CS331 Runtime Complexity

Algorithm Analysis

So far, our runtime analysis has been based on empirical data — i.e., runtimes obtained from coding and running our algorithms on larger and larger input and plotting the problem size against runtime to determine the growth in runtime.

This data is very sensitive to:

- platform (OS/compiler/interpreter)
- concurrent tasks
- implementation details (vs. high-level algorithm)

Also, we would like the actual function of the growth, not just a graph of the growth function.

Reframing the problem: Given an algorithm that takes *input size* ***n***, find a function ***T***(***n***) that describes the *runtime growth* of the algorithm.

input size might be:

- the *magnitude of the input value* (e.g., for numeric input)
- the *number of items* in the input (e.g., as in a list)

An algorithm may also be dependent on *more than one input*.

```
def sort(vals):                  def factorial(n):          def gcd(m, n):
# input size = len(vals)         # input size = n           # input size = (m, n)
```

fundamentally, runtime is determined by the *primitive operations* carried out during execution of the algorithm (in compiled code, by the interpreter, etc.). Let's count how many operations and the cost of each operation (system dependent, we will use constants) for factorial.

```
def factorial(n):              cost    times
  prod = 1                     c1      1
  for k in range(2, n+1):      c2      n-1
    prod *= k                  c3      n-1
  return prod                  c4      1
```

$T(n) = c_1 + (n - 1)(c_2 + c_3) + c_4$

Messy! Per-instruction costs are machine specific, and obscure big picture runtime trends.

Simplification #1: We can combine constants into new constants and then it is easy to see that runtime is *linear* w.r.t. input size.

$T(n) = c_1 + (n - 1)(c_2 + c_3) + c_4$

$T(n) = (c_2 + c_3)(n - 1) + (c_1 + c_4)$

$T(n) = C_1(n - 1) + C_2$ This is the equation of a line with slope C_1

Linear Search (on unsorted data) as a Case Study for Algorithm Analysis

Write pseudocode Linear Search. Then construct a run time analysis function T(n) by assigning a different constant (c1, c2, c3, etc) to each type of statement (the run time for statements of that type), and counting how many times each statement executes for an input size n. Then sum up the constants times the execution counts. You may need to define variables other than n if there are event-controlled iterations with an unknown number of loops.

For a more descriptive analysis we need to consider the various generic execution flows and input structures that result in those generic execution flows. There are 3 more descriptive resource use analyses: 1) worst case (usually used) 2) average case (sometimes used) 3) best case (hardly ever used). For your T(n) function from above determine the best case and worst case T(n)s.

Sorting as a Case Study for Algorithm Analysis

Write pseudocode InsertionSort. Then draw pictures for a sample run on 5 random numbers showing comparisons/swaps.

For insertion sort you wrote above, construct a run time analysis function T(n) by assigning a different constant (c1, c2, c3, etc) to each type of statement (the run time for statements of that type), and counting how many times each statement executes for an input size n. Then sum up the constants times the execution counts. You may need to define variables other than n if there are event-controlled iterations with an unknown number of loops.

For your T(n) function from above determine the best case, worst case and average case T(n)s.

Asymptotic Analysis – To simplify comparing the resource usage of different algorithms for the same problem

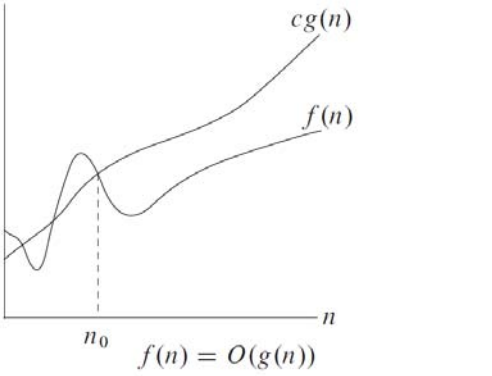
- ignore machine dependent constants
- look at the growth of T(n) as n-> infinity
- as you double n, what does T(n) do?? double?? square??

