knapsack_report_final

December 30, 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]:
```python
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. The general algorithm is to explore all possible subsets of items, calculating the total value and weight for each subset, and updating the optimal solution if the current subset's weight is less than or equal to the knapsack's capacity and its value exceeds the previous optimal value.

Here are the psuedo code and its implementation of backtracking algorithm to solve 01-knapsack problem:

def $GeneralKanpsack(list)$**:**

global variables: X,optP,optX

if $len(list) = n$ **then**

if $\sum_{i=0}^{n-1} w_i x_i \leq M$ **then**

if $\sum_{i=0}^{n-1} p_i x_i \geq optP$ **then**

$optP \leftarrow (\sum_{i=0}^{n-1} p_i x_i)$

$optX \leftarrow [x_0, ..., x_{n-1}]$

else

$GeneralKanpsack(list + [1])$

$GeneralKanpsack(list + [0])$

[2]:
```
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. The pruning method here works by avoiding branches of the search tree where the current weight exceeds the knapsack's capacity. Before making a choice for the current item, the algorithm checks if adding the current item's weight ($weights[currl]$) would exceed the capacity of the knapsack. If it does, the algorithm will only explore the possibility of excluding the item ($choS = [0]$).

def $Pruning(list, curW)$**:**
  global variables: X,optP,optX
  if $len(list) = n$ **then**
    if $\sum_{i=0}^{n-1} w_i x_i \leq M \quad and \quad \sum_{i=0}^{n-1} p_i x_i > optP$ **then**
      $optP \leftarrow (\sum_{i=0}^{n-1} p_i x_i)$
      $optX \leftarrow [x_0, ..., x_{n-1}]$
  else
    if $curW + w_{cur(l)} \leq M$ **then**
      $C_l \leftarrow \{1, 0\}$
    else
      $C_l \leftarrow \{0\}$
    for x in C$_l$: **Pruning(list+[x], curW+w**$_{cur(l)} x$)
  return optX

[4]:
```python
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[currl] <= 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]:
```
def knapsack_generate_test_cases(
        fname: str,
        tnum: int,
        init: 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.
        - init:     the initial value of nodes number
        - step:     the step test size is increased each loop
        - maxRate:  maximum rate
        - minRate:  minimum rate
        - maxValue: maximum of item value.
    '''

    file = open(fname, 'w')

    test_size = init # the size of test case
    count = 1
```

```
while count <= tnum:

    test_case = '' # test case

    '''
    Step 1. Generate the capacity of knapsack
    '''
    test_capacity = random.randint(
        int(minRate * test_size * 2 * step),
        int(maxRate * test_size * 2 * step)
    )
    test_case += str(test_capacity) + '#'

    '''
    Step 2. 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 3. 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 4. Write the test_case into the file {fname}
        '''
        print(test_case)
        if count != tnum:
            test_case += '\n'
        file.write(test_case)

        count += 1
        test_size += step

    file.close()
```

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

[7]:
```
testFile = 'knapsack_test_cases'
num = 12 # Number of test cases
init = 4
step = 1
maxRate = 3.5
minRate = 0.8
maxValue = 20
print('All tests generated:\n')
knapsack_generate_test_cases( testFile, num, init, step, maxRate, minRate,␣
 ↪maxValue )
```

```
All tests generated:

23#20 1 19 18#18 6 8 15#0 0 1 1
15#15 1 11 1 16#2 10 5 7 8#1 0 1 0 1
31#14 8 15 9 11 12#8 16 9 8 17 12#1 0 1 0 0 1
14#10 7 15 6 7 7 9#1 3 4 5 6 6 2#1 0 1 0 0 1 1
47#2 11 14 7 18 15 3 11#18 5 14 20 13 10 11 6#0 0 1 0 1 1 0 1
60#7 4 3 11 10 16 5 6 11#15 11 7 7 10 21 17 7 9#0 0 0 1 1 1 0 1 1
64#6 3 12 17 11 5 4 13 5 13#21 11 15 6 13 17 21 14 7 11#0 0 1 1 1 0 0 1 0 1
37#18 4 11 9 9 19 12 4 14 17 6#6 5 5 8 3 3 3 11 9 6 2#1 0 1 0 1 1 1 0 1 1 1
41#16 15 3 4 11 7 11 17 8 7 20 6#6 2 2 3 7 8 7 9 4 6 9 4#1 1 0 0 0 0 1 1 1 0 1 1
20#2 14 8 12 7 13 5 1 9 1 6 13 20#2 4 2 2 5 2 4 2 3 4 1 5 4#0 1 1 1 0 1 0 0 0 0
1 1 1
23#5 13 3 8 17 3 1 12 20 13 14 4 5 12#2 4 4 3 4 4 3 5 3 3 2 5 2 4#0 1 0 1 1 0 0
0 1 1 1 0 0 1
85#5 17 11 20 11 12 10 2 18 14 14 1 2 4 12#10 5 15 7 12 18 4 9 10 13 17 16 19 16
8#0 1 0 1 1 0 1 1 1 1 1 0 0 0 1
```

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. Unlike the demand in 01-knapsack problem, in the fractional one, we can take fractions of items so that we can maximumly utilize the capacity

We implement this algorithm based on a greedy approach: always pick the item with the highest $\frac{value}{weight}$ ratio and take as much of it as possible. If the item can't fit completely into the knapsack, take the fraction of it that fits to make the total weights meets the capacity.

def $FractionalBest(v, w, M)$**:**
  global variables: X,optP,optX
  permute the indices so that $\frac{p_i}{w_i}$ **is decreasing**
  for $i \in \{0, 1, ..., n-1\}$ **do**
    if $curW < M$ **then**
      if $curW + w_i \leq M$ **then**
        $x_i \leftarrow 1$
        $curW \leftarrow curW + w_i$
        $curP \leftarrow curP + p_i$
      else
        $x_i \leftarrow \frac{M - curW}{w_i}$
        $curW \leftarrow M$
        $curP \leftarrow curP + p_i x_i$
   return P

[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
    """

    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]:
```
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)
```

