CHAPTER 3

Section 3.1

1. *max* := 1, i := 2, *max* := 8, i := 3, *max* := 12, i := 4, i := 5, i := 6, i := 7, *max* := 14, i := 8, i := 9, i := 10, i := 11

3.
```
procedure AddUp(a_1, ..., a_n: integers)
  sum := a_1
  for i := 2 to n
    sum := sum + a_i
  return sum
```

5.
```
procedure duplicates(a_1, a_2, ..., a_n: integers in
    nondecreasing order)
  k := 0 {this counts the duplicates}
  j := 2
  while j ≤ n
    if a_j = a_{j-1} then
      k := k + 1
      c_k := a_j
      while j ≤ n and a_j = c_k
        j := j + 1
    j := j + 1
  {c_1, c_2, ..., c_k is the desired list}
```

7.
```
procedure last even location(a_1, a_2, ..., a_n: integers)
  k := 0
  for i := 1 to n
    if a_i is even then k := i
  return k {k = 0 if there are no evens}
```

9.
```
procedure palindrome check(a_1 a_2 ... a_n: string)
  answer := true
  for i := 1 to ⌊n/2⌋
    if a_i ≠ a_{n+1-i} then answer := false
  return answer
```

11.
```
procedure interchange(x, y: real numbers)
  z := x
  x := y
  y := z
```
The minimum number of assignments needed is three.

13. Linear search: $i := 1$, $i := 2$, $i := 3$, $i := 4$, $i := 5$, $i := 6$, $i := 7$, *location* := 7; binary search: $i := 1$, $j := 8$, $m := 4$, $i := 5$, $m := 6$, $i := 7$, $m := 7$, $j := 7$, *location* := 7

15.
```
procedure insert(x, a_1, a_2, ..., a_n: integers)
  {the list is in order: a_1 ≤ a_2 ≤ ··· ≤ a_n}
  a_{n+1} := x + 1
  i := 1
  while x > a_i
    i := i + 1
  for j := 0 to n - i
    a_{n-j+1} := a_{n-j}
  a_i := x
  {x has been inserted into correct position}
```

17.
```
procedure first largest(a_1, ..., a_n: integers)
  max := a_1
  location := 1
  for i := 2 to n
    if max < a_i then
      max := a_i
      location := i
  return location
```

19.
```
procedure mean-median-max-min(a, b, c: integers)
  mean := (a + b + c)/3
  {the six different orderings of a, b, c with respect
    to ≥ will be handled separately}
  if a ≥ b then
    if b ≥ c then median := b; max := a; min := c
    ⋮
  (The rest of the algorithm is similar.)
```

21.
```
procedure first-three(a_1, a_2, ..., a_n: integers)
  if a_1 > a_2 then interchange a_1 and a_2
  if a_2 > a_3 then interchange a_2 and a_3
  if a_1 > a_2 then interchange a_1 and a_2
```

23.
```
procedure onto(f: function from A to B where
    A = {a_1, ..., a_n}, B = {b_1, ..., b_m}, a_1, ..., a_n,
    b_1, ..., b_m are integers)
  for i := 1 to m
    hit(b_i) := 0
  count := 0
  for j := 1 to n
    if hit(f(a_j)) = 0 then
      hit(f(a_j)) := 1
      count := count + 1
  if count = m then return true else return false
```

25.
```
procedure ones(a: bit string, a = a_1 a_2 ... a_n)
  count:= 0
  for i := 1 to n
    if a_i := 1 then
      count := count + 1
  return count
```

27.
```
procedure ternary search(s: integer, a_1, a_2, ..., a_n:
    increasing integers)
  i := 1
  j := n
  while i < j - 1
    l := ⌊(i + j)/3⌋
    u := ⌊2(i + j)/3⌋
    if x > a_u then i := u + 1
    else if x > a_l then
      i := l + 1
      j := u
    else j := l
  if x = a_i then location := i
  else if x = a_j then location := j
```

```
    else location := 0
    return location {0 if not found}
```

29.
```
procedure find a mode(a_1, a_2, ..., a_n: nondecreasing
        integers)
    modecount := 0
    i := 1
    while i ≤ n
        value := a_i
        count := 1
        while i ≤ n and a_i = value
            count := count + 1
            i := i + 1
        if count > modecount then
            modecount := count
            mode := value
    return mode
```

31.
```
procedure find duplicate(a_1, a_2, ..., a_n: integers)
    location := 0
    i := 2
    while i ≤ n and location = 0
        j := 1
        while j < i and location = 0
            if a_i = a_j then location := i
            else j := j + 1
        i := i + 1
    return location
    {location is the subscript of the first value that
     repeats a previous value in the sequence}
```

