# Extended Introduction to Computer Science CS1001.py 

## Chapter E <br> Lecture 11

## Recursion (cont. cont.)

- Memoization

Michal Kleinbort, Amir Rubinstein

School of Computer Science
Tel-Aviv University
Fall Semester 2023-24
http://tau-cs1001-py.wikidot.com

* Slides based on a course designed by Prof. Benny Chor


## Recursion: Plan

- Definition and basic examples
- Fibonacci
- factorial
- Recursive binary search
- Sorting
- Quick-Sort
- Merge-Sort
- Towers of Hanoi (and the "monster of Hanoi")
- Improving recursion with memoization
- An example from-Game theory-Chomp! (removed this semester)


## Computing Fibonacci Numbers

- We coded Fibonacci numbers, using recursion, as following:

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

- But surely nothing could go wrong with such simple and elegant code...
- To investigate this, let us explore the running time of fibonacci on $n=30,35,40,45, \ldots$


## Recursion Trees (reminder)

- Recursion trees are a common visual representation of a recursive process. For example, here is the recursion tree for fibonacci, for $n=6$ :

- Even for $n=6$ we have the same values computed over and over. This is highly wasteful and causes a huge overhead.


## Fibonacci Time Complexity

- Time complexity - overall number of operations in the whole tree


Tree depth
= $n-1$
$=O(n)$

- Length of shortest path from root to leaf: $\sim \frac{n}{2}$
- Length of longest path from root to leaf : $\sim n$
- Let $s$ denote the number of nodes in the tree. Then:

$$
(\sqrt{2})^{\mathrm{n}}=2^{\frac{\mathrm{n}}{2}} \leq \mathrm{s} \leq 2^{\mathrm{n}}
$$

## Fibonacci Time Complexity

- Time complexity - overall number of operations in the whole tree

- We have $<2^{n}-1$ nodes in the tree
- Each node takes $\mathrm{O}(1)$ operations (disregarding the size of the numbers being added)
- Thus, time complexity to compute the $n^{\prime}$ 'th Fibonacci number is $O\left(2^{n}\right)$ (not tight)
- Tight bound is $\Theta\left(\left(\frac{1+\sqrt{5}}{2}\right)^{n}\right)$ (without proof)


## Actual Time Measurement

- We have written the function elapsed, that measures the CPU time taken to execute a given expression (given as a string) repeat times.
- It returns the result in seconds.
- Note that the code first imports the time module, and uses the built-in eval function (this function invokes the interpreter).

```
import time # imports the Python time module
def elapsed(expression, repeat=1):
    t1 = time.perf_counter()
    for i in range(repeat):
        eval(expression)
    t2 = time.perf_counter()
    return t2-t1
```


## Actual Time Measurement: Example

```
>>> elapsed("sum(range(10**7))")
0.33300399999999897
>>> elapsed("sum(range(10**8))")
3.362785999999998
>>> elapsed("sum(range(10**9))")
34.029920000000004
```


## Fibonacci Time Measurements

```
>>> elapsed("fibonacci(30)")
0.31555
>>> elapsed("fibonacci(35)")
3.4169379999999996
>>> elapsed("fibonacci(40)")
38.288004
>>> elapsed("fibonacci(45)")
432.662887 # over 7 minutes !!
```

- These results demonstrate the exponential running time of our code (but they do not replace the time complexity analysis).


## Intuition for Improving Efficiency

- Instead of computing each call from scratch, the value for each input can be computed just once. Rather than re-computing it, we will fetch the value from memory, when needed.
- The technique of storing values instead of re-computing them is called memoization (resembles the term memorization).
- In other contexts, this technique is often used as part of dynamic programming (will be studies in the Algorithms course).


## Fibonacci: Recursive Code with Memoization

- We will use a dictionary named fib_dict, will contain the Fibonacci numbers already computed. We initialize the dictionary with fib_dict $=\{0: 1,1: 1\}$.
- fibonacci2 is an envelope function, which calls the recursive fib2.

```
def fibonacci2(n):
    """ Envelope function for Fibonacci,
        employing memoization in a dictionary """
    fib_dict = {0:1, 1:1} # initial dictionary
    return fib2(n, fib_dict)
def fib2(n, fib_dict):
    if n not in fib_dict:
        res = fib2(n-1, fib_dict) + fib2(n-2, fib_dict)
        fib_dict[n] = res
    return fib_dict[n]
```

- Let us again explore the running time of $f$ ibonacci2 on $n=30,35,40,45, \ldots$


## Time Measurements

- This small change implies a huge performance difference:

```
>>> elapsed("fibonacci(35)")
8.245295905878606
>>> elapsed("fibonacci2(35)")
9.053997947143898e-05
>>> elapsed("fibonacci(40)")
92.6057921749773
>>> elapsed("fibonacci2(40)")
9.909589798251517e-05
>>> elapsed("fibonacci(45)")
432.662887 # over 7 minutes !!
>>> elapsed("fibonacci2(45)")
0.00011015042046658152
1 6
```


## Time Complexity of Code with Memoization

- How does the recursion tree for the code with memoization look like?
- What is its depth?
- What is the time complexity of the function?
- In class, on board.


