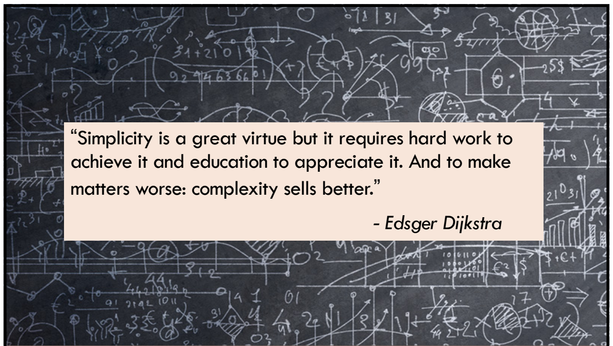


"Simplicity is a great virtue but it requires hard work to achieve it and education to appreciate it. And to make matters worse: complexity sells better."

- Edsger Dijkstra

ASYMPTOTIC COMPLEXITY

Lecture 10
CS2110 – Spring 2019

Announcements

2

- Next Mon-Tues: Spring Break
- No recitation next week
- Regrade requests will be processed this weekend
- Prelim is on Tuesday, 12 March. Prelim
  Review for prelim Sunday, 10 March, 1-3PM
  Next Thursday, we will tell you
  - What time you will be assigned to take it
  - What to do if you can't take it then but can take the other one
  - What to do if you can't take it that evening.
  - What to do if authorized for more time or quiet space

Help in providing code coverage

3

White-box testing: make sure each part of program is "exercised" in at least one test case. Called code coverage.

Eclipse has a tool for showing you how good your code coverage is! Use it on A3 (and any programs you write)

JavaHyperText entry:

code coverage

We demo it.

What Makes a Good Algorithm?

4

Suppose you have two possible algorithms that do the same thing; which is *better*?

What do we mean by *better*?
- Faster?
- Less space?
- Simpler?
- Easier to code?
- Easier to maintain?
- Required for homework?

FIRST, Aim for simplicity, ease of understanding, correctness.

SECOND, Worry about efficiency only when it is needed.

How do we measure speed of an algorithm?

Basic Step: one "constant time" operation

5

Constant time operation: its time doesn't depend on the size or length of anything. Always roughly the same. Time is bounded above by some number

Basic step:
- Input/output of a number
- Access value of primitive-type variable, array element, or object field
- assign to variable, array element, or object field ***
- do one arithmetic or logical operation
- method call (not counting arg evaluation and execution of method body)

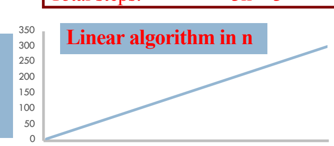
Counting Steps

6

```
// Store sum of 1..n in sum
sum= 0;
// inv: sum = sum of 1..(k-1)
for (int k= 1; k <= n; k= k+1){
    sum= sum + k;
}
```

Statement:	# times done
sum= 0;	1
k= 1;	1
k <= n	n+1
k= k+1;	n
sum= sum + k;	n
Total steps:	3n + 3

All basic steps take time 1.
There are n loop iterations.
Therefore, takes time proportional to n.



Not all operations are basic steps

7

// Store n copies of 'c' in s
s= "";
// inv: s contains k-1 copies of 'c'
for (int k= 1; k <= n; k= k+1){
  s= s + 'c';
}

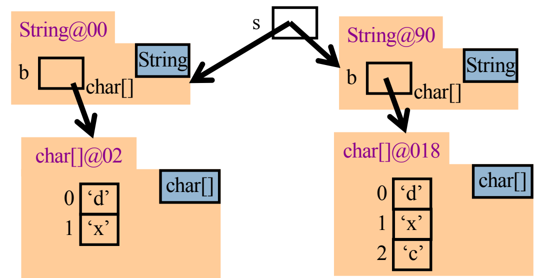
Statement:	# times done
s= "";	1
k= 1;	1
k <= n	n+1
k= k+1;	n
s= s + 'c';	n
Total steps:	3n + 3

Catenation is not a basic step. For each k, catenation creates and fills k array elements.

String Catenation

8

s= s + "c"; is NOT constant time.
It takes time proportional to 1 + length of s



Basic steps executed in s= s + 'c';

9

s= s + 'c'; // Suppose length of s is k

1. Create new String object, say C basic steps.
2. Copy k chars from object s to the new object: k basic steps
3. Place char 'c' into the new object: 1 basic step.
4. Store pointer to new object into s: 1 basic step.