33.
```
procedure find decrease(a_1, a_2, ..., a_n: positive
        integers)
    location := 0
    i := 2
    while i ≤ n and location = 0
        if a_i < a_{i-1} then location := i
        else i := i + 1
    return location
    {location is the subscript of the first value less than
     the immediately preceding one}
```

35. At the end of the first pass: 1, 3, 5, 4, 7; at the end of the second pass: 1, 3, 4, 5, 7; at the end of the third pass: 1, 3, 4, 5, 7; at the end of the fourth pass: 1, 3, 4, 5, 7

37.
```
procedure better bubblesort(a_1, ..., a_n: integers)
    i := 1; done := false
    while i < n and done = false
        done := true
        for j := 1 to n − i
            if a_j > a_{j+1} then
                interchange a_j and a_{j+1}
                done := false
        i := i + 1
    {a_1, ..., a_n is in increasing order}
```

39. At the end of the first, second, and third passes: 1, 3, 5, 7, 4; at the end of the fourth pass: 1, 3, 4, 5, 7 **41. a)** 1, 5, 4, 3, 2; 1, 2, 4, 3, 5; 1, 2, 3, 4, 5; 1, 2, 3, 4, 5 **b)** 1, 4, 3, 2, 5; 1, 2, 3, 4, 5; 1, 2, 3, 4, 5; 1, 2, 3, 4, 5 **c)** 1, 2, 3, 4, 5; 1, 2, 3, 4, 5; 1, 2, 3, 4, 5; 1, 2, 3, 4, 5 **43.** We carry out the linear search algorithm given as Algorithm 2 in this section, except that we replace $x \neq a_i$ by $x < a_i$, and we replace the **else** clause with **else** $location := n + 1$. **45.** $2 + 3 + 4 + \cdots + n = (n^2 + n - 2)/2$ **47.** Find the location for the 2 in the list 3 (one comparison), and insert it in front of the 3, so the list now reads 2, 3, 4, 5, 1, 6. Find the location for the 4 (compare it to the 2 and then the 3), and insert it, leaving 2, 3, 4, 5, 1, 6. Find the location for the 5 (compare it to the 3 and then the 4), and insert it, leaving 2, 3, 4, 5, 1, 6. Find the location for the 1 (compare it to the 3 and then the 2 and then the 2 again), and insert it, leaving 1, 2, 3, 4, 5, 6. Find the location for the 6 (compare it to the 3 and then the 4 and then the 5), and insert it, giving the final answer 1, 2, 3, 4, 5, 6.

49.
```
procedure binary insertion sort(a_1, a_2, ..., a_n:
        real numbers with n ≥ 2)
    for j := 2 to n
        {binary search for insertion location i}
        left := 1
        right := j − 1
        while left < right
            middle := ⌊(left + right)/2⌋
            if a_j > a_middle then left := middle + 1
            else right := middle
        if a_j < a_left then i := left else i := left + 1
        {insert a_j in location i by moving a_i through a_{j−1}
            toward back of list}
        m := a_j
        for k := 0 to j − i − 1
            a_{j−k} := a_{j−k−1}
        a_i := m
    {a_1, a_2, ..., a_n are sorted}
```

51. The variation from Exercise 50 **53. a)** Two quarters, one penny **b)** Two quarters, one dime, one nickel, four pennies **c)** A three quarters, one penny **d)** Two quarters, one dime **55.** Greedy algorithm uses fewest coins in parts (a), (c), and (d). **a)** Two quarters, one penny **b)** Two quarters, one dime, nine pennies **c)** Three quarters, one penny **d)** Two quarters, one dime **57.** The 9:00–9:45 talk, the 9:50–10:15 talk, the 10:15–10:45 talk, the 11:00–11:15 talk **59. a)** Order the talks by starting time. Number the lecture halls 1, 2, 3, and so on. For each talk, assign it to lowest numbered lecture hall that is currently available. **b)** If this algorithm uses n lecture halls, then at the point the nth hall was first assigned, it *had* to be used (otherwise a lower-numbered hall would have been assigned), which means that n talks were going on simultaneously (this talk just assigned and the $n - 1$ talks currently in halls 1 through $n - 1$). **61.** Here we assume that the men are the suitors and the women the suitees.

