# Suggested Solutions to Midterm Problems

1. Prove *by induction* that, for a complete binary tree, one of the two subtrees under the root is a full binary tree and the other is a complete binary tree. An empty tree may be considered a full binary tree and also a complete binary tree. (Note: full binary trees are special cases of complete binary trees.)

   *Solution.* For a complete binary tree with $n$ ($\geq 1$) nodes, its nodes can be numbered 1 through $n$ compactly such that the root is numbered 1 and, for a node numbered $i$ ($\geq 1$), its left child (if existent) is numbered $2i$ and its right child (if existent) is numbered $2i+1$. Conversely, a binary tree whose nodes can be compactly numbered as above must be a complete binary tree.

   The height (or depth) of a complete binary tree is the number of levels (or parent-child edges) one needs to go through from the root to the last ($n$-th) node; the height of a single-node tree is 0 by the definition. For convenience, the empty tree is considered of height $-1$. The number of nodes of a full binary tree can be calculated as $2^{h+1}-1$, where $h$ is the height of the tree. We say that a complete binary tree is *proper* if it is not a full binary tree.

   To facilitate the inductive proof, we refine/strengthen the proposition in the problem statement as follows. Note that the empty tree satisfies the proposition vacuously.

   > For a complete binary tree with $n$ ($\geq 1$) nodes, the two subtrees of the root satisfy exactly one of the following four conditions:
   >
   > (a) The two subtrees are both full binary trees of the same height. In this case, the entire tree is a full binary tree, i.e., $n = 2^i - 1$ for some $i \geq 1$, and the $n$-th node is in the right subtree if it is not empty.
   >
   > (b) The left subtree is a proper complete binary tree and is one-level taller than the right subtree, which is a full binary tree. In this case, the $n$-th node is in the left subtree.
   >
   > (c) The left subtree is a full binary tree and is as tall as as the right subtree, which is a proper complete binary tree. In this case, the $n$-th node is in the right subtree.
   >
   > (d) The two subtrees are both full binary trees and the left subtree is one-level taller than the right subtree. In this case, the $n$-th node is in the left subtree.

   Now the proposition can readily be proven by reversed induction on the number of nodes.

   Base cases: consider the complete binary trees with 0, 1, 3, 7, $\ldots$, $2^i - 1$, $\ldots$ nodes, i.e., all full binary trees, including the empty (full binary) tree. For the empty tree, the proposition holds vacuously. And, for every nonempty full binary tree, the two subtrees of the root are clearly also full binary trees. Condition (a) of the refined propostion holds.

   Inductive step: assuming that the proposition holds for an arbitrary complete binary tree with $n$ nodes, we need to show that the proposition also holds for the complete binary tree with $n-1$ nodes, where $n-1 \geq 2$ and $n-1 \neq 2^i - 1$ for any $i \geq 2$. The $(n-1)$-node tree is obtained from the $n$-node tree by removing the $n$-th node. For each of the four conditions the $n$-node tree may satisfy, we argue that the $(n-1)$-node tree also satifies one of the four conditions:

(a) In this case, the removal of the $n$-th node turns the right subtree into a proper complete binary tree of the same height and it follows that Condition (c) holds for the $(n-1)$-node tree.

(b) In this case, the $n$-th node is in the left subtree, which is a proper complete binary tree. And, as $n-1 \neq 2^i - 1$ for any $i \geq 2$, after the removal of the $n$-th node, the left subtree remains a proper complete binary tree of the same height. So, Condition (b) also holds for the $(n-1)$-node tree.

(c) In this case, after the removal of the $n$-th node, the right subtree either remains a proper complete binary tree of the same height or becomes a full binary tree one-level shorter. Consequently, either Condition (c) or Condition (d) holds for the $(n-1)$-node tree.

(d) In this case, the removal of the $n$-th node turns the left subtree into a proper complete binary tree of the same height and therefore Condition (b) holds for the $(n-1)$-node tree.

