Scientific Programming Practical 20

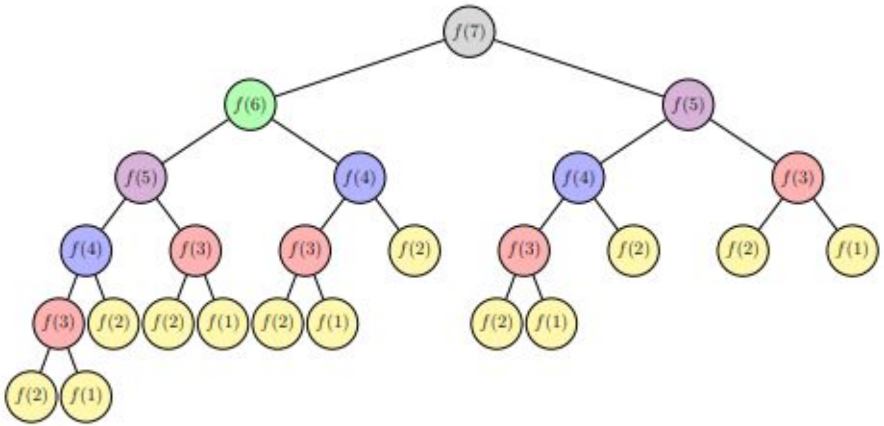
Introduction

Luca Bianco - Academic Year 2018-19
luca.bianco@fmach.it

Dynamic Programming

Dynamic programming is a way to **optimize recursive algorithms**.

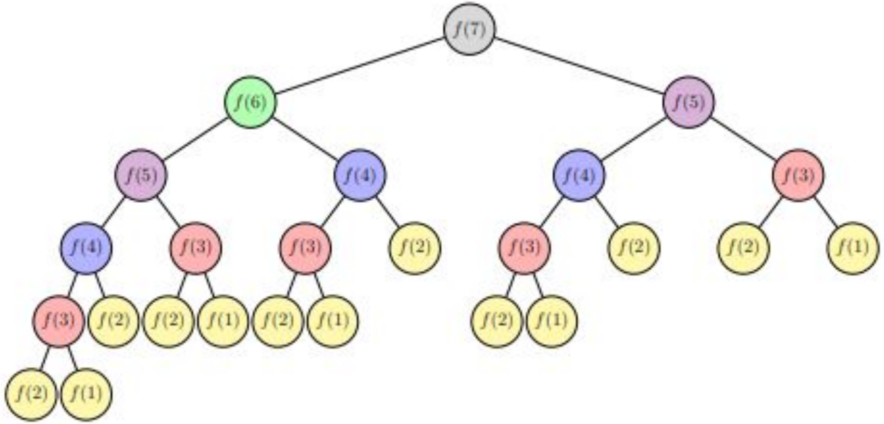
Every time we find an algorithm with repetitive recursive calls to smaller subproblems to solve a bigger problem, we can optimize it by using **dynamic programming**.



Dynamic Programming

Two approaches

1. **Top-down:** solve the problem breaking it down in smaller subproblems. If the subproblem has already been solved, then the answer has already been saved somewhere. If it has not already been solved, compute a solution and store it. This method is called **memoization**;
2. **Bottom-up:** solve the problem starting from the most trivial subproblems going up until the complete problem has been solved. Smaller subproblems are guaranteed to be solved before bigger ones. This method is called **dynamic programming**.



Example: fibonacci numbers

Consider the classic example of the computation of Fibonacci numbers:

0,1,1,2,3,5,8,13,21,34,55,89,144,233,377,610,987,...

which can be computed with the following recursive formula:

$$F(n) = \begin{cases} n & \text{if } n == 0 \text{ or } n == 1 \\ F(n-1) + F(n-2) & \text{if } n > 1 \end{cases}$$

Example: fibonacci numbers

$$F(n) = \begin{cases} n & if\ n == 0\ or\ n\ == 1 \\ F(n-1) + F(n-2) & if\ n > 1 \end{cases}$$

