Programmieren I (Python)

Christian Osendorfer

2023-12-15

Efficiency

Efficient Programs?

- Computers are fast and getting faster - so maybe efficient programs don’t matter?
- Data sets can be very large (e.g., in 2014, Google served 30,000,000,000,000 pages, covering 100,000,000 GB - how long does it tak to search throught these with brute force?)
- Simple solutions may simply not scale with size in acceptable manner.
- separate **time and space efficiency** of a program.
    - tradeoff between them:
        - can sometimes pre-compute results that are stored.
        - then use “lookup” to retrieve.
- focus on time efficiency.

Efficient Programs

Challenges in understanding efficiency of solutions to a computational problem:

- A program can be implemented in many different ways.
- You can solve a problem using only a handful of different algorithms.
- Separate **choices of implementation** from **choices of more abstract algorithm**.
- Measure with a timer
- Count the operations
- Abstract notation of order of growth

Timing a program

- Use `time` module in python!

```
>>> from time import perf_counter
>>> def longrunning_function():
...     for i in range(1, 11):
...         time.sleep(i / i ** 2)
...
>>> start = perf_counter() # python 3.7: perf_counter_ns
>>> longrunning_function()
>>> end = perf_counter()
>>> execution_time = (end - start)
```

Timing a program

Timing a program is **inconsistent**.

- Our goal: evaluate different algorithms.
- But:
  - *Runtime* varies between implementations and computers.
- Runtime varies for different inputs but cannot really express a relationship between inputs and time.

Counting operations

- Assume the follwing steps take constant time:
    - mathematical operations
    - comparisons
    - assignments
    - accessing objects in memory
- Then count the number of operations executed as function of size of input.

```python
def c_to_f(c):
  return c*9/5 + 32 # 3 operations
```

Counting operations

- Counting operations is better than timing, but still depends on detailed implementation of algorithm.
  - It is also not clear which operations to count.
- Count varies for different inputs and can come up with a relationship between inputs the the count.

A better way?

- Focus on the idea of counting operations in an algorithm!
  - Without worrying about small variations in implementation.
- Focus on how algorithms perform when size of problem gets arbitrarily large.
- Relate time needed to complete a computation against the size of the input to the problem.
- Decide on what to measure, given the actual number of steps may depend on specific instance of input.

Different inputs, different program runs

- Express efficiency in terms of size of input!
  - What is the input? A number? A list?

```
def search_elem(L, e):
  """Search for element e in list L."""
  for i in L:
    if i==e:
      return True
  return False
```

- If e is first element in L: best case.
- If e is not in L: worst case.
- If looking through about half of L: average case.
- How to measure this behavior in a general way? Focus on the worst case!

Orders of growth

- Evaluate program’s efficiency when input is very big.
  - What does it mean that input is very big?
- Express the growth of program’s runtime as input of size grows.
- Put an upper bound on growth: as tight as possible.
- No need to be precise: **order of** instead of **exact**.
- That is, only consider the largest factor in runtime.
  - what part of the program takes the longest to run?
- Goal: Tight upper bound on growth of runtime, as a function of input size, in worst case.

Big-Oh notation

- Big-Oh (that is: $O(\cdot)$)) notation measures an **upper bound on the asymptotic growth**.
  - often called *order of growth*.
- $O(\cdot)$ is used to describe the worst case
  - worst case occurs often and is the bottleneck when a program runs.
  - express rate of growth of a program relative to input size.
  - evaluate algorithm **not** machine or implementation.
- Focus on dominant terms:
  - $n^2 + n + 1$: $O(n^2)$.
  - $100000n^2 - 1000000000000n$: $O(n^2)$.
  - $0.00000001n \log n + 100000000n$: $O(n \log n)$.
  - $n^{100} + 3^n$: $O(3^n)$.

Law of addition

- Used with sequential statements.
- $O(f(n)) + O(g(n)) = O(f(n) + g(n))$.

```
for i in range(n): # O(n)
  l = i*i
for j in range(n*n): # O(n^2)
  k = j*j
```

- is $O(n^2)$ because of dominant term.

