Combinatorics
Mathematical Programming with Python

https://people.sc.fsu.edu/~jburkardt/classes/mpp_2023/combinatorics/combinatorics.pdf

---

Seeking the shortest round trip connecting state capitals (lower 48 only).

Combinatorics

- *Combinatorics asks simple questions about finite collections of objects;*
- *Questions: number, ordering, arrangement, grouping;*
- *Fundamental objects: compositions, permutations, sample sequences, subsets, strings, trees, words;*
- *Tasks: enumeration, next element, random element, ranking, unranking;*
- *Optimization over all permutations (traveling salesperson)*
- *Optimization over all subsets (knapsack)*
- *Finding a subset of numbers with a given sum (SubsetSum)*
- *Likelihood of begin dealt five cards in a sequence (Poker)*
- *How many of each element needed to match a value (McNuggets)*
- *Dividing a set of valuable items equally among a group (Partition Problem).*
- *Reconstructing a sequence given many fragments (Partial Digest)*
- *Distance between two DNA strands (Levenshtein)*

1 Permutations: Assigning Order to Elements

If we suppose we have a set of n labeled objects, then a permutation of these objects assigns a rank or order to each object. For instance, our objects might be the seven spectral colors, represented by the initial letters

```
colors = np.array ( ['R','O','Y','G','B','I','V'] )
```

If this is the natural order, then we can list the ranks of the letters as follows: then one permutation is

```
0 1 2 3 4 5 6
R O Y G B I V
```

If we wish to list the colors in alphabetical order, then we must permute them. Here, we list the alphabetized letters, and their permuted indices:

```
4 3 5 1 0 6 2
B G I O R V Y
```

Even when we are working with nonnumeric objects such as the names of colors, it is convenient to identify them with their indices on a list, and work strictly with the numeric indices. Then the original list is represented by the values $[0, 1, 2, ..., n-1]$ and a permutation is presented simply as a scrambling of these indices. Permutations are often represented by the letter π im mathematical text, or p in programming. Given a list of values in proper order, and a permutation p, we can print the permuted list as follows:

```
colors = np.array ( ['R','O','Y','G','B','I','V'] )
p = (4,3,5,1,0,6,2,1)
print ( colors[p] )
B G I O R V Y
```

In some cases (as in heuristic approaches to the traveling salesperson problem), we may want to consider random permutations. To do so, we need to initialize the random number generator first. Here is an example in which we randomly permute the colors array:

```
from numpy.random import default_rng
rng = default_rng ( )
colors = np.array ( ['R','O','Y','G','B','I','V'] )
p = rng.permutation ( 7 )
print ( p )
[5 1 0 4 6 2 3]
print ( colors[p] )
['I' 'O' 'R' 'B' 'V' 'Y' 'G']
```

Given a list of n distinct objects, how many permutations are there? To define a permutation, we have to write down n distinct indices. We have n choices for index 0, then $n-1$ for index 1, and so on, until we have just 1 choice for index $n-1$. This means that the number of possible permutations is

$$n \times n-1 \times n-1 \times ... \times 2 \times 1 \equiv n!$$

a quantity known as n *factorial*. The factorial is a vital function in combinatorics, but it comes with some problems.

2 The Factorial Function

The values of the factorial function grow large very quickly.

n	n!
0!	1
1!	1
2!	2
3!	6
4!	24
5!	120
...	...
10!	3,628,800
...	...
20!	2,432,902,008,176,640,000
...	...
50!	3.04140932010^{64}
...	...
100!	9.33262154410^{157}

Mathematically, such values are perfectly reasonable; computationally, they can sometimes be difficult to use when accurate answers are desired. We will return to this problem shortly, when talk about counting the number of combinations. For now, we note that there are at least three functions available to compute a factorial using Python:

1. `math.factorial(n)`;
2. `numpy.math.factorial(n)`;
3. `scipy.special.factorial(n)`, where n can be an integer or a numpy integer array.

3 Listing the "Next" Permutation

Suppose you wanted to list every permutation of n objects. For instance, if $n = 3$, we want to be able to create, perhaps one at a time, the following sequence of permutations:

```
0:  0, 1, 2
1:  0, 2, 1
2:  1, 0, 2
3:  1, 2, 0
4:  2, 0, 1
5:  2, 1, 0
```

You might notice that this list is very orderly; it is, essentially, an "alphabetical" list of the index vectors. Such an ordering is more properly called "lexicographic", or "dictionary order". For small values of n, it's not too hard to generate such a list, but when n is large, we will be asking a computer to take over. How can we teach it to start at the first permutation, and then repeatedly generate the next one, until we reach the end?

We agree that the 0-th permutation of n objects can be represented by the array `[0,1,2,...,n-1]`. Now suppose that we have the k-th permutation array, and we want to figure out the $k+1$-th array, using the lexicographic order. To do so, we need to know which digits to rearrange in the array. Here is an algorithm for doing so, as applied to the permutation vector `p=[1,3,2,0]` which has `rank=11`:

1. If the input value of `rank` is -1, this is the first call. Return immediately with

   ```
   p = [0,1,...,n-1] = np.arange ( n )
   rank = 0
   ```

2. Otherwise, seek the highest index `i` for which `p[i]` is smaller than `p[i+1]`.

```
p = [1,3,2,0]
p[2] > p[3]
p[1] > p[2]
p[0] < p[1], so i = 0
```

3. If no such `i` could be found, return a null permutation, we have completed the permutation list.
4. Find highest index `j > i` such that `p[i] < p[j]`.

```
p = [1,3,2,0]
p[0] is 1
p[3] is not greater than p[0]
p[2] is greater than p[0], so j = 2
```

5. Interchange elements `i` and `j`.

```
p = [1,3,2,0]
i = 0, j = 2
p = [2,3,1,0] after interchange
```

6. Reverse the elements between indices `i+1 (=1)` and `n-1 (=3)`.

```
p = [2,0,1,3] after reversal
rank = rank + 1
This is permutation #12
```

It is worth the effort to try to write the corresponding Python function `p,rank=next_permutation(p,rank,n)`, and the exercises offer you the challenge of doing so. Otherwise, you can use the code that is provided on the website.