Big-O Notation

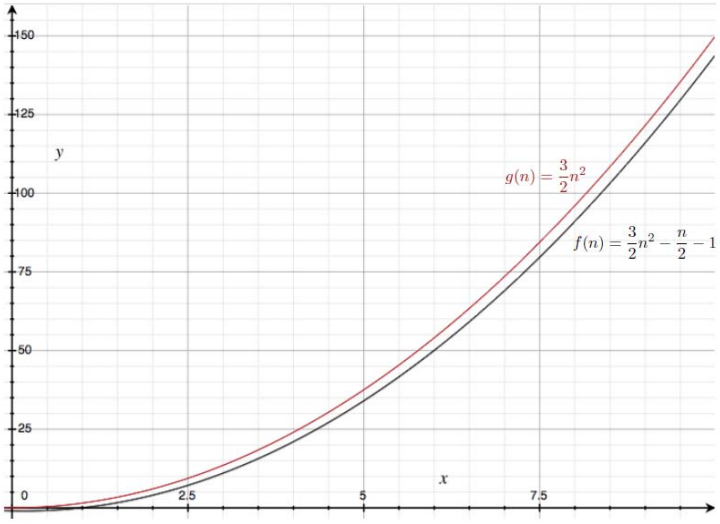
- Simplification #2: Drop lower order terms
- Simplification #3: Ignore leading constants
- Concentrate on growth

Formally, f(n) = O(g(n)) means that there exists constants c, n0 such that 0 <= f(n) <= c·g(n) for all n>=n0.

We read it as f(n) is big-O of g(n). g(n) is an asymptotic upper bound on f(n). We generally try to find the tightest upper bound.



(from Cormen, Leiserson, Riest, and Stein, Introduction to Algorithms)



For your best case, average case, worst case T(n) functions from above give the asymptotic function (Theta notation)

Given the problem sizes and worst-case runtime for one of the problem sizes, and what you know about each algorithm, predict the missing runtimes.

	n=100	n=200	n=400	n=800
Linear search	10 seconds			
Binary search		8 seconds		
Insertion Sort			320 seconds	

Algorithm Analysis Examples

Given each coded function construct a run time analysis function T(n) by assigning a different constant (c1, c2, c3, etc.) to each type of statement (the run time for statements of that type), and counting how many times each statement executes for an input size n in the worst case. Then sum up the constants times the execution counts. You may need to define variables other than n if there are event-controlled iterations with an unknown number of loops. Then use asymptotic analysis to simplify your run time analysis function T(n) and choose the matching answer.

1.

```
def f1(lst):      # assume n=len(lst)
    r = 0
    m = 100
    if len(lst) < m:
        m = len(lst)
    for x in range(m):
        r += x
```

A) O(1) constant
B) O(lg n) logarithmic
C) O(n) linear
D) O(n^2) quadratic
E) O(2^n) exponential

2.

```
def f2(n):        # find the square root of a number
    r = n / 2
    d = 1e-10
    while abs(n - r**2) > d:
        r = (r + n/r) / 2
    return r
```

A) O(1) constant
B) O(lg n) logarithmic
C) O(n) linear
D) O(n^2) quadratic
E) O(2^n) exponential

3.

```
def f3(lst):      # assume n=len(lst)
    while True:
        swapped = False
        for i in range(len(lst)-1):
            if lst[i] > lst[i+1]:
                lst[i], lst[i+1] = lst[i+1], lst[i]
                swapped = True
        if not swapped:
            break
```

A) O(1) constant
B) O(lg n) logarithmic
C) O(n) linear
D) O(n^2) quadratic
E) O(2^n) exponential

4.

```
def f4(n):
    counters = [0] * n
    while not all(counters):
        for i in range(n):
            if counters[i]:
                counters[i] = 0
            else:
                counters[i] = 1
                break
```

A) O(1) constant
B) O(lg n) logarithmic
C) O(n) linear
D) O(n^2) quadratic
E) O(2^n) exponential

5.

```
def f5(lst):
    n = len(lst)
    for i in range(n*100):
        a = random.randrange(n)
        b = random.randrange(n)
        lst[a], lst[b] = lst[b], lst[a]
```

A) O(1) constant
B) O(lg n) logarithmic
C) O(n) linear
D) O(n^2) quadratic
E) O(2^n) exponential

6.

```
int fun(int n) {
  int count = 0;
  for (int i = 0; i < n; i++)
     for (int j = i; j > 0; j--)
        count = count + 1;
  return count;
}
```

A) O(n)
B) O(n^2)
C) O(n*Logn)
D) O(nLognLogn)

7.

```
void function(int n) {
  int i, j, count=0;
  for (i=n/2; i <= n; i++)
    for (j = 1; j <= n; j = j*2)
      count++;
}
```

A) O(log n)
B) O(n^2)
C) O(n^2 log n)
D) O(n log n)

8.

```
int fun(int n){
  int count = 0;
  for (int i = n; i > 0; i /= 2)
     for (int j = 0; j < i; j++)
        count += 1;
  return count;
}
```

A) O(n^2)

B) O(nLogn)

C) O(n)

D) O(nLognLogn)