$\square$

2. Consider bounding summations by integrals. We already know that, if $f(x)$ is monotonically *increasing*, then

$$\sum_{i=1}^{n} f(i) \leq \int_{1}^{n+1} f(x)dx.$$

(a) The sum may also be bounded from below as follows:

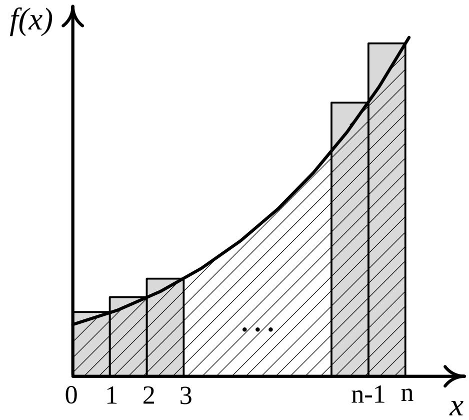
$$\int_{0}^{n} f(x)dx \leq \sum_{i=1}^{n} f(i).$$

Show that this is indeed the case.

*Solution.* Given that $f(x)$ is monotonically increasing, we have

$$\begin{array}{rcl}
\int_0^1 f(x)dx & \leq & f(1) \\
\int_1^2 f(x)dx & \leq & f(2) \\
\int_2^3 f(x)dx & \leq & f(3) \\
 & \cdots & \\
\int_{n-2}^{n-1} f(x)dx & \leq & f(n-1) \\
\int_{n-1}^{n} f(x)dx & \leq & f(n) \\
\hline
\int_0^n f(x)dx & \leq & \sum_{i=1}^{n} f(i)
\end{array}$$

So, the lower bound for the summation $\sum_{i=1}^{n} f(i)$ is correct. This is also easily seen by comparing the areas (on the $R \times R$ plane) defined by the formulae on the two sides. As shown in the following diagram, the integral $\int_0^n f(x)dx$ equals the area under the curve that is shaded with thin parallel lines. The area is apparently no larger than the total area of the vertical bars which represents $\sum_{i=1}^{n} f(i)$.




$\square$

(b) Prove, using this bounding technique, that $\sum_{i=1}^{n} \frac{1}{i} = \Theta(\log n)$. Note that $\frac{1}{i}$ actually decreases when $i$ increases.

*Solution.* As $\frac{1}{i}$ is monotonically decreasing and the bounding technique cannot be directly applied, we rewrite the sum as $\sum_{i=1}^{n} \frac{1}{(n+1)-i}$. Now we have a monotonically increasing $f(x) = \frac{1}{(n+1)-x}$, for $x < n+1$. We know that $\int \frac{1}{(n+1)-x} dx = -\ln((n+1)-x)$, for $x < n+1$.
$\sum_{i=1}^{n} \frac{1}{i} = \sum_{i=1}^{n} \frac{1}{(n+1)-i} \geq \int_0^n \frac{1}{(n+1)-x} dx = -\ln((n+1)-n) - (-\ln((n+1)-0)) = \ln(n+1) \geq \ln n \geq \frac{1}{\log e} \log n$. So, $\sum_{i=1}^{n} \frac{1}{i} = \Omega(\log n)$.
$\sum_{i=1}^{n} \frac{1}{i} = \sum_{i=1}^{n} \frac{1}{(n+1)-i} = 1 + \sum_{i=1}^{n-1} \frac{1}{(n+1)-i} \leq 1 + \int_1^n \frac{1}{(n+1)-x} dx = 1 + (-\ln((n+1)-n) - (-\ln((n+1)-1))) = 1 + \ln n \leq \frac{1}{\log e} \log n + \frac{1}{\log e} \log n \leq \frac{2}{\log e} \log n$ (for $n \geq 3$). So, $\sum_{i=1}^{n} \frac{1}{i} = O(\log n)$.
It follows that $\sum_{i=1}^{n} \frac{1}{i} = \Theta(\log n)$. $\square$

3. Consider the problem of merging two skylines, which is a useful building block for computing the skyline of a number of buildings. A skyline is an alternating sequence of $x$ coordinates and $y$ coordinates (heights), ending with an $x$ coordinate (as discussed in class). The sequence of coordinates may be coveniently stored in an array, say $A$, with $A[0]$ storing the first $x$ coordinate, $A[1]$ the first $y$ coordinate, $A[2]$ the second $x$ coordinate, etc.