Applications in which it is useful to be able to generate permutations automatically include the traveling salesperson problem, and the computation of the determinant of a matrix.

4 The Matrix Determinant using Permutations

Let A be an $n \times n$ matrix. The determinant, written $\det(A)$, is defined by

$$\det(A) = \sum_{\pi \in S_n} \operatorname{sgn}(\pi) \prod_{i=0}^{n} a_{i,\pi_i}$$

where S_n is the set of all permutations on n objects, and $\operatorname{sgn}\pi$ is the sign of the permutation, defined by

$$\operatorname{sgn}(\pi) = \begin{cases} +1, \pi \text{ is the product of an even number of transpositions;} \\ -1, \pi \text{ is the product of an odd number of transpositions.} \end{cases}$$

You are probably more comfortable seeing an example, such as:

$$A = \begin{pmatrix} a & b & c \\ d & e & f \\ g & h & i \end{pmatrix}$$

for which we have

$$\det(A) = aei + bfg + cdh - afh - bdi - ceg$$

Clearly, we need a way to compute the sign of a permutation p before we can write a code to compute determinants! Here's an outline of how such a code would work:

```
Make a copy of p called q
Initialize the sign s = 1
For each index i
  while q[i] is not equal to i
    s = -s
    swap q[i] and q[j]  (carefully!)
```

Here is an example of how this procedure would compute the sign of the permutation $p = [4, 2, 3, 1, 0]$:

```
s = 1
4 2 3 1 0  <-- 4 in position 0, swap with 0 in position 4, s = -1
0 2 3 1 4  <-- 0 in position 0, OK
0 2 3 1 4  <-- 2 in position 1, swap with 3 in position 2, s = +1
0 3 2 1 4  <-- 3 in position 1, swap with 1 in position 3, s = -1
0 1 2 3 4  <-- 1 in position 1, 2 in position 2, 3 in position 3, 4 in position 4, DONE
so s = sign(p) is -1.
```

The exercises ask you to write a function `s = permutation_sign(p)`.

5 Computing Matrix Determinants

Now if we have a matrix A, and the functions `permutation_next()` and `permutation_sign()`, we can compute the determinant of the matrix. Here is how the code might go:

```
def matrix_determinant ( A )

  n = A.shape
  d = 1.0
  rank = -1
  while ( True ):
    p, rank = permutation_next ( p, rank, n )
    if ( rank == -1 )
      break
    s = permutation_sign ( p )
    for i in range ( 0, n ):
      s = s * A[i,p[i]]
    d = d + s

  return d
```

The exercises ask you to create this function, to test it on some random matrices, and compare with the value returned by `scipy.linalg.det()`.

By the way, matrix determinants are usually *not* computed this way. There is a much more efficient method that is a by-product of Gaussian elimination. However, the permutation approach is not useless. There is a function related to the determinant called the "matrix permanent", whose definition is similar to the determinant, except that the factor of the permutation sign is removed. It turns out that Gaussian elimination cannot help in this case, and so the permutation approach is of use.

6 Traveling on a Budget: The Traveling Salesperson Problem

A salesperson is assigned the task of visiting n cities, starting and ending at one city. The traveler is given a map that reports the distance between each pair of cities. Because of a tight budget, the traveler is required to determine the round trip that requires the least total distance. How is this to be done?

For computational convenience, let us replace each city's name by an identification number. Since we will be working with Python, these numbers will run from 0 to $n-1$. For the traveler's round trip. the itinerary is simply a list of the cities in the order in which they are visited. The first and last items on the list will be the same. If we drop the last item, the remaining list represents a permutation of the n cities. Replacing the city names by their identification numbers, we now have a list p of n distinct integers between 0 and $n-1$, in other words, a permutation.

Suppose we have been given an $n \times n$ array called `distance` such that the distance between cities i and j is `distance[i,j]`. Then, given a permutation p that represents an itinerary, the mileage of this proposed trip from `p[0]` to `p[1]`, then from `p[1]` to `p[2]`, and so on until we return to `p[0]`, is:

```
mileage = distance[p[0],p[1]] + distance[p[1],p[2]] + ... + distance[p[n-1],p[0]]
```

7 Brute Force Solution of the Traveling Salesperson Problem

Given that we can automatically generate every possible itinerary for the traveling salesperson, we can clearly find an itinerary with the shortest total mileage, using a brute force approach:

Brute Force Solution of Traveling Salesperson Problem

Given a list of n cities, and a table of distances between every pair of cities, generate every possible itinerary as a permutation, compute the associated mileage, and report an itinerary that achieves the lowest mileage.

Our brute force approach guarantees that we find a best solution. It turns out that there will be at least n such "best" solutions, unless we specify a particular starting city. Even then, it is possible to have multiple solutions. But having multiple solutions is not an issue; we will be happy if we can find any solution that minimizes the total distance.

