CSE 247
Data Structures

Algorithm Analysis

Algorithm Origin

- The word Algorithm is on the name of Muslim author, renowned mathematician, Abu Ja'far Mohammad Ibn Musa al-Khowarizmi.
- He was born in the eighth century at Khwarizm, in present known as Uzbekistan.

Algorithm definition

- An algorithm is a well-defined procedure that takes some value as input and produces some value as output.

Problem-solving consists of the following five steps:

1. Problem understanding: Define exactly what you are trying to solve the problem
2. Formulation of the problem: Discovery of the fact to break down the complexity
3. Developing an algorithm: understand different alternatives to solve the problem and pick one
4. Implementation of an algorithm: deliver the outcome essential to solve the problem
5. Running it on data

Why Algorithm

- Understanding algorithm is essential for a computer scientist.
- Algorithm is independent of any programming language. Thus, it is the mathematical theory behind a program.
- Algorithm and data structures are interdependent as most of the fast algorithms are fast due to the use of fast data structures and vice versa.
- Many of the courses in the computer science program deal with efficient algorithms and data structures such as they apply to various applications: compilers, operating systems, databases, artificial intelligence, computer graphics, and vision, etc.

Algorithm efficiency

- An important question is: How efficient is an algorithm or piece of code? Efficiency covers lots of resources, including:
- CPU (time) usage
- memory usage
- disk usage
- network usage

Factors affect on algorithms efficiency

- Different implementations may cause an algorithm to run faster/slower
- Some algorithms run faster on some computers
- Algorithms may perform differently depending on data (e.g., sorting often depends on what is being sorted)

Algorithm efficiency

- To compute time exactly, we should count the number of CPU cycles it takes to perform each operation in the algorithm.
- This time computation would be a bit tedious and is not a very practical approach.

- Instead we will count the number of times primitive operations are executed in an algorithm.

- By *primitive operation* we mean a simplest operation such as loop, boolean operations, assignments, exchanges etc.

- we are usually interested in the **worst case**: what are the **max** operations that might be performed for a given problem size (other cases are best case and average case)

Analyzing Algorithm

- In analyzing the performance of an algorithm, usually we are interested in the amount of memory required by an algorithm – its ***space complexity*** – and the time taken for an algorithm to run – its ***time complexity.***

Introduction to Time Complexity

- Be careful to differentiate between:
  - **Performance**: how much time/memory/disk/… is actually used when a program is run. This depends on the machine, compiler, etc. as well as the code.
  - **Complexity**: how do the resource requirements of a program or algorithm scale, i.e., what happens as the size of the problem being solved gets larger.

- Complexity affects performance but not the other way around.

Algorithm Analysis

- How to estimate the time required for an algorithm.
- How to use techniques that drastically reduce the running time of an algorithm.

Analyze above question on linear and Binary search

Search Algorithms

- Linear/ sequential search
- Binary search

Example: Searching Sorted Array

Algorithm 1: Linear Search

```
int search(int A[], int N, int Num) {
    int index = 0;                                      1

    while ((index < N) && (A[index] < Num)){            N+1
        index++;                                        N
    }

   if ((index < N) && (A[index] == Num))                1
       return index;                                    1
  else
       return -1;
}
```

$F(N)=2N+4$

Analyzing Search Algorithm 1

Operations to count: how many times Num is compared to member of array.

Best-case: find the number we are looking for at the first position in the array (1 + 1 = 2 comparisons) O(1)

Average-case: find the number on average half-way down the array (sometimes longer, sometimes shorter)
(N/2+1 comparisons) O(N)

Worst-case: have to compare Num to very element in the array (N + 1 comparisons) O(N)

