Algorithms and Data Structures

CS-CO-412

David Vernon
Professor of Informatics
University of Skövde
Sweden

david@vernon.eu
www.vernon.eu

Complexity of Algorithms

Lecture 2

Topic Overview

- **Analysis of complexity of algorithms**
  - **Time complexity**
  - **Big-O Notation**
  - **Space complexity**
- Introduction to complexity theory
  - P, NP, and NP-Complete classes of algorithm

Motivation

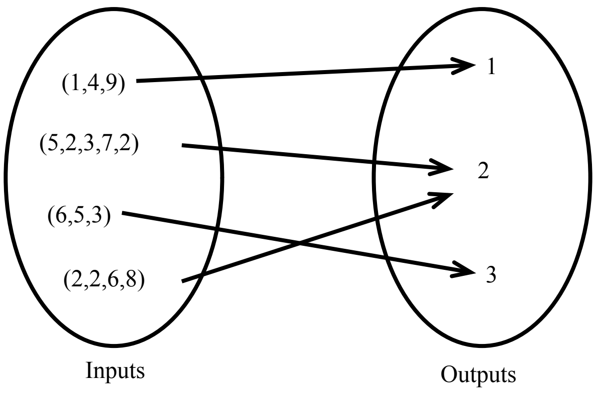
- Complexity Theory
  - Easy problems (sort a million items in a few seconds)
  - Hard problems (schedule a thousand classes in a hundred years)
  - What makes some problems hard and others easy (computationally) and how do we make hard problems easier?
  - Complexity Theory addresses these questions

Complexity Analysis

- Why do we write programs?
  - to perform some specific tasks
  - to solve some specific problems
  - We will focus on "solving problems"
  - What is a "problem"?
  - We can view a problem as a mapping of "inputs" to "outputs"

Complexity Analysis

For example, Find Minimum



Complexity Analysis

How to describe a problem?

- Input
  - Describe what an input looks like
- Output
  - Describe what an output looks like and how it relates to the input

Complexity Analysis

An instance is an assignment of values to the input variables

An instance of the Find Minimum function

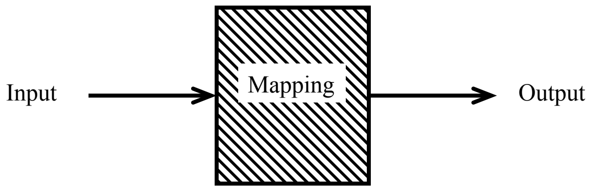
$N = 10$
$(a_1, a_2,\ldots, a_N) = (5,1,7,4,3,2,3,3,0,8)$

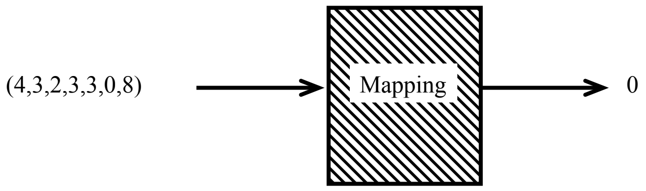
Another instance of the Find Minimum Problem

$N = 10$
$(a_1, a_2,\ldots, a_N) = (15,8,0,4,7,2,5,10,1,4)$

Complexity Analysis

A problem can be considered as a black box





Complexity Analysis

Example: Sorting

Input: A sequence of N numbers $a_1 \ldots a_n$

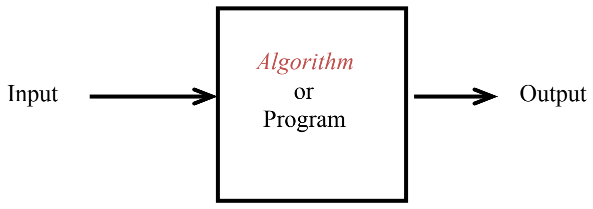
Output: the permutation (reordering) of the input sequence such that $a_1 \leq a_2 \leq \ldots \leq a_n$

Complexity Analysis

How do we solve a problem?

