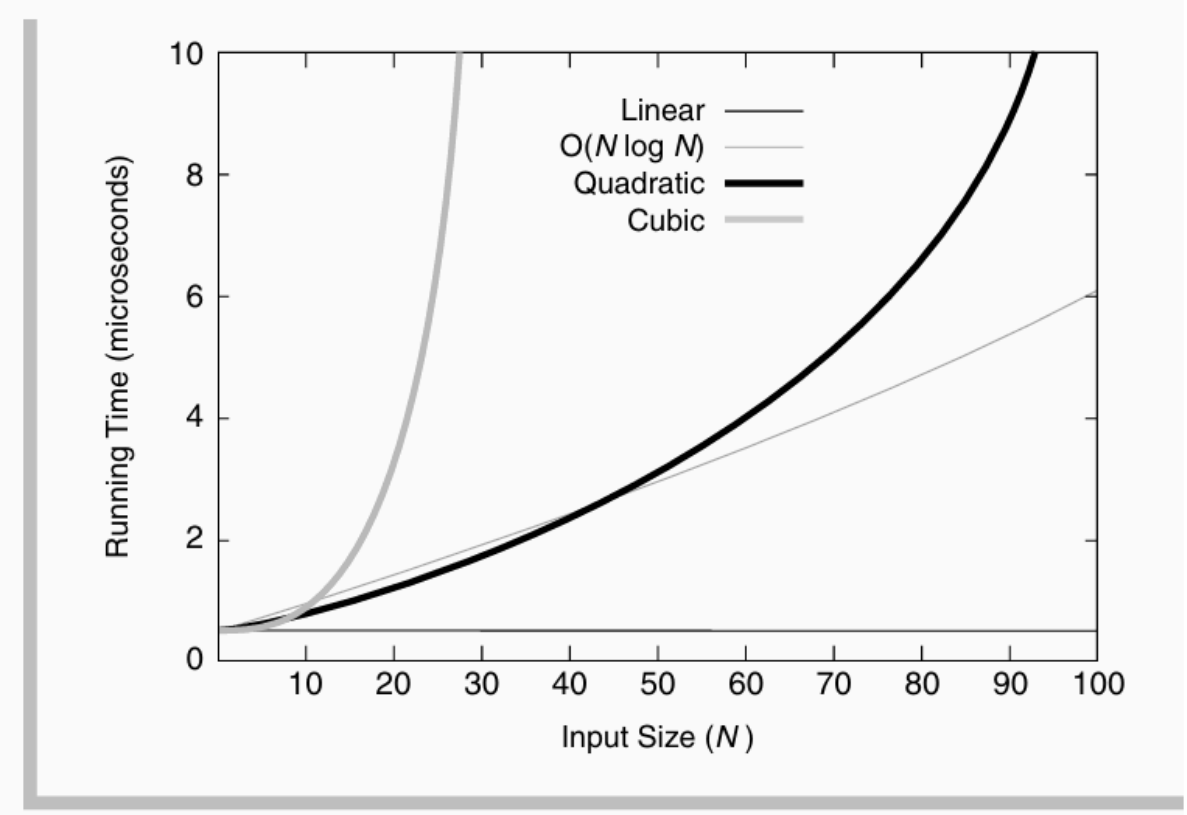
iRAT #002 UT3

Data structures and problem solving in Java - Mark Allen Weiss

- ✅ ~~5. Algorithm analysis~~
- ✅ ~~5.1 What is algorithm analysis?~~
- ✅ ~~5.2 Examples of algorithm running times~~
- ✅ ~~5.4 General Big-Oh rules~~
- ✅ ~~5.5 Logarithms~~

Algorithm analysis & What is algorithm analysis

- Running time of a problem is most strongly connected with the amount of input it processes
- If not, it can depend on a lower note on the speed of the host machine or the quality of the compiler; this, as we'll see later on, is the constant c.
- A mostly correct algorithm is still not useful if it requires too many resources such as memory
- Algorithm analysis: study of the amount of time and resources an algorithm requires to run
- A direct correlation between time and input is the signature of linear algorithms
- Underneath are four graphs showcasing running times of various algorithms with relatively small amounts of input



