knapsack_report

December 29, 2024

Course	Combinatorial Algorithms
Semester	**2024 Winter**
Assignment	**01**
Group	**03**
Member 01	**999014681 Mingshan, LI**
Member 02	**999014772 Shunxi, XIAO**
Member 03	**999022064 Weizhi, LU**

1 Q1. Knapsack: Bounding Functions and Branch and Bound

1. Implement in Python the algorithm that makes use of the fractional knapsack as a bounding function to further prune the decision tree of the 01-knapsack.
2. Moreover, using the same bounding function, implement the branch and bound strategy for the 01-knapsack.
3. Provide test cases to ensure the correctness of your programs.
4. Report on the comparison of the running times of the backtracking, the bounding, and the branch and bound implementations.

[1]:
```
import time
import random
from numpy import dot
import matplotlib.pyplot as plt
```

1.1 Part 1. Knapsack - General Backtracking

As the starting point of the implementation of other variants backtracking algorithms, it is reasonable to implement the general backtracking solution of 01-knapsack problem at the beginning.

From the materials and refeernces of this coouse, we have the following psuedo code of backtracking algorithm to solve 01-knapsack problem

[2]:
```python
def knapsack_general(values: list, weights: list, capacity: int) -> list:
    '''
    The general backtracking algorithms solving 01-knapsack problem.

    Argumetns:
        - values:   the list of values of items
        - weights:  the list of weights of items
        - capacity: the capacity of knapsack
    Return:
        - optX:     the optimal solution
    '''

    # global variable

    optP = 0          # optimal profit   of 01-knapsack problem
    optX = []         # optimal solution of 01-knapsack problem
    N = len(values) # number of items

    # recursive part

    def knapsack_general_recursive( currX: list = [] ) -> None:
        '''
        The recursive part of general backtracking algorithms solving␣
↪01-knapsack problem.

        Argumetns:
            - currX: current solution
        '''
        nonlocal optP, optX, N

        '''
        Step 1: Check feasibility of current solution {currX}
        '''
        if len(currX) == N:
```

```
            currW = dot(weights, currX) # current weight of current solution␣
↪{currX}
            currP = dot(values, currX)  # current profit of current solution␣
↪{currX}

            # Check whether current solution {currX} is better

            if currW <= capacity and currP > optP:

                optP = currP
                optX = currX[:]

        else:

            '''
            Step 2: Construct the choice set for current solutiuon {currX}
            '''
            choS = [0, 1]

            '''
            Step 3: For each possible next solution, call the algorithm␣
↪recursively
            '''
            for x in choS:

                knapsack_general_recursive( currX + [x] )

    knapsack_general_recursive( [] )

    return optX
```

Next, we check the correctness of the general backtracking algorithm `knapsack_general` implemented above.

[3]:
```
Capacity = 5
Weights  = [4, 3, 7]
Values   = [1, 2, 3]
Solution = [0, 1, 0]

optX = knapsack_general( Values, Weights, Capacity )

if optX == Solution:
    print("True")
else:
    print("False")
```

```
True
```

Additionally, we can modify `knapsack_general`, to implement a variant of backtracking algorithm with a simple pruning method.

[4]:
```
def knapsack_pruning(values: list, weights: list, capacity: int) -> list:
    '''
    A backtracking algorithms solving 01-knapsack problem with
    a simple pruning method.

    Argumetns:
        - values:   the list of values of items
        - weights:  the list of weights of items
        - capacity: the capacity of knapsack
    Return:
        - optX:     the optimal solution
    '''

    # global variable

    optP = 0          # optimal profit   of 01-knapsack problem
    optX = []         # optimal solution of 01-knapsack problem
    N = len(values) # number of items

    # recursive part

    def knapsack_pruning_recursive( currX: list = [] ) -> None:
        '''
        The recursive part of knapsack_pruning.

        Argumetns:
            - currX:   current solution
        '''
        nonlocal optP, optX, N
        currl = len(currX)
        currX_ = currX + [0] * (N - currl)
        currW = dot(weights, currX_) # current weight of current solution␣
 ↪{currX}

        '''
        Step 1: Check feasibility of current solution {currX}
        '''
        if len(currX) == N:

            currP = dot(values, currX)  # current profit of current solution␣
 ↪{currX}

            # Check whether current solution {currX} is better
```

```
            if currW <= capacity and currP > optP:

                optP = currP
                optX = currX[:]

        else:

            '''
            Step 2: Construct the choice set for current solutiuon {currX}, do␣
↪pruning
            '''
            if currW + weights[currI] <= capacity:
                choS = [0, 1]
            else:
                choS = [0]

            '''
            Step 3: For each possible next solution, call the algorithm␣
↪recursively
            '''
            for x in choS:

                knapsack_pruning_recursive( currX + [x] )

    knapsack_pruning_recursive( [] )

    return optX
```

Checking the correctness of `knapsack_pruning`.

