THE USE OF KNAPSACK PACKING 0-1 (DYNAMIC PROGRAMMING) TO SOLVE OPERATIONS OPTIMISATION – WORKFORCE SCHEDULING AGAINST A DEADLINE

Word count: 1990

Table of Contents

1. System requirements and dynamic knapsack packing

1.1. Functional requirements

To complete a marketing campaign, there are limited resources available– time measured in days. The campaign value is measured here by conversion rate, defined as the percentage of people that saw an ad and then purchased the product. Each possible task in the marketing campaign has its associated conversion rate and number of days needed to complete – the input of the problem. The output should be the optimal subset of tasks that maximise the conversion rate and are finished before a given deadline and the maximised conversion rate. See appendix (dynamic knapsack user interface) for a visual of this user interface. The user that benefits from this system is the marketing manager that has the duty to prioritize the tasks that fit within a deadline. The data about the tasks needs to be collected into a csv file by the manager before importing it in the program. Then, the program is formatting it and running the dynamic knapsack on the data. The manager needs to translate the output into policy – present the selected tasks to his team.

1.2. Non - functional requirements

A marketing campaign duration can take between 4 weeks (28 days) and 1 year (365 days) , according to experts[1]. Assuming we are talking about a big company, they could have more campaigns running at the same time. Even so, the algorithm does not need to run continuously, only at the start of a marketing campaign. Also, Marketing Guides mentions only around 25 tasks for marketing campaigns.[2] Given these, the system is not required to handle very large data.

1.3. Dynamic knapsack – how it works

We build a matrix having *number of items(n)* +1 rows and *maximum capacity(m)* + 1 columns. Each entry K_{ij} will be the maximum value obtain using the first i items and having j maximum capacity. At each step the algorithm decides whether the maximum value for the given capacity is achieved by either including the i-th item or not. Therefore, K_{nm} will hold the maximum value that can be obtained using all items and the maximum capacity, which is the solution. By looking at the result matrix we can find the included items. Starting from K_{nm}, if value does not equal the one on the previous row, it means that the item was included, and we subtract the capacity by the capacity of the current item. Then, we go down by a row and repeat the process until either the capacity or the value is 0.

2. How the code works

2.1. Source and structure of the code

Section	Description	Source	Changes I made
1. Recursive programming	Implementation of knapsack using recursive programming. The algorithm is printing only the maximum value, not the items.	Re-used[3] but changed	-I added print statements to print the parameters for every time I call the function recursively -I added more comments
• Testing on example	Created input values. Run the function on them. Printed the solution in blue.	Coded fully myself	
2. Memorisation technique	Implementation of knapsack using memorisation technique (extension of the recursive approach to not call the function with the same arguments twice). The algorithm is printing only the maximum value, not the items.	Re-used[3] but changed	-I added print statements – print the initial matrix, print the parameters for every time I call the function recursively and what element in the matrix changed, and print the final matrix -I added more comments
• Testing on example	Created input values. Run the function on them. Printed the solution in blue.	Coded fully myself	
3. Generating datasets	Created the functions used for generating tasks' value and time	Coded fully myself	
4. Exhaustive enumeration	Implementation of knapsack using exhaustive enumeration. Also prints the items.	Re-used (from Lecture Week 5) but changed	-changed the name of the variables to be relevant to the problem -added print statement to print all the combinations of items and print the

			best solution every time it changes -added comments
• Testing on example	Created small input values	Coded fully myself	
• Ignore print statements	Same as the algorithm before but ignored print statements (easier to run for larger datasets)	Re-used (from Lecture Week 5) but changed	
• Testing on example	Created a bigger example – generated a list of 26 items	Coded fully myself	
• Timed Big(O) Plot	Runtime plot for exhaustive enumeration	Coded fully myself	
• Theoretical Big(O) Plot	Theoretical plot that counts iterations for exhaustive enumeration	Coded fully myself	
5. Dynamic programming	Implementation of knapsack using dynamic programming. Also prints the items.	Re-used[4] but changed	-changed the name of the variables to be relevant to the problem -decomposed the function into 3 functions -the code only printed the weight (duration) of tasks – I printed also the conversion rate and the name for each task - added print statements to print matrix K every time an element changes, and print how the conversion rate and capacity change when subtracting an item that was included -added comments
• Testing on example	Created small input values	Coded fully myself	
• Ignore print statements	Ignored the print statements to improve runtime for larger data	Re-used [4] but changed	

• Generated a csv file	Generated a file with 26 tasks	Coded fully myself	
• User interface	User inputs a csv file and a deadline. Output is the items and the maximum value.	Coded fully myself	
• Timed Big(O) plot	Maximum duration as constant	Coded fully myself	
• Timed Big(O) plot	Number of tasks as constant	Coded fully myself	
• Theoretical Big(O) plot	Number of tasks as constant	Coded fully myself	
• Theoretical Big(O) plot	Maximum duration as constant	Coded fully myself	

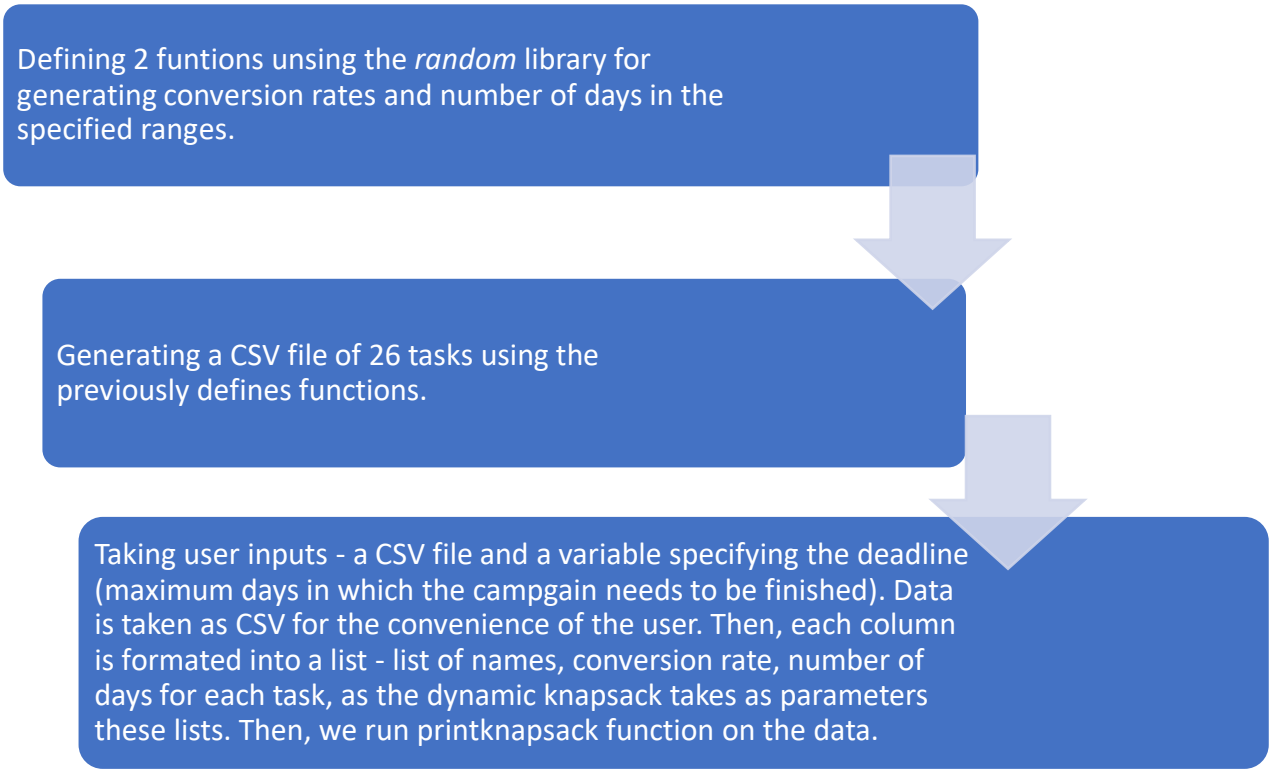
You can see above a contents table explaining for each element of the code its description, its source, and what changes I made to any re-used code.
The Code file contains 4 variations of the knapsack packing. The first 2 variations don't print the items, only the maximum value. The exhaustive enumeration performed worse on runtime then the dynamic knapsack. For these reasons I decided to solve my problem using the dynamic knapsack.

2.2. Dynamic Knapsack

Originally, the code was built in one function, but I split it into 3 functions. See the scheme bellow:

Also, I changed the variable names in the functions so they are related to marketing campaigns and I added print statements to illustrate how the algorithm works. (see 2.1. for descriptions of these) and to print the name and the value associated with each task. The next cell of the code runs these functions again without the extra print statements to improve the runtime for larger datasets.

The flow of the code is illustrated bellow:



2.3. Generalisation of the code

The same code I used can be applied to different problems. For example, the list *number of days* that stores the duration of each task can be replaced by a list storing the cost in money associated with each task. Moreover, this code can be applied to a completely different industry/problem: transport logistics – selecting which items to place in a van with a weight constraint based on value given by the revenue they generate.

2.4. Noteworthy coding features

Recursive funtions

The first implementation of the knapsack uses recursive functions. For example, inside an if-statement, it calls itself twice in order to decide if the value is maximised by including or not including the item.