   Design a linear-time procedure that prints out the resulting skyline from merging two given skylines. Please present the procedure in suitable pseudocode. The procedure should be named `merge_skylines` and invoked by `merge_skylines(A,m,B,n)`, where `A` and `B` are the two input skylines and `A[m]` and `B[n]` store the final $x$ coordinate of skyline `A` and that of skyline `B` respectively.

   *Solution.*

```
merge_skylines(A,m,B,n)
// assume m,n >= 2.
begin
  if A[0] < B[0] then
    print A[0], A[1];
    merge_a(A[1], 0, A[2..m], m-2, B, n);
```

```
  else
    if A[0] > B[0] then
      print B[0], B[1];
      merge_b(0, B[1], A, m, B[2..n], n-2);
    else // A[0] = B[0]
      if A[1] < B[1] then
        print B[0], B[1];
        merge_b(A[1], B[1], A[2..m], m-2, B[2..n], n-2);
      else // A[1] > B[1] or A[1] = B[1] (given A[0] = B[0])
        print A[0], A[1];
        merge_a(A[1], B[1], A[2..m], m-2, B[2..n], n-2);
      end if;
    end if;
  end if;
end

merge_a(ya, yb, A, m, B, n);
// ya, yb are the previous y coordinates of A and B, respectively.
// ya > yb.
begin
  if m = 0 and n = 0 then
    if A[0] < B[0] then
      print A[0], yb, B[0];
      return;
    else
      print A[0];
      return;
    end if;
  end if;
  if m = 0 then
    if A[0] < B[0] then
      print A[0], yb, each entry of B;
      return;
    else
      merge_a(ya, yb, A, m, B[2..n], n-2);
      return;
    end if;
  end if;
  if n = 0 then
    if A[0] < B[0] then
      if A[1] < yb then
        print A[0], yb;
        merge_b(A[0], yb, A[2..m], m-2, B, n);
        return;
      else
        print A[0], A[1];
        merge_a(A[1], yb, A[2..m], m-2, B, n);
        return;
      end if;
```

```
    else
      print each entry of A;
      return;
    end if;
  end if;
  if A[0] < B[0] then
    if A[1] > yb then
      print A[0], A[1];
      merge_a(A[1], yb, A[2..m], m-2, B, n);
    else
      print A[0], yb;
      merge_b(A[1], yb, A[2..m], m-2, B, n);
    end if;
  else
    if A[0] > B[0] then
      if B[1] > ya then
        print B[0], B[1];
        merge_b(ya, B[1], A, m, B[2..n], n-2);
      else
        merge_a(ya, yb, A, m, B[2..n], n-2);
      end if;
    else // A[0] = B[0]
      if A[1] < B[1] then
        if B[1] = ya then
          merge_b(ya, B[1], A[2..m], m-2, B[2..n], n-2);
        else
          merge_a(A[1], B[1], A[2..m], m-2, B[2..n], n-2);
        end if;
      else // A[1] > B[1] or A[1] = B[1] (given A[0] = B[0])
        print A[0], A[1];
        merge_a(A[1], B[1], A[2..m], m-2, B[2..n], n-2);
      end if;
    end if;
  end if;
end

merge_b(ya, yb, A, m, B, n);
// ya, yb are the previous y coordinates of A and B, respectively.
// ya < yb.
// analogous to merge_a.
```

□

4. The Knapsack Problem that we discussed in class is defined as follows: Given a set $S$ of $n$ items, where the $i$th 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 to solve the problem. Modify the algorithm to solve a variation of the knapsack problem where each item has an *unlimited* supply. In your algorithm, please change the type of $P[i,k].belong$ into integer and use it to record the number of copies of item $i$ needed.

