Algorithmic Speed

BBM 101 - Introduction to Programming I

Hacettepe University
Fall 2016

Fuat Akal, Aykut Erdem, Erkut Erdem

Slides based on material prepared by E. Grimson, J. Guttag and C. Terman in MITx 6.00.1x
Measuring complexity

• Goals in designing programs
  1. It returns the correct answer on all legal inputs
  2. It performs the computation efficiently

• Typically (1) is most important, but sometimes (2) is also critical, e.g., programs for collision detection

• Even when (1) is most important, it is valuable to understand and optimize (2)
Computational complexity

• How much time will it take a program to run?
• How much memory will it need to run?

• Need to balance minimizing computational complexity with conceptual complexity
  – Keep code simple and easy to understand, but where possible optimize performance
How do we measure complexity?

- Given a function, would like to answer: “How long will this take to run?”
- Could just run on some input and time it.
- Problem is that this depends on:
  1. Speed of computer
  2. Specifics of Python implementation
  3. Value of input
- Avoid (1) and (2) by measuring time in terms of number of basic steps executed
Measuring basic steps

• Use a random access machine (RAM) as model of computation
  – Steps are executed sequentially
  – Step is an operation that takes constant time
    • Assignment
    • Comparison
    • Arithmetic operation
    • Accessing object in memory

• For point (3), measure time in terms of size of input
But complexity might depend on value of input?

```python
def linearSearch(L, x):
    for e in L:
        if e == x:
            return True
    return False
```

- If x happens to be near front of L, then returns True almost immediately
- If x not in L, then code will have to examine all elements of L
- Need a general way of measuring
Cases for measuring complexity

- **Best case**: minimum running time over all possible inputs of a given size
  - For linearSearch – constant, i.e. independent of size of inputs
- **Worst case**: maximum running time over all possible inputs of a given size
  - For linearSearch – linear in size of list
- **Average (or expected) case**: average running time over all possible inputs of a given size
- We will focus on worst case – a kind of **upper bound** on running time
def fact(n):
    answer = 1
    while n > 1:
        answer *= n
        n -= 1
    return answer

• Number of steps
  – 1 (for assignment)
  – 5*n (1 for test, plus 2 for first assignment, plus 2 for second assignment in while; repeated n times through while)
  – 1 (for return)
• 5*n+2 steps
• But as n gets large, 2 is irrelevant, so basically 5*n steps
Example

• What about the multiplicative constant (5 in this case)?

• We argue that in general, multiplicative constants are not relevant when comparing algorithms
Example

def sqrtExhaust(x, eps):
    step = eps**2
    ans = 0.0
    while abs(ans**2 - x) >= eps and ans <= max(x, 1):
        ans += step

    return ans

• If we call this on 100 and 0.0001, will take one billion iterations of the loop
  – Have roughly 9 steps within each iteration
def sqrtBi(x, eps):
    low = 0.0
    high = max(1, x)
    ans = (high + low)/2.0
    while abs(ans**2 - x) >= eps:
        if ans**2 < x:
            low = ans
        else:
            high = ans
        ans = (high + low)/2.0
    return ans

• If we call this on 100 and 0.0001, will take thirty iterations of the loop
  – Have roughly 10 steps within each iteration
• 1 billion or 9 billion versus 30 or 300 – it is size of problem that matters
Measuring complexity

• Given this difference in iterations through loop, multiplicative factor (number of steps within loop) probably irrelevant

• Thus, we will focus on measuring the complexity as a function of input size
  – Will focus on the largest factor in this expression
  – Will be mostly concerned with the worst case scenario
Asymptotic notation

- Need a formal way to talk about relationship between running time and size of inputs
- Mostly interested in what happens as size of inputs gets very large, i.e. approaches infinity
Example

def f(x):
    for i in range(1000):
        ans = i
    for i in range(x):
        ans += 1
    for i in range(x):
        for j in range(x):
            ans += 1

