# Algoritmos y Estructuras de Datos

## Introducción a las Estructuras de Datos y Algoritmos

Universidad Peruana de Ciencias Aplicadas

2020

# Outline

# Outline I

# Computerphile



**Dr. James Clewett**

The Art of Abstraction https://www.youtube.com/watch?v=p7nGcY73epw

# Definición



## Abstracción

Es el proceso de quitar características de algo para reducir su complejidad.

## Tipo de dato

Un tipo particular de ítem, definido por los valores que éste puede tomar, el lenguaje de programación usado o las operaciones que se realizan sobre éste.

# Tipos de abstracción



## Lenguajes de programación

Assembler, Lenguajes orientados a objetos, lenguajes lógicos, funcionales (lambdas y funciones de primera clase, por ejemplo, se tratarán posteriormente en el curso), etc.

## Control de flujo

Programación estructurada, estructuras de control, etc.

## De datos

Tipos de datos y tipos de datos abstractos como Registros, Clases, Interfaces, etc.

# Outline I

# Eficiencia



- Una solución es eficiente si resuelve el problema dentro de sus limitaciones de recursos.
  - Espacio (Estructura de datos)
  - Tiempo (Algoritmos)
- El costo de una solución es la cantidad de recursos que esta consume en tiempo y espacio.
- Existen tres tipos generales de análisis de tiempo:
  - Mejor caso,
  - caso promedio y
  - peor caso.
- Existen otros tipos de análisis pero no nos concentraremos en ellos.

# Selección de una estructura de datos



- Analizar el tipo de input posible en el problema.
- Analizar el problema para determinar las limitaciones de recursos a los que la solución debe adaptarse.
- Determinar las operaciones básicas que deben ser soportadas. Evaluar las limitaciones de recursos para cada una de estas operaciones.
- Seleccionar la estructura de datos que mejor se adecúe a estos requerimientos.

Nota: Generalmente buscamos la solución más simple.

# Debemos preguntarnos...



- Las inserciones se harán siempre en el primer registro, al final o en base a algún criterio?
- Se puede eliminar la información?
- Se procesan los datos en algún orden específico o se accede aleatoriamente?

Estas interrogantes nos ayudan a eliminar algunas posibilidades.

# Estructuras de datos



- Cada Estructura de datos tiene costos y beneficios.
- Raramente una estructura de datos es mejor que otra en todas las circunstancias.
- Una estructura de datos requiere:
  - Espacio para cada registro,
  - tiempo para ejecutar cada operación básica y
  - esfuerzo de programación.

# ...Estructuras de datos



- Cada problema tiene limitaciones de espacio y tiempo.
- Solo después de un cuidadoso análisis de las características del problema podremos decidir en la mejor estructura de datos para la solución.
- Ejemplo: En un banco
  - Abrir una cuento - minutos.
  - Transacciones - segundos.
  - Cierre de cuentas - horas (trasnoche).

# Eficiencia de algoritmos



- Siempre existen diversas soluciones a un mismo problema, como escoger entre ellos?
- Suelen existir dos objetivos, muchas veces en conflicto, al construir el programa:
  - Diseñar un algoritmo que sea fácil de entender, codificar y mantener.
  - Diseñar un algoritmo que haga uso eficiente de los recursos del computador.
- El objetivo 1 es preocupación del ingeniero de software.
- El objetivo 2 es preocupación de las ciencias de computación, en el análisis de estructura de datos y algoritmos. Cómo medimos el costo de un algoritmo?

# Cómo medir la eficiencia



- Análisis asintótico de algoritmos.
  - Recursos críticos.
  - Factores que afectan el tiempo de ejecución.
  - Para la mayoría de algoritmos, el tiempo de ejecución depende de los parámetros de entrada (size).