Here is the distance table for a simple case involving five cities

```
    0  1  2  3  4
  +--------------
0 | 0  3  4  2  9
1 | 3  0  4  6  3
2 | 4  4  0  5  8
3 | 2  6  5  0  6
4 | 9  3  8  6  0
```

The brute force approach discovers the itinerary `p=[0,3,4,1,2]` with the following mileage:

```
From  To  Distance

  0   3         2
  3   4         6
  4   1         3
  1   2         4
  2   0         4
                --
       Total:  19
```

So why is there no cheering for our solution? Recall that the number of permutations of n objects is $n!$, and suppose we wanted to solve a problem involving a visit to each of the 50 state capitals in the United States?

Look above at the discussion of the factorial function, read, very slowly, the value of 50!, and think about whether you will live long enough to list every permutation of 50 cities. The brute force approach seems to be able to handle 10 cities, but even 20 cities is not a reasonable problem size for this approach. Although there are methods that have found the exact solution for the 50 state capital problem, we will concentrate on simple approaches that give us a hope of approximating a good solution.

8 Traveling Salesperson Heuristics

Given that larger TSP problems are extremely difficult to solve, many suggestions have been made for coming up with approximate solutions. In each case we start with a basic itinerary `0-->?-->0` and try to fill in the blanks.

- *Nearest neighbor*: From your current location, add a trip to the nearest city that has not been visited yet. Repeat until all cities have been reached, and go home.
- *Greedy algorithm*: Find the shortest unused city-to-city trip which does not create a "short circuit" and which does not involve a city which already has a trip in and a trip out; Add this trip to your itinerary. Keep doing this until your itinerary includes every city.
- *Insertion*: Choose an unvisited city k at random. For every pair of consecutive cities i and j already in your itinerary, compute the length of the revised itinerary $i- > k- > j$. Choose to insert k between the cities i and j which minimize this change in trip length.
- *Transpose*: Start with an itinerary chosen at random. Pick two distinct non-neighboring cities i and j. Reverse the order of all the cities in the itinerary between i and j. If this reduces the total length, keep the change. Keep trying.

9 Nearest Neighbor Solution of Traveling Salesperson Problem

When faced with a problem that is too large for the brute force approach, we may have to give up our guarantee of finding the minimal solution. Now we need a some criterion that we believe will take us close to a minimal solution. An obvious idea is to build the itinerary by picking a starting city at random, and for each next leg of the itinerary, traveling to the nearest unvisited city. By always choosing the shortest unused route, it is hoped that the resulting round trip will come close to being minimal.

```
pick i at random and set p[0] = i

while some cities are still unvisited
  dj = inf
  j = -1
  for all cities k
    if k is not in p
      if distance[i,k] < dj
        j = k
        dj = distance[i,k]
  Set next entry of p to j
```

Here is a distance table for a nine city tour:

```
  0    4  100  100  100  100    7  100  100
  4    0    9  100  100  100   11   20  100
100    9    0    6    2  100  100  100  100
100  100    6    0   10    5  100  100  100
100  100    2   10    0   15  100    1    5
100  100  100    5   15    0  100  100   12
  7   11  100  100  100  100    0    1  100
```

```
 100   20  100  100    1  100    1    0    3
 100  100  100  100    5   12  100    3    0
```

It turns out that the brute force method can still handle this size problem, and returns a minimal tour distance of 50. In this case, the nearest neighbor method also finds a tour of the minimal distance. In general, we will not be so lucky, but we will be happy to have an approximate answer.

The traveling salesperson problem is an abstract model which turns out to have hundreds of real-life applications, especially for transportation, delivery, circuit design, and networking. There are many powerful algorithms for getting very good approximations to the minimal solution, even when the number of cities is in the thousands or millions.

10 Subsets: Selecting Only Some Elements

Often, a combinatorial task starts by saying "pick a subset of a given set S of n objects". Sometimes, to analyze such a task, we actually have to methodically consider every possible such subset, starting with the empty set ϕ and finishing with the full set S. If we are going to do computations, we need a simple way to plod through this list one at a time.

The natural way to represent a subset of n items is with a choice vector x of length n, for which x_i is 1 if item i is to be included in a given subset. Interestingly, such binary vectors of length n are essentially representing the integers from 0 to $2^n - 1$, For instance, for $n = 3$, we need to generate 8 vectors. In the natural way to do this, the order in which we generate each vector is also the integer corresponding to the binary representation:

```
order   vector       expansion
 #0: ( 0, 0, 0 ) = 0*4 + 0*2 + 0*1 = 0
 #1: ( 0, 0, 1 ) = 0*4 + 0*2 + 1*1 = 1
 #2: ( 0, 1, 0 ) = 0*4 + 1*2 + 0*1 = 2
 #3: ( 0, 1, 1 ) = 0*4 + 1*2 + 1*1 = 3
 #4: ( 1, 0, 0 ) = 1*4 + 0*2 + 0*1 = 4
 #5: ( 1, 0, 1 ) = 1*4 + 0*2 + 1*1 = 5
 #6: ( 1, 1, 0 ) = 1*4 + 1*2 + 0*1 = 6
 #7: ( 1, 1, 1 ) = 1*4 + 1*2 + 1*1 = 7
```

How can we generate these values computationally? We can imagine creating a function `next_binary_vector(x)`. It takes as input the current `x`, modifies it by adding 1, and returns the result as the new value of `x`. We initialize the code with the zero vector, call it repeatedly to get more `x` values, and stop when we get back to the zero vector.

