## Section 1.1

1) Write an algorithm that finds the largest number in a list (an array) of n numbers.

Input: positive integer n, list of numbers S indexed from 1 to n

Output: the maximum element in the list S

```
number findMax(int n, const keytype S[ ]) {
    index i;
    number max = S[1];
    for(i =2; i<=n; i++)
            if(S[i] > max)
                    max = S[i];
    return max;
}
```

2) Write an algorithm that finds the m smallest numbers in a list of n numbers.

Input: positive integer n, list of numbers S indexed from 1 to n, empty list of numbers Small indexed from 1 to m.

Output: the list Small with the smallest m numbers in it.

```
void find_m_Smallest(int n, const keytype S[ ], keytype Small[]) {
    index i, j, k;
    Small[1] = S[1];
    for(i =2; i<=n; i++) {
            if (j <=m)
                    find index j where S[i] needs to be inserted in Small, such that\
                                    Small stays sorted in non-decreasing order;
            for (k = m-1; k>=j, k--)
                    Small[k+1) = Small[k];          //shift right
            Small[j] = Small[i];                    //insert S[i]
    }
}
```

3) Write an algorithm that prints out all the subsets of three elements of a set of n elements. The elements of this set are stored in a list that is the input to the algorithm.

Input: positive integer n, list of numbers S indexed from 1 to n

Output: none (the results are simply printed)

```
void nC3(int n, const keytype S[ ]) {
   index i, j, k;
   for (i =1; i<=n; i++)
         for (j=i+1; j<=n;, j++)
               for (k=j+1; k<=n; k++)
                     cout <<S[i] <<S[j] <<S[k] <<endl;
```

4) Write an Insertion Sort algorithm (Insertion Sort is discussed in Section 7.2) that uses Binary Search to find the position where the next insertion should take place.

```
void insertionsort (int n, keytype S[]) {
      index i , j ;
      index &location
      keytype x;
      for (i = 2; i <= n; i++) {
            x = S[ i ];
            binsearch(i, S, x, location);
            for (j=i-1; j>=location; j--)
                  S[ j + 1] = S[ j ];        //shift right
            S[ j ] = x;                      //insert S[i] in final position
      }
}
```

The function *binsearch* is the one in Algorithm 1.5, but modified so that, if the search is unsuccessful, location is the index of the first element greater than x:

```
void binsearch (int n , const keytype S [ ] , keytype x , index& location) {
      index low , high , mid ;
      bool found = false
      low = 1; hi gh = n; location = 0;
      while ( low <= hi gh && !found) {
            mid = ⌊(low + high)⌋/2 ;
            if (x == S [mid ])
                  location = mid ;
            elseif  (x < S [mid ])
```

```
                hi gh = mid □ 1;
            else
                low = mid + 1 ;
        }
    }
```

5) Write an algorithm that finds the greatest common divisor of two integers.

Input: two integers *a* and *b*

Output: the greatest common divisor of *a* and *b*

```
int gcd(int a, int b) {
    if(b == 0)
        return a;
    return gcd(b, a%b) ;
}
```

6) Write an algorithm that finds both the smallest and largest numbers in a list of n numbers. Try to find a method that does at most 1.5n comparisons of array items.

If we simply write one loop to find the minimum and one to find the maximum, there will be 2(n-1) comparisons. We can reduce this number by comparing pairs of numbers; this finds max(a,b) and min(a,b) in one comparison. Then we need only 2 more comparisons to update the overall min and max, for a total of 3 comparisons for each pair of numbers. The total nr. of comparisons ≈ 3n/2 = 1.5n.

7) Write an algorithm that determines whether or not an almost complete binary tree is a heap.

Input: a pointer *tree* to an almost complete binary tree

Output: true if the input tree is a heap, false otherwise

```
bool isHeap(tree_pointer tree) {
```

```
        if(tree->left == NULL && tree-> right == NULL)
                return true;
        if(tree->key < tree->left->key)
                return false;
        if(tree->right != NULL && tree->key < tree->right->key)
                return false;
        if(tree->right == null)
                return isHeap(tree->left);
        return isHeap(tree->left) && isHeap(tree->right);
}
```

**Section 1.2**

8) Under what circumstances, when a searching operation is needed, would Sequential Search (Algorithm 1.1) not be appropriate?

If the data is already sorted and

- there are a very large number of elements, or
- the access to the elements is slow (e.g. they are on magnetic tape)

9) Give a practical example in which you would not use Exchange Sort (Algorithm 1.3) to do a sorting task.

Is we had to sort an array of 10 million elements, the number of comparisons in Exchange Sort would be $\approx 10^7 \times 10^7 = 10^{14}$. Even performing one comparison every nanosecond, the program would still run for $10^5$ seconds, which is more than one day. A more efficient algorithm like Mergesort would run in less than half a second.

If we tried Exchange Sort algorithm for 100M elements, it would take almost 4 months, whereas Mergesort would finish in about 5 seconds!

