**Tuesday, September 4, 2018**

**Efficiency analysis:**

* Resources
* Worst-case
* Big-O
* 8 efficiency classes
* Experimental method

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  |  |
|  | Algorithm  P.code Ink Drawings Analyze |  |  |
|  |  | Efficiency class  Ex: O(n) |
|  |  |  |

Resource: countable thing consumed by running algorithms:

* Time steps
* Space used
* Energy used

Why time?

* Precious to humans (don’t want to keep people waiting
* Not renewable
* Historical
* Simplest, easiest to learn

Usually focus on worst-case (w.c) performance; not Average-case and Best-case

Pessimistic; design for challenging situations.

Complexity function:

f(n) = # of resources consumed by an input of size n,

(usually in worst-case)

Ex:

f(n) = 3n + 6

n: "Size measure"

f(x) = 7x + 2

Multiple size measures graph G = (V, G)

V = { vertices}

G = { edges }

n = # of vertices

m = # of edges

Definition of O:

For any complexity function f(n),

O(f(n)) = { g(n) | There exists c > 0, t ≥ 0, such that g(n) ≤ c.f(n), for all n ≥ t }

g(n) ≤ c. f(n) must be true when n ≥ t, and g(n) ≤ c. f(n) doesn’t matter when n < t

t = size of inputs that are "sufficiently large" for patterns to work.

O = " On the order of "

O(f(n)) = { Complexity functions equivalent to f(n) for purpose of analysis algorithms}

Ex:

f1(n) = 3n +1

f2(n) = 3n +2

f1 # f2

Let f(n) = n

O(n) = { complexity function's equivalent. To f(n) = n}

O(1), O(n), O(n^2)

Ex: O(n2) filling a matrix, double for loop.

Observation 1:

Some instances take fewer resources than the worst-case. That’s why g(n) <= c. f(n) not just " = ".

Observation 2:

Some instances do not follow trends that's why inequality only needs to be true when n≥ t

Ex:

Print\_array (v)

If V is empty 1 1

Return 1

Else

Print "Arrays" 1

For x in V 1

Print x 1

Print "end of array" 1

f(n) = # of steps

|  |  |  |
| --- | --- | --- |
| f(n) = | 2 | When n=0 |
| f(n) = | 2n+3 | When n > 0 |

t = "threshold" for min n that follows worst-case pattern when n < t, doesn’t matter.

Observation 3:

Resources consumption involves a constant factor that depends on an algorithm's implementation.

Implementation: working code on a specific computer.

Total algorithm = # **algorithm steps executed** x CPU instruction/step x seconds/CPU instruction

**(focus in this class)** (Programming C++ or Python) (Computer Hardware)

Let C = multiplicative run-time factor

from computer environment

Algorithm runtime = f(n) x c

3n is on the order of n

3n ϵ O(n) 3n ≤ c. n Ɐn>= 0, when c = 3

2n + 4 ϵ O(n)

Top 8 classes:

Intractable

Tractable

|  |  |  |
| --- | --- | --- |
| Notation | Name | Examples |
| O(1)  O(log(n))  O(nlog(n))  O(n^2)  O(n^3)      O(c ^n), c>1  O(2^n)  O(n!) | Constant  Logarithmic  "n-log-n"  "quadratic"  "cubic"      Exponential    factorial | Get 1 array element  Binary search tree  Merge sort  Two nested for loops  3 nested for loops      Subsets    permutations |

**Tuesday, September 6, 2018**

**Efficiency Analysis Continue**

Experimental / scientific

or

Math/ proofs

1. **Experimental**
2. **Scientific Method**

* Hypothesis
* Experiment, test hypothesis
* Evaluate results; support or reject hypothesis.

Hypothesis: algorithm x has time eff. O(f(n))

Experiment: implement alg.; run on several inputs of different sizes; measure elapsed time

Evaluation: draw scatter plot with best-fit line, decide whether curve matches efficiency class.

Ex:

Sum (V):

Total = 0

for x in V

Total +=x

return total

So:

Hypothesis: sum alg. Takes O(n) time.

Experiments: implement in C++, write main() function that calls sum() function for n = 1000, 2000,….1000000;

Measure each elapsed time.

|  |  |
| --- | --- |
| N | Time (ms) |
| 1,000  2,000  3,000  100,000 | 2  4.1  2.9  198 |

CONS:

* Labor- intensive, time consuming.
* Potential for skewed conclusion from data.
* Linked to implementation environment.

1. **Math analysis:**
2. Pick a "computational model"; math definition for alg., one unit of time, "standard Model"
3. Count (combinatorics) how many time steps the alg. Uses in the worst-case; derives a time complexity func. g(n)

Ex:

g(n) = 2n^2 + 7n -3

1. Prove our g(n) is an element of an eff. Class

Ex:

Lemma: 2n^2 + 7n- 3 ϵ O(n^2)

Proof: ….

CONS:

* Computational model needs to simulate real-word run-times well

1. **Standard model:**

Based on a realistic CPU w/memory, instructions

CPU RAM

* Complete instruction set

One instruction is limited to :

* Read O(1) bits from memory.
* Compute O(1) output bits, write them to memory.
* Each output bit could be computed by a Boolean circuit.

Random access: any byte of memory can be read/written in O(1) time.

1 step of time = anything that could be achieved w/ O(1) CPU instructions

Chronological step count:

* Visualize the alg. Running on a worst-case input
* Keep tally of how many steps it takes

g(n) = Ʃ all the steps

* Rules for P.code (pseudocode) syntax (if, for, return, etc..)
  + Two kinds of alg. Where doesn’t work:
    1. Recursive algorithm. (Master method instead)
    2. Can over-count with amortized-style data structures.

**Notes:**

* 1. Focusing on worst case complexity has an important benefit because:
     + By definition values are less than or equal to the maximum complexity value, so statements about worst-case complexities are both strong and conservative.
     + It simplifies mathematical analyses.