

Iterative Algorithms: Dependent Nested Loops

In this lecture we continue the analysis of iterative algorithms.

- We consider nested loops with the control variable of the inner loop dependent on the control variable of the outer loop.
- The performance of the algorithm with dependent nested loop is illustrated by an example.

1. Dependent Nested Loops

The analysis of nested loops may be more complicated if the inner loop depends on the outer loop.

```
for i ← 1 to n do
  for j ← 1 to i do
    application
```

In the example above the inner loop is executed once the first iteration, twice the second iteration, three times the third iteration, and so forth. Thus the application code is performed

$$1 + 2 + 3 + \dots + n = \frac{1}{2} n(n+1)$$

times. It follows that if the application code consists of a constant number of basic operations, then the running time of the algorithm is $O(n^2)$.

2. Example: Two Algorithms to Find Prefix Averages

In this example we analyze two algorithms that compute the so-called *prefix averages* of a sequence of numbers. Given an array $X[0..n-1]$ of n numbers, we want to compute an array A such that $A[i]$ is the average of elements $X[0], \dots, X[i]$ for $i=0, \dots, n-1$, that is

$$A[i] = \frac{1}{i+1} (X[0] + X[1] + \dots + X[i]).$$

Our first algorithm *prefixAverages1* implements the formula above.

```

ALGORITHM prefixAverages1( $X[0..n-1]$ )
//Input: an  $n$ -element array  $X[0..n-1]$ 
//Output: an  $n$ -element array  $A[0..n-1]$  such that
// $A[i]$  is the average of elements  $X[0], \dots, X[i]$ 

for  $i \leftarrow 0$  to  $n-1$  do
     $sumX \leftarrow 0$ 
    for  $j \leftarrow 0$  to  $i$  do
         $sumX \leftarrow sumX + X[j]$ 
     $A[i] \leftarrow sumX / (i + 1)$ 
output  $A[0..n-1]$ 

```

By filling in the table below show how *prefixAverages1* operates on the list of $n=6$ numbers 5, 7, 9, 3, 16, 14

i	j	$sumX$	$X[0]$	$X[1]$	$X[2]$	$X[3]$	$X[4]$	$X[5]$	$A[0]$	$A[1]$	$A[2]$	$A[3]$	$A[4]$	$A[5]$
0		0	5	7	9	3	16	14						
0	0													
1		0												
1	0													
1	1													
2		0												
2	0													
2	1													
2	2													
3		0												
3	0													
3	1													
3	2													
3	3													
4		0												
4	0													
4	1													
4	2													
4	3													
4	4													
5		0												
5	0													
5	1													
5	2													
5	3													
5	4													
5	5								5	6	7	6	8	9

- What are the algorithm's basic operations?
- How many times are the basic operations executed?
- Determine the class $O(?)$ the algorithm belongs to.

In what follows we discuss another algorithm to solve the same problem.

Algorithm *prefixAverages2* uses the following property:

$$A[i-1] = (\underline{X[0] + X[1] + \dots + X[i-1]}) / i$$

$$A[i] = (\underline{X[0] + X[1] + \dots + X[i-1]} + X[i]) / (i+1)$$

The underlined parts are the same in the two expressions. It follows that in iteration *i* we can use the value *sumX* found in iteration *i-1*.

```

ALGORITHM prefixAverages2(X[0..n-1])
//Input: an n-element array X[0..n-1]
//Output: an n-element array A[0..n-1] such that
//A[i] is the average of elements X[0], ..., X[i]

sumX ← 0
for i ← 0 to n-1 do
    sumX ← sumX + X[i]
    A[i] ← sumX / (i + 1)
output A[0..n-1]
    
```

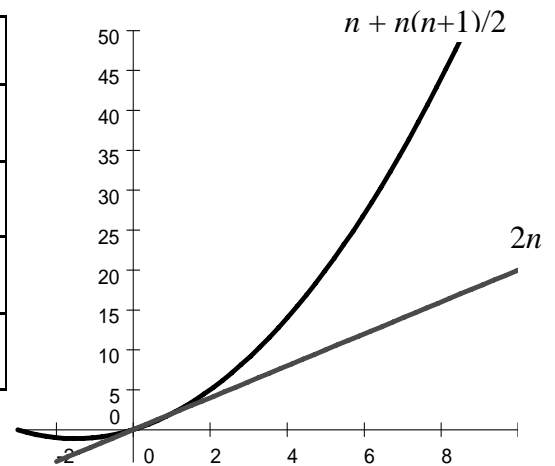
By filling in the table below show how *prefixAverages2* operates on the list of *n=6* numbers 5, 7, 9, 3, 16, 14

<i>i</i>	<i>sumX</i>	X[0]	X[1]	X[2]	X[3]	X[4]	X[5]	A[0]	A[1]	A[2]	A[3]	A[4]	A[5]
0	0	5	7	9	3	16	14						
0	5												
1													
2													
3													
4													
5								5	6	7	6	8	9

- (a) What are the algorithm's basic operations?
- (b) How many times are the basic operations executed?
- (c) Determine the class $O(?)$ the algorithm belongs to.

