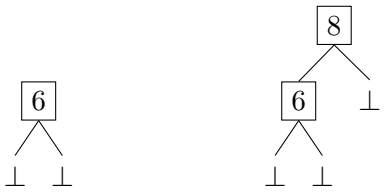
Suggested Solutions to Midterm Problems

1. The set of all full binary trees that store non-negative integer key values may be defined inductively as follows.

   (a) $FBT(k, \perp, \perp, 0)$, for any non-negative integer k, is a full binary tree of height 0.

   (b) If t_l and t_r are full binary trees of height h, then $FBT(k, t_l, t_r, h+1)$, for any non-negative integer k, is a full binary tree of height $h+1$.

   Please give a similar inductive definition for the set of all complete binary trees (of the form $CBT(\cdot, \cdot, \cdot, \cdot)$) that store non-negative integer key values. For instance, $CBT(6, \perp, \perp, 0)$ is a single-node complete binary tree storing key value 6 and $CBT(8, CBT(6, \perp, \perp, 0), \perp, 1)$ is a complete binary tree with two nodes — the root and its left child, storing key values 8 and 6 repsectively. Pictorially, they may be depicted as below.



   Solution. The set of all (non-empty) complete binary trees may be defined as follows:

   (a) $CBT(k, \perp, \perp, 0)$, for any non-negative integer k, is a complete binary tree of height 0, and $CBT(k_1, CBT(k_2, \perp, \perp, 0), \perp, 1)$, for any non-negative integers k_1 and k_2, is a (proper) complete binary tree of height 1.

   (b) Suppose t_l and t_r are complete binary trees.

      i. If t_l is a full binary tree of height h and t_r is a complete binary tree of height h, then $CBT(k, t_l, t_r, h+1)$, for any non-negative integer k, is a complete binary tree of height $h+1$.

      ii. If t_l is a complete tree of height h and t_r is a full binary tree of height $h-1$, then $CBT(k, t_l, t_r, h+1)$, for any non-negative integer k, is a complete binary tree of height $h+1$.

   Here by saying "a complete binary tree t is a full binary tree," we mean that, when every occurrence of CBT in $t = CBT(\cdot, \cdot, \cdot, \cdot)$ is replaced by FBT, it is indeed a full binary tree according to the definition of $FBT(\cdot, \cdot, \cdot, \cdot)$. (Mathematically, a CBT is a full binary tree if it is isomorphic to some FBT.)

   Note: as the definition is intended to exclude the empty tree as a complete binary tree, it includes as a base case the (proper) complete binary tree (with two nodes) of height 1, to avoid the need, in the inductive step, of treating the special case of a complete binary tree without a right child. □

2. Prove *by induction* that the sum of the heights of all nodes in a complete binary tree with n nodes is at most $n-1$. You may assume it is known that the sum of the heights of all

nodes in a *full* binary tree of height h is $2^{h+1} - h - 2$. (Note: a single-node tree has height 0.)

Solution. From the solution to the preceding problem, we see that a complete binary tree is:

(a) a single-node tree of height 0,

(b) a two-node tree of height 1 where the root has a left child,

(c) composed from a full binary tree of height h with n_l nodes and a complete (possibly full) binary tree of height h with n_r nodes as the left and the right subtrees of the root, resulting in a tree of height $h+1$ with $n_l + n_r + 1$ nodes, or

(d) composed from a complete (possibly full) binary tree of height h with n_l nodes and a full binary tree of height $h-1$ with n_r nodes as the left and the right subtrees of the root, resulting also in a tree of height $h+1$ with $n_l + n_r + 1$ nodes.

Let $G(n)$ denote the sum of the heights of all nodes in a complete binary tree with n nodes. For a full binary tree (as a special case of complete binary trees) with $n = 2^{h+1} - 1$ nodes where h is the height of the tree, we already know that $G(n) = 2^{h+1} - (h+2) = n - (h+1) \leq n - 1$. With this as a basis, we prove that $G(n) \leq n - 1$ for the general case of arbitrary complete binary trees by induction on the number n (≥ 1) of nodes.

Base case ($n = 1$ or $n = 2$): When $n = 1$, the tree contains a single node whose height is 0. So, $G(n) = 0 \leq 1 - 1 = n - 1$. When $n = 2$, the tree has one additional node as the left child of the root. The height of the root is 1, while that of its left child is 0. So, $G(n) = 1 \leq 2 - 1 = n - 1$.