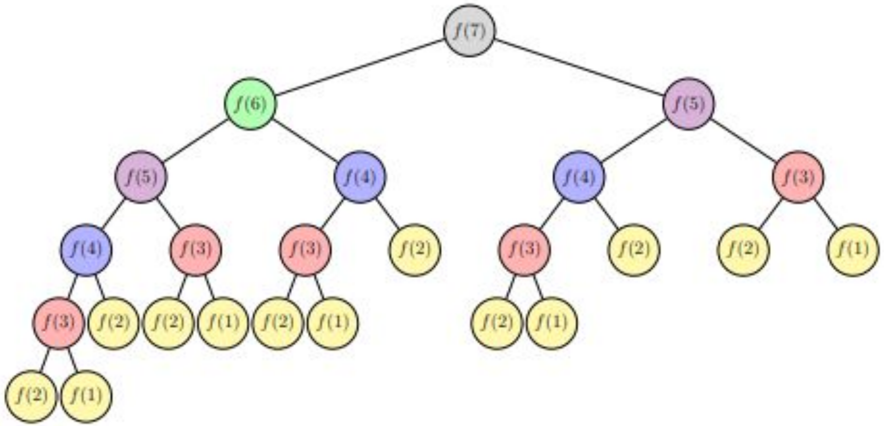
Recursive code

```
import time

def fib(n):
    if n <= 1:
        return n
    else:
        return fib(n - 1) + fib(n - 2)


for i in range(20):
    print("Fib({})= {}".format(i, fib(i)))


for i in range(35,38):
    start_t = time.time()
    print("\nFib({})= {}".format(i, fib(i)))
    end_t = time.time()
    print("It took {:.2f}s".format(end_t-start_t))
```

```
Fib(0)= 0
Fib(1)= 1
Fib(2)= 1
Fib(3)= 2
Fib(4)= 3
Fib(5)= 5
Fib(6)= 8
Fib(7)= 13
Fib(8)= 21
Fib(9)= 34
Fib(10)= 55
Fib(11)= 89
Fib(12)= 144
Fib(13)= 233
Fib(14)= 377
Fib(15)= 610
Fib(16)= 987
Fib(17)= 1597
Fib(18)= 2584
Fib(19)= 4181

Fib(35)= 9227465
It took 4.39s

Fib(36)= 14930352
It took 6.88s

Fib(37)= 24157817
It took 10.58s
```

Example: fibonacci numbers

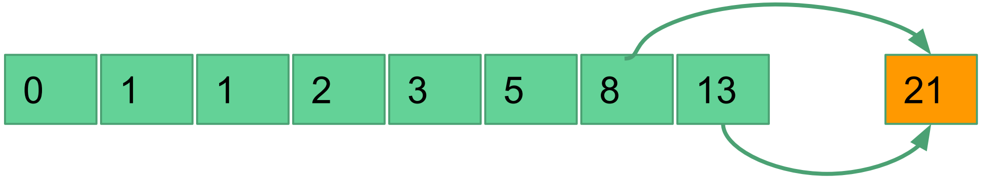
$$F(n) = \begin{cases} n & if\ n == 0\ or\ n\ == 1 \\ F(n-1) + F(n-2) & if\ n > 1 \end{cases}$$



Example: fibonacci numbers

$$F(n) = \begin{cases} n & \text{if } n == 0 \text{ or } n == 1 \\ F(n-1) + F(n-2) & \text{if } n > 1 \end{cases}$$

We can use dynamic programming to avoid computing over and over again the same values:



Example: fibonacci numbers

$$F(n) = \begin{cases} n & \text{if } n == 0 \text{ or } n \;== 1 \\ F(n-1) + F(n-2) & \text{if } n > 1 \end{cases}$$

We can use dynamic programming to avoid computing over and over again the same values:

```
import time

def fib_dp(n):
    fib = [0]* (n+1)
    if n > 1:
        fib[1] = 1

    for i in range(2,n + 1):
        fib[i] = fib[i-2] + fib[i - 1]

    return fib[n]

for i in range(20):
    print("Fib({})= {}".format(i, fib_dp(i)))


for i in range(35,38):
    start_t = time.time()
    print("\nFib({})= {}".format(i, fib_dp(i)))
    end_t = time.time()
    print("It took {:.2f}s".format(end_t-start_t))

#we can even do:
for i in range(1000,1003):
    start_t = time.time()
    print("\nFib({})= {}".format(i, fib_dp(i)))
    end_t = time.time()
    print("It took {:.2f}s".format(end_t-start_t))
```

