Computabilidad y Complejidad

Con MT aprobada suman el puntaje correspondiente.

Lo que queda:

- Complejidad
- Repaso y metodología extra para análisis de algoritmos

Aunque el primer uso que le daremos a la Notación Asintótica será en el contexto de Complejidad, se puede introducir la idea asociada al análisis de algoritmos

Algoritmia

- Diseño (creación-propuesta)
- Análisis (comportamiento)

} ==> De algoritmos

Problema Computacional (relación con MT)

"Un problema computacional consiste en la especificación de un conjunto de datos de entrada junto con la salida requerida para cada entrada."

"Un problema computacional consiste en una caracterización de un conjunto de datos de entrada, junto con una especificación de la salida deseada en base a cada entrada."

"We can also view an algorithm as a tool for solving a well-specified computational problem. The statement of the problem specifies in general terms the desired input/output relationship. The algorithm describes a specific computational procedure for achieving that input/output relationship."

Instancias (relación con MT)

"Un problema computacional tiene una o más instancias, valores particulares de los datos de entrada, sobre las cuales se puede ejecutar un algoritmo para resolver el problema"

"Un problema computacional tiene una o más instancias (valores particulares que toman los datos de entrada)."

"In general, an instance of a problem consists of the input (satisfying whatever constraints are imposed in the problem statement) needed to compute a solution to the problem."

Análisis de Algoritmos-Leiserson: "The theoretical study of computer-program performance and resource usage"

Análisis de Eficiencia: Cantidad de recursos que utiliza (tiempo, espacio)

Definición de Wikipedia (Analysis of algorithms)
"To analyze an algorithm is to determine the amount of resources (such as time and storage) necessary to execute it. Most algorithms are designed to work with inputs of arbitrary length. Usually the efficiency or ~~**complexity**~~ of an algorithm is stated as a function relating the input length to the number of steps (time complexity) or storage locations (space or memory complexity) required to execute the algorithm."

¿Por qué hacer análisis de algoritmos?
- Comparación teórica
- Escalabilidad (cuánto más grande se puede, hasta dónde llego)
- Matemática algorítmica provee un lenguaje de comportamiento de programas
- Se podría generalizar el estudio a otros recursos de cómputo

Recursos
- Tiempo
- Espacio/almacenamiento

Cómo medir recursos (tiempo/espacio) de manera objetiva:
1) Empírico
   - Tiempo de ejecución (depende de la máquina)
   - Cantidad de instr. (depende del lenguaje y del compilador, "tedioso")
2) Teórico

El análisis teórico está apoyado por el principio de invarianza
> "Según el principio de invarianza, dos implementaciones diferentes del mismo algoritmo no difieren en tiempo de ejecución más que en una constante multiplicativa."

Lo cual es, por lo menos, intuitivamente cierto.

¿De qué dependen los recursos, dado un algoritmo? Existe algún tipo de dependencia de la E/ o "tamaño de la E/": cuantificación. Similitud con cadena de entrada de MT.

De hecho: cuantificar E/ (tamaño) y obtener una función de la cantidad de recursos para una E/

Definiciones de cantidades de recursos en función de la entrada:

1) Dado un algoritmo A, el tiempo de ejecución $t_A(n)$ de A es la cantidad de pasos, operaciones o acciones elementales que debe realizar el algoritmo al ser ejecutado en una instancia de tamaño n.

2) El espacio $e_A(n)$ de A es la cantidad de datos elementales que el algoritmo necesita al ser ejecutado en una instancia de tamaño n, sin contar la representación de la entrada.

Estas definiciones son ambiguas en dos sentidos:

No especifica claramente cuáles son las operaciones o datos elementales.

Dado que existe más de una instancia de tamaño n no está claro cuál de ellas es la que se tiene en cuenta para el análisis (o cuáles, si son relevantes).

