CSC148H Week 11

Ilir Dema, Michael Miljanovic

Summer 2021

A Fib Function

You know how to trace and explain a function like this.

```
def fib(n: int) -> int:
    """Return the nth fibonacci number"""  if n < 2:
        return n
    else:
        return fib(n - 1) + fib(n - 2)
```

A Fib Function…

- Unfortunately, the time taken to run the function is prohibitive when n gets large
- How prohibitive?
  - I have another version of the function that counts the number of function calls required to compute fib(n)
  - We'll see that the number of function calls grows very quickly

    . . .

Memoization

- The problem is that the **same** recursive calls are repeated over and over, redoing the same work
- A general technique for avoiding this problem is called **memoization**
- It involves storing the results of recursive calls, and looking up those results rather than setting off a chain of recursive calls again

Memoized Fib

```
def fib_mem(n: int) -> int:
    """Return the nth fibonacci number"""
    known = {}
    def fib_helper(m: int) -> int:
        if m not in known:  if
            m < 2:
                known[m] = m
            else:
                known[m] = fib_helper(m - 1) + fib_helper(m - 2)
                return known[m]
    return fib_helper(n)
```

Memoized Fib...

- It's a lot faster now
- But try fib_mem(1000)
- Python does have a way to change the maximum recursion depth, but that just delays the problem
- We can use an iterative algorithm that avoids recursion entirely
- Strategy: solve for small values of n first, then work up to larger values — this is what the memoization does too!

Dynamic Programming Fib

```
def fib_dyn(n: int) -> int:
    """Return the nth fibonacci number"""
    known = {0: 0, 1: 1}
    for m in range(2, n + 1):
        known[m] = known[m - 1] + known[m - 2]
    return known[n]
```

Dynamic Programming Fib...

- **Dynamic Programming** is a bottom-up approach that avoids recursion
- Useful when recursion causes the same subproblems to be solved repeatedly

- For Fib, the dictionary is unnecessary
- Just keep the previous and current Fib, and use them to calculate the next Fib. Try it!

When NOT to Use Recursion

- Two scary situations for recursion
  - When the parameter value can be very large (stack overflow)
  - When recursive calls solve the same problems over and over
- It’s generally advised to avoid recursion in these cases

Runtime Analysis

- Measuring elapsed runtime is one way to get a sense of an algorithm's efficiency
- However, runtime depends on the computer on which the program is run, the programming language being used, and lots of other factors
- Instead, we will characterize time efficiency in a way that
  - Is independent of the particular computer, and
  - Ignores small differences between algorithms

Runtime Analysis…

- A **step** is a basic unit of computation that can be carried out in a fixed amount of time
- We want to determine the number of steps that an algorithm takes as a function of its input size
- How we define input size depends a lot on the problem
  - e.g. for the sorting problem, the input size is the number of items to sort
- Typically the input size is the number of elements in the input
- For a given algorithm, we'll use the function $T(n)$ to denote the number of steps that the algorithm takes on input size n

Segment-Sum

```
def max_segment_sum(lst: list) -> int:  '''Return
    maximum segment sum of lst.'''  max_so_far =
    0
    for lower in range(len(lst)):
        for upper in range(lower, len(lst)):
            cur_sum = 0
            for i in range(lower, upper + 1):
                cur_sum = cur_sum + lst[i]
            max_so_far = max(max_so_far, cur_sum)
    return max_so_far
```

Analyzing the Segment-Sum Algorithm

- Question: how many times is cur_sum = cur_sum + lst[i] executed? (This is the same as asking for the number of iterations of the inner loop)
- The outer loop executes n times
- The middle loop is executed at most n times for each iteration of the outer loop
- So, the middle loop executes at most n^2 times
- The inner loop is executed at most n times for each iteration of the middle loop
- So, the inner loop executes at most n^3 times
- Similarly, cur_sum = 0 and max_so_far = ... are executed at most n^2 times
- Conclusion: an upper bound on the number of steps we execute is $n^3 + 2n^2$

Analyzing Segment-Sum Algorithm...

n	n^3	$2n^2$
10	1000	200
20	8000	800
30	27000	1800
40	64000	3200
50	125000	5000
100	1000000	20000
200	8000000	80000
300	27000000	180000
400	64000000	320000
500	125000000	500000
1000	1000000000	2000000

Observation: as n increases, the n^3 term in $n^3 + 2n^2$ comes to dominate, and $2n^2$ doesn't contribute much to $T(n)$

Big Oh

- Even if we had $T(n) = n^3 + 4n^2$, or $T(n) = n^3 + 50n^2$, when n becomes large, the n^2 term will be much smaller than the n^3 term
- To measure the efficiency of an algorithm, we focus only on the approximate number of steps it takes
- We are not concerned with deriving an exact value for $T(n)$, so we ignore its constant factors and nondominant terms
- Big Oh notation makes this idea precise

Big Oh...

- Say we have an algorithm whose running time on input of size n is $f(n)$
- Three requirements:
  - We want to bound $f(n)$ from above by $g(n)$
  - We want $g(n)$ to be a reasonable estimate; that is, it only "overestimates" by a constant c
  - We only require $g(n)$ to be such an estimate for sufficiently large values of n (since we don't care about small problem instances)
- This all amounts to requiring that $f(n) \leq cg(n)$ for all $n >= n_0$, and some positive constant c
- We then say that $f(n) = O(g(n))$

Properties of Big Oh

- Constant factors disappear
  - e.g. $6n$ and $n/2$ are both $O(n)$
- Lower-order terms disappear
  - e.g. $n^5 + n^3 + 6n^2$ is $O(n^5)$
- We can often just look at the loop structure of a program to determine its growth rate

Big Oh Bounds

- Big oh gives us an upper bound on the time that our algorithm takes to execute
- It gives no guarantee that the bound is "close" to what actually happens
- It's equally valid to say that the segment-sum code is $O(n^3)$, $O(n^6)$, $O(2^n)$, etc.
- However, saying that it is $O(n^3)$ gives us the most useful information
- $O(n^3)$ is a "tight bound" (i.e. most accurate bound): there is no smaller function q for which it is still $O(q)$

Exponential Cliff

- An important divide is the one between polynomial algorithms and exponential algorithms
- If there is no polynomial-time algorithm to solve a problem, we can solve it only for very small instances
- Even if computer speeds keep increasing, exponential algorithms explode too quickly with problem size to be feasible
- Problems that take more than $O(k^n)$ for $k > 1$ are considered intractable to solve