Total of (C+2) + k basic steps.

In the algorithm, s= s + 'c'; is executed n times:
s= s + 'c'; with length of s = 0
s= s + 'c'; with length of s = 1
…
s= s + 'c'; with length of s = n-1
Total of n*(C+2) + (0 + 1 + 2 + … n-1) basic steps

Basic steps executed in s= s + 'c';

10

s= s + 'c'; // Suppose length of s is k

In the algorithm, s= s + 'c'; is executed as follows:
s= s + 'c'; with length of s = 0
s= s + 'c'; with length of s = 1
…
s= s + 'c'; with length of s = n-1
Total of n*(C+2) + (0 + 1 + 2 + … n-1) basic steps

0 + 1 + 2 + … n-1 = n(n-1) / 2. Gauss figured this out in the 1700's
= $n^2/2 – n/2$.
mathcentral.uregina.ca/qq/database/qq.02.06/jo1.html

Basic steps executed in s= s + 'c';

11

s= s + 'c'; // Suppose length of s is k

In the algorithm, s= s + 'c'; is executed as follows:
s= s + 'c'; with length of s = 0
s= s + 'c'; with length of s = 1
…
s= s + 'c'; with length of s = n-1
Total of n*(C+2) + (0 + 1 + 2 + … n-1) basic steps

Total of n*(C+2) + $n^2/2 – n/2$ basic steps

Total of n*(C+2) + $\mathbf{n^2}/2 – n/2$ basic steps. Quadratic in n.

Not all operations are basic steps

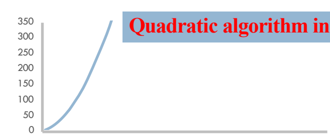
12

// Store n copies of 'c' in s
s= "";
// inv: s contains k-1 copies of 'c'
for (int k= 1; k <= n; k= k+1){
  s= s + 'c';
}

Statement:	# times	# steps
s= "";	1	1
k= 1;	1	1
k <= n	n+1	1
k= k+1;	n	1
s= s + 'c';	see to left	
Total steps:	…	

Total steps:
2n + 3 +
n*(C+2) + $\mathbf{n^2}/2 – n/2$
for s= s + 'c';

Quadratic algorithm in n



Linear versus quadractic

13

```
// Store sum of 1..n in sum
sum= 0;
// inv: sum = sum of 1..(k-1)
for (int k= 1; k <= n; k= k+1)
    sum= sum + n
```

Linear algorithm

```
// Store n copies of 'c' in s
s= "";
// inv: s contains k-1 copies of 'c'
for (int k= 1; k = n; k= k+1)
    s= s + 'c';
```

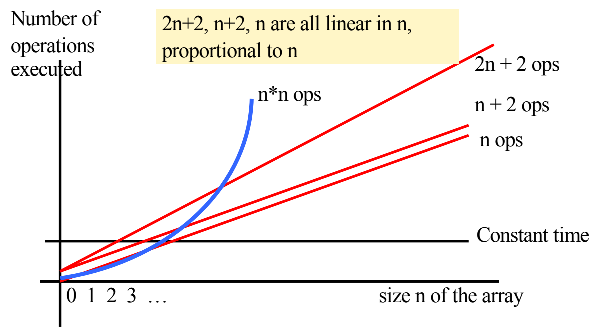
Quadratic algorithm

In comparing the runtimes of these algorithms, the exact number of basic steps is not important. What's important is that
One is linear in n —takes time proportional to n
One is quadratic in n —takes time proportional to n^2

Looking at execution speed

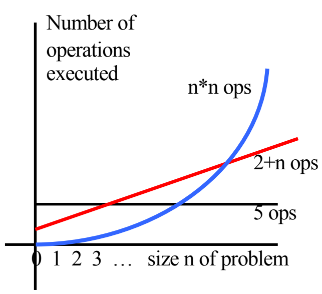
14

2n+2, n+2, n are all linear in n, proportional to n



What do we want from a definition of "runtime complexity"?