```python
    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. The idea is if the bound of the current partial solution is less than or equal to the current best solution (optP), the search branch will be pruned. The key steps are: 1. **Feasibility Check**: - If the solution is complete (all items considered), check its feasibility. - If feasible and the profit is higher than the current best, update the optimal solution and profit. 2. **Bounding**: - Use `getBound` to calculate the profit bound for the current partial solution. - If the bound is less than or equal to the current best profit, prune the branch. 3. **Constructing Choices**: - For the current item, determine feasible choices (`0` or `1`) based on weight capacity. - Prune infeasible choices where the item's weight exceeds the remaining capacity. 4. **Recursive Exploration**: - Recursively explore each choice, updating the solution as needed.apacity.

Here we have the psuedo code and its implementation of **bounding** algorithm to solve 01-knapsack problem:

def $Bounding(list, curW)$**:**
- **global variables: X,optP,optX**
- **if** $len(list) = n$ **then**
  - **if** $\sum_{i=0}^{n-1} w_i x_i \leq M \quad and \quad \sum_{i=0}^{n-1} p_i x_i > optP$ **then**
    - $optP \leftarrow (\sum_{i=0}^{n-1} p_i x_i)$
    - $optX \leftarrow [x_0, ..., x_{n-1}]$
- **else**
  - **Bound** $\leftarrow \sum_{i=0}^{n-1} p_i x_i +$ **fractional_rest_best**
  - **if** $Bound \leq optP$ **then**
    - $return$
  - **else**
    - **if** $curW + w_{cur(l)} \leq M$ **then**
      - $C_l \leftarrow 1, 0$
    - **else**
      - $C_l \leftarrow 0$
    - **for x in C**$_l$: **Bounding(list+[x], curW+w**$_{cur(l)} x$)
  - **return** optX

```python
[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
```

```python
        - 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
            '''
```

```python
            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[currl] <= 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]:
```python
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( [20, 1, 19, 18], [18, 6, 8, 15], 23 ) = [0, 0, 1, 1]
solution: [0, 0, 1, 1]
result:   True

Test No:  02
items:    05
knapsack( [15, 1, 11, 1, 16], [2, 10, 5, 7, 8], 15 ) = [1, 0, 1, 0, 1]
solution: [1, 0, 1, 0, 1]
result:   True

Test No:  03
items:    06
knapsack( [14, 8, 15, 9, 11, 12], [8, 16, 9, 8, 17, 12], 31 ) = [1, 0, 1, 0, 0,
1]
solution: [1, 0, 1, 0, 0, 1]
result:   True

Test No:  04
items:    07
knapsack( [10, 7, 15, 6, 7, 7, 9], [1, 3, 4, 5, 6, 6, 2], 14 ) = [1, 0, 1, 0, 0,
1, 1]
solution: [1, 0, 1, 0, 0, 1, 1]
result:   True

Test No:  05
items:    08
knapsack( [2, 11, 14, 7, 18, 15, 3, 11], [18, 5, 14, 20, 13, 10, 11, 6], 47 ) =
[0, 0, 1, 0, 1, 1, 0, 1]
solution: [0, 0, 1, 0, 1, 1, 0, 1]
result:   True

Test No:  06
items:    09
knapsack( [7, 4, 3, 11, 10, 16, 5, 6, 11], [15, 11, 7, 7, 10, 21, 17, 7, 9], 60
) = [0, 0, 0, 1, 1, 1, 0, 1, 1]
solution: [0, 0, 0, 1, 1, 1, 0, 1, 1]
result:   True
```

```
Test No:  07
items:    10
knapsack( [6, 3, 12, 17, 11, 5, 4, 13, 5, 13], [21, 11, 15, 6, 13, 17, 21, 14,
7, 11], 64 ) = [0, 0, 1, 1, 1, 0, 0, 1, 0, 1]
solution: [0, 0, 1, 1, 1, 0, 0, 1, 0, 1]
result:   True