[5]:
```
Capacity = 5
Weights  = [4, 3, 7]
Values   = [1, 2, 3]
Solution = [0, 1, 0]

optX = knapsack_pruning( Values, Weights, Capacity )

if optX == Solution:
    print("True")
else:
    print("False")
```

```
True
```

1.2 Part 2. Test Cases

File `p1_knapsack_test_cases` is responsible to store all test cases that we are going to run later.

After checking the correctness of `knapsack_general` and `knapsack_pruning`, for later usage, it would be convenient if we implement a test cases generator first, with `knapsack_pruning` generating the solution of each cese.

[6]:
```python
def knapsack_generate_test_cases(
        fname: str,
        tnum: int,
        step: int,
        maxRate: float,
        minRate: float,
        maxValue: int ) -> None:
    '''
    Generate test cases for 01-knapsack problem in file {fname}.

    Arguments:
        - fname:    the name of file that stores all the test cases.
        - tnum:     the number of test cases to generate.
        - step:     the step of the size of test cases.
        - maxRate:  maximum rate of weight / capacity.
        - minRate:  minimum rate of weight / capacity.
        - maxValue: maximum of item value.
    '''

    file = open(fname, 'w')
    count = 1

    while count <= tnum:

        test_case = '' # test case

        '''
        Step 1. Generate the size of test case
        '''
        test_size = step * count # the size of test case

        '''
        Step 2. Generate the capacity of knapsack
        '''
        test_capacity = test_size * step
        test_case += str(test_capacity) + '#'

        '''
        Step 3. Generate the values and weights of all items
        '''
        test_weights = ''
        test_values = ''
        for i in range(test_size):
```

```
            weight = random.randint(
                int(minRate * test_capacity / test_size),
                int(maxRate * test_capacity / test_size)
            )

            value = random.randint(1, maxValue)

            if i == test_size - 1:
                test_weights += str(weight)
                test_values += str(value)
            else:
                test_weights += str(weight) + ' '
                test_values += str(value) + ' '
        test_case += test_values  + '#'
        test_case += test_weights + '#'

        '''
        Step 4. Generate the solution with knapsack_general
        '''
        values = list(map(int, test_values.split()))
        weights = list(map(int, test_weights.split()))
        solution = knapsack_pruning( values, weights, test_capacity )
        test_solution = ' '.join(map(str, solution))
        test_case += test_solution

        '''
        Step 5. Write the test_case into the file {fname}
        '''
        print(test_case)
        if count != tnum:
            test_case += '\n'
        file.write(test_case)

        count += 1

    file.close()
```

Then, we can apply this function to generate some test cases.

[7]:

```
testFile = 'knapsack_test_cases'
num = 5 # Number of test cases
step = 4
maxRate = 3.5
minRate = 0.8
maxValue = 20
print('All tests generated:\n')
knapsack_generate_test_cases( testFile, num, step, maxRate, minRate, maxValue )
```

```
All tests generated:

16#8 6 17 5#3 8 3 7#1 1 1 0
32#4 7 4 13 12 13 11 9#7 8 7 6 8 13 7 12#0 1 0 1 1 0 1 0
48#10 12 10 5 5 14 7 2 18 2 7 20#12 9 5 8 13 11 6 3 12 3 6 13#0 0 1 0 0 1 0 0 1
0 1 1
64#1 15 11 12 1 4 2 5 14 17 14 4 9 4 14 1#10 14 8 6 8 4 7 3 10 7 13 7 4 11 14
5#0 1 1 1 0 0 0 0 1 1 0 0 1 0 1 0
80#7 17 15 12 2 18 10 5 6 12 17 14 3 17 20 7 15 3 6 2#7 14 5 10 10 4 9 12 11 5 8
10 4 14 12 11 8 13 5 3#0 1 1 0 0 1 0 0 0 1 1 1 0 1 1 0 1 0 0 0
```

1.3 Part 3. Knapsack - Fractional Knapsack as a Bounding Function

Thanks to the materials of the course, we already have the implemetation of fraction knapsack as follow.