Write an algorithm that implements the mapping

Takes an *input* in and produces a correct *output*



Complexity Analysis

- How do we judge whether an algorithm is good or bad?
- Analyse its efficiency
  - Determined by the amount of computer resources consumed by the algorithm
- What are the important resources?
  - Amount of memory (space complexity)
  - Amount of computational time (time complexity)

Complexity Analysis

Consider the amount of resources

memory space and time

that an algorithm consumes

as a function of the size of the input to the algorithm.

Complexity Analysis

- Suppose there is an assignment statement in your program

  x := x +1

- We'd like to determine:
  - The time a single execution would take
  - The number of times it is executed: Frequency Count

Time Complexity

- Product of execution time and frequency is *approximately* the total time taken
- But, since the execution time will be very machine dependent (and compiler dependent), we neglect it and concentrate on the frequency count
- Frequency count will vary from data set to data set (*input to the algorithm*)

Time Complexity

Program 1

```
x := x + 1
```

Frequency = 1

Program 2

```
FOR i := 1 to n
DO
   x := x + 1
END
```

Frequency = n

Program 3

```
FOR i := 1 to n
DO
   FOR j := 1 to n
   DO
      x := x + 1
   END
END
```

Frequency = n^2

Time Complexity

- Program 1
  - statement is not contained in a loop (implicitly or explicitly)
  - Frequency count is 1
- Program 2
  - statement is executed *n* times
- Program 3
  - statement is executed n^2 times

Big-O Notation

- 1, *n*, and n^2 are said to be different and increasing orders of magnitude

  (e.g. let $n = 10 \Rightarrow 1, 10, 100$)

- We are interested in determining the order of magnitude of the time complexity of an algorithm

Big-O Notation

- Let's look at an algorithm to print the n^{th} term of the Fibonnaci sequence

0 1 1 2 3 5 8 13 21 34 …

$t_n = t_{n-1} + t_{n-2}$

$t_0 = 0$

$t_1 = 1$

Big-O Notation

```
1   procedure fibonacci {print nth term}
2      read(n)
3      if n<0
4         then print(error)
5         else if n=0
6            then print(0)
7            else if n=1
8               then print(1)
9               else
10                 fnm2 := 0;
11                 fnm1 := 1;
12                 FOR i := 2 to n DO
13                    fn := fnm1 + fnm2;
14                     fnm2 := fnm1;
15                     fnm1 := fn
16                  end
17                  print(fn);
```

step	n<0
1	1
2	1
3	1
4	1
5	0
6	0
7	0
8	0
9	0
10	0
11	0
12	0
13	0
14	0
15	0
16	0
17	0

Big-O Notation

```
1   procedure fibonacci {print nth term}
2      read(n)
3      if n<0
4         then print(error)
5         else if n=0
6            then print(0)
7            else if n=1
8               then print(1)
9               else
10                 fnm2 := 0;
11                 fnm1 := 1;
12                 FOR i := 2 to n DO
13                    fn := fnm1 + fnm2;
14                     fnm2 := fnm1;
15                     fnm1 := fn
16                  end
17                  print(fn);
```

step	n=0
1	1
2	1
3	1
4	0
5	1
6	1
7	0
8	0
9	0
10	0
11	0
12	0
13	0
14	0
15	0
16	0
17	0

Big-O Notation

```
1   procedure fibonacci {print nth term}
2      read(n)
3      if n<0
4         then print(error)
5         else if n=0
6            then print(0)
7            else if n=1
8               then print(1)
9               else
10                 fnm2 := 0;
11                 fnm1 := 1;
12                 FOR i := 2 to n DO
13                    fn := fnm1 + fnm2;
14                     fnm2 := fnm1;
15                     fnm1 := fn
16                  end
17                  print(fn);
```

step	n=1
1	1
2	1
3	1
4	0
5	1
6	0
7	1
8	1
9	0
10	0
11	0
12	0
13	0
14	0
15	0
16	0
17	0