List comprehensions and complicated plots

The theoretical Big(O) plot for the dynamic knapsack is a plot composed of many other sublots, generated using list comprehensions to generate the values on the x and y axis, inside a loop that generates different values of the constant (as the complexity has 2 variables one is taken as a constant) – also the number of subplots.

3. The complexity of the code

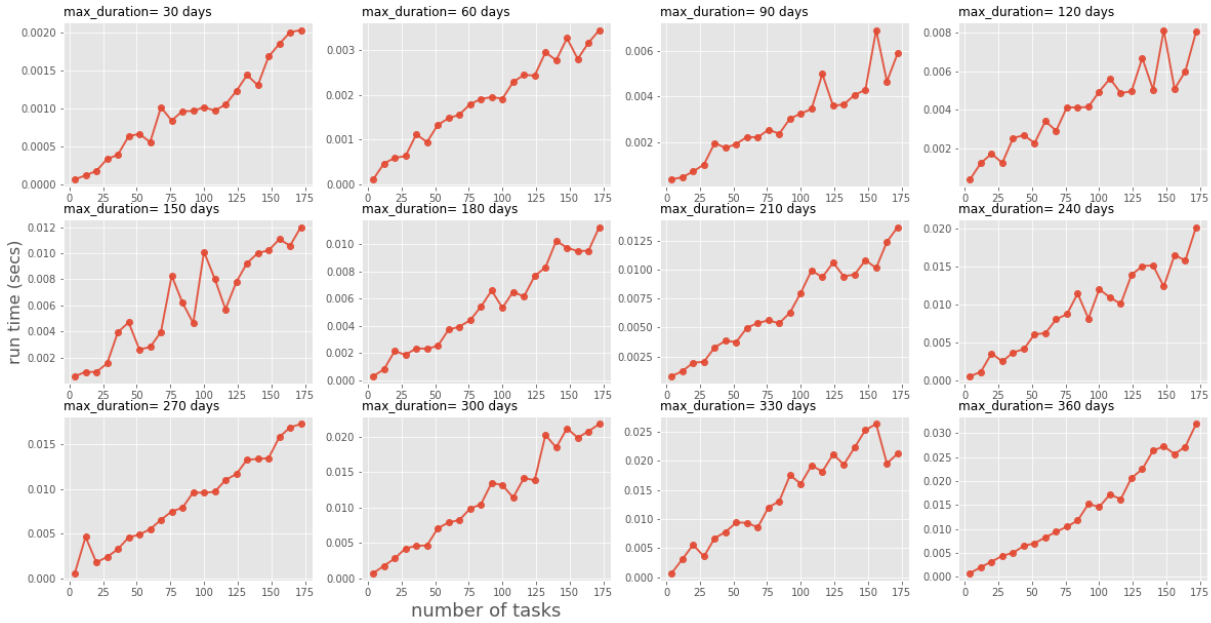
The time complexity of the dynamic knapsack is O (N W), where N is the number of tasks and W is the maximum number of days. The business problem addressed influences how these variables vary in size (see section 4 for a detailed explanation). W ranges from 30 to 360 while N ranges from 4 to 26. However, I took larger values for N (4 to 156) in the runtime plots in order to better asses the performance of the algorithm. So, for the smallest dataset the complexity would be 4 * 30= 120 loops, while for the largest dataset the complexity would be 156 * 360 = 56160 loops.

As the time complexity depends on 2 variables, I treated each of them as a constant and generated 2 plots constructed by multiple sub-pots (each sub-plot represents a different value of the constant variable).

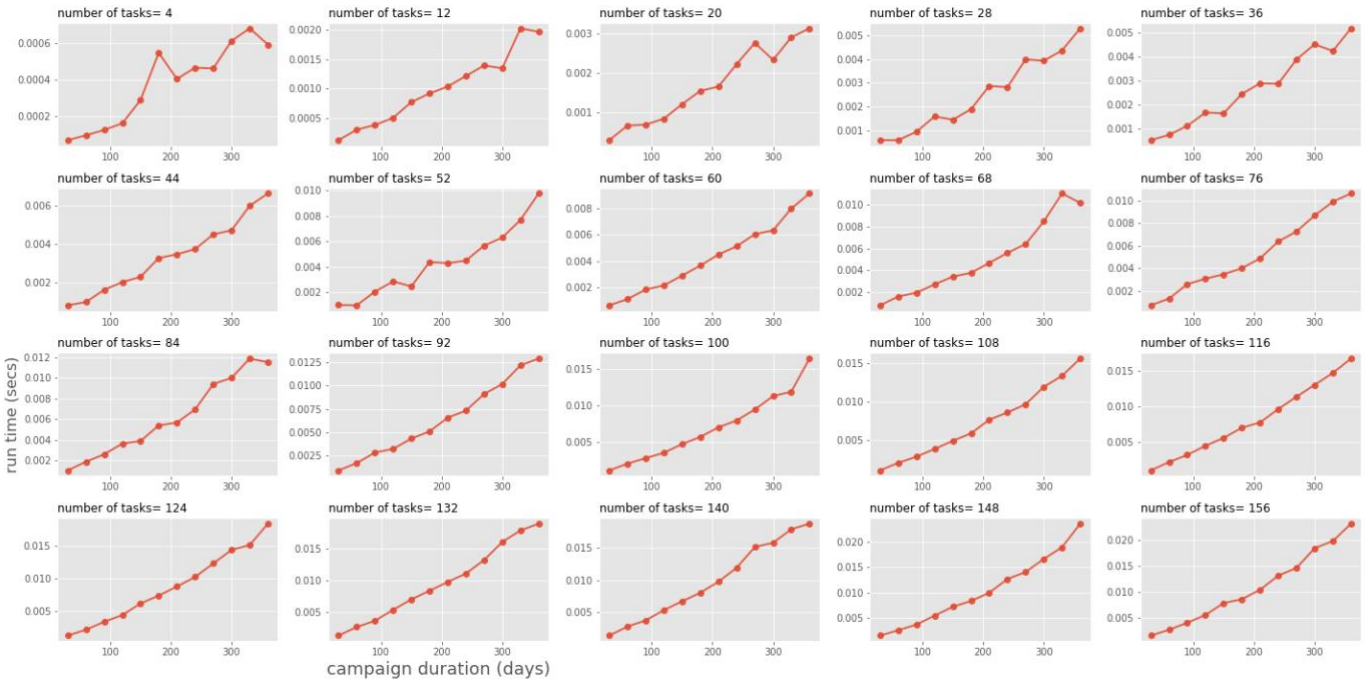
Runtime plots

The algorithm seems to perform really well in terms of runtime as the longest it runs for is 0.03 seconds for the largest dataset (N=176, W=360). For a very small dataset (N=4, W=30), its runtime is just 0.002. In addition, the code file illustrates a runtime plot for the exhaustive enumeration approach with time complexity $O(2^n)$. We can see that this alternative performs a lot worse – 60 seconds for N=25.