Test No:  08
items:    11
knapsack( [18, 4, 11, 9, 9, 19, 12, 4, 14, 17, 6], [6, 5, 5, 8, 3, 3, 3, 11, 9,
6, 2], 37 ) = [1, 0, 1, 0, 1, 1, 1, 0, 1, 1, 1]
solution: [1, 0, 1, 0, 1, 1, 1, 0, 1, 1, 1]
result:   True

Test No:  09
items:    12
knapsack( [16, 15, 3, 4, 11, 7, 11, 17, 8, 7, 20, 6], [6, 2, 2, 3, 7, 8, 7, 9,
4, 6, 9, 4], 41 ) = [1, 1, 0, 0, 0, 0, 1, 1, 1, 0, 1, 1]
solution: [1, 1, 0, 0, 0, 0, 1, 1, 1, 0, 1, 1]
result:   True

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

Test No:  11
items:    14
knapsack( [5, 13, 3, 8, 17, 3, 1, 12, 20, 13, 14, 4, 5, 12], [2, 4, 4, 3, 4, 4,
3, 5, 3, 3, 2, 5, 2, 4], 23 ) = [0, 1, 0, 1, 1, 0, 0, 0, 1, 1, 1, 0, 0, 1]
solution: [0, 1, 0, 1, 1, 0, 0, 0, 1, 1, 1, 0, 0, 1]
result:   True

Test No:  12
items:    15
knapsack( [5, 17, 11, 20, 11, 12, 10, 2, 18, 14, 14, 1, 2, 4, 12], [10, 5, 15,
7, 12, 18, 4, 9, 10, 13, 17, 16, 19, 16, 8], 85 ) = [0, 1, 0, 1, 1, 0, 1, 1, 1,
1, 1, 0, 0, 0, 1]
solution: [0, 1, 0, 1, 1, 0, 1, 1, 1, 1, 1, 0, 0, 0, 1]
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. Instead of calculating the total value for every combination of items, we use a bound to estimate the maximum value we can achieve with the remaining capacity. The bound gives us an upper limit for the total value of a branch (partial solution). If the bound is less than or equal to the current best solution, we prune that branch and do not explore further, considering only the choice of excluding the item then.

Here we have the psuedo code and its implementation of **branch and bounding** algorithm to solve 01-knapsack problem:

```
def BranchAndBound(list):
    global variables: X,optP,optX
    if len(list) = n then
        if Σ_{i=0}^{n-1} w_i x_i ≤ M   and   Σ_{i=0}^{n-1} p_i x_i > optP then
            optP ← (Σ_{i=0}^{n-1} p_i x_i)
            optX ← [x_0, ..., x_{n-1}]
    else
        if curW + w_cur(l) ≤ M then
            C_l ← {1, 0}
        else
            C_l ← {0}
        for x in C_l do
            nextchoice add list+[x]
            nextbound add fractional_best(x)
        if nextbound[0] ≤ optP then
            return
        if len(C_l) = 2   and   nextbound[0] < nextbound[1] then
            switch so that nextbound is decreasing
        for l in nextchoice do
            BranchAndBound(l)
        return optX
```

[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][:]