Big-O Notation

```
1   procedure fibonacci {print nth term}
2      read(n)
3      if n<0
4         then print(error)
5         else if n=0
6            then print(0)
7            else if n=1
8               then print(1)
9               else
10                 fnm2 := 0;
11                 fnm1 := 1;
12                 FOR i := 2 to n DO
13                    fn := fnm1 + fnm2;
14                     fnm2 := fnm1;
15                     fnm1 := fn
16                  end
17                  print(fn);
```

step	n>1
1	1
2	1
3	1
4	0
5	1
6	0
7	1
8	0
9	1
10	1
11	1
12	n
13	n-1
14	n-1
15	n-1
16	n-1
17	1

Big-O Notation

step	n<0	n=0	n=1	n>1
1	1	1	1	1
2	1	1	1	1
3	1	1	1	1
4	1	0	0	0
5	0	1	1	1
6	0	1	0	0
7	0	0	1	1
8	0	0	1	0
9	0	0	0	1
10	0	0	0	1
11	0	0	0	1
12	0	0	0	n
13	0	0	0	n-1
14	0	0	0	n-1
15	0	0	0	n-1
16	0	0	0	n-1
17	0	0	0	1

Big-O Notation

- The cases where *n*<0, *n*=0, *n*=1 are not particularly instructive or interesting
- In the case where *n*>1, we have the total statement frequency of

  $9 + n + 4(n-1) = 5n + 5$

Big-O Notation

- $9 + n + 4(n-1) = 5n + 5$
- We write this as *O*(*n*), ignoring the constants
- *This is called Big-O notation*
- More formally, *f*(*n*) = *O*(g(*n*))
  where *g*(*n*) is an **asymptotic upper bound** for *f(n)*

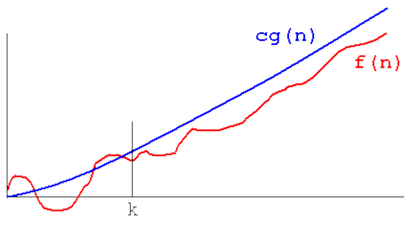
Big-O Notation

- The notation $f(n) = O(g(n))$ has a precise mathematical definition
- Read $f(n) = O(g(n))$ as
  f of *n* is big-O of *g* of *n*
- Definition:
  Let $f, g: Z^+ \rightarrow R^+$

  $f(n) = O(g(n))$ if there exist two constants *c* and *k* such that $f(n) \le c\, g(n)$ for all $n \ge k$

Big-O Notation

Suppose $f(n)= 2n^2+4n+10$
and $f(n) = O(g(n))$ where $g(n)=n^2$



<u>*Proof:*</u>
$f(n)= 2n^2+4n+10$
$f(n) \le 2n^2+ 4n^2+10n^2$ for $n \ge 1$
$f(n) \le 16n^2$
$f(n) \le 16g(n)$ Where c = 16 and $k=1$

Time & Space Complexity

- *f(n)* will normally represent the computing time of some algorithm

  Time complexity *T*(*n*)

- *f*(*n*) can also represent the amount of memory an algorithm will need to run

  Space complexity *S*(*n*)

Time Complexity

- If an algorithm has a time complexity of *O*(*g*(*n*)) it means that its execution will take no longer than a constant times *g*(*n*)
- More formally, g(n) is an **asymptotic upper bound** for *f(n)*

Remember

- $f(n) \leq c\ g(n)$

  n is typically the size of the data set

Time Complexity

O(1)	Constant (computing time)
O(*n*)	Linear (computing time)
O(n^2)	Quadratic (computing time)
O(n^3)	Cubic (computing time)
O(2^n)	Exponential (computing time)

