write algorithms that perform the operations

u x 10^m; u divide 10^m; u rem 10^m,

where u represents a large integer, m is a nonnegative integer, divide returns the quotient in integer division, and rem returns the remainder. Analyze your algorithms, and show that these operations can be done in linear time.

Solution: These are some possible algorithms. Remember that the large numbers are represented as arrays of integers (digits), as explained in Section 2.6.1.

i) For **u × 10^m**:

- Find the end of the array u; //constant time if we maintain the index of the end
- Insert m zero digits; //Θ(m)

(ii) For **u divide 10^m**:

- Find the end of the array u; // constant time if we maintain the index of the end
- Delete the last m digits; //Θ(m)

(iii) For **u rem 10^m**:

- Find the end of the array u; // constant time if we maintain the index of the end
- Copy the last m digits in new array; //Θ(m)

33) Modify Algorithm 2.9 (Large Integer Multiplication) so that it divides each n-digit integer into
(a) three smaller integers of n=3 digits (you may assume that n = 3k).
(b) four smaller integers of n=4 digits (you may assume that n = 4k).
Analyze your algorithms, and show their time complexities in order notation.

Solution:

(a) large_integer **prod3** (large_integer u, large_integer v) {

large_integer *x*1, y1, z1, *x*2, y2, z2, temp;

int n, m;

n = max (nr. of digits in u, nr. of digits in v)

if (u==0 || v==0) return 0;

else if (n <= threshold) return (u × v);

else {

m = floor(n/3);

*x*1 = u divide 10^{2m} ; temp = u rem 10^{2m}; y1 = temp divide10^{m}; z1 = temp rem 10^{m};

*x*2 = v divide 10^{2m} ; temp = u rem 10^{2m}; y2 = temp divide10^{m}; z2 = temp rem 10^{m};

return

prod3 (*x*1, *x*2) × 10^{4m} +

(prod3 (*x*1, y2) + prod3 (*x*2, y1)) × 10^{3m} +

(prod3 (*x*1, z2) + (prod3 (*x*2, z1) + (prod3 (y1, y2)) × 10^{2m} +

(prod3 (y1, z2) + (prod3 (z1, y2)) × 10^{m} + (prod3 (z1, z2);

}

}

Complexity analysis: W(n) = 9W(n/3) + c·n for n > 1, n = 3^k, and W(1) = 1.

Solving it as a non-homogeneous equation with the substitution T(3^k) = x_k, we have the roots 9 and 3, so the dominant term in the solution is $9^k = (3^k)^2 = n^2$. The algorithm is $\Theta(n^2)$.

(b) The function large_integer **prod4** (large_integer u, large_integer v)

is written in a manner similar to **prod3**() above. Each argument is now broken into four parts: w1, x1, y1, z1, and w2, x2, y2, z2.

Complexity analysis: The recurrence is

$W(n) = 16W(n/4) + c \cdot n$ for $n > 1$, $n = 4^k$, and $W(1) = 1$.

Solving it as a non-homogeneous equation with the substitution $T(4^k) = x_k$, we have the roots 16 and 4, so the dominant term in the solution is $16^k = (4^k)^2 = n^2$. The algorithm is still $\Theta(n^2)$.

Section 2.7

34) Implement both Exchange Sort and Quicksort algorithms on your computer to sort a list of n elements. Find the lower bound for n that justifies application of the Quicksort algorithm with its overhead.

Solutions and bounds will vary.

35) Implement both the standard algorithm and Strassen's algorithm on your computer to multiply two n x n matrices ($n = 2^k$). Find the lower bound for n that justifies application of Strassen's algorithm with its overhead.

Solutions and bounds will vary.

36) Suppose that on a particular computer it takes $12n^2$ μs to decompose and recombine an instance of size ***n*** in the case of Algorithm 2.8 (Strassen). Note that this time includes the time it takes to do all the additions and subtractions. If it takes n^3 μs to multiply two ***n x n*** matrices using the standard algorithm, determine thresholds at which we should call the standard algorithm instead of dividing the instance further. Is there a unique optimal threshold?

Solution: At the end of Section 2.3, there are mentioned three possible methods to deal with the cases where n is not a power of 2; this solution uses the second one, i.e. if at some step of the algorithm n is odd, we pad the matrix with one row and one column of zeroes to make the size even. The recursion is:

$$T(n) = \begin{cases} 7T\left(\left\lceil \frac{n}{2} \right\rceil\right) + 12n^2, & \text{if n is even} \\ 7T\left(\left\lceil \frac{n}{2} \right\rceil\right) + 12(n+1)^2, & \text{if n is odd} \end{cases}$$

We find the threshold ***t*** separately for n even and odd. In either case, $\left\lceil \frac{t}{2} \right\rceil < t$, so the recursion at $\left\lceil \frac{t}{2} \right\rceil$ is $\left\lceil \frac{t}{2} \right\rceil^3$.