Inductive step ($n > 2$): If n happens to be equal to $2^{h+1} - 1$ for some $h \geq 1$, i.e., the tree is full, then we are done; note that this in particular covers the case of $n = 3 = 2^{1+1} - 1$. Otherwise, suppose $2^{h+1} - 1 < n < 2^{h+2} - 1$ ($h \geq 1$), i.e., the tree is a "proper" complete binary tree with height $h + 1 \geq 2$. There are two cases to consider:

Case 1: The left subtree is full of height h with n_l nodes and the right one is complete also of height h with n_r nodes (such that $n_l + n_r + 1 = n$). From the special case of full binary trees and the induction hypothesis, $G(n_l) = 2^{h+1} - (h+2) = n_l - (h+1)$ and $G(n_r) \leq n_r - 1$. $G(n) = G(n_l) + G(n_r) + (h+1) \leq (n_l - (h+1)) + (n_r - 1) + (h+1) = (n_l + n_r + 1) - 2 \leq n - 1$.

Case 2: The left subtree is complete of height h with n_l nodes and the right one is full of height $h-1$ with n_r nodes. From the induction hypothesis and the special case of full binary trees, $G(n_l) \leq n_l - 1$ and $G(n_r) = 2^h - (h+1) = n_r - h$. $G(n) = G(n_l) + G(n_r) + (h+1) \leq (n_l - 1) + (n_r - h) + (h+1) = (n_l + n_r + 1) - 1 = n - 1$. □

3. The Knapsack Problem that we discussed in class is defined as follows: Given a set S of n items, where the ith item has an integer size $S[i]$, and an integer K, find a subset of the items whose sizes sum to exactly K or determine that no such subset exists.

   We have described in class an algorithm (see the Appendix) to solve the problem. Please provide, for each of the following two requirements, a modification to the algorithm that meets the requirement. (For each case, you may just indicate the changes that should be made to the original algorithm.)

   (a) The values of $P[i,k].belong$ ($0 \leq i \leq n$ and $0 \leq k \leq K$) record the subset (if one exists) with the fewest items whose sizes sum to K.

   Solution. To obtain the subset (if one exists) with the fewest items, we should try the larger items before the smaller ones. For instance, suppose S contains four items

with sizes $S[1] = 1$, $S[2] = 2$, $S[3] = 3$, and $S[4] = 4$ and K is 6. If we try the items in the given order, we will get the combination of $S[1]$, $S[2]$, and $S[3]$. On the other hand, if we try the larger items earlier, then we will get the combination of $S[4]$ and $S[2]$, one item fewer than the previous combination. So, to meet the requirement, we simply sort the items in decreasing order of their sizes, before starting the original procedure. □

(b) The type of $P[i,k].exist$ becomes integer and it gives the number of possible subsets of items from $S[1..i]$ whose sizes sum to exactly k. In this case, the values of $P[i,k].belong$ are not useful and can be omitted.

Solution. The basic idea is for $P[i,k].exist$ to inherit the value of $P[i-1,k].exist$ which records the number of subsets from 1-st to $(i-1)$-th items and hence is also the number of subsets from 1-st to i-th items but excluding i-th item that meet the requirement of having item sizes summing to k. To this, we then add $P[i-1,k-S[i]].exist$, if $k - S[i] \geq 0$, which is the number of subsets from 1-st to $(i-1)$-th items, each of which contains items whose sizes sum to $k - S[i]$ and hence is also the number of subsets from 1-st to i-th items always including i-th item, each of which contains items whose sizes sum to k.

Algorithm Knapsack (S, K)**;**
begin
  $P[0,0].exist := 1$;
  for $k := 1$ **to** K **do**
    $P[0,k].exist := 0$;
  for $i := 1$ **to** n **do**
    for $k := 0$ **to** K **do**
      $P[i,k].exist := P[i-1,k].exist$;
      if $k - S[i] \geq 0$ **then**
        $P[i,k].exist := P[i,k].exist + P[i-1,k-S[i]].exist$;
end

□

4. You are asked to design a schedule for a round-robin tennis tournament. There are $n = 2^k$ $(k \geq 1)$ players. Each player must play every other player, and each player must paly one match per round for $n-1$ rounds. Denote the players by $P_1, P_2, \ldots, P_n$. Output the schedule for each player. (Hint: use divide and conquer in the following way. First, divide the players into two equal groups and let them play within the groups for the first $\frac{n}{2} - 1$ rounds. Then, design the games between the groups for the other $\frac{n}{2}$ rounds.)