Well, that's how the function looks like from the outside. What's the magic that happens inside the function? To carry this out using our vector representation, we start at the rightmost entry of `x`. If it's a 0, we change it to a 1 and exit. Otherwise, we change it to a 0, and move one position to the left if possible. If we can't move left, then we have just reset `x` to all 0 values, and our list is completed.

```
def binary_vector_next ( x ):

  i = len ( x )  # start just past the last entry

  while ( 0 < i ):
    i = i - 1
    if ( x[i] == 0 ):
      x[i] = 1
      break
    else
```

```
    x[i] = 0
  return x
```

To demonstrate, let's use this function to print out the list of choice vectors when $n = 3$:

```
  import numpy as np

  n = 3
  x = np.zeros ( n )
  while ( True )
    print ( x )
    x = binary_vector_next ( x ):
    if ( all ( x == 0 ):
      break
```

You might have other ways of writing this loop, but as written, the loop correctly generates and prints the vectors in the natural order, and terminates as soon as a zero vector is returned.

We will see several instances shortly in which `next_binary_vector()` will be useful.

11 Exact Change Only!

Suppose we are trying to pay a bill at the checkout counter, and we have an ample supply of every coin. After some fumbling around, we manage to come up with a pile of coins that matches the bill exactly. This process is known as the *change making problem:*

The Change Making Problem

Given an unlimited supply of coins, which are minted in n denominations, with the i-th denomination having value v_i. Given a bill of size B, is there a collection of coins which matches this bill exactly? If so, how many different ways are there to do this? What combination involves the fewest number of coins?

Obviously, if there is a "penny" coin worth 1 cent, then there is always a solution, although not one that anyone would like. Generally, there will be many solutions. A brute force approach could be tried, although it's not clear right away how to march through all the possibilities. What most people use is a greedy algorithm, which we will now explore.

12 Greedy Algorithm for Change Making

Let's suppose we are dealing with US currency, and we only have the more common coins of denominations 1, 5, 10 and 25 cents, and no coins or bills of higher value. Let's suppose we need to pay a bill of $1.37. A "greedy" algorithm would first try to use as many quarters as possible, then revert to dimes, followed by nickels and pennies. The process might be something like this:

```
Owe  Coins   Pay
     Q D N P
---  ------- ---
137  0 0 0 0   0
 12  5 0 0 0 125
  2  5 1 0 0 135
  0  5 1 0 1 137
```

So we pay our bill with 5 quarters, a dime, and 2 pennies, for a total of 8 coins. While there are many other combinations of coins corresponding to the bill, this arrangement is optimal in the sense that it minimizes the number of coins.

The greedy algorithm will find a solution, and in fact the optimal solution, as long as the denominations of the available coins have the "canonical" property. For a noncanonical set of coins, the greedy algorithm might not minimize the number of coins, and for a set that does not include a 1 cent coin, the greedy algorithm might fail to find a coin combination that matches a desired sum even if such a solution exists. You are encouraged to think about this question and come up with some examples.

The exercises ask you write a Python code `change_greedy()` to produce a collection of coins that match a given bill.

13 Diophantine Algorithm for Change Making

A diophantine equation has the form

$$c_1x_1 + c_2x_2 + ... + c_nx_n = b$$

where the coefficient vector c and right hand side b are known integer values, and a solution vector of integers x is sought.

For the change making problem, we take the c vector to contain the coin values, the b value to be the desired sum, and the x values to be the number of coins of each type. Naturally, we also impose the condition that every x value must be nonnegative.

We can use the following function so solve our coin problems:

```
x = diophantine_nonnegative ( c, b )
```

where `c` and `b` are the coefficients and right hand side, and the output `x` is an $sols \times n$ array of solutions, with each row containing an assignment for the number of coins. Of course, there might be no solutions, in which case we are getting back an empty array.

Suppose we want to form the sum of 46 cents using pennies, nickels, dimes and quarters. We set

```
  c = np.array ( [ 1, 5, 10, 25 ] )
  b = 46
  x = diophantine_solve ( c, b )
```

and get back a 39×4 array `x` of solutions whose first rows are:

```
[[46.  0.  0.  0.]
 [41.  1.  0.  0.]
 [36.  2.  0.  0.]
 [36.  0.  1.  0.]
 [31.  3.  0.  0.] ...
```

14 Dynamic Algorithm for Change Making

Another version of the change making problem asks simply: *what is the smallest number of coins I need in order to form a given sum?*. Just by convention, if it is not actually possible to form the sum, we will return the answer of ∞, or, as Python would say, `np.inf`.

As usual, let v be the vector of coin values, and b be the value we are trying to form. Now we create a vector a of length $b + 1$. For i from 0 up to b, we wish to compute $a[i]$ as the minimum number of coins needed to form the sum i. We can initialize all entries of a to `np.inf`. Then, if coin i has the value v, we can set `a[v]=1`. How do we update the rest of the table?

Suppose that we have correctly computed all the values of `a` up to `a[i-1]`, and now we want to compute `a[i]`. Suppose we have a coin of value c, and suppose we know `a[i-c]`, the minimum number of coins necessary to form $i - c$. Then one way to form the sum i is take the coins that form $i - c$, add the coin of value c, which would require a total of `a[i-c]+1` coins. We replace the previous value of `a[i]` by `a[i-c]+1` if this is less. Making this check for every coin allows us to get the correct value for `a[i]`, and we can move on to compute the next entry of `a`.