For n even, we have $7/8 \cdot t^3 + 12t^2 = t^3$, with solution t = 96.

For n odd, we have $7((t+1)/2)^3 + 12(t+1)^2 = t^3$, which leads to a cubic equation. Solving numerically, we find t = 118.8, so for n ≥ 119 we split, but for n ≤ 117 we take the direct approach.

Since the thresholds are different, there is no unique optimal threshold: in between 96 and 118 we take different decisions based on whether n is odd or even.

Section 2.8

37) Use the divide-and-conquer approach to write a recursive algorithm that computes n!. Define the input size (see Exercise 36 in Chapter 1), and answer the following questions.
Does your function have an exponential time complexity?
Does this violate the statement of case 1 given in Section 2.8?

Solution: Input: n and m (m = 1 for first call)
Output: n!

```
integer factorial (n, m) {
    if (n==m)
        return n;
    else {
        if (m > n)
            return 1;
        else
            return (factorial (n, ⌊(n+m)/2⌋) × factorial (⌊(n+m)/2⌋ + 1, m));
    }
}
```

Complexity analysis:

In the above algorithm, an instance of size ***n – m + 1*** is divided into two instances, each of almost size ***(n – m + 1)/2***. The recurrence is T(n) = 2T(n/2) + 1, which is easily solved as a non-homogeneous equation (Theorem B.3) or by direct expansion to yield $T(n) = n - 1 \in \Theta(n)$.

If, however, we define the input size as the number of bits required to represent the input ***n*** (as in Exercise 36 in Chapter 1), then:

- The input size is $\log_2 n$.
- The output size is $\log_2(n!) = \log_2(2) + \log_2(3) + \ldots + \log_2(n/2 - 1) + \log_2(n/2) + \ldots + \log_2(n)$.
- Let $n = 2^k$
  - The input size is k
  - The output size is $\log_2(2) + \log_2(3) + \ldots + \log_2(n/2 - 1) + (k - 1) + \ldots + k$.
  - Since the sequence is strictly increasing, the sum above is less than
  - $(k - 1) + (k - 1) + \ldots + (k - 1) = (k - 1)\cdot n/2 = (k - 1)2^{k-1}$.
- Therefore, the relationship between input and output is $k \rightarrow \Omega(k \cdot 2^k)$, which means a **super-exponential** time complexity.

The statement of case 1 in Section 2.8 is <u>not</u> violated, since the two instances are <u>not</u> each of size almost equal to the input size; as seen in the last sum above, the second instance is $(k - 1)2^{k-1}$, that is super-exponential in the input k.

38) Suppose that, in a divide-and-conquer algorithm, we always divide an instance of size n of a problem into n sub-instances of size n/3, and the dividing and combining steps take linear time. Write a recurrence equation for the running time T (n), and solve this recurrence equation for T (n). Show your solution in order notation.

<u>Solution</u>: $T(n) = n(T(n/3) + C)$, for $n > 1$, and $T(1) = 0$.

We consider $n = 3^k$ and substitute $T(3^k)$ by x_k to obtain the recurrence $x_k = 3^k(x_{k-1} + C)$. This hasn't any of the general forms covered in Appendix B, so we perform direct substitution:

After a few steps, a pattern emerges. Below is the expression after *m* substitutions:

$x_k = 3^{k+(k-1)+\ldots+(k-m+1)}x_{k-m} + [3^{k+(k-1)+\ldots+(k-(m-1))} + 3^{k+(k-1)+\ldots+(k-(m-2))} + \ldots + 3^{k+(k-1)} + 3^k]\cdot C$, and so, after k-1 substitutions, we have:

$x_k = 3^{k+(k-1)+\ldots+2+1}x_0 + [3^{k+(k-1)+\ldots+2+1} + 3^{k+(k-1)+\ldots+2} + \ldots + 3^{k+(k-1)} + 3^k]\cdot C =$

$= \quad 0 \quad + [3^{k+(k-1)+\ldots+2+1} + 3^{k+(k-1)+\ldots+2} + \ldots + 3^{k+(k-1)} + 3^k]\cdot C =$

$= 3^k \cdot [3^{(k-1)+\ldots+2+1} + 3^{(k-1)+\ldots+2} + \ldots + 3^{(k-1)} + 1]\cdot C$

Complexity analysis: The first exponent in the bracket above is k(k-1)/2, as shown in Example A.1. We replace all the terms in the bracket with the first one, obtaining the inequality

$$x_k < 3^k(k\text{-}1)3^{k(k-1)/2}C \approx n(\lg n)n^{(\lg n)/2},$$

so the algorithm is $O(\, n^{1+(\lg n)/2} \lg n)$.