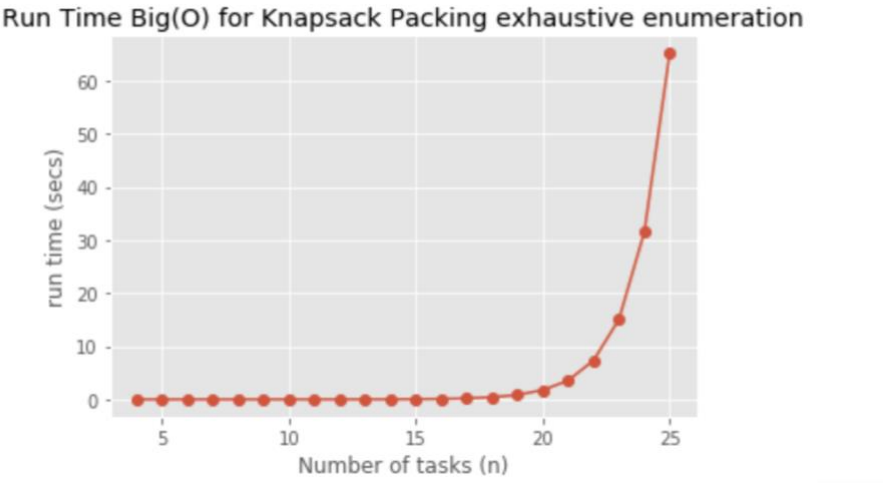
W - constant



N - constant



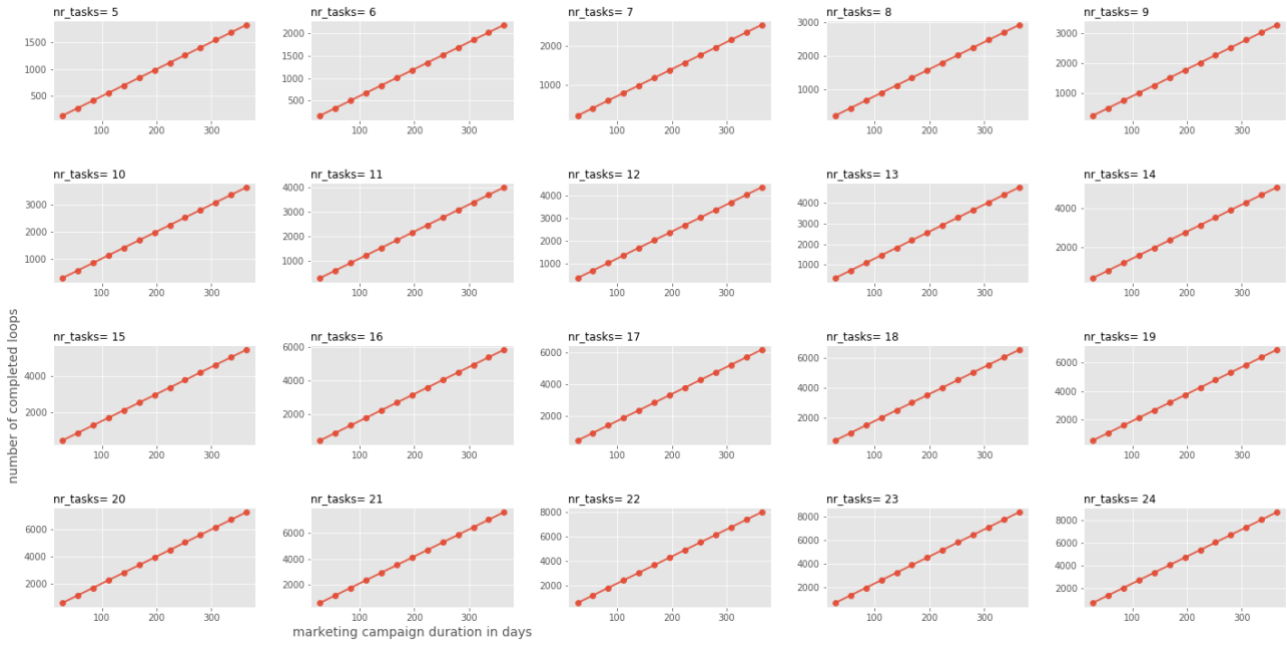
Exhaustive Enumeration



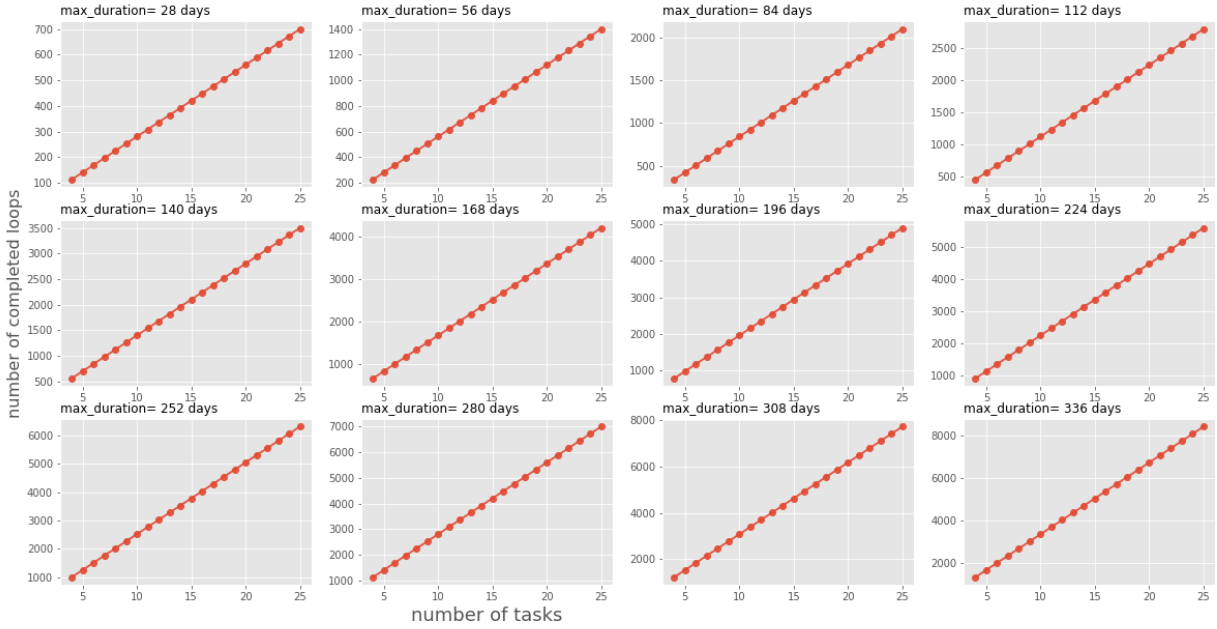
Theoretical Big(O) plots

In the best case scenario, the number of loops is around 100. In the worst case scenario (N=25, W=360), the algorithm iterates approximately 8000 times. The dynamic knapsack performs better than the exhaustive enumeration for the largest dataset (the latter completes more then 30 million iterations). However, for smaller datasets (N<6 for example) the exhaustive enumeration performs better ($2^5 < 5 \times 30$).

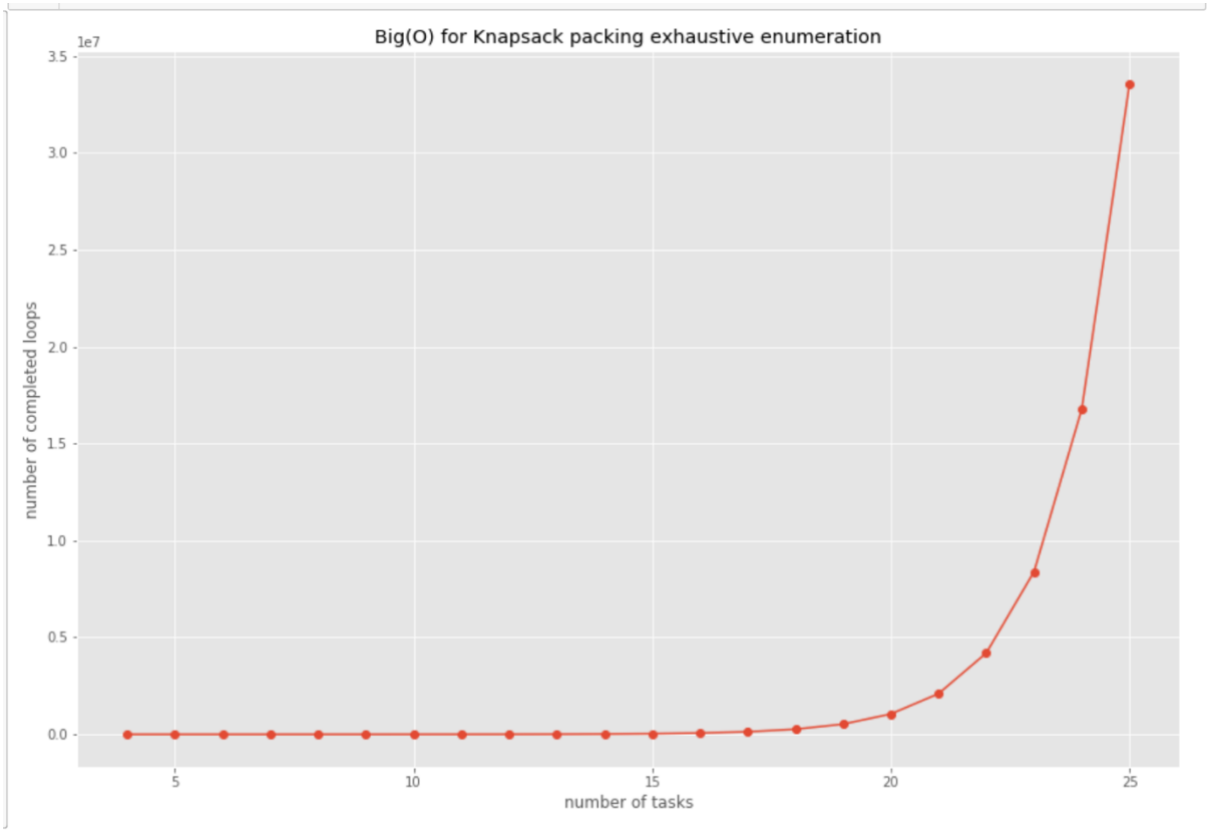
N - constant



W - constant



Exhaustive enumeration



Limitations

The runtime performance is good enough for the size of the data needed for the business problem as the worst case is 0.02 seconds. However, as the time complexity is not linear, it will perform worse on larger numbers.

4. Data used for testing the code

As described in 1.2., experts claim the duration of a marketing campaign is between 28 and 365 days and the possible number of tasks related to marketing campaigns are approximately just under 30. The conversion rate is expressed as a percentage, so it has values ranging from 0-100. The duration of each task could be between 1 and 400 (the algorithm will ignore values larger than the deadline).