Solution.

Algorithm Tournament(n);
// The number of players $n = 2^k$ for some $k \geq 1$.
// $O[r,i] = j$ indicates that in Round r $(1 \leq r \leq n-1)$ the opponent of P_i is P_j.
begin
  $Schedule(1, n, 1)$;
end

procedure Schedule(L, R, r);
begin

```
if R − L = 1 then
    O[r, L], O[r, R] := R, L;
else
    M := ⌊(L+R)/2⌋;
    Schedule(L, M, r);        // M − L + 1 = R − (M + 1) + 1 (half of the players).
    Schedule(M + 1, R, r);
    // Rounds r to (r + (M − L) − 1) have been scheduled by the recursive calls.
    // Next schedule Rounds (r + (M − L)) to (r + (M − L) + (M − L))
    // between players L to M and players (M + 1) to R.
    for k from (M − L) to ((M − L) + (M − L)) do
        for i from L to M do
            O[r + k, i] := (M + 1) + (((i − L) + (k − (M − L))) mod (M − L + 1));
            O[r + k, (M + 1) + (((i − L) + (k − (M − L))) mod (M − L + 1))] := i;
            // Alternatively,
            // O[r + k, i] := (M + 1) + ((i + k) mod (M − L + 1));
            // O[r + k, (M + 1) + ((i + k) mod (M − L + 1))] := i;
end
```

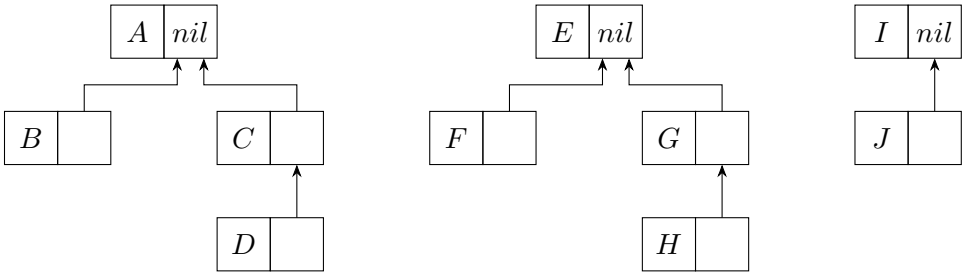
□

5. Consider the solutions to the union-find problem discussed in class. Suppose we start with a collection of ten elements: A, B, C, D, E, F, G, H, I, and J, and the following sequence of operations are performed: union(A,B), union(C,D), union(E,F), union(G,H), union(I,J), union(A,D), union(F,G), union(D,J), union(J,H).

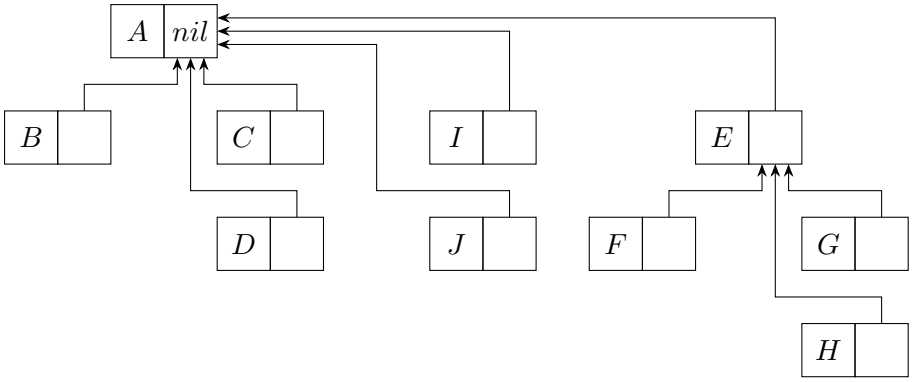
   Assuming that both the balancing and the path compression techniques are used, draw (1) a diagram showing the grouping of these ten elements immediately after union(F,G) is performed and (2) another after the whole sequence of operations are performed. In the case of combining two groups of the same size, please always point the second group to the first.

   Solution.

