DATA STRUCTURES & ALGORITHMS

ALGORITHM ANALYSIS

issntt@nus.edu.sg

On the last ~~episode~~ lecture

Image by Robin Higgins from Pixabay

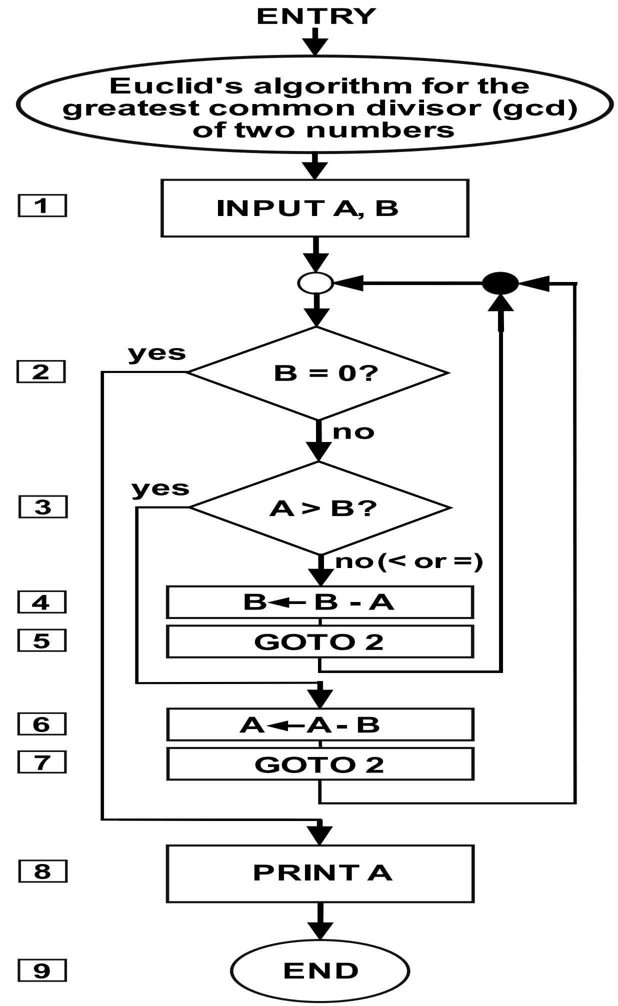
So, we have **two implementations** for **the same** ADT List. Can we just **ignore Linked List** and **always use Array List** in our coding?

Outline

- **Algorithms**
- Efficiency of Algorithms
  - Experiments approach and its issues
  - Number of operations
  - Asymptotic analysis – Big-Oh notation

Algorithms

An algorithm is a **step-by-step procedure** for **performing some task** in a **finite** amount of **time**



This example shows the Euclid's algorithm depicted using a flow chart

Image by [Wikipedia]

An example Task

Given an integer N, **compute** the **sum** of all integer **from 1 to N**, including N

Image by Gerd Altmann from Pixabay

How many different algorithms can you think of?

3 algorithms to compute the sum

Algorithm 1

Add 1, 2, 3… n to sum

```
sum = 0
for i = 1 to N
   sum = sum + i
```

Algorithm 2

Add 1, (1+1), (1+1+1), … (1+1+1… +1) to sum

```
sum = 0
for i = 1 to N
  for j = 1 to i
    sum = sum + 1
```

Algorithm 3

Use the sum of consecutive numbers formula*

```
sum =
  N * (N + 1) / 2
```

Which algorithm is more efficient?

* Sum of consecutive numbers

Outline

- Algorithms
- **Efficiency of Algorithms**
  - Experiments approach and its issues
  - Number of operations
  - Asymptotic analysis – Big-Oh notation

What does "efficiency" mean?

- **Time**: **How fast** an algorithm runs
- **Space**: **How much memory** an algorithm needs
- **Usually**, we care **more about time** than space

Image by Nile from Pixabay

Why?

Next

How can we **analyze** an **algorithm** to see if it runs **faster** than another one?

Image by Thomas Wolter from Pixabay

Hey! Computers are **very fast** nowadays. So who care?