In the following table we compare the two algorithms.

	Algorithm <i>prefixAverages1</i>	Algorithm <i>prefixAverages2</i>
Summation	$1+2+\dots+n = \frac{1}{2}n(n+1)$	n
Division	N	n
Total	$n + n(n+1)/2$	$2n$
Complexity class	$O(n^2)$	$O(n)$



Clearly, the linear-time algorithm *prefixAverages2* outperforms quadratic-time algorithm *prefixAverages1* and as the size n of the input array increases, the difference in the performance of the two algorithms becomes really significant.

3. Logarithmic Loops

Consider two examples in which the controlling variable is multiplied or divided in each iteration.

```

i ← 1
while i ≤ n do
    application code
    i ← i * 2
    
```

```

i ← n
while i ≥ 1 do
    application code
    i ← i / 2
    
```

In order to estimate the number of iterations (say, k), we analyse the value of i for each iteration.

Iteration	Value of i (at the beginning of iteration)
1	1
2	2
3	4
4	8
5	16
6	32
⋮	⋮
k	2^{k-1}

Iteration	Value of i (at the beginning of iteration)
1	n
2	$n/2$
3	$n/4$
4	$n/8$
5	$n/16$
6	$n/32$
⋮	⋮
k	$n/2^{k-1}$

The condition for termination:
 $2^{k-1} \leq n$
 or equivalently
 the number of iterations $k-1 \leq \log_2 n$
 $k \leq 1 + \log_2 n$

The condition for termination:
 $n/2^{k-1} \geq 1$
 or equivalently
 the number of iterations $k-1 \leq \log_2 n$
 $k \leq 1 + \log_2 n$

For both codes, if the application code of the body of the loop requires constant time, then the running time of the algorithm is $O(\log_2 n)$.

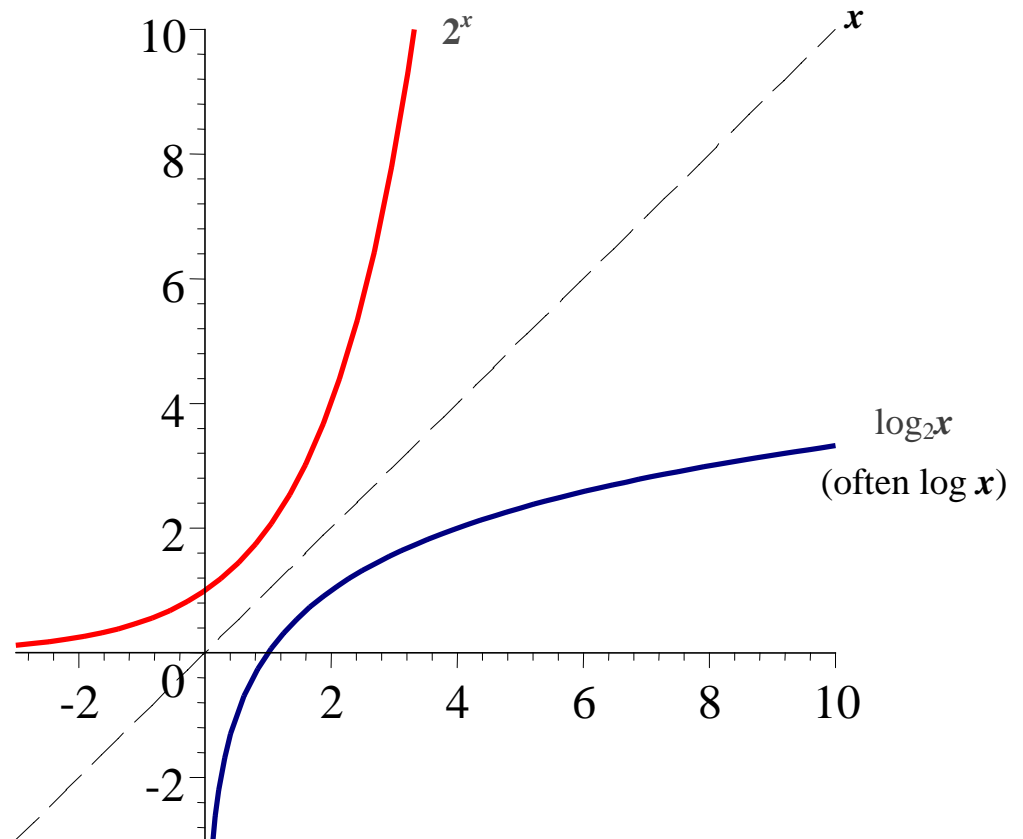
Exponent: $b^x = b \times b \times \dots \times b$

$\underbrace{\hspace{10em}}$
 (x times)

$$(b^a)^c = b^{ac}$$

$$b^a b^c = b^{a+c}$$

$$\frac{b^a}{b^c} = b^{a-c}$$



Logarithm:

Logarithm of a number x in base b is the power to which b must be raised in order to produce x .

$\log_b x$ is the unique real number y such that $b^y = x$

$$\log_b(xy) = \log_b x + \log_b y$$

$$\log_b x^y = y \log_b x$$