            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]:
```
knapsack_run_test_cases(testFile, knapsack_branchAndBound)
```

```
Test No:  01
items:    04
knapsack( [20, 1, 19, 18], [18, 6, 8, 15], 23 ) = [0, 0, 1, 1]
solution: [0, 0, 1, 1]
result:   True

Test No:  02
items:    05
knapsack( [15, 1, 11, 1, 16], [2, 10, 5, 7, 8], 15 ) = [1, 0, 1, 0, 1]
solution: [1, 0, 1, 0, 1]
result:   True

Test No:  03
items:    06
knapsack( [14, 8, 15, 9, 11, 12], [8, 16, 9, 8, 17, 12], 31 ) = [1, 0, 1, 0, 0,
1]
solution: [1, 0, 1, 0, 0, 1]
```

```
result:   True

Test No:  04
items:    07
knapsack( [10, 7, 15, 6, 7, 7, 9], [1, 3, 4, 5, 6, 6, 2], 14 ) = [1, 1, 1, 0, 0,
0, 1]
solution: [1, 0, 1, 0, 0, 1, 1]
result:   True

Test No:  05
items:    08
knapsack( [2, 11, 14, 7, 18, 15, 3, 11], [18, 5, 14, 20, 13, 10, 11, 6], 47 ) =
[0, 1, 1, 0, 1, 1, 0, 0]
solution: [0, 0, 1, 0, 1, 1, 0, 1]
result:   True

Test No:  06
items:    09
knapsack( [7, 4, 3, 11, 10, 16, 5, 6, 11], [15, 11, 7, 7, 10, 21, 17, 7, 9], 60
) = [0, 0, 0, 1, 1, 1, 0, 1, 1]
solution: [0, 0, 0, 1, 1, 1, 0, 1, 1]
result:   True

Test No:  07
items:    10
knapsack( [6, 3, 12, 17, 11, 5, 4, 13, 5, 13], [21, 11, 15, 6, 13, 17, 21, 14,
7, 11], 64 ) = [0, 0, 1, 1, 1, 0, 0, 1, 0, 1]
solution: [0, 0, 1, 1, 1, 0, 0, 1, 0, 1]
result:   True

Test No:  08
items:    11
knapsack( [18, 4, 11, 9, 9, 19, 12, 4, 14, 17, 6], [6, 5, 5, 8, 3, 3, 3, 11, 9,
6, 2], 37 ) = [1, 0, 1, 0, 1, 1, 1, 0, 1, 1, 1]
solution: [1, 0, 1, 0, 1, 1, 1, 0, 1, 1, 1]
result:   True

Test No:  09
items:    12
knapsack( [16, 15, 3, 4, 11, 7, 11, 17, 8, 7, 20, 6], [6, 2, 2, 3, 7, 8, 7, 9,
4, 6, 9, 4], 41 ) = [1, 1, 0, 0, 1, 0, 0, 1, 1, 0, 1, 1]
solution: [1, 1, 0, 0, 0, 0, 1, 1, 1, 0, 1, 1]
result:   True

Test No:  10
items:    13
knapsack( [2, 14, 8, 12, 7, 13, 5, 1, 9, 1, 6, 13, 20], [2, 4, 2, 2, 5, 2, 4, 2,
3, 4, 1, 5, 4], 20 ) = [0, 1, 1, 1, 0, 1, 0, 0, 0, 0, 1, 1, 1]
```

```
solution: [0, 1, 1, 1, 0, 1, 0, 0, 0, 0, 1, 1, 1]
result:   True

Test No:  11
items:    14
knapsack( [5, 13, 3, 8, 17, 3, 1, 12, 20, 13, 14, 4, 5, 12], [2, 4, 4, 3, 4, 4,
3, 5, 3, 3, 2, 5, 2, 4], 23 ) = [0, 1, 0, 1, 1, 0, 0, 0, 1, 1, 1, 0, 0, 1]
solution: [0, 1, 0, 1, 1, 0, 0, 0, 1, 1, 1, 0, 0, 1]
result:   True

Test No:  12
items:    15
knapsack( [5, 17, 11, 20, 11, 12, 10, 2, 18, 14, 14, 1, 2, 4, 12], [10, 5, 15,
7, 12, 18, 4, 9, 10, 13, 17, 16, 19, 16, 8], 85 ) = [0, 1, 0, 1, 1, 0, 1, 1, 1,
1, 1, 0, 0, 0, 1]
solution: [0, 1, 0, 1, 1, 0, 1, 1, 1, 1, 1, 0, 0, 0, 1]
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.

Moreover, to better visualize the comparision, we can use matplot to draw a graph that shows the running times of these algorithms.

[15]:
```
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('Items Number')
        plt.ylabel('Running Time (seconds)')
        plt.title('Running Time Comparison of Knapsack Algorithms')
        plt.legend()
        plt.grid(True)
        plt.tight_layout()
        plt.show()
```

Finally, we excecute the following code snippet to finish.

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

algoNames = [
    'Backtracking-General',
    'Backtracking-Pruning',
    'Backtracking-Bounding',
    'Backtracking-BranchAndBound'
]

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

```
------------------ Test No.1 ------------------

items:    04
values:   [20, 1, 19, 18]
weights:  [18, 6, 8, 15]
solution: [0, 0, 1, 1]

algorithm:    Backtracking-General
runningtime:  0.0002530000
correctness:  True
knapsack([20, 1, 19, 18],[18, 6, 8, 15],23) = [0, 0, 1, 1]

algorithm:    Backtracking-Pruning
runningtime:  0.0001480000
correctness:  True
knapsack([20, 1, 19, 18],[18, 6, 8, 15],23) = [0, 0, 1, 1]

algorithm:    Backtracking-Bounding
runningtime:  0.0003250000
correctness:  True
knapsack([20, 1, 19, 18],[18, 6, 8, 15],23) = [0, 0, 1, 1]

algorithm:    Backtracking-BranchAndBound
runningtime:  0.0004520000
correctness:  True
knapsack([20, 1, 19, 18],[18, 6, 8, 15],23) = [0, 0, 1, 1]

------------------ Test No.2 ------------------

items:    05
```

```
values:   [15, 1, 11, 1, 16]
weights:  [2, 10, 5, 7, 8]
solution: [1, 0, 1, 0, 1]

algorithm:    Backtracking-General
runningtime:  0.0003790000
correctness:  True
knapsack([15, 1, 11, 1, 16],[2, 10, 5, 7, 8],15) = [1, 0, 1, 0, 1]

algorithm:    Backtracking-Pruning
runningtime:  0.0001420000
correctness:  True
knapsack([15, 1, 11, 1, 16],[2, 10, 5, 7, 8],15) = [1, 0, 1, 0, 1]

algorithm:    Backtracking-Bounding
runningtime:  0.0005490000
correctness:  True
knapsack([15, 1, 11, 1, 16],[2, 10, 5, 7, 8],15) = [1, 0, 1, 0, 1]

algorithm:    Backtracking-BranchAndBound
runningtime:  0.0004690000
correctness:  True
knapsack([15, 1, 11, 1, 16],[2, 10, 5, 7, 8],15) = [1, 0, 1, 0, 1]