[8]:
```python
def fractional(v, w, W) -> list:
    """
    the fractional knapsack

    Arguments:
        - v: the list of values
        - w: the list of weights
        - W: the capacity

    Return:
        - x: optimal fractional solution
    """

    s, v, w = sort(v, w)

    x, c, i = [0]*len(v), W, 0

    while 0 < c and i < len(v):

        x[i] = 1 if w[i] <= c else c/w[i]
        c   -= w[i] * x[i]
        i   += 1

    x = restore(s, x)

    return x

def sort(v, w):
    """
    sort the vectors of values and weights
    by value/weight ratio in decreasing order
    """
```

```python
    z = list(zip(range(len(v)),zip(v, w)))

    z.sort(key=lambda k: (k[1][0]/k[1][1]), reverse=True)

    s, z = zip(*z)

    return s, *map(list,zip(*z))

def restore(s, x):
    """
    in conjunction with sort restores the
    solution x its original order of elements
    """

    z = list(zip(s, x))

    z.sort()

    z, r = map(list,zip(*z))

    return r
```

Thus, we are able to implement a bounding function with the help of the funcitons mentioned above.

[9]:
```python
def getBound(values: list, weights: list, capacity: int, currX: list, algo) ->␣
↪float:
    '''
    Calculate the bound of profit of current solution {currX}.

    Arguments:
        - values:   list of item values
        - weights:  list of item weights
        - capacity: capacity of knapsack
        - currX:    current solution
        - algo:     algorithnm used to calculate the bound
    Return:
        - currP + optP_rX: the bound of the profit for currX
    '''
    N = len(values)
    currl = len(currX)
    currX_ = currX + [0] * (N - currl)
    currP = dot(values, currX_)  # current profit of current solution {currX}
    currW = dot(weights, currX_) # current weight of current solution {currX}
    opt_rX = [] if N == currl else fractional( values[currl:], weights[currl:],␣
↪capacity - currW )
```

```
    optP_rX = 0 if N == currl else dot( values[currl:], opt_rX )
    return currP + optP_rX
```

Next, with the help of the fractional knapsack as a bounding function, we are able to implement the bounding version of backtracking algorithm solving 01-knapsack problem.

[10]:
```
def knapsack_bounding(values: list, weights: list, capacity: int) -> list:
    '''
    The backtracking algorithms solving 01-knapsack problem that makes the
    use of the fractional knapsack as a bounding function.

    Argumetns:
        - values:   the list of values of items
        - weights:  the list of weights of items
        - capacity: the capacity of knapsack
    Return:
        - optX:     the optimal solution
    '''

    # global variable

    optP = 0         # optimal profit   of 01-knapsack problem
    optX = []        # optimal solution of 01-knapsack problem
    N = len(values) # number of items

    # recursive part

    def knapsack_bounding_recursive( currX: list = [] ) -> None:
        '''
        The recursive part of knapsack_fkBound.

        Argumetns:
            - currX:   current solution
        '''
        nonlocal optP, optX, N
        currl = len(currX)
        currX_ = currX + [0] * (N - currl)
        currW = dot(weights, currX_) # current weight of current solution␣
↪{currX}
        currP = dot(values, currX_)  # current profit of current solution␣
↪{currX}

        '''
        Step 1: Check feasibility of current solution {currX}
        '''
        if len(currX) == N:
```

```
            # Check whether current solution {currX} is better

            if currW <= capacity and currP > optP:

                optP = currP
                optX = currX[:]

        else:

            '''
            Step 2: Calculate the bound of the current solution {currX}, do␣
↪boundingly pruning
            '''
            bound = getBound( values, weights, capacity, currX, fractional ) #␣
↪bound for the profit of currX
            if bound <= optP: return # boundingly pruning

            '''
            Step 3: Construct the choice set for current solutiuon {currX}, do␣
↪pruning
            '''
            if currW + weights[currI] <= capacity:
                choS = [0, 1]
            else:
                choS = [0]

            '''
            Step 4: For each possible next solution, call the algorithm␣
↪recursively
            '''
            for x in choS:

                knapsack_bounding_recursive( currX + [x] )

    knapsack_bounding_recursive( [] )

    return optX
```

Check the correctness of `knapsack_fkBound` implemented above, using the test cases generated at Part 2.