Example: Searching Sorted Array

- Algorithm 2: Binary Search

Item to be searched = 23

Step 1

1	**5**	**7**	**8**	**13**	**19**	**20**	**23**	**29**
0	1	2	3	4	5	6	7	8

a [mid] = 13
13 < 23
beg = mid + 1 = 5
end = 8
mid = (beg + end)/2 = 13 / 2 = 6

Step 2

1	**5**	**7**	**8**	**13**	**19**	**20**	**23**	**29**
0	1	2	3	4	5	6	7	8

a [mid] = 20
20 < 23
beg = mid + 1 = 7
end = 8
mid = (beg + end)/2 = 15 / 2 = 7

Step 3

1	**5**	**7**	**8**	**13**	**19**	**20**	**23**	**29**
0	1	2	3	4	5	6	7	8

a [mid] = 23
23 = 23
loc = mid

Return location 7

Search Algorithm 2: Binary Search

```
int search(int A[], int Num) {
  int first = 0;                                  1
  int last = A.length - 1;                        1
  int mid = (first + last) / 2;                   1

  while (first < last) {                          logN+1
     if (Num< A[mid] )                            logN
          last = mid - 1;
    else if(A[mid]==Num)                          logN
          return mid;     //Num found at mid
    else
          first = mid + 1;                        logN
          mid = (first + last) / 2;               logN
  }
  if (A[mid] == Num)                              1
   return mid;     // Num found at mid            1
  else
   return -1;     // Num not found
}
```

$$F(N)=5\log N+6$$

100

50 50

25 25

12 13

6 6

3 3

1 2

1 1

Analyzing Binary Search

One comparison after loop
First time through loop, toss half of array (2 comps)
Second time, half remainder (1/4 original) 2 comps
Third time, half remainder (1/8 original) 2 comps
…

Loop Iteration	Remaining Elements
1	N/2
2	N/4
3	N/8
4	N/16
…	
??	1

How long to get to 1?

$= N/2 + N/2/2 + N/4/2 + \ldots + 1$

$= N/2 + N/4 + N/8 + N/16 + \ldots + 1$

$= N/2^1 + N/2^2 + N/2^3 + N/2^4 + \ldots + 1$

$= N/2^k >= 1$, where k= 1, 2, 3,…

$= \quad N >= 2^k$

Apply log

$\log N >= k \log 2$

Analyzing Binary Search (cont)

Looking at the problem in reverse, how long to double the number until we get to N?

$$N = 2^X$$

For a list of ten thousand elements, how many total number of statement will be executed to search an element in both algorithms?

Binary search in worst-case $O(\log_2 N) = \log_2 (10000) = 13$

Sequential search in worst-case $O(N) = 10000$

Analysis: A Better Approach

Idea: characterize performance in terms of key operation(s)

- Sorting:
  - count number of times two values compared
  - count number of times two values swapped
- Search:
  - count number of times value being searched
- Recursive function:
  - count number of recursive calls

Analysis in General

Want to comment on the "general" performance of the algorithm

- Measure for several examples, but what does this tell us in general?
- Instead, assess performance in an abstract manner

Idea: analyze performance as size of problem grows

Examples:

- Sorting: how many comparisons for array of size N?
- Searching: #comparisons for array of size N

May be difficult to discover a reasonable formula

The growth rate of function as N Grows

Function	10	100	1000	10000	100000
$\log_2 N$	3	6	9	13	16
N	10	100	1000	10000	100000
$N \log_2 N$	30	664	9965	10^5	10^6
N^2	10^2	10^4	10^6	10^8	10^{10}
N^3	10^3	10^6	10^9	10^{12}	10^{15}
2^N	10^3	10^{30}	10^{301}	10^{3010}	10^{30103}

How to Compare Formulas?

$$50N^2 + 31N^3 + 24N + 15 \qquad 3N^2 + N + 21 + 4 * 3^N$$

Answer depends on value of N:

N	$50N^2+31N^3+24N+15$	$3N^2+N+21+4*3^N$
1	120	37
2	511	71
3	1374	159
4	2895	397
5	5260	1073
6	8655	3051
7	13266	8923
8	19279	26465
9	26880	79005
10	36255	236527

What Happened?

N	$3N^2 + N + 21 + 4*3^N$	$4*3^N$	$\%ofTotal$
1	37	12	32.4
2	71	36	50.7
3	159	108	67.9
4	397	324	81.6
5	1073	972	90.6
6	3051	2916	95.6
7	8923	8748	98.0
8	26465	26244	99.2
9	79005	78732	99.7
10	236527	236196	99.9

- Higher-order term dominated the sum

Order of Magnitude Analysis

Measure speed with respect to the part of the sum that grows quickest

$$50N^2 + 31N^3 + 24N + 15$$

$$3N^2 + N + 21 + 4 * 3^N$$

Ordering:

$$1 < \log_2 N < N < N\log_2 N < N^2 < N^3 < 2^N < 3^N$$

Order of Magnitude Analysis (cont)

