Write a backtracking algorithm to solve:

Given $n$ positive integers $\mathbf{x} = x_1, x_2, \ldots, x_n \in \mathbb{Z}$ and $y \in \mathbb{Z}^+$ does there exist $\mathbf{y} \in \mathbf{x}$: $\sum y_i = y$ for $y_i \in \mathbf{x}$.

Create a binary tree where each level $i$, refers to the weight $w_i$.

Each node has two branches one 0 one 1. 0 means total the weight represented by the child is not included in the solution, 1 means it is.

Traverse the tree in a depth-first manner, back up and by next branch when clear that weights exceed $y$.

Test: let $S$ be the accumulated sum at level $i$.

Since weights are in ascending order, if $S + w_i > y$ then no solution on branch.

$w = (w_1, w_2, \ldots, w_n)$ the weight vector. $w_i < w_i+1$

$y = (y_1, y_2, \ldots, y_n)$ is the solution vector, initialize to $\mathbf{0}$.

$y \in \mathbb{Z}_+^n$. $y \leq \mathbf{x} w_i$

$s = w_1 y_1 + w_2 y_2 + \ldots + w_n y_n$ $k \leq n$

Backtrack will mean to decrement $k$ until $y_k = 1$. Then set $y_k = 0$. If $k = 0$ then finished.

Subset Sum ($w$, $y$)

$y = \mathbf{0}$, $s = 0$, $k = 0$ $n = \text{length}(w)$.

Boolean done = false

while (!done)

$k = k+1$

if ($k > n$)

Call Back track

else
\[ y_k = 1 \]
\[ S = 0 \]
for \( i = 1 \) to \( k \)
\[ S = S + y_i \cdot x_i \quad // \text{current value of } S \]
if \( (S = y) \)
return \( 1 \) \quad // \text{done}
if \( (S > y) \)
\[ \text{call Back track} \]
Back track()
if \( (k \leq n) \)
\[ y_k = 0 \]
\[ k = k - 1 \]
if \( (y_k = 1) \)
\[ y_k = 0 \]
else
\[ \text{while } (k > 0 \land y_k = 0) \]
\[ k = k - 1 \]
if \( (k > 0) \)
\[ y_k = 0 \]
else
done = true.
13/14. a) Give a backtracking algorithm to solve the knapsack problem.

b) Which technique (dynamic / backtracking) is more efficient?

// Suppose we have n items with sizes given by weights w1...wN and
// values given by profit p1...pN. Let y be the capacity of the sack.
// Let: \( \sum w_i < y \) \( \sum p_i y_i \) maximized
// \( y = (y_i) \) is a binary vector.
// use an upper bound on the value of best feasible solution to
// expanding a given node and its descendants. If the upper bound
// is not larger than the value of best solution so far, 
discard node.
// \text{upper}(p,w,k,y) \text{ determines an upper bound on the}
// \text{best solution obtainable by expanding any node } y \text{ at level } k+1.
// \text{upper}(p,w,k,y) = \text{upper}(p,w,k+1) \text{ if } \frac{p_i}{w_i} \geq \frac{p_{i+1}}{w_{i+1}} \text{, } 1 \leq i < n

\text{Bound}(p,w,k,y) \text{ is}
\begin{align*}
p &= \text{current profit total} \\
w &= \text{current weight total} \\
k &= \text{index of last removed item}
\end{align*}

b = p \quad c = w \\
\text{for } i = k+1 \text{ to } n \\
c = c + w_i \\
P(c,y) \\
b = b + p_i \\
\text{else} \\
\text{return } \left[ \frac{b + (1 - (c - y)) \times P(i)}{w_i} \right] \quad \text{ //new profit}
\text{return } b \\
\text{The bound for a feasible left child } y(B) = 1 \text{ for a node } y \text{ is the same as for } y = \text{Bound}(.) \text{ does not need to}
KnapSack (y, n, W, P, realweight, realprofit, s) S

\[ \text{cw} = \text{cp} = 0 \quad \text{//current weight / profit} . \]
\[ k = 1 \]
\[ \text{finalprofit} = 1 \]

\[ \text{while } (k < n \text{ and } \text{cw + W(k) < y}) \]
\[ \text{//place k into knapsack} \]
\[ \text{cw} = \text{cw} + W(k) \]
\[ \text{cp} = \text{cp} + P(k) \]
\[ \text{z}(k) = 1 \]
\[ k = k + 1 \]