See below the functions I defined for choosing random values for conversion rate and duration of tasks.

```
import random

def rollDie_conv_rate():
    return random.randint(0,100)

def rollDie_nr_days():
    return random.randint(1,401)
```

I used an input deadline of 365 days and I generated a csv file with 26 tasks using a loop and the previous functions. See below the first rows:

]:

	Name	Conversion rate	Number of days
0	task 0	75	49
1	task 1	37	336
2	task 2	50	35
3	task 3	11	336
4	task 4	100	131
5	task 5	31	343
6	task 6	97	109
7	task 7	9	114
8	task 8	74	239
9	task 9	92	2
10	task 10	87	15

I have also run the algorithm on smaller data in order to better understand how it works – by looking at the additional print statements. I have used real marketing task names for the smaller data.

```
#Test on example
name=["naming","telemarketing","social media"]
conv_rate = [ 60, 100, 120 ]
nr_days = [ 1, 2, 3 ]
max_Capacity = 5
nr_tasks = len(conv_rate)

printknapSack(max_Capacity, nr_days, conv_rate,name, nr_tasks)
```

See below how I generated data for one of the runtime plots. I explained the reason for choosing the ranges for number of tasks and deadline in section 3. For every value on the x axis (maximum time in this case) I generated a new dataset using the previously defined rollDie functions.

```
for nr_tasks in range(4,163,8):
        num+=1
        max_Capacity=[n for n in range(30,361,30)]

        times=[]

        for n in max_Capacity:
            conv_rate=[]
            nr_days=[]
            name=[]
            for i in range(nr_tasks):
                conv_rate.append(rollDie_conv_rate())
                nr_days.append(rollDie_nr_days())
                names=str(i)
                names= "task "+names
                name.append(names)
```

5. Conclusions

5.1. Good design principles – praise and improvements

Understanding user needs

The code could be improved in terms of understanding user needs. The interface could allow users to modify the deadline for the marketing campaign at any given time. Moreover, the interface could allow users to add new tasks to the programme and then calculate the new optimal subset based on that.

Simplicity

The code follows good design principles in terms of simplicity. There are no unnecessary variables or loops. Duplicating code has been avoided by working with functions and calling them again. For example, the functions that generate random numbers for conversion rate and duration are called 5 times. So the code is also re-useable. If we look at the dynamic implementation, we can see that there are no global variables, as the functions return values determined by the input values. Moreover, I used suggestive variable and function names such as showItems() or nr_tasks.

Alligned functionality

The code follows some good design principles in terms of aligned functionality. I modified the dynamic knapsack function and decomposed it into 3 functions, aligned with the functional requirements -one function for building the matrix that stores the maximum conversion rate, one function for finding the items needed and a main function that calls the previous two.

5.2. Alternative approaches

I have looked at 3 alternative approaches for solving this problem in the code file: the recursive and exhaustive enumeration approaches, that have 2^n complexity, which is really bad in runtime, and the memorisation technique (however, this does not print the items). Other alternative approach is the Greedy implementation of the 0-1 knapsack (the implementation can be found in lecture 5). This one performs better in runtime as it has log-linear complexity, but it is not guaranteed to provide an optimal solution.

5.3. Code generalisation

The structure of the code can be modified so the code is used on different problems. Currently the code takes one copy of each task as a whole. One variation might be the possibility to take fractions of each task to maximise the value gain – complete a task by 40% and another by 60% for instance. Another variation might be having multiple constraints (both time and money constraint).

5.4. Limitations in the test data

I had no access on real data about marketing campaign tasks and had to generate data. The duration and conversion rate for each task are not realistic as they are generated randomly. The range of these values is realistic, but the correspondence is not. For example, I could generate a task with a duration of 5 days and conversion rate 100%, which is very unlikely. Also, it is possible that conversion rate for one task is actually a range (between 50-60% for example).

6. Code Appendix

Knapsack packing

Contents page

Recursive programming

https://www.geeksforgeeks.org/0-1-knapsack-problem-dp-10/

```
1]:
```

```python
#for making colored outputs

from colorama import Fore
```

```
0]:
```

```python
'''
Returns the maximum value that can be put in a knapsack of capacity W

e.g. for knapsack
wt = [1,1,1]
val = [10, 20, 30]
W = 2


it returns 260
'''

def knapSack(W, wt, val, n):

    # Base Case
    if n == 0 or W == 0:
        return 0

    # If weight of the nth item is
    # more than Knapsack of capacity W,
    # then this item cannot be included
    # in the optimal solution
    if (wt[n-1] > W):
        return knapSack(W, wt, val, n-1)

    # return the maximum of two cases:
    # (1) nth item included
    # (2) not included
    else:

        print(max(
            val[n-1] + knapSack(
                W-wt[n-1], wt, val, n-1),
            knapSack(W, wt, val, n-1)) , "n= ",n , "W= ", W)
        return max(
            val[n-1] + knapSack(
                W-wt[n-1], wt, val, n-1),
            knapSack(W, wt, val, n-1))

# end of function knapSack
```

Testing on example

In [51]:

```
# test on example
wt = [1,1,1]
val = [10, 20, 30]
W = 2
n = len(val)

print (Fore.BLUE+"\n maximum value= ",knapSack(W, wt, val, n) )
```

```
10 n=  1 W=  1
20 n=  2 W=  1
10 n=  1 W=  1
10 n=  1 W=  1
10 n=  1 W=  2
30 n=  2 W=  2
10 n=  1 W=  1
10 n=  1 W=  2
50 n=  3 W=  2
10 n=  1 W=  1
20 n=  2 W=  1
10 n=  1 W=  1
10 n=  1 W=  1
10 n=  1 W=  2
30 n=  2 W=  2
10 n=  1 W=  1
10 n=  1 W=  2

 maximum value=  50
```

Memorisation technique

https://www.geeksforgeeks.org/0-1-knapsack-problem-dp-10/

View Insert Cell Kernel Widgets Help

```
Returns the maximum value that can be put in a knapsack of capacity W

e.g.
for knapsack val = [10,20,30]
wt = [1,1,1]
W = 2

it returns 50
'''

import numpy as np


# We initialize the matrix with -1 at first.
t = [[-1 for i in range(W + 1)] for j in range(n + 1)]

print("initial matrix= \n", np.matrix(t),'\n')


def knapsack(wt, val, W, n):

    # base conditions
    if n == 0 or W == 0:
        return 0

    #in case we already calculated the element we don't recurse again
    if t[n][W] != -1:
        return t[n][W]

    # return the maximum of two cases:
    # (1) nth item included
    # (2) not included
    if wt[n-1] <= W:


        t[n][W] = max(
            val[n-1] + knapsack(
                wt, val, W-wt[n-1], n-1),
            knapsack(wt

                     , val, W, n-1))
        print ("t[n][W]= ", t[n][W],"n= ",n, "W = ", W)
        return t[n][W]

    # If weight of the nth item is
    # more than Knapsack of capacity W,
    # then this item cannot be included
    # in the optimal solution

    elif wt[n-1] > W:
        t[n][W] = knapsack(wt, val, W, n-1)
        return t[n][W]

```

```
initial matrix=
 [[-1 -1 -1]
 [-1 -1 -1]
 [-1 -1 -1]
 [-1 -1 -1]]
```

Testing on example

53]:

```
# test on example
wt = [1,1,1]
val = [10, 20, 30]
W = 2
n = len(val)


print(Fore.BLUE + '\n',"maximum value= ",knapsack(wt, val, W, n),'\n')


print(Fore.BLACK+"final matrix = \n" ,np.matrix(t))
```

```
t[n][W]=  10 n=  1 W =  1
t[n][W]=  20 n=  2 W =  1
t[n][W]=  10 n=  1 W =  2
t[n][W]=  30 n=  2 W =  2
t[n][W]=  50 n=  3 W =  2

 maximum value=  50

final matrix =
 [[-1 -1 -1]
 [-1 10 10]
 [-1 20 30]
 [-1 -1 50]]
```

Generating datasets

In [7]:

```
'''
generated values for conversion rate between 0-100 (percentages)

generated values for number of days smaller than 400 (the deadline is 365 days so we assume the longest task
can be 400 - those larger than 365 will be ignored)

'''

import random

def rollDie_conv_rate():
    return random.randint(0,100)

def rollDie_nr_days():
    return random.randint(1,401)

```

Printing the items as well

Exhaustive enumeration

Source: lecture notebook

```
:
'''
the function returns all the possible combinations of a list

e.g. for ['A','B','C']

it returns [['C'], ['B'], ['B', 'C'], ['A'], ['A', 'C'], ['A', 'B'], ['A', 'B', 'C']]
'''

def combinations(items):
    combinations = []
    for i,item in enumerate(items):    #loops for all the items to append all the combinations to an empty list
        combinations  = combinations + [[item]] + [(c+[item]) for c in combinations]
    print("combinations= ",combinations,'\n')
    return combinations
```

```
:
'''
the function creates the class Item with its members number of days and conversion rate
'''

class Item(object):

    def __init__(self,nr_days,conv_rate):



        self.nr_days = nr_days
        self.conv_rate = conv_rate

    def __repr__(self):
        return "<item: nr_days={} conv_rate={}>".format( self.nr_days, self.conv_rate)
```

```
25]:
'''
the function takes as input the items and the maximum capacity and prints the best conversion rate and the items
needed to obtain it

e.g. for [Item( 2, 4),
            Item( 3, 5),
            Item( 7, 6) ]

            and maximum capacity 9 it prints the conversion rate 10 given
            by [<item: nr_days=2 conv_rate=4>, <item: nr_days=7 conv_rate=6>]
'''

def solveKnapsackProblemEE(items,maxCapacity):


    bestSolution, bestconv_rate = [], 0

    for item_combo in combinations(items):    #loop through all possible combinations and find the total nr
                                              #of days and conversion rate

        item_combo_nr_days = sum([item.nr_days for item in item_combo])
        item_combo_conv_rate = sum([item.conv_rate for item in item_combo])

        print("item_combo_nr_days= ", item_combo_nr_days, "item_combo_conv_rate= ", item_combo_conv_rate)

        if item_combo_nr_days <= maxCapacity and item_combo_conv_rate > bestconv_rate:  #add the variables to the
                                                                                        #best solution in case they satisfy the condition

            bestSolution, bestconv_rate = item_combo, item_combo_conv_rate
            print('\n',"bestSolution= ", bestSolution, "bestconv_rate= ", bestconv_rate)

    print("\n",Fore.BLUE+ "bestSolution= ", bestSolution,"\n" "bestconv_rate = ", bestconv_rate)
```