Outline

- Algorithms
- **Efficiency of Algorithms**
  - **Experimental approach and its issues**
  - Number of operations
  - Asymptotic analysis – Big-Oh notation

Approach 1 – Experiments

We **implement** each of algorithms, **then measure** how long they run

```
static void sum1(long n)
{
  Stopwatch sw =
    Stopwatch.StartNew();

  long sum = 0;
  for (long i = 1;
         i <= n; i++)
    sum += i;

  Console.WriteLine(
    "Run {1}ns.",
    ElapsedNanoSecond(sw));
}
```

```
static void sum2(long n) {
  Stopwatch sw =
    Stopwatch.StartNew();

  long sum = 0;
  for (long i = 1;
         i <= n; i++) {
    for (long j = 1;
           j <= i; j++)
      sum += 1;
  }

  Console.WriteLine(
      "Run {1}ns.",
      ElapsedNanoSecond(sw));
}
```

Same for `sum3(long n)`

The code is just for illustrative purpose

Approach 1 – Experiments

```
static void Main(string[] args)  {
    sum1(1000);
    sum1(10000);
    sum1(100000);

    sum2(1000);
    sum2(10000);
    sum2(100000);

    sum3(1000);
    sum3(10000);
    sum3(100000);
}
```

```
Run 2000ns.
Run 13500ns.
Run 132200ns.

Run 822100ns.
Run 81774400ns.
Run 8129553800ns.

Run 100ns.
Run 100ns.
Run 100ns.
```

What issues may this approach have?

Issues of Approach 1

Difficult to **directly compare** 2 algorithms

- The measured times will **vary** greatly **from machine to machine**
- It may likely **vary from run to run**, even the same machine

Can only be done on a **limited set of inputs**

- E.g., with n = 100, 1000, 100000
- How about 10000000?

Algorithms must be **fully implemented**

- Are you willing to **spend** a lot of **time implementing complicated algorithms** just to see they don't run fast enough?

Looking for improvements

Image by truthseeker08 from Pixabay

If only there is a solution that:

1. **Not depend** on the **hardware** and **software**
2. **Not need** to **implement algorithms** first
3. **All possible inputs** are considered
4. Evaluating the **relative efficiency** only

Outline

- Algorithms
- **Efficiency of Algorithms**
  - Experimental approach and its issues
  - **Number of operations**
  - Asymptotic analysis – Big-Oh notation

Approach 2 - Key idea

Instead,

count the

number of

operations

in the

algorithm

Image by Pexels from Pixabay

Primitive Operations

A primitive operation has **constant execution time**, including:

1. Assign a value to a variable
2. Follow an object reference
3. Perform an arithmetic operation
4. Compare two numbers
5. Access a single element of an array by index
6. Call a method
7. Return from a method

Quiz

Count the number of primitive operations for the algorithm below

Counting sum of number from 1 to N

Input: N

```
sum = 0
loop i from 1 to N
   sum = sum + i
```

```
1 + 1 + 3N + 1
```

Do we need to **implement** the algorithm **before** being able to **count**?

A challenging question

There are one elephant and one mouse

- The elephant weighs 5880 kg
- The mouse weighs 6.5g

What is their total weight?

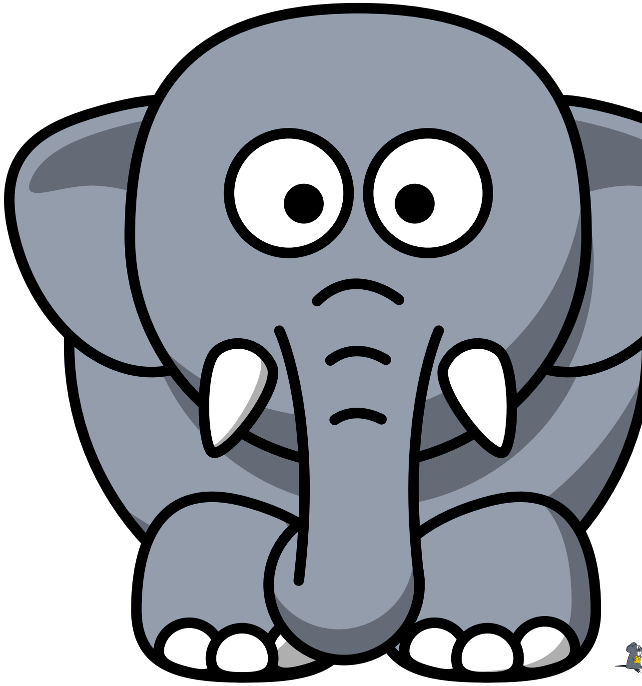


Image by OpenClipart-Vectors from Pixabay

Basic Operations

Basic operations are the **most significant contributor** to its total time. The following operations are ~~***not basic***~~:

1. ~~*Assign a value to a variable*~~
2. ~~*Follow an object reference*~~
3. Perform an arithmetic operation
   - ~~*Operations that control the loop*~~
4. Compare two numbers
5. Access a single element of an array by index
6. ~~*Call a method*~~
7. ~~*Return from a method*~~

Count **basic operations only**!