## Time Complexity of Code with Memoization

- The recursion tree of fib2:

- At each node the amount of work not including recursive calls is expected $O(1)$, due to the the dictionary operations.

Expected time complexity: $T(n)=O(n)$

## Diagnostic Printing

```
def fib2(n, fib_dict):
    if n not in fib_dict:
        print("start n =", n, fib_dict)
        res = fib2(n-1, fib_dict) + fib2(n-2, fib_dict)
        fib_dict[n] = res
        print("end n =", n, fib_dict)
    return fib_dict[n]
```

```
>>> fibonacci2(6)
start n = 6 {0: 1, 1: 1}
start n = 5 {0: 1, 1: 1}
start n = 4 {0: 1, 1: 1}
start n = 3 {0: 1, 1: 1}
start n = 2 {0: 1, 1: 1}
end n = 2 {0: 1, 1: 1, 2: 2}
end n = 3 {0: 1, 1: 1, 2: 2, 3: 3}
end n = 4 {0: 1, 1: 1, 2: 2, 3: 3, 4: 5}
end n = 5 {0: 1, 1: 1, 2: 2, 3: 3, 4: 5, 5: 8}
end n = 6 {0: 1, 1: 1, 2: 2, 3: 3, 4: 5, 5: 8, 6: 13}
```

- See also this useful animation:


## Pushing Recursion Depth to the Limit

```
>>> fibonacci2(990)
571829406815633979529643697006273045106845980748991112071
673038743714031497887739023091610769764627307772654802298
784361803421747114571265690519449915873452164193174293407
940201977897716937097604164288130909
>>> fibonacci2(1000)
Traceback (most recent call last):
# removed most of the error message
fib dict[n] = fibonacci2(n-1)+fibonacci2(n-2)
RuntimeError: maximum recursion depth exceeded
```

- What the \$\#* \& is going on?


## Python Recursion Depth

- While recursion provides a powerful and very convenient means to designing and writing code, this convenience is not for free.
- Each time we call a function, Python (and every other programming language) adds another "frame" (memory environment) to the current one. This entails allocation of memory for local variables, function parameters, etc.
- Nested recursive calls, like the one we have in fibonacci2, build a deeper and deeper "stack" of such frames.
- Most programming languages' implementations limit this recursion depth. Specifically, Python has a nominal default limit of 1,000 on recursion depth. However, the user (you, that is), can modify the limit (within reason, of course).


## Changing Python Recursion Depth

- You can import the Python sys library, find out what the limit is, and also change it.

```
>>> import sys
>>> sys.getrecursionlimit() # find recursion depth limit
1000
>>> sys.setrecursionlimit(20000) # change limit to 20,000
>>> fibonacci2(3000)
664390460366960072280217847866028384244163512452783259405579765542621214
1612192573964498109829998203911322268028094651324463493319944094349260190
4534272374918853031699467847355132063510109961938297318162258568733693978
4373527897555489486841726131733814340129175622450421605101025897173235990
66277020375643878651753054710112374884914025268612010403264702514559895667
590213501056690978312495943646982555831428970135422715178460286571078062467
510705656982282054284666032181383889627581975328137149180900441221912485637
512169481172872421366781457732661852147835766185901896731335484017840319755
9969056510791709859144173304364898001 # hurray
```


## Reversed Order of Calls

- As you have probably understood, Python evaluates expressions from left to right (except for when otherwise dictated by precedence of operators).
- Suppose we changed the order of calls inside fibonacci2: first we call $\mathrm{n}-2$, then $\mathrm{n}-1$.

```
def fib2_reverse(n, fib_dict):
    if n not in fib_dict:
        res = fib2_reverse(n-2, fib_dict) + fib2_reverse(n-1, fib_dict)
        fib_dict[n] = res
    return fib_dict[n]
```

- HW: How does the recursion tree look like now? Recursion depth? Time complexity?


## Fibonacci: Iterative (Non Recursive) Solution

- We saw that memoization improved the performance of computing Fibonacci numbers dramatically (the function fibonacci2).
- We now show that to compute Fibonacci numbers, the recursion can be eliminated altogether.
- This time, we will maintain a list data structure, denoted fibb. Its elements will be fibb[0], fibb[1], fibb[2], ..., fibb[n] ( $n+1$ elements altogether for computing $F_{n}$ ).
- Upon generating the list, all its values are set to 0 .
- Next, we initialize the values fibb[0] = fibb[1] = 1 .
- And then we simply iterate, determine the value of the $k$-th element, fibb[k], after fibb[k-2], and fibb[k-1] were already determined.
- No recursion implies no nested function calls, hence reduced overhead (and no need to confront Python's recursion depth limit :-).