**Section 1.3**

10) Define basic operations for your algorithms in Exercises 1-7, and study the performance of these algorithms. If a given algorithm has an every-case time complexity, determine it. Otherwise, determine the worst-case time complexity.

Exercise 1: Basic operation is comparison. Every-time count $T(n)=n-1$ .
Exercise 2: Basic operation is the assignment needed in the shift phase. Worst-case count $W(n,m) = n \times m$, if array is initially sorted in reverse order.
Exercise 3: Basic operation is accessing an array element. Every-time count $T(n)=n(n-1)(n-2)$.
Exercise 4: Basic operation is assigning an array element. Worst-case count $W(n) = 3 + 4 + \ldots + n = n(n+1)/2 - 3$, if array is initially sorted in reverse order.
Exercise 5: Basic operation is a%b, which is performed once per recursive call. Lamé proved in 1844 that, when the initial numbers are Fibonacci numbers $a = F_{n+2}$, $b = F_{n+1}$, then the algorithm will perform exactly n steps, so the worst case does not have a closed-form for every number n.
Exercise 6: Basic operation is comparison. Every-time count $T(n) = 3\lceil n/2 \rceil$.
Exercise 7: Basic operation is comparison. Worst-time count is 2 for every internal node of the heap, which is $\approx 2n/2 = n$.

11) Determine the worst-case, average-case, and best-case time complexities for the basic Insertion Sort and for the version given in Exercise 4, which uses Binary Search.

In this solution, we only count <u>comparisons</u>:

Basic Insertion Sort: $B(n) = n$ (array already sorted), $A(n) = n^2/4$, $W(n) = n^2/2$ (array reverse sorted). Proofs can be found in Section 7.2.

Insertion Sort with Binary Search: $B(n) = 2\lg n$ (each new key is always inserted in the middle of the array, so only 2 comparisons are needed to find its location … if repeated keys are allowed, than an array of identical elements only requires 1 comparison per key!), $A(n) = n\lg n$, $W(n) = n\lg n$.

12) Write a (n) algorithm that sorts n distinct integers, ranging in size between 1 and kn inclusive, where k is a constant positive integer. (Hint: Use a kn-element array.)

Input: positive integer n, positive integer k, array of numbers S indexed from 1 to n

Output: sorted array S

```
void sort(int n, int S[]) {
        int temp[] = int[k*n]; //initially all zeros
        index i, j;
        for(i=1; i<=n; i++) {
                temp[S[i]] = x;
        }
        for(i=1, j=0; i<=k*n; i++) {
                if(temp[i] != 0) {
                        a[j] = temp[i];
                        j++;
                }
        }
}
```

13) Algorithm A performs $10n^2$ basic operations, and algorithm B performs $300\ln n$ basic operations. For what value of n does algorithm B start to show its better performance?

For n = 7, we have 490 operations in A, and 584 in B, so A is better, but for for n = 8, we have 640 operations in A and 624 in B, so B is better (and stays that way for all n >= 8).

14) There are two algorithms called Alg1 and Alg2 for a problem of size n. Alg1 runs in $n^2$ microseconds and Alg2 runs in $100n \log n$ microseconds. Alg1 can be implemented using 4 hours of programmer time and needs 2 minutes of CPU time. On the other hand, Alg2 requires 15 hours of programmer time and 6 minutes of CPU time. If programmers are paid 20 dollars per hour and CPU time costs 50 dollars per minute, how many times must a problem instance of size 500 be solved using Alg2 in order to justify its development cost?

For instance size n = 500, the problem is solved in $500^5 \times 10^{-6} = 25 \times 10^4 \times 10^{-6} = 0.25$ s with Alg1, and in $100 \times 500 \times \log 500 \times 10^{-6} = 0.1349$ s with Alg2.

The break-even point is found from this linear equation:

4*20 + 2*50 + y*0.25*50/60 = 15*20 + 6*50 + y*0.1349*50/60.

The solution is y =4379 times.

**Section 1.4**

15) Show directly that $f(n) = n^2 + 3n^3 \in (n^3)$. That is, use the definitions of O and Ω to show that f(n) is in both $O(n^3)$ and $\Omega(n^3)$.

We show that $n^2 + 3n^3 \in O(n^3)$ because for $n \geq 0$,

$$n^2 + 3n^3 \leq 4n^3 ,$$

we can take c = 4, N = 0 to obtain our result.

We show that $n^2 + 3n^3 \in \Omega(n^3)$ because for $n \geq 0$,

$$n^2 + 3n^3 \geq 3n^3 ,$$

we can take c = 3, N = 0 to obtain our result.

Thus, since $n^2 + 3n^3 \in O(n^3)$ and $n^2 + 3n^3 \in \Omega(n^3)$, $n^2 + 3n^3 \in \Theta(n^3)$.

16) Using the definitions of O and Ω, show that

$$6n^2 + 20n \in O(n^3), \text{ but } 6n^2 + 20n \notin \Omega(n^3).$$

Starting at N = 9, $6n^2 + 20n < n^3$, so we can take c = 1, N = 9 in the definition of O.