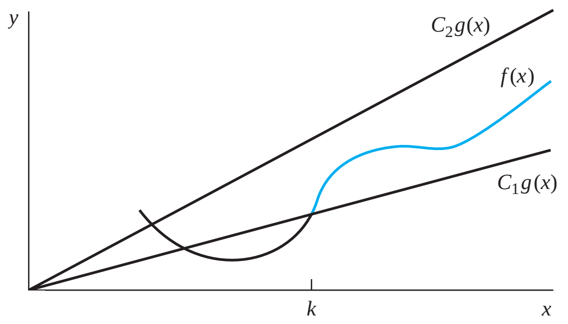
```
procedure stable(M_1, M_2, ..., M_s, W_1, W_2, ..., W_s:
    preference lists)
for i := 1 to s
    mark man i as rejected
for i := 1 to s
    set man i's rejection list to be empty
for j := 1 to s
```

set woman j's proposal list to be empty
while rejected men remain
for $i := 1$ **to** s
if man i is marked rejected **then** add i to the proposal list for the woman j who ranks highest on his preference list but does not appear on his rejection list, and mark i as not rejected
for $j := 1$ **to** s
if woman j's proposal list is nonempty **then** remove from j's proposal list all men i except the man i_0 who ranks highest on her preference list, and for each such man i mark him as rejected and add j to his rejection list
for $j := 1$ **to** s
match j with the one man on j's proposal list
{This matching is stable.}

63. If the assignment is not stable, then there is a man m and a woman w such that m prefers w to the woman w' with whom he is matched, and w prefers m to the man with whom she is matched. But m must have proposed to w before he proposed to w', because he prefers the former. Because m did not end up matched with w, she must have rejected him. Women reject a suitor only when they get a better proposal, and they eventually get matched with a pending suitor, so the woman with whom w is matched must be better in her eyes than m, contradicting our original assumption. Therefore the marriage is stable. **65.** Run the two programs on their inputs concurrently and report which one halts.