Tipos de instancias (¿Por qué) - Tipos de análisis (si hay dependencia de datos): **peor**-mejor-promedio-probabilístico

Operaciones Elementales:

Una operación elemental es aquella cuyo tiempo de ejecución está acotado por una constante que depende sólo de la implementación usada

No depende del tamaño de la entrada

Por lo tanto sólo interesa el número de operaciones elementales

En general, las sumas, multiplicaciones y asignaciones son operaciones elementales

Ej para contar operaciones elementales:

Proc Addone(T[1...n])	Op. (semántica operacional)	Costo	Veces (semántica operacional)

```
Proc Addone(T[1...n])
   for i := 1 to n do
      T[i] := T[i] + 1
```

a) No hay casos posibles (es independiente de la entrada)
b) Son ecuaciones relativamente "grandes" (en cantidades de constantes) y además no se tiene una idea muy precisa desde el punto de vista teórico de la relación entre las constantes.

Por lo tanto: notación asintótica

a) Utilizaremos la noción de tiempo ignorando las constantes multiplicativas que lo afectan.
b) El objetivo es caracterizar en forma simple la eficiencia (o uso de recursos) de los algoritmos.
c) Se pretende también independizarlo de las características específicas de la implementación.
d) Se denomina asintótica porque analiza el comportamiento de las funciones en el límite, es decir considerando sólo su tasa de crecimiento.

Definiciones y Propiedades

1) "Una función **t(n)** está **en el orden de f(n)** si existe una constante real positiva **c** y un umbral **u0** tal que $t(n) \leq c.f(n)$ para todo $n > u0$"

2) Brassard-Bratley:

Conjuntos: N (incluye el 0), $R^{\geq 0}$, R^{+}: p. 8

Cuantificadores, propiedades de conjuntos: $\exists$, $\forall$, con ∞, relaciones de "fortaleza/debilidad" o tamaño de conjuntos: p. 10-11, $\exists$, $\exists^{\infty}$, $\forall^{\infty}$, $\forall$

O(f(n)): p. 80

$$O(f(n)) = \{t: N \rightarrow R^{\geq 0} / (\exists\, c \in R^{+}) (\forall^{\infty}\, n \in N) [t(n) \leq c\ f(n)]\}$$

¿Podría ser R^{+}?

Notación y terminología:

Conjuntos ==> $\in$

"está en el orden de" (conjuntos)

"es"

$n^2 = O(n^3)$ (one-way equality, podría relacionarse con el "es")

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

$2n^2 + O(n) = O(n^2)$

Se acepta que $t(n) \in O(f(n))$ sii $\exists\ c \in R^+, n_0 \in N$ tq $0 \leq t(n) \leq c\ f(n);\ n \geq n_0$

Sin poner restricciones para $n < n_0$, t(n) y f(n) podrían dar valores negativos o no estar definidas, por ejemplo:

$O(n / \log n)$, $n = 0$ y $n = 1$ no están definidos

$t(n) = n^3 - 3n^2 - n - 8 \in O(n^3)$ aunque $n \leq 3$ ==> $t(n) < 0$

¿Para qué se usa esto?

Se asocia a $t_A(n)$ o $e_A(n)$, tiempo o espacio de un algoritmo A, su "clase" de requerimientos.

La relación t(n) está en el orden de f(n) se denota como $t(n) \in O(f(n))$

Notar la diferencia entre t(n) y $t(n) \in O(f(n))$... ¿poría llegar a ser relevante?

Otra definición para O(f(n))

$O(f(n)) = \{t: N \rightarrow R^+: \exists\ c \in R^+, n_0 \in N$ tq $t(n) \leq c\ f(n);\ n \geq n_0\}$

Es más "precisa" que $\forall^\infty$, porque pone las excepciones al ppio.

Es más "precisa" con R^+, ¿Por qué?