Example: fibonacci numbers

$$F(n) = \begin{cases} n & \textit{if } n == 0 \textit{ or } n \;\; == 1 \\ F(n-1) + F(n-2) & \textit{if } n > 1 \end{cases}$$

We can use dynamic programming to avoid computing over and over again the same values:

```python
import time

def fib_dp(n):
    fib = [0]* (n+1)
    if n > 1:
        fib[1] = 1

    for i in range(2,n + 1):
        fib[i] = fib[i-2] + fib[i - 1]

    return fib[n]

for i in range(20):
    print("Fib({})= {}".format(i, fib_dp(i)))


for i in range(35,38):
    start_t = time.time()
    print("\nFib({})= {}".format(i, fib_dp(i)))
    end_t = time.time()
    print("It took {:.2f}s".format(end_t-start_t))

#we can even do:
for i in range(1000,1003):
    start_t = time.time()
    print("\nFib({})= {}".format(i, fib_dp(i)))
    end_t = time.time()
    print("It took {:.2f}s".format(end_t-start_t))
```

```
Fib(0)= 0
Fib(1)= 0
Fib(2)= 1
Fib(3)= 2
Fib(4)= 3
Fib(5)= 5
Fib(6)= 8
Fib(7)= 13
Fib(8)= 21
Fib(9)= 34
Fib(10)= 55
Fib(11)= 89
Fib(12)= 144
Fib(13)= 233
Fib(14)= 377
Fib(15)= 610
Fib(16)= 987
Fib(17)= 1597
Fib(18)= 2584
Fib(19)= 4181

Fib(35)= 9227465
It took 0.00s

Fib(36)= 14930352
It took 0.00s

Fib(37)= 24157817
It took 0.00s
```

```
Fib(1000)= 43466557686937456435688527675...49228875
It took 0.00s

Fib(1001)= 70330367711422815821835254877...23403501
It took 0.00s

Fib(1002)= 113796925398360272257523782552...72632376
It took 0.00s
```

Greedy paradigm

In the **greedy programming** paradigm, at each step of the computation the choice that seems the best **at the time** is always taken. In other words, greedy algorithms build a solution by choosing the **local best value** in the hope that this would lead to a **global best solution**. As we will see later, this is not always guaranteed.

Greedy paradigm: example

Example: Let's write a method that a coffee selling machine can use to give change using the least amount of coins of 50,20,10, 5 and 1 cents.

GREEDY CHOICE: always pick the biggest coin

```
def giveChange(amount):
    #res counts the used coins
    #having value 50,20,10,5,1
    print("Computing change of {} cents".format(amount))
    coins = [50,20,10,5,1]
    res = [0,0,0,0,0]
    while amount > 0:
        nextCoin = 0
        #order makes greedy choice! First 50,then 20, 10...
        if amount > 50:
            res[0] += 1
            nextCoin = 50
        elif amount > 20:
            res[1] += 1
            nextCoin = 20

        elif amount > 10:
            res[2] += 1
            nextCoin = 10
        elif amount > 5:
            res[3] +=1
            nextCoin = 5
        else:
            res[4] += 1
            nextCoin = 1

        amount -= nextCoin

    for i in range(len(coins)):
        if res[i] > 0:
            print("{} x {} cent coins".format(res[i],coins[i]))
```

Greedy paradigm: example

Example: Let's write a method that a coffee selling machine can use to give change using the least amount of coins of 50,20,10, 5 and 1 cents.