\[ \text{if } (k = n) \]
\[ \text{finalprofit} = \text{cp} \]
\[ \text{finalweight} = \text{cw} \]
\[ k = n \]
\[ z = z \]

\[ \text{else} \]
\[ z(k) = 0 \quad \text{//y so object k not in sack} \]

\[ \text{while (Bound(cp, cw, k, y) < finalprofit)} \]
\[ \text{while } (k > 0, z(k) = 1) \]
\[ k = k - 1 \quad \text{//find last included weight} \]
\[ \text{if } k = 0 \]
\[ \text{return} \]
\[ z(k) = 0 \]
\[ \text{cw} = \text{cw} - W(k) \quad \text{//remove kth item} \]
\[ \text{cp} = \text{cp} - P(k) \]
\[ k = k + 1 \]

this algorithm generates a \( 2^n \) binary tree in reversed lexicographic order. it takes \( O(n) \) to check

bounding function at \( O(2^n) \) right children \( \Rightarrow \) time complexity \( O(2^n) \)

for large \( n \), the dynamic approach is better.
Give backtracking algorithm to solve the assignment problem defined as follows. Given \( n \) employees to be assigned to \( n \) jobs such that the cost of assigning the \( i \)th person to the \( j \)th job is \( c_{ij} \), find an assignment that minimizes total cost. Assume that the cost is non-negative \( c_{ij} \geq 0 \) for \( 1 \leq i, j \leq n \).

Variables:
- \( E = \{e_1, e_2, \ldots, e_n\} \) Employees
- \( J = \{j_1, j_2, \ldots, j_n\} \) Jobs
- \( C = \{c_{11}, c_{12}, \ldots, c_{1n}\}
- \{c_{21}, c_{22}, \ldots, c_{2n}\}
- \{c_{n1}, c_{n2}, \ldots, c_{nn}\} \)

Constraints:
- All costs \( c_{ij} \geq 0 \)
- Cost is minimized
- Employee to job is 1 to 1 relationship.

Assignment Problem (cost matrix: \( C \))

```
// Get min matrix
// min rows
for i = 1 to n
  for j = 1 to n
    rowmin = Min(c_{ij})  // Subtract min value from each entry in i
    c_{ij} -= rowmin
    bound += rowmin
  end
end

// min Col
for i = 1 to n
  for j = 1 to n
    colmin = Min(c_{ij})  // Col min value
    c_{ij} -= colmin
    bound += colmin
  end
end

n x n matrix d  // decision matrix
```
Problem 13.17 cost

\[ \text{Colspace} \{1...n\} \]
\[ \text{Rowspace} \{1...n\} \]
\[ \text{Assignment} \ (C, \text{rowspace}, \text{colspace}) \]

Assignment (Matrix M, int R, int C, int index)
\[ \text{col} = \text{FindZero}(R, \text{index}) \] // Find zero in row index, return col index

\[ \text{d} = \text{CalculateBand}(M, C, \text{index}, \text{col}) \]
\[ \text{Takeband} = \text{CalculateBand2}(M, \text{index}, \text{col}) \]

\[ \text{If} \ (\text{takeband} < \text{d} + \text{don'tTakeband}) \]
\[ \text{Remove index from R} \]
\[ \text{Remove col from C} \]
\[ \text{Assignmen}t \ (M, R, C, 1) \]

\[ \text{Else} \]
\[ C, \text{index}, \text{col} = \infty \]
\[ \text{Update matrix M} \]
\[ \text{Does the row/col reduction adds to bound total.} \]
\[ \text{Assignmen}t \ (M, R, C, 1) \]
\[ \text{d} = \text{index}, \text{col} = 0; \]
\[ \text{bound: don'tTakeband} \]

\[ \text{Explanation} \]

1. Find least cost reduction matrix such that every row has a zero, calculate lower bound.
2. Choose the zero in col; find row; analyze lower bound; if C{i} was removed, set to infinity, choose better result, store in decision
3. Repeat step 2 until out of entries
4. Return bound as min cost to have a decision matrix which corresponds to the cost matrix.
A 1 denotes that job i is done by person j in jth row.
Problem 13.19

Apply the branch-and-bound algorithm for the traveling salesman problem discussed in 13.5 on instance...

\[
\begin{bmatrix}
\infty & 3 & 0 & 8 \\
0 & \infty & 5 & 10 \\
6 & 0 & 2 & \infty \\
\end{bmatrix}
\Rightarrow
\begin{bmatrix}
3 & 0 & \infty & 8 \\
0 & 4 & \infty & 2 \\
0 & 0 & \infty & 2 \\
\end{bmatrix}
\]