This idea is hard to see at first, but let's take an example. Suppose we are trying to compute `a[30]`, the smallest number of coins needed to make a sum of 30. This value was initialized to `np.inf`. Assume we have already worked out the value of `a` for indices 0 through 29, which are:

```
1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30
1  2  3  4  1  2  3  4  5  1  2  3  4  5  2  3  4  5  6  2  3  4  5  6  1  2  3  4  5 Inf
```

So now we think about how 30 can be made by adding one coin to a lower sum:

```
30 = old + coin  coins(old) + 1 = coins(30)
--   ---   ----  ---------- ---   ---------
30 =  29 +    1       a[29] + 1 = 6
30 =  25 +    5       a[25] + 1 = 2
30 =  20 +   10       a[20] + 1 = 3
30 =   5 +   25        a[5] + 1 = 2
```

And so we can conclude that the minimum number of coins needed to make a sum of 30 is 2.

A sketch of a corresponding program would be:

```
def change_dynamic ( ? ):
  Set a to an array of length b+1
  Initialize all entries of a to infinity
  For coin i of value v, set a[v] = 1
  for each sum s from 0 to b
    for each coin i of value v
      if 0 < s - v
        if a[s-v]+1 < a[s] then a[s] = a[s-v] + 1
```

The exercises ask you to write and test a corresponding Python program.

15 Polynomial Algorithm for Change Making

Our answers to the change making problem ignore the order in which the coins are arranged. Thus, giving a quarter plus a dime is regarded as the same solution as giving a dime and then a quarter. If we are willing to distinguish solutions based on the ordering of the coins, then we can look at a remarkable approach to counting and representing the solutions of our change making problem. Somewhat unbelievably, this method involves polynomials.

Assuming that we are using the four smallest US coins, consider the polynomial:

$$p(x) = x^1 + x^5 + x^{10} + x^{25}$$

and yawn while I tell you that the coefficients of this polynomial encapsulate the information that there is 1 way to make a sum of 1 cent, and similarly 1 way to make 5, 10 or 25 cents, and that no other sum is possible using just one coin.

But notice what happens when we look at $p^2(x)$:

$$p^2(x) = 1x^2 + 2x^6 + 1x^{10} + 2x^{11} + 2x^{15} + 1x^{20} + 2x^{26} + 2x^{30} + 2x^{35} + 1x^{50}$$

which turns out to say that, using exactly 2 US coins, there is just 1 way to make 2 cents, 2 ways to make 6 cents (1+5 and 5+1), and so on.

The beauty of this system is that it's so easy to multiply polynomials. Moreover, if we are only interested in, say, sums up to 100, we can simply drop higher order terms from our polynomial as they arise. Using this idea,

$$\begin{aligned} p^3(x) = {} & 1x^3 + 3x^7 + 3x^{11} + 3x^{12} + 1x^{15} + 6x^{16} + 3x^{20} + 3x^{21} + 3x^{25} + 3x^{27} + 1x^{30} \\ & + 6x^{31} + 3x^{35} + 6x^{36} + 6x^{40} + 3x^{45} + 3x^{51} + 3x^{55} + 3x^{60} + 1x^{75} \end{aligned}$$

and it is worthwhile to check that you can really believe that there are, for instance, 6 ways of making the sum of 31 cents using three coins (where we count different orderings of the coins as different solutions).

The exercises ask you to write a Python function which accepts two polynomials $p_1(x)$ and $p_2(x)$, producing the product polynomial $p_3(x)$. An easy approach is to assume each polynomial is of degree at most 100, and store an array of 101 coefficients, most of which are zero. Then the coefficient $c_3(k)$ of polynomial $p_3(x)$ is the sum of all products $c_1(i) * c_2(j)$ where $i + j = k$. Try this on a simple polyomial first!

16 Can We Serve Our Guests? Chicken McNuggets

At one time, McDonalds sold packages of Chicken McNuggets in three sizes, containing 6, 9, or 20 pieces. Supposing you want to buy exactly n McNuggets, how many ways can this be done? Let M(n) represent this value. For example, if you want 38 pieces, you can either order 6+6+6+20 or 9+9+20, and these are the only ways to get 38 McNuggets, so M(38)=2.

If you think about this, the McNuggets problem is analogous to the change making problem. We can ask whether it is possible to make a sum of 38, using "coins" with "value" 6, 9 and 20. Related questsion ask for the minimal number of "coins" (separate packages) needed to make 38, or the number of different combinations of packages that will total 38.

For this problem, instead of asking for the minimal number of packages, let's ask for the number of different ways of achieving our desired sum. This number might be 0, (which it certainly is if we want 1, 2, 7, or 19 MacNuggets, for instance). We will once again use the "dynamic programming" approach. In order to determine $M(n)$, we will compute the values associated with smaller arguments, and then consider reaching a total of n by adding one more package of a given size.

We reason as follows:

- There is no way to achieve the numbers 0 through 5.
- if n is at least 6, there are $M(n-6)$ ways to achieve n by ordering an additional 6 pack.
- if n is at least 9, there are $M(n-9)$ ways to achieve n by ordering an additional 9 pack.
- if n is at least 20, there are $M(n-20)$ ways to achieve n by ordering an additional 20 pack.
- There is no other way to achieve $M(n)$.

