# Algoritmos y Estructuras de Datos

Técnicas de Diseño de Algoritmos

1

## Técnicas de diseño de algoritmos

- Libro "The Algorithm design manual", Skienna
  - Capitulo 1, "Introduction to Algorithm design"
    - Introduction
    - 1.1 "Robot Tour Optimization"
    - 1.2 "Selecting the right job"
    - 1.3 "Reasoning about correctness"
- Dividir y Conquistar:
  - Libro: "Estructuras de datos en JAVA", Mark Allen Weiss, 4ta edición
    - Capítulo 7, Sección 7.5
- Programación Dinámica:
  - Libro: "The algorithm desing manual", Skiena
    - Capítulo 8: secciones Introducción, 8.1, 8.1.1, 8.1.2, 8.1.3
  - Libro: "Estructuras de Datos en Java", Mark Allen Weiss, 4ta edición
    - Capítulo 7, Sección 7.6
- Algoritmos Ávidos:
  - "Capítulo": Cap 4. Algoritmos Ávidos
    - 4.1 Introducción
    - 4.2 Problema del cambio



2

## Técnicas de diseño de algoritmos

- A lo largo del curso estudiaremos distintos tipos de algoritmos.
- Estos algoritmos poseen ciertas propiedades que hace que respondan a algún tipo especial de diseño.
- A cada algoritmo que estudiemos, lo clasificaremos en alguna de estas técnicas de diseño.



3

## Técnicas de diseño de algoritmos

UCU

- Las técnicas que presentaremos son:
  - Divide y Vencerás
  - Ávidas
  - Programación dinámica



4

## "Divide y vencerás"

UCU

- Técnica que hace uso de la recursión
- Algoritmos compuestos de 2 partes
  - Divide (la recursion)
  - Vencerás (el procesamiento adicional)
- Tradicionalmente, algoritmos con al menos 2 llamadas recursivas
- Subproblemas disjuntos (no solapados)



5

## "Divide y vencerás"

UCU

- Plantear el problema de la siguiente forma:
  - Descomponer el problema en **k** sub problemas, cada uno con entrada **n/k.** (**k** ≥ 2)
  - Resolver los sub problemas independientemente haciendo la llamada recursiva correspondiente.
  - Combinar las soluciones devueltas por cada llamada recursiva.
  - Devolver la combinación obtenida.



6

## "Divide y vencerás"

UCU

- Son algoritmos muy eficientes.
- Para un problema de tamaño N que se divide en cada llamada en dos sub problemas de igual tamaño y que el procesamiento adicional ("vencerás") tiene un orden lineal (O(N)), el orden del tiempo de ejecución del algoritmo resulta O(N*logN)[1].

1: "Estructuras de Datos en Java", Mark Allen Weiss, 4ta edición, página 318



7

## "Divide y vencerás"

UCU

Ejemplo: clasificación por mezcla

```
mergeSort( int[] a, int[] aux, int izq, int der)
Com
   Si (izq < der) entonces
        int medio = (izq + der) / 2
        mergeSort(a, aux, izq, medio)
        mergeSort(a, aux, medio+1, der)
        mezclar(a, aux, izq, medio+1,izq, der)
    fin si
Fin
```

"Estructuras de Datos en Java", Mark Allen Weiss, 4ta edición, página 353



8

## Algoritmos "ávidos" o "voraces"

UCU

- Construyen la solución en etapas sucesivas, tratando siempre de tomar una decisión "óptima" para cada etapa.
- En general, son fáciles de implementar y producen soluciones eficientes.
- La búsqueda de "óptimos" locales no tiene por qué conducir siempre a un óptimo global. Es necesario probar su correctitud, o encontrar un contra ejemplo para mostrar que no necesariamente arroja el resultado deseado.



9


3

## Algoritmos “ávidos”

UCU

- Dado **C** (entradas) , el algoritmo ávido devuelve en cada iteración un conjunto **S** tal que $S \subseteq C$
- S “prometedor”
- elementos de la técnica:
  - **C** conjunto de candidatos (entradas)
  - Función **solución:** Comprueba, en cada paso, si el subconjunto actual de candidatos elegidos forma una solución.
  - Función de **selección:** Informa cuál es el elemento más prometedor para completar la solución.
  - Función de **factibilidad:** Informa si a partir de un conjunto se puede llegar a una solución.
  - Función **objetivo:** Es aquella que queremos maximizar o minimizar.