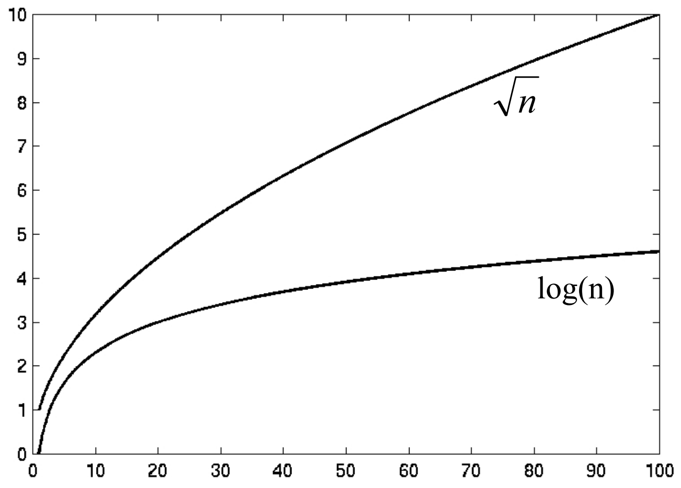
O(log *n*) is faster than *O*(*n*) for sufficiently large *n*

O(*n* log *n*) is faster than *O*(n^2) for sufficiently large *n*

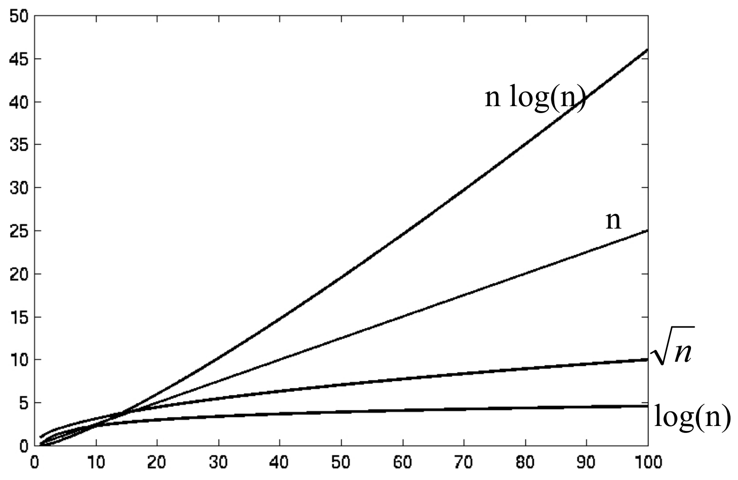
Time Complexity

n	O(1)	O(log2(n))	O(n)	O(nlog2(n))	O(n^2)	O(n^3)	O(n^4)	O(2^n)	O(n^n)
1	7	0.0	1	0.0	1	1	1	2	1
2	7	1.0	2	2.0	4	8	16	4	4
3	7	1.6	3	4.8	9	27	81	8	27
4	7	2.0	4	8.0	16	64	256	16	256
5	7	2.3	5	11.6	25	125	625	32	3125
6	7	2.6	6	15.5	36	216	1296	64	46656
7	7	2.8	7	19.7	49	343	2401	128	823543
8	7	3.0	8	24.0	64	512	4096	256	16777216
9	7	3.2	9	28.5	81	729	6561	512	3.87E+08
10	7	3.3	10	33.2	100	1000	10000	1024	1E+10
11	7	3.5	11	38.1	121	1331	14641	2048	2.85E+11
12	7	3.6	12	43.0	144	1728	20736	4096	8.92E+12
13	7	3.7	13	48.1	169	2197	28561	8192	3.03E+14
14	7	3.8	14	53.3	196	2744	38416	16384	1.11E+16
15	7	3.9	15	58.6	225	3375	50625	32768	4.38E+17
16	7	4.0	16	64.0	256	4096	65536	65536	1.84E+19
17	7	4.1	17	69.5	289	4913	83521	131072	8.27E+20
18	7	4.2	18	75.1	324	5832	104976	262144	3.93E+22
19	7	4.2	19	80.7	361	6859	130321	524288	1.98E+24
20	7	4.3	20	86.4	400	8000	160000	1048576	1.05E+26
21	7	4.4	21	92.2	441	9261	194481	2097152	5.84E+27
22	7	4.5	22	98.1	484	10648	234256	4194304	3.41E+29
23	7	4.5	23	104.0	529	12167	279841	8388608	2.09E+31
24	7	4.6	24	110.0	576	13824	331776	16777216	1.33E+33
25	7	4.6	25	116.1	625	15625	390625	33554432	8.88E+34
26	7	4.7	26	122.2	676	17576	456976	67108864	6.16E+36
27	7	4.8	27	128.4	729	19683	531441	1.34E+08	4.43E+38
28	7	4.8	28	134.6	784	21952	614656	2.68E+08	3.31E+40
29	7	4.9	29	140.9	841	24389	707281	5.37E+08	2.57E+42
30	7	4.9	30	147.2	900	27000	810000	1.07E+09	2.06E+44