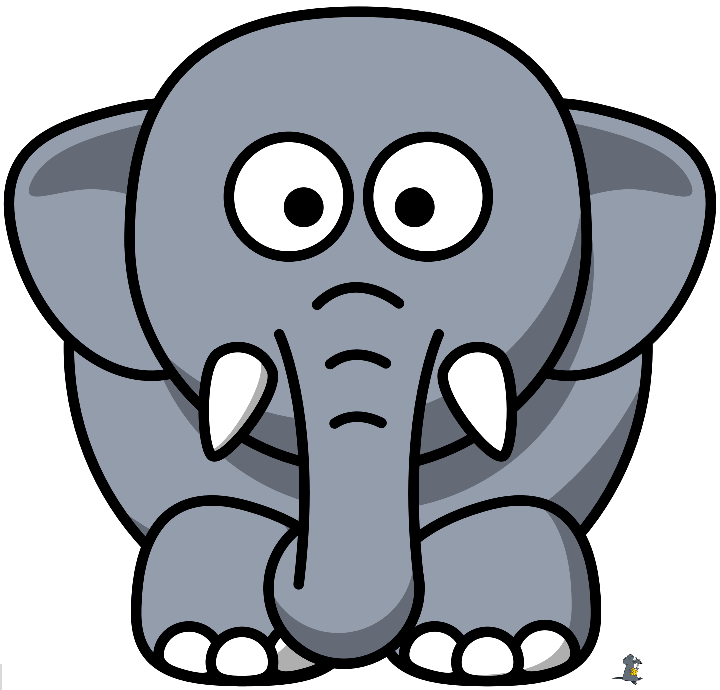


Image by OpenClipart-Vectors from Pixabay

Basic Operations

Let’s re-count only the number of basic operations for the algorithm below

Counting sum of number from 1 to N

Input: N

```
sum = 0
for i = 1 to N
   sum = sum + i
```

N

Have we considered **all** the **possible inputs** in our counting?

Quiz

Count the number of basic operations for the algorithm below

Find a value in an array

Input: an array and a value

```
loop i from 0 to array’s length - 1
  if (arr[i] == value)
     return i
return -1
```

Does the number of operations **depend on** the **input values**?

Best Case and Worst Case

For some algorithms, execution **time depends not only on** the **size** of the data set, **but also** the data **values**

Best case – the number of basic operations **never less than**

- Last example: the best case is 1

Worst case – the number of basic operations **never more than**

- Last example: the worst case is N

Average cases – the number of basic operations **averaged over all possible inputs**

Can you guess: which one is more of our interest?

Next

The **number** of basic **operations** in the **worst cases** of two algorithms solving the **same problem** are

Algorithm 1	Algorithm 2
$7n + n\log n + 2$	$2n + 1 + n^2$

In general, which one is faster?

Outline

- Algorithms
- **Efficiency of Algorithms**
  - Experimental approach and its issues
  - Number of operations
  - **Asymptotic analysis – Big-Oh notation**

Asymptotic Analysis

Explore by yourself

Taking a **“big-picture” approach**

Estimate the number of basic operations executed, **rather than** the **exact** number