Testing on example

In [226]:
```
testItems = [Item( 2, 4),
             Item( 3, 5),
             Item( 7, 6) ]
```

In [227]:
```
solveKnapsackProblemEE(testItems, maxCapacity=9)
```

```
combinations=  [[<item: nr_days=2 conv_rate=4>], [<item: nr_days=3 conv_rate=5>], [<item: nr_days=2 conv_rate=4>, <it
em: nr_days=3 conv_rate=5>], [<item: nr_days=7 conv_rate=6>], [<item: nr_days=2 conv_rate=4>, <item: nr_days=7 conv_r
ate=6>], [<item: nr_days=3 conv_rate=5>, <item: nr_days=7 conv_rate=6>], [<item: nr_days=2 conv_rate=4>, <item: nr_da
ys=3 conv_rate=5>, <item: nr_days=7 conv_rate=6>]]

item_combo_nr_days=  2 item_combo_conv_rate=  4

 bestSolution=  [<item: nr_days=2 conv_rate=4>] bestconv_rate=  4
item_combo_nr_days=  3 item_combo_conv_rate=  5

 bestSolution=  [<item: nr_days=3 conv_rate=5>] bestconv_rate=  5
item_combo_nr_days=  5 item_combo_conv_rate=  9

 bestSolution=  [<item: nr_days=2 conv_rate=4>, <item: nr_days=3 conv_rate=5>] bestconv_rate=  9
item_combo_nr_days=  7 item_combo_conv_rate=  6
item_combo_nr_days=  9 item_combo_conv_rate=  10

 bestSolution=  [<item: nr_days=2 conv_rate=4>, <item: nr_days=7 conv_rate=6>] bestconv_rate=  10
item_combo_nr_days=  10 item_combo_conv_rate=  11
item_combo_nr_days=  12 item_combo_conv_rate=  15

 bestSolution=  [<item: nr_days=2 conv_rate=4>, <item: nr_days=7 conv_rate=6>]
bestconv_rate =  10
```

Ignore print statements

In [9]:
```
'''
I ran these functions again to ignore the print statements for the larger dataset
'''


def combinations(items):
    combinations = []
    for i,item in enumerate(items):    #loops for all the items to append all the combinations to an empty list
        combinations  = combinations + [[item]] + [(c+[item]) for c in combinations]
    #print("combinations= ",combinations,'\n')
    return combinations

def solveKnapsackProblemEE(items,maxCapacity):


    bestSolution, bestconv_rate = [], 0

    for item_combo in combinations(items):   #loop through all possible combinations and find the total nr
                                              #of days and conversion rate

        item_combo_nr_days = sum([item.nr_days for item in item_combo])
        item_combo_conv_rate = sum([item.conv_rate for item in item_combo])

        #print("item_combo_nr_days= ", item_combo_nr_days, "item_combo_conv_rate= ", item_combo_conv_rate)

        if item_combo_nr_days <= maxCapacity and item_combo_conv_rate > bestconv_rate: #add the variables to the
                                                                                       #best solution in case they satisfy the conditions

            bestSolution, bestconv_rate = item_combo, item_combo_conv_rate
            #print('\n',"bestSolution= ", bestSolution, "bestconv_rate= ", bestconv_rate)

    print(Fore.BLUE+ "bestSolution= ", bestSolution,"\n" "bestconv_rate = ", bestconv_rate)

```

Testing on example - no print statements

In [56]:
```
'''
generated a list of classes of 26 items (tasks durations and conversion rate) using the previosly
defined generators
'''

testItems=[]
for _ in range(26):
    conv_rate=rollDie_conv_rate()
    nr_days=rollDie_nr_days()
    testItems.append(Item(nr_days,conv_rate))
testItems
```

Out[56]:
```
[<item: nr_days=157 conv_rate=35>,
 <item: nr_days=58 conv_rate=66>,
 <item: nr_days=94 conv_rate=3>,
 <item: nr_days=46 conv_rate=78>,
 <item: nr_days=200 conv_rate=52>,
 <item: nr_days=97 conv_rate=52>,
 <item: nr_days=15 conv_rate=8>,
 <item: nr_days=126 conv_rate=96>,
 <item: nr_days=199 conv_rate=28>,
 <item: nr_days=353 conv_rate=79>,
 <item: nr_days=200 conv_rate=97>,
 <item: nr_days=364 conv_rate=53>,
 <item: nr_days=192 conv_rate=52>,
 <item: nr_days=128 conv_rate=88>,
 <item: nr_days=201 conv_rate=55>,
 <item: nr_days=377 conv_rate=85>,
 <item: nr_days=349 conv_rate=39>,
 <item: nr_days=342 conv_rate=66>,
 <item: nr_days=139 conv_rate=96>,
 <item: nr_days=245 conv_rate=95>,
 <item: nr_days=121 conv_rate=8>,
 <item: nr_days=167 conv_rate=86>,
 <item: nr_days=356 conv_rate=22>,
 <item: nr_days=20 conv_rate=21>,
 <item: nr_days=155 conv_rate=14>,
 <item: nr_days=272 conv_rate=11>]
```

```
<item: nr_days=272 conv_rate=11>]
```

In [58]:
```

#solved the knapsack for the generated test items and deadline of 365 days

solveKnapsackProblemEE(testItems, maxCapacity=365)
```

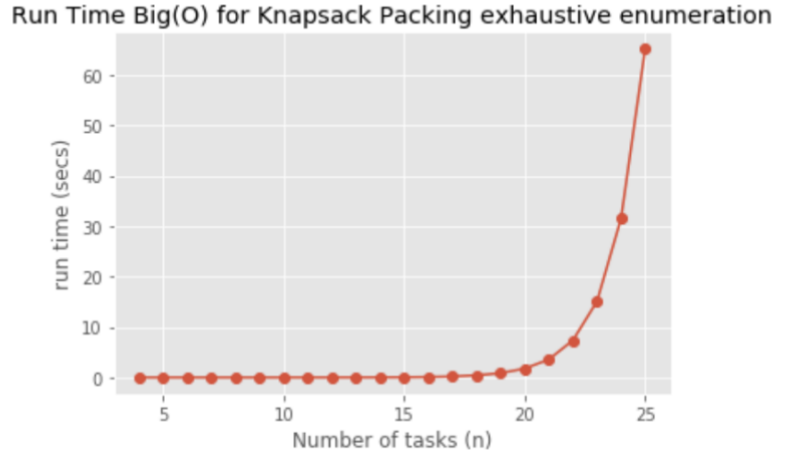
```
bestSolution=  [<item: nr_days=58 conv_rate=66>, <item: nr_days=46 conv_rate=78>, <item: nr_days=126 conv_rate=96>, <
item: nr_days=128 conv_rate=88>]
bestconv_rate =  328
```

Timed Big(O) plot

In [13]:
```
'''
Bellow I created a runtime Big(O) plot for the exhaustive enumeration knapsack, for n in (4,26) in order to see
the runtime in seconds for these values of n
'''


%matplotlib inline
from matplotlib import pyplot as plt
plt.style.use('ggplot')

from timeit import repeat
from functools import partial

nvals = [n for n in range(4,26)]

times = []
for n in nvals:
    testItems=[]
    for _ in range(n):
        conv_rate=rollDie_conv_rate()
        nr_days=rollDie_nr_days()
        testItems.append(Item(nr_days,conv_rate))            # generate a list of n items

    f = partial(solveKnapsackProblemEE,testItems,maxCapacity=70) # make function f() equivalent
                                                #to solveKnapsackProblemEE(testItems,maxCapacity=70)
    time = min(repeat(f,number=3))/3   # record 3 timings to run function f 10x
                                       # take minimum of these 3 timings, and divide by 3
    times.append(time)

plt.xlabel('Number of tasks (n)')
plt.ylabel('run time (secs)')
plt.title('Run Time Big(O) for Knapsack Packing exhaustive enumeration')
plt.plot(nvals, times, 'o-');

```

```
plt.title('Run Time Big(O) for Knapsack Packing exhaustive enumeratio
plt.plot(nvals, times, 'o-');
```