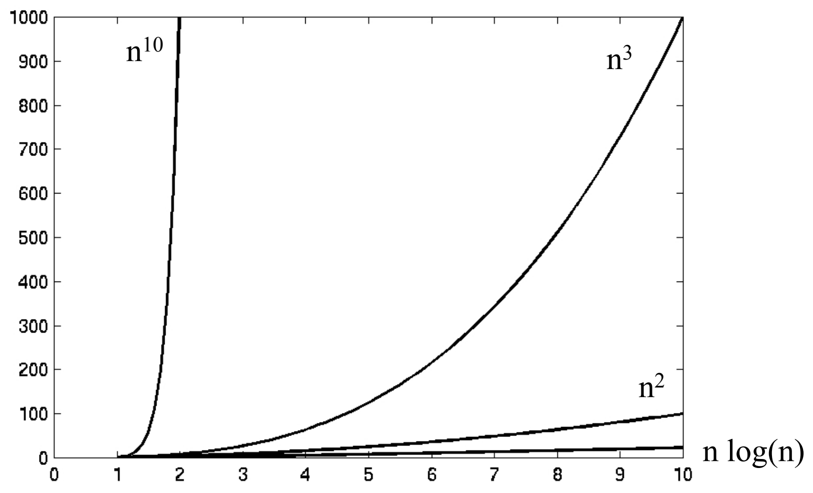
Time Complexity



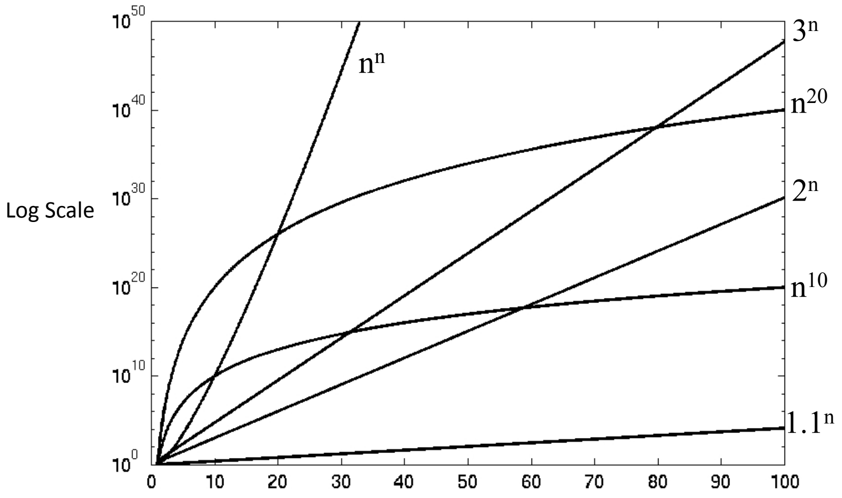
Time Complexity



Time Complexity



Time Complexity



Time Complexity

$f1(n) = 10\ n + 25\ n^2$	$O(n^2)$
$f2(n) = 20\ n \log n + 5\ n$	$O(n \log n)$
$f3(n) = 12\ n \log n + 0.05\ n^2$	$O(n^2)$
$f4(n) = n^{1/2} + 3\ n \log n$	$O(n \log n)$