- Big-Oh notation is used to compare growth rates in running times using the most dominant term in a function
- For instance: running time in a given quadratic algorithm is specified as $O(N^2)$ (pronounced order en-squared)
- For small amounts of inputs, making comparisons between functions is difficult because leading constants become very significant
- Also, when the amount of input is so small it becomes insignificant, a simpler algorithm is the way to go, not necessarily the one with the lowest running time
- It is of note to say that even the cleverest programming tricks cannot make a bad algorithm efficient or make it competition for a linear algorithm, even when this one is badly written
- Always optimize the algorithm
- Following, a table of functions in order of increasing growth rate

Function	Name
c	Constant
$\log N$	Logarithmic
$\log^2 N$	Log-squared
N	Linear
$N \log N$	$N \log N$
N^2	Quadratic
N^3	Cubic
2^N	Exponential

General Big-Oh rules

- As we've said before, Big-Oh notation is used to examine the most dominant term in a given function
- This notation gives us a worst-case-scenario estimate bound to guide us into a more efficient algorithm

Definition

(Big-Oh) $T(N)$ is $O(F(N))$ if there are positive constants c and N_0 such that $T(N) \leq cF(N)$ when $N \geq N_0$.

Meaning:

There is a point N_0 such that for all values of N that are past this point, $T(N)$ (running time) is bounded by some multiple of $F(N)$ (whichever function).

- $F(N)$ only means N doesn't have to be linear necessarily, it can also be a function.
- There is also Big-Omega (greater than or equal to), Big-Theta (greater than) and Little-Oh (lower than). Big-Oh is similar to less than or equal to

Mathematical Expression	Relative Rates of Growth
$T(N) = O(F(N))$	Growth of $T(N)$ is $\leq$ growth of $F(N)$.
$T(N) = \Omega(F(N))$	Growth of $T(N)$ is $\geq$ growth of $F(N)$.
$T(N) = \Theta(F(N))$	Growth of $T(N)$ is $=$ growth of $F(N)$.
$T(N) = o(F(N))$	Growth of $T(N)$ is $<$ growth of $F(N)$.

- Most basic rule:

> Running time of a loop is at most the running time of the statements inside the loop (plus tests) times the number of iterations.

- We must also keep in mind that worst-case bound have no influence in average-bounds, and having a better worst-case bound doesn't mean the algorithm will be the most efficient one regardless of input amount
- In case of an increase in input amount, we can make these estimated calculations to get the new running time:
- In general, if the amount of input increases by a factor of f, the cubic algorithm's running time increases by a factor of f^3; same applies for quadratic functions, in which running time increases by a factor of f^2

The logarithm

- In computer science, the logarithm's base defaults to 2
- Logarithms grow slowly, consequently, its performance is much closer to that of a linear algorithm and thus much more desired
- Rule of thumb: each time we duplicate the input, the runtime only grows by 1 unit.

Pensando la computación como un científico - A. Downey

Secciones:

- [ ] 11 Arreglos
- [x] ~~14.1 Referencias en objetos~~
- [x] ~~14.2 La clase Nodo~~
- [x] ~~14.3 Listas como colecciones~~
- [x] ~~14.4~~
- [x] ~~15.1~~
- [x] ~~15.2~~
- [x] ~~15.3~~
- [x] ~~15.11~~
- [x] ~~15.12~~
- [x] ~~15.13~~
- [x] ~~16.1~~
- [x] ~~16.3~~
- [x] ~~16.4~~

Arreglos

(…)

Listas enlazadas

Referencias a objetos

Las variables de instancia de una clase pueden ser tanto arreglos como objetos, no solo datos de tipo primitivo. Incluso, un objeto puede contener dentro de sí una referencia a un objeto del mismo tipo.