GREEDY CHOICE: always pick the biggest coin

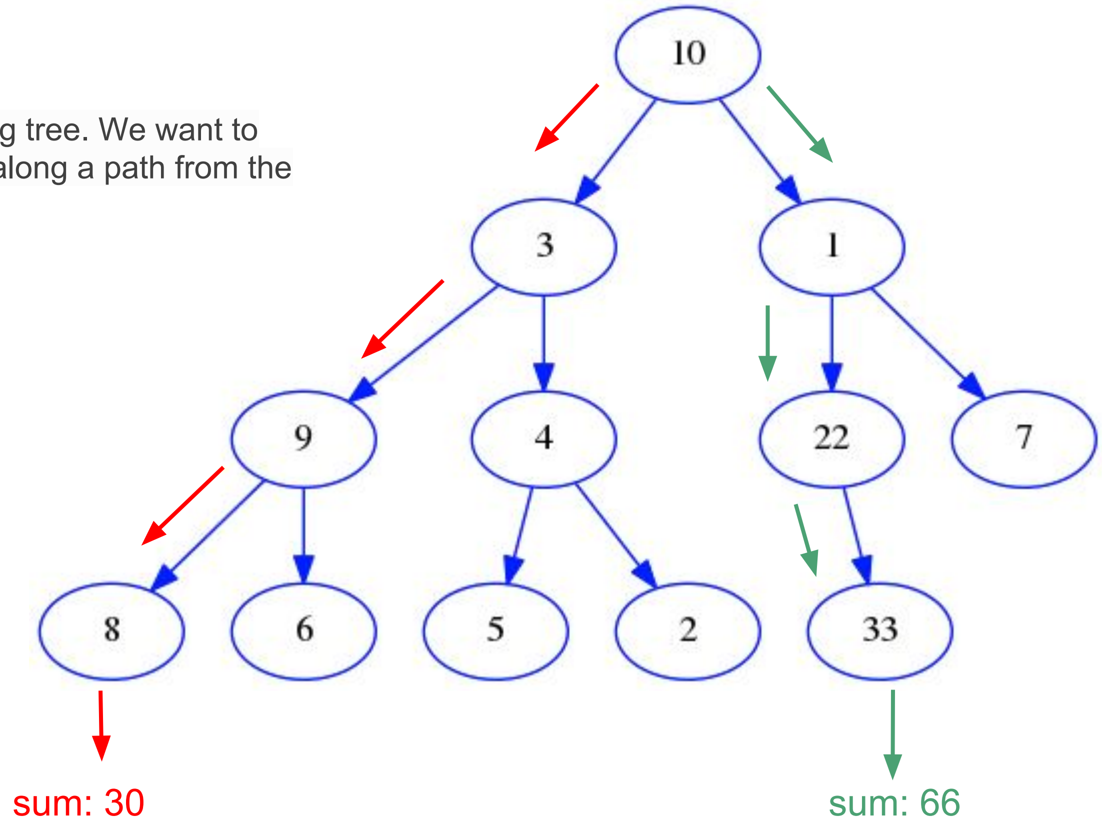
```python
def giveChange(amount):
    #res counts the used coins
    #having value 50,20,10,5,1
    print("Computing change of {} cents".format(amount))
    coins = [50,20,10,5,1]
    res = [0,0,0,0,0]
    while amount > 0:
        nextCoin = 0
        #order makes greedy choice! First 50,then 20, 10...
        if amount > 50:
            res[0] += 1
            nextCoin = 50
        elif amount > 20:
            res[1] += 1
            nextCoin = 20

        elif amount > 10:
            res[2] += 1
            nextCoin = 10
        elif amount > 5:
            res[3] +=1
            nextCoin = 5
        else:
            res[4] += 1
            nextCoin = 1

        amount -= nextCoin

    for i in range(len(coins)):
        if res[i] > 0:
            print("{} x {} cent coins".format(res[i],coins[i]))
```

```python
giveChange(36)
giveChange(72)
giveChange(232)
```

```
Computing change of 36 cents
1 x 20 cent coins
1 x 10 cent coins
1 x 5 cent coins
1 x 1 cent coins
Computing change of 72 cents
1 x 50 cent coins
1 x 20 cent coins
2 x 1 cent coins
Computing change of 232 cents
4 x 50 cent coins
1 x 20 cent coins
1 x 10 cent coins
2 x 1 cent coins
```

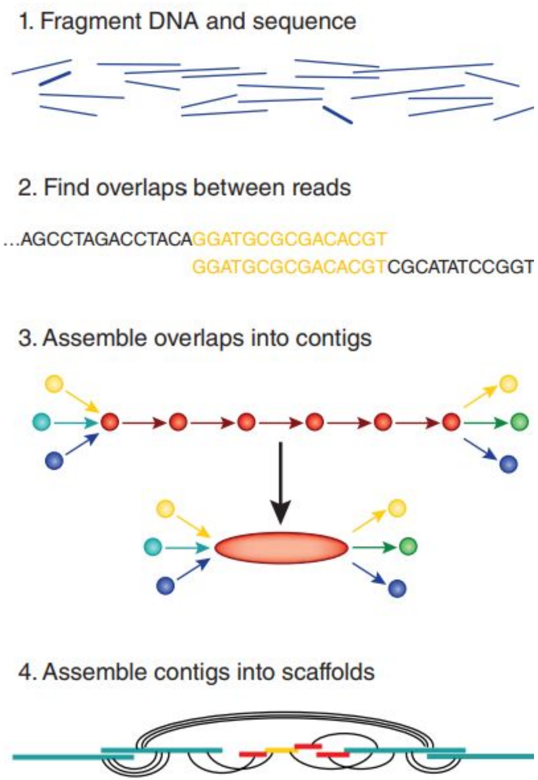
Greedy algos: not always finding the right solution

