Week 10 – S02: Summary Dynamic Programming

50.004 Introduction to Algorithms

Dr. Subhajit Datta

ISTD, SUTD

Write a program to compute the n’th Fibonacci number using recursion!

- If you are wondering what a Fibonacci number is
  - https://en.wikipedia.org/wiki/Fibonacci_number
- If you are wondering what recursion is
  - ☹
  - https://en.wikipedia.org/wiki/Recursion

What does this program do?

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

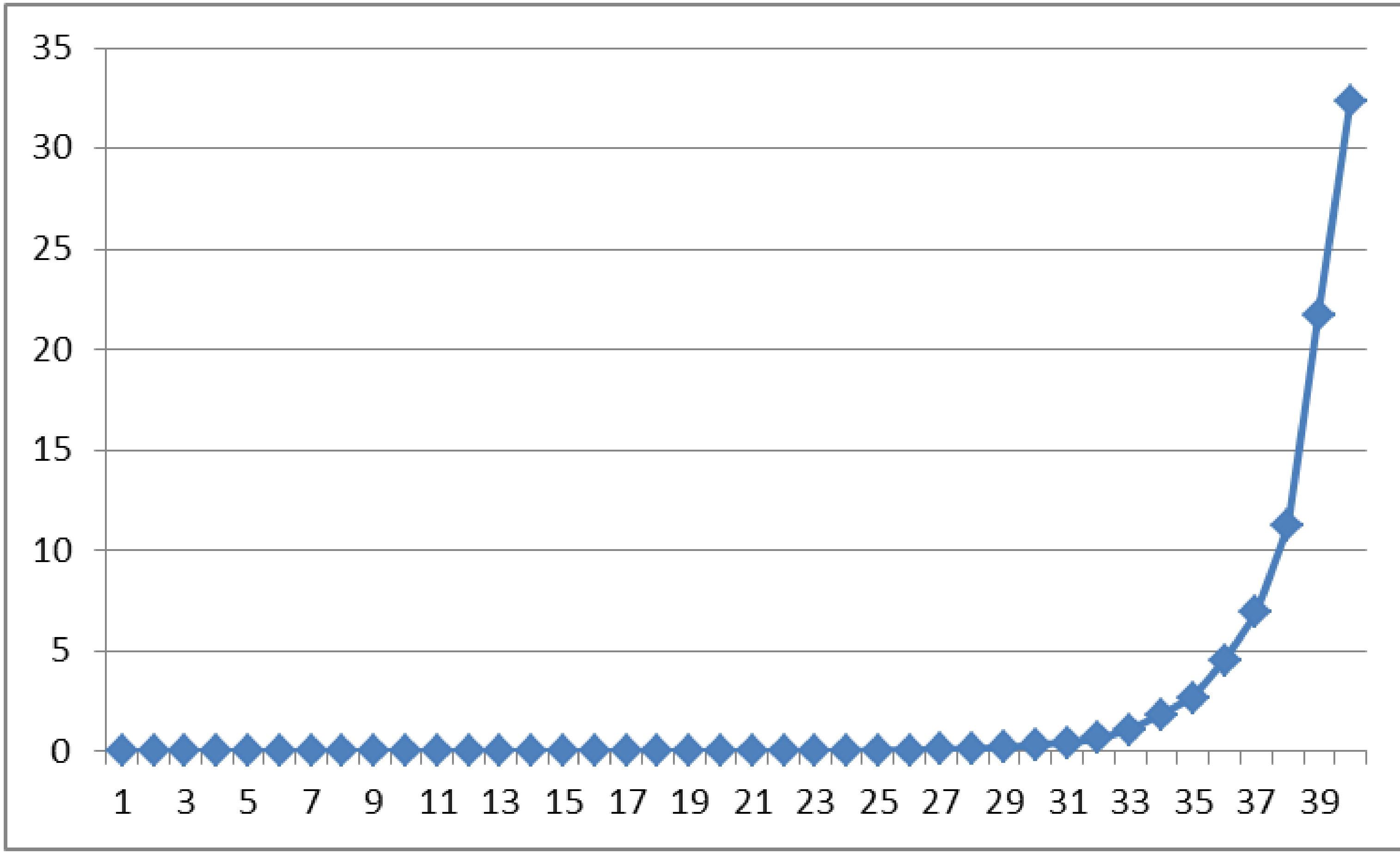
Plot the running times, for *n*=1,2,…,40