10

## Algoritmos “ávidos” Funcionamiento básico

UCU

1. Elegir el ***mejor*** elemento de ***C*** posible (elemento *más prometedor*)
2. Retirarlo del conjunto ***C*** de candidatos
3. Comprobar si produce una solución ***factible***, y si es así, lo incluye en ***S***
4. Si no es factible, descartar
5. Repetir 1-4 hasta alcanzar la función objetivo o agotar los elementos de ***C***



11

## Seudo de algoritmo ávido

UCU

```
AlgoritmoAvido(entrada:CONJUNTO):CONJUNTO;

Variables x:ELEMENTO; solucion: CONJUNTO; encontrada: BOOLEAN;

COMIENZO
 encontrada:=FALSE; crear(solucion);
 MIENTRAS NO EsVacio(entrada) Y (NO encontrada) HACER
          x:=SeleccionarCandidato(entrada);
          SI EsPrometedor(x,solucion) ENTONCES
            Incluir(x,solucion);
            SI EsSolucion(solucion) ENTONCES
              encontrada:=TRUE
            FINSI
          FINSI
   FIN MIENTRAS;
DEVOLVER solucion;
FIN AlgoritmoAvido
```



12


4

## El problema de la optimización del tour del robot (Skienna 1.1)

- Un brazo de robot debe recorrer varios puntos de un circuito en el menor tiempo posible (recorriendo la menor distancia).
- Se elige un punto de inicio y se "visita"
- Mientras haya puntos por visitar
  - Elegir el punto más cercano al último visitado
  - Visitar ese punto, y agregarlo a la secuencia e salida



13

## El problema de la optimización del tour del robot (Skienna 1.1)

- El algoritmo diseñado es simple, fácil de comprender y de implementar.
- Arroja un resultado posible en forma eficiente.
- Lamentablemente, el resultado que arroja no necesariamente es la solución óptima buscada.
- Por más que se cambie el criterio de selección (por ejemplo, en vez de elegir el más próximo al último visitado, ir eligiendo los pares más cercanos), no se obtiene una solución óptima para todas las instancias de entrada.



14

## Problemas de optimización

- Cuando el objetivo es encontrar una solución que maximiza o minimiza una función (por ejemplo el "tour del robot").
- Algunas veces los algoritmos ávidos brindan una solución adecuada, pero no siempre; requieren prueba de que devuelven la mejor solución posible.
- La solución ideal sería computar todas las soluciones posibles y quedarse con la solución óptima.
- Usualmente es muy fácil escribir un algoritmo recursivo que realice esta tarea, pero el problema es que resulta muy ineficiente, con órdenes del tiempo de ejecución excesivamente altos.



15


5

## Trabajo de Aplicación 5 – Ejercicio 1 Cambio de monedas

UCU

Cambio de monedas

- Se desea diseñar un algoritmo que, dado un monto, devuelva la ***menor*** cantidad de monedas correspondiente.
- Supongamos que existen monedas de $11, $5 y $1
  - ¿cuántas monedas de cada tipo se devolvería para $15?
  - ¿cuántas monedas de cada tipo se devolvería para $14?

16

16

## Trabajo de Aplicación 5 – Ejercicio 1 Cambio de monedas

UCU

```
Algoritmo cambiarAvido (de tipo entero importe, de tipo array de enteros
valor[] ) devuelve un array de enteros, la cantidad de monedas de cada valor.
Comienzo
    Cantidades = nuevo array de enteros { 0 , 0 , 0 }
    Desde i = 0 hasta valor.largo – 1 hacer
        Mientras valor [i] < Importe hacer
            Cantidades[i] ++
            Importe = importe – valor [i]
        Fin mientras
    Fin desde
    Devolver cantidades
Fin
```

17

17

## Trabajo de Aplicación 5 – Ejercicio 1 Cambio de monedas

UCU

1. Conjunto candidato: valores diferentes de monedas
2. Conjunto solución: cantidad de monedas de cada valor
3. Función de selección: el tipo de moneda de mayor valor
4. Función de factibilidad: que la moneda no haya sido considerada
5. Función de solución: la suma del valor de la moneda por la cantidad es el valor deseado.
6. Función objetivo: la cantidad de monedas es la mínima posible.