On the other hand, no matter how large c were chosen, the limit $c(6n^2 + 20n)/n^3$ is zero, so $c(6n^2 + 20n)$ cannot stay $> n^3$which contradicts the definition of Ω.

17) Using the Properties of Order in Section 1.4.2, show that

$$5n^5 + 4n^4 + 6n^3 + 2n^2 + n + 7 \in \Theta(n^5).$$

Apply Property 7, with $g(n) = n^4 + 6n^3/4 + 2n^2/4 + n/4 + 7/4$, $h(n) = n^5$ , c = 4, and d = 5.

A direct proof is also possible: take $f(n)= 5n^5 + 4n^4 + 6n^3 + 2n^2 + n + 7$, and we have

$5n^5 + 4n^4 + 6n^3 + 2n^2 + n + 7 <= 5n^5 + 4n^4 + 6n^3 + 2n^2 + n + 7 <= 5n^5 + 4\,n^5 + 6\,n^5 + 2\,n^5 + n^5 + 7\,n^5$

$5n^5 <= 5n^5 + 4n^4 + 6n^3 + 2n^2 + n + 7 <= 25n^5$

Therefore the order will be $\Theta(n^5)$, with c1=5 and c2=25, where n>=1.

18) Let $p(n) = a_k n^k + a_{k-1}n^{k-1} + \ldots + a_1 n + a_0$, where $a_k > 0$. Using the Properties of Order in Section 1.4.2, show that $p(n) \in \Theta(n^5)$.

As in Example 1.23, we repeatedly apply Properties 6 and 7 to show:

$a_k n^k \in \Theta(n^k)$

$a_k n^k + a_{k-1}n^{k-1} \in \Theta(n^k)$

…

$a_k n^k + a_{k-1}n^{k-1} + \ldots + a_1 n + a_0 \in \Theta(n^k)$

As a technicality, if some of the lower-order coefficients $a_{k-1}, \ldots a_0$ are ≤ 0, but in this case we simply have a negative function $a_i n^i$, which is obviously in $O(n^k)$.

19) The function $f(x) = 3n^2+10n \log n+1000n+4\log n+9999$ belongs in which of the following complexity categories:

(a) $\theta(\lg n)$ (b) $\theta(n^2 \log n)$ (c) $\theta(n)$ (d) $\theta(n \lg n)$

(e) $\theta(n^2)$ (f) None of these

$f(n) \in \Theta(n^2)$, by repeatedly applying Properties 6 and 7 (or "throwing out" all the lower-order terms).

20) The function $f(x) = (\log n)^2 + 2n + 4n + \log n + 50$ belongs in which of the following complexity categories:

(a) $\theta(\lg n)$ (b) $\theta((\log n)^2)$ (c) $\theta(n)$ (d) $\theta(n \lg n)$

(e) $\theta(n(\lg n)^2)$ (f) None of these

$f(n) \in \Theta(n)$, by repeatedly applying Properties 6 and 7 (or "throwing out" all the lower-order terms).

Note: $(\log n)^2$ is not listed in Property 6, but we can apply Theorem 1.4 (L'Hôpital's Rule) twice to show that $\lim[(\log n)^2/n] = \lim 2[(\log n)/n] = \lim[2/n] = 0$, when $n \to \infty$. Now Theorem 1.3 ensures that $(\log n)^2 \in o(n)$.

21) The function f(x) = n + $n^2$ + $2^n$ + $n^4$ belongs in which of the following complexity categories:

(a) $\theta(n)$ (b) $\theta(n^2)$ (c) $\theta(n^3)$ (d) $\theta(n \lg n)$
(e) $\theta(n^4)$ (f) None of these

Solution: None of these; it is actually $\Theta(2^n)$, according to Property 6.

22) Group the following functions by complexity category.

$n \ln n$ $(\lg n)^2$ $5n^2 + 7n$ $n^{5/2}$
$n!$ $2^{n!}$ $4^n$ $n^n$ $n^n + \ln n$
$5^{\lg n}$ $\lg(n!)$ $(\lg n)!$ $\sqrt{n}$ $e^n$ $8n + 12$ $10^n + n^{20}$

Solution: From "small" to "large", the groups are:

G1 = { $(\lg n)^2$ }
G2 = { $5^{\lg n}$ }
G3 = { $\sqrt{n}$ }
G4 = { 8n +12 }
G5 = { nln n, lg(n!) }
G6 = { $5n^2 + 7n$ }
G7 = { $n^{5/2}$ }
G8 = { (lg n)! }
G9 = { $e^n$ }
G10 = { $4^n$ }
G11 = { $10^n + n^{20}$ }
G12 = { $n^n$ , $n^n$ + ln(n)}
G13 = { $2^{n!}$ }

As examples, consider some of the more difficult functions:

- f(n) = $5^{\lg n}$ → With algebraic manipulations we have: f(n) = $5^{\log5(n)/\log5(2)}$ = $(5^{\log5(n)})^{1/\log5(2)} \approx n^{0.43} \in o(n^{0.5} = \sqrt{n})$, which explains G2.
- f(n) = (lg n)! → We first change the logarithm into a natural logarithm: f(n) = (ln n)/ln2)!, and for simplicity we ignore the constant factor ln2 ≈ 0.69, so we consider f(n) ≈ (ln n)! Then we apply Stirling's formula ($n! \approx \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$, where e is the base of the natural logarithm), substituting ln n for n. Leaving out the constant factors,

we have f(n) ≈ sqrt(ln n)·(ln n)$^{\ln n}$/n (*). Consider only (ln n)$^{\ln n}$; if we write it in the form $n^{g(n)}$, where g(n) is an unknown function, we can take logs on both sides to find g(n) = ln(ln n). Formula (*) above thus becomes f(n) ≈ sqrt(ln n)· n $^{\ln(\ln n)-1}$ . This is obviously larger than any polynomial, so let's compare it with the exponential $e^n$: taking logs on both sides we find that ln(f(n)) ∈ O($e^n$), which explains why G8 comes before G9.

- f(n) = lg(n!) → We first change the base of the logarithm: f(n) = ln(n!) / ln2. Then we apply Stirling's formula to ln(n!): f(n) ≈ [ ln $\sqrt{2\pi}$ + 0.5ln(n) + n(ln(n) – 1) ]/ln2 ∈Θ(nln(n) = Θ(nlg n), which explains G5.

23) Establish Properties 1, 2, 6, and 7 of the Properties of Order in Section 1.4.2.

Property 1: g(n)∈O(f(n)) if and only if f(n)∈ Ω(g(n)).

Proof: "If and only if" means double implication.

We prove first that g(n)∈O(f(n)) implies f(n)∈ Ω(g(n)). According to the definition of O, there exist c, N > 0 such that, for all n ≥ N, g(n) ≤ cf(n). This means f(n) ≥ $c^{-1}$g(n), which means f(n)∈ Ω(g(n)).

The opposite implication is proved in the same manner.

Property 2: g(n) ∈ Θ(f(n)) if and only if f(n)∈ Θ(g(n)).

Proof: We prove first that g(n)∈Θ(f(n)) implies f(n)∈ Θ(g(n)). According to the definition of Θ, there exist c, d, N > 0 such that, for all n ≥ N, cf(n) ≤ g(n) ≤ df(n) ⇨

⇨ $d^{-1}$g(n) ≤ f(n) ≤ $c^{-1}$g(n) ⇨ f(n)∈ Θ(g(n)).

The opposite implication is proved in the same manner.

Property 6: Here we have to compare each category with the next in the sequence.

As an example, we prove **$b^n$ ∈ o(n!)**:

Proof: According to the definition of "little-oh", we have to show that, for every positive c, there exists N such than $b^n$ ≤ cn!, for all n ≥ N. Let $n_0$ be the first integer with the property $n_0$ > 2b. We rewrite the desired inequality thus:

$b^{n0}\cdot b^{n-n0} \le c(1\cdot 2\cdot \ldots \cdot n_0 \cdot \ldots \cdot n) \Leftrightarrow \frac{b}{c}\cdot\frac{b^{n0}}{n_0!} \le \frac{n_0(n_0+1)\ldots n}{b\ \cdot\ b\ \ \ldots\ \ b}$, where the left-hand side

is a constant C, and each ratio on the right-hand-side is > 2. Therefore we simply make N large enough so that C < $2^{N-n0+1}$, or N =

Property 7: If c>=0, d>0, g(n) ∈ O(f(n)), and h(n) ∈ Θ(f(n)), then c*g(n)+d*h(n) ∈ Θ(f(n))

Proof: According to the definitions of O and Θ, there are positive constants c1, N1, c2, d2, N2 such that

$g(n) \le c1 \cdot f(n)$ for all $n \ge N1$

$c2 \cdot f(n) \le h(n) \le d2 \cdot f(n)$ for all $n \ge N2$

We multiply the first inequality by c and the second one by d, and denote by N the maximum of N1 and N2:

$cg(n) \le cc1 \cdot f(n)$ (*)

$dc2 \cdot f(n) \le dh(n) \le dd2 \cdot f(n)$ (**) for all $n \ge N$

Adding (*) and the right inequality in (**), we obtain

$cg(n) + dh(n) \le (cc1 + dd2) \cdot f(n)$ for all $n \ge N$, which is the right-hand-side inequality in the definition of Θ.

From the left-hand-side inequality of (**), we have $dc2 \cdot f(n) \le dh(n)$, and we add cg(n) to obtain

$dc2 \cdot f(n) \le cg(n) + dh(n)$ for all $n \ge N$, which is the left-hand-side inequality in the definition of Θ.

24) Discuss the reflexive, symmetric, and transitive properties for asymptotic comparisons (O, Ω, Θ, o).

Reflexivity: O, Ω, and Θ are reflexive, which is shown by taking N = 1 and c = 1 in their respective definitions. o is not reflexive, since, by contradiction, it would lead to the conclusion that every positive real constant c has the property $1 \le c$.