To begin with, it is necessary for us to implement a test cases builder.

[11]:
```
def build_tests(fname: list) -> list:
    '''
    Return a list consisting of all test cases in file {fname}.
    '''
```

```
    file  = open(fname, "r")
    lines = file.read().split("\n")
    file.close()

    tests = []

    for line in lines:

        test = line.split("#")

        W = int(test[0])                          # capacity of knapsack
        v = list(map(int, test[1].split())) # values of items
        w = list(map(int, test[2].split())) # weights of items
        s = list(map(int, test[3].split())) # solution of test case

        assert len(v) == len(w) and len(w) == len(s)

        tests += [(W,v,w,s)]

    return tests

def knapsack_run_test_cases( fname: str, algo ) -> None:
    '''
    Run all the test cases in file {fname} with the algorithm to test {algo}.

    Arguments:
        - fanme: the name of the file that stores all the test cases
        - algo:  the algorithm that we are going to test with
    '''

    count = 0
    tests = build_tests(fname)

    for test in tests:

        W, v, w, expectedSol = test

        '''
        W: knapsack capcity
        v: item values
        w: item weights
        expectedSol: solution
        '''

        ourSol = algo(v, w, W)

        expectedProfit = dot( v, expectedSol )
```

```
        ourProfit      = dot( v, ourSol)

        flag = True if expectedSol == ourSol or expectedProfit == ourProfit␣
↪else False

        print(
            f'Test No:  {count+1:02d}',
            f'items:    {len(v):02d}',
            f'knapsack( {v}, {w}, {W} ) = {ourSol}',
            f'solution: {expectedSol}',
            f'result:   {flag}',
            sep = '\n',
            end = '\n\n'
        )

        count += 1
```

Now, we are able to run all the test cases to check the correctness of `knapsack_fkBounding`.

[12]:
```
knapsack_run_test_cases(testFile, knapsack_bounding)
```

```
Test No:  01
items:    04
knapsack( [8, 6, 17, 5], [3, 8, 3, 7], 16 ) = [1, 1, 1, 0]
solution: [1, 1, 1, 0]
result:   True

Test No:  02
items:    08
knapsack( [4, 7, 4, 13, 12, 13, 11, 9], [7, 8, 7, 6, 8, 13, 7, 12], 32 ) = [0,
1, 0, 1, 1, 0, 1, 0]
solution: [0, 1, 0, 1, 1, 0, 1, 0]
result:   True

Test No:  03
items:    12
knapsack( [10, 12, 10, 5, 5, 14, 7, 2, 18, 2, 7, 20], [12, 9, 5, 8, 13, 11, 6,
3, 12, 3, 6, 13], 48 ) = [0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 1, 1]
solution: [0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 1, 1]
result:   True

Test No:  04
items:    16
knapsack( [1, 15, 11, 12, 1, 4, 2, 5, 14, 17, 14, 4, 9, 4, 14, 1], [10, 14, 8,
6, 8, 4, 7, 3, 10, 7, 13, 7, 4, 11, 14, 5], 64 ) = [0, 1, 1, 1, 0, 0, 0, 0, 1,
1, 0, 0, 1, 0, 1, 0]
solution: [0, 1, 1, 1, 0, 0, 0, 0, 1, 1, 0, 0, 1, 0, 1, 0]
result:   True
```

```
Test No:  05
items:    20
knapsack( [7, 17, 15, 12, 2, 18, 10, 5, 6, 12, 17, 14, 3, 17, 20, 7, 15, 3, 6,
2], [7, 14, 5, 10, 10, 4, 9, 12, 11, 5, 8, 10, 4, 14, 12, 11, 8, 13, 5, 3], 80 )
= [0, 1, 1, 0, 0, 1, 0, 0, 0, 1, 1, 1, 0, 1, 1, 0, 1, 0, 0, 0]
solution: [0, 1, 1, 0, 0, 1, 0, 0, 0, 1, 1, 1, 0, 1, 1, 0, 1, 0, 0, 0]
result:   True
```

1.4 Part 4. Knapsack - Branch and Bound Strategy

Based on the implementation of `knapsack_bounding`, with the idea of greedy strategy, we can now implement the branh-and-bound version of backracking algorithm.