------------------ Test No.3 ------------------

items:    06
values:   [14, 8, 15, 9, 11, 12]
weights:  [8, 16, 9, 8, 17, 12]
solution: [1, 0, 1, 0, 0, 1]

algorithm:    Backtracking-General
runningtime:  0.0005180000
correctness:  True
knapsack([14, 8, 15, 9, 11, 12],[8, 16, 9, 8, 17, 12],31) = [1, 0, 1, 0, 0, 1]

algorithm:    Backtracking-Pruning
runningtime:  0.0004620000
correctness:  True
knapsack([14, 8, 15, 9, 11, 12],[8, 16, 9, 8, 17, 12],31) = [1, 0, 1, 0, 0, 1]

algorithm:    Backtracking-Bounding
runningtime:  0.0008560000
correctness:  True
knapsack([14, 8, 15, 9, 11, 12],[8, 16, 9, 8, 17, 12],31) = [1, 0, 1, 0, 0, 1]

algorithm:    Backtracking-BranchAndBound
runningtime:  0.0004340000
```

```
correctness:  True
knapsack([14, 8, 15, 9, 11, 12],[8, 16, 9, 8, 17, 12],31) = [1, 0, 1, 0, 0, 1]

------------------ Test No.4 ------------------

items:    07
values:   [10, 7, 15, 6, 7, 7, 9]
weights:  [1, 3, 4, 5, 6, 6, 2]
solution: [1, 0, 1, 0, 0, 1, 1]

algorithm:    Backtracking-General
runningtime:  0.0010470000
correctness:  True
knapsack([10, 7, 15, 6, 7, 7, 9],[1, 3, 4, 5, 6, 6, 2],14) = [1, 0, 1, 0, 0, 1,
1]

algorithm:    Backtracking-Pruning
runningtime:  0.0010090000
correctness:  True
knapsack([10, 7, 15, 6, 7, 7, 9],[1, 3, 4, 5, 6, 6, 2],14) = [1, 0, 1, 0, 0, 1,
1]

algorithm:    Backtracking-Bounding
runningtime:  0.0016980000
correctness:  True
knapsack([10, 7, 15, 6, 7, 7, 9],[1, 3, 4, 5, 6, 6, 2],14) = [1, 0, 1, 0, 0, 1,
1]

algorithm:    Backtracking-BranchAndBound
runningtime:  0.0008300000
correctness:  True
knapsack([10, 7, 15, 6, 7, 7, 9],[1, 3, 4, 5, 6, 6, 2],14) = [1, 1, 1, 0, 0, 0,
1]

------------------ Test No.5 ------------------

items:    08
values:   [2, 11, 14, 7, 18, 15, 3, 11]
weights:  [18, 5, 14, 20, 13, 10, 11, 6]
solution: [0, 0, 1, 0, 1, 1, 0, 1]

algorithm:    Backtracking-General
runningtime:  0.0021880000
correctness:  True
knapsack([2, 11, 14, 7, 18, 15, 3, 11],[18, 5, 14, 20, 13, 10, 11, 6],47) = [0,
0, 1, 0, 1, 1, 0, 1]

algorithm:    Backtracking-Pruning
```

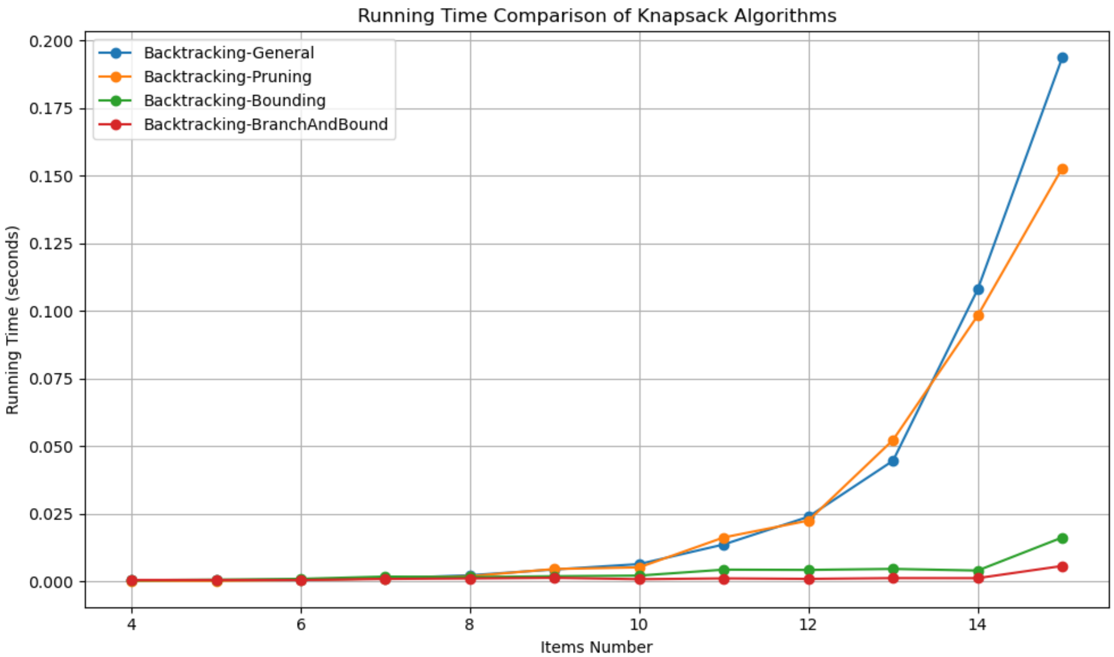
```
runningtime:  0.0018510000
correctness:  True
knapsack([2, 11, 14, 7, 18, 15, 3, 11],[18, 5, 14, 20, 13, 10, 11, 6],47) = [0,
0, 1, 0, 1, 1, 0, 1]