Symmetry: Only Ω is symmetrical, as implied by property 2. For all the other it's easy to give simple counterexamples, e.g. n and $n^2$.

Transitivity: They are all transitive. For example, we prove that o is:

Assume f is in o(g) and g is in o(h). Let c>0, and we need to find an N such that $f(n) \le ch(n)$ for all $n \ge N$.

Because g is in o(h), let N1 be that integer with the property that $g(n) \le ch(n)$ for all $n \ge N1$. Because f is in o(g), let N2 be that integer with the property that $f(n) \le g(n)$ for all $n \ge N2$. For all $n \ge \max(N1, N2)$, we have $f(n) \le g(n) \le ch(n)$, so $f(n) \le ch(n)$. Q.e.d.

25) Suppose you have a computer that requires 1 minute to solve problem instances of size n =1,000. Suppose you buy a new computer that runs 1,000 times faster than the old one. What instance sizes can be run in 1 minute, assuming the following time complexities T(n) for our algorithm?

a) $T(n) = n$ Answer: $10^6$

b) $T(n) = n^3$ Answer: 104

c) $T(n) = 10^n$ Answer: 1003

26) Derive the proof of Theorem 1.3.

$$\lim_{n\to\infty}\frac{g(n)}{f(n)}=\begin{cases} c & \text{implies } g(n)\in\Theta(f(n)) \text{ if } c>0 \\ 0 & \text{implies } g(n)\in o(f(n)) \\ \infty & \text{implies } f(n)\in o(g(n)) \end{cases}$$

If the limit is c > 0, then for any ε > 0 there is N > 0 such that g(n)/f(n) is in between c-ε and c+ε for all n ≥ N ⇨ (c-ε)f(n) ≤ g(n) ≤ (c+ε)f(n) for all n ≥ N, which is the definition of g(n) ∈ Θ(f(n)).

If the limit is 0, then for any ε > 0 there is N > 0 such that g(n)/f(n) < ε for all n ≥ N ⇨ g(n) ≤ εf(n) for all n ≥ N, which is the definition of g(n) ∈ o(f(n)).

If the limit is ∞, we can apply the definition of the limit as above, or simply notice that the reciprocal limit f(n)/g(n) is zero, and use the previous case.

27) Show the correctness of the following statements.

(a) $\lg n \in O(n)$

(b) $n \in O(n \lg n)$

(c) $n \lg n \in O(n^2)$

(d) $2^n \in \Omega(5^{\ln n})$

(e) $\lg^3 n \in o(n^{0.5})$

For all these results, we use Theorem 1.3 and Theorem 1.4 ( L'Hôpital's rule):

a) lim[(lg n)/n] = lim[(lg n)'/n'] = lim[1/n]/ln2 = 0, so lg n is in o(n), which, according to Theorem 1.2, is included in O(n), so lg n is in O(n).

b) lim[n/(nlg n)] = lim[1/(nlg'n + lg n)] = 0, and continue as in (a).

c) lim[nlg n/$n^2$] = lim[(lg n)/n] same as (a).

d) With algebraic manipulations we have $5^{\lg n} = 5^{\log 5(n)/\log 5(2)} = (5^{\log 5(n)})^{1/\log 5(2)} \approx n^{0.43}$ , ao $\lim[5^{\lg n}/2^n] = \lim[n^{0.43}/2^n] = 0.43 \cdot \ln 2 \cdot \lim[1/n^{0.57}2^n] = 0$, so as above we have that $5^{\lg n} \in O(2^n)$, which, by Property 1, implies $2^n \in \Omega(5^{\lg n})$.

e) $\lim[\lg^3 n/n^{0.5}] = \lim[(3\lg^2 n/(n\ln 2)) / (0.5/n^{0.5})] = C_1\lim[\lg^2 n/n^{0.5}] = \ldots = C_2\lim[\lg n/n^{0.5}] = \ldots = C_3\lim[1/n^{0.5}] = 0$, and from Theorem 1.3 this implies that $\lg^3 n \in o(n^{0.5})$.

**Additional Exercises**

28) Presently we can solve problem instances of size 30 in 1 minute using algorithm A, which is a $\Theta(2^n)$ algorithm. On the other hand, we will soon have to solve problem instances twice this large in 1 minute. Do you think it would help to buy a faster (and more expensive) computer?

$2^{30}$ operations in 60 s means that the machine performs 17,895,697.0(6) op. per second, or that one operation takes 55.87 nanoseconds.
A problem instance twice as large requires $2^{2n} = 2^{60}$ op., which, on the same machine, would take 64.4 billion years (!). The new computer would have to be at least 10 billion times faster, so a realistic approach is to find a more efficient algorithm.

29) Consider the following algorithm:

```
for ( i = 1 ; i <= 1 . 5 n ; i++)
        cout << i ;
for ( i = n ; i >= 1 ; i - - )
        cout << i ;
```

(a) What is the output when n = 2, n = 4, and n = 6?