   Immediately after union(F,G):



   After the whole sequence of operations are performed:



□

6. Consider rearranging the following array into a max heap using the *bottom-up* approach.

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
3	2	7	5	1	13	8	6	4	11	10	14	15	12	9

Please show the result (i.e., the contents of the array) after a new element is added to the current collection of heaps (at the bottom) until the entire array has become a heap.

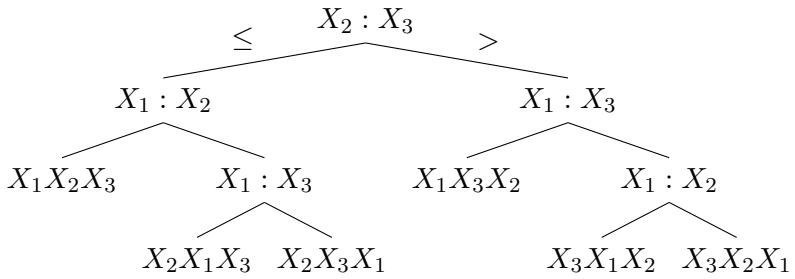
Solution.

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
3	2	7	5	1	13	8	6	4	11	10	14	15	12	9
3	2	7	5	1	13	<u>12</u>	6	4	11	10	14	15	<u>8</u>	9
3	2	7	5	1	<u>15</u>	12	6	4	11	10	14	<u>13</u>	8	9
3	2	7	5	<u>11</u>	15	12	6	4	<u>1</u>	10	14	13	8	9
3	2	7	<u>6</u>	11	15	12	<u>5</u>	4	1	10	14	13	8	9
3	2	<u>15</u>	6	11	<u>14</u>	12	5	4	1	10	<u>7</u>	13	8	9
3	<u>11</u>	15	6	<u>10</u>	14	12	5	4	1	<u>2</u>	7	13	8	9
<u>15</u>	11	<u>14</u>	6	10	<u>13</u>	12	5	4	1	2	7	<u>3</u>	8	9

□

7. Draw a decision tree of the Mergesort algorithm for the case of $A[1..3]$, i.e., $n = 3$. In the decision tree, you must indicate (1) which two elements of the original input array are compared in each internal node and (2) the sorting result in each leaf. Please use X_1, X_2, X_3 (not $A[1]$, $A[2]$, $A[3]$) to refer to the elements (in this order) of the original input array A.

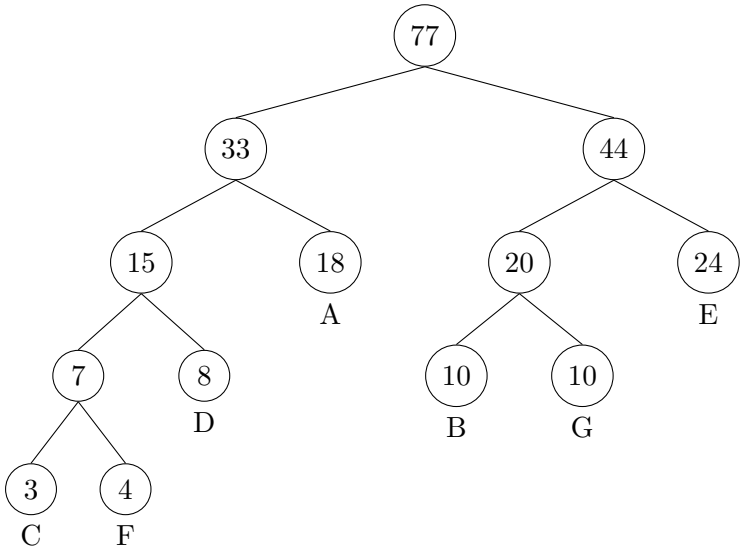
Solution.



□

8. Construct a Huffman code tree for a text composed from seven characters A, B, C, D, E, F, and G with frquencies 18, 10, 3, 8, 24, 4, and 10 respectively. And then, list the codes for all the characters according to the code tree.

Solution.



Character	Frequency	Code
A	18	01
B	10	100
C	3	0000
D	8	001
E	24	11
F	4	0001
G	10	101

□

9. The *next* table is a precomputed table (for $B = b_1 b_2 \cdots b_m$) that plays a critical role in the KMP algorithm. Under what condition (regarding $b_1 b_2 \cdots b_i$) does $next[i]$ (for $i > 0$) get a 0? And under what condition can it be safely set to -1 (without missing a potential match)?