This suggests that we write a program like the following:

```
function nugget_numbers ( n )

  Let m be an array of n+1 entries
  Let the zeroth entry of m be 1
```

```
For i from 1 up to n
  Initialize m[i] to zero
  if ( 6 <= i ), add m[i-6]
  if ( 9 <= i ), add m[i-9]
  if ( 20 <= i ), add m[i-20]
```

The exercises ask you to write a Python implementation of this algorithm, and to compute the values of $M(n)$ up to $n = 50$.

17 Separate but Equal Sums?: The Partition Problem

When kids get together for a team sport, it may be necessary to divide them into two teams. It's only fair to try to ensure that the two teams are as evenly matched as possible. A parallel mathematical problem is known as the *partition problem*, and asks whether, given a list of numbers (not a set, because we allow repetitions!), we can split the numbers into two groups with the same sum. Assuming we are working with integers, we can immediately conclude that there is no perfect solution if the numbers sum to an odd value. Thus, in some cases, we must conclude that at least in some cases, exact equality is impossible. Thus, we rephrase the problem:

The Partition Problem

Given a list of n numbers v, with total sum T, divide the numbers into two groups with sums S_0 and S_1 such that we minimize $|T/2 - S_0| + |T/2 - S_1|$.

18 The Partition Problem: A Brute Force Algorithm

Any solution of the partition problem must assign the i-th number v_i to one group or the other. We can represent this assignment by x_i, and for convenience, we will assume the two groups are labeled "0" and "1". One way to determine a best solution (there might be several) is to simply consider every possible such assignment vector x, compute the sums, evaluate $G(x) = |T - S_0| + |T - S_1|$, and remember the lowest value of $G(x)$ that we encounter. This approach is guaranteed to work, requires almost no thought or programming genius, and is known as the brute force approach.

As an example

```
      2 10  3  8  5  7  9  5  3  2  S0  S1   G

  0   0  0  0  0  0  0  0  0  0  0  54   0  54
  1   0  0  0  0  0  0  0  0  0  1  52   2  50
  2   0  0  0  0  0  0  0  0  1  0  51   3  48
  3   0  0  0  0  0  0  0  0  1  1  49   5  44
  4   0  0  0  0  0  0  0  1  0  0  49   5  44
....
1020  1  1  1  1  1  1  1  1  0  0   5  49  44
1021  1  1  1  1  1  1  1  1  0  1   3  51  48
1022  1  1  1  1  1  1  1  1  1  0   2  52  50
1023  1  1  1  1  1  1  1  1  1  1   0  54  54
```

Actually, it turns out that there are 23 different choice vectors x^* for which both S_0 and S_1 are exactly 27, so that we achieve the optimal value $G(x^*) = 0$, a perfect match. But, as you can see, the brute force approach, while simple, requires a lot of work, namely evaluating 2^n separate cases. This is not humanly interesting for $n = 10$, but the computer will do it for us. As n grows larger, though, even the computer will

gradually be unable to produce a result in a reasonable time. Nonetheless, the brute force approach gives us an opening for small problems.

In order to perform the brute force approach, we need to be able to generate each possible choice vector x, that is, all the binary vectors of length n. Luckily, the function `next_binary_vector()` will do exactly this task for us. The first set is the one chosen by x, and the second set is the remaining items. So our program simply has to generate the next `x`, divide the set into two subsets, compute and compare the sums, and, if they are equal, increment the number of solutions found.

19 The Partition Problem: A Greedy Algorithm

The brute force approach will solve small partition problems in reasonable time. Bugt because it requires 2^n steps to deal with a problem involving n items, it is useless even for seemingly moderate values of n. Suppose that even an approximate solution would be valuable to us. Then we can consider a greedy algorithm, that runs as follows:

The Greedy Partition Algorithm

Process the data in order, starting with the largest. Award the next data item to whichever group has the smaller sum. In the event of a tie, award the item in any way you wish.

In Python, we can make a descending sorted copy of a list v using the command:

```
v = v.sort ( reverse = True )  # "reverse" means largest first
```

The exercises ask you to implement the greedy partition algorithm.

20 The Knapsack Problem: The Optimal Steal

In the knapsack problem, a thief has a knapsack that can carry no more than $Wmax$ pounds. The thief is in a room with n items, with item i having weight w_i and value v_i. The thief wants to get away with the maximum value in goods, while not exceeding the weight limit of the knapsack.

Let x represent a particular choice vector, that is, a list of which items the thief might propose to steal. Then x_i is 1 if the thief proposes to steal item i and 0 otherwise. We can determine the total weight and value of the items to be stolen:

$$W(x) = \sum_{i=0}^{n} w_i x_i$$

$$V(x) = \sum_{i=0}^{n} v_i x_i$$

However, a choice vector x can be immediately rejected if $W(x) > Wmax$. Each choice vector that is not rejected is called a *feasible choice*. Note that the list of feasible choices is not empty; there is always at least one (why?). The problem is to find the choice vector with the biggest payoff. The thief asks our help in making the best choice, and we can now phrase this as

The Knapsack Problem

Of all feasible choice vectors x, find an optimal choice vector x^* which maximizes the value $V(x)$.

21 The Knapsack Problem: Brute Force

To help the thief, we have to find the feasible choice vector x that maximizes $V(x)$. Unlike problems in analysis, for this problem we only have a finite number of choices to consider. So one approach to consider would be to simply work out every possible x vector, determine its corresponding weight, and if that weight is no bigger than $Wmax$, work out the profit, and let x^* be the choice vector for which you saw the biggest profit.