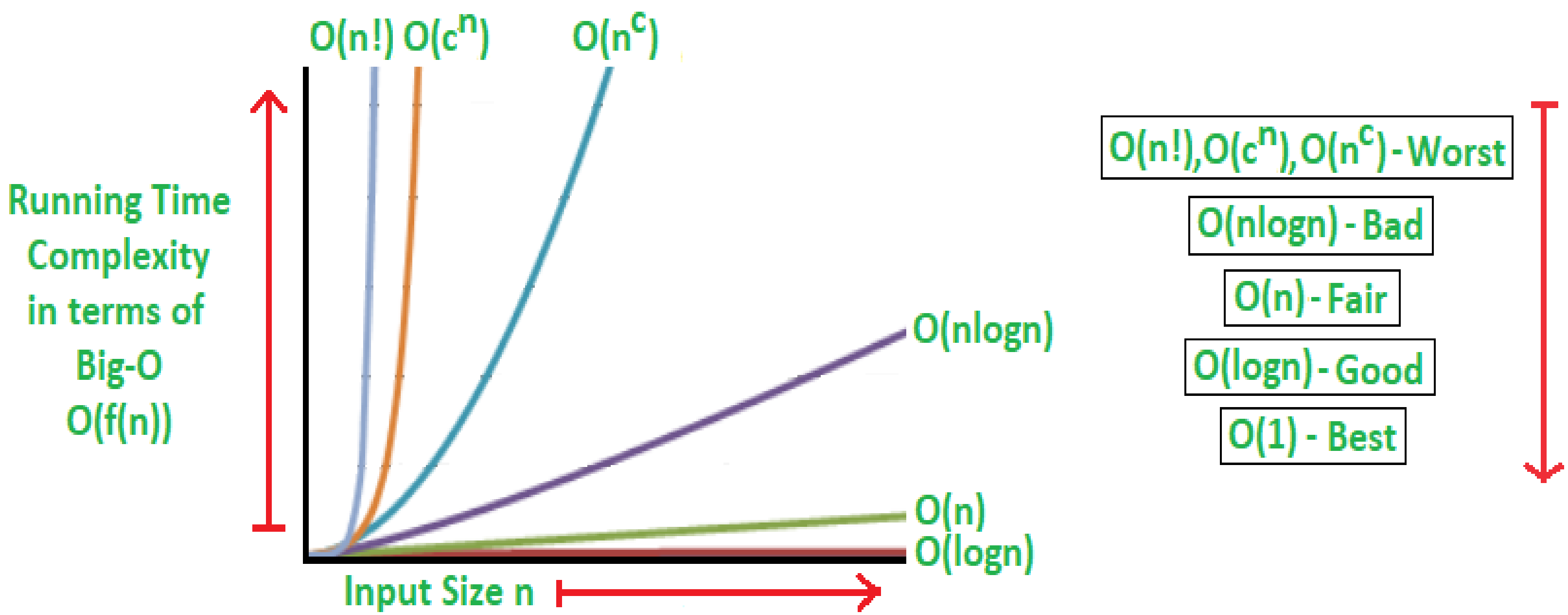
Furthermore, simply ignore any constants in front of term and simply report general class of the term:

$50N^2 + 31N^3 + 24N + 15$ grows proportionally to N^3

$3N^2 + N + 21 + 4 * 3^N$ grows proportionally to 3^N

When comparing algorithms, determine formulas to count operation(s) of interest, then compare dominant terms of formulas.

The growth rate of function as N Grows



Analysis where Results Vary

Types of analyses:

- Best-case: how fastest an algorithm can run for a problem of size N?
- Average-case: on average how fast does an algorithm run for a problem of size N?
- Worst-case: how longest an algorithm can run for a problem of size N?

Motivation for Asymptotic Analysis

- Count operations instead of time
- Focus on how the performance scales:
  - If list is twice as long, how much more time does it take to search it?
- Go beyond input size, for example the data structure used to store the data.