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

Plot the running times, for *n*=1,2,…,40

```
def f(n):
    if n <= 2:
        return 1
    else:
        return f(n-1) + f(n-2)
import time
for i in range(1,41):
    elapsedTime = 0.0
    start = time.time()
    result = f(i)
    end = time.time()
    elapsedTime = end-start
    print i, result, elapsedTime
```

Plot the running times, for n=1,2,…,40



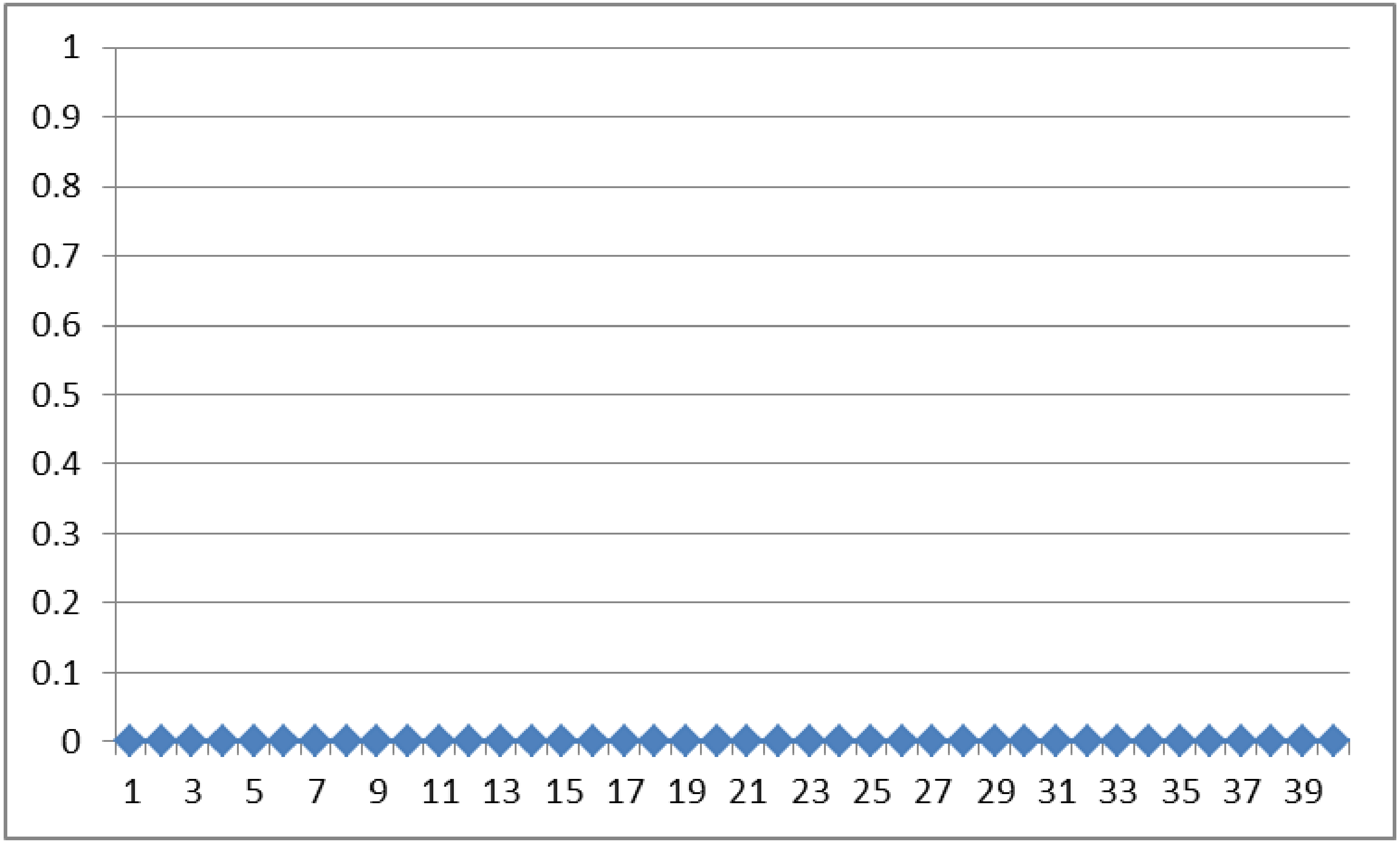
Now, what does this program do?

```
table = {}
def fNew(n):
    if n in table:
        x = table[n]
    elif n <= 2:
        x = 1
        table[n] = x
    else:
        x = fNew(n-1) + fNew(n-2)
        table[n] = x
    return x
```