Law of multiplication

- Used with **nested** statements (e.g. loops)
- $O(f(n)) * g(n)) = O(f(n) * g(n))$.

```
for i in range(n): # O(n)
  for j in range(n): # O(n)
    l = i*j
```

- is $O(n^2)$: The outer loop goes n times over the inner loop.

Complexity classes

- $O(1)$: Constant running time.
- $O(\log n)$: Logarithmic running time.
- $O(n)$: Linear running time.
- $O(n \log n)$: Log-linear running time.
- $O(n^c)$: Polynomial running time (c is a constant).
- $O(c^n)$: Exponential running time (c is a constant).

Linear Complexity

- **Simple** iterative algorithms are typically linear in complexity.

```python
def linear_search(L, e):
  """Search for element e in list L.
  L is **not** sorted."""
  found = False
  for i in range(len(L)):
    if e == L[i]:
      found = True
  return found
```

- Must look through all elements to decide if it’s **not** there.
- Assume:
  - Constant time to access list element at index `i`.
  - Constant time to compare two elements.
- $O(n)$, where n is the length of `L`.

Linear Complexity

```
def fact_iter(n):
  """Compute factorial of n>= 0, iteratively."""
  prod = 1
  for i in range(1, n+1):
    prod *= i
  return prod
```

- Assumes constant time for multiplication of two (large?!?) integers!

Linear Complexity

```python
def fact_recur(n):
  """Recursive factorial of n, n >= 0"""
  if n <= 1:
    return 1
  else:
    return n * fact_recur(n-1)
```

- Iterative and recursive implementations are the **same** order of growth **in this task**!
  - Recursive version may run a bit slower due to overhead of function calls.

Linear complexity

```
def virfib_iter(n):
  if n == 0:
    return 0
  elif n == 1:
    return 1
  else:
    fib_i = 0
    fib_ii = 1
    for i in range(n-1):
      tmp = fib_i
      fib_i = fib_ii
      fib_ii = tmp + fib_i
    return fib_ii
```

- Iterative Virahanka-Fibonacci is linear in n.

Quadratic Complexity

```python
def isSubset(L1, L2):
  """Check if L2 is
  proper subset of L1."""
  for e1 in L1:
    matched = True
    for e2 in L2:
      if e1 == e2:
        matched = True
        break
    if not matched:
      return False
  return True
```

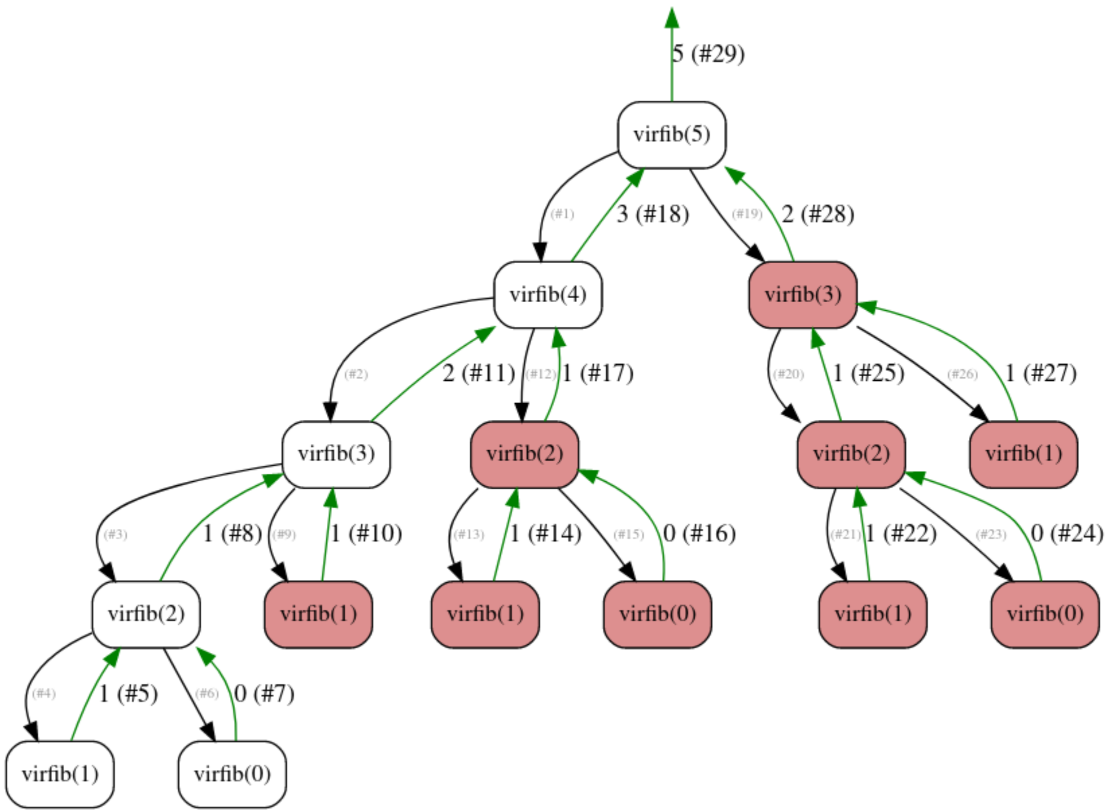
- Outer loop executed `len(L1)` many times.
- Each iteration will execute inner loop **up to** `len(L2)` times, with constant number of operations.
- $O(\mathrm{len}(\mathrm{L1}) \times \mathrm{len}(\mathrm{L2}))$.

