eecs281 DATA STRUCTURES AND ALGORITHMS

Lecture 2: Algorithm Analysis
(Review of Some 203 Material)

The Importance of Runtime Cost

Amazon claims that just an extra one tenth of a second on their response times will cost them 1% in sales [http://highscalability.com/latency-everywhere-and-it-costs-you-sales-how-crush-it]

Google said they noticed that just a half a second increase in latency caused traffic to drop by a fifth [http://glinden.blogspot.com/2006/11/marissa-mayer-at-web-20.html]

Measuring Runtime Cost

How do you measure runtime cost?

Assume can search 10 names/ms

Population (size)	Linear	Binary
EECS281 (230)	23 ms	0.8 ms
UM (40 000)	4 secs	1.5 ms
MI (9 mil)	15 mins	2.3 ms
Shanghai (23 mil)	38 mins	2.4 ms
US (311 mil)	8 hours 38 mins	2.8 ms
China (1.3 bil)	1 day 13 hours	3.0 ms
World (6.9 bil)	8 days	3.3 ms

An 8-year old may take longer!

How to Measure Runtime Cost?

Using wall-clock time to measure runtime cost can be tricky:

- depends on cpu speed
- depends on cpu load(!)
- (but we'll come back to this later)

Note: runtime cost == runtime complexity
== runtime == complexity
== time complexity

How to Measure Runtime Cost?

Instead we use operation count to measure runtime cost:

- pick one operation that is performed most often (e.g., ADD, SUB, MUL, DIV, CMP, LOAD, or STORE)
- count the number of times that operation is performed

Roofing Problem's Runtime Cost

A builder can carry two tiles on his shoulder as he climbs up the ladder
He then climbs down and carries two more tiles up the ladder
Each round trip (up and down the ladder) costs $2

- What is the cost to build a shed (8 tiles)?
  2*ceil(8/2)=8
- What is the cost to build a gazebo (128 tiles)?
  2*ceil(128/2)=128
- What is the cost to build a house (2048 tiles)?
  2*ceil(2048/2)=2048

What is the ONE operation we counted in this case?

Operation Count: `findMax()`

Write a function that given an array of N elements finds the index of the largest element

Example:

- `int findMax(int *a, int N)`
- given `int a=[3 8 2 7 9 1 5]`
- `findMax(a, 7) = 4`

Operation Count: `findMax()`

What is the fixed cost of the algorithm?

Which operation should we count to compute the variable cost?

How many times is this operation executed in the worst case?

A More Complicated Function

```
void
f(int N, int *p)
// p an array N integers
{
 int i, j, k, l;
 i = N*2; j = i+1;
 k = i-3*400*j; l = k/2;
 if (l < 0) { i = N+j; } else { i = log2(k); }
 for (i = 0; i < N; i++) { j = p[i]*3; }
 for (i = 0; i < N; i++) {
  for (k = i; k < N; k++) { j = p[j]*3; }
 }
 return;
}
```

How do we go about computing the time complexity of this function?

Four Accounting Rules

Rule 1: Consecutive Statements: $S_1; S_2; S_3; \ldots ; S_N;$

The runtime R of a sequence of statements is the runtime of the statement with the max runtime: $\max(R(S_1), R(S_2), \ldots, R(S_N))$

```
i = N*2; j = i+1;
k = i-3*400*j; l = k/2;
```

Rule 2: Conditional Execution: if (S_1) S_2; else S_3;

The runtime of a conditional execution is $\max(R(S_1), R(S_2), R(S_3))$

```
if (l < 0) { i = N+j; }
else { i = log2(k); }
```

Four Accounting Rules

Rule 3: Iteration/Loop: for $(S_1; S_2; S_3)$ S_4;

The runtime of a loop with N iterations is $\max(R(S_1), N{*}R(S_2), N{*}R(S_3), N{*}R(S_4))$

```
for (i = 0; i < N; i++) { j = p[i]*3; }
```

Rule 4: Nested Loop:

for $(S_1; S_2; S_3)$ { for $(S_4; S_5; S_6)$ { $S_7;$ } }

Count inside out: the runtime is the max of the runtime of each statement multiplied by the total number of times each statement is executed

```
for (i = 0; i < N; i++) {
  for (k = i; k < N; k++) {
   j = p[j]*3;
  }
}
```

A More Complicated Function

```
void
f(int N, int *p)
// p an array N integers
{
 int i, j, k, l;
 i = N*2; j = i+1;
 k = i-3*400*j; l = k/2;
 if (l < 0) { i = N+j; } else { i = log2(k); }
 for (i = 0; i < N; i++) { j = p[i]*3; }
 for (i = 0; i < N; i++) {
  for (k = i; k < N; k++) { j = p[j]*3; }
 }
 return;
}
```

What is the time complexity of this function?

Aside: Arithmetic Series

What is the sum of all integers from:

- 0 to 4 inclusive?
- 1 to 25 inclusive?
- 3 to 12 inclusive?

Operation Count: `computeRank()`

Write a function that given an unsorted array a of N elements computes an array r, where $r[i]$ is the rank of $a[i]$ in a

The rank of an element is: the number of elements in the array that is smaller than it, plus the number of elements to its left that is equal to it

Example:

- `void computeRank(int *a, int *r, int N)`
- given `int a=[4 3 9 3 7]`
- results in `r=[2 0 4 1 3]`

What Affects Runtime?

The algorithm

Implementation details
- skills of the programmer

Compiler (and options used)
- `g++ -g` (for debugging)
- `g++ -O3` (for speed)

CPU/memory speed
- and workload (other programs running concurrently)

Amount of data processed (N = input size)

Runtime Scalability

If N (problem size) is small, longish runtime is tolerable, for large N ($N > 10^4$, or $N \rightarrow \infty$ in general), only algorithms with low time complexity are usable

Scalability: an algorithm/data structure is not scalable if its complexity grows so fast that it requires more resources than available for the expected input size

More generally: an algorithm/data structure is not scalable if its complexity grows faster than the rate of increase in input size

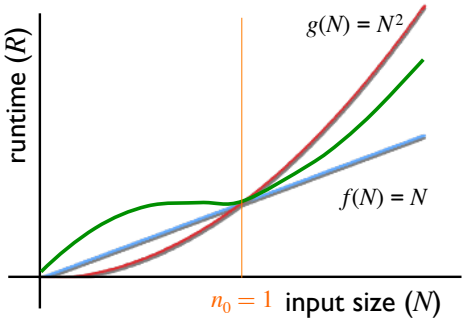
Runtime vs. Input Size

Rate of growth of runtime depends mostly on input size, independent of most other factors

- CPU speed, compiler, etc.

In algorithm analysis, we're usually only interest in large N, larger than some n_0

To ease discussion, we usually compare the complexity of our algorithm against some well-known "landmark" functions



Landmark Functions

constant: 1

logarithmic: $\log N$

square root: $\sqrt{N}$

linear: N

loglinear: $N \log N$

quadratic: N^2

cubic: N^3

in general, polynomial: $N^k, k \geq 1$

exponential: $a^N, a > 1,$
usually $a = 2$

factorial: $N!$