6.00.1.x - Week 6 - Algorithmic Complexity

October 22, 2020

1 Video 1: Program Efficiency

- Normally there is a trade off between **time and space** efficiency of a program.
- For the most part, we will focus on time efficiency.

- While there are many ways to implement a program, there are only a handful of algorithms that can solve a given problem.

How can I evaluate efficiency?

- Measure with a timer.
- Count the operations.
- Abstract notion of **order of growth**. This is the most appropriate way of assessing the impact of choices of algorithm in solving a problem.

Timing a program

[8]:
```python
import time
def c_to_f(c):
    return c*9/5 + 32
t0 = time.clock()
c_to_f(100000)
t1 = time.clock() - t0
print('t =', t0, ':', t1, 's,')
```

```
t = 1512.6299181 : 0.00010560000009718351 s,
```

```
C:\Users\pablo\anaconda3\lib\site-packages\ipykernel_launcher.py:4:
DeprecationWarning: time.clock has been deprecated in Python 3.3 and will be
removed from Python 3.8: use time.perf_counter or time.process_time instead
  after removing the cwd from sys.path.
C:\Users\pablo\anaconda3\lib\site-packages\ipykernel_launcher.py:6:
DeprecationWarning: time.clock has been deprecated in Python 3.3 and will be
removed from Python 3.8: use time.perf_counter or time.process_time instead
```

Problems with this idea

- Time varies between algorithms.
- Time varies between implementations.
- Time varies between computers.
- Time is not predictable based on small inputs.
- Time varies for different inputs but cannot really express a relationship between inputs and time.

Counting operations

- Assume certain steps (comparisons, assignments, accesing objects in memory, etc.) take a constant time.
- Then count the number of operations executed as function of size on input.

Problems with this idea

- Count depends on implementations.
- No real definition of which operations to count.

What would I like to do?

- I want to evaluate algorithm.
- I want to evaluate scalability.
- I want to evaluate in terms of input size.

- When choosing which case we'll study, we'll normally pick the worst case scenario.

Goals

- Evaluate efficiency when input is big.
- Express the growth of program's run time as input size grows.
- I want to put an upper bound on growth.
- I do not need to be precise: an order of growth will do.

Types of orders of growth

- Constant
- Linear
- Quadratic

- Logarithmic
- nlogn
- Exponential

2 Video 2: Big Oh Notation

- Big Oh notation measures an **upper bound on the asymptotic growth**, often called order of growth.
- Big Oh or O() is used to describe worst case:
  - Worst case occurs often and is the bottleneck when a program runs.
  - Express rate of growth of program relative to the input size.
  - Evaluate algorithm, not machine or implementation.

Let's see an example:

[10]:
```python
def fact_iter(n):
    """assumes n an int >= 0"""
    # 1 step here
    answer = 1
    # 5 steps inside the loop
    while n > 1:
        answer *= n
        n -= 1
    # 1 final step
    return answer
```

- The number of the steps is 1 + 5n + 1.
- The order of this algorithm is O(n), since the addition and multiplicative factors are not relevant.

Simplification examples

- n^2 + 2n + 2 = O(n^2)
- log(n) + n + 4 = O(n)
- 0.0001 n log(n) * 300n = O(n log n)
- 2 n^30 + 3^n = O(3^n)

We focus on the **dominant terms**.

- O(1) - constant
- O(log n) - logarithmic.
- O(n) - linear
- O(n log n) - loglinear.
- O(n^c) - polynomial.

- O(c^n) - exponential.

C is a constant, n is the size of the input.

- The higher I am in the hierarchy, the more efficient the algorithm is.

Law of Addition for O():

- Used with **sequential** statements.

O(n) + O(n^2) = O(n + n^2) = O(n^2)

Law of Multiplication for O():

- Used with **nested** statements/loops.

O(n) * O(n) = O(n^2) because the outer loop goes n times and the inner loop goes n times for every outer loop tier.

---

3 Video 3: Complexity Classes

Let's see examples of each one of the complexity classes.

O(1) - Constant

- Complexity independent of the input.
- Few interesting algorithms in this class.

O(log n) - Logarithmic

- Bisection search.
- Binary search of a list.

O(n) - Linear

- Searching a list in sequence to see if an element is present.
- Add characters of a string, assumed to be composed of decimal digits.
- factorial_iterative() and factorial_recursive() are both O(n). One does n for loops, the other does n function calls.

Log-Linear

- Very commonly used log-linear algorithm is merge sort.