Exponential Complexity

- Towers of Hanoi, consisting of n discs. The pegs are labeled A, B and C.
  - A is the starting peg and B is the goal peg.
- Denote the number of necessary moves for n discs with t_n. That is, for a tower consisting of $n-1$ discs, we would use t_{n-1} to denote the number of moves to solve this instance of the problem.
- Solve the problem by first moving $n-1$ discs from A to C, then move the biggest disc from A to B, then move $n-1$ discs from C to B.
- This gives a recurrence relation for t_n!
  - $t_n = t_{n-1} + 1 + t_{n-1} = 2t_{n-1} + 1$
  - Unfold recursively! $t_n = 2t_{n-1} + 1 = 2(2t_{n-2} + 1) + 1 = 2(2(2t_{n-3} + 1) + 1) + 1 = \cdots$
  - $t_n = 2^n - 1$.
- An example of the Master Theorem.

Exponential complexity

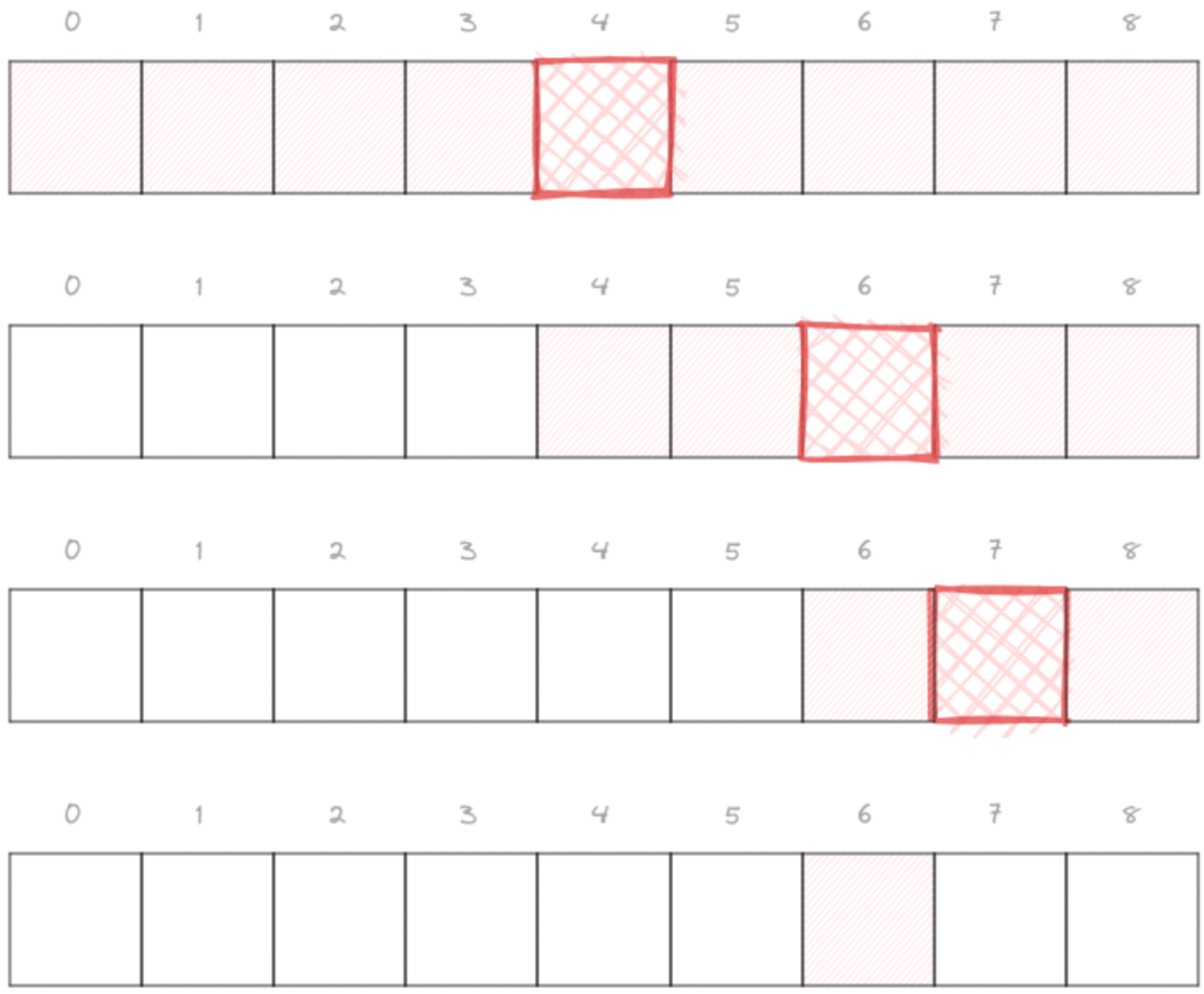
```python
def virfib(n):
    """Recursive Virahanka-Fibonacci number for n >= 1."""
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return virfib(n-1) + virfib(n-2)
```



Logarithmic Complexity

- How can it be possible to spend less then n units of work for a list with n elements?
- The list (or in general, the data) must have some structure, apriori.
- For example, searching an element in a **sorted** list.
- Because the list is sorted, one can determine in which half of the full list the element must be.
  - Simply compare to the **middle element** of the full list.
  - The same process can now be applied to the selected half part of the full list.
    - Recursion!

Logarithmic Complexity



Logarithmic Complexity

- The element that is searched for may be found during the splitting process (i.e. it is the element that (roughly) halfs the list).
- In worst case, how many steps does it take until it is clear that the element is **not** in the list?
  - At every step, the size of the relevant list is halfed.
  - How many steps i until only one element is left?