Consider the following tree. We want to get the biggest sum along a path from the root to a leaf.



Another DP example: finding overlaps

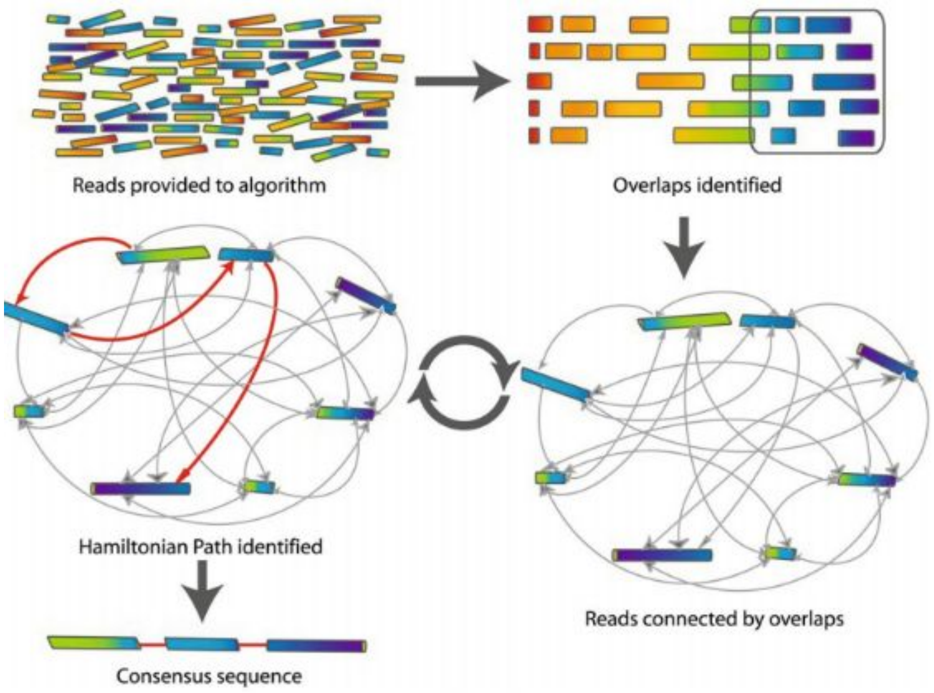
Genome sequence assembly



[from M. Baker, Nature Methods, 2014]