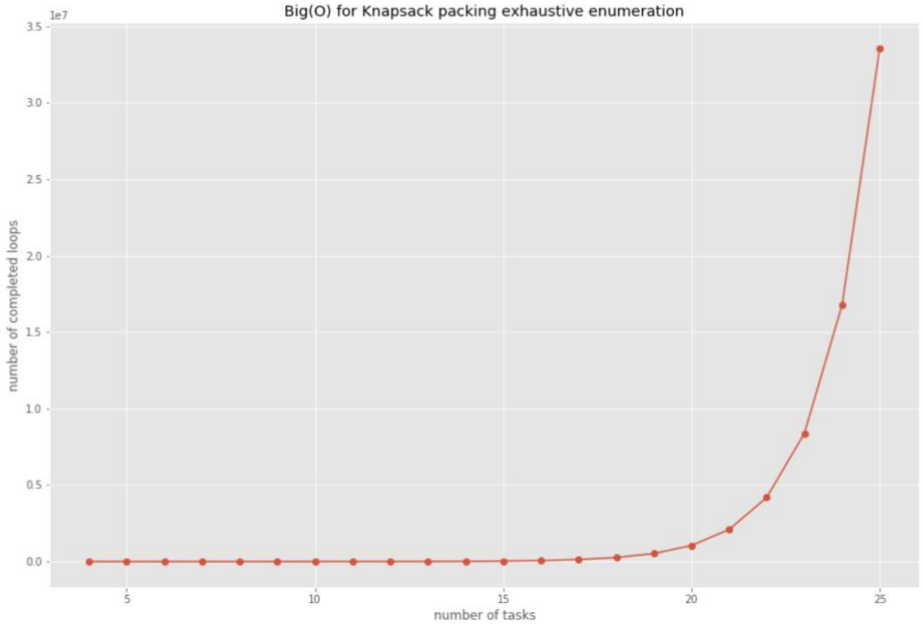
```
bestSolution=  [<item: nr_days=22 conv_rate=87>, <item: nr_days=41 conv_r
bestconv_rate =  245
```



Theoretical Big(O) plot

In [22]:

```
'''
Bellow I created a theoretical Big(O) plot for the exhaustive enumeration knapsack, for n in (4,26) in order to se
the number of completed loops for these values of n
'''


plt.rcParams['figure.figsize'] = [15, 10]

nr_tasks    = [n for n in range(4,26)]   # n = 4,5, 6 , .., 25
loopvals = [2**n for n in nr_tasks]

plt.xlabel('number of tasks')
plt.ylabel('number of completed loops')
plt.title('Big(O) for Knapsack packing exhaustive enumeration')
plt.plot(nvals, loopvals, 'o-');
```



Dynamic programming

geeks for geeks https://www.geeksforgeeks.org/printing-items-01-knapsack/

[2]:

```
'''
the function builds the table K of size max_Capacity+1 x nr_tasks+1
'''

def buildK(max_Capacity,nr_days,conv_rate,nr_tasks):
    K = [[0 for max_Capacity in range(max_Capacity + 1)]
            for i in range(nr_tasks + 1)]

    print("Procces of building matrix K: \n")
    # Build table K[][] in bottom
    # up manner
    for i in range(nr_tasks + 1):
        for j in range(max_Capacity + 1):
            if i == 0 or j == 0:  #base case
                K[i][j] = 0
            elif nr_days[i - 1] <=j:             #maximum of two cases, for weight j:
                                                        # (1) item i included
                                                          # (2) item i not included

                K[i][j] = max(conv_rate[i - 1]
                  + K[i - 1][j - nr_days[i - 1]],
                                K[i - 1][j])
            else:
                K[i][j] = K[i - 1][j]           # If weight of the nth item is
                                                          # more than Knapsack of capacity j,
                                                        # then this item cannot be included
                                                            # in the optimal solution

            print("i= ",i,"j= ",j,"K= ",K)

    print("\n Final value of matrix K: ", K)
    return K

'''
the fuction finds the items that make the best conversion rate
'''


def showItems(max_Capacity,nr_days,conv_rate,name,nr_tasks,res,K):
    j = max_Capacity
    for i in range(nr_tasks, 0, -1):
        if res <= 0:
            break
        # either the result comes from the
        # top (K[i-1][j]) or from (conv_rate[i-1]
        # + K[i-1] [j-nr_days[i-1]]) as in Knapsack
        # table. If it comes from the latter
        # one/ it means the item is included.
        if res == K[i - 1][j]:
            continue
        else:

            # This item is included.
            print(Fore.BLUE+"nr_days= ",nr_days[i - 1],"conversion rate= ",conv_rate[i-1],"name= ",name[i-1])




            # Since this item is included
            # its conv_rate is deducted
            res = res - conv_rate[i - 1]
            j = j - nr_days[i - 1]

            print(Fore.BLACK+ "\n", "conv_rate= ",res, "max_Capacity= ",j,'\n')
'''
```

```
            print(Fore.BLACK+ "\n", "conv_rate= ",res, "max_Capacity= ",j,'\n')
'''
the function bellomax_Capacity takes as input the items and the maximum capacity and prints the best conversion ra
the items needed to obtain it

e.g. for [Item( task 1,2, 4),
             Item( task 2, 3, 5),
             Item( task 3, 7, 6) ]

             and maximum capacity 9 it prints the conversion rate 10 given
             by [<item: task 1 nr_days=2 conv_rate=4>, <item: task 3 nr_days=7 conv_rate=6>]
'''


def printknapSack(max_Capacity, nr_days, conv_rate,name, nr_tasks):

    K=buildK(max_Capacity,nr_days,conv_rate,nr_tasks)

    # stores the result of Knapsack

    res = K[nr_tasks][max_Capacity]

    print(Fore.BLUE+ "\n","maximum conv_rate= ",res ,'\n')

    showItems(max_Capacity,nr_days,conv_rate,name,nr_tasks,res,K)

```

Testing on example

In [71]:

```
#Test on example
name=["naming","telemarketing","social media"]
conv_rate = [ 60, 100, 120 ]
nr_days = [ 1, 2, 3 ]
max_Capacity = 5
nr_tasks = len(conv_rate)

printknapSack(max_Capacity, nr_days, conv_rate,name, nr_tasks)
```

```
Procces of building matrix K:

i=  0 j=  0 K=  [[0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0]]
i=  0 j=  1 K=  [[0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0]]
i=  0 j=  2 K=  [[0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0]]
i=  0 j=  3 K=  [[0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0]]
i=  0 j=  4 K=  [[0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0]]
i=  0 j=  5 K=  [[0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0]]
i=  1 j=  0 K=  [[0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0]]
i=  1 j=  1 K=  [[0, 0, 0, 0, 0, 0], [0, 60, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0]]
i=  1 j=  2 K=  [[0, 0, 0, 0, 0, 0], [0, 60, 60, 0, 0, 0], [0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0]]
i=  1 j=  3 K=  [[0, 0, 0, 0, 0, 0], [0, 60, 60, 60, 0, 0], [0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0]]
i=  1 j=  4 K=  [[0, 0, 0, 0, 0, 0], [0, 60, 60, 60, 60, 0], [0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0]]
i=  1 j=  5 K=  [[0, 0, 0, 0, 0, 0], [0, 60, 60, 60, 60, 60], [0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0]]
i=  2 j=  0 K=  [[0, 0, 0, 0, 0, 0], [0, 60, 60, 60, 60, 60], [0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0]]
i=  2 j=  1 K=  [[0, 0, 0, 0, 0, 0], [0, 60, 60, 60, 60, 60], [0, 60, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0]]
i=  2 j=  2 K=  [[0, 0, 0, 0, 0, 0], [0, 60, 60, 60, 60, 60], [0, 60, 100, 0, 0, 0], [0, 0, 0, 0, 0, 0]]
i=  2 j=  3 K=  [[0, 0, 0, 0, 0, 0], [0, 60, 60, 60, 60, 60], [0, 60, 100, 160, 0, 0], [0, 0, 0, 0, 0, 0]]
i=  2 j=  4 K=  [[0, 0, 0, 0, 0, 0], [0, 60, 60, 60, 60, 60], [0, 60, 100, 160, 160, 0], [0, 0, 0, 0, 0, 0]]
i=  2 j=  5 K=  [[0, 0, 0, 0, 0, 0], [0, 60, 60, 60, 60, 60], [0, 60, 100, 160, 160, 160], [0, 0, 0, 0, 0, 0]]
i=  3 j=  0 K=  [[0, 0, 0, 0, 0, 0], [0, 60, 60, 60, 60, 60], [0, 60, 100, 160, 160, 160], [0, 0, 0, 0, 0, 0]]
i=  3 j=  1 K=  [[0, 0, 0, 0, 0, 0], [0, 60, 60, 60, 60, 60], [0, 60, 100, 160, 160, 160], [0, 60, 0, 0, 0, 0]]
i=  3 j=  2 K=  [[0, 0, 0, 0, 0, 0], [0, 60, 60, 60, 60, 60], [0, 60, 100, 160, 160, 160], [0, 60, 100, 0, 0, 0]]
i=  3 j=  3 K=  [[0, 0, 0, 0, 0, 0], [0, 60, 60, 60, 60, 60], [0, 60, 100, 160, 160, 160], [0, 60, 100, 160, 0, 0]]
i=  3 j=  4 K=  [[0, 0, 0, 0, 0, 0], [0, 60, 60, 60, 60, 60], [0, 60, 100, 160, 160, 160], [0, 60, 100, 160, 180, 0]]
i=  3 j=  5 K=  [[0, 0, 0, 0, 0, 0], [0, 60, 60, 60, 60, 60], [0, 60, 100, 160, 160, 160], [0, 60, 100, 160, 180, 22
0]]

 Final value of matrix K:  [[0, 0, 0, 0, 0, 0], [0, 60, 60, 60, 60, 60], [0, 60, 100, 160, 160, 160], [0, 60, 100, 16
0, 180, 220]]

 maximum conv_rate=  220

nr_days=  3 conversion rate=  120 name=  social media

 conv_rate=  100 max_Capacity=  2

nr_days=  2 conversion rate=  100 name=  telemarketing

 conv_rate=  0 max_Capacity=  0
```