[13]:
```python
def knapsack_branchAndBound(values: list, weights: list, capacity: int) -> list:
    '''
    The backtracking algorithms solving 01-knapsack problem that makes the
    use of the fractional knapsack as a bounding function.

    Argumetns:
        - values:   the list of values of items
        - weights:  the list of weights of items
        - capacity: the capacity of knapsack
    Return:
        - optX:     the optimal solution
    '''

    # global variable

    optP = 0         # optimal profit   of 01-knapsack problem
    optX = []        # optimal solution of 01-knapsack problem
    N = len(values) # number of items

    # recursive part

    def knapsack_branchAndBound_recursive( currX: list = [] ) -> None:
        '''
        The recursive part of knapsack_fkBound.

        Argumetns:
            - currX:   current solution
        '''
        nonlocal optP, optX, N
        currl = len(currX)
        currX_ = currX + [0] * (N - currl)
        currW = dot(weights, currX_) # current weight of current solution␣
↪{currX}
```

```
        currP = dot(values, currX_)  # current profit of current solution␣
↪{currX}

        '''
        Step 1: Check feasibility of current solution {currX}
        '''
        if len(currX) == N:

            # Check whether current solution {currX} is better

            if currW <= capacity and currP > optP:

                optP = currP
                optX = currX[:]

        else:

            '''
            Step 2: Construct the choice set for current solutiuon {currX}
            '''
            if currW + weights[currl] <= capacity: # simple pruning
                choS = [0, 1]
            else:
                choS = [0]

            '''
            Step 3: Find the next solution with higher possible value (greedy␣
↪strategy)
            '''
            nextChoices = []
            nextBounds  = []

            for i in range( len(choS) ):

                nextChoices.append( currX[:] + [choS[i]] )
                nextBound = getBound( values, weights, capacity, currX +␣
↪[choS[i]], fractional)
                nextBounds.append( nextBound )

            # Sort nextChoices and nextBounds so that nextBounds is in␣
↪decreasing order.
            if len(choS) == 2 and nextBounds[0] < nextBounds[1]:

                nextBounds[0],  nextBounds[1]  = nextBounds[1],  nextBounds[0]
                nextChoices[0], nextChoices[1] = nextChoices[1][:],␣
↪nextChoices[0][:]
```

```python
            if nextBounds[0] <= optP: return

            '''
            Step 4: For each possible next solution, call the algorithm␣
↪recursively
            '''
            for i in range( len(nextChoices) ):

                knapsack_branchAndBound_recursive( nextChoices[i] )

    knapsack_branchAndBound_recursive( [] )

    return optX
```

Now, we check the correctness of `knapsack_branchAndBound`.

[14]:
```python
knapsack_run_test_cases(testFile, knapsack_branchAndBound)
```

```
Test No:  01
items:    04
knapsack( [8, 6, 17, 5], [3, 8, 3, 7], 16 ) = [1, 1, 1, 0]
solution: [1, 1, 1, 0]
result:   True

Test No:  02
items:    08
knapsack( [4, 7, 4, 13, 12, 13, 11, 9], [7, 8, 7, 6, 8, 13, 7, 12], 32 ) = [0,
1, 0, 1, 1, 0, 1, 0]
solution: [0, 1, 0, 1, 1, 0, 1, 0]
result:   True

Test No:  03
items:    12
knapsack( [10, 12, 10, 5, 5, 14, 7, 2, 18, 2, 7, 20], [12, 9, 5, 8, 13, 11, 6,
3, 12, 3, 6, 13], 48 ) = [0, 1, 1, 0, 0, 0, 1, 0, 1, 1, 0, 1]
solution: [0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 1, 1]
result:   True

Test No:  04
items:    16
knapsack( [1, 15, 11, 12, 1, 4, 2, 5, 14, 17, 14, 4, 9, 4, 14, 1], [10, 14, 8,
6, 8, 4, 7, 3, 10, 7, 13, 7, 4, 11, 14, 5], 64 ) = [0, 1, 1, 1, 0, 0, 0, 0, 1,
1, 1, 0, 1, 0, 0, 0]
solution: [0, 1, 1, 1, 0, 0, 0, 0, 1, 1, 0, 0, 1, 0, 1, 0]
result:   True

Test No:  05
items:    20
```

```
knapsack( [7, 17, 15, 12, 2, 18, 10, 5, 6, 12, 17, 14, 3, 17, 20, 7, 15, 3, 6,
2], [7, 14, 5, 10, 10, 4, 9, 12, 11, 5, 8, 10, 4, 14, 12, 11, 8, 13, 5, 3], 80 )
= [0, 1, 1, 0, 0, 1, 0, 0, 0, 1, 1, 1, 0, 1, 1, 0, 1, 0, 0, 0]
solution: [0, 1, 1, 0, 0, 1, 0, 0, 0, 1, 1, 1, 0, 1, 1, 0, 1, 0, 0, 0]
result:   True
```