(b) What is the time complexity T(n)? You may assume that the

input n is divisible by 2.

a) When n=2, the output is:
1 2 3 2 1
When n=4, the output is:
1 2 3 4 5 6 4 3 2 1
When n=6, the output is:
1 2 3 4 5 6 7 8 9 6 5 4 3 2 1

b) $T(n) = 1.5n + n = O(n)$

30) Consider the following algorithm:

```
j = 1 ;
while ( j <= n/2) {
        i = 1 ;
        while ( i <= j ) {
                cout << j << i ;
                i++;
        }
        j++;
}
```

(a) What is the output when n = 6, n = 8, and n = 10?

(b) What is the time complexity T(n)? You may assume that the input n is divisible by 2.

a) When n=6, the output is:
1 1 2 1 2 2 3 1 3 2 3 3
When n=8, the output is:
1 1 2 1 2 2 3 1 3 2 3 3 4 1 4 2 4 3 4 4
When n=10, the output is:
1 1 2 1 2 2 3 1 3 2 3 3 4 1 4 2 4 3 4 4 5 1 5 2 5 3 5 4 5 5

b) $T(n) = (n^2/8) - (n/4) = O(n^2)$

31) Consider the following algorithm:

```
for ( i = 2 ; i <= n ; i++) {
        for ( j = 0 ; j <= n) {
                cout << i << j ;
                j = j + floor(n/4) ;
        }
}
```

(a) What is the output when n = 4, n = 16, n = 32?

Solution: n = 4: 2 0 2 1 2 2 2 3 2 4 3 0 3 1 3 2 3 3 3 4 4 0 4 1 4 2 4 3 4 4

n = 16: 2 0 2 4 2 8 2 12 2 16 3 0 3 4 3 8 3 12 3 16 4 0 4 4 4 8 4 12 4 16 5 0 5 4 5 8 5 12 5 16 6 0 6 4 6 8 6 12 6 16 7 0 7 4 7 8 7 12 7 16 8 0 8 4 8 8 8 12 8 16 9 0 9 4 9 8 9 12 9 16 10 0 10 4 10 8 10 12 10 16 11 0 11 4 11 8 11 12 11 16 12 0 12 4 12 8 12 12 12 16 13 0 13 4 13 8 13 12 13 16 14 0 14 4 14 8 14 12 14 16 15 0 15 4 15 8 15 12 15 16 16 0 16 4 16 8 16 12 16 16

n = 32: too long to show here.

(b) What is the time complexity T(n). You may assume that n is divisible 4.

Solution: $T(n) = (n-1)(1+n/4) \in O(n^2)$

32) What is the time complexity T(n) of the nested loops below? For simplicity, you may assume that n is a power of 2. That is, n = 2k for some positive integer k.

```
for ( i = 1 ; i <= n ; i++) {
        j = n ;
        while ( j >= 1) {
                < body o f the while loop> //Needs ( 1 ) .
                j = floor(j/2) ;
        }
}
```

Solution: $T(n) = n*\log n = O(n*\log n)$

33) Give an algorithm for the following problem and determine its time complexity. 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 first list should have the larger half of the elements, and the second list should have the smaller half, so we have to find the median. The easiest way to do it is to sort the list, which is $\Theta(n^2)$ with Exchange Sort – Algorithm 1.3.

The median can, however, be found very efficiently with the "Median of Medians" algorithm, whose complexity is only $\Theta(n)$, after which the list can be partitioned using

the median as a pivot in another $\Theta(n)$ operations, which makes the entire algorithm $\Theta(n)$.

34) What is the time complexity T(n) of the nested loops below? For simplicity, you may assume that n is a power of 2. That is, n = 2k for some positive integer k.

```
i = n ;
while ( i >= 1) {
        j = i ;
        while ( j <= n) {
                < body o f the while loop> //Needs ( 1 ) .
                j = 2 * j;
        }
        i = floor(i /2) ;
}
```

Solution: $T(n) = (\log^2 n + \log n)/2 = O(\log^2 n)$

35) Consider the following algorithm, where the array A is indexed 1 through n:

```
int add_them ( int n , int A[ ] ) {
        index i , j , k ;
        j = 0 ;
        for ( i = 1 ; i <= n ; i++)
                j = j + A[i] ;
        k = 1 ;
        for ( i = 1 ; i <= n ; i++)
                k = k + k ;
        return j + k ;
}
```

(a) If n = 5 and the array A contains 2, 5, 3, 7, and 8, what is returned?

Solution: 57

(b) What is the time complexity T(n) of the algorithm?

Solution: T(n) = 2n

(c) Try to improve the efficiency of the algorithm.

Solution: Since the second loop simply turns k into $2^n$, we can calculate $2^n$ directly, e.g. with the library function "power": k = pow(2, n). As explained in Ch.3, there are algorithms more efficient than linear to calculate powers, based on Dynamic Programming.

36) Consider the following algorithm:

```
int any_equal ( int n , int A[ ] [ ] ) {
    index i , j , k ,m;
    for ( i = 1 ; i <= n ; i++)
            for ( j = 1 ; j <= n ; j++)
                    for ( k = 1 ; k <= n ; k++)
                            for (m = 1 ; m <= n ; m++)
                                    if (A[ i ] [ j ]==A[ k ] [m] && ! ( i==k && j==m))
                                            return 1 ;
    return 0 ;
}
```

(a) What is the best case time complexity of the algorithm (assuming n > 1)?