Another DP example: finding overlaps

Motivation: long read (PacBio, Oxford Nanopore) genome assembly



Overlap identification: build the graph

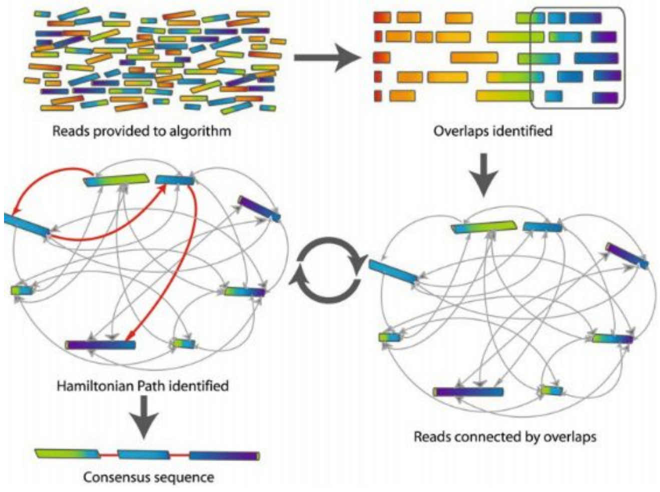
N reads:
O(N^2) overlaps

Layout: simplify the graph and build contigs

Consensus: finalize contigs as consensus of all reads

Another DP example: finding overlaps

Motivation: long read (PacBio, Oxford Nanopore) genome assembly



Overlap identification: build the graph

Layout: simplify the graph and build contigs

Consensus: finalize contigs as consensus of all reads

Assembly: consensus of overlapping reads

```
TAGATTACACAGATTACTGA TTGATGGCGTAA CTA
TAGATTACACAGATTACTGACTTGATGGCGTAAACTA
TAG TTACACAGATTATTGACTTCATGGCGTAA CTA
TAGATTACACAGATTACTGACTTGATGGCGTAA CTA
TAGATTACACAGATTACTGACTTGATGGCGTAA CTA

TAGATTACACAGATTACTGACTTGATGGCGTAA CTA
```

Another DP example: finding overlaps

Given two reads ***X*** and ***Y*** the optimal overlap among them is the longest suffix of ***X*** that matches (possibly with some mismatches) a prefix of ***Y***.

```
X = ATCGGTTCGGTGAAGTAA

Y = CGGTGACGTTACCATATCCAG

overlap:

ATCGGTTCGGTGAAGTAA
       CGGTGACGTTACCATATCCAG
```

Another DP example: finding overlaps

Alignment score

	A	C	G	T	-
A	0	4	2	4	8
C	4	0	4	2	8
G	2	4	0	4	8
T	4	2	4	0	8
-	8	8	8	8	8

```
X = ATCGGTTCGGTGAAGTAA

Y = CGGTGACGTTACCATATCCAG

overlap:

ATCGGTTCGGTGAAGTAA
       CGGTGACGTTACCATATCCAG
```

Given the two reads X and Y

The **global alignment recurrence:**

$$A[i,j] = min \begin{cases} A[i-1,j] + score(x[i-1], "-") \\ A[i,j-1] + score("-", y[j-1]) \\ A[i-1,j-1] + score(x[i-1], y[j-1] \end{cases}$$

This provides a **N+1** x **M+1** matrix (where N and M are the lengths of X and Y)

The first row of the matrix corresponds to "-" in *X* and all elements are initialized to ∞, while the first column corresponds to a "-" in *Y* and all elements are initialized to 0.

Another DP example: finding overlaps

	A	C	G	T	-
A	0	4	2	4	8
C	4	0	4	2	8
G	2	4	0	4	8
T	4	2	4	0	8
-	8	8	8	8	8

$$A[i,j] = min \begin{cases} A[i-1,j] + score(x[i-1], "-") \\ A[i,j-1] + score("-", y[j-1]) \\ A[i-1,j-1] + score(x[i-1], y[j-1] \end{cases}$$

	-	C	G	G	T	G	A	C	G	T	T	A	C	C	A	T	A	T	C	C	A	G
-	0	inf	inf	inf	inf	inf	inf	inf	inf	inf	inf	inf	inf	inf	inf	inf	inf	inf	inf	inf	inf	inf
A	0	4	12	20	28	36	44	52	60	68	76	84	92	100	108	116	124	132	140	148	156	164
T	0	2	8	16	20	28	36	44	52	60	68	76	84	92	100	108	116	124	132	140	148	156
C	0	0	6	12	18	24	32	36	44	52	60	68	76	84	92	100	108	116	124	132	140	148
G	0	4	0	6	14	18	26	34	36	44	52	60	68	76	84	92	100	108	116	124	132	140
G	0	4	4	0	8	14	20	28	34	40	48	54	62	70	78	86	94	102	110	118	126	132
T	0	2	8	8	0	8	16	22	30	34	40	48	56	64	72	78	86	94	102	110	118	126
T	0	2	6	12	8	4	12	18	26	30	34	42	50	58	66	72	80	86	94	102	110	118
C	0	0	6	10	14	12	8	12	20	28	32	38	42	50	58	66	74	82	86	94	102	110
G	0	4	0	6	14	14	14	12	12	20	28	34	42	46	52	60	68	76	84	90	96	102
G	0	4	4	0	8	14	16	18	12	16	24	30	38	46	48	56	62	70	78	86	92	96
T	0	2	8	8	0	8	16	18	20	12	16	24	32	40	48	48	56	62	70	78	86	94
G	0	4	2	8	8	0	8	16	18	20	16	18	26	34	42	50	50	58	66	74	80	86
A	0	4	6	4	12	8	0	8	16	22	24	16	22	30	34	42	50	54	62	70	74	82
A	0	4	6	8	8	14	8	4	10	18	26	24	20	26	30	38	42	50	58	66	70	76
G	0	4	4	6	12	8	16	12	4	12	20	28	28	24	28	34	40	46	54	62	68	70
T	inf	2	8	8	6	14	12	18	12	4	12	20	28	30	28	28	36	40	48	56	64	72
A	inf	10	4	10	12	8	14	16	20	12	8	12	20	28	30	32	28	36	44	52	56	64
A	inf	inf	inf	inf	14	14	8	16	18	20	16	8	16	24	28	34	32	32	40	48	52	58