# Veamos un ejemplo

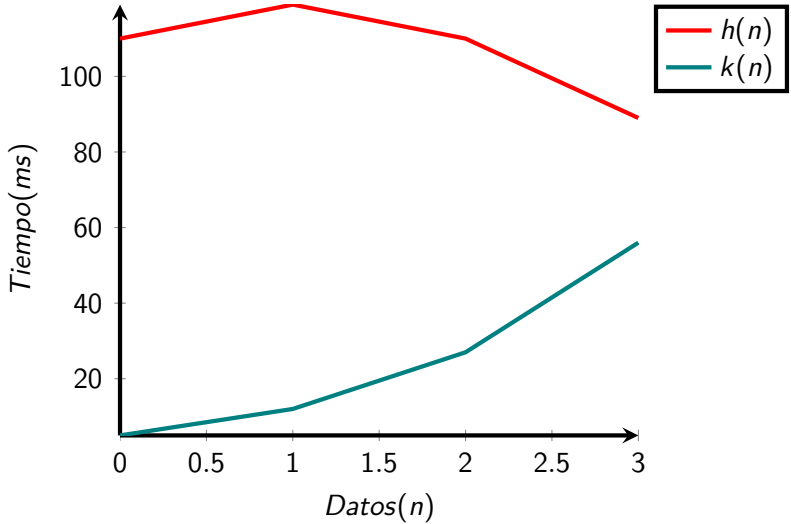


- Después de realizar el análisis asintótico de dos algoritmos, h y k, obtuvimos las siguientes fórmulas de tiempo en base al número de datos n:
  - $h(n) = n^3 - 12n^2 + 20n + 110$
  - $k(n) = n^3 + n^2 + 5n + 5$
- ¿Cuál algoritmo es mejor?