Solution. The value of $next[i]$ is determined by the length of the longest prefix of $b_1 b_2 \cdots b_{i-1}$ that is also a suffix of $b_1 b_2 \cdots b_{i-1}$. When no such prefix exists, $next[i]$ gets a 0.

During a search for string B in string A using KMP, when b_j is compared against a_i and the comparison fails, $b_{next[j]+1}$ is tried next against a_i. When $next[j] = 0$, it is b_1 that is compared with a_i. If the comparison fails, then b_1 will be compared against a_{i+1}, according to the case for $next[j]+1 = 0$, i.e., $next[j] = -1$. When $b_1 = b_j$, the comparison between b_1 and a_i is doomed to fail (since $b_1 = b_j \neq a_i$) and the comparison could have been saved. To achieve the saving, we can set $next[j]$ to -1 (instead of 0) when b_j happens to be equal to b_1. □

10. The Fibonacci word sequence of bit strings is defined as follows:

$$F(n) = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ F(n-1) \cdot F(n-2) & \text{if } n \geq 2 \end{cases}$$

Here $\cdot$ denotes the operation of string concatenation. The first six Fibonacci words (from $F(0)$ to $F(5)$) are: 0, 1, 10, 101, 10110, 10110101.

Design an algorithm that, given a bit pattern p and a number n, determines whether p occurs in $F(n)$. For instance, 1101 occurs in $F(5)$, but not $F(4)$. Please present your algorithm in adequate pseudocode. Explain why it is correct and give an analysis of its time complexity. The more efficient your algorithm is, the more points you will be credited for this problem.

Solution. When the given bit pattern p is of length 1, i.e., p equals "0" or "1", it can be trivially determined whether p occurs in $F(n)$ for the given n. So, let us consider $|p| \geq 2$ and assume that m is the smallest number such that $|p| \leq |F(m)|$.

We claim that, if p does not occur in $F(m)$, $F(m+1)$, $F(m+2)$, or $F(m+3)$, then it will never occur in any Fibonacci word later in the sequence. With this claim (to be proven later), an algorithm that, given a bit pattern p and a number n, determines whether p occurs in $F(n)$ may proceed as function **Find** defined below. If p occurs in $F(n)$, **Find**(p, n) returns the smallest m $(0 \leq m \leq n)$ such that p occurs in $F(m)$ already; otherwise, it returns -1. Function **Find** assumes the availability of the KMP algorithm and the preprocessing routine to compute the *next* table.

```
function Find(p, n) : integer;
begin
    if p = "0" and n ≥ 0 then Find := 0
    else if p = "1" and n ≥ 1 then Find := 1
    else if p is non-empty and n ≥ 2 then
        p_len := the length of p;
        m := 2;
        F_m, F_m1, F_m2 := "10", "1", "0";
        l_m, l_m1, l_m2 := 2, 1, 1;
        while l_m < p_len do
            m := m + 1;
            F_m, F_m1, F_m2 := F_m1 · F_m2, F_m, F_m1;
            l_m, l_m1, l_m2 := l_m1 + l_m2, l_m, l_m1;
        compute the next table for p (to use KMP);
        found, i := false, 0;
        while not found and i < 4 do
            if p is a substring of F_m (checked by KMP) then
                found := true
            else
                i := i + 1;
                F_m, F_m1, F_m2 := F_m1 · F_m2, F_m, F_m1;
        if found then Find := m + i
        else Find := −1
    else Find := −1
end
```

The length of $F(m)$ is at most $2 \times |p|$, while the length of $F(m+3)$ is at most $8 \times |F(m)|$. So, $|F(m)|$, $|F(m+1)|$, $|F(m+2)|$, and $|F(m+3)|$ are all $O(|p|)$. Checking whether p is a substring of $F(m+i)$, for $0 \leq i \leq 3$, using the KMP algorithm in the second while loop takes $4 \times O(|p|)$, i.e., $O(|p|)$. The first while loop takes $O(|p|)$ time, mainly to compute the final $F(m)$. The computation of *next* also takes $O(|p|)$ time. So, the total time complexity is $O(|p|)$.