Notación omega: $\Omega(\ )$

B-B: p. 86

$\Omega(f(n)) = \{t: N \rightarrow R^+: \exists\ c \in R^+, n_0 \in N$ tq $t(n) \geq c\ f(n);\ n \geq n_0\}$

En realidad, tanto O( ) como $\Omega(\ )$ son ambiguas/"caprichosas", la notación más precisa es $\Theta(\ )$

B-B: p. 87

$\Theta(f(n)) = \{t: N \rightarrow R^+: \exists\ c_1, c_2 \in R^+, n_0 \in N$ tq $c_1\ f(n) \leq t(n) \leq c_2\ f(n);\ n \geq n_0\}$

Relaciones entre las funciones que pertenecen a O(), $\Omega()$ y $\Theta()$:

Regla de Dualidad: $g(n) \in \Omega(f(n))$ sii $f(n) \in O(g(n))$

$g(n) \in \Theta(f(n))$ sii $g(n) \in O(f(n))$ y $g(n) \in \Omega(f(n))$

$\Theta(f(n)) = O(f(n)) \cap \Omega(f(n))$

¿Cómo se demuestran? (definiciones de referencia)

Reflexividad y Transitividad de la pertenencia a O(), $\Omega()$ y $\Theta()$, ¿por qué?

El umbral n_0 puede puede resultar resultar útil pero nunca es necesario cuando se consideran funciones estrictamente positivas ($t, f: N \rightarrow R^+$).

Regla del Umbral:

Sean f, t: $N \rightarrow R^+$: $t(n) \in O(f(n))$ <==> existe una constante real positiva c tal que $t(n) \leq c\ f(n)$ para todo natural n.

Se tiene la definición de referencia

$O(f(n)) = \{t: N \rightarrow R^+: \exists\ c \in R^+, n_0 \in N$ tq $t(n) \leq c\ f(n); n \geq n_0\}$

Idea de la demostración: se debe demostrar ==> y <==

==>) Sean f, t: $N \rightarrow R^+$: $t(n) \in O(f(n))$ ==> existe una constante real positiva c tal que $t(n) \leq c\ f(n)$ para todo natural n.

<==) Sean f, t: $N \rightarrow R^+$: $t(n) \in O(f(n))$ <== existe una constante real positiva c tal que $t(n) \leq c\ f(n)$ para todo natural n.

Sean f, t: $N \rightarrow R^+$ existe una constante real positiva c tal que $t(n) \leq c\ f(n)$ para todo natural n ==> $t(n) \in O(f(n))$

==>: tenemos que $t(n) \in O(f(n))$, es decir que existe $a \in R^+$ y $n_0 \in N$ tq $t(n) \leq a\ f(n)$, $n \geq n_0$

$b = máx\{t(n) / f(n)\}$ con $0 \leq n < n_0$...

==> $t(n) / f(n) \leq b$

==> $t(n) \leq b\ f(n)$

<==: con $n_0 = 0$ y ya está

Asociada a la idea de "capturar" el crecimiento asintótico ¿qué pasa con la suma de funciones?

Regla del Máximo:

Sean f, g: $N \rightarrow R^+$, $O(f(n) + g(n)) = O(max(f(n), g(n)))$

Idea de la demostración: se debería demostrar la = de conjuntos, se usará que

$f(n) + g(n) = min(f(n), g(n)) + max(f(n), g(n))$

$0 \leq min(f(n), g(n)) \leq max(f(n), g(n))$ si se suma $max(f(n), g(n))$ a todo

$max(f(n), g(n)) \leq f(n) + g(n) \leq 2\ max(f(n), g(n))$

$t(n) \in O(f(n) + g(n))$ ==> $t(n) \in O(max(f(n), g(n))$

$t(n) \in O(max(f(n), g(n))$ ==> $t(n) \in O(f(n) + g(n))$

¿Cómo se prueba en general? ¿Qué es en general?

La Regla del Máximo vale para Θ()

Relacionado con la notación: $t(n) = 12\ n^3 - 5\ n^2 + 13\ n + 36$, e identificamos que lo que determina el crecimiento asintótico es $f(n) = n^3$ ==> simplificación. Nótese la inconsistencia de tratar $-5\ n^2$ (de la

separación de términos de t) como si fuera g: $N \rightarrow R^+$.