| n | h(n) | k(n) |
|---|---|---|
| 0 | 110 | 5 |
| 1 | 119 | 12 |

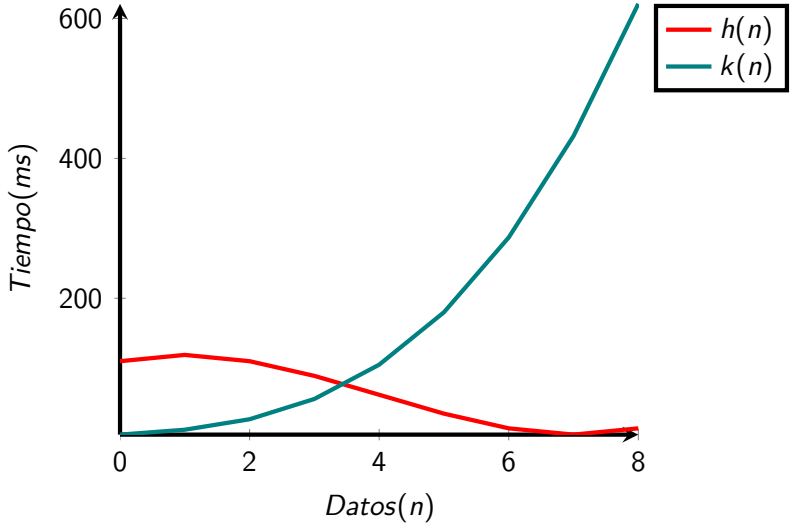
# De 0 a 3 datos





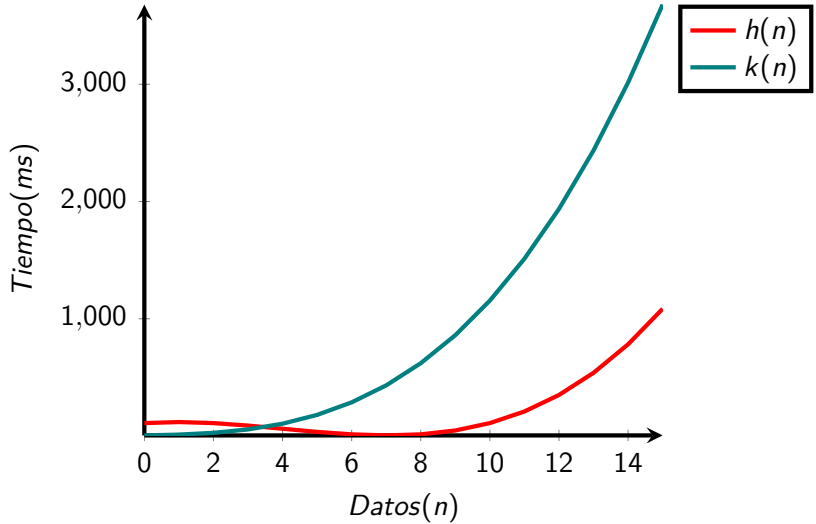
# De 0 a 8 datos





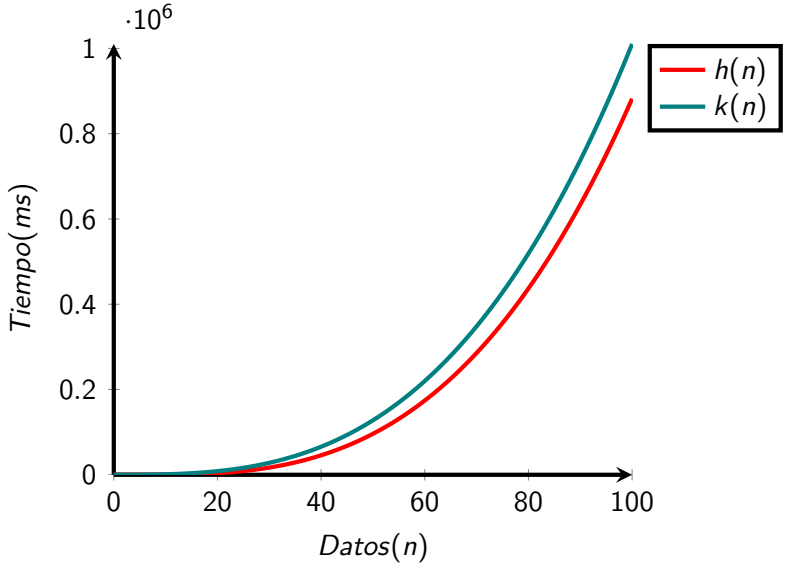
# De 0 a 15 datos





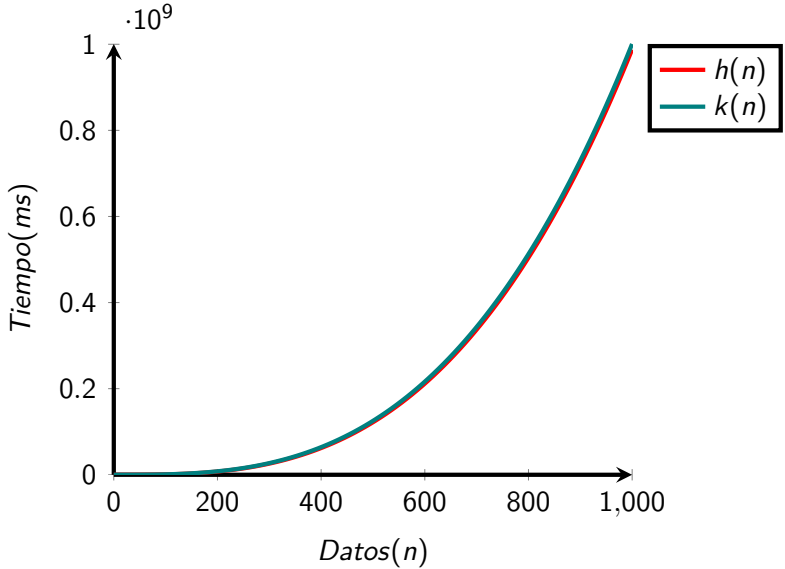
# De 0 a 100 datos





# De 0 a 1000 datos





# Notación Big O



- El análisis detallado de tiempo no nos permite determinar la diferencia de tiempos entre los algoritmos de forma inmediata.
- Por esta razón surge la notación Big O.
  - Big O representa el Peor caso.
- Nos permite evaluar los términos de upper bounds, es decir proporcional a lo máximo que podría demorar un algoritmo.

# Cómo realizar el cálculo de Big O



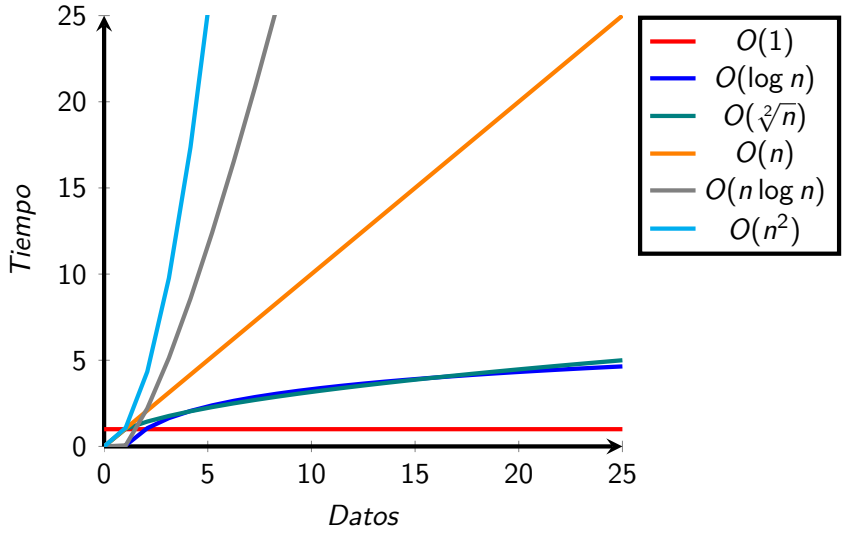
- Borrar términos de orden menor.
- Borrar las constantes.

