Lecture 3

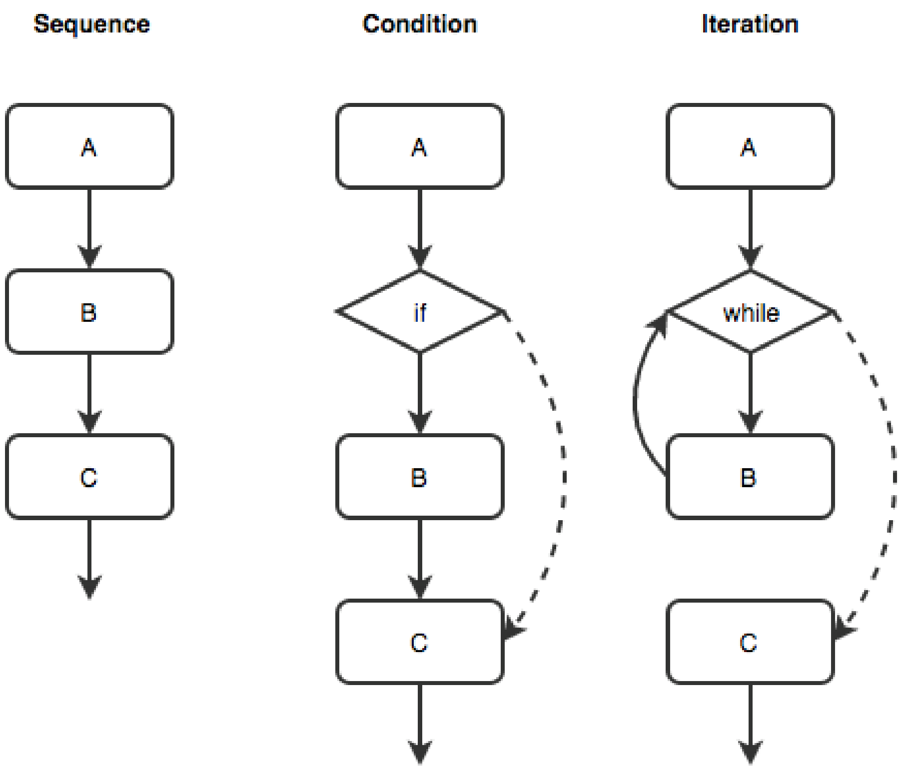
Algorithmic Complexity

Dr. Yusuf H. Sahin
Istanbul Technical University

sahinyu@itu.edu.tr

Algorithm Efficiency Matters

- When comparing algorithms for the same problem, one might be significantly more efficient. Recognizing and selecting the most efficient algorithm is essential.
- **Algorithmics:** The study of designing and analyzing efficient algorithms
  - focusing on systematic techniques for efficiency



- In linear functions (no loops or recursion), efficiency depends on the number of instructions and the computer's speed, typically not impacting overall program performance.
- As we study specific examples, we generally discuss the algorithm's efficiency as a function of the number of elements to be processed.

$$f(n) = efficiency$$

Linear Loops

- Let's begin with a basic loop. Assuming i is an integer, the loop will run 1000 times. The number of iterations is directly related to the loop's multiplier.

```
for (i = 0; i < 1000; i++)
    application code
```

$$f(n) = n$$

- The answer is not always that simple. For instance, take the following loop into consideration.

```
for (i = 0; i < 1000; i += 2)
    application code
```

$$f(n) = \frac{n}{2}$$

- The time cost for the loop is a linear function of n.

Logarithmic Loops

- In a logarithmic loop, the variable is multiplied or divided with each iteration.

Multiply Loops

```
for (i = 0; i < 1000; i *= 2)
    application code
```

Divide Loops

```
for (i = 1000; i < 1;  i /= 2)
    application code
```

- Loop continues while the following conditions are true.

multiply $2^{Iterations} < 1000$

divide $1000 / 2^{Iterations} >= 1$

$$f(n) = logn$$

Multiply		Divide	
Iteration	Value of `i`	Iteration	Value of `i`
1	1	1	1000
2	2	2	500
3	4	3	250
4	8	4	125
5	16	5	62
6	32	6	31
7	64	7	15
8	128	8	7
9	256	9	3
10	512	10	1
(exit)	1024	(exit)	0

Nested Loops

- When analyzing nested loops, it's important to determine the number of iterations completed by each loop.

#Iterations = #outer loop iterations * #inner loop iterations

- Next, we examine three types of nested loops: linear-logarithmic, quadratic, and dependent quadratic.

Linear Logarithmic

```
for (i = 0; i < 10; i++)
    for (j = 1; j < 10; j *= 2)
        application code
```

$$f(n) = nlogn$$

Quadratic

```
for (i = 0; i < 10; i++)
    for (j = 0; j < 10; j++)
        application code
```

$$f(n) = n^2$$

Dependent Quadratic

```
for (i = 0; i < 10; i++)
    for (j = 0; j < i; j++)
        application code
```

$$f(n) = n\left(\frac{n-1}{2}\right)$$

Big-O Notation

- We've shown that the number of statements executed for n elements is a function of n, expressed as f(n).
- While the equation may be complex, a dominant factor usually dictates the result's magnitude.
- This key factor, called big-O, represents the function's order
  - such as O(n) for linear growth.
  - O(n^2) for quadratic.