Asymptotic notation

- Big O notation
  - *Symbol is O,*
  - Worst case analysis (find upper bound of the function)
- Big Omega notation
  - Symbol is Ω,
  - Best case analysis (find lower bound of the function)
- Big Theta notation
  - Symbol is θ,
  - Average case analysis (find upper bound and lower bound where both are same)

Big-O Notation

- Computer scientists like to categorize algorithms using Big-O notation.
- For sufficiently large N, the value of a function is largely determined by its dominant term.
- Big-O notation is used to capture the most dominant term in function and to represent the growth rate.
- Big-O notation also allows us to establish a relative order among functions by comparing dominant terms
- Thus, if running time of an algorithm is $O(N^2)$, then ignoring constant , it can be guaranteeing that running time bound to quadratic function.

Analyzing Running Time

T(n), or the running time of a particular algorithm on input of size n, is taken to be the number of times the instructions in the algorithm are executed. Pseudo code algorithm illustrates the calculation of the mean (average) of a set of n numbers:

```
1. n = read input from user
2. sum = 0
3. i = 0
4. while i < n
5.     number = read input from user
6.     sum = sum + number
7.     i = i + 1
8. mean = sum / n
```

The computing time for this algorithm in terms on input size n is: $T(n) = 4n + 5$.

Statement	Number of times executed
1	1
2	1
3	1
4	n+1
5	n
6	n
7	n
8	1

Big O Analysis

- $F(n) = Og(n)$, means $f(n)$ and $g(n)$ grow in same way as their input grows.
- In practice, Big oh analysis ignore constant and estimate the rate of growth of a function by considering the $g(n)$ a closest upper bound of $f(n)$.
- $10000000=O(1)$, because the number of steps doesn't change with the input size n.
- Keep only dominant term

Big-O Notation

- Let's consider a function F on N, where N is a measure of the size of the problem we try to solve, e.g., F(N) is O(C g(N)) if: So, then we can say F(N) grows no faster than g(N).

Example 1

F(N)=2N+2

F(N) <= c(g(N))

2N+2<= 2N+2N

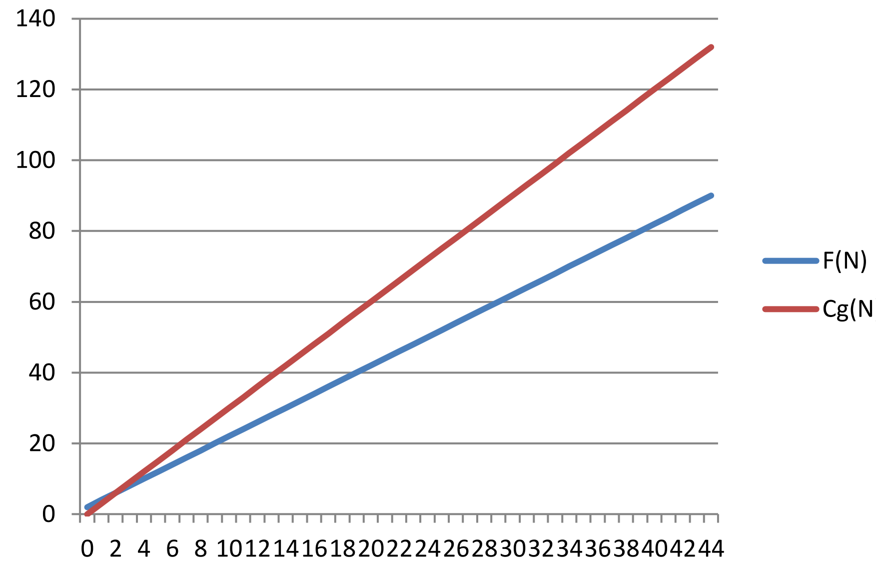
<= 4N where N>=1

So c=4 and g(N)=N

We can say,

c(g(N)) is upper bound of F(N)

Thus, F(N) is classify as *O*(N)



- Although n^2, n^3 are also upper bound of F(N), however nearest upper bound should be selected.

To keep this equality N start from
2N+2<=4N
2<=4N-2N
2<=2N ➔ N>=1

Quratulain

Big-O Notation

- If you tell a computer scientist that they can choose between two algorithms:
  - one whose time complexity is O(N)
  - Another is O(N^2),

then the O(N) algorithm will likely be chosen.

Question

$F(n)=2n^2 + 3n + 10$

Find out the Big O of F(n)?