Additional Exercises

39) Implement both algorithms for the Fibonacci Sequence (Algorithms 1.6 and 1.7). Test each algorithm to verify that it is correct. Determine the largest number that the recursive algorithm can accept as its argument and still compute the answer within 60 seconds. See how long it takes the iterative algorithm to compute this answer.

```
/*Program that times Fibonacci function calls*/
#include <time.h>
#include <iostream>
using namespace std;

//Recursive fibonacci
int fibo_rec(int n){
   if ( n == 0 || n == 1 )       /* base case */
      return n;
   else                                        /* recursive step */
      return fibo_rec(n - 1) + fibo_rec(n - 2);
}

//Iterative Fibonacci
int fibo_iter(int n){
    if ( n == 0 || n == 1 )      /* base case */
      return n;
    int minus_two = 0;
    int minus_one = 1;
    int fi;
    for (int i=2; i<=n; i++){
           fi = minus_two + minus_one;
           minus_two = minus_one;
           minus_one = fi;
    }
    return fi;
}

int main(){
    time_t begin, end;
    int N;
    cout << ("Enter positive integer: ");
    cin  >> N;

    begin = time(NULL);
    cout << "Fibonacci recursive: " << fibo_rec(N) << endl;
```

```
end    = time(NULL);
cout << "\ttime = " << difftime(end, begin) << " s" << endl;

begin = time(NULL);
cout << "Fibonacci iterative: " << fibo_iter(N) << endl;
end    = time(NULL);
cout << "\ttime = " << difftime(end, begin) << " s" << endl <<endl;

return 0;
}
```

As seen from the following output, on the particular machine on which the program was run, N=44 was the largest Fibonacci number that required less than 60 s recursively. The iterative solution took less than 1 s (and probably actually less than 1 microsecond!).

```
Enter positive integer: 44
Fibonacci recursive: 701408733
        time = 54 s
Fibonacci iterative: 701408733
        time = 0 s
```

```
Enter positive integer: 45
Fibonacci recursive: 1134903170
        time = 88 s
Fibonacci iterative: 1134903170
        time = 0 s
```

40) Problem: Search for a key x in a sorted two-dimensional array of m × n values. The array is sorted by row and column.

Input: positive integers n and m, two-dimensional sorted array of keys S[][] indexed by pairs (1, 1) through (n,m), address of two integer variables x and y to store result.

Output: index pair of key if found, (0,0) otherwise.

```
void find(int n, int m, keytype S[][], int &x, int &y) {
    index low = 1, high = n, mid = (low+high)/2;
    while(low <= high) {
            if(S[mid][1] <= key && key <= S[mid][m-1]) { break; }
            if(S[mid][m-1] < key)  { low = mid + 1; }
            else  { high = mid – 1; }
            mid = (low+high)/2;
    }
    x = mid;
    low = 1;
    high = m-1;
    mid = (low + high)/2;
    while(low <= high) {
```

```
            if(S[x][mid] == key) {
                y = mid;
                return;
            }
            if(S[x][mid] < key) { low = mid + 1; }
            else { high = mid – 1; }
            mid = (low+high)/2;
        }
        x = 0;
        y = 0;
    }
```

Time complexity is $O(\log_2 n + \log_2 m)$.

41) Suppose that there are $n = 2^k$ teams in an elimination tournament, in which there are n=2 games in the first round, with the $n/2 = 2^{k-1}$ winners playing in the second round, and so on.

(a) Develop a recurrence equation for the number of rounds in the tournament.

(b) How many rounds are there in the tournament when there are 64 teams?

(c) Solve the recurrence equation of part (a).

Solution:

(a) Obviously, with 1 team, there are no rounds needed, so $T(1) = 0$. With 2 teams, one round is needed, so $T(2) = 1$. In general, $T(2^k) = 1 + T(2^{k-1})$.
(b) $T(64) = T(2^6) = 1 + 1 + 1 + 1 + 1 + 1 + T(1) = 6$.
(c) Use simple induction on k to show that $T(2^k) = k$, so in terms of n we have $T(n) = \log_2 n$.

42) Problem: Tile an $n \times n$ board (where n is a power of 2) with trominoes so that only one given square x on the board is left uncovered and no two trominoes overlap.

Input: integer n, location x of square to be left uncovered

Output: a board tiled according to the problem specifications

```
void tile(int n, location x) {
```

```
    if(n==2) {
        tile with one tromino so that all rules are obeyed (only one possibility);
        return;
    }
    divide board in to four n/2 × n/2 sub-boards;
    place one tromino so that each of its three squares covers one square in each of the
            three sub-boards that do not contain x;
    let x2, x3, and x4 be the locations of the squares in each sub-board that have been
            covered by the placed tromino;
    tile(n/2, x);
    tile(n/2, x2);
    tile(n/2, x3);
    tile(n/2, x4);
}
```

43) a) Divide the nine coins into 3 groups of 3.