Time Complexity

- Arithmetic of Big-O notation

  if

  $T_1(n) = O(f(n))$ and $T_2(n) = O(g(n))$

  then

  $T_1(n) + T_2(n) = O(max(f(n), g(n))$

Time Complexity

- Arithmetic of Big-O notation

  if

  $f(n) \leq g(n)$

  then

  $O(f(n) + g(n)) = O(g(n))$

Time Complexity

- Arithmetic of Big-O notation

  if

  $T_1(n) = O(f(n))$ and $T_2(n) = O(g(n))$

  then

  $T_1(n)\ T_2(n) = O(f(n)\ g(n))$

Time Complexity

- Rules for computing the time complexity
  - the complexity of each read, write, and assignment statement can be taken as $O(1)$
  - the complexity of a sequence of statements is determined by the summation rule
  - the complexity of an if statement is the complexity of the executed statements, plus the time for evaluating the condition

Time Complexity

- Rules for computing the time complexity
  - the complexity of an if-then-else statement is the time for evaluating the condition plus the larger of the complexities of the then and else clauses
  - the complexity of a loop is the sum, over all the times around the loop, of the complexity of the body and the complexity of the termination condition

Time Complexity

- Given an algorithm, we analyse the frequency count of each statement and total the sum
- This may give a polynomial $P(n)$:

  $\mathrm{P}(n) = c_k\, n^k + c_{k-1}\, n^{k-1} + \ldots + c_1\, n + c_0$

  where the c_i are constants, c_k are non-zero, and n is a parameter

Time Complexity

- If the big-O notation of a portion of an algorithm is given by:

  $\mathrm{P}(n) = \mathrm{O}(n^k)$

  and on the other hand, if any other step is executed 2^n times or more, we have:

  $\mathrm{c}\, 2^n + \mathrm{P}(n) = \mathrm{O}(2^n)$

Time Complexity

- What about computing the complexity of a recursive algorithm?
- In general, this is more difficult
- The basic technique
  - Identify a recurrence relation implicit in the recursion

    $T(n) = f(T(k)),\ k \in \{1, 2, \dots, n\text{-}1\}$
  - Solve the recurrence relation by finding an expression for $T(n)$ in term which do not involve $T(k)$

Time Complexity

```
int factorial(int n) {
   int factorial_value;

   factorial_value = 0;

   /* compute factorial value recursively */

   if (n <= 1) {
      factorial_value = 1;
   }
   else {
      factorial_value = n * factorial(n-1);
   }
   return (factorial_value);
}
```

Time Complexity

Let the time complexity of the function be *T*(*n*)

… which is what we want to compute!

Now, let's try to analyse the algorithm

Time Complexity

```
                                                              n>1

int factorial(int n)
{
   int factorial_value;                                       1

   factorial_value = 0;                                       1

   if (n <= 1) {                                              1
      factorial_value = 1;                                    0
   }
   else {                                                     1
      factorial_value = n * factorial(n-1);                   T(n-1)
   }
   return (factorial_value);                                  1
}
```

Time Complexity

$T(n) = 5 + T(n\text{-}1)$

$T(n) = c + T(n\text{-}1)$

$T(n\text{-}1) = c + T(n\text{-}2)$

$T(n) = c + c + T(n\text{-}2)$
$= 2c + T(n\text{-}2)$

$T(n\text{-}2) = c + T(n\text{-}3)$

$T(n) = 2c + c + T(n\text{-}3)$
$= 3c + T(n\text{-}3)$

$T(n) = ic + T(n\text{-}i\ )$

Time Complexity

$T(n) = ic + T(n\text{-}i\ )$

Finally, when $i = n$-1

$T(n) = (n\text{-}1)c + T(n\text{-}(n\text{-}1)\ )$
$= (n\text{-}1)c + T(1)$
$= (n\text{-}1)c + d$

Hence, $T(n) = \mathrm{O}(n)$

Space Complexity

Compute the space complexity of an algorithm by analysing the storage requirements (as a function on the input size) in the same way

Space Complexity

For example

- if you read a stream of *n* characters
- and only ever store a constant number of them,
- then it has space complexity $O(1)$

Space Complexity

For example

- if you read a stream of n records
- and store all of them,
- then it has space complexity O(n)

Space Complexity

For example

- if you read a stream of n records
- and store all of them,
- and each record causes the creation of (a constant number) of other records,
- then it still has space complexity O(n)