Solution: The best-case occurs when A[1][1] is equal to A[1][2]; the algorithm returns 1 after only the second iteration of the innermost loop (i = j = k = 1, m=2). Time complexity is B(n) = 2 (counting comparisons of array elements).

(b) What is the worst case time complexity of the algorithm?

Solution: The worst -case occurs when all the elements of A are unique, in which case there are $W(n) = n^4$ comparisons.

(c) Try to improve the efficiency of the algorithm.

Solution: Each pair of different elements is compared twice, so we can improve the efficiency by comparing only "ahead". Also, we can avid comparing elements with themselves, because in that case ! ( i==k && j==m) is false, so the if statement is redundant. The improved pseudocode is:

```
int any_equal ( int n , int A[ ] [ ] ) {
    index i , j , k ,m, index_first, index_second;
    for ( i = 1 ; i <= n ; i++)
            for ( j = 1 ; j <= n ; j++) {
                    index_first = n*i + j;
                    for ( k = 1 ; k <= n ; k++)
                            for (m = 1 ; m <= n ; m++) {
                                    index_second = n*k + m;
                                    if (index_second > index_first)
                                            if (A[ i ] [ j ]==A[ k ] [m])
                                            return 1 ;
                            {
            }
    return 0 ;
}
```

Now the worst case is $W'(n) = (n^2-1) + (n^2-2) + (n^2-3) + \ldots + (n^2-(n^2-1)) = n^2(n^2-1) - \sum_{i=1}^{n^2-1} i$ and the sum can be calculated as in Example A.1, yielding $n^2(n^2-1) - n^2(n^2-1)/2 = n^2(n^2-1)/2$.

(d) What property holds for the array A if the algorithm returns 0?

Solution: The array has no duplicates.

(e) What property holds for the array A if the algorithm returns 1?

Solution: The array has at least one pair of duplicates.

37) Give a $\Theta(\lg n)$ algorithm that computes the remainder when $x^n$ is divided by p. For simplicity, you may assume that n is a power of 2. That is, $n = 2^k$ for some positive integer k.

Solution: If we have an integer, m , we divide it by p to obtain the remainder: $m = q \cdot p + r$. The square $n = m^2 = m \cdot m$ is $(q \cdot p + r)(q \cdot p + r) = (q^2p + 2qr)p + r^2$, which shows that only $r^2$ must be considered when calculating the remainder of $m^2$ modulo p:

$m^2 \% p = r^2 \% p$.

Back to our problem, let $m = x^{n/2}$ . We split the problem the problem into two identical subproblems: $x^n = x^{n/2} \cdot x^{n/2}$. All we need to do now is to find $r = x^{n/2} \% p$, so we can write $T(n) = T(n/2) + C$, where C is the constant cost of squaring r and taking $r^2 \% p$. Since $n = 2^k$, n/2 is always an integer, until n = 1, so we have recursively $T(n) = T(n/2) + C = T(n/4) + 2C = T(n/8) + 3C = \ldots = T(n/2^j) + jC$. It's clear that when $n/2^j = 1$, j is lg n, so the solutions is $T(n) = (\lg n)C \in \Theta(\lg n)$.

38) Explain in English what functions are in the following sets.

(a) $n^{O(1)}$

Solution: All functions f(n) bounded above by a polynomial of the form $Cn^D$, with C, D > 0: $f(n) \leq Cn^D$.

(b) $O(n^{O(1)})$

Solution: All functions g(n) bounded above by Ef(n), whereE > 0, and f(n) is as above. We have $g(n) \leq Ef(n) \leq ECn^D = Fn^D$, with F, D > 0, therefore $O(n^{O(1)}) = n^{O(1)}$, meaning that the set of functions at (b) is the same as that at (a).

(c) $O(O(n^{O(1)}))$

Solution: As above, we have that $O(n^{O(1)}) = O(n^{O(1)}) = n^{O(1)}$.

39) Show that the function $f(n) = n^2 \sin n$ is in neither O(n) nor Θ(n).

Solution:
For $n \approx 2k\pi + \pi/2$ , $f(n) \approx n^2 \sin(\pi/2) = n^2$, so $f$ is not in $O(n)$.
For $n \approx 2k\pi$ , $f(n) \approx n^2 \sin(0) = 0$, so $f$ is not in $\Omega(n)$.

Technical note: the approximations above can be given a clear mathematical meaning, based on the irrationality of π, which implies that multiples of π will approach integers an infinite number of times; integers from that series can be chosen for n. A detailed proof, however, is outside the scope of this course.

For a more elementary (if a bit contrived) example, consider the function g(n) defined to be 0 for n even and $n^2$ for n odd.

40) Justify the correctness of the following statements assuming that f(n) and g(n) are asymptotically positive functions.