Focus on the **growth rate**

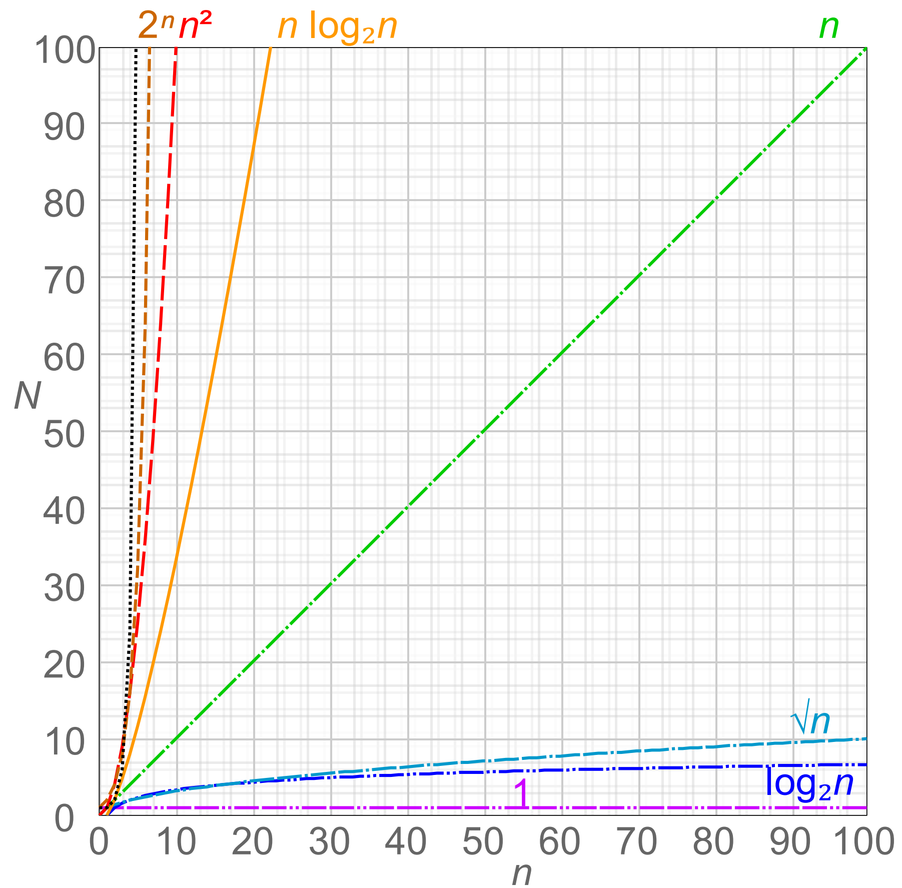


Image by Cmglee - Own work, CC BY-SA 4.0, Wikipedia

Big-Oh Notation

Explore by yourself

- Let $f(n)$ and $g(n)$ be 2 functions mapping positive integers to positive real numbers

- A function $f(n)$ is of order at most $g(n)$, i.e., $f(n)$ is $O(g(n))$ if:
  - There is a real constant $c>0$, and a positive integer N
  - Such that $f(n)<=cxg(n)$ for all $n>=N$
  - It means, $cxg(n)$ is an **upper bound** on $f(n)$ when **n is large enough**