Space Complexity

For example

- if you read a stream of n records
- and store all of them,
- and each record causes the creation of a number of other records (and the number is proportional to the size of the data set n)
- then it has space complexity O(n^2)

Time vs Space Complexity

In general, we can often decrease the time complexity but this will involve an increase in the space complexity

and *vice versa* (decrease space, increase time)

This is the *time-space tradeoff*

Time vs Space Complexity

For example

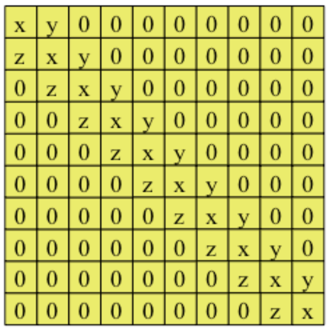
- the average time complexity of an iterative sort (e.g. bubble sort) is $O(n^2)$
- but we can do better:
- the average time complexity of the Quicksort is $O(n \log n)$
- But the Quicksort is recursive and the recursion causes an increase in memory requirements (*i.e.* an increase in space complexity)

Time vs Space Complexity

For example

- The space complexity of 2-D matrix is O(n^2)
- If the matrix is sparse we can do better: we can represent the matrix as a 2-D linked list and often reduce the space complexity to O(n)
- But the time taken to access each element will rise (*i.e.* the time complexity will rise)

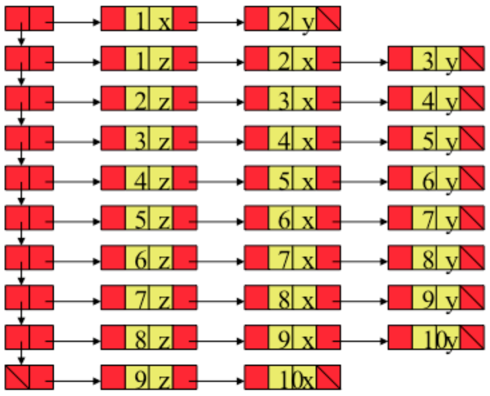
Time vs Space Complexity



x	y	0	0	0	0	0	0	0	0
z	x	y	0	0	0	0	0	0	0
0	z	x	y	0	0	0	0	0	0
0	0	z	x	y	0	0	0	0	0
0	0	0	z	x	y	0	0	0	0
0	0	0	0	z	x	y	0	0	0
0	0	0	0	0	z	x	y	0	0
0	0	0	0	0	0	z	x	y	0
0	0	0	0	0	0	0	z	x	y
0	0	0	0	0	0	0	0	z	x

n x n matrix:

O(n^2) space complexity



2x(2 + 4 + 4) + (n-2)x(2 + 4 + 4 + 4)

= 20 + 14n - 28 = 14n - 8:

O(n) space complexity

Time vs Space Complexity

Order of space complexity for the matrix representation of the banded matrix is $O(n^2)$ >> order of space complexity for the linked list representation $O(n)$

However, the matrix implementation will sometimes be more effective:

Time vs Space Complexity

n^2 <= 14n – 8

n^2 -14n + 8 <= 0

n = ± 13 is the cutoff at which the list representation is more efficient in terms of storage space

Typically, in real engineering problems, n can be much greater than 100 and the saving is very significant

Worst-case and average-case complexity

So far we have looked only at worst-case complexity (i.e. we have developed an upper-bound on complexity)

However, there are times when we are more interested in the average-case complexity (especially it differs significantly)