Reduction matrix lower bound = 11

\[
\begin{bmatrix}
1 & 2 & 3 & 4 \\
2 & 3 & 4 & 5 \\
3 & 4 & 5 & 6 \\
\end{bmatrix}
\Rightarrow
\begin{bmatrix}
1 & 2 & 3 & 4 \\
2 & 3 & 4 & 5 \\
3 & 4 & 5 & 6 \\
\end{bmatrix}
\]

Band = 11

Solution Edges

2 to 1
1 to 3
3 to 4
4 to 2

Cost = 11
**Problem 13.20**

Consider the knapsack problem (see 7.6). Use branch & bound and a suitable lower bound to solve the instance of this problem in Ex. 7.6.

**Capacity: 9**

<table>
<thead>
<tr>
<th>Item</th>
<th>Size</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>7</td>
</tr>
</tbody>
</table>

**Find Value**

<table>
<thead>
<tr>
<th>Value</th>
<th>Item</th>
<th>Value/Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.5</td>
<td>1</td>
<td>1.5</td>
</tr>
<tr>
<td>1.3</td>
<td>2</td>
<td>1.3</td>
</tr>
<tr>
<td>1.25</td>
<td>3</td>
<td>1.25</td>
</tr>
<tr>
<td>1.1</td>
<td>4</td>
<td>1.1</td>
</tr>
</tbody>
</table>

Sort them high/low:

1. 4: 1.5
2. 2: 1.3
3. 3: 1.25

**2 Solutions**

Items 1, 2, 3 or Items 4, 3, both have size 9.

**Items 1, 2, 3**

- Total Value = 0
- Size = 6
- Bound = 12.67

Don't take Item 1.

**Items 2, 3**

- Size = 7
- Bound = 12.67

Take Item 2.

**Items 3**

- Done

**Value = 10**

**Items 4, 3**

- Value = 0
- Size = 0
- Bound = 9

Don't take Item 4.

**Items 4**

- Size = 5
- Bound = 12

Take Item 4.

**Value = 7**

- Size = 2
- Bound = 9

Don't take Item 2.

**Value = 10**

- Size = 5
- Bound = 12

Take Item 2.

**Value = 7**

- Size = 2
- Bound = 8

Don't take Item 3.

**Value = 12**

- Size = 9
- Bound = 12

Don't take Item 3.

**Value = 12**

- Size = 5
- Bound = 7

Take Item 3.

**Value = 7**

- Size = 5
- Bound = 7

Take Item 3.

* Done Value = 12 + 7 = 19
Problem 13.21

Carry out a branch-and-bound procedure to solve the following instance of the assignment problem (Exercise 13.17). There are 4 employees, 4 jobs. The cost function is represented by the matrix below. Row i corresponds to the i-th employee, column j corresponds to the j-th job.

\[
\begin{bmatrix}
3 & 5 & 2 & 4 \\
6 & 7 & 5 & 3 \\
3 & 7 & 4 & 5 \\
8 & 5 & 4 & 0
\end{bmatrix}
\]

\[
\begin{bmatrix}
1 & 3 & 0 & 2 \\
3 & 4 & 2 & 0 \\
0 & 4 & 1 & 2 \\
4 & 1 & 0 & 2
\end{bmatrix}
\]

\[
\begin{bmatrix}
1 & 2 & 0 & 2 \\
3 & 3 & 2 & 0 \\
0 & 3 & 1 & 2 \\
4 & 0 & 0 & 2
\end{bmatrix}
\]

\[
\begin{bmatrix}
1 & 2 & 3 & 4 \\
2 & 1 & 3 & 0 \\
3 & 3 & 2 & 2 \\
4 & 0 & 2 & 2
\end{bmatrix}
\]

\[
\begin{bmatrix}
1 & 2 & 3 & 4 \\
2 & 1 & 3 & 0 \\
3 & 3 & 2 & 2 \\
4 & 0 & 2 & 2
\end{bmatrix}
\]

\[
\begin{bmatrix}
1 & 2 & 3 & 4 \\
2 & 1 & 3 & 0 \\
3 & 3 & 2 & 2 \\
4 & 0 & 2 & 2
\end{bmatrix}
\]

Total cost = 13

<table>
<thead>
<tr>
<th>Employee</th>
<th>Job</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
</tr>
</tbody>
</table>