1.5 Part 5. Comparison of Running Times

Finally, we compare the running times of the following three variants of backtracking algorithms that solves 01-knapsack problem. - Backtracking: General - Backtracking: Pruning - Backtracking: Bounding - Backtracking: Branch-and-Bound

We implement the following function to make a comparison of all variants of backtracking algorithms.

To better visualize the comparision, we can use matplot to draw a graph.

[15]:
```python
def compare_knapsack_algos( fname: str, algos: list, names: list, if_plt: bool)␣
↪-> None:

    count = 0
    tests = build_tests( fname )
    item_numbers  = []
    running_times = []

    for test in tests:

        capacity, values, weights, sol_expected = test

        print('------------------ ' + f'Test No:{count+1}' + '␣
↪------------------\n')
        print(f'items:    {len(values):02d}')
        print(f'values:   {values}')
        print(f'weights:  {weights}')
        print(f'solution: {sol_expected}\n')

        item_numbers.append( len(values) )
        item_running_times = []

        for i in range(len(algos)):

            startT    = time.process_time()
            sol_algo  = algos[i](values, weights, capacity)
            endT      = time.process_time()
            elapT     = endT - startT

            item_running_times.append( elapT )
```

```
            optP_expected = dot(values, sol_expected)
            optP_algo     = dot(values, sol_algo)
            flag          = True if optP_expected == optP_algo else False

            print(
                f'algorithm:    {names[i]}',
                f'runningtime:  {elapT:.10f}',
                f'correctness:  {flag}',
                f'knapsack({values},{weights},{capacity}) = {sol_algo}',
                sep = '\n',
                end = '\n\n'
            )

        running_times.append( item_running_times )

        count += 1

    # Plotting the results
    if if_plt:

        running_times = list(zip(*running_times))  # Transpose for easier␣
↪plotting
        plt.figure(figsize=(10, 6))
        for i in range(len(algos)):
            plt.plot(item_numbers, running_times[i], label=names[i], marker='o')

        plt.xlabel('Item Number')
        plt.ylabel('Running Time (seconds)')
        plt.title('Running Time Comparison of Knapsack Algorithms')
        plt.legend()
        plt.grid(True)
        plt.tight_layout()
        plt.show()
```

To better visualize the comparision, we can use matplot to draw a graph.