set to infinity to force overlap len > 3

travel back from minimum value to min value to find overlap

```
X = ATCGGTTCGGTGAAGTAA

Y = CGGTGACGTTACCATATCCAG

overlap:

ATCGGTTCGGTGAAGTAA
       CGGTGACGTTACCATATCCAG
```

Finding overlaps: build the matrix

	-	C	G	G	T	G	A	C	G	T	T	A	C	C	A	T	A	T	C	C	A	G
-	0	inf	inf	inf	inf	inf	inf	inf	inf	inf	inf	inf	inf	inf	inf	inf	inf	inf	inf	inf	inf	inf
A	0	4	12	20	28	36	44	52	60	68	76	84	92	100	108	116	124	132	140	148	156	164
T	0	2	8	16	20	28	36	44	52	60	68	76	84	92	100	108	116	124	132	140	148	156
C	0	0	6	12	18	24	32	36	44	52	60	68	76	84	92	100	108	116	124	132	140	148
G	0	4	0	6	14	18	26	34	36	44	52	60	68	76	84	92	100	108	116	124	132	140
G	0	4	4	0	8	14	20	28	34	40	48	54	62	70	78	86	94	102	110	118	126	132
T	0	2	8	8	0	8	16	22	30	34	40	48	56	64	72	78	86	94	102	110	118	126
T	0	2	6	12	8	4	12	18	26	30	34	42	50	58	66	72	80	86	94	102	110	118
C	0	0	6	10	14	12	8	12	20	28	32	38	42	50	58	66	74	82	86	94	102	110
G	0	4	0	6	14	14	14	12	12	20	28	34	42	46	52	60	68	76	84	90	96	102
G	0	4	4	0	8	14	16	18	12	16	24	30	38	46	48	56	62	70	78	86	92	96
T	0	2	8	8	0	8	16	18	20	12	16	24	32	40	48	48	56	62	70	78	86	94
G	0	4	2	8	8	0	8	16	18	20	16	18	26	34	42	50	50	58	66	74	80	86
A	0	4	6	4	12	8	0	8	16	22	24	16	22	30	34	42	50	54	62	70	74	82
A	0	4	6	8	8	14	8	4	10	18	26	24	20	26	30	38	42	50	58	66	70	76
G	0	4	4	6	12	8	16	12	4	12	20	28	28	24	28	34	40	46	54	62	68	70
T	inf	2	8	8	6	14	12	18	12	4	12	20	28	30	28	28	36	40	48	56	64	72
A	inf	10	4	10	12	8	14	16	20	12	8	12	20	28	30	32	28	36	44	52	56	64
A	inf	inf	inf	inf	14	14	8	16	18	20	16	8	16	24	28	34	32	32	40	48	52	58

```python
def score(a,b):
    mat = {}
    mat["A"] = {"A" : 0, "C" : 4, "G" : 2, "T" : 4, "-" : 8}
    mat["C"] = {"A" : 4, "C" : 0, "G" : 4, "T" : 2, "-" : 8}
    mat["G"] = {"A" : 2, "C" : 4, "G" : 0, "T" : 4, "-" : 8}
    mat["T"] = {"A" : 4, "C" : 2, "G" : 4, "T" : 0, "-" : 8}
    mat["-"] = {"A" : 8, "C" : 8, "G" : 8, "T" : 8, "-" : 8}

    return mat[a][b]


def computeMatrix(X,Y, minLen = 0):
    # + 1 is for leading "-"
    x_len = len(X) + 1
    y_len = len(Y) + 1
    print(X)
    print(Y)
    A = []
    #initialize first column to 0 and first row to infinite

    for i in range(x_len - minLen + 1):
        A.append([0])
    for i in range(minLen -1):
        A.append([math.inf])
    for i in range(y_len -1 ):
        A[0].append(math.inf)

    for i in range(1,x_len):
        for j in range(1,y_len):
            c1 = A[i-1][j] + score(X[i-1], "-")
            c2 = A[i][j-1] + score("-", Y[j-1])
            c3 = A[i-1][j-1] + score(X[i-1],Y[j-1])
            #print("i,j: {},{}".format(i,j))
            A[i].append(min([c1,c2,c3]))
    if minLen != 0:
        for i in range(0,minLen):
            A[-1][i] = math.inf
    return A
```