*Solution.*

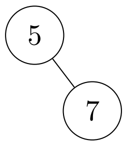
**Algorithm Knapsack_Unlimited** $(S, K)$**;**
**begin**
    $P[0,0].exist := true$;
    $P[0,0].belong := 0$;
    **for** $k := 1$ **to** $K$ **do**
        $P[0,k].exist := false$;
    **for** $i := 1$ **to** $n$ **do**
        **for** $k := 0$ **to** $K$ **do**
            $P[i,k].exist := false$;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

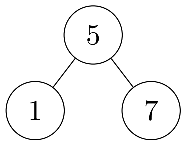
$\square$

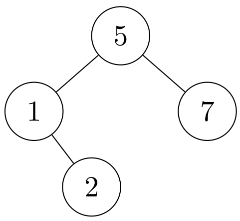
5. Show all intermediate and the final AVL trees formed by inserting the numbers 5, 7, 1, 2, 4, 3, and 6 (in this order) into an empty tree. Please use the following ordering convention: the key of an internal node is larger than that of its left child and smaller than that of its right child. If re-balancing operations are performed, please also show the tree before re-balancing and indicate what type of rotation is used in the re-balancing.

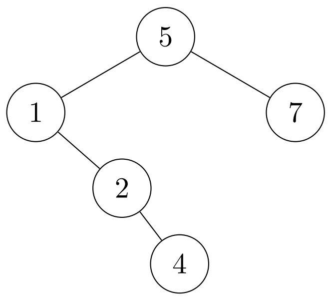
*Solution.*

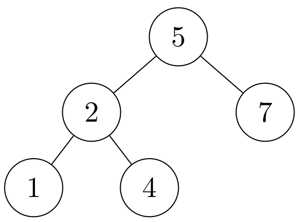
Insert 5: 

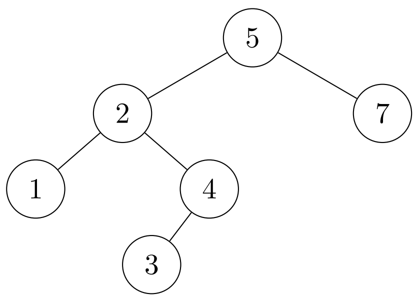

Insert 7: 

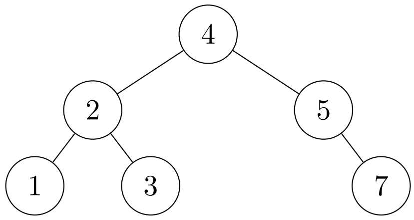

Insert 1: 


Insert 2: 

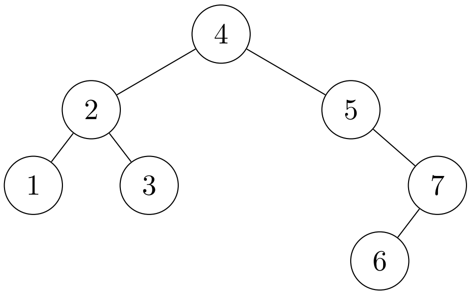

Insert 4: 


Single rotation at 1: 

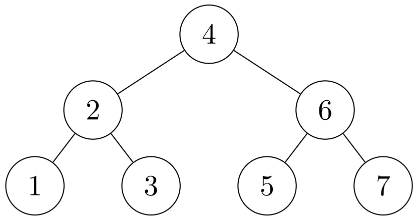

Insert 3: 



Double rotation at 5:



Insert 6:



Double rotation at 5:

□

6. Below is the Mergesort algorithm in pseudocode:

**Algorithm Mergesort** $(X, n)$;<br>
**begin** $M\_Sort(1, n)$ **end**

**procedure M_Sort** $(Left, Right)$;<br>
**begin**<br>
    **if** $Right - Left = 1$ **then**<br>
        **if** $X[Left] > X[Right]$ **then** $swap(X[Left], X[Right])$<br>
    **else if** $Left \neq Right$ **then**<br>
            $Middle := \lceil \frac{1}{2}(Left + Right) \rceil$;<br>
            $M\_Sort(Left, Middle - 1)$;<br>
            $M\_Sort(Middle, Right)$;<br>
            // the merge part<br>
            $i := Left$; $j := Middle$; $k := 0$;<br>
            **while** $(i \leq Middle - 1)$ and $(j \leq Right)$ **do**<br>
                    $k := k + 1$;<br>
                    **if** $X[i] \leq X[j]$ **then**<br>
                        $TEMP[k] := X[i]$; $i := i + 1$<br>
                    **else** $TEMP[k] := X[j]$; $j := j + 1$;<br>
            **if** $j > Right$ **then**<br>
                **for** $t := 0$ **to** $Middle - 1 - i$ **do**<br>
                    $X[Right - t] := X[Middle - 1 - t]$<br>
            **for** $t := 0$ **to** $k - 1$ **do**<br>
                $X[Left + t] := TEMP[1 + t]$<br>
**end**

Given the array below as input, what are the contents of array *TEMP* after the merge part is executed for the first time and what are the contents of *TEMP* when the algorithm terminates? Assume that each entry of *TEMP* has been initialized to 0 when the algorithm starts.

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 7 | 6 | 3 | 8 | 5 | 10 | 11 | 2 | 1 | 12 | 4 | 9 |

*Solution.*

The contents of array *TEMP* after the merge part is executed for the first time:

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 3 | 6 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

The contents of array *TEMP* when the algorithm terminates:

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 0 | 0 |

□

7. The partition procedure in the Quicksort algorithm chooses an element as the pivot and divide the input array $A[1..n]$ into two parts such that, when the pivot is properly placed in $A[i]$, the entries in $A[1..(i-1)]$ are less than or equal to $A[i]$ and the entries in $A[(i+1)..n]$ are greater than or equal to $A[i]$. Please design an extension of the partition procedure so that it chooses two pivots and divides the input array into three parts. Assuming the two pivots are eventually placed in $A[i]$ and $A[j]$ ($i < j$) respectively, the entries in $A[1..(i-1)]$ are less than or equal to $A[i]$, the entries in $A[(i+1)..(j-1)]$ are greater than or equal to $A[i]$ and less than or equal to $A[j]$, and the entries in $A[(j+1)..n]$ are greater than or equal to $A[j]$.

   Please present your extension in adequate pseudocode and make assumptions wherever necessary. Give an analysis of its time complexity. The more efficient your algorithm is, the more points you will be credited for this problem.

   *Solution.*

```
Partition3(X, Left, Right);
begin
  if X[Left] > X[Right] then
    swap(X[Left], X[Right])
  end if;
  pivot1 := X[Left];
  pivot2 := X[Right];
  i := Left;
  k := Right;
  j := Left + 1;
  while (j < k) do
    if X[j] < pivot1 then
      i := i + 1;
      swap(X[i], X[j]);
      j := j + 1;
    else
      if X[j] > pivot2 then
        k := k - 1;
```

```
        swap(X[j], X[k]);
      else
        j := j + 1;
      end if;
    end if;
  end while;
  swap(X[Left], X[i]);
  swap(X[Right], X[k]);
end
```

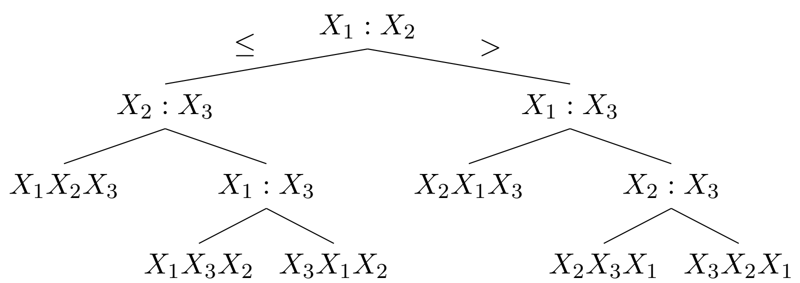
Each iteration of the main (while) loop, taking a constant amount of time, either increments $j$ or decremets $k$ by 1 and hence shortens the distance between $j$ and $k$ by 1. As the initial distance between $j$ and $k$ equals $\texttt{Right} - (\texttt{Left} + 1)$, at most $n-2$ iterations will be executed. It follows that the algorithm is linear-time. □