[16]:
```
algos = [
    knapsack_general,
    knapsack_pruning,
    knapsack_bounding,
    knapsack_branchAndBound
]

algoNames = [
    'Backtracking-General',
    'Backtracking-Pruning',
    'Backtracking-Bounding',
```

```python
    'Backtracking-BranchAndBound'
]

compare_knapsack_algos( testFile, algos, algoNames, True )
```

```
------------------ Test No:1 ------------------

items:    04
values:   [8, 6, 17, 5]
weights:  [3, 8, 3, 7]
solution: [1, 1, 1, 0]

algorithm:    Backtracking-General
runningtime:  0.0004780000
correctness:  True
knapsack([8, 6, 17, 5],[3, 8, 3, 7],16) = [1, 1, 1, 0]

algorithm:    Backtracking-Pruning
runningtime:  0.0003900000
correctness:  True
knapsack([8, 6, 17, 5],[3, 8, 3, 7],16) = [1, 1, 1, 0]

algorithm:    Backtracking-Bounding
runningtime:  0.0007120000
correctness:  True
knapsack([8, 6, 17, 5],[3, 8, 3, 7],16) = [1, 1, 1, 0]

algorithm:    Backtracking-BranchAndBound
runningtime:  0.0004220000
correctness:  True
knapsack([8, 6, 17, 5],[3, 8, 3, 7],16) = [1, 1, 1, 0]

------------------ Test No:2 ------------------

items:    08
values:   [4, 7, 4, 13, 12, 13, 11, 9]
weights:  [7, 8, 7, 6, 8, 13, 7, 12]
solution: [0, 1, 0, 1, 1, 0, 1, 0]

algorithm:    Backtracking-General
runningtime:  0.0021730000
correctness:  True
knapsack([4, 7, 4, 13, 12, 13, 11, 9],[7, 8, 7, 6, 8, 13, 7, 12],32) = [0, 1, 0,
1, 1, 0, 1, 0]

algorithm:    Backtracking-Pruning
runningtime:  0.0008220000
correctness:  True
```

```
knapsack([4, 7, 4, 13, 12, 13, 11, 9],[7, 8, 7, 6, 8, 13, 7, 12],32) = [0, 1, 0,
1, 1, 0, 1, 0]

algorithm:    Backtracking-Bounding
runningtime:  0.0009580000
correctness:  True
knapsack([4, 7, 4, 13, 12, 13, 11, 9],[7, 8, 7, 6, 8, 13, 7, 12],32) = [0, 1, 0,
1, 1, 0, 1, 0]

algorithm:    Backtracking-BranchAndBound
runningtime:  0.0010900000
correctness:  True
knapsack([4, 7, 4, 13, 12, 13, 11, 9],[7, 8, 7, 6, 8, 13, 7, 12],32) = [0, 1, 0,
1, 1, 0, 1, 0]

------------------ Test No:3 ------------------

items:    12
values:   [10, 12, 10, 5, 5, 14, 7, 2, 18, 2, 7, 20]
weights:  [12, 9, 5, 8, 13, 11, 6, 3, 12, 3, 6, 13]
solution: [0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 1, 1]

algorithm:    Backtracking-General
runningtime:  0.0174230000
correctness:  True
knapsack([10, 12, 10, 5, 5, 14, 7, 2, 18, 2, 7, 20],[12, 9, 5, 8, 13, 11, 6, 3,
12, 3, 6, 13],48) = [0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 1, 1]

algorithm:    Backtracking-Pruning
runningtime:  0.0144520000
correctness:  True
knapsack([10, 12, 10, 5, 5, 14, 7, 2, 18, 2, 7, 20],[12, 9, 5, 8, 13, 11, 6, 3,
12, 3, 6, 13],48) = [0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 1, 1]

algorithm:    Backtracking-Bounding
runningtime:  0.0020580000
correctness:  True
knapsack([10, 12, 10, 5, 5, 14, 7, 2, 18, 2, 7, 20],[12, 9, 5, 8, 13, 11, 6, 3,
12, 3, 6, 13],48) = [0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 1, 1]

algorithm:    Backtracking-BranchAndBound
runningtime:  0.0020490000