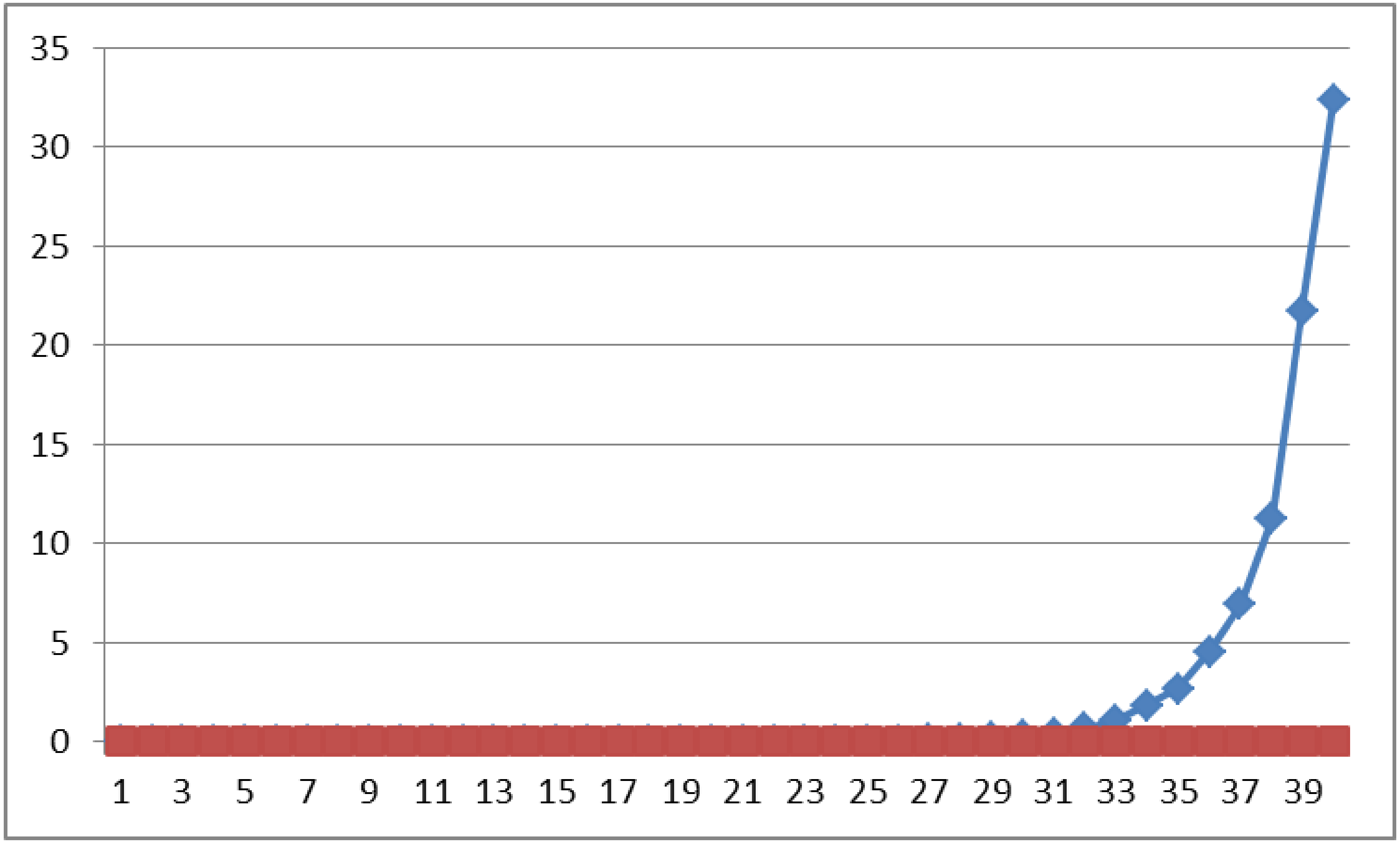
Plot the running times, for n=1,2,…,40

```
table = {}
def fNew(n):
    if n in table:
        x = table[n]
    elif n <= 2:
        x = 1
        table[n] = x
    else:
        x = fNew(n-1) + fNew(n-2)
        table[n] = x
    return x
import time
for i in range(1,41):
    elapsedTime = 0.0
    start = time.time()
    result = f(i)
    end = time.time()
    elapsedTime = end-start
    print i, result, elapsedTime
```