(a) $f(n) + g(n) \in O(\max(f(n)) ; g(n))$

Solution: $f(n) + g(n) \leq 2\max(f(n) + g(n))$

(b) $f^2(n) \in \Omega(f(n))$

Solution: Assuming that, for $n > N$, $f(n) > 1$, we have $f^2(n) \geq f(n)$ for $n > N$.

(c) $f(n) + o(f(n)) \in \Theta(f(n))$, where $o(f(n))$ means any function $g(n) \in o(f(n))$

Solution: $f(n) + o(f(n)) \geq f(n) + cf(n) = (1+c)f(n) \Rightarrow f(n) + o(f(n)) \in O(f(n))$
On the other hand, $f(n) + o(f(n)) \leq f(n) \Rightarrow f(n) + o(f(n)) \in \Omega(f(n))$

41) Give an algorithm for the following problem. 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 minimized. Determine the time complexity of your algorithm. You may assume that n is a multiple of 2.

Solution: Let the set of all list nodes be denoted by A. The algorithm needs a temporary set of n/2 elements, TempSet.

```
int min_diff = LARGE_NUMBER;
int diff;
for (each subset S of size n/2 of A) {
        diff = abs(sum(S) – sum(A-S));
        if (diff < min_diff) {
                min_diff = diff;
                TempSet = S;
        }
}
print min_diff, TempSet;
```

Each iteration of the for loop has a constant worst-case cost (one comparison, an integer assignment, a set assignment), and the loop executes n-choose-n/2 times, so the algorithm is in $\Theta\left(\frac{n!}{((n/2)!)^2}\right)$.

42) Algorithm 1.7 (nth Fibonacci Term, Iterative) is clearly linear in n, but is it a linear-time algorithm? In Section 1.3.1 we defined the input size as the size of the input. In the case of the nth Fibonacci term, n is the input, and the number of bits it takes to encode n could be used as the input size. Using this measure, the size of 64 is lg64= 6, and the size of 1,024 is lg1,024= 10. Show that Algorithm 1.7 is exponential-time in terms of its input size. Show further that any algorithm for computing the nth Fibonacci term must be an exponential-time algorithm because the size of the output is exponential in the input size. (See Section 9.2 for a related discussion of the input size.)

Solution:

Proof of exponential output size: We can find a closed-form formula for the nth Fibonacci number by solving the recursion F(n) = F(n-1) + F(n-2) directly, with the tools in Appendix B. The associated homogeneous equation $x^2 = x + 1$ has two roots: r1 = $\frac{1+\sqrt{5}}{2}$, r2 = $\frac{1-\sqrt{5}}{2}$, so the solution has the form $F(n) = C_1 r_1^n + C_2 r_2^n$. The constants are determined from the values of F(0) and F(1), and they turn out to be both equal to $1/\sqrt{5}$, so we have

$$F(n) = \left(\left(\frac{1+\sqrt{5}}{2}\right)^n + \left(\frac{1-\sqrt{5}}{2}\right)^n\right) / \sqrt{5}$$

Given that the absolute value of r2 is < 1, the second power in the expression tends to 0 as $n \to \infty$, so we have

$$F(n) \approx \left(\frac{1+\sqrt{5}}{2}\right)^n / \sqrt{5} \approx 1.618^n / 2.236$$

The size of the output is $\lg(F(n)) \approx n\lg(1.618) \approx 0.69n = 0.69 \cdot 2^{\lg n}$, therefore exponential in terms of the size of the input.

Proof of exponential time: the lgn bits used to represent the input n are being processed to generate about $0.69 \cdot 2^{\lg n}$ bits of the output F(n). We define our fundamental operation as being the generation of one bit, since any binary computer has this type of operation at the hardware level.

43) Determine the time complexity of Algorithm 1.6 (nth Fibonacci Term, Recursive) in terms of its input size (see Exercise 42).

Solution: According to Exercise 42, any algorithm for computing the nth Fibonacci term must be an exponential-time algorithm, so in particular Algorithm 1.6 is exponential, too.

44) Can you verify the correctness of your algorithms for Exercises 1 to 7?

Solution: We prove the correctness of the algorithm from Exercise 1, which finds the maximum in an array of n numbers.

```
Input: positive integer n, list of numbers S indexed from 1 to n
Output: the maximum element in the list S
    number findMax(int n, const keytype S[ ]) {
        index i;
        number max = S[1];
        for(i =2; i<=n; i++)
                if(S[i] > max)
                        max = S[i];
        return max;
    }
```

Proof: Induction on n: If n = 1, the algorithm returns max = S[1] which is the only number in the array. This is correct.

Induction Hypothesis: The algorithm is correct for n.

Induction step: We show that the algorithm is correct for n+1: When j = n, max contains the true maximum of the sub-array S[1…n]. When j = n+1, there are two cases:

- if S[n+1] > max, it means that the last element is largest than any among the first n, so the algorithm correctly changes max to the last element.
- If S[n+1] ≤ max, it means that among the first n elements there is (at least) one that is ≥ S[n+1], so the algorithm correctly leaves max unchanged.