Se puede utilizar $\max(7\ n^3, 5\ n^3 - 5\ n^2, 13\ n, 36)$

Pero hay que demostrar que $5\ n^3 - 5\ n^2 \geq 0$ ¿Cómo?

¿Cómo sería con $t(n) = 12\ n^3 \log n - 5\ n^2 + \log^2 n + 36$? (notar que $\log n \not\exists\ n = 0$)

Si se pudieran utilizar funciones que no son $N \rightarrow R^+$: !!!!

$$\Theta(n) = \Theta(n + n^2 - n^2) = \Theta(\max(n, n^2, -n^2)) = \Theta(n^2)$$

Interpretación práctica de la Regla del Máximo: intuitivamente secuencias de tareas ==> suma ==> máximo de los involucrados.

La idea de la notación **asintótica** tiene relación con la idea de crecimiento infinito de la E/ y del *comportamiento* de las funciones *en el límite*, de allí que se puede relacionar la notación asintótica con los límites (p. 84 B-B), **Regla del Límite**:

1) $\lim_{n \rightarrow inf.} f(n) / g(n) \in R^+$ ==> $f(n) \in O(g(n))$ y $g(n) \in O(f(n))$

2) $\lim_{n \rightarrow inf.} f(n) / g(n) = 0$ ==> $f(n) \in O(g(n))$ y $g(n) \notin O(f(n))$

3) $\lim_{n \rightarrow inf.} f(n) / g(n) \rightarrow inf.$ ==> $f(n) \notin O(g(n))$ y $g(n) \in O(f(n))$

Considerando los tres conjuntos de funciones definidos por la notación asintótica:

1) $\lim_{n \rightarrow inf.} f(n) / g(n) \in R^+$ ==> $f(n) \in \Theta(g(n))$

2) $\lim_{n \rightarrow inf.} f(n) / g(n) = 0$ ==> $f(n) \in O(g(n))$ y $f(n) \notin \Theta(g(n))$

3) $\lim_{n \rightarrow inf.} f(n) / g(n) \rightarrow + inf.$ ==> $f(n) \in \Omega(g(n))$ y $f(n) \notin \Theta(g(n))$

A partir de ahora: reglas y métodos para análisis de algoritmos más estrategias de diseño con *su correspondiente* tipo de análisis:

(1) Estructuras de control
(2) Barómetro
} Usualmente aplicado a peor caso, son "en detalle"

(3) Análisis del caso promedio
(4) Análisis amortizado
} Usualmente dependiente del "tipo de entrada"

(5) Recurrencias } ¿? Un tipo específico de algoritmos

(6) No veremos análisis asociados a algoritmos específicos *de* estructuras de datos

(7) Diseño + análisis
- Greedy
- Divide-and-Conquer
- Dynamic programming
- Algoritmos probabilísticos

(1) Análisis de estructuras de control (algoritmos parte-por-parte)

Secuencias:

P_1; P_2 → Regla del máximo

si $t_1(n)$ y $t_2(n)$ ==> $t_{1;2}(n) = t_1(n) + t_2(n) \in O(\max(t_1, t_2))$ y $t_{1;2}(n) \in \Theta(\max(t_1, t_2)$

Selección (dep. de n asumido):

If (cond) t_1
Then (cuerpo then) t_2
Else (cuerpo else) t_3

Se considera directamente el peor caso: $t_1 + \max(t_2, t_3)$ es equivalente al $\max(t_1, t_2, t_3)$

Iteraciones for o *ciclos uniformes*:

For i ← 1 to m
Do P(i) (puede ser P(i, n))

Siendo t(i) el tiempo de P(i)
Si t(i) no depende de i ==> m t(i) = m t(...) ¿De qué dependería?
Si t(i) depende de i ==> $\Sigma_{i=1}^{m}$ t(i)