Plot the running times, for n=1,2,…,40



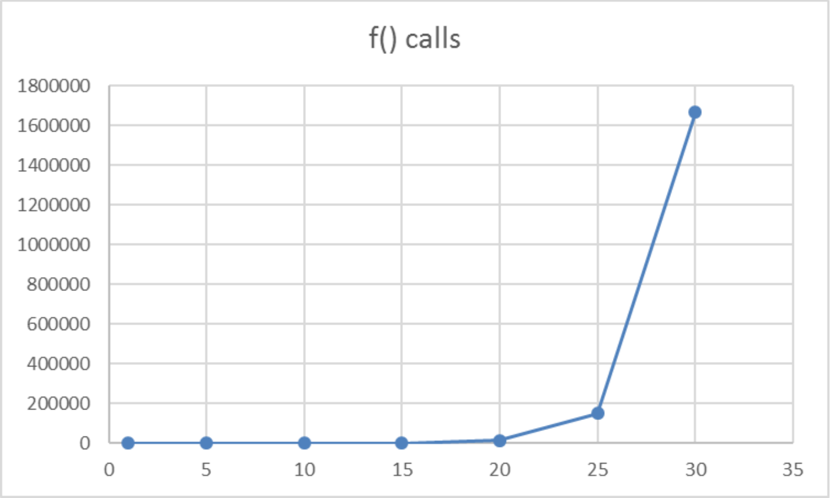
Comparing f() and fNew() running times

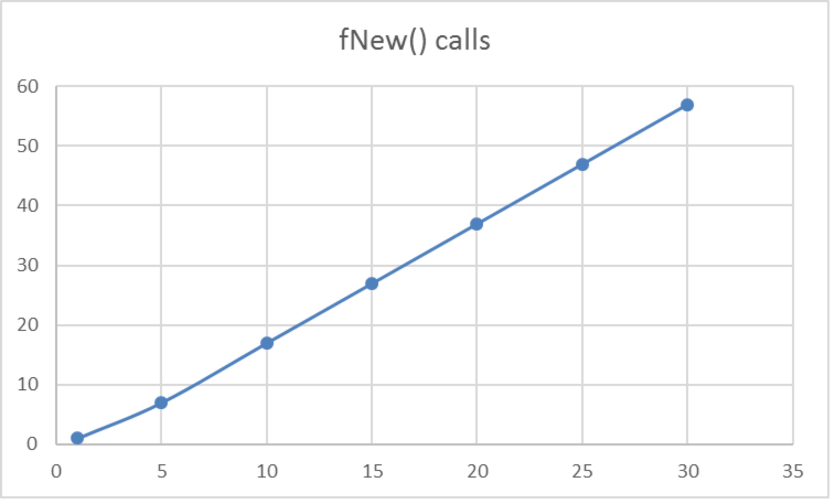


Counting number of calls to f() and fNew()

```
n = 30
def f(n):
    f.count +=1
    if n <= 2:
        return 1
    else:
        return f(n-1) + f(n-2)
f.count = 0
print n, f(n), f.count
########################################
table = {}
def fNew(n):
    fNew.count +=1
    if n in table:
        fibo = table[n]
    elif n <= 2:
        fibo = 1
        table[n] = fibo
    else:
        fibo = fNew(n-1) + fNew(n-2)
        table[n] = fibo
    return fibo

fNew.count = 0
print n, fNew(n), fNew.count
```





Calculating Fibonacci numbers with a *table*

```
table = {}
def fiboTopDown(n):
    if n in table:
        fibo = table[n]
    elif n <= 2:
        fibo = 1
        table[n] = fibo
    else:
        fibo = fiboTopDown(n-1) + fiboTopDown(n-2)
        table[n] = fibo
    return fibo
```

We will soon see why this is "top down"

The basic idea

```python
table = {}
def fiboTopDown(n):
    if n in table:
        fibo = table[n]
    elif n <= 2:
        fibo = 1
        table[n] = fibo
    else:
        fibo = fiboTopDown(n-1) + fiboTopDown(n-2)
        table[n] = fibo
    return fibo
```

Create a table

Check if what you need is in the table

If not in table, compute and store it in table for future use!

Exercise: Write a factorial function using this technique

```python
table = {}
def fiboTopDown(n):
    if n in table:
        fibo = table[n]
    elif n <= 2:
        fibo = 1
        table[n] = fibo
    else:
        fibo = fiboTopDown(n-1) + fiboTopDown(n-2)
        table[n] = fibo
    return fibo
```