- The big-O notation can be derived from $f(n)$ using the following steps:

  1- In each term, set the coefficient of the term to 1.

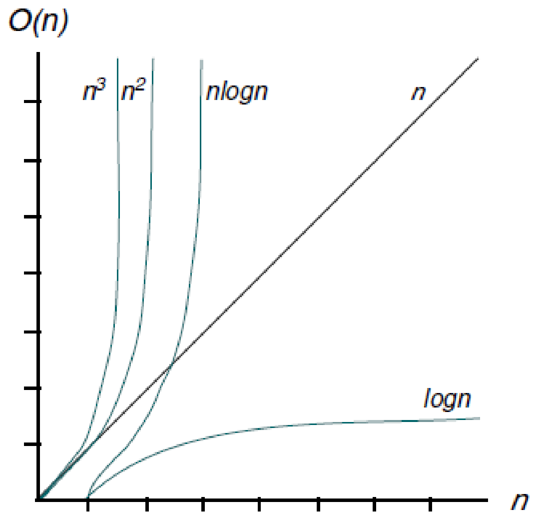
  2- Keep the largest term in the function and discard the others. Terms are ranked from lowest to highest as logn n nlogn n^2 n^3 ... n^k 2^n n!

$$f(n) = n\frac{(n+1)}{2} = \frac{1}{2}n^2 + \frac{1}{2}n \longrightarrow O(n^2)$$

Standard Measures of Efficiency

- For $n = 10.000$:

Efficiency	Big-O	Iterations	Estimated Time
Logarithmic	$O(\log n)$	14	microseconds
Linear	$O(n)$	10,000	seconds
Linear logarithmic	$O(n(\log n))$	140,000	seconds
Quadratic	$O(n^2)$	$10{,}000^2$	minutes
Polynomial	$O(n^k)$	$10{,}000^k$	hours
Exponential	$O(c^n)$	$2^{10{,}000}$	intractable
Factorial	$O(n!)$	10,000!	intractable



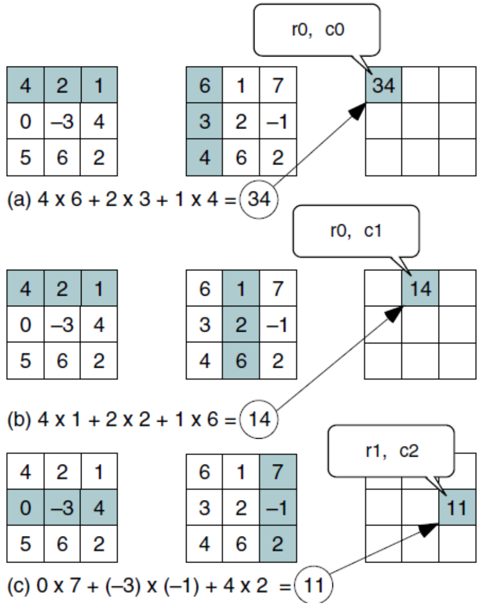
- When working with just 10 elements and the time is a fraction of a second, the difference between two algorithms is negligible. However, as the number of elements increases, the performance gap between algorithms can become significant.

Big-O Analysis Examples

- Adding matrices:

$$\begin{bmatrix} 4 & 2 & 1 \\ 0 & -3 & 4 \\ 5 & 6 & 2 \end{bmatrix} + \begin{bmatrix} 6 & 1 & 7 \\ 3 & 2 & -1 \\ 4 & 6 & 2 \end{bmatrix} = \begin{bmatrix} 10 & 3 & 8 \\ 3 & -1 & 3 \\ 9 & 12 & 4 \end{bmatrix}$$

- Multiplying Square matrices:



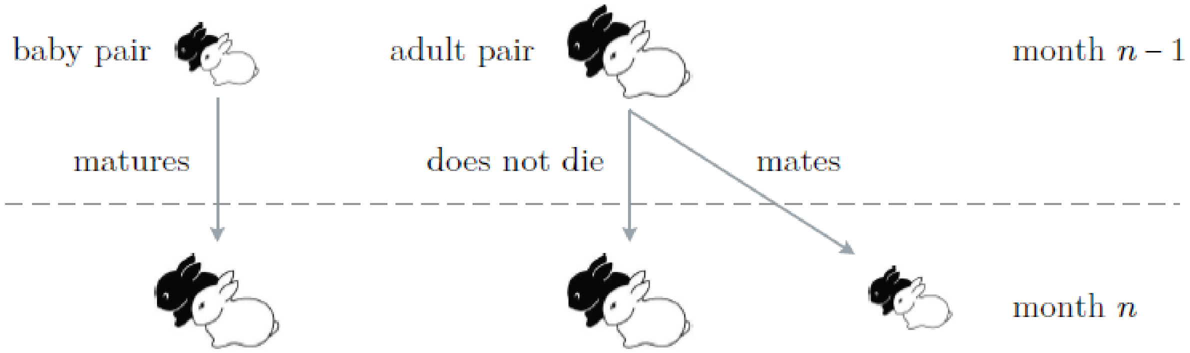

```
Algorithm addMatrix (matrix1, matrix2, size, matrix3)
Add matrix1 to matrix2 and place results in matrix3
   Pre  matrix1 and matrix2 have data
        size is number of columns or rows in matrix
   Post matrices added--result in matrix3
1  loop (not end of row)
   1  loop (not end of column)
      1  add matrix1 and matrix2 cells
      2  store sum in matrix3
   2  end loop
2  end loop
end addMatrix
```

```
Algorithm multiMatrix (matrix1, matrix2, size, matrix3)
Multiply matrix1 by matrix2 and place product in matrix3
   Pre  matrix1 and matrix2 have data
        size is number of columns and rows in matrix
   Post matrices multiplied--result in matrix3
1  loop (not end of row)
   1  loop (not end of column)
      1  loop (size of row times)
         1  calculate sum of
                (all row cells) * (all column cells)
         2  store sum in matrix3
   2  end loop
2  end loop
3  return
end multiMatrix
```