Ignore print statements

Wrote the functions buildK and showItems again to comment the print statements for the bigger dataset

In [3]:

```
'''
the function builds the table K of size max_Capacity + 1 x nr_tasks + 1
'''

def buildK(max_Capacity,nr_days,conv_rate,nr_tasks):
    K = [[0 for max_Capacity in range(max_Capacity + 1)]
            for i in range(nr_tasks + 1)]


    # Build table K[][] in bottom
    # up manner
    for i in range(nr_tasks + 1):
        for max_Capacity in range(max_Capacity + 1):
            if i == 0 or max_Capacity == 0:  #base case
                K[i][max_Capacity] = 0
            elif nr_days[i - 1] <= max_Capacity:           #maximum of two cases:
                                                                # (1) last item included
                                                                  # (2) not included

                K[i][max_Capacity] = max(conv_rate[i - 1]
                  + K[i - 1][max_Capacity - nr_days[i - 1]],
                                K[i - 1][max_Capacity])
            else:
                K[i][max_Capacity] = K[i - 1][max_Capacity]           # If weight of the nth item is
                                                                    # more than Knapsack of capacity W,
                                                                   # then this item cannot be included
                                                                      # in the optimal solution

            #print("i= ",i,"max_Capacity= ",max_Capacity,"K= ",K)


    return K

'''
the fuction finds the items that make the best conversion rate
'''


def showItems(max_Capacity,nr_days,conv_rate,name,nr_tasks,res,K):
    max_Capacity = max_Capacity
    for i in range(nr_tasks, 0, -1):
        if res <= 0:
            break
        # either the result comes from the
        # top (K[i-1][max_Capacity]) or from (conv_rate[i-1]
        # + K[i-1] [max_Capacity-nr_days[i-1]]) as in Knapsack
        # table. If it comes from the latter
        # one/ it means the item is included.
        if res == K[i - 1][max_Capacity]:
            continue
        else:

            # This item is included.
            print("nr_days= ",nr_days[i - 1],"conversion rate= ",conv_rate[i-1], "name= ",name[i-1])




            # Since this item is included
            # its conv_rate is deducted
            res = res - conv_rate[i - 1]
            max_Capacity = max_Capacity - nr_days[i - 1]

            #print(Fore.BLACK+ "\n", "conv_rate= ",res, "max_Capacity= ", max_Capacity,'\n')
```

Generate csv file

Solve the problem

In [8]:

```
'''
generated a csv file of 26 items with 3 columns (name of task, tasks durations and conversion rate) using the prev
defined generators
'''

import pandas as pd

# Creating an empty Dataframe with column names only
dfnew = pd.DataFrame(columns=['Name','Conversion rate','Number of days'])

for i in range(26):
    conv_rate= rollDie_conv_rate()
    nr_days= rollDie_nr_days()
    names=str(i)
    names= "task "+names
    dfnew = dfnew.append({'Name':names,'Conversion rate': conv_rate,'Number of days': nr_days}, ignore_index=True)

# Converts data frame into a csv file without an index column
dfnew.to_csv("problemdata.csv",index=False)


dfnew
```

Out[8]:

	Name	Conversion rate	Number of days
0	task 0	75	49
1	task 1	37	336
2	task 2	50	35
3	task 3	11	336
4	task 4	100	131
5	task 5	31	343
6	task 6	97	109
7	task 7	9	114
8	task 8	74	239
9	task 9	92	2
10	task 10	87	15
11	task 11	67	209
12	task 12	9	118
13	task 13	12	323

User interface

In [11]:

```
''' User interface - inputting the csv file and the deadline  to see as a result the maximum conversion rate
that can be achieved and the tasks needed for that'''

data = input("Specify csv file name - ")
deadline=input("Specify maximum number of days for the marketing campaign -deadline- ")
deadline=int(deadline)

''' Read from the csv file and convert the inputs to the required format - made each column a list  '''

data1=pd.read_csv(data)
name=data1['Name'].to_list()
conv_rate=data1['Conversion rate'].to_list()
nr_days=data1['Number of days'].to_list()
nr_tasks=len(nr_days)

#solved the knapsack for the generated test items and deadline specified by the user

printknapSack(deadline, nr_days, conv_rate,name, nr_tasks)
```

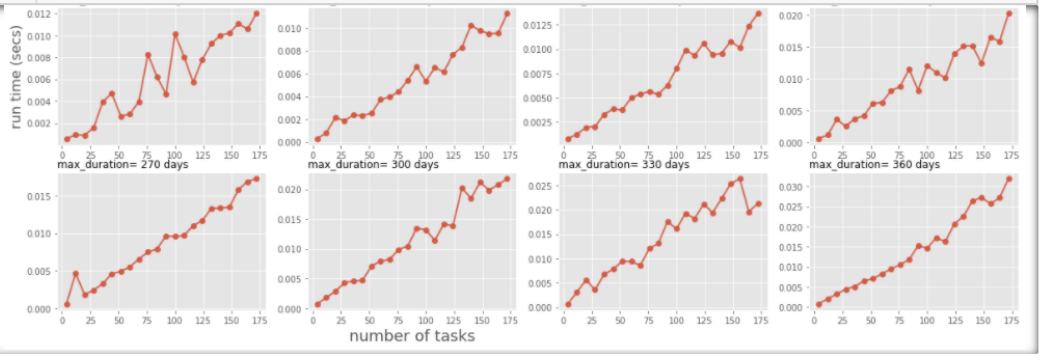
```
Specify csv file name - problemdata.csv
Specify maximum number of days for the marketing campaign -deadline- 365

 maximum conv_rate=  614

nr_days=  19 conversion rate=  71 name=  task 18
nr_days=  36 conversion rate=  92 name=  task 14
nr_days=  15 conversion rate=  87 name=  task 10
nr_days=  2 conversion rate=  92 name=  task 9
nr_days=  109 conversion rate=  97 name=  task 6
nr_days=  131 conversion rate=  100 name=  task 4
nr_days=  49 conversion rate=  75 name=  task 0
```

Timed Big(O) plot

Maximum duration as constant

In [12]:

```
'''
Bellow I created a timed Big(O) plot with multiple sublots for the dynamic knapsack, for nr of tasks in (4,180,8)
max duration in (30,361,30) in order to see the runtime in seconds for these values of nr of tasks and
max duration - I took max duration as a constants for every subplot
'''
from timeit import repeat
from functools import partial


%matplotlib inline
from matplotlib import pyplot as plt
plt.style.use('ggplot')

plt.figure(figsize=(20,10))

# multiple line plot
num=0

for max_Capacity in range(30,361,30):
        num+=1
        nr_tasks=[n for n in range(4,180,8)]

        times=[]

        for n in nr_tasks:
            conv_rate=[]
            nr_days=[]
            name=[]
            for i in range(n):
                conv_rate.append(rollDie_conv_rate())
                nr_days.append(rollDie_nr_days())
                names=str(i)
                names= "task "+names
                name.append(names)
                                                # generate a list of n items

            f = partial(printknapSack,max_Capacity, nr_days, conv_rate,name, n) # make function f() equivalent
                                              #to solveKnapsackProblemEE(testItems,maxCapacity,nr_days,
                                                #conv_rate,name,n)
            time = min(repeat(f,number=3))/3   # record 3 timings to run function f 10x
                                             # take minimum of these 3 timings, and divide by 10
            times.append(time)



        plt.subplot(3,4, num)

        # Plot the lineplot
        plt.plot(nr_tasks, times, 'o-', linewidth=1.9, alpha=0.9)


    # Add title
        title=str(max_Capacity)
        title1="max_duration= " +title +" days"
        plt.title(title1, loc='left', fontsize=12, fontweight=0 )

        if num==10:
            plt.xlabel('number of tasks',fontsize=18)

        if num==5:
            plt.ylabel('run time (secs)',fontsize=16)

# general title
plt.suptitle("Big(O) for dynamic programming knapsack (maximum campaign duration as constant)", fontsize=13, fontw

plt.savefig("figure1.png") # save as png
```