There's nothing wrong with this approach, except that it will quickly become useless for large problems. But let's start here, work out how many choices we have to deal with, and how we could make sure we consider each and every one in an orderly way. Once again, we can use our function `next_binary_vector()` for this task. Each time we are presented with a new choice vector `x`, we check whether it stays within the weight limit. If so, we evaluate its profit. If this profit beats our previous record, we have found the latest candidate for the best choice of objects to steal.

Here is how this might look as a program:

```
def knapsack_brute ( Wmax, w, v ):
  import numpy as np
  n = len ( w )
  vbest = - np.inf
  xbest = []

  x = np.zeros ( n )
  while ( True ):
    wx = np.dot ( w, x )
    if ( wx <= Wmax ):
      vx = np.dot ( v, x )
      if ( vbest < vx ):
        vbest = vx
        xbest = x
    x = binary_vector_next ( x )
    if ( all ( x == 0 ) ):
      break

  return xbest, vbest
```

Suppose our knapsack has a capacity of 104 pounds, and the objects we can steal have the following properties:

```
  Object   Value  Weight

       1    $442      41
       2    $525      50
       3    $511      49
       4    $593      59
       5    $546      55
       6    $564      57
       7    $617      60
```

Since $n = 7$, we have $2^7 = 128$ choice vectors to consider, which is not really a challenge for our computer (although much to many for us to work through with a pencil!). The winning choice is:

```
  Object   Value  Weight  Choice

       1    $546      55     0
       2    $442      41     1
       3    $511      49     0
       4    $593      59     1
```

```
       5    $525      50     0
       6    $564      57     0
       7    $617      60     1

Sum:       $1652     169     3
```

Here, we were lucky to exactly fill our knapsack. In most such problems, we will end up having some capacity left over, without being able to add any more items to our load.

22 The Knapsack Problem: Greed is Good

As the number n of items grows, the number of choices grows exponentially, as 2^n. This means that 10 items givens about a thousand choices, 20 is a million, and 30 a billion. It's clear that seemingly moderate values of n can result in computational work loads beyond our patience or budget. For such larger problems, we need to search for methods that we can believe have a reasonable chance of approaching the maximal profit that we can't afford to compute directly.

If the thief can't afford to check all the options, then a reasonable approach is to be greedy; in this case, that means to choose the most precious items first. Thus, the thief would take a gold object first, rather than a wooden one, no matter how big the two items are. Thus, the thief would first calculate a value per pound, use this to order the objects, and then fill the knapsack with items in this order. Here is the ratio for our previous example:

```
  Object   Value  Weight  Value/Weight  Rank

       1    $546      55      9.9         6
       2    $442      41     10.7         1
       3    $511      49     10.4         3
       4    $593      59     10.0         5
       5    $525      50     10.5         2
       6    $564      57      9.8         7
       7    $617      60     10.2         4
```

Now the thief would reason as follows:

```
                        W =             0  V =                 0
  Add item 2 of rank 1: W =  0 + 41 =  41, V =   0 + 442 =  442
  Add item 5 of rank 2: W = 41 + 50 =  91, V = 442 + 525 =  967
  Add item 3 of rank 3: W = 91 + 49 = 140, V = 967 + 511 = 1478
  Can't add any more items without exceeding weight limit.
```

Note that the strategy almost works. We have 30 pounds of capacity left. If we could add 30 pounds of item 7 to our take, this would be an additional 30 * \$10.2 = \$306 of profit, giving us a total profit of \$1778 which exceeds the optimal profit. The point is that we only have the options of taking or leaving each object, and so we can't squeeze that extra value into our knapsack. However, if we use the greedy option, then the amount of unused space left over and the per-pound value of the remaining objects gives us an upper bound on the optimal haul. In particular, if we exactly fill the knapsack, we have an optimal solution.

23 Combinations: Selecting a Certain Number of Elements

A combination is a selection of k items from a set of n items. Thus, in particular, a combination is a kind of subset. This is also called a *combination without replacement*: the k items are distinct. Two combinations

are said to be equal if they are equal as sets, that is, they have exactly the same elements. Thus, the combinations (1,2,3) and (2,1,3) are equal. The number of distinct combinations has a special symbol ${}_nC_k$ or $\binom{n}{k}$, and often pronounced *"n choose k"*. For instance, given a set of 5 objects, the number of combinations of 0, 1, 2, 3, 4 or 5 items is, respectively 1, 5, 10, 10, 5, 1.

You may realize that these numbers represent row 5 of Pascal's triangle:

```
                1
              1   1
            1   2   1
          1   3   3   1
        1   4   6   4   1
      1   5   10  10  5   1
    1   6   15  20  15  6   1
  1   7   21  35  35  21  7   1
```

Column k of row n of Pascal's triangle is the value ${}_nC_k$.

24 Evaluating the Number of Combinations ${}_nC_k$

We have more important things to talk about, but it turns out that we do need to stop for a moment and consider the fact that sometimes, computing the number of combinations is a little more dangerous than we might realize.

There is a simple formula for the value of ${}_nC_k$:

$$ {}_nC_k = \frac{n!}{k!(n-k)!} $$

and it is an interesting exercise to convince yourself that, for $0 \leq k \leq n$, the quantity ${}_nC_k$ is always a positive integer. It is also a surprise to see that each entry in Pascal's triangle is found by adding the two entries immediately above it, where we assume a zero value in cases where we are at the first or last entry of a row. Since we're only dealing with integers, it may seem as though this computation will be trouble free. However, the fact that the factorial function is involved in our calculation means that, even for "reasonable" values of n and k, we may run across products and ratios of very large numbers, even in cases where the final result is a small integer.