correctness:  True
knapsack([10, 12, 10, 5, 5, 14, 7, 2, 18, 2, 7, 20],[12, 9, 5, 8, 13, 11, 6, 3,
12, 3, 6, 13],48) = [0, 1, 1, 0, 0, 0, 1, 0, 1, 1, 0, 1]

------------------ Test No:4 ------------------
```

```
items:    16
values:   [1, 15, 11, 12, 1, 4, 2, 5, 14, 17, 14, 4, 9, 4, 14, 1]
weights:  [10, 14, 8, 6, 8, 4, 7, 3, 10, 7, 13, 7, 4, 11, 14, 5]
solution: [0, 1, 1, 1, 0, 0, 0, 0, 1, 1, 0, 0, 1, 0, 1, 0]

algorithm:    Backtracking-General
runningtime:  0.3504470000
correctness:  True
knapsack([1, 15, 11, 12, 1, 4, 2, 5, 14, 17, 14, 4, 9, 4, 14, 1],[10, 14, 8, 6,
8, 4, 7, 3, 10, 7, 13, 7, 4, 11, 14, 5],64) = [0, 1, 1, 1, 0, 0, 0, 0, 1, 1, 0,
0, 1, 0, 1, 0]

algorithm:    Backtracking-Pruning
runningtime:  0.2946180000
correctness:  True
knapsack([1, 15, 11, 12, 1, 4, 2, 5, 14, 17, 14, 4, 9, 4, 14, 1],[10, 14, 8, 6,
8, 4, 7, 3, 10, 7, 13, 7, 4, 11, 14, 5],64) = [0, 1, 1, 1, 0, 0, 0, 0, 1, 1, 0,
0, 1, 0, 1, 0]

algorithm:    Backtracking-Bounding
runningtime:  0.0092570000
correctness:  True
knapsack([1, 15, 11, 12, 1, 4, 2, 5, 14, 17, 14, 4, 9, 4, 14, 1],[10, 14, 8, 6,
8, 4, 7, 3, 10, 7, 13, 7, 4, 11, 14, 5],64) = [0, 1, 1, 1, 0, 0, 0, 0, 1, 1, 0,
0, 1, 0, 1, 0]

algorithm:    Backtracking-BranchAndBound
runningtime:  0.0066440000
correctness:  True
knapsack([1, 15, 11, 12, 1, 4, 2, 5, 14, 17, 14, 4, 9, 4, 14, 1],[10, 14, 8, 6,
8, 4, 7, 3, 10, 7, 13, 7, 4, 11, 14, 5],64) = [0, 1, 1, 1, 0, 0, 0, 0, 1, 1, 1,
0, 1, 0, 0, 0]

------------------ Test No:5 ------------------

items:    20
values:   [7, 17, 15, 12, 2, 18, 10, 5, 6, 12, 17, 14, 3, 17, 20, 7, 15, 3, 6,
2]
weights:  [7, 14, 5, 10, 10, 4, 9, 12, 11, 5, 8, 10, 4, 14, 12, 11, 8, 13, 5, 3]
solution: [0, 1, 1, 0, 0, 1, 0, 0, 0, 1, 1, 1, 0, 1, 1, 0, 1, 0, 0, 0]

algorithm:    Backtracking-General
runningtime:  5.9054850000
correctness:  True
knapsack([7, 17, 15, 12, 2, 18, 10, 5, 6, 12, 17, 14, 3, 17, 20, 7, 15, 3, 6,
2],[7, 14, 5, 10, 10, 4, 9, 12, 11, 5, 8, 10, 4, 14, 12, 11, 8, 13, 5, 3],80) =
[0, 1, 1, 0, 0, 1, 0, 0, 0, 1, 1, 1, 0, 1, 1, 0, 1, 0, 0, 0]
```

```
algorithm:     Backtracking-Pruning
runningtime:   3.7693390000
correctness:   True
knapsack([7, 17, 15, 12, 2, 18, 10, 5, 6, 12, 17, 14, 3, 17, 20, 7, 15, 3, 6,
2],[7, 14, 5, 10, 10, 4, 9, 12, 11, 5, 8, 10, 4, 14, 12, 11, 8, 13, 5, 3],80) =
[0, 1, 1, 0, 0, 1, 0, 0, 0, 1, 1, 1, 0, 1, 1, 0, 1, 0, 0, 0]

algorithm:     Backtracking-Bounding
runningtime:   0.0127860000
correctness:   True
knapsack([7, 17, 15, 12, 2, 18, 10, 5, 6, 12, 17, 14, 3, 17, 20, 7, 15, 3, 6,
2],[7, 14, 5, 10, 10, 4, 9, 12, 11, 5, 8, 10, 4, 14, 12, 11, 8, 13, 5, 3],80) =
[0, 1, 1, 0, 0, 1, 0, 0, 0, 1, 1, 1, 0, 1, 1, 0, 1, 0, 0, 0]

algorithm:     Backtracking-BranchAndBound
runningtime:   0.0014010000
correctness:   True
knapsack([7, 17, 15, 12, 2, 18, 10, 5, 6, 12, 17, 14, 3, 17, 20, 7, 15, 3, 6,
2],[7, 14, 5, 10, 10, 4, 9, 12, 11, 5, 8, 10, 4, 14, 12, 11, 8, 13, 5, 3],80) =
[0, 1, 1, 0, 0, 1, 0, 0, 0, 1, 1, 1, 0, 1, 1, 0, 1, 0, 0, 0]
```