## Ejemplos

$$
\begin{aligned}
h(n) = n^3 - 12n^2 + 20n + 110 &\longrightarrow h(n) = O(n^3) \\
k(n) = n^3 + n^2 + 5n + 5 &\longrightarrow k(n) = O(n^3) \\
f(n) = 3n^3 + 90n^2 - 5n + 6046 &\longrightarrow f(n) = O(n^3)
\end{aligned}
$$

# Orden de algoritmos





# Análisis de algoritmos



- Asumir que la cantidad de datos a procesar es un número grande.
- ¿Qué contar?
  - Operaciones de comparación
  - Operaciones aritméticas
  - Copiado de datos
  - Asignaciones
- Tener conocimiento matemático

# ...Análisis de algoritmos



- Ejemplo: hallar el promedio de todos los números en el arreglo A.

```
int sum = 0;
int n = length(A);
for (int i=0; i < n; i++)
  sum = sum + A[i];
double prom = sum / n;
```

  - Tiempo detallado: $4 + 6n + 2 = 6n + 6$
  - Tiempo asintótico: $O(n)$

# Reglas para calcular el tiempo



- Regla 1: sentencias secuenciales
- Cada operación y asignación tiene peso 1, funciones tienen su propio peso. Luego sumar todo.

```
i = x; // --> 1
a = b + 2 * 3; // --> 3, 2 operaciones y una asignación
v[i] = v[0]; // --> 3, 2 accesos a arreglos y una asignación
b = binSearch(v, 10); // --> 1 + log 10, binSearch(v, n): log(n)
cout << b; // --> 1, 1 por cada operador <<
```

  - Tiempo detallado: $9 + \log 10$
  - Tiempo asintótico: $1 \longrightarrow$ constante!

# Reglas para calcular el tiempo



- Regla 2: estructuras repetitivas
- Identificar el número de iteraciones y multiplicar por la expresión interior

```
for (int i = 0; i < n; ++i) { // --> 1 + n(1 + INTERNA + 2)
  sum = sum + A[i]; // --> 3
  cout << sum; // --> 1
}
```

  - La inicialización del for cuenta por 1 (asignación)
  - el número de iteraciones del for (n en este caso) multiplica al valor obtenido en la parte interna del for y se le agrega la expresión correspondiente a la condición de finalización (1 en nuestro caso) y la expresión de incremento (2 en nuestro caso por el operador $++$)
  - Tiempo detallado: $1 + n(1 + 3 + 1 + 2) = 1 + 7n$
  - Tiempo asintótico: $O(n) \longrightarrow$ Lineal

# Reglas para calcular el tiempo



- Regla 3: repetitivas anidadas

```
for (int i = 0; i < n; ++i) { // --> 1 + n(1 + __ + 2)
  for (int k = 0; k < n/2; k += 2) { // --> 1 + (n / 4)(2 + __ + 2)
    cout << i * k; // --> 2
  }
}
```

  - El for interno solo realiza $\frac{n}{4}$ iteraciones ya que el límite superior es $\frac{n}{2}$ y k incrementa de 2 en 2.
  - Tiempo detallado: $1 + n(1 + 1 + \frac{n}{4}(2 + 2 + 2) + 2) = \frac{3n^2}{2} + 4n + 1$
  - Tiempo asintótico: $O(n^2) \longrightarrow$ Cuadrático

# Reglas para calcular el tiempo



- Regla 4: repetitivas consecutivas

```
for (int i=0; i < n; i++) // --> 1 + n(1 + __ + 2)
  cout << i; // --> 1
for (int i = 0; i < 10000; ++i) { // --> 1 + 10000(1 + __ + 2)
  for (int k = 1; k < n; k *= 2) { // --> 1 + (log n)(1 + __ + 2)
    cout << i * k; // --> 2
  }
}
```

  - Tiempo detallado:
    $1+n(1+1+2)+1+10000(1+1+(\log n)(1+2+2)+2) = 4n+50000(\log n)+40002$
  - Tiempo asintótico: $O(n)$

# Reglas para calcular el tiempo



- Regla 5: if ... else
- Escoger el máximo de la expresión verdadera y de la falsa

```
if (a > b) {// --> 1 + max(INTERNA IF, INTERNA ELSE)
  a = binSearch(v, b); // --> 1 + log n
} else {
  a = v[b]; // --> 2
}
```

  - Tiempo de expresión dentro del IF más el máximo valor entre la expresión obtenida del bloque IF y el bloque ELSE.
  - Tiempo detallado: $1 + \log n$
  - Tiempo asintótico: $O(\log n)$