An Intro Example for Recursion

- In a rabbit population, the rules for the growth are as listed below:
  - Initially, a newly born pair of rabbits, one male and one female, are placed in a field.
  - The rabbits take one month to mature.
  - Mature rabbit pairs mate at the beginning of every month, and give birth to another pair of newly born rabbits at the beginning of the next month.
  - The female rabbit always gives birth to one male and one female rabbit, who will only mate with themselves.
  - The rabbits never die.

- A recursive function $A(n)$ could be written to obtain the number of adult pairs for month n.
- Another function $B(n)$ shows the baby pairs for that month.
- Write the definitions for functions A and B!



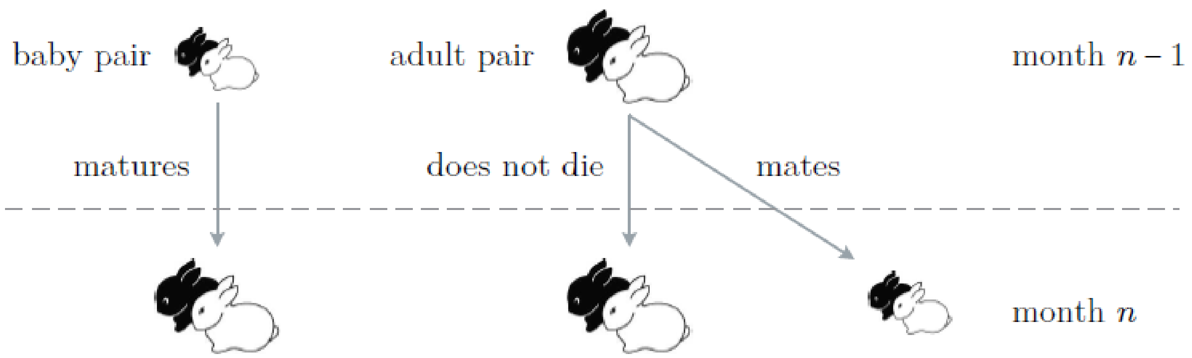
From «Introduction to Recursive Programming 1st Edition»,Manuel Rubio-Sanchez

An Intro Example for Recursion

- In a rabbit population, the rules for the growth are as listed below:
  - Initially, a newly born pair of rabbits, one male and one female, are placed in a field.
  - The rabbits take one month to mature.
  - Mature rabbit pairs mate at the beginning of every month, and give birth to another pair of newly born rabbits at the beginning of the next month.
  - The female rabbit always gives birth to one male and one female rabbit, who will only mate with themselves.
  - The rabbits never die.

$$A(n) = \begin{cases} 0 & if\ n = 1 \\ A(n-1) + B(n-1) & if\ n > 1 \end{cases}$$

$$B(n) = \begin{cases} 1 & if\ n = 1 \\ A(n-1) & if\ n > 1 \end{cases}$$



So, what are the time complexities of these functions?

From «Introduction to Recursive Programming 1st Edition»,Manuel Rubio-Sanchez

Into the rabbit hole…

```c
#include <stdio.h>

int B(int n);

// Function A(n)
int A(int n) {
    if (n == 1)
        return 0;
    else
        return A(n - 1) + B(n - 1);
}

int B(int n) {
    if (n == 1)
        return 1;
    else
        return A(n - 1);
}

int main() {
    int n = 2;  // Example input
    printf("A(%d) = %d\n", n, A(n));
    printf("B(%d) = %d\n", n, B(n));
    return 0;
}
```

Into the rabbit hole…

```c
#include <stdio.h>

int B(int n);

// Function A(n)
int A(int n) {
    if (n == 1)
        return 0;
    else
        return A(n - 1) + B(n - 1);
}

int B(int n) {
    if (n == 1)
        return 1;
    else
        return A(n - 1);
}

int main() {
    int n = 2;  // Example input
    printf("A(%d) = %d\n", n, A(n));
    printf("B(%d) = %d\n", n, B(n));
    return 0;
}
```

Time complexity of A(n) and B(n) are both $O(2^n)$