Complexity is $1000 + 2x + 2x^2$, if each line takes one step
Example

• $1000+2x+2x^2$
• If $x$ is small, constant term dominates
  – E.g., $x = 10$ then 1000 of 1220 steps are in first loop
• If $x$ is large, quadratic term dominates
  – E.g. $x = 1,000,000$, then first loop takes $0.000000005\%$ of time, second loop takes $0.0001\%$ of time (out of $2,000,002,001,000$ steps)!
Example

• So really only need to consider the nested loops (quadratic component)
• Does it matter that this part takes $2x^2$ steps, as opposed to say $x^2$ steps?
  – For our example, if our computer executes 100 million steps per second, difference is 5.5 hours versus 2.25 hours
  – On the other hand if we can find a linear algorithm, this would run in a fraction of a second
  – So multiplicative factors probably not crucial, but order of growth is crucial
Rules of thumb for complexity

• Asymptotic complexity
  – Describe running time in terms of number of basic steps
  – If running time is sum of multiple terms, keep one with the largest growth rate
  – If remaining term is a product, drop any multiplicative constants

• Use “Big O” notation (aka Omicron)
  – Gives an upper bound on asymptotic growth of a function
Complexity classes

• $O(1)$ denotes constant running time
• $O(\log n)$ denotes logarithmic running time
• $O(n)$ denotes linear running time
• $O(n \log n)$ denotes log-linear running time
• $O(n^c)$ denotes polynomial running time ($c$ is a constant)
• $O(c^n)$ denotes exponential running time ($c$ is a constant being raised to a power based on size of input)
Constant complexity

• Complexity independent of inputs
• Very few interesting algorithms in this class, but can often have pieces that fit this class
• Can have loops or recursive calls, but number of iterations or calls independent of size of input
Logarithmic complexity

• Complexity grows as log of size of one of its inputs
• Example:
  – Bisection search
  – Binary search of a list
def intToStr(i):
    digits = '0123456789'
    if i == 0:
        return '0'
    result = ''
    while i > 0:
        result = digits[i%10] + result
        i = i//10
    return result
Logarithmic complexity

def intToStr(i):
    digits = '0123456789'
    if i == 0:
        return '0'
    result = ''
    while i > 0:
        result = digits[i%10]
        + result
        i = i//10
    return result

• Only have to look at loop as no function calls
• Within while loop constant number of steps
• How many times through loop?
  – How many times can one divide i by 10?
  – $O(\log(i))$
Linear complexity

• Searching a list in order to see if an element is present
• Add characters of a string, assumed to be composed of decimal digits

```python
def addDigits(s):
    val = 0
    for c in s:
        val += int(c)
    return val
```

• $O(len(s))$
Linear complexity

• Complexity can depend on number of recursive calls

```python
def fact(n):
    if n == 1:
        return 1
    else:
        return n*fact(n-1)
```

• Number of recursive calls?
  – Fact(n), then fact(n-1), etc. until get to fact(1)
  – Complexity of each call is constant
  – $O(n)$
Log-linear complexity

• Many practical algorithms are log-linear
• Very commonly used log-linear algorithm is merge sort
• Will return to this
Polynomial complexity

• Most common polynomial algorithms are quadratic, i.e., complexity grows with square of size of input

• Commonly occurs when we have nested loops or recursive function calls
def isSubset(L1, L2):
    for e1 in L1:
        matched = False
        for e2 in L2:
            if e1 == e2:
                matched = True
                break
        if not matched:
            return False
    return True
def isSubset(L1, L2):
    for e1 in L1:
        matched = False
        for e2 in L2:
            if e1 == e2:
                matched = True
                break
        if not matched:
            return False
    return True

• Outer loop executed $\text{len}(L1)$ times
• Each iteration will execute inner loop up to $\text{len}(L2)$ times
• $O(\text{len}(L1)\times\text{len}(L2))$
• Worst case when $L1$ and $L2$ same length, none of elements of $L1$ in $L2$
• $O(\text{len}(L1)^2)$
Quadratic complexity