8. Below is a variant of the insertion sort algorithm.

**Algorithm Insertion_Sort** $(A, n)$;
**begin**
    **for** $i := 2$ **to** $n$ **do**
        $x := A[i]$;
        $j := i$;
        **while** $j > 1$ and $A[j-1] > x$ **do**
            $A[j] := A[j-1]$;
            $j := j - 1$;
        **end while**
        $A[j] := x$;
    **end for**
**end**

Draw a decision tree of the 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.

*Solution.*




□

9. Consider the text data compression problem we have discussed in class; the problem statement is given below.

Given a text (a sequence of characters), find an encoding for the characters that satisfies the prefix constraint and that minimizes the total number of bits needed to encode the text.

Prove that the two characters with the lowest frequencies must be among the deepest leaves (farthest from the root) in the final code tree.

*Solution.* Denote the characters in the text by $c_1$, $c_2$, $\cdots$, $c_n$ and their frequencies by $f_1$, $f_2$, $\cdots$, $f_n$. Given an encoding $E$ in which a bit string $s_i$ represents $c_i$, the length (number of bits) of the text encoded by using $E$ is $\sum_{i=1}^{n} |s_i| \cdot f_i$. In the code tree corresponding to $E$, the depth of the leaf representing character $c_i$ equals the length of the encoding $s_i$ for $c_i$. We observe that at the deepest level in the code there must be at least two leaves; otherwise, we may remove the only leaf and take its parent as a new leaf, obtaining a better code tree.

Assume toward a contradiction that one of the two characters, say $c_j$, with the lowest frequencies is at a level shallower than that of a character, say $c_k$, with a higher frequency such that $|s_j| < |s_k|$. Since $|s_j| < |s_k|$ and $f_j < f_k$, $s_j \cdot f_j + s_k \cdot f_k > s_j \cdot f_k + s_k \cdot f_j$. It follows that

$$\sum_{i=1}^{n} |s_i| \cdot f_i > (\sum_{i=1, i\neq j, i\neq k}^{n} |s_i| \cdot f_i) + s_j \cdot f_k + s_k \cdot f_j.$$

If we swap the characters $c_j$ and $c_k$, then we will get a better code tree, a contradiction. □

10. The *next* table is a precomputed table that plays a critical role in the KMP algorithm. For every position $j$ of the second input string $b_1 b_2 \ldots b_m$ (to be matched against the first input string), the value of $next[j]$ tells the length of the longest proper prefix that is equal to a suffix of $b_1 b_2 \ldots b_{j-1}$; the value of $next[0]$ is set to $-1$ to fit in the KMP algroithm. For each of the following instances of *next*, give a string of letters $a$ and $b$ that gives rise to the table or argue that no string can possibly produce the table.

(a)

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|
| $-1$ | 0 | 0 | 1 | 1 | 2 | 3 | 4 | 5 |

*Solution.* There are a few strings that may produce this *next* table, e.g., *abaabaaba* or *abaabaabb*. □

(b)

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|
| $-1$ | 0 | 1 | 2 | 3 | 5 | 1 | 2 | 3 |

*Solution.* No string can possibly give arise to this *next* table, as the value of $next[6]$ should be no more than 4 (but it is 5 here). □

## Appendix

- Below is 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
```

- Below is an alternative algorithm for partition in the Quicksort algorithm:

```
Partition (X, Left, Right);
begin
    pivot := X[left];
    i := Left;
    for j := Left + 1 to Right do
        if X[j] < pivot then i := i + 1;
                             swap(X[i], X[j]);
    Middle := i;
    swap(X[Left], X[Middle])
end
```