Las listas enlazadas toman provecho de esta característica.

Las listas enlazadas están compuestas por nodos, donde cada uno de estos tiene una referencia al siguiente nodo después de sí. Tienen también, una unidad de datos que denominamos carga.

La clase *Nodo*

```
public class Nodo {
   int carga;
   Nodo prox;

   public nodo(int carga, Nodo prox) {
      this.carga = carga;
      this.Nodo = prox;
   }
}

public static void main(String[] args) {
   Nodo nodo = new Nodo(1, null) // Lista de 1 nodo
   ...
   nodo1.prox = nodo2; // Crear listas con más de 1 nodo y enlazarlos
}
```

Listas como colecciones

Las listas son formas de ensamblar varios objetos en una sola entidad, generalmente llamadas colecciones.

En las listas enlazadas, el primer nodo sirve como referencia para toda la lista entera. Si quisiéramos pasar la lista como un parámetro a un método, solo bastaría con pasar el primer nodo de la lista en cuestión.

Podemos usar una variable nodo como iterador para recorrer las listas enlazadas, igualando esta variable al prox de cada nodo sucesivamente.

Listas y recursión

…

Pilas

Un tipo de dato abstracto (TAD) que especifica un conjunto de operaciones y la semántica de estas, pero no su implementación.

- Simplifica el trabajo de especificar un algoritmo ya que no tenemos que pensar en la implementación de sus operaciones
- Escribir un algoritmo que pueda ser usado para cualquier implementación de un TAD

- TAD reconocidos como el TAD Pila tienen implementaciones estandarizadas en bibliotecas y son reutilizadas por muchos programadores
- Las operaciones en los TAD ofrecen un lenguaje de alto nivel para especificar algoritmos

Podemos clasificar el código que utiliza el TAD del que lo implementa, siendo el primero el código cliente y el segundo el código proveedor, ya que provee un conjunto de servicios estándar.

El TAD Pila

Una pila es una colección, lo cual significa que es una estructura de datos que contiene varios elementos.

Un TAD se define por las operaciones que puede realizar. En el caso de una pila, esta puede:

- constructor: crear una nueva pila vacía
- apilar: agregar elementos al final de la pila
- desapilar: eliminar el último elemento agregado a la pila y devolverlo
- `estaVacía`: indicar si la pila está vacía o no

Las pilas son un tipo de estructura LIFO: $Last\ in,\ First\ out.$

El Objeto $Stack$ de Java

El Objeto Stack es un objeto pre-incorporado en la clase $java.util$ de Java que implementa el TAD Pila.

Estas son las operaciones del Stack:

- Constructor: `Stack pila = new Stack();`
- Apilar: push
- Desapilar: pop
- `estaVacía`: `.isEmpty`

También es un tipo de estructura genérico, porque podemos agregar datos de cualquier tipo dentro de ella. Sin embargo, en la implementación Stack de Java, solo podemos agregar objetos, no datos de tipo primitivo. Si quisiéramos agregar un valor de tipo primitivo a una pila, debemos hacerlo a través de un $wrapper$ que haga que este valor se comporte como un objeto.

El retorno del método pop es de tipo $Object$, ya que la pila no sabe qué tipo de dato estamos apilando. Debemos de castear los elementos que guardemos en la pila manualmente al

sacarlos de esta. Si casteamos de manera incorrecta recibiremos el mensaje de error `ClassCastException`.

Este ejemplo agrega elementos de tipo Nodo a una pila:

```
Stack pila = new Stack();
pila.push(lista.cabeza)
Object obj = pila.pop();
Nodo nodo = (Nodo) obj;

// O, alternativamente e iterando sobre todos los elementos de la lista
for (Nodo nodo = lista.cabeza; nodo != null; nodo = nodo.prox) {
   pila.push(nodo);
   }
Object obj = pila.pop();
Nodo nodo = (Nodo) obj;
```

Implementando $TAD's$

Uno de los principales objetivos de los $TAD's$ es separar los intereses del cliente de los del proveedor.