  - Solve $1 = n/2^i$
  - $i = \log n$ (base of $\log$ is not important!)

Logarithmic complexity – binary search ?

```
def binary_search(L, e):
  """Find e in L, L is sorted!"""
  if L == []:
    return False
  elif len(L) == 1:
    return L[0] == e
  else:
    half = len(L) // 2
    if L[half] >= e:
      return binary_search(L[:half], e)
    else:
      return binary_search(L[half:], e)
```

- However this is not $O(\log n)$. It is $O(n \log n)$.
  - Each recursive call uses the **slice** operator: A full copy of the elements is generated.
  - The cost is due to how Python handles slices! This might be different in a different language!!

Logarithmic complexity – binary search !

```
def binary_search(L, e):
  def _bin_search_helper(L, e, low, high):
    if high == low:
      return L[low] == e
    mid = (low + hight)//2
    if L[mid] == e:
      return True
    elif e < L[mid]:
      if low == mid:
        return False
      else:
        return _bin_search_helper(L, e, low, mid-1)
    else:
      return _bin_search_helper(L, e, mid+1, high)
  if len(L) == 0:
    return False
  else:
    return _bin_search_helper(L, e, 0, len(L) - 1)
```

Searching a list

- Linear search takes $O(n)$.
- Binary search takes $O(\log(n))$.
  - But the list must be sorted upfront!
  - How expensive is sorting?
  - Is it reasonable to spend that work before searching?
- For one-time (or a few-time) search, sorting before binary search is not preferable!
  - $O(\mathrm{SORT}) + O(\log n) < O(n)$, i.e. sorting needs to be better than linear!
  - This is not possible! (Because we need to look at each element at least once).
- Better: sort **once**, search very often!
  - $O(\mathrm{SORT}) + K \times O(\log n)$: For very large K, cost for sorting is irrelevant **iff**?
- Best sorting algorithms (based on comparisons!) are $O(n \log n)$!