Compare the weights of two of the groups (first weighing).

If either of the groups is heavier, choose the heavier group.

If both groups are equal, choose the third group that wasn't weighed.

Compare the weights of two coins in the heavier group (second weighing).

If one is heavier, that is the counterfeit coin.

If both are equal, the un-weighed coin is the counterfeit coin.

b) Problem: Given n coins indexed from 1 to n, one of which is counterfeit and heavier than all the others, find the index of the counterfeit coin.

Input: integer n, n coins indexed from 1 to n

Output: index of the counterfeit coin

```
int findCounterfeit(int n) {
        index i = n, j = 1;
        while(i>=3) {
                int w1 = weight(j, j + i/3 - 1);
                int w2 = weight(j+i/3, j+2*i/3 - 1);
```

```
            if(w1 < w2)
                j = j + i/3;
            else if(w1 == w2)
                j = j + 2*i/3 + 1;
            i = i/3;
        }
        return j;
}
```

Here weight(x, y) returns the total weight of the coins of index x through index y.

The complexity of this algorithm is $O(\log_3 n)$.

44) Write a recursive $\Theta(n \cdot \lg n)$ algorithm whose parameters are three integers x, n, and p, and which computes the remainder when x^n is divided by p. For simplicity, you may assume that n is a power of 2, that is, that $n = 2^k$ for some positive integer k.

Solution: If we have two integers, m and n, we can divide each by p to obtain their remainders: $m = q1 \cdot p + r1$, $n = q2 \cdot p + r2$.

The product $m \cdot n$ is $(q1 \cdot p + r1)(q2 \cdot p + r2) = (q1 \cdot q2p + q1 \cdot r2 + q2 \cdot r1)p + r1 \cdot r2$, which shows that only $r1 \cdot r2$ must be considered when calculating the remainder of $m \cdot n$ modulo p:

$m \cdot n \,\%\, p = r1 \cdot r2 \,\%\, p$.

Back to our problem, we use the divide and conquer approach by splitting the problem thus: $x^n = x^{n/2} \cdot x^{n/2}$.

In general n is not even, we have to use floor and ceiling: $x^n = x^{\left\lfloor \frac{n}{2} \right\rfloor} \cdot x^{\left\lceil \frac{n}{2} \right\rceil}$, and then we recursively solve the two smaller instances until we reach the base case n = 1.

For complexity analysis, we have the recursion $T(n) = T(\left\lfloor \frac{n}{2} \right\rfloor) + T(\left\lceil \frac{n}{2} \right\rceil) + C$, where C is the (constant) cost of multiplying two integers and finding their remainder modulo p. If we now make the simplifying assumption $n = 2^k$, then the instances are always even, and we have the recursion $T(n) = 2T(n/2) + C$, which is solved as a non-homogeneous recursion (Theorem B.3): $T(n) = (2C-1)n - C \in \Theta(n)$. Using the techniques in Section B.4, we can prove that the algorithm is $\Theta(n)$ for all n.

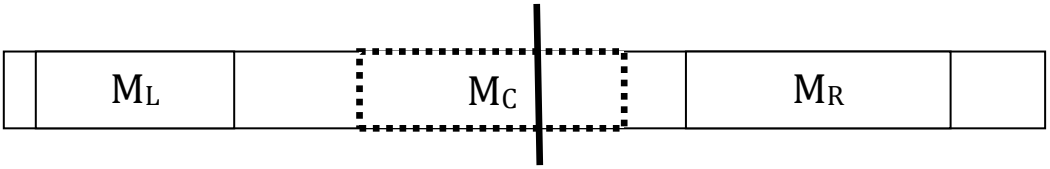
If we know in advance that $n = 2^k$, the algorithm becomes $\Theta(\lg n)$.

45) Use the divide-and-conquer approach to write a recursive algorithm that finds the maximum sum in any contiguous sublist of a given list of n real values.

Analyze your algorithm, and show the results in order notation.

Solution: This is known as the Maximum Subarray problem, and there are several algorithms that solve it. Here we present the divide-and-conquer algorithm published by John Bentley in the journal article *Programming Pearls* (Communications of the ACM, vol.27, 1984):

- Split the array in half, as suggested by the vertical line in the figure:



- Recursively calculate the maximum subarrays in the left and right halves, M_L, and M_R.
- Combine the solutions:
  - We have to take into account the possibility that the maximum subarray straddles the boundary, so we have to find M_C.
  - Return max(M_L, M_C, M_R).

Complexity analysis: Finding M_C is done in ***n*** comparisons (worst case), since all we need to do is move left and right from the boundary until we hit the first negative element on each side. The maximum of 3 elements is found with 2 comparisons.

We have the recursion $T(n) = 2T(n/2) + (n+2)$, which, as in the case of Mergesort, is proven to be $\Theta(n \cdot \lg n)$.