We will need to evaluate ${}_nC_k$ computationally. Although we can do it from the basic formula, it is preferable to use software that knows about shortcuts and efficient techniques for getting the answer accurately.

Thus, MATLAB can evaluate the quantity with the command

```
nck = nchoosek ( n, k )
```

while Python has two ways of quickly computing the result, with an important difference:

```
from math import comb
nck1 = comb ( n, k )

from scipy.special import comb
nck2 = comb ( n, k )
```

These two Python functions may seem to give the same results, but `nck1`, the `math` result, is computed using integer arithmetic, while `nck2`, the `scipy.special` result, is computed with real arithmetic and returned as a real number. The important thing to realize is that the computation of ${}_nC_k$ can involve very large integers,

even when the final result is small. In Python, **integers have no upper size limit**, and so such results can be computed exactly and correctly. However, a computation using real arithmetic may become inexact, or blow up.

To see an example of inaccuracy, we do have to use fairly large values of n and k, but similar values are needed for many computations for games involving decks of cards:

```
>>> from math import comb
>>> v1 = comb ( 100, 50 )
>>> v1
100891344545564193334812497256
>>> from scipy.special import comb
>>> v2 = comb ( 100, 50 )
>>> v2
1.0089134454556415e+29   # the last three visible digits are ...415; that final 5 is wrong!
```

To see an example of blowup, we won't bother trying to guess why we are using such large values of n and k, we will just marvel that it doesn't bother `math.comb()` at all:

```
>>> from math import comb
>>> v1 = comb ( 10000, 5000 )
>>> v1
1591790263...0553649120  #  huge number of correct digits omitted here
>>> from scipy.special import comb
>>> v2 = comb ( 10000, 5000 )
>>> v2
inf                      #  the result can't even be approximated in real arithmetic
```

25 Exercises

1. Follow the description of the the algorithm for computing the next permutation, and compute the next permutation for each of these examples:
   (a) p = [1,2,5,3,4]
   (b) p = [2,3,1,5,4]
   (c) p = [3,5,4,2,1]
   (d) p = [4,1,5,3,2]
   (e) p = [5,3,4,2,1]
2. Follow the description of the algorithm for computing the next permutation, and write a Python function of the form `p,rank=next_permutation(p,rank,n)`. Test it by computing and printing the permutations of 4 objects.
3. Write a Python function `s = permutation_sign(p)` that computes the sign of a permutation.
4. Create the function `matrix_determinant()`. Test it on some random matrices. Compare your results with the value returned by `scipy.linalg.det()`.
5. Write a Python function which implements the brute force approach to the traveling salesperson problem, and solve the five city example given in the text.
6. Use the brute force method to solve the traveling salesperson problem for the nine city distance matrix.
7. Implement the nearest neighbor algorithm for the traveling salesperson problem. Verify that it works by applying it to the nine city distance matrix. Then download the file `forty_eight.txt` and seek a minimal tour of the 48 "continental" US capital cities.
8. Write a Python program that implements the `change_greedy()` algorithm. Use it to compute a collection of coins that add up to 96 cents.
9. Write a Python program that implements the `change_dynamic()` algorithm. Use it to compute the minimum number of coins needed to form every sum from 1 to 50.

10. Write a Python program `polynomial_multiply()`, which multiplies two polynomials $p_1(x)$ and $p_2(x)$, which are each represented as an array of no more than 101 coefficients. Demonstrate that your program is correct by computing several polynomial products.
11. Use your Python program `polynomial_multiply()` to report the number of ways of computing every sum up to 100 cents, using any 4 US coins; for this approach, we count two solutions as different if they involve the same coins, but in different orders.
12. Write a Python program which handles the MacNuggets problem, computing $M(n)$, the number of ways of ordering exactly n MacNuggets. Compute results for $0 \leq n \leq 50$
13. Use the brute force method to solve the partition problem for the set of values [2,10,3,8,5,7,9,5,3,2]. An article claims there are 23 solutions, but you probably got a different number. Explain the difference.
14. Use the brute force method to solve the partition problem for the set of values [771,121,281,854,885,734,486,1003,83,62]. How many solutions did you find? How many choice vectors x need to be generated to answer this question?
15. Write an implementation of the greedy partition method. Demonstrate it on the data [2,10,3,8,5,7,9,5,3,2], for which you probably will not get an exact match.
16. Use the greedy method to solve the partition problem for the set of values [771,121,281,854,885,734,486,1003,83,62]. How close did you come to a perfect match?
17. Another algorithm for the partition problem is known as the "Karmarkar-Karp" algorithm or the "Largest Difference Method" (LDM). Find documentation on this algorithm, implement it yourself in Python, and use it on the set of data [771,121,281,854,885,734,486,1003,83,62]. See *"The Easiest Hard Problem"* by Brian Hayes, American Scientist, March-April 2002.
18. Run the `knapsack_brute()` code on the sample data and verify that the best result has a value of $900.
19. The discussion of the greedy algorithm for the knapsack problem concludes with the statement "In particular, if we exactly fill the knapsack, we have an optimal solution." Explain in your own words why this must be true.
20. Apply the greedy algorithm to the following knapsack problem, and report your result. What is an upper bound for the optimal profit?

```
W = 750
v =135, 139, 149, 150, 156, 163, 173, 184, 192, 201, 210, 214, 221, 229, 240
w = 70,  73,  77,  80,  82,  87,  90,  94,  98, 106, 110, 113, 115, 118, 120
```