Si nos encontramos del lado del proveedor, debemos generar código cliente para poder testear nuestras implementaciones.

Podemos cambiar la implementación sin cambiar el código cliente.

Implementación del TAD Pila usando Arreglos

Las variables de instancia de esta implementación serán arreglos de $Objects$ que contendrán elementos de la pila y un índice entero que llevará la cuenta del siguiente espacio vacío.

Inicialmente, el arreglo está vacío y el índice es 0.

Operaciones fundamentales de la Pila:

- `push`: copiar una referencia del Objeto a agregar y sumarle 1 al índice
- pop: restarle 1 al índice y copiar el elemento afuera

Es importante señalar que el tamaño del arreglo no es igual a la cantidad de Objetos en la pila.

```
public class Stack {
   Object[] arreglo;
   int indice;

   public Stack() {
      this.arreglo = new Object[128];
      this.indice = 0;
   }

   public boolean isEmpty() {
      return indice == 0;
   }

   public void push(Object elem) {
      arreglo[indice] = elem;
      indice++;
   }

   public Object pop() {
      indice--;
      return arreglo[indice];
   }
```

Redimensionando un Arreglo

Si bien no podemos cambiar el tamaño de un Arreglo una vez este ha sido inicializado, podemos verificar si este está lleno cada vez que se quiera agregar un nuevo elemento a la pila. De esta forma, si encontramos que efectivamente, la pila se ha llenado, podemos disparar un nuevo método que genere el mismo Arreglo, pero con el doble de capacidad.

Los métodos Redimensionar y Lleno pueden ser privados, y solo accedidos dentro de la clase Pila, de esta forma reforzando las barreras entre código cliente y proveedor.

Una precondición del método Redimensionar es que el Arreglo esté lleno.

La postcondición del método es que el nuevo Arregla sea el doble de grande que el anterior, o sea, `Array.lenght > index` .

```
public void push(Object item) {
   if (lleno()) redimensionar ();
```

```
    arreglo[indice] = item;
    indice++;
}

private boolean lleno() {
    return indice == arrgelo.lenght;
}

private void redimensionar() {
    Object[] nuevoArreglo = new Object[arreglo.lenght * 2];

    for (int = 0; i < arreglo.lenght; i++) {
        nuevoArreglo[i] = arreglo[i]
    }
    arreglo = nuevoArreglo;
}
```

TAD Cola

Operaciones:

- constructor: inicializa una nueva cola vacía
- agregar: agrega un elemento
- quitar: elimina el primer elemento agregado y lo devuelve
- `estaVacía`: verifica si la cola está vacía

Una implementación genérica de una Cola se puede hacer con la clase pre-incorporada de Java de `LinkedList`. Esta contiene una única variable de instancia, que es la lista que la implementa (`LinkedList lista = new LinkedList();`).

Cola enlazada

Para poder implementar la cola de forma que sus métodos sean consistentes, debemos tener una referencia al primer y al último lugar de la cola.

De esta forma tenemos dos índices: el primer cliente en la cola y el próximo cliente, en un arreglo de $Objects$ similar a la implementación del TAD Pila con Arreglos. Esto es porque el primer cliente no tiene por qué estar al inicio de la fila necesariamente, dependiendo de la disciplina de cola que usemos.

Es importante señalar que los índices no son referencias sino enteros que representan las posiciones dentro de la cola.

Cualquier método con la palabra $static$ es un método de clase, sino es un método de objeto.

Es válido mandar $null$ como argumento de un método de clase, pero no es válido llamar un método de instancia sobre un objeto nulo.

Podemos utilizar un buffer circular para poder utilizar el siguiente espacio libre.

```
prox++;
if (prox == arreglo.length) prox = 0;
```

$Wrappers\ y\ Helpers.$

$Veneer$ (enchapado).