Aclarar *inicio*, *fin*, *paso*: *m* = (*fin – inicio*) *% paso* + 1

Diferencia entre peor caso y tamaño de la entrada

Usualmente, se pueden asociar (o intentar) los t(i) a alguna serie de números:
Serie aritmética: a, a+d, a+2d, a+3d, ...: $s_m = am + m (m-1) d / 2$
Serie geométrica: a, ar, ar^2, ar^3, ...: $s_m = a (1-r^m) / (1-r)$ si $r \neq 1$, $s_m = am$ si $r = 1$

Sumas útiles: $\Sigma_{i=1...n} i = n (n+1) / 2$, $\Sigma_{i=1...n} i^2 = n (n+1) (2n+1) / 6$

Iteraciones o *ciclos no uniformes*: while y repeat. Dos formas de análisis:
1) Función de variables que decrece
2) Idea de recursión/recurrencia

Análisis de iteraciones no uniformes (while y repeat):
1) Función de variables que decrece
El valor de la función decrece a medida que se llevan a cabo más iteraciones
El valor de la función debe ser un entero positivo ==> el alg. termina
¿Cuándo? Entender la forma en que decrece la función. Ej: bin_search
Algoritmo: pág. 103 B-B

```
function bin_search(T[1...n], x)        // x está en T
    i ← 1; j ← n
    While i < j Do                                  ==> Cantidad determinada por la dif.
        k ← (i + j) % 2
        Case x < T[k]: j ← k-1
             x = T[k]: i, j ← k         // Return k
             x > T[k]: i  ← k+1
    Return i
```

¿Tiene casos/instancias posibles?
¿Cuál sería la función de las variables que decrece?
$d = j - i + 1$ i.e., cantidad de elems. entre los cuales se busca (*similitud* con n)
$d \leq 0$ ¿Tiene sentido?
$d = 1$ Es el elemento, se encontró (el elemento está)
$d \geq 2$ ¿Qué pasa?

¿Decrece? Veamos los valores de i, j y d al principio y al final de una iteración cualquiera. i', j', y d'. Análisis de caso por caso, es decir el "case":

1) $x < T[k]$:
$j' = (i + j) \div 2 - 1$; $i' = i$;
$d' = (i + j) \div 2 \cancel{-1} - i \cancel{+1} \leq (i + j) / 2 - i = - i/2 + j/2 < - i/2 + j/2 + 1/2 =$
$= (j - i + 1) / 2 = d/2 < d$

2) $x > T[k]$:
$j' = j$; $i' = (i + j) \div 2 + 1$;
$d' = j - (i + j) \div 2 \cancel{+1} \cancel{-1} \leq j - (i + j) / 2 < j - (i + j) / 2 + 1/2 =$
$= j - (j - i - 1) / 2 = (j - i + 1) / 2 = d/2 < d$

3) $x = T[k]$:
$d' = 1 \leq d/2$

¿Cuántas iteraciones? d_l: valor de $j^{(l)}-i^{(l)}+1$ al final de la iteración l
$d_0 = n$
$d_1 = n/2$
...
$d_i = n/2^i$
Peor caso, "todas" las iteraciones hasta que $d_l = 1$
$d_l = n/2^l = 1$; ¿l? $n = 2^l$ ==> $\log_2 n = \log_2 2^l$ ==> $l = \log_2 n$ ==> $\lceil l = \log_2 n \rceil$

Análisis de iteraciones no uniformes (while y repeat):

2) Recurrencia

t(d): t para resolver el problema con d elementos, j-i+1

$$t(d) = \begin{cases} 1 & d = 1 \\ t(d/2) + c & d > 1 \end{cases}$$

Análisis de iteraciones no uniformes: resumen

Dos formas: **1)** función de variables involucradas que decrece
2) Recurrencia

Las dos formas son equivalentes en dificultad (¿y en resultado?)