Create a table

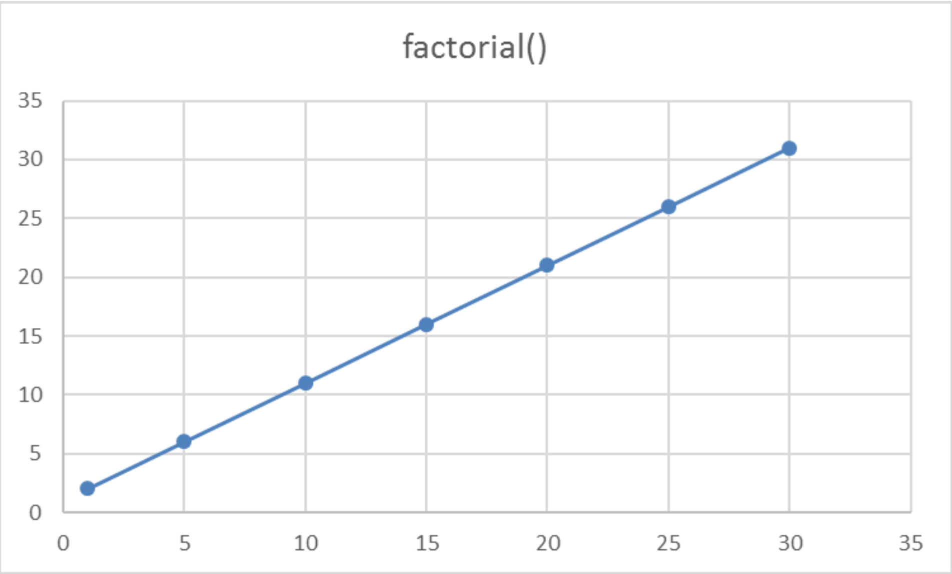
Check if what you need is in the table

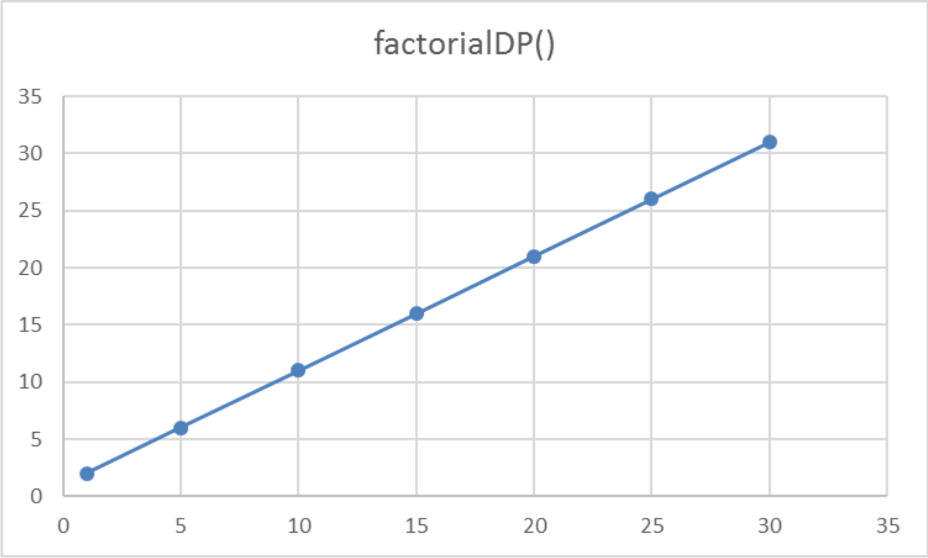
If not in table, compute and store it in table for future use!

Calculating factorials, the DP way

```
table = {0: 1}
def factorial(number):
    if number in table:
        return table[number]
    else:
        result = factorial(number - 1) * number
        table[number] = result
        return result
```

DP only improves life, if applies in the proper context!






```
n = 10
def factorial(n):
    factorial.count += 1
    if n==0:
        return 1
    else:
        return factorial(n-1)*n
factorial.count = 0
print n, factorial(n), factorial.count

table = {0: 1}
def factorialDP(number):
    factorialDP.count +=1
    if number in table:
        return table[number]
    else:
        result = factorialDP(number - 1) * number
        table[number] = result
        return result
factorialDP.count = 0
print n, factorialDP(n), factorialDP.count
```

Dynamic programming (DP)

- Dynamic programming, solves problems by combining the solutions to sub-problems
- Dynamic programming applies when the sub-problems overlap
  - That is, when sub-problems share sub-sub-problems
- A dynamic-programming algorithm solves each sub-sub-problem just once
  - Then saves its answer in a table
  - Thereby avoiding the work of re-computing the answer every time it solves each sub-sub-problem

Have you seen something like this before?

- Dynamic programming, solves problems by combining the solutions to sub-problems
- Dynamic programming applies when the sub-problems overlap
  - That is, when sub-problems share sub-sub-problems
- A dynamic-programming algorithm solves each sub-sub-problem just once
  - Then saves its answer in a table
  - Thereby avoiding the work of re-computing the answer every time it solves each sub-sub-problem.

This is divide-and-conquer, right?

- Dynamic programming, solves problems by combining the solutions to sub-problems
- Dynamic programming applies when the sub-problems overlap
  - That is, when sub-problems share sub-sub-problems
- A dynamic-programming algorithm solves each sub-sub-problem just once
  - Then saves its answer in a table
  - Thereby avoiding the work of re-computing the answer every time it solves each sub-sub-problem.

No!

- Dynamic programming, solves problems by combining the solutions to sub-problems
- Dynamic programming applies when the sub-problems overlap
  - That is, when sub-problems share sub-sub-problems
  - In this context, divide-and-conquer algorithms do more work than necessary
    - Repeatedly solving the common sub-sub-problems

What does “dynamic programming” mean?