algorithm:    Backtracking-Bounding
runningtime:  0.0015290000
correctness:  True
knapsack([2, 11, 14, 7, 18, 15, 3, 11],[18, 5, 14, 20, 13, 10, 11, 6],47) = [0,
0, 1, 0, 1, 1, 0, 1]

algorithm:    Backtracking-BranchAndBound
runningtime:  0.0010140000
correctness:  True
knapsack([2, 11, 14, 7, 18, 15, 3, 11],[18, 5, 14, 20, 13, 10, 11, 6],47) = [0,
1, 1, 0, 1, 1, 0, 0]

------------------ Test No.6 ------------------

items:    09
values:   [7, 4, 3, 11, 10, 16, 5, 6, 11]
weights:  [15, 11, 7, 7, 10, 21, 17, 7, 9]
solution: [0, 0, 0, 1, 1, 1, 0, 1, 1]

algorithm:    Backtracking-General
runningtime:  0.0043360000
correctness:  True
knapsack([7, 4, 3, 11, 10, 16, 5, 6, 11],[15, 11, 7, 7, 10, 21, 17, 7, 9],60) =
[0, 0, 0, 1, 1, 1, 0, 1, 1]

algorithm:    Backtracking-Pruning
runningtime:  0.0045070000
correctness:  True
knapsack([7, 4, 3, 11, 10, 16, 5, 6, 11],[15, 11, 7, 7, 10, 21, 17, 7, 9],60) =
[0, 0, 0, 1, 1, 1, 0, 1, 1]

algorithm:    Backtracking-Bounding
runningtime:  0.0018200000
correctness:  True
knapsack([7, 4, 3, 11, 10, 16, 5, 6, 11],[15, 11, 7, 7, 10, 21, 17, 7, 9],60) =
[0, 0, 0, 1, 1, 1, 0, 1, 1]

algorithm:    Backtracking-BranchAndBound
runningtime:  0.0012430000
correctness:  True
knapsack([7, 4, 3, 11, 10, 16, 5, 6, 11],[15, 11, 7, 7, 10, 21, 17, 7, 9],60) =
[0, 0, 0, 1, 1, 1, 0, 1, 1]
```

```
------------------ Test No.7 ------------------

items:    10
values:   [6, 3, 12, 17, 11, 5, 4, 13, 5, 13]
weights:  [21, 11, 15, 6, 13, 17, 21, 14, 7, 11]
solution: [0, 0, 1, 1, 1, 0, 0, 1, 0, 1]

algorithm:    Backtracking-General
runningtime:  0.0063200000
correctness:  True
knapsack([6, 3, 12, 17, 11, 5, 4, 13, 5, 13],[21, 11, 15, 6, 13, 17, 21, 14, 7,
11],64) = [0, 0, 1, 1, 1, 0, 0, 1, 0, 1]

algorithm:    Backtracking-Pruning
runningtime:  0.0051090000
correctness:  True
knapsack([6, 3, 12, 17, 11, 5, 4, 13, 5, 13],[21, 11, 15, 6, 13, 17, 21, 14, 7,
11],64) = [0, 0, 1, 1, 1, 0, 0, 1, 0, 1]

algorithm:    Backtracking-Bounding
runningtime:  0.0020630000
correctness:  True
knapsack([6, 3, 12, 17, 11, 5, 4, 13, 5, 13],[21, 11, 15, 6, 13, 17, 21, 14, 7,
11],64) = [0, 0, 1, 1, 1, 0, 0, 1, 0, 1]

algorithm:    Backtracking-BranchAndBound
runningtime:  0.0007270000
correctness:  True
knapsack([6, 3, 12, 17, 11, 5, 4, 13, 5, 13],[21, 11, 15, 6, 13, 17, 21, 14, 7,
11],64) = [0, 0, 1, 1, 1, 0, 0, 1, 0, 1]