Hasta acá fue el análisis de estructuras de control, veamos algo un poco más general/otras formas

(1) Estructuras de control
(2) Barómetro
} Usualmente aplicado a peor caso, son "en detalle"

(3) Análisis del caso promedio
(4) Análisis amortizado
} Usualmente dependiente del "tipo de entrada"

(5) Recurrencias } ¿? Un tipo específico de algoritmos

(6) No veremos estructuras de datos

(7) Diseño + análisis

(2) Barómetro

Una instr. barómetro es la que se ejecuta tantas veces como cualquier otra excepto quizás una cantidad constante (acotada) de veces.

Si t(n) es el tiempo del algoritmo a analizar, t(n) es $\Theta(f(n))$ donde f(n) es la cantidad de veces que se ejecuta la instrucción barómetro

Dos cosas:

1) Instrucción barómetro
2) Encontrar f(n). Se usan los principios de las estructuras de control

Ej: select(T[1..n]), pág. 106 B-B

```
Function select_sort(T[1...n])          // n pasadas
    For i ← 1 to n-1 Do
        minj ← i; minx ← T[i]
        For j ← i+1 to n Do
            If T[j] < minx Then
                minj ← j
                minx ← T[j]
        T[minj] ← T[i]
        T[i] ← minx
```

Notar la diferencia entre t(n) y Θ(f(n))

Se ven algunos detalles extra en el ejemplo del algoritmo pigeonhole del libro, p. 105

(3) Caso promedio (sin detalles)

Uso implícito/explícito de distr. de probabilidades de las instancias de tamaño n

En principio, para ordenar vectores, el tiempo de las n! posibles dividido n!

Asumir que son todas igualmente probables y usar esto en el análisis, tal como en el ej. de Insertion sort, p. 62 del libro, con el análisis hecho en la pág. 111

En este caso específico, se tiene en cuenta la distribución de probabilidad de los números del arreglo en partes cada vez menores del mismo.

(4) Análisis amortizado (sin detalles)

Ejemplo p. 113

```
Procedure IncBin(Cntr[1...m])          // Cntr[i] ∈ {0, 1}
    j ← m+1
    Repeat
        j ← j – 1
       C[j] ← 1 – C[j]
    Until C[j] = 1   Or   j = 1
```

Es muy poco probable que en todas las llamadas se tenga siempre el peor caso (*a veces*)

Promedio de t(n) en llamadas sucesivas, no independientes

Tres métodos = { Agregado, Contable, Potencial

(5) Recurrencias

Funciones de t dadas en función de sí mismas: "An equation that defines a function in terms of its own value on smaller inputs" (valores iniciales o *de límite/contorno*)

¿De dónde vienen? D&C (estrategia top-down de diseño)

D-S/C-C (Divide-Solve/Conquer-Combine)

Idea: Factorial(n)

```
Factorial(n)
   if (n=0)
      return 1
   else
     return n * Factorial(n-1)
```

$$t_{fac}(n) = \begin{cases} 1 & n = 0 \text{ (solamente el return)} \\ t_{fac}(n-1) + 2 & n \geq 1 \text{ (return y *, más lo que "cueste" con n-1)} \end{cases}$$

A priori, se podría ver cómo evolucionan los valores de las llamadas y las ops. asoc.

$t(n) = t(n-1) + 2$ (1)
$t(n) = t(n-2) + 2 + 2$ (2)
...
$t(n) = t(0) + 2 + ... + 2$ (n veces + 2) (n)
$t(n) = t(0) + 2\,n$
$t(n) = 2\,n + 1$

La propuesta sería, entonces, $t(n) = 2\,n + 1$, a esto se le llama "sustitución"

Se debería probar por inducción para $n \geq 1$
¿n = 0? $t(0) = 2x0 + 1 = 1$, $t(0) = 1$
Hi) $t(h) = 2\,h + 1$
Dem) ¿$t(h+1) = 2(h+1) + 1$?
$t(h+1) = t(h) + 2 = 2\,h + 1 + 2 = 2\,(h+1) + 1$

recurrencia Hi

Brassard-Bratley:

Intelligent guesswork
Cambio de variables
Recurrencias homogéneas → ecuación característica → polinomio característico → ... → *Solución*
Recurrencias no homogéneas → Recurrencias homogéneas → ... → Solución → *Backtrack* (relación con las no homogéneas)

En vez de esto, usaremos (y veremos algunas ideas asociadas a) dos "recetas" donde se suele incluir en un f(n) el costo de la división y de la *recombinación*

Cookbooks:

a) Llamada recursiva con la entrada menos una constante (como factorial)

$$t(n) = \begin{cases} c & n \le b \\ a\ t(n-b) + f(n) & n > b, \text{ con } f(n) \in O(n^k) \end{cases}$$

¿Qué es f(n)?

$$t(n) \in \begin{cases} O(n^{k+1}) & \text{si } a = 1 \\ O(a^{n \text{ div } b}) & \text{si } a > 1 \end{cases}$$

Vale con todos $\Theta(\ )$

Ejs: factorial y Hanoi (pág. 110)

```
Procedure Hanoi (m, i, j)       // Traslada los m anillos más pequeños de i a j (1, 2, 3)
   If m > 0 Then
      Hanoi (m-1, i, 6-i-j)
      write "i → j"
      Hanoi(m-1, 6-i-j, j)
```

b) Llamada recursiva con la entrada dividida por una constante (¿como cuál?)

$$t(n) = \begin{cases} c & n \le b \\ a\ t(n/b) + f(n) & n > b, \text{ con } f(n) \in O(n^k) \end{cases}$$

$$t(n) \in \begin{cases} O(n^{\log_b a}) & \text{si } a > b^k \\ O(n^k \log_b n) & \text{si } a = b^k \\ O(n^k) & \text{si } a < b^k \end{cases}$$

Vale con todos $\Theta(\ )$. Teorema Maestro o Método Maestro.

Fibonacci

Recursiva

```
FibRec(n)
   if (n < 2)
     then return n
     else return FibRec(n-1) + FibRec(n-2)
```

¿Recurrencia?

Iterativa

```
FibIter(n)
   i ← i; j ← 0
   for k ← 1 to n do
       j ← i + j
       i ← j – i // fib (k-1)
   return j
```

(6) No veremos estructuras de datos

(7) En general para los de diseño + análisis: hay más énfasis en diseño que en análisis

¿Divide and Conquer?

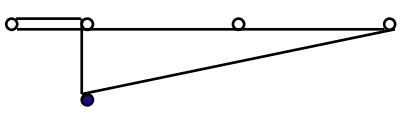
Algoritmos Greedy

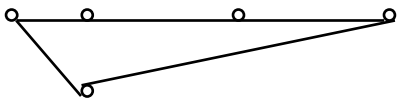
Se aplican a problemas de optimización, construyendo la solución paso a paso de manera iterativa

Toman decisiones en cada paso con la información disponible en el momento de la decisión, dependiendo del estado de avance/solución

En cada paso, es usual tener que elegir entre un conjunto de alternativas o candidatos que se pueden evaluar entre sí y se elige el mejor (en realidad el que *parece mejor*)

Ejemplo del viajante de comercio





Programación dinámica

Estrategia bottom up

Se comienza resolviendo las partes o problemas más sencillos posibles

Se reutilizan resultados intermedios (*tablas*)

Ej. fibonacci: f(n) = f(n-1) + f(n-2)

Algoritmos probabilísticos

Cuando se debe tomar una decisión, se toma al azar, no se computan costos p/ evaluar.

a) Numéricos: intervalo de confianza sobre la respuesta, ej: 90% de acierto para x ± y

b) Monte Carlo: respuesta exacta con alta probabilidad, pero puede ser errónea a veces

Testeo de primalidad (si un número es primo...), ¾ "verdadero"

c) Las Vegas: respuesta exacta o no respuesta