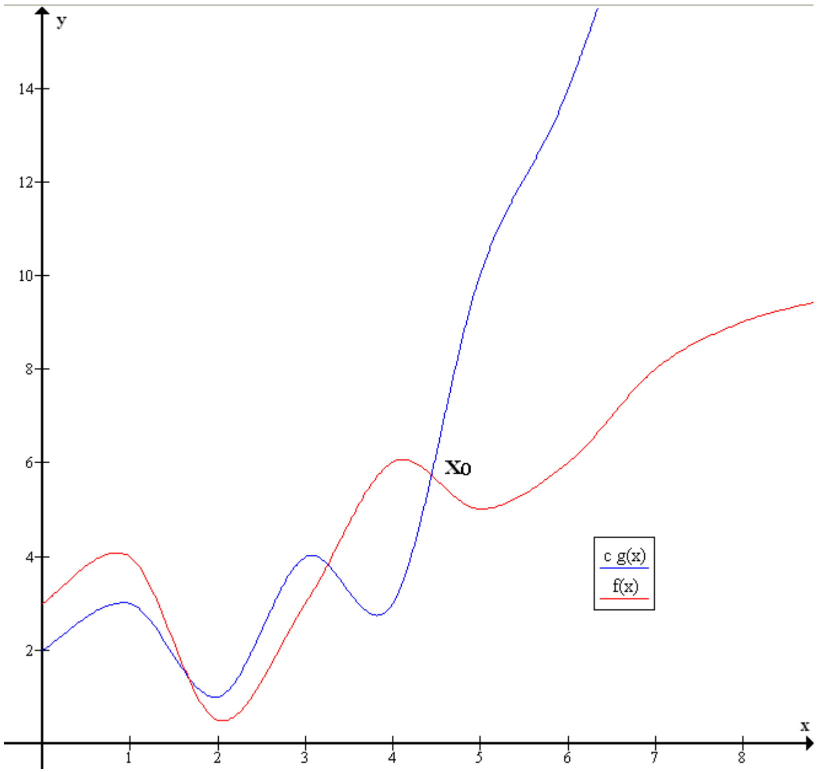


Image by Fede_Reghe, Wikipedia,

Properties of Big-Oh Notation

Explore by yourself

$O(k(g(n)) = O(g(n))$ for a constant k

$O(g_1(n))+O(g_2(n)) = O(g_1(n)+g_2(n))$

$O(g_1(n))xO(g_2(n)) = O(g_1(n)xg_2(n))$

$O(g_1(n)+g_2(n)+g_3(n)) =$
$O(max(g_1(n),g_2(n),g_3(n))$

$O(max(g_1(n),g_2(n),g_3(n)) =$
$max(O(g_1(n)),O(g_2(n)),O(g_3(n)))$

Properties of the Big-Oh

Allows us to **ignore** the **constant factors**

$$O(k(g(n)) = O(g(n)) \text{ for a constant } k$$

For example:

$t(n) = n$ is $O(n)$

$t(n) = 2n$ is also $O(n)$

$t(n) = 5n$ is also $O(n)$

$t(n) = 8n$ is also $O(n)$

Hint: What is the elephant? What are mouses?

Properties of the Big-Oh

Allows us to **ignore** the **lower-order terms**

$$O(g_1(n)+g_2(n)+g_3(n)) = O(\max(g_1(n),g_2(n),g_3(n)))$$
$$= \max(O(g_1(n)),O(g_2(n)),O(g_3(n)))$$

For example:

$t(n) = n^4 + 2n^3 + 1$ is $O(n^4)$

$t(n) = 2n^3 + 3n + 4$ is $O(n^3)$

$t(n) = n^2 + 2\log n$ is $O(n^2)$

$t(n) = 2n + 3\log n$ is $O(n)$

$t(n) = 4\log n + 4$ is $O(\log n)$

Hint: What is the elephant? What are mouses?

Quiz

What is the time efficiency of the following code?

Calculate the sum from 1 to n

```
sum = 0
loop i from 1 to n
  loop j from 1 to i
    sum = sum + 1
```

$f(n) = n^2/2 + n/2$, which is $O(n^2)$

Properties of the Big-Oh

Allows us to **ignore** the **less complex segments** among the **consecutive ones** in a program

```
for (int i = 0; i < n; i++)
    arr[i] = i;

for (int i = 0; i < n; i++)
    for (int j = 0; j < n; j++)
        arr[j] += arr[j] + i + j;
```

Segment S1

Segment S2

Complexity of S1 is $O(n)$
Complexity of S2 is $O(n^2)$
Complexity of the program is $\max(O(n), O(n^2))$, which is $O(n^2)$

Properties of the Big-Oh

Allows us to **ignore** the **less complex branches** in an *`if/else`* statement

```
if (condition)
    S1
else
    S2
```

```
Complexity of the program is:
O(condition) + max(O(S1), O(S2))
```

Quiz

Self study

What is the time efficiency of the following code?

```
if (n % 2 == 0)
   for (i = 1; i <= n; i++)
      for (j = 1; j <= 10; j++)
         sum = sum + j;
   for (i = 1; i <= n/2; i++)
      sum = sum + i;

else
   for (i = 1; i <= n; i++)
      for (j = 1; j <= n; j++)
         sum = sum + j;
```

$O(n^2)$

Question

Self study

An algorithm takes 0.5 ms for input size 100. How long will it take for input size 500 if the running time is the following (assume low-order terms are negligible):

a) $O(N)$

b) $O(N \log N)$

c) $O(N^2)$

d) $O(N^3)$

7 most common functions

The following table lists 7 most growth-rate functions and the estimated time required to process **one million** items

Growth-Rate function	Time
`O(1)`	Extremely fast (even with more items)
`O(log n)`	0.0000199 seconds
`O(n)`	1 second
`O(n log n)`	19.9 seconds
$O(n^2)$	11.6 days
$O(n^3)$	31,709.8 years
$O(2^n)$	$10^{301,016}$ years

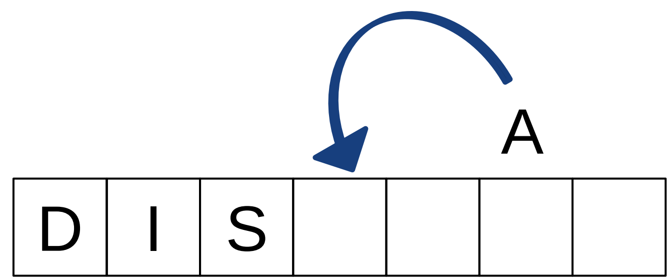
Example

Self study

What is the complexity of implement ADT List method *Add()* using an array?

```
public void Add(string newElement) {
  arr[numElements] = newElement;
  numElements++;

  EnsureCapacity();
}
private void EnsureCapacity() {
  int capacity = arr.Length - 1;
  if (numElements >= capacity) {
    // Replace with a new bigger array
    int newCapacity = capacity * 2;
    string[] newArr = new string[newCapacity];
    arr.CopyTo(newArr, 0);
    arr = newArr;
  }
}
```