------------------ Test No.8 ------------------

items:    11
values:   [18, 4, 11, 9, 9, 19, 12, 4, 14, 17, 6]
weights:  [6, 5, 5, 8, 3, 3, 3, 11, 9, 6, 2]
solution: [1, 0, 1, 0, 1, 1, 1, 0, 1, 1, 1]

algorithm:    Backtracking-General
runningtime:  0.0135920000
correctness:  True
knapsack([18, 4, 11, 9, 9, 19, 12, 4, 14, 17, 6],[6, 5, 5, 8, 3, 3, 3, 11, 9, 6,
2],37) = [1, 0, 1, 0, 1, 1, 1, 0, 1, 1, 1]

algorithm:    Backtracking-Pruning
runningtime:  0.0161850000
correctness:  True
knapsack([18, 4, 11, 9, 9, 19, 12, 4, 14, 17, 6],[6, 5, 5, 8, 3, 3, 3, 11, 9, 6,
```

```
2],37) = [1, 0, 1, 0, 1, 1, 1, 0, 1, 1, 1]

algorithm:    Backtracking-Bounding
runningtime:  0.0042650000
correctness:  True
knapsack([18, 4, 11, 9, 9, 19, 12, 4, 14, 17, 6],[6, 5, 5, 8, 3, 3, 3, 11, 9, 6,
2],37) = [1, 0, 1, 0, 1, 1, 1, 0, 1, 1, 1]

algorithm:    Backtracking-BranchAndBound
runningtime:  0.0010360000
correctness:  True
knapsack([18, 4, 11, 9, 9, 19, 12, 4, 14, 17, 6],[6, 5, 5, 8, 3, 3, 3, 11, 9, 6,
2],37) = [1, 0, 1, 0, 1, 1, 1, 0, 1, 1, 1]

------------------ Test No.9 ------------------

items:    12
values:   [16, 15, 3, 4, 11, 7, 11, 17, 8, 7, 20, 6]
weights:  [6, 2, 2, 3, 7, 8, 7, 9, 4, 6, 9, 4]
solution: [1, 1, 0, 0, 0, 0, 1, 1, 1, 0, 1, 1]

algorithm:    Backtracking-General
runningtime:  0.0238200000
correctness:  True
knapsack([16, 15, 3, 4, 11, 7, 11, 17, 8, 7, 20, 6],[6, 2, 2, 3, 7, 8, 7, 9, 4,
6, 9, 4],41) = [1, 1, 0, 0, 0, 0, 1, 1, 1, 0, 1, 1]

algorithm:    Backtracking-Pruning
runningtime:  0.0224200000
correctness:  True
knapsack([16, 15, 3, 4, 11, 7, 11, 17, 8, 7, 20, 6],[6, 2, 2, 3, 7, 8, 7, 9, 4,
6, 9, 4],41) = [1, 1, 0, 0, 0, 0, 1, 1, 1, 0, 1, 1]

algorithm:    Backtracking-Bounding
runningtime:  0.0041640000
correctness:  True
knapsack([16, 15, 3, 4, 11, 7, 11, 17, 8, 7, 20, 6],[6, 2, 2, 3, 7, 8, 7, 9, 4,
6, 9, 4],41) = [1, 1, 0, 0, 0, 0, 1, 1, 1, 0, 1, 1]

algorithm:    Backtracking-BranchAndBound
runningtime:  0.0008550000
correctness:  True
knapsack([16, 15, 3, 4, 11, 7, 11, 17, 8, 7, 20, 6],[6, 2, 2, 3, 7, 8, 7, 9, 4,
6, 9, 4],41) = [1, 1, 0, 0, 1, 0, 0, 1, 1, 0, 1, 1]

------------------ Test No.10 ------------------

items:    13
```

```
values:   [2, 14, 8, 12, 7, 13, 5, 1, 9, 1, 6, 13, 20]
weights:  [2, 4, 2, 2, 5, 2, 4, 2, 3, 4, 1, 5, 4]
solution: [0, 1, 1, 1, 0, 1, 0, 0, 0, 0, 1, 1, 1]

algorithm:    Backtracking-General
runningtime:  0.0445540000
correctness:  True
knapsack([2, 14, 8, 12, 7, 13, 5, 1, 9, 1, 6, 13, 20],[2, 4, 2, 2, 5, 2, 4, 2,
3, 4, 1, 5, 4],20) = [0, 1, 1, 1, 0, 1, 0, 0, 0, 0, 1, 1, 1]

algorithm:    Backtracking-Pruning
runningtime:  0.0522220000
correctness:  True
knapsack([2, 14, 8, 12, 7, 13, 5, 1, 9, 1, 6, 13, 20],[2, 4, 2, 2, 5, 2, 4, 2,
3, 4, 1, 5, 4],20) = [0, 1, 1, 1, 0, 1, 0, 0, 0, 0, 1, 1, 1]

algorithm:    Backtracking-Bounding
runningtime:  0.0045400000
correctness:  True
knapsack([2, 14, 8, 12, 7, 13, 5, 1, 9, 1, 6, 13, 20],[2, 4, 2, 2, 5, 2, 4, 2,
3, 4, 1, 5, 4],20) = [0, 1, 1, 1, 0, 1, 0, 0, 0, 0, 1, 1, 1]

algorithm:    Backtracking-BranchAndBound
runningtime:  0.0011360000
correctness:  True
knapsack([2, 14, 8, 12, 7, 13, 5, 1, 9, 1, 6, 13, 20],[2, 4, 2, 2, 5, 2, 4, 2,
3, 4, 1, 5, 4],20) = [0, 1, 1, 1, 0, 1, 0, 0, 0, 0, 1, 1, 1]

