1 Basic Idea - Partition Number

**Definition 1.1** 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.

<table>
<thead>
<tr>
<th>( n )</th>
<th>6=6</th>
<th>6=5+1</th>
<th>6=4+2</th>
<th>6=3+3</th>
<th>6=2+2+2</th>
<th>6=1+1+1+1+1+1</th>
</tr>
</thead>
<tbody>
<tr>
<td>( p(n) )</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
</tr>
</tbody>
</table>

**Question:** How to compute \( p(n) \)?

```plaintext
PartitionsI(num, d)//d-max digit
  if (num ≤ 1) or (d = 1)
    return 1
  if d > num
    d ← num
  res ← 0
  for i ← d downto 1
    res+ = PartitionsI(num – i, i)
  return res
Partitions(n)
  return PartitionsI(n, n)
```

**Question:** What is the running time of Partitions\((n)\)?

Running time of Partitions\((n)\) is \( \Theta(p(n)) \).

Easy to verify:

\[
3^{\sqrt{n}/4} ≤ p(n) ≤ n^n
\]

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

In fact, Hardy and Ramanujan (1918) showed:
\[ p(n) \approx \frac{e^{\pi \sqrt{2n/3}}}{4n^{3/2}} \]

**Question:** Is there a faster algorithm?

**Question:** Why is this algorithm so slowwwwwwwwwwwwwwwwwww?

**Answer:**

\textbf{Partitions}(num, max\_digit) is called a lot of times with the same parameters.

**Idea:** Cache results:

\begin{verbatim}
PartitionsI_C(num, d)
    if (num \leq 1) or (max\_digit = 1)
        return 1
    if d > num
        d \leftarrow num
    if \langle num, max\_digit \rangle in cache
        return cache(\langle num, max\_digit \rangle)
    res \leftarrow 0
    for i \leftarrow d downto 1
        res\leftarrow res + \text{PartitionSI}_C(num - i, i)
    cache(\langle num, max\_digit \rangle) \leftarrow res
    return res

Partitions_S_C(n)
    return PartitionsI_C(n, n)
\end{verbatim}

We implement cache using hash table.

**Question:** What is the running time of MPARTITIONS\_C?

**Question:** How many entries are stored in the cache?

**Question:** What is the running time of MPARTITIONS\_C?

**Argument:**

1. If a call to MPARTITIONSI\_C 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\_C(n) is \(O(n^3)\).

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

**Observation:** Basic speedup idea is very generic...
1.1 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 in dynamic programming over Memoization.

2 Dynamic programing made easy

1. Solve the problem using recursion - easy (?)).

2. Modify the recursive program so that it caches the results.

3. Dynamic programming: Modify the cache into mutli-dim array.

3 Fibonacci numbers

\[
\text{FibR}(n) \\
\text{if } n \leq 1 \\
\text{return } 1 \\
\text{return } \text{FibR}(n-1) + \text{FibR}(n-2)
\]

Q: Running time of FibR(n)?

\[
O(F_n) \text{ where } F_n \text{ is the } n\text{th Fibonacci number.}
\]

\[
F_n = \frac{1}{\sqrt{5}} \left[ \left( \frac{1 + \sqrt{5}}{2} \right)^n + \left( \frac{1 - \sqrt{5}}{2} \right)^n \right] = \Theta(\phi^n)
\]

\[
\phi = \frac{1 + \sqrt{5}}{2}
\]
FibDP($n$)
  if $n \leq 1$
    return 1
  if $F[n]$ initialized
    return $F[n]$
  $F[n] \leftarrow \text{FibDP}(n-1) + \text{FibDP}(n-2)$
  return $F[n]$

Q: Running time of $\text{FibDP}(n)$?
A: $O(n)$-linear.

Observe that we can fill the table in the other direction, and reduce space to $O(1)$:

FibI($n$)
  $\text{prev} \leftarrow 0$, $\text{curr} \leftarrow 1$
  for $i = 1$ to $n$
    $\text{next} \leftarrow \text{curr} + \text{prev}$
    $\text{prev} \leftarrow \text{curr}$
    $\text{curr} \leftarrow \text{next}$
  return $\text{curr}$

Running time of $\text{FibI}$ identical to the running time of $\text{FibDP}$.

Q: Can we do better?
A: Yes.

$\begin{pmatrix} y \\ x+y \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix}$

$\begin{pmatrix} f_{n-1} \\ f_n \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} \begin{pmatrix} f_{n-2} \\ f_{n-1} \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}^2 \begin{pmatrix} f_{n-3} \\ f_{n-2} \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}^{n-3} \begin{pmatrix} f_2 \\ f_1 \end{pmatrix}$

Thus, computing the $n$th Fibonacci number can be done by computing $\begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}^{n-3}$.

How to this quickly?

$a \times b \times c = (a \times b) \times c = a \times (b \times c)$

FastExp($a,n$)
  if $n = 0$ then return 1
  if $n = 1$ then return $a$
  if $n$ is even then
    return $(\text{FastExp}(a,n/2))^2$
  else
    return $a \times (\text{FastExp}(a,\frac{n-1}{2}))^2$
What is the running time of FastExp?

A: $O(\log n)$.

Thus, we can compute in $f_n$ is $O(\log n)$ time!

But... $f_n$ has $\approx \log_{10} 1.68...^n = \Theta(n)$ digits?

We assumed that the time to handle a number if $O(1)$ which is not true in practice if the numbers are large. Be careful with this assumption...

## 4 Edit Distance

**Question 4.1** Given two strings $A$ and $B$ how many edit operations do I have to make to turn $A$ into $B$?

*Example:*

$A = \text{"har-peled"}$

$B = \text{"sharp eyed"}$

sharp eyed

*Distance:* 4

Operations:

1. Insert character.
2. Delete character.
3. Replace.

Price of each operation is one. How to compute the edit-distance (min # of edit operations needed)?

List the edit operations from left to right. Thus, edit distance is just an alignment problem:

<table>
<thead>
<tr>
<th>h</th>
<th>a</th>
<th>r</th>
<th>p</th>
<th>e</th>
<th>l</th>
<th>e</th>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td>s</td>
<td>h</td>
<td>a</td>
<td>r</td>
<td>p</td>
<td>&lt;space&gt;</td>
<td>e</td>
<td>y</td>
</tr>
</tbody>
</table>

Edit price: 4.

Insert: [ ]
Delete: [ ]
Replace: [ ]
Ignore: [ ]

Idea: Let us look on the last character and decide which of the categories it falls into:
Running time of $\text{ed}(\ldots)$? Clearly exponential... At least $3^n$.

Q: How many different recursive calls $\text{ed}$ performs?
A: $O(m \times n)$ different calls.

Q: Running time of $\text{edM}(\ldots)$?
A: $O(m \times n)$.

Understanding what the algorithm really do:
\textbf{edDP}(A[1..m], B[1..n])
for i=1 to m \(T[i, 0] \leftarrow i\)
for j=1 to n \(T[0, j] \leftarrow j\)
for i \leftarrow 1 \text{ to } m
    for j \leftarrow 1 \text{ to } n
        \(p_{\text{insert}} = T[i, j - 1] + 1\)
        \(p_{\text{delete}} = T[i - 1, j] + 1\)
        \(p_{\text{r/ignore}} = T[i - 1, j - 1] + \left[ A[i] \neq B[j] \right]\)
        \(T[i, j] \leftarrow \min(p_{\text{insert}}, p_{\text{delete}}, p_{\text{r/ignore}})\)
\textbf{return} \(T[m, n]\)

Q: How to reduce the space to linear (instead of quadratic)?
A: Just compute the matrix two rows in a time.