‘Bellman … explained that he invented the name “dynamic programming” to hide the fact that he was doing mathematical research at RAND under a Secretary of Defense who “had a pathological fear and hatred of the term, research.” He settled on “dynamic programming” because it would be difficult give it a “pejorative meaning” and because “It was something not even a Congressman could object to.” ’

[John Rust 2006]

http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.87.2819&rep=rep1&type=pdf

Yes, same Bellman from Bellman-Ford algorithm!

Richard E. $\boxed{\text{Bellman}}$

(1920–1984)

IEEE Medal of Honor, 1979

http://www.amazon.com/Bellman-Continuum-Collection-Works-Richard/dp/9971500906

Why learn DP?

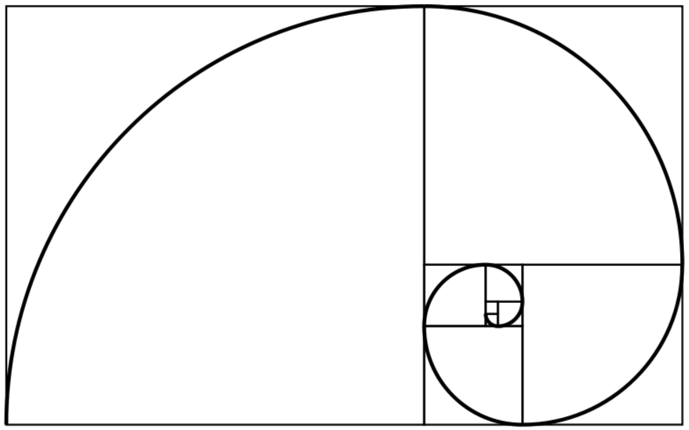
- So far …
  - BFS, DFS, Dijkstra, Bellman-Ford …
  - Algorithms for specific situations
- Dynamic programming
  - A general perspective on designing algorithms

Dynamic programming: Applications

- Typically applied to optimization problems
- Such problems can have many possible solutions
  - Each solution has a value
- We wish to find a solution with the optimal (minimum or maximum) value
- The solution is called *an* optimal solution
  - Not *the* optimal solution
  - There may be several solutions that achieve the optimal value

Fibonacci numbers

$$F_1 = F_2 = 1,\ F_n = F_{n-1} + F_{n-2}$$



Naïve algorithm

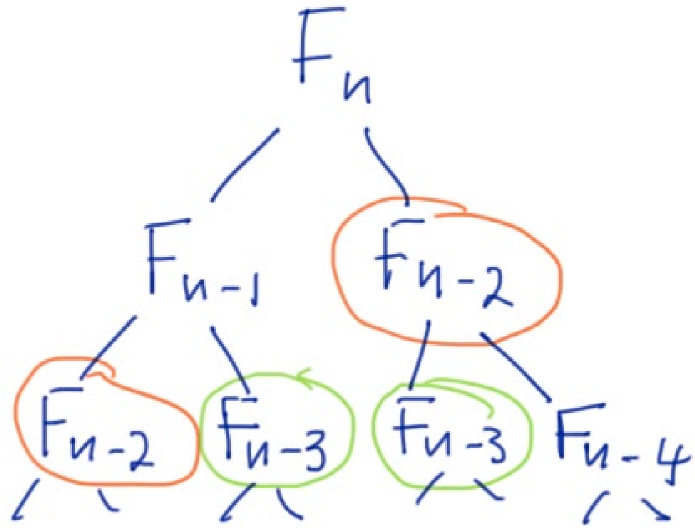
fib(n):

if $n \le 2 : f = 1$

else $f = \text{fib}(n-1) + \text{fib}(n-2)$

return f

$$T(n) = T(n-1) + T(n-2) + O(1)$$
$$\ge 2T(n-2) + O(1) \ge 2^{n/2}$$



Can we avoid the exponential cost?