------------------ Test No.11 ------------------

items:    14
values:   [5, 13, 3, 8, 17, 3, 1, 12, 20, 13, 14, 4, 5, 12]
weights:  [2, 4, 4, 3, 4, 4, 3, 5, 3, 3, 2, 5, 2, 4]
solution: [0, 1, 0, 1, 1, 0, 0, 0, 1, 1, 1, 0, 0, 1]

algorithm:    Backtracking-General
runningtime:  0.1078590000
correctness:  True
knapsack([5, 13, 3, 8, 17, 3, 1, 12, 20, 13, 14, 4, 5, 12],[2, 4, 4, 3, 4, 4, 3,
5, 3, 3, 2, 5, 2, 4],23) = [0, 1, 0, 1, 1, 0, 0, 0, 1, 1, 1, 0, 0, 1]

algorithm:    Backtracking-Pruning
runningtime:  0.0981950000
correctness:  True
knapsack([5, 13, 3, 8, 17, 3, 1, 12, 20, 13, 14, 4, 5, 12],[2, 4, 4, 3, 4, 4, 3,
5, 3, 3, 2, 5, 2, 4],23) = [0, 1, 0, 1, 1, 0, 0, 0, 1, 1, 1, 0, 0, 1]

algorithm:    Backtracking-Bounding
```

```
runningtime:  0.0039330000
correctness:  True
knapsack([5, 13, 3, 8, 17, 3, 1, 12, 20, 13, 14, 4, 5, 12],[2, 4, 4, 3, 4, 4, 3,
5, 3, 3, 2, 5, 2, 4],23) = [0, 1, 0, 1, 1, 0, 0, 0, 1, 1, 1, 0, 0, 1]

algorithm:    Backtracking-BranchAndBound
runningtime:  0.0011170000
correctness:  True
knapsack([5, 13, 3, 8, 17, 3, 1, 12, 20, 13, 14, 4, 5, 12],[2, 4, 4, 3, 4, 4, 3,
5, 3, 3, 2, 5, 2, 4],23) = [0, 1, 0, 1, 1, 0, 0, 0, 1, 1, 1, 0, 0, 1]

------------------ Test No.12 ------------------

items:    15
values:   [5, 17, 11, 20, 11, 12, 10, 2, 18, 14, 14, 1, 2, 4, 12]
weights:  [10, 5, 15, 7, 12, 18, 4, 9, 10, 13, 17, 16, 19, 16, 8]
solution: [0, 1, 0, 1, 1, 0, 1, 1, 1, 1, 1, 0, 0, 0, 1]

algorithm:    Backtracking-General
runningtime:  0.1936780000
correctness:  True
knapsack([5, 17, 11, 20, 11, 12, 10, 2, 18, 14, 14, 1, 2, 4, 12],[10, 5, 15, 7,
12, 18, 4, 9, 10, 13, 17, 16, 19, 16, 8],85) = [0, 1, 0, 1, 1, 0, 1, 1, 1, 1, 1,
0, 0, 0, 1]

algorithm:    Backtracking-Pruning
runningtime:  0.1527880000
correctness:  True
knapsack([5, 17, 11, 20, 11, 12, 10, 2, 18, 14, 14, 1, 2, 4, 12],[10, 5, 15, 7,
12, 18, 4, 9, 10, 13, 17, 16, 19, 16, 8],85) = [0, 1, 0, 1, 1, 0, 1, 1, 1, 1, 1,
0, 0, 0, 1]

algorithm:    Backtracking-Bounding
runningtime:  0.0161620000
correctness:  True
knapsack([5, 17, 11, 20, 11, 12, 10, 2, 18, 14, 14, 1, 2, 4, 12],[10, 5, 15, 7,
12, 18, 4, 9, 10, 13, 17, 16, 19, 16, 8],85) = [0, 1, 0, 1, 1, 0, 1, 1, 1, 1, 1,
0, 0, 0, 1]

algorithm:    Backtracking-BranchAndBound
runningtime:  0.0056290000
correctness:  True
knapsack([5, 17, 11, 20, 11, 12, 10, 2, 18, 14, 14, 1, 2, 4, 12],[10, 5, 15, 7,
12, 18, 4, 9, 10, 13, 17, 16, 19, 16, 8],85) = [0, 1, 0, 1, 1, 0, 1, 1, 1, 1, 1,
0, 0, 0, 1]
```


Running Time Comparison of Knapsack Algorithms
Backtracking-General
Backtracking-Pruning
Backtracking-Bounding
Backtracking-BranchAndBound
Running Time (seconds)
Items Number
0.200
0.175
0.150
0.125
0.100
0.075
0.050
0.025
0.000
4
6
8
10
12
14