15

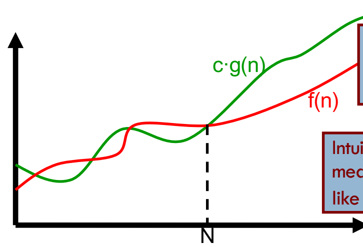


1. Distinguish among cases for large n, not small n
2. Distinguish among important cases, like
   - n*n basic operations
   - n basic operations
   - log n basic operations
   - 5 basic operations
3. Don't distinguish among trivially different cases.
   - 5 or 50 operations
   - n, n+2, or 4n operations

"Big O" Notation

16

Formal definition: f(n) is O(g(n)) if there exist constants c > 0 and N ≥ 0 such that for all n ≥ N, f(n) ≤ c·g(n)



Get out far enough (for n ≥ N) f(n) is at most c·g(n)

Intuitively, f(n) is O(g(n)) means that f(n) grows like g(n) or slower

Prove that $(2n^2 + n)$ is $O(n^2)$

17

Formal definition: f(n) is O(g(n)) if there exist constants c > 0 and N ≥ 0 such that for all n ≥ N, f(n) ≤ c·g(n)

Example: Prove that $(2n^2 + n)$ is $O(n^2)$

Methodology:

Start with f(n) and slowly transform into c · g(n):

- Use = and <= and < steps
- At appropriate point, can choose N to help calculation
- At appropriate point, can choose c to help calculation

Prove that $(2n^2 + n)$ is $O(n^2)$

18

Formal definition: f(n) is O(g(n)) if there exist constants c > 0 and N ≥ 0 such that for all n ≥ N, f(n) ≤ c·g(n)

Example: Prove that $(2n^2 + n)$ is $O(n^2)$

f(n)
= <definition of f(n)>
$2n^2 + n$
<= <for $n \geq 1$, $n \leq n^2$>
$2n^2 + n^2$
= <arith>
$3*n^2$
= <definition of $g(n) = n^2$>
3*g(n)

Transform f(n) into c·g(n):
- Use =, <= , < steps
- Choose N to help calc.
- Choose c to help calc

Choose
N = 1 and c = 3

Prove that 100 n + log n is O(n)

19

Formal definition: f(n) is O(g(n)) if there exist constants $c > 0$ and $N \geq 0$ such that for all $n \geq N$, $f(n) \leq c \cdot g(n)$

f(n)

= <put in what f(n) is>

100 n + log n

<= <We know $\log n \leq n$ for $n \geq 1$>

100 n + n

= <arith>

101 n

= <g(n) = n>

101 g(n)

Choose
N = 1 and c = 101

O(…) Examples

20

Let $f(n) = 3n^2 + 6n - 7$
- f(n) is $O(n^2)$
- f(n) is $O(n^3)$
- f(n) is $O(n^4)$
- …

$p(n) = 4 n \log n + 34 n - 89$
- p(n) is O(n log n)
- p(n) is $O(n^2)$

$h(n) = 20 \cdot 2^n + 40n$

h(n) is $O(2^n)$

$a(n) = 34$
- a(n) is O(1)

Only the *leading* term (the term that grows most rapidly) matters

If it's $O(n^2)$, it's also $O(n^3)$ etc! However, we always use the smallest one

Do NOT say or write f(n) = O(g(n))

21

Formal definition: f(n) is O(g(n)) if there exist constants $c > 0$ and $N \geq 0$ such that for all $n \geq N$, $f(n) \leq c \cdot g(n)$

f(n) = O(g(n)) is simply WRONG. Mathematically, it is a disaster. You see it sometimes, even in textbooks. Don't read such things.

Here's an example to show what happens when we use = this way.

We know that n+2 is O(n) and n+3 is O(n). Suppose we use =

n+2 = O(n)
n+3 = O(n)

But then, by transitivity of equality, we have n+2 = n+3.
We have proved something that is false. Not good.

Problem-size examples

22

- Suppose a computer can execute 1000 operations per second; how large a problem can we solve?

operations	1 second	1 minute	1 hour
n	1000	60,000	3,600,000
n log n	140	4893	200,000
n^2	31	244	1897
$3n^2$	18	144	1096
n^3	10	39	153
2^n	9	15	21

Commonly Seen Time Bounds

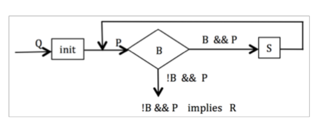
23

O(1)	constant	excellent
O(log n)	logarithmic	excellent
O(n)	linear	good
O(n log n)	n log n	pretty good
$O(n^2)$	quadratic	maybe OK
$O(n^3)$	cubic	maybe OK
$O(2^n)$	exponential	too slow

Search for v in b[0..]