minLen to enforce
minimum length of overlap

	A	C	G	T	-
A	0	4	2	4	8
C	4	0	4	2	8
G	2	4	0	4	8
T	4	2	4	0	8
-	8	8	8	8	8

$$A[i,j] = min\begin{cases} A[i-1,j] + score(x[i-1], " - ") \\ A[i,j-1] + score(" - ", y[j-1]) \\ A[i-1,j-1] + score(x[i-1], y[j-1] \end{cases}$$

Finding overlaps: build the matrix

```python
def plotMatrix(Mat, X,Y):
    X = "-" + X
    outStr = "\t-" +"\t"+ "\t".join(list(Y))

    for i in range(len(X)):
        outStr+="\n" + X[i] +"\t"+ "\t".join([str(x) for x in Mat[i]])
    print(outStr)


X = "CTCGGCCCTAGG"
Y = "GGCTCTAGGCCC"

A = computeMatrix(X,Y,minLen = 5)

print("The overlap matrix:")
plotMatrix(A,X,Y)
```

$$A[i,j] = min \begin{cases} A[i-1,j] + score(x[i-1], "-") \\ A[i,j-1] + score("-", y[j-1]) \\ A[i-1,j-1] + score(x[i-1], y[j-1] \end{cases}$$

```
CTCGGCCCTAGG
GGCTCTAGGCCC
The overlap matrix:
```

	-	G	G	C	T	C	T	A	G	G	C	C	C
-	0	inf	inf	inf	inf	inf	inf	inf	inf	inf	inf	inf	inf
C	0	4	12	20	28	36	44	52	60	68	76	84	92
T	0	4	8	14	20	28	36	44	52	60	68	76	84
C	0	4	8	8	16	20	28	36	44	52	60	68	76
G	0	0	4	12	12	20	24	30	36	44	52	60	68
G	0	0	0	8	16	16	24	26	30	36	44	52	60
C	0	4	4	0	8	16	18	26	30	34	36	44	52
C	0	4	8	4	2	8	16	22	30	34	34	36	44
C	0	4	8	8	6	2	10	18	26	34	34	34	36
T	inf	4	8	10	8	8	2	10	18	26	34	36	36
A	inf	12	6	12	14	12	10	2	10	18	26	34	40
G	inf	20	12	10	16	18	16	10	2	10	18	26	34
G	inf	inf	inf	inf	inf	20	22	18	10	2	10	18	26

http://qcbprolab.readthedocs.io/en/latest/practical20.html

Exercises

1. Catalan numbers (info here) are defined by the following recurrence equation:

$$C(n) = \begin{cases} 1 & if\ n = 0 \\ \sum_{i=0}^{n-1} C_i \dot{C}_{n-1-i} & if\ n > 1 \end{cases}$$

the first few values are reported below:

$$1, 1, 2, 5, 14, 42, 132, 429, 1430, 4862, 16796, 58786,$$
$$208012, 742900, 2674440, 9694845, 35357670, 129644790,$$
$$477638700, 1767263190, 6564120420, 24466267020, 91482563640, \ldots$$

a. Write a recursive function `recCatalan(n)` to compute the n-th catalan number;
b. Write a dynamic programming function `dpCatalan(n)` to compute the n-th catalan number.

Test your code with:

```
catN = []
for i in range(0,15):
    catN.append(recCatalan(i))

print("First 15 catalan numbers:")
print(catN)
```

Finally, check how long it takes to compute the 20th catalan number with the recursive algorithm and with the dynamic programming one.

Show/Hide Solution

2. Recall the code that computes the overlap matrix given two DNA reads (strings). The code is