## Iterative Fibonacci Solution: Python Code

```
def fibonacci3(n):
    """ Iterative Fibonacci ,
        keeps all values in a list """
if n<=1:
    return 1
    else:
    fib_list = [None for i in range(n+1)]
    fib_list[0] = fib_list[1] = 1 # initialize
    for k in range(2, n+1):
        fib_list[k] = fib_list[k-1] + fib_list[k-2]
    return fib_list[n]
```


## Recursive vs. Iterative: Timing

- Let us now do some performance comparisons: fibonacci2 vs. fibonacci3:

```
>>> import sys
>>> sys.setrecursionlimit(20000)
>>> elapsed("fibonacci2(2000)")
0.003454221497536104
>>> elapsed("fibonacci3(2000)")
0.0008148609599825107
```

- As we mentioned already, recursive calls require maintenance operations and memory allocation ("frames"), thus tend to have a negative influence on running time, compared to the analogous iterative solution.


## Iterative Fibonacci Solution Using O(1) Memory

- No, we are not satisfied yet.
- Think about the algorithm's execution flow. Suppose we have just executed the assignment fib_list[4] = fib_list[3] + fib_list[2]. This entry will subsequently be used to determine fib_list[5] and then fib_list[6]. But then we make no further use of fib_list[4]. It just lies, basking happily, in the memory.
- The following observation holds in "real life" as well as in the "computational world":

Time and space (memory, at least a computer's memory) are important resources that have a fundamental difference: Time cannot be re-used, while memory (space) can be.

## Iterative Fibonacci Reusing Memory

- At any point in the computation, we can maintain just the last two values, use them to compute the next one, and get rid of the "earlier" one.
- In practice, we will maintain two variables, prev and curr. Every iteration, those will be updated. Normally, we would need a third variable next for keeping a value temporarily. However Python supports the "simultaneous" assignment of multiple variables (first the right hand side is evaluated, then the left hand side is assigned).


## Iterative Fibonacci Solution: Python Code

```
def fibonacci4(n):
    """ Fibonacci in O(1) memory """
    if n<=1:
        return 1 # base case
    else:
        prev = 1
        curr = 1
        for i in range(n-1): # n-1 iterations (count carefully)
            curr, prev = prev+curr, curr
                # simultaneous assignment
            return curr
```

>>> for $i$ in range $(0,7): \#$ sanity check
print(fibonacci4(i))
1
1
2
3
5
8
13

## Iterative Fibonacci Code, Reusing Memory: Performance

- Reusing memory can surely help if memory consumption is an issue. Does it help with runtime as well?

```
>>> elapsed("fibonacci3(100000)",number=10)
6.150758999999999
>>> elapsed("fibonacci4(100000)", number=10)
1.8084930000000004
```

- We see that there is about 50-70\% saving in time, although both solutions work in $\mathrm{O}(\mathrm{n})$ time (assuming arithmetic operations take $\mathrm{O}(1)$ ). Not dramatic, but significant in certain circumstances.
- The fibonacci4 function uses $\mathrm{O}(1)$ memory vs. the $\mathrm{O}(\mathrm{n})$ memory usage of fibonacci3 (again, disregarding the size of the numbers themselves).


## Closed Form Formula

- And to really conclude our Fibonacci excursion, we note that there is a closed form formula for the $n$-th Fibonacci number,

$$
F_{n}=\frac{\left(\frac{1+\sqrt{5}}{2}\right)^{n+1}-\left(\frac{1-\sqrt{5}}{2}\right)^{n+1}}{\sqrt{5}}
$$

- You can verify this by induction. You will even prove it in the discrete mathematics course.
- This function seems to have neither recursion nor loops, thus runs in O(1) time, ignoring arithmetical operations' complexity. However, there are hidden loops in the ** operation, and when $n$ is large, this is not negligible.

```
def fibonacci5(n):
    return round (((1+5**0.5)** (n+1)-(1-5**0.5)**(n+1))/(2** (n+1)*5**0.5))
```


## Closed Form Formula: Code, and Danger

```
def fibonacci5(n):
    return round(((1+5**0.5)** (n+1)-(1-5**0.5)** (n+1))/(2** (n+1)*5**0.5))
# sanity check
>>> for i in range(10, 60, 10):
    print(i, fibonacci4(i), fibonacci5(i))
10 89 89
2010946 10946
30 1346269 1346269
40 165580141 165580141
5020365011074 20365011074
```

- However, being aware that floating point arithmetic in Python (and other programming languages) has finite precision, we are not convinced, and push for larger values:


## Closed Form Formula: Code, and Danger

- However, being aware that floating point arithmetic in Python (and other programming languages) has finite precision, we are not convinced, and push for larger values:

```
>>> for i in range(40, 90):
    if fibonacci4(i) != fibonacci5(i)
    print(i, fibonacci4(i), fibonacci5(i))
    break
```

70308061521170129308061521170130

Bingo!

## Reflections: Memoization, Iteration, Memory Reuse

- In the Fibonacci numbers example, all the techniques above proved relevant and worthwhile performance wise. These techniques won't always be applicable for every recursive implementation of a function.
- Consider quicksort as a specific example. In any specific execution, we never call quicksort on the same set of elements more than once (think why this is true).
- So memoization is not applicable to quicksort. And replacing recursion by iteration, even if applicable, may not be worth the trouble and surely will result in less elegant and possibly more error prone code.
- Even if these techniques are applicable, the transformation is often not automatic, and if we deal with small instances where performance is not an issue, such optimization may be a waste of effort.