4 Video 4: Analyzing Complexity

Polynomial

- Most common is quadratic.
- Commonly occurs when we have nested loops or recursive function calls.

5 Video 5: More Analyzing Complexity

Exponential

- Recursive functions where more than one recursive call for each size of problem.
- Many important problems are inherently exponential. So sometimes, we rather approximate solutions quickly than finding the most accurate guess.

6 Video 6: Recursion Complexity

- Interesting comments on methods that appear to be of a given order of growth with respects to the length of its contents, and a different order of growth with respects to its actual length. Re-Check.
- Fibonacci analyzed again.
  - Iterative is O(n).
  - Recursive is O(c^n).
- Interesting analysis regarding the complexity that comes with lists vs the complexity that comes with dictionaries.

7 Video 7: Search Algorithms

Search algorithm: Method for finding an item or group of items with specific properties within a collection of items.

- Collection could be implicit. Example: find square root as a search problem.
- Collection could be explicit. Example: Is a student record in a stored collection of data?

Searching algorithms

- Linear search
  - Brute force search
  - List does not have to be sorted.
  - Complexity is O(n), where n is len(L).
  - Accesing a list item is constant time, regardless the list being homogeneous or heterogeneous. Explanation in video.
- Bisection search.
  - List MUST be sorted to give correct answer.
  - Different implementations of the algorithm are possible.

8 Video 8: Bisection Search

- Linear search on sorted list is still O(n) - still might have to go all the way through the list.

Bisection search

- Complexity is O(log n) - where n is len(L).

Implementation 1

- Recursive way.
- Making copies of the list adds up to a large complexity.
- O(n log n). O(n) for a tighter bound because length of list is halved each recursive call.

Implementation 2

- Define a function and a helper function.
- Instead of copying the list, just remember the indexes that you used in the last step in order to do the recursive call.
- O(log n).

- Using linear search, search for an element is O(n) - regardless of whether the list is sorted or not.
- Using binary search, search for an element is O(log n), assuming the list is sorted.

When does it make sense to sort first then search?

- When sorting is less than O(n). And that is **never** true.

Amortized cost

- Maybe I sort the list once and perform multiple searches.

9 Video 9: Bogo Sort

- Randomly assign the elements in a certain order and check whether the elements are in order or not.
- Complexity of the bogo sort: O(?). It is unbounded.

10 Video 10: Bubble Sort

- Compare consecutive pairs of elements.
- Swap elements in pair such that smaller is first.
- When reach end of list, start over again.
- Stop when no more swaps have been made.
- Complexity = O(n^2) where n is len(L).

11 Video 11: Selection Sort

- First step
  - Extract minimum element.
  - Swap it with element at index 0.
- Subsequent step.
  - Remaining sublist, extract minimum element.
  - Swap it with the element at index 1.
- Keep the left portion of the list sorted
  - At ith step, first i elements in list are sorted.
  - All other elements are bigger than first i elements.

Is there a loop invariant?

- given prefix of list L[0:i] and suffix L[i+1:len(L)], then prefix is sorted and no element in prefix is larger than smallest element in suffix.
- I prove this is true by induction.

1. base case: prefix empty, suffix whole list – invariant true.
2. induction step: move minimum element from suffix to end of prefix. Since invariant true before move, prefix sorted after append.
3. when exit, prefix is entire list, suffix empty, so sorted.

Implementation

- I don't want to copy the list! It is very inefficient.
- Complexity is O(n^2) where n is len(L). Timing wise probably more efficient than the bubble sort.

12 Video 12: Merge and Sort

- Use a divide and conquer approach.

1. If list is of length 0 or 1, already sorted.
2. If list has more than one element, split into two lists and sort each.
3. Merge sorted sublists
    1. Look at first element of each, move smaller to end of the result.
    2. When one list empty, just copy rest of other list.

Example of merging (see slide 36/42).

Complexity of the merging: linear in the length of the lists.

Implementation

The merge sort algorithm itself is recursive.

Final complexity analysis

- At first recursion level
    - n/2 elements in each list.
    - O(n) + O(n) = O(n) where n is len(L).
- At second recursion level
    - n/4 elements in each list.
    - two merges: O(n) where n is len(L).
- Each recursion level is O(n).
- Dividing list in half with each recursive call. O(log n) where n is len(L).
- Overall complexity is **O(n log n)) where n is len(L)**.

O(n log n) is the fastest a sort can be.