Can we say that Big O of above F(n) is $O(n^3)$?

Answer

- $F(n)=2n^2 + 3n +10$

Can we say that Big O of F(n) is $O(n^3)$?

- Answer is YES according to Big O definition. Because
  - Big O defines the upper bound of the function and
  - n^3 is upper bound of function $2n^2 + 3n +10$.

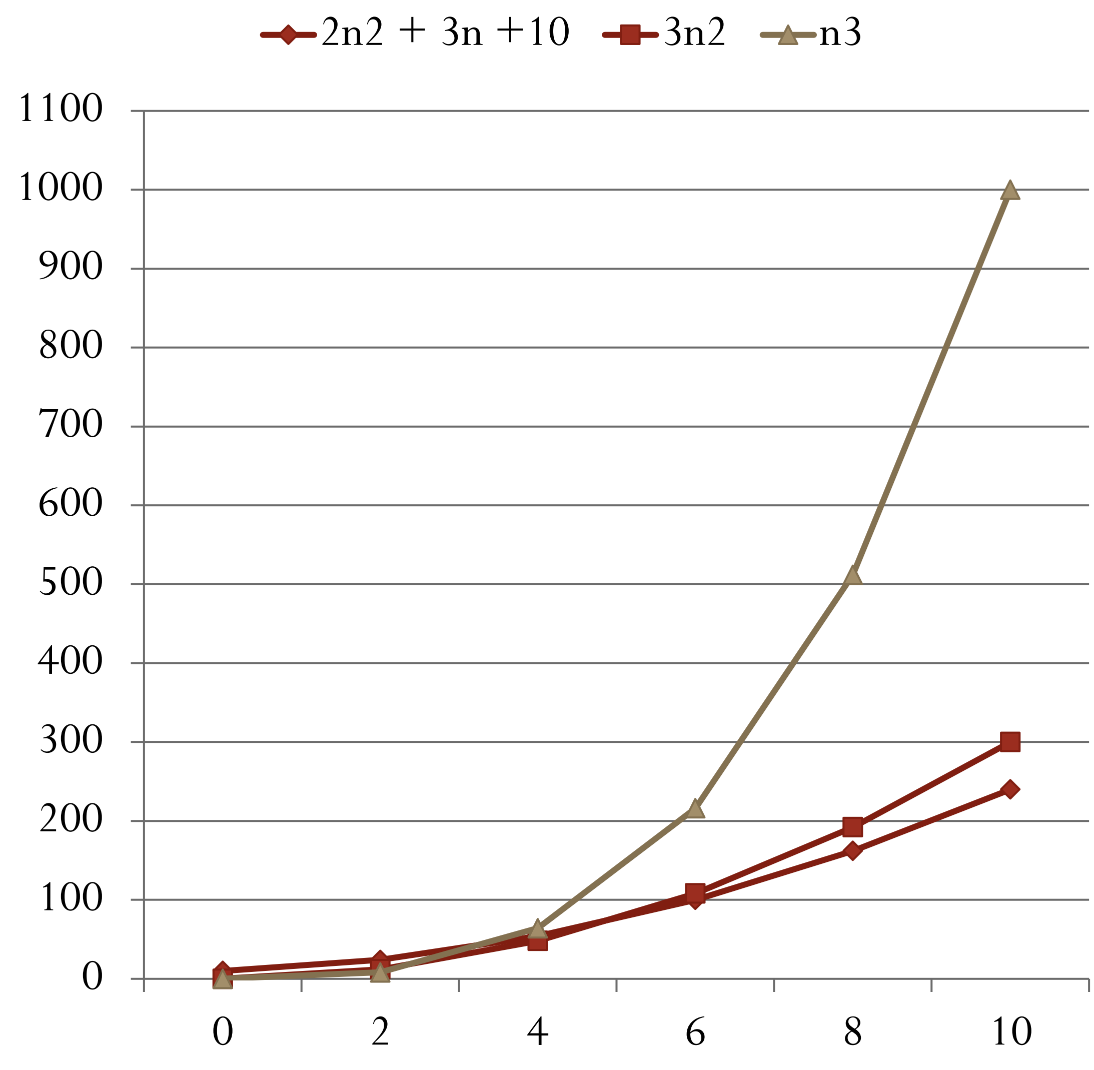
However, for analysis in practice the closest upper bound should be selected. Here for F(n), $3n^2$ is the closest upper bound where g(n) is n^2

Thus F(n) is $O(n^2)$.