24

Q: v is in array b
Store in i the index of the first occurrence of v in b:
R: v is not in b[0..i-1] and b[i] = v.



Methodology:
1. Define pre and post conditions.
2. Draw the invariant as a combination of pre and post.
3. Develop loop using 4 loopy questions.

Practice doing this!

Search for v in b[0..]

25

Q: v is in array b
Store in i the index of the first occurrence of v in b:
R: v is not in b[0..i-1] and b[i] = v.

pre: b [0 .. b.length): v in here

post: b [0 .. i): $\neq v$ | [i]: v | ? .. b.length

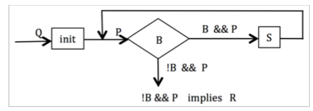
inv: b [0 .. i): $\neq v$ | [i .. b.length): v in here

Methodology:
1. Define pre and post conditions.
2. Draw the invariant as a combination of pre and post.
3. Develop loop using 4 loopy questions.

Practice doing this!

The Four Loopy Questions

26



- Does it start right?
  Is {Q} init {P} true?
- Does it continue right?
  Is {P && B} S {P} true?
- Does it end right?
  Is P && !B => R true?
- Will it get to the end?
  Does it make progress toward termination?

Search for v in b[0..]

27

Q: v is in array b
Store in i the index of the first occurrence of v in b:
R: v is not in b[0..i-1] and b[i] = v.

pre: b [0 .. b.length): v in here

post: b [0 .. i): $\neq v$ | [i]: v | ? .. b.length

inv: b [0 .. i): $\neq v$ | [i .. b.length): v in here

```
i= 0;
while (b[i] != v) {
    i= i+1;
}
```

Each iteration takes constant time.
Worst case: b.length iterations

Linear algorithm: O(b.length)

Binary search for v in sorted b[0..]

28

// b is sorted. Store in i a value to truthify R:
// b[0..i] <= v < b[i+1..]

pre: b [0 .. b.length): sorted

post: b [0 .. i]: $\leq v$ | [i+1 .. b.length): $> v$

inv: b [0 .. i]: $\leq v$ | ? | [k .. b.length): $> v$

b is sorted. We know that. To avoid clutter, don’t write in it invariant

Methodology:
1. Define pre and post conditions.
2. Draw the invariant as a combination of pre and post.
3. Develop loop using 4 loopy questions.

Practice doing this!

Binary search for v in sorted b[0..]

29

// b is sorted. Store in i a value to truthify R:
// b[0..i] <= v < b[i+1..]

pre: b [0 .. b.length): sorted

post: b [0 .. i]: $\leq v$ | [i+1 .. b.length): $> v$

inv: b [0 .. i]: $\leq v$ | ? | [k .. b.length): $> v$

[0 .. i]: $\leq v$ | [i+1 .. e]: $\leq v$ | | [k ..): $> v$

```
i= -1;
k= b.length;
while (i+1< k) {
   int e=(i+k)/2;
   // -1 ≤ i < e < k ≤ b.length
   if (b[e] <= v) i= e;
   else k= e;
}
```

Binary search for v in sorted b[0..]

30

// b is sorted. Store in i a value to truthify R:
// b[0..i] <= v < b[i+1..]

pre: b [0 .. b.length): sorted

post: b [0 .. i]: $\leq v$ | [i+1 .. b.length): $> v$

inv: b [0 .. i]: $\leq v$ | ? | [k .. b.length): $> v$

```
i= -1;
k= b.length;
while (i+1< k) {
   int e=(i+k)/2;
   // -1 ≤ i < e < k ≤ b.length
   if (b[e] <= v) i= e;
   else k= e;
}
```

Each iteration takes constant time.

Logarithmic: O(log(b.length))

Worst case: log(b.length) iterations

Binary search for v in sorted b[0..]

31

// b is sorted. Store in i a value to truthify R:
// b[0..i] <= v < b[i+1..]

This algorithm is better than binary searches that stop when v is found.