18

18

## Trabajo de Aplicación 5 – Ejercicio 1 Cambio de monedas

**Contra ejemplo:**

**¿Qué pasa si los valores disponibles de monedas son**

**$10, $5, $1 ?**

19

19

## Trabajo de Aplicación 5 – Ejercicio 1 Cambio de monedas

Algoritmo cambiarAvido (de tipo entero importe, de tipo array de enteros valor[] ) *devuelve un array de enteros, la cantidad de monedas de cada valor.* **"n" cantidad de monedas diferentes**

```
Comienzo
    Cantidades = nuevo array de enteros { 0 , 0 , 0 }   O(1)
    Desde i = 0 hasta valor.largo – 1 hacer   O(n)
        Mientras valor [i] < Importe hacer    O(1)
            Cantidades[i] ++    O(1)
            Importe = importe – valor [i] O(1)
        Fin mientras
    Fin desde
    Devolver cantidades
Fin
```

**Por regla de la suma, O(n)**

20

20

## Programación dinámica

- Usualmente la ineficiencia resulta de que el algoritmo recursivo computa los mismos sub-problemas varias veces.
- La solución entonces es almacenar los resultados y reusarlos cuando sea necesario, en vez de volver a calcularlos.
- A la técnica de implementar un algoritmo recursivo en forma eficiente almacenando resultados intermedios se le llama PROGRAMACIÓN DINÁMICA

Algoritmos y Estructuras de Datos 26

26

## Programación dinámica

- Analizar cuidadosamente cuáles son los **subproblemas** superpuestos que se repiten en diferentes llamadas recursivas.
- Identificar si una solución óptima puede ser obtenida mediante la combinación de soluciones óptimas de sus subproblemas (**subestructura óptima**).
- Almacenar con estructuras de datos adecuadas los resultados parciales para que sean usados cada vez que se necesiten ("**memoización"**).
- Son los elementos que permiten diseñar un algoritmo de programación dinámica, usualmente no recursivo, para obtener una solución óptima.



27

## Factorial "memoizado" TA6 – Ej 1

- ¿Es posible realizar esta función por medio de programación dinámica?
- Definir:
  - subproblemas,
  - Subestructura óptima
  - Memoización



28

## Factorial "memoizado"

```
function factorial (n entero no negativo)
COM
SI n = 0
   devolver 1
   SINO, SI n está en tabla
      devolver valor-tabla-para-n
      SINO
        x = factorial(n – 1)
        guardar x en el la posición n de tabla // recordar
    devolver x
    FINSI
FIN
```



29



8

## Programación Dinámica
## Números de Fibonacci

UCU

- Los números de Fibonacci se definen recursivamente:
  - F0 = 0
  - F1 = 1
  - Fi = Fi-1 + Fi-2 para todo i > 1.

***Fibonacci (k)***

**Entrada**: entero k no negativo

**Salida**: el k-esimo número de Fibonacci

```
COM
SI k = 0 o k = 1
      devolver k
  SINO
      devolver Fibonacci (k - 1) + Fibonacci (k - 2)
FINSI
FIN
```





## Análisis del Algoritmo Recursivo de Fibonacci

UCU

$n_k$ indica el número de llamadas recursivas hechas por ***Fibonacci (k)***. Entonces

- $n_0 = 1$
- $n_1 = 1$
- $n_2 = n_1 + n_0 + 1 = 1 + 1 + 1 = 3$
- $n_3 = n_2 + n_1 + 1 = 3 + 1 + 1 = 5$
- $n_4 = n_3 + n_2 + 1 = 5 + 3 + 1 = 9$
- $n_5 = n_4 + n_3 + 1 = 9 + 5 + 1 = 15$
- $n_6 = n_5 + n_4 + 1 = 15 + 9 + 1 = 25$
- $n_7 = n_6 + n_5 + 1 = 25 + 15 + 1 = 41$
- $n_8 = n_7 + n_6 + 1 = 41 + 25 + 1 = 67$.

```
Algoritmo Fibonacci (k)
COM
SI k = 0 o k = 1
  devolver k
  SINO
    devolver
        Fibonacci (k - 1) +
        Fibonacci (k - 2)
FINSI
FIN
```





## TA6 – Ej 2 Progamación dinámica – Fibonacci

UCU

- ¿Cumple este algoritmo recursivo con los requerimientos para programación dinámica?
  - –¿Sub-problemas superpuestos?
  - –¿Subestructuras óptimas?
  - –¿podemos usar “Memoización”?







9

## Fibonacci

UCU

*Memo* ← {} //**diccionario, mapa**
**fib(n)**
COM
SI **n** no está en *memo*
SI n <= 2
memo[n] ← 1
SINO
*memo*[n] ← fib(n – 1) + fib(n – 2)
devolver *memo*[n]
FIN

- O(n)
- Usa espacio O(n)
- primero ***descomponemos recursivamente*** en sub-problemas y luego se calculan y almacenan los resultados



33

## Fibonacci

UCU

**fib(n)**
**COM**
SI n = 0
devolver 0
SINO
fibAnterior ← 0,
fibActual ← 1
REPETIR (n – 1) veces // cortamos para n = 1
fibNuevo ← fibAnterior + fibActual
fibAnterior ← fibActual
fibActual ← fibNuevo
FIN REPETIR
devolver fibActual
**FIN**

- O(n)



34