- Precondición: afirmación sobre un objeto que debe cumplirse para asegurar que un método funcione correctamente.
- Invariante: afirmación sobre un objeto que debe ser cierta en todo momento (exceptuando los momentos en los que se modifique dicho objeto).
- Postcondición: algo que debe ser cierto una vez un método ha sido ejecutado, siempre y cuando se cumplan sus precondiciones.
- Tipo de dato abstracto (TAD): tipo de dato (usualmente una colección de objetos) que puede ser definida por una serie de operaciones o métodos pero que puede ser implementada de varias formas.

`var + " "` - converts a primitive type into a String type in an easy and compact way.

Lecturas básicas de la Cátedra

Ordenes

Tiempo de ejecución de un programa

Al elegir qué algoritmo usar, debemos seguir los siguientes objetivos:

- Algoritmo fácil de entender, codificar y depurar
- Uso eficiente de los recursos y ejecución veloz

La mayoría de las veces, no podemos conseguir todos estos objetivos dependiendo de la cantidad de veces en la que se va a ejecutar el programa.

Medición del tiempo de ejecución de un programa

El tiempo de ejecución de un programa depende de factores como los siguientes

- Input
- Calidad del código generado por el compilador
- La naturaleza y rapidez de las instrucciones usadas en el programa
- Complejidad de tiempo del algoritmo base del programa

Se denomina $T(n)$ el tiempo de ejecución de un programa con un input de tamaño .

Cuando el tiempo de ejecución es función de una entrada específica, entonces se define $T(n)$ como el tiempo de ejecución del peor caso.

También podemos calcular el tiempo promedio de ejecución, cosa que es bastante más compleja y casi siempre va en concordancia con el peor caso.

El hecho de que diversos factores (como los mencionados anteriormente) influencien el tiempo de ejecución impide que este pueda ser expresado en unidades de tiempo normales.

Es por esto que se dice que “el algoritmo es proporcional a n^2”.

Notación asintótica - Orden del tiempo de ejecución

Se dice que el tiempo de ejecución de un programa es de orden n cuando existen dos constantes enteras positivas c y n_0, tales que

$$\forall\ n \bar{>} n_0,\ T(N) \bar{<} c * n$$

Se dice que $T(n)$ es $O(f(n))$ si existen dos constantes enteras positivas c y n_0 tales que,

$$\forall\ n \bar{>} n_0,\ T(N) \bar{<} c * f(n)$$

En ese caso se dice que el programa tiene velocidad de crecimiento $f(n)$.

Si $T(n)$ es $O(f(n))$, $f(n)$ es una cota superior para la velocidad de $T(n)$.

Tiempo de ejecución

Si un programa se va a ejecutar poco, el costo de su desarrollo es dominante.

Si el programa recibe poca cantidad de entradas, el factor constante de la función puede llegar a ser más importante que la velocidad de crecimiento.

Un algoritmo eficiente pero complicado puede no llegar a ser adecuado por temas de mantenimiento.

Algunos algoritmos pierden su eficiencia por requerir demasiado espacio.

En los algoritmos numéricos la precisión y estabilidad son tan importantes como la eficiencia.

Guías básicas para la especificación de algoritmos en seudocódigo

1. Descripción de la solución en lenguaje natural.
2. Especificación del algoritmo en un seudocódigo formal detallado: comenzamos por describir las precondiciones y postcondiciones. Luego se escribe el seudocódigo en sí mismo siguiendo estos lineamientos, que son fundamentales para:
    - completar la comprensión del algoritmo
    - identificar y entender las condiciones de borde
    - diseñar pruebas para verificar la corrección de los resultados esperados
3. Escritura del programa en un lenguaje de programación.

Estándares

Especificar claramente soluciones algorítmicas, independientes del lenguaje de programación.

- Siempre indicar el fin de una estructura de control de flujo
- Siempre se prefieren los Mientras, se desaconseja el uso de los bucles Para
- Usar nombres significativos
- Usar notación camello
- Obviar tipos de datos si son obvios, triviales o no aportan a la comprensión total del algoritmo
- Asignación: ←