1. Gives good info when v not in b.
2. Works when b is empty.
3. Finds first occurrence of v, not arbitrary one.
4. Correctness, including making progress, easily seen using invariant

```
i= -1;
k= b.length;
while ( i+1< k) {
  int e=(i+k)/2;
  // -1 ≤ e < k ≤ b.length
  if (b[e] <= v) i= e;
  else k= e;
}
```

Each iteration takes constant time.

Logarithmic: O(log(b.length))

Worst case: log(b.length) iterations

Dutch National Flag Algorithm

32

Dutch National Flag Algorithm

Dutch national flag. Swap b[0..n-1] to put the reds first, then the whites, then the blues. That is, given precondition Q, swap values of b[0.n-1] to truthify postcondition R:

```
      0                              n
Q: b |               ?               |
      0                              n
R: b | reds  |  whites  |  blues     |
      0                              n
P1: b| reds | whites | blues |   ?   |
      0                              n
P2: b| reds | whites |   ?   | blues |
```

Suppose we use invariant P1.

What does the repetend do?

2 swaps to get a red in place

Dutch National Flag Algorithm

Dutch national flag. Swap b[0..n-1] to put the reds first, then the whites, then the blues. That is, given precondition Q, swap values of b[0.n-1] to truthify postcondition R:

```
      0                              n
Q: b |               ?               |
      0                              n
R: b | reds  |  whites  |  blues     |
      0                              n
P1: b| reds | whites | blues |   ?   |
      0                              n
P2: b| reds | whites |   ?   | blues |
```

Suppose we use invariant P2.

What does the repetend do?

At most one swap per iteration

Compare algorithms without writing code!

Dutch National Flag Algorithm: invariant P1

```
      0                              n
Q: b |               ?               |
      0                              n
R: b | reds  |  whites  |  blues     |
      0    h       k        p        n
P1: b| reds | whites | blues |   ?   |
```

```
h= 0; k= h; p= k;
while ( p != n ) {
   if (b[p] blue)  p= p+1;
   else if (b[p] white) {
         swap b[p], b[k];
         p= p+1; k= k+1;
   }
   else { // b[p] red
         swap b[p], b[h];
         swap b[p], b[k];
         p= p+1; h=h+1; k= k+1;
   }
}
```

Dutch National Flag Algorithm: invariant P2

```
      0                              n
Q: b |               ?               |
      0                              n
R: b | reds  |  whites  |  blues     |
      0    h       k        p        n
P2: b| reds | whites |   ?   | blues |
```

```
h= 0; k= h; p= n;
while ( k != p ) {
   if (b[k] white)  k= k+1;
   else if (b[k] blue) {
         p= p-1;
         swap b[k], b[p];
   }
   else { // b[k] is red
         swap b[k], b[h];
         h= h+1; k= k+1;
   }
}
```

36

Asymptotically, which algorithm is faster?

37

Invariant 1

0 h	k	p	n
reds | whites	blues	?	

```
h= 0; k= h; p= k;
while ( p != n ) {
    if (b[p] blue)        p= p+1;
    else if (b[p] white) {
            swap b[p], b[k];
            p= p+1; k= k+1;
    }
    else { // b[p] red
            swap b[p], b[h];
            swap b[p], b[k];
            p= p+1; h=h+1; k= k+1;
    }
}
```

Invariant 2

0 h	k	p	n
reds | whites	?	blues	

```
h= 0; k= h; p= n;
while ( k != p ) {

    if (b[k] white)         k= k+1;
    else if (b[k] blue) {
            p= p-1;
            swap b[k], b[p];
    }
    else { // b[k] is red
            swap b[k], b[h];
            h= h+1; k= k+1;
    }
}
```

Asymptotically, which algorithm is faster?

38

Invariant 1

0 h	k	p	n
reds | whites	blues	?	

might use 2 swaps per iteration

```
    if (b[p] blue)        p= p+1;
    else if (b[p] white) {
            swap b[p], b[k];
            swap b[p], b[h];
            swap b[p], b[k];
            p= p+1; h=h+1; k= k+1;
    }
}
```

Invariant 2

0 h	k	p	n
reds | whites	?	blues	

uses at most 1 swap per iteration

```
    if (b[k] white)         k= k+1;
    else if (b[k] blue) {
            p= p-1;
            swap b[k], b[h];
            h= h+1; k= k+1;
    }
}
```

These two algorithms have the same asymptotic running time (both are O(n))