# Tipos de análisis



- Mejor caso (best case): si es que el input causa que tome el menor tiempo. Big $\Omega$
- Peor caso (worst case): da una idea del máximo tiempo que un algoritmo puede tomar. Big O
- Caso promedio (average case): hay que tomar muchos detalles y aplicar algo de estadística para encontrar el tiempo esperado sobre todos los casos posibles del input. Big $\Theta$

# ¿Por qué medir la eficiencia de un algoritmo?



- Nos ayuda a entender la escalabilidad.
- Nos permite discernir entre lo que es posible y lo imposible.
- Es un lenguaje que nos permite predecir el comportamiento de un programa.
- La eficiencia es la "moneda" en la computación.
- Buscar la rapidez de un programa es divertido! :D
- Debemos convertir esto en un hábito.

# Outline I

# Programación genérica

## Definición

Consiste en elaborar algoritmos en los que uno o más tipos de datos son especificados posteriormente.

## Ventajas

En la mayoría de los casos permite disminuir la cantidad de código e incrementa la reusabilidad de los algoritmos y tipos de datos abstractos.
Es una característica presente en los principales lenguajes de programación tipados.

## C++: Templates

C++ implementa la programación genérica con los Templates (plantillas en español), las cuales se aplican a funciones y TDA definidos por el usuario.
Los templates son construcciones usadas en **tiempo de compilación** para construir familias de funciones, clases, estructuras, etc.

# Considere las siguientes funciones



```
int sumaInt(int a, int b) {return a + b;}
float sumaFloat(float a, float b) {return a + b;}
double sumaDouble(double a, double b) {return a + b;}
int main() {
  double x = 10.5, y = 20.75;
  printf("Entero: %d\n", sumaInt(x, y));
  printf(" Float: %f\n", sumaFloat(x, y));
  printf("Double: %lf\n", sumaDouble(x, y));
  return 0;
}
```

Salida:

```
Entero: 30
 Float: 31.250000
Double: 31.250000
```

# Simplificando un poco

```
double suma(double a, double b) {
  return a + b;
}
int main() {
  double x = 10.5, y = 20.75;
  printf("Entero: %d\n", suma(x, y));
  printf(" Float: %f\n", suma(x, y));
  printf("Double: %lf\n", suma(x, y));
  return 0;
}
```

Salida: podemos ver que el resultado entero no se muestra correctamente. ¿Por qué?

```
Entero: 0
 Float: 31.250000
Double: 31.250000
```

# Solución con templates



```
template<typename T>
T suma(T a, T b) {
  return a + b;
}
int main() {
  double x = 10.5, y = 20.75;
  printf("Entero: %d\n", suma<int>(x, y));
  printf(" Float: %f\n", suma<float>(x, y));
  printf("Double: %lf\n", suma<double>(x, y));
  return 0;
}
```

# La sintaxis



Los templates se definen anteponiendo la siguiente declaración:

```
template<typename T> o template<class T>
```

A continuación viene la declaración de la función, clase o estructura, por ejemplo:

```
template<typename T>
T sumar(T a, T b) {
  return a + b
}
```

Para llamar a esta función usamos la siguiente sintaxis:

```
cout << "La suma es: " << sumar<int>(x, y);
```

# Clases con templates



```
template <typename T>
class CCuadrado {
  T lado;
public:
  CCuadrado(T lado);
  ~CCuadrado();
  T Area();
};
```

```
template <typename T>
CCuadrado<T>::CCuadrado(T lado) {
  this->lado = lado;
}
template <typename T>
CCuadrado<T>::~CCuadrado(){}
template <typename T>
T CCuadrado<T>::Area() {
  return lado * lado;
}
```

Los templates son expandidos a funciones con tipos explícitos en tiempo de compilación cuando son usadas con un tipo explícito. Por tal razón, para evitar problemas en tiempo de enlace (linking), se recomienda implementar los métodos de clases template en el mismo archivo .h donde se declara la clase template

# Outline I

# Referencias



Thomas H. Cormen, Charles E. Leirserson, Ronald L. Rivest, Clifford Stein. **Introduction to Algorithms**. Third edition, The MIT Press, Cambridge, Massachusetts, 2009.

Bjarne Stroustrup. **Programming: Principles and practice using C++**. Addison-Wesley, Upper Saddle River, NJ, Boston, 2009. Capítulo 19 sección 3, p. 656.

Standard C++ Foundation **Templates, C++ FAQ**. https://isocpp.org/wiki/faq/templates