Most of the time O(1)
Rarely worst-case O(n)

Example

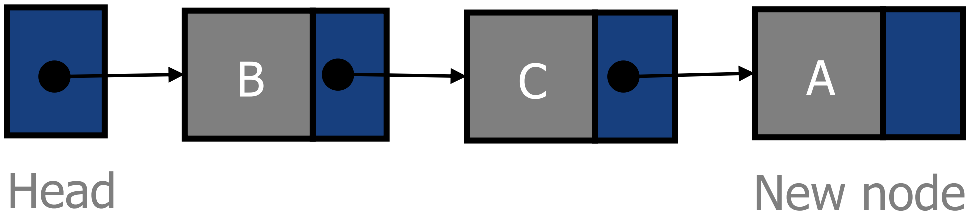
Self study

What is the complexity of implement ADT List method *Add()* using a Linked List?

```
public void Add(string element)
{
  Node newNode = new Node(element);

  if (numElements == 0) {
    Head = newNode;
  }
  else
  {
    Node lastNode = GetLastNode();
    lastNode.Next = newNode;
  }

  numElements++;
}
```




```
Most of the time O(n)
Rarely best-case O(1)
```

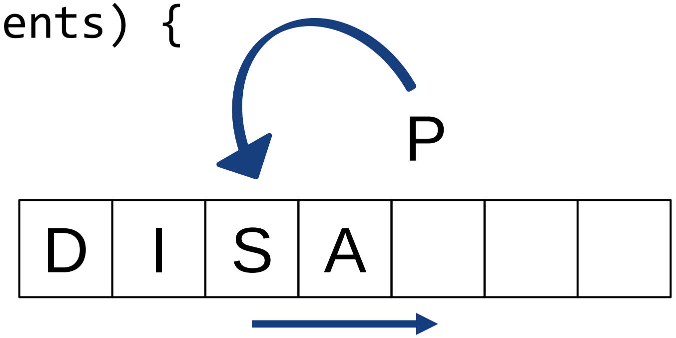
Quiz

Self study

What is the complexity of implement List method *`Insert()`* to a given position using an array?

```
public void Insert(int index, string newElement) {
  // Allow inserting to the end
  if (index >= 0 && index <= numElements) {
    if (index < numElements)
      MakeRoom(index);
    arr[index] = newElement;
    numElements++;
    EnsureCapacity();
  } // else Invalid index
}

// Shift entries toward the end of the array
private void MakeRoom(int index) {
  for (int i = numElements;
              i >= index; i--)
    arr[i + 1] = arr[i];
}
```




```
Most of the time:
Insert to pos other than end O(n)
Insert to the end O(1)
```

ADT List time efficiency

Self study

Operation	Using Array	Using Linked List	Using Linked List with Tail *
Add(e)	O(1)	O(n)	O(1)
Insert(pos, e)	O(n), O(n), O(1)	O(1), O(n)	O(1), O(n), O(1)
RemoveAt(pos)	O(n), O(n), O(1)	O(1), O(n)	O(1), O(n), O(1)
Replace(pos, e)	O(1)	O(1), O(n)	O(n)
GetAt(pos)	O(1)	O(1), O(n)	O(1), O(n), O(1)
Contains(e)	O(n)	O(n)	O(n)

O(n), O(n), O(1)

At the beginning of the list
At the other positions
At the end of the list

* Advanced students may read more about Linked List with Tail in sections 14.20-14.24 [Carrano 2016]

Question

Self study

When implementing apps **adding / removing** to / from the **beginning** of the list **frequently**, which List implementation should we choose?

In practice, at the beginning we just **pick an implementation**, e.g., Array List, and **optimize later** when really needed

Image by Clker-Free-Vector-Images from Pixabay

Readings

- Data structures and abstractions with Java, 4ed – Chapter 4, The efficiency of Algorithms, *Frank M.Carrano and Timothy M. Henry*