Proof of the claim:

$$\begin{aligned} F(m+2) &= F(m+1) \cdot F(m) \\ F(m+3) &= F(m+2) \cdot F(m+1) \\ &= [F(m+1) \cdot F(m)] \cdot [F(m) \cdot F(m-1)] \end{aligned}$$

Given that p (with $|p| \leq |F(m)|$) is not a substring of $F(m)$, $F(m+1)$, $F(m+2)$, or $F(m+3)$, it is also not a substring of $F(m+1) \cdot F(m)$ or $F(m) \cdot F(m)$.

$$\begin{aligned} F(m+4) &= F(m+3) \cdot F(m+2) \\ &= [F(m+2) \cdot F(m+1)] \cdot [F(m+1) \cdot F(m)] \\ &= [F(m+2) \cdot F(m+1)] \cdot [F(m) \cdot F(m-1) \cdot F(m)] \end{aligned}$$

Since p is not a substring of $F(m+3)$ or $F(m+2)$, for p to occur in $F(m+4)$, it must straddle on the boundary between $F(m+3)$ and $F(m+2)$ and be a substring of $F(m+1) \cdot F(m)$, which contradicts the known fact that p is not a substring of $F(m+1) \cdot F(m)$.

$$\begin{aligned} F(m+5) &= F(m+4) \cdot F(m+3) \\ &= [F(m+3) \cdot F(m+2)] \cdot [F(m+2) \cdot F(m+1)] \\ &= [F(m+3) \cdot F(m+1) \cdot F(m)] \cdot [F(m+1) \cdot F(m) \cdot F(m+1)] \\ &= [F(m+3) \cdot F(m+1) \cdot F(m)] \cdot [F(m) \cdot F(m-1) \cdot F(m) \cdot F(m+1)] \end{aligned}$$

Since p is not a substring of $F(m+4)$ or $F(m+3)$, for p to occur in $F(m+5)$, it must straddle on the boundary between $F(m+4)$ and $F(m+3)$ and be a substring of $F(m) \cdot F(m)$, which contradicts the known fact that p is not a substring of $F(m) \cdot F(m)$.

All subsequent cases may be reasoned similarly; the whole proof could be presented more formally as a proof by induction on i (≥ 0) as an increment in $F(m+i)$.

□

Appendix

- An algorithm for determining whether a solution to the (original) Knapsack Problem exists:

```
Algorithm Knapsack (S, K);
begin
    P[0,0].exist := true;
    for k := 1 to K do
        P[0,k].exist := false;
    for i := 1 to n do
        for k := 0 to K do
            P[i,k].exist := false;
            if P[i-1,k].exist then
              P[i,k].exist := true;
              P[i,k].belong := false
```

```
            else if k − S[i] ≥ 0 then
                    if P[i − 1, k − S[i]].exist then
                       P[i, k].exist := true;
                       P[i, k].belong := true
end
```

- The Mergesort algorithm:

```
Algorithm Mergesort (X, n);
begin  M_Sort(1, n)  end

procedure M_Sort (Left, Right);
begin
    if Right − Left = 1 then
       if X[Left] > X[Right] then swap(X[Left], X[Right])
    else if Left ≠ Right then
            Middle := ⌈½(Left + Right)⌉;
            M_Sort(Left, Middle − 1);
            M_Sort(Middle, Right);
            // the merge part
            i := Left;  j := Middle;  k := 0;
            while (i ≤ Middle − 1) and (j ≤ Right) do
                    k := k + 1;
                    if X[i] ≤ X[j] then
                       TEMP[k] := X[i];  i := i + 1
                    else TEMP[k] := X[j];  j := j + 1;
            if j > Right then
               for t := 0 to Middle − 1 − i do
                    X[Right − t] := X[Middle − 1 − t]
            for t := 0 to k − 1 do
                X[Left + t] := TEMP[1 + t]
end
```

- The KMP algorithm (assuming *next*):

```
Algorithm String_Match (A, n, B, m);
begin
    j := 1;  i := 1;
    Start := 0;
    while Start = 0 and i ≤ n do
        if B[j] = A[i] then
            j := j + 1;  i := i + 1
        else
            j := next[j] + 1;
            if j = 0 then
                j := 1;  i := i + 1;
        if j = m + 1 then Start := i − m
end
```

- The algorithm for computing the *next* table in the KMP algorithm:

```
Algorithm Compute_Next (B, m);
begin
    next[1] := −1;  next[2] := 0;
    for i := 3 to m do
        j := next[i − 1] + 1;
        while B[i − 1] ≠ B[j] and j > 0 do
            j := next[j] + 1;
        next[i] := j
end
```