Sorting

Bubble sort

- Compare consecutive list elements. If order is incorrect, fix it locally.
  - **Fixing** means swapping two consecutive elements.
  - Larger elements *bubble* upwards.
- Largest **unsorted** element is at the end of the list after one pass over the list!
  - At most n passes are necessary, where every pass costs at most $O(n)$.
  - $O(n * n)$.
  - One can stop algorithm, if a pass happens without swapping elements.

Bubble Sort

```
def bubble_sort(L):
  swap = True
  while swap:
    swap = False
    for j in range(0, len(L)-1):
      if L[j] > L[j+1]:
        swap = True
        L[j], L[j+1] = L[j+1], L[j]
```

- Inner loop is doing comparisions (always n many!)
  - An element can bubble up!
- Outer loop does multiple passes
  - In this implementation, potentially way less than n many!

Selection Sort

- How can one think more structured about sorting?
- Assume that the list to be sorted follows the following assumptions:
  - All elements from index 0 to some index $i-1$ are properly sorted (ascending) and
  - These elements are all smaller or equal to the unsorted elements from index i on.
- How does that help?
  - How can one determine the next element that should be added to the sorted part?
- Find the smallest element from index i on (the remaining list) and put it at index i.
- Start the algorithm with $i = 0$!

Selection Sort

```python
def selection_sort(L):
  prefix_idx = 0
  while prefix_idx != len(L):
    for i in range(prefix_idx, len(L)):
      if L[i] < L[prefix_idx]:
        L[prefix_idx], L[i] = L[i], L[prefix_idx]
    prefix_idx += 1
```

- Outer Loop needs n steps.
- Inner Loop is $O(n) - \text{prefix\_idx}$ steps.
- $O(n^2)$.

Insertion Sort

- Maybe too many assumptions need to hold? What if only the first $i - 1$ elements of a list are sorted?
  - No longer needed that these are also smaller than the unsorted elements!
  - So we don't need to find the next smallest element! Maybe we can save a costly (linear) search?
- Simply take the next element (at index i) and properly insert it at the right position!
  - **Sorting by *insertion*** !
- Any issues?
  - **Properly inserting** is costly! Need to find the right spot and move existing elements by one?!
- Thoughts?
  - Binary search and Linked Lists??

Insertion Sort

```
def insertion_sort(L):
  i = 0
  while i < len(L):
    j = i
    while j > 0:
      if b[j-1] > b[j]
        b[j-1], b[j] = b[j], b[j-1]
      j -= 1
    i += 1
```

- Still $O(n^2)$.
- ???

Merge Sort

- Binary Search worked well because it divided the problem in two equal halfs.
- What would that be for sorting?
  - Two half lists, sorted. Then merge these sorted lists.
  - Amazing instance for **Divide and Conquer** algorithms.
    - The resulting sublists of one recursive call are ordered!

```python
def merge_sort(L):
  if len(L) < 2:
    return L[:]
  else:
    middle = len(L) // 2
    left = merge_sort(L[:middle])
    right = merge_sort(L[middle:])
    return merge(left, right)
```

- Note the copies! Every **recursive layer** costs about n steps!
  - But how deep is the recursion tree?
  - And how costly is `merge`?

Merge Sort

```
def merge(left, right):
  result = []
  i, j = 0, 0
  while i < len(left) and j < len(right):
    if left[i] < right[j]:
      result.append(left[i])
      i += 1
    else:
      result.append(right[j])
      j += 1
  while i < len(left):
    result.append(left[i])
    i += 1
  while j < len(right):
    result.append(right[j])
    j += 1
  return result
```

- Overall complexity is $O(n \log n)$!