Find intersection of two lists, return a list with each element appearing only once

```python
def intersect(L1, L2):
    tmp = []
    for e1 in L1:
        for e2 in L2:
            if e1 == e2:
                tmp.append(e1)
    res = []
    for e in tmp:
        if not(e in res):
            res.append(e)
    return res
```
Quadratic complexity

```python
def intersect(L1, L2):
    tmp = []
    for e1 in L1:
        for e2 in L2:
            if e1 == e2:
                tmp.append(e1)
    res = []
    for e in tmp:
        if not(e in res):
            res.append(e)
    return res
```

- First nested loop takes $\text{len}(L1)\times\text{len}(L2)$ steps
- Second loop takes at most $\text{len}(L1)$ steps
- Latter term overwhelmed by former term
- $O(\text{len}(L1)\times\text{len}(L2))$
Exponential complexity

• Recursive functions where more than one recursive call for each size of problem
  – Towers of Hanoi

• Many important problems are inherently exponential
  – Unfortunate, as cost can be high
  – Will lead us to consider approximate solutions more quickly
def genSubsets(L):
    res = []
    if len(L) == 0:
        return [[]]  # list of empty list
    smaller = genSubsets(L[:-1])
    # get all subsets without last element
    extra = L[-1:]
    # create a list of just last element
    new = []
    for small in smaller:
        new.append(small + extra)
    # for all smaller solutions, add one with last element
    return smaller + new
    # combine those with last element and those without
Exponential complexity

```python
def genSubsets(L):
    res = []
    if len(L) == 0:
        return [[]]
    smaller = genSubsets(L[:-1])
    extra = L[-1:]
    new = []
    for small in smaller:
        new.append(small+extra)
    return smaller+new
```

- Assuming append is constant time
- Time includes time to solve smaller problem, plus time needed to make a copy of all elements in smaller problem
Exponential complexity

```python
def genSubsets(L):
    res = []
    if len(L) == 0:
        return [[]]
    smaller = genSubsets(L[:-1])
    extra = L[-1:]
    new = []
    for small in smaller:
        new.append(small + extra)
    return smaller + new
```

- But important to think about size of smaller
- Know that for a set of size k there are $2^k$ cases
- So to solve need $2^{n-1} + 2^{n-2} + \ldots + 2^0$ steps
- Math tells us this is $O(2^n)$
Complexity classes

• $O(1)$ denotes constant running time
• $O(\log n)$ denotes logarithmic running time
• $O(n)$ denotes linear running time
• $O(n \log n)$ denotes log-linear running time
• $O(n^c)$ denotes polynomial running time ($c$ is a constant)
• $O(c^n)$ denotes exponential running time ($c$ is a constant being raised to a power based on size of input)
Comparing complexities

• So does it really matter if our code is of a particular class of complexity?
• Depends on size of problem, but for large scale problems, complexity of worst case makes a difference
Constant versus Logarithmic

Constant (20) vs. Log

- constant(20)
- log

Input Size

Time
Observations

• A logarithmic algorithm is often almost as good as a constant time algorithm
• Logarithmic costs grow very slowly
Logarithmic versus Linear

Log vs. Linear

Time

Input Size
Observations

• Logarithmic clearly better for large scale problems than linear
• Does not imply linear is bad, however
Linear versus Log-linear

![Linear vs. Log-linear](image)

- *linear*
- *log-linear*
Observations

• While $\log(n)$ may grow slowly, when multiplied by a linear factor, growth is much more rapid than pure linear.

• $O(n \log n)$ algorithms are still very valuable.
Log-linear versus Quadratic

![Log-linear vs. Quadratic Chart]

- **log-linear**
- **quadratic**

- **Input Size**
- **Time**
Observations

• Quadratic is often a problem, however.
• Some problems inherently quadratic but if possible always better to look for more efficient solutions
Quadratic versus Exponential

• Exponential algorithms very expensive
  – Right plot is on a log scale, since left plot almost invisible given how rapidly exponential grows

• Exponential generally not of use except for small problems