Answer cont.

Closest upper bound is $3n^2$

	$2n^2 + 3n + 10$	$3n^2$	n^3
0	10	0	0
2	24	12	8
4	54	48	64
6	100	108	216
8	162	192	512
10	240	300	1000



Question

- Suppose you have two algorithms (Algorithm A and Algorithm B) and that both algorithms are O(N). Does this mean that they both take the same amount of time to run?

Answer:

No, it might not take same amount of time, reasons are:

- Constant factor (Algo A= 2n+8, Algo B= 100n+7)
- Hardware
- Worst case depends on data occurrence/arrival (e.g algorithm A has lots of worst cases while Algorithm B might not.)

Example

We will now look at various Java code segments and analyze the time complexity of each:

```
int count = 0;
int sum = 0;
while( count < N )
{
sum += count;
System.out.println( sum );
count++;
}
```

Total number of constant time statements executed...?

Example

```
int count = 0;
int sum = 0;
    while( count<N )    n+1
    {
    int index = 0;              n
    while( index < N )       n(n+1)
    {
    sum += index * count;          n(n)
    System.out.println( sum );   n(n)
    index++;                         n(n)
    }
    count++ ;                              n              4n^2+4n +2 =O(n^2)
    }
```

Total number of constant time statements executed ….?

Example

```
int count = 1; // suppose N and M >1
int sum = 0;
    while( count<N )
    {
        int index = 0;
        while( index < M )
        {
            sum += index * count;
            System.out.println( sum );
            index++;
        }
        count=count*2 ;
    }
```

Total number of constant time statements executed ….?

Example

```
int count = 0; // suppose N and M >1
int sum = 0;
    while( N>count )
    {
        int index = 0;
        while( index < M )
            {
            sum += index * count;
            System.out.println( sum );
            index++;
            }
        N=N/2 ;
    }
```

Total number of constant time statements executed ….?

Example

```
int count = 0;
    while( count < N )
    {
    int index = 0;
    while( index < count + 1 )
    {
    sum++;
    index++;
    }
    count++;
    }
```

Find bigOh…?

Example

```
int count = N;
int sum = 0;
    while( count > 0 )
    {
    sum += count;
    count = count / 2;
    }
```

Find bigOh…?

Quiz

- What is the tightest correct upper bound on the running time of a insert operation on an dynamic array currently containing n elements in the worst-case?

- Ans.

If the dynamic array needs to be resized to accommodate the new element, it will take $O(n)$ to create a new array of double size and copy all the current elements into it.

Quiz

- Suppose that we have an algorithm ALGO with running time exactly $10n^2$. How much slower does ALGO get when you double the input size n?
- twice as slow
- 4 times slower
- 10 times slower
- 40 times slower

Answer:

Initially, the running time is n^2. On doubling input size, it becomes $(2n)^2 = 4n^2$,i.e. it increases four times.

Quiz

- Suppose that we have different algorithms with time complexity $\log n$, n, n^2, n^3 . How much slower do each of these algorithms get when you double the input size n?
- Suppose for n=100 and double size is 200.

Time Complexity	**n= 100 (data size)**	**2n (double data size)**	**slower**
logn	6	7	Approx. same
n	100	200	2-time slower
nlogn	600	1400	More than 2-time slower
n^2	10,000	40000	4-time slower
n^3	10,00,000	80,00,000	8-time slower

Quiz

- What is the result of running reduce(vals) when vals refers to an integer array with values [1,2,5,3]?

```
public static void reduce (int[] vals) {
 int minIndex =0;
for (int i=0; i < vals.length; i++) {
    if (vals[i] < vals[minIndex] ) {
            minIndex = i;
}
  }
 int minVal = vals[minIndex];
 for (int i=0; i < vals.length; i++) {
        vals[i] = vals[i] - minVal;
 }
}
```

Result is [0,1,4,2]

O(n)

Reference

- https://introprogramming.info/english-intro-csharp-book/read-online/chapter-19-data-structures-and-algorithm-complexity/
- https://www.geeksforgeeks.org/understanding-time-complexity-simple-examples/