Section 3.2

1. The choices of C and k are not unique. **a)** $C = 1, k = 10$ **b)** $C = 4, k = 7$ **c)** No **d)** $C = 5, k = 1$ **e)** $C = 1, k = 0$ **f)** $C = 1, k = 2$ **3.** $x^4 + 9x^3 + 4x + 7 \le 4x^4$ for all $x > 9$; witnesses $C = 4, k = 9$ **5.** $(x^2+1)/(x+1) = x - 1 + 2/(x+1) < x$ for all $x > 1$; witnesses $C = 1, k = 1$ **7.** The choices of C and k are not unique. **a)** $n = 3, C = 3, k = 1$ **b)** $n = 3, C = 4, k = 1$ **c)** $n = 1, C = 2, k = 1$ **d)** $n = 0, C = 2, k = 1$ **9.** $x^2 + 4x + 17 \le 3x^3$ for all $x > 17$, so $x^2 + 4x + 17$ is $O(x^3)$, with witnesses $C = 3, k = 17$. However, if x^3 were $O(x^2 + 4x + 17)$, then $x^3 \le C(x^2 + 4x + 17) \le 3Cx^2$ for some C, for all sufficiently large x, which implies that $x \le 3C$ for all sufficiently large x, which is impossible. Hence, x^3 is not $O(x^2 + 4x + 17)$. **11.** $3x^4 + 1 \le 4x^4 = 8(x^4/2)$ for all $x > 1$, so $3x^4 + 1$ is $O(x^4/2)$, with witnesses $C = 8, k = 1$. Also $x^4/2 \le 3x^4 + 1$ for all $x > 0$, so $x^4/2$ is $O(3x^4+1)$, with witnesses $C = 1, k = 0$. **13.** Because $2^n \le 3^n$ for all $n > 0$, it follows that 2^n is $O(3^n)$, with witnesses $C = 1, k = 0$. However, if 3^n were $O(2^n)$, then for some C, $3^n \le C \cdot 2^n$ for all sufficiently large n. This says that $C \ge (3/2)^n$ for all sufficiently large n, which is impossible. Hence, 3^n is not $O(2^n)$. **15.** All functions for which there exist real numbers k and C with $|f(x)| \le C$ for $x > k$. These are the functions $f(x)$ that are bounded for all sufficiently large x. **17.** There are constants C_1, C_2, k_1, and k_2 such that $|f(x)| \le C_1|g(x)|$ for all $x > k_1$ and $|g(x)| \le C_2|h(x)|$ for all $x > k_2$. Hence, for $x > \max(k_1, k_2)$ it follows that $|f(x)| \le C_1|g(x)| \le C_1C_2|h(x)|$. This shows that $f(x)$ is $O(h(x))$. **19.** 2^{n+1} is $O(2^n)$; 2^{2n} is not. **21.** $1000 \log n$, $\sqrt{n}$, $n \log n$, $n^2/1000000$, 2^n, 3^n, $2n!$ **23.** The algorithm that uses $n \log n$ operations **25. a)** $O(n^3)$ **b)** $O(n^5)$ **c)** $O(n^3 \cdot n!)$ **27. a)** $O(n^2 \log n)$ **b)** $O(n^2(\log\ n)^2)$ **c)** $O(n^{2^n})$ **29. a)** Neither $\Theta(x^2)$ nor $\Omega(x^2)$ **b)** $\Theta(x^2)$ and $\Omega(x^2)$ **c)** Neither $\Theta(x^2)$ nor $\Omega(x^2)$ **d)** $\Omega(x^2)$, but not $\Theta(x^2)$ **e)** $\Omega(x^2)$, but not $\Theta(x^2)$ **f)** $\Omega(x^2)$ and $\Theta(x^2)$ **31.** If $f(x)$ is $\Theta(g(x))$, then there exist constants C_1 and C_2 with $C_1|g(x)| \le |f(x)| \le C_2|g(x)|$. It follows that $|f(x)| \le C_2|g(x)|$ and $|g(x)| \le (1/C_1)|f(x)|$ for $x > k$. Thus, $f(x)$ is $O(g(x))$ and $g(x)$ is $O(f(x))$. Conversely, suppose that $f(x)$ is $O(g(x))$ and $g(x)$ is $O(f(x))$. Then there are constants C_1, C_2, k_1, and k_2 such that $|f(x)| \le C_1|g(x)|$ for $x > k_1$ and $|g(x)| \le C_2|f(x)|$ for $x > k_2$. We can assume that $C_2 > 0$ (we can always make C_2 larger). Then we have $(1/C_2)|g(x)| \le |f(x)| \le C_1|g(x)|$ for $x > \max(k_1, k_2)$. Hence, $f(x)$ is $\Theta(g(x))$. **33.** If $f(x)$ is $\Theta(g(x))$, then $f(x)$ is both $O(g(x))$ and $\Omega(g(x))$. Hence, there are positive constants C_1, k_1, C_2, and k_2 such that $|f(x)| \le C_2|g(x)|$ for all $x > k_2$ and $|f(x)| \ge C_1|g(x)|$ for all $x > k_1$. It follows that $C_1|g(x)| \le |f(x)| \le C_2|g(x)|$ whenever $x > k$, where $k = \max(k_1, k_2)$. Conversely, if there are positive constants C_1, C_2, and k such that $C_1|g(x)| \le |f(x)| \le C_2|g(x)|$ for $x > k$, then taking $k_1 = k_2 = k$ shows that $f(x)$ is both $O(g(x))$ and $\Theta(g(x))$.

35. 

37. If $f(x)$ is $\Theta(1)$, then $|f(x)|$ is bounded between positive constants C_1 and C_2. In other words, $f(x)$ cannot grow larger than a fixed bound or smaller than the negative of this bound and must not get closer to 0 than some fixed bound. **39.** Because $f(x)$ is $O(g(x))$, there are constants C and k such that $|f(x)| \le C|g(x)|$ for $x > k$. Hence, $|f^n(x)| \le C^n|g^n(x)|$ for $x > k$, so $f^n(x)$ is $O(g^n(x))$ by taking the constant to be C^n. **41.** Because $f(x)$ and $g(x)$ are increasing and unbounded, we can assume $f(x) \ge 1$ and $g(x) \ge 1$ for sufficiently large x. There are constants C and k with $f(x) \le Cg(x)$ for $x > k$. This implies that $\log f(x) \le \log C + \log g(x) < 2 \log g(x)$ for sufficiently large x. Hence, $\log f(x)$ is $O(\log g(x))$. **43.** By definition there are positive constraints C_1, C_1', C_2, C_2', k_1, k_1', k_2, and k_2' such that $f_1(x) \ge C_1|g(x)|$ for all $x > k_1$, $f_1(x) \le C_1'|g(x)|$ for all $x > k_1'$, $f_2(x) \ge C_2|g(x)|$ for all $x > k_2$, and $f_2(x) \le C_2'|g(x)|$ for all $x > k_2'$. Adding the first and third inequalities shows that $f_1(x) + f_2(x) \ge (C_1 + C_2)|g(x)|$ for all $x > k$ where