Worst-case and average-case complexity

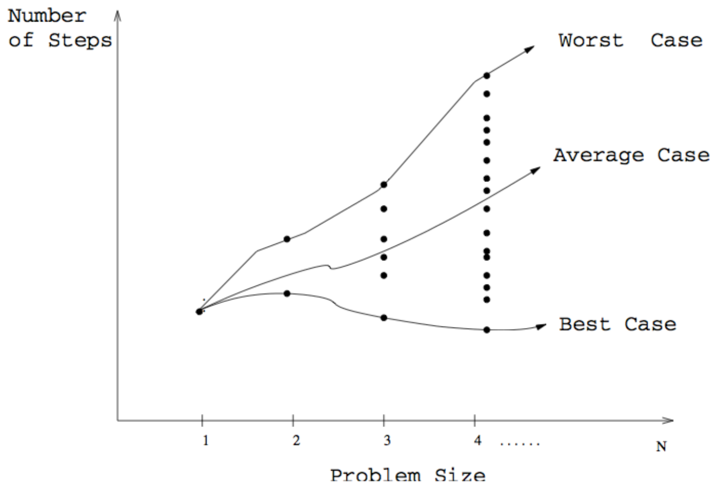
For example

the Quicksort algorithm has

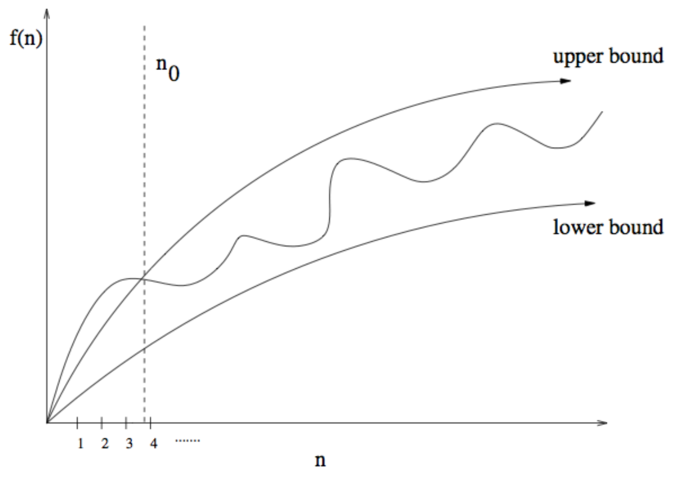
$T(n) = O(n^2)$, worst case (for inversely sorted data)

$T(n) = O(n \log_2 n)$, average case (for randomly ordered data)

Worst-case and average-case complexity



Worst-case and average-case complexity



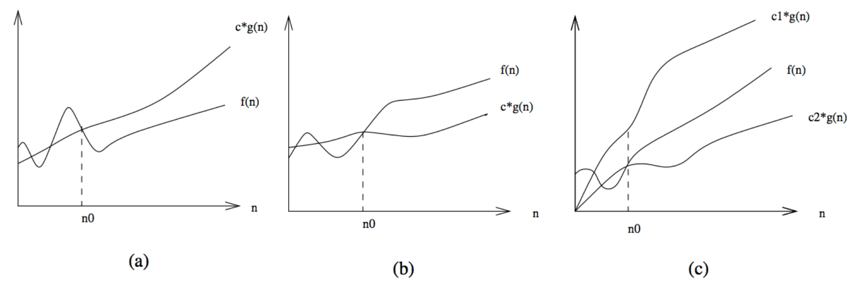
Worst-case and average-case complexity

$f(n) = O(g(n))$ means $c \cdot g(n)$ is an *upper bound* on $f(n)$. Thus there exists some constant c such that $f(n)$ is always $\leq c \cdot g(n)$, for large enough n (i.e. , $n \geq n_0$ for some constant n_0).

$f(n) = \Omega(g(n))$ means $c \cdot g(n)$ is a *lower bound* on $f(n)$. Thus there exists some constant c such that $f(n)$ is always $\geq c \cdot g(n)$, for all $n \geq n_0$.

$f(n) = \Theta(g(n))$ means $c_1 \cdot g(n)$ is an upper bound on $f(n)$ and $c_2 \cdot g(n)$ is a lower bound on $f(n)$, for all $n \geq n_0$. Thus there exist constants c_1 and c_2 such that $f(n) \leq c_1 \cdot g(n)$ and $f(n) \geq c_2 \cdot g(n)$. This means that $g(n)$ provides a nice, tight bound on $f(n)$.

Worst-case and average-case complexity



Illustrating the big (a) O, (b) Ω, and (c) Θ notations