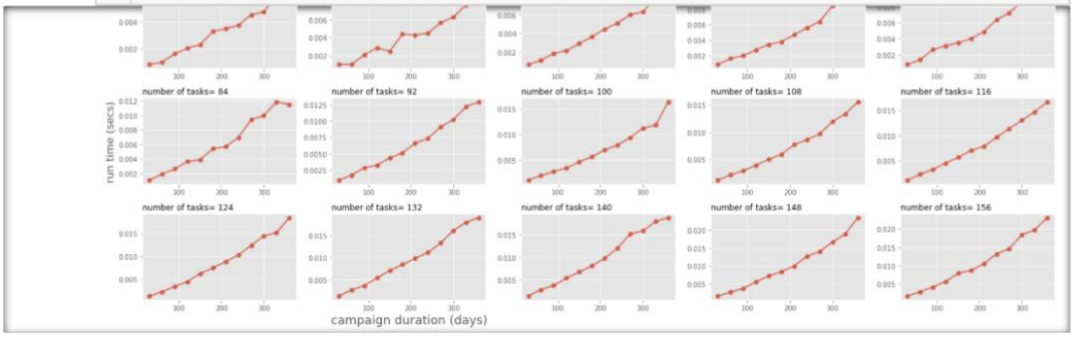


Timed Big(O) Plot

Number of tasks as constant

In [13]:

```
'''
Bellow I created a timed Big(O) plot with multiple sublots for the dynamic knapsack, for nr of tasks in (4,163,8)
max duration in (30,361,30) in order to see the runtime in seconds for these values of nr of tasks and
max duration - I took nr of tasks as a constants for every subplot
'''

plt.figure(figsize=(20,10))

import matplotlib.pyplot as plt


# multiple line plot
num=0

for nr_tasks in range(4,163,8):
        num+=1
        max_Capacity=[n for n in range(30,361,30)]

        times=[]

        for n in max_Capacity:
            conv_rate=[]
            nr_days=[]
            name=[]
            for i in range(nr_tasks):
                conv_rate.append(rollDie_conv_rate())
                nr_days.append(rollDie_nr_days())
                names=str(i)
                names= "task "+names
                name.append(names)                    # generate a list of n items

            f = partial(printknapSack,n, nr_days, conv_rate,name, nr_tasks) # make function f() equivalent
                                                #to solveKnapsackProblemEE(testItems,n,nr_days,conv_rate,name,nr_t
            time = min(repeat(f,number=3))/3   # record 3 timings to run function f 10x
                                            # take minimum of these 3 timings, and divide by 10
            times.append(time)



        plt.subplot(4,5, num)

        plt.tight_layout()

        # Plot the lineplot
        plt.plot(max_Capacity, times, 'o-', linewidth=1.9, alpha=0.9)


    # Add title
        title=str(nr_tasks)
        titel="number of tasks= " +title
        plt.title(titel, loc='left', fontsize=12, fontweight=0 )

        if num==17:
            plt.xlabel('campaign duration (days)',fontsize=18)

        if num==11:
            plt.ylabel('run time (secs)',fontsize=16)

# general title
plt.suptitle("Big(O) for dynamic programming knapsack (number of tasks as constant)", fontsize=13, fontweight=0, c

plt.savefig("figure2.png") # save as png
```

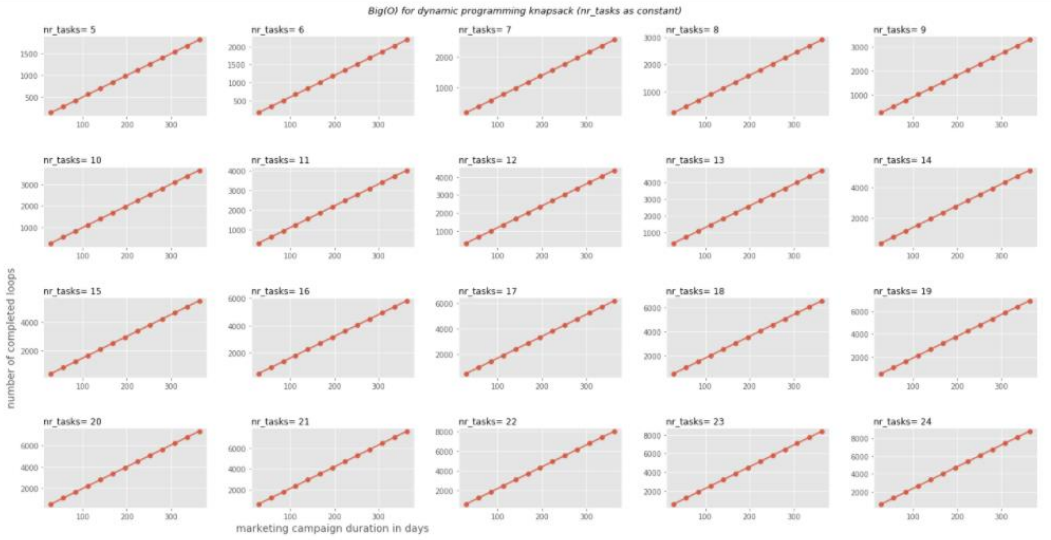


Theoretical Big(O) plot

Number of tasks as constant

In [15]:

```
'''
Bellow I created a theoretical Big(O) plot with multiple sublots for the dynamic knapsack, for nr of
tasks in (4,26) and max duration in (28,340,28) in order to see the number of counted loops for these
values of nr of tasks and max duration - I took nr of tasks as a constants for every subplot
'''

plt.figure(figsize=(20,10))


# multiple line plot
num=0

for nr_tasks in range(5,25):
    num+=1
    max_Capacity=[n for n in range(28,366,28)]

    loopvals= [nr_tasks * n for n in max_Capacity]


    plt.subplot(4,5, num)



    plt.tight_layout()

        # Plot the lineplot
    plt.plot(max_Capacity, loopvals, 'o-', linewidth=1.9, alpha=0.9)


    # Add title
    title=str(nr_tasks)
    title1="nr_tasks= " +title
    plt.title(title1, loc='left', fontsize=12, fontweight=0 )

    if num==17:
        plt.xlabel('marketing campaign duration in days',fontsize=14)
    if num==11:
         plt.ylabel('number of completed loops',fontsize=14)

# general title
plt.suptitle("Big(O) for dynamic programming knapsack (nr_tasks as constant)", fontsize=13, fontweight=0, color='b

plt.savefig("figure3.png") # save as png
```



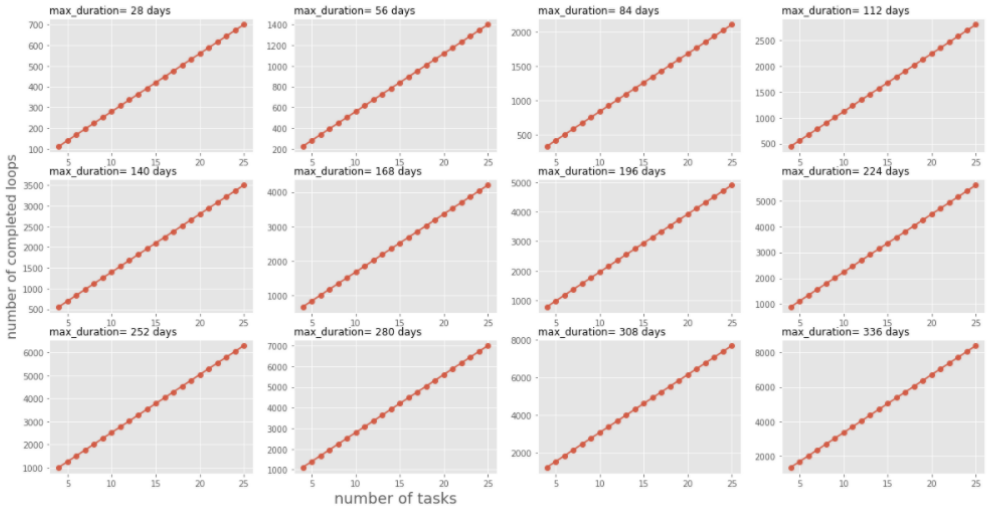
Theoretical Big(O) Plot

Maximum duration as constant

In [16]:

```
'''
Bellow I created a theorectical Big(O) plot with multiple sublots for the dynamic knapsack, for nr of
tasks in (4,26) and max duration in (28,340,28) in order to see the number of counted loops for these
values of nr of tasks and max duration - I took max duration as a constants for every subplot
'''

plt.figure(figsize=(20,10))


# multiple line plot
num=0

for max_Capacity in range(28,340,28):
    num+=1
    nr_tasks=[n for n in range(4,26)]

    loopvals= [max_Capacity * n for n in nr_tasks]


    plt.subplot(3,4, num)
        # Plot the lineplot
    plt.plot(nr_tasks, loopvals, 'o-', linewidth=1.9, alpha=0.9)


    # Add title
    title=str(max_Capacity)
    title1="max_duration= " +title +" days"
    plt.title(title1, loc='left', fontsize=12, fontweight=0 )

    if num==10:
        plt.xlabel('number of tasks',fontsize=18)

    if num==5:
        plt.ylabel('number of completed loops',fontsize=16)

# general title
plt.suptitle("Big(O) for dynamic programming knapsack (maximum campaign duration as constant)", fontsize=13, fontwe

plt.savefig("figure4.png") # save as png
```

Big(O) for dynamic programming knapsack (maximum campaign duration as constant)



7. References

1. How long should a marketing campaign last? [Internet]. Red-fern.co.uk. 2021 [cited 12 January 2021]. Available from: https://www.red-fern.co.uk/blog/insights/how-long-should-a-marketing-campaign-last.html

2. Marketing Campaigns | Marketing MO [Internet]. Marketing MO. 2021 [cited 15 December 2020]. Available from: http://www.marketingmo.com/strategic-planning/marketing-campaigns/

3. 0-1 Knapsack Problem | DP-10 - GeeksforGeeks [Internet]. GeeksforGeeks. 2021 [cited 8 December 2020]. Available from: https://www.geeksforgeeks.org/0-1-knapsack-problem-dp-10/

4. Printing Items in 0/1 Knapsack - GeeksforGeeks [Internet]. GeeksforGeeks. 2021 [cited 14 January 2021]. Available from: https://www.geeksforgeeks.org/printing-items-01-knapsack/