Partition number: For a positive integer $n$, $p(n)$ is the number of different ways to represent $n$ as a decreasing sum of positive integers.

1. $6 = 6$
2. $6 = 5 + 1$
3. $6 = 4 + 2$
4. $6 = 4 + 1 + 1$
5. $6 = 3 + 3$
6. $6 = 3 + 2 + 1$
7. $6 = 3 + 1 + 1 + 1$
8. $6 = 2 + 2 + 2$
9. $6 = 2 + 2 + 1 + 1$
10. $6 = 2 + 1 + 1 + 1 + 1$
11. $6 = 1 + 1 + 1 + 1 + 1 + 1$

Thus, $p(6) = 11$.

Question: How to compute $p(n)$?
**PARTITIONSINNER**(num, max_digit):
  if (num ≤ 1) or (max_digit = 1)
    return 1
  if max_digit > num
    max_digit ← num

  res ← 0
  for i ← max_digit downto 1 do
    res+ = **PARTITIONSINNER**(num – i, i)

  return res

**PARTITIONS**(n):
  return **PARTITIONSINNER**(n, n)

**Question**: What is the running time of **PARTITIONS**(n)?
Running time of \textsc{Partitions}(n) is $\Theta(p(n))$.

Easy to verify:

$$3^{\sqrt{n}/4} \leq p(n) \leq n^n$$

[Exercise: Prove those bounds (or better).]

In fact, Hardy and Ramanujan (1918) showed:

$$p(n) \approx \frac{e^{\pi\sqrt{2n/3}}}{4n\sqrt{3}}$$

\textbf{Question:} Is there a faster algorithm?

\textbf{Question:} Why is this algorithm so slowwwwwwwwwwwwwwwwwwwww?
**Question:** Why is this algorithm so slow? 

**Answer:** `PARTITIONS(num, max_digit)` is called a lot of times with the same parameters.

**Idea:** Cache results:

```plaintext
MPARTITIONSINNER(num, max_digit):
  if (num ≤ 1) or (max_digit = 1)
    return 1
  if max_digit > num
    max_digit ← num

  if ⟨num, max_digit⟩ in cache
    val ← cache ⟨⟨num, max_digit⟩⟩
    return val

  res ← 0
  for i ← max_digit downto 1 do
    res+ = MPARTITIONSINNER( num − i, i )
  cache(⟨num, max_digit⟩) ← res

  return res
```

```plaintext
MPARTITIONS(n):
  return MPARTITIONSINNER(n, n)
```

We implement cache using hash table.

**Question:** What is the running time of `MPARTITIONS`?

**Question:** How many entries are stored in the cache?
**Question:** What is the running time of MPARTITIONS?

**Argument:**

1. If a call to MPARTITIONSinner takes (by itself) more than constant time, then we perform a store in the cache.

2. Number of store operations in the cache is $O(n^2)$.

3. We charge the work in the loop to the resulting store. The work in the loop is $O(n)$.

4. Running time of MPARTITIONS($n$) is $O(n^3)$.

**Observation:** Analysis is sloppy. Might be possible to do better.

**Observation:** Basic speedup idea is very generic...
Memoization:
Take a recursive function and cache the results as the computations goes on. Before trying to compute a value, check if it was already computed and if it is already in the cache. If so, return result from the cache.
   If it is not in the cache, compute it and store it in the cache.

When does it work: When there is a lot of inefficiency in the computation of the recursive function because we perform the same call again and again.

When it does NOT work:
1. When the number of different recursive function calls (i.e., the different values of the parameters in the recursive call) is “large”.
2. When the function has side effects.

Question: Can we do better than caching?

More pain more gain:
In a lot of cases we can analyse the recursive calls, and store them directly in an array. This technique is *dynamic programming*. We can sometime save space and improve running time.