Notas de la clase 1 – complejidad algorítmica

Francisco Soulignac

March 16, 2019

Aclaración: este es un punteo de la clase para la materia AED3. Se distribuye como ayuda memoria de lo visto en clase y, en cierto sentido, es un reemplazo de las diapositivas que se distribuyen en otros cuatrimestres. Sin embargo, no son material de estudio y no suplanta ni las clases ni los libros. Peor aún, puede contener "herrorez" y podría faltar algún tema o discusión importante que haya surgido en clase. Finalmente, estas notas fueron escritas en un corto período de tiempo. En resumen: **estas notas no son para estudiar sino para saber qué hay que estudiar**.

Tiempo total: 260 minutos

1 Motivación (10 mins)

- Problema: transformar conjunto de intervalos propios a unitario

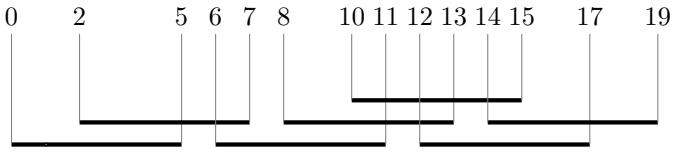
  Input: una cadena con s's y t's, donde el i-ésimo s y el i-ésimo t definen un intervalo I_i.

  Output: una asignación de un valor a cada s y cada t de forma tal que todos los intervalos midan lo mismo.

  Ejemplo: input = *sststsststsstt*; output posible donde todos miden 5 (ver imagen).

Problema de transformar intervalos propios a unitarios

Ejemplo de un output para el input *sststsststsstt* donde todos los intervalos miden 5.



- Algoritmo simple basado en ajustar extremos de intervalos.

- No analizamos si es correcto, dado que no es el objetivo en esta clase; la demostración de que es correcto se encuentra en [3].

unitary.py

```
def unitary(model):
  """transforma modelo propio en unitario donde todos miden 1"""
  #memoria para el output -> S[i] = posicion de i-esimo s.
  #La del i-esimo t es la de S[i] + 1.0
  #el resultado se guarda en res
```

```
    S = [(0,1) for i in range(0, len(model)//2)]
    res = list()

    #posicion actual e indices del ultimo s y t vistos
    pos = (0,1)
    s, t = 0, 0

    for i in range(0, len(model)):
      if model[i] == 's':
        pos = suma(pos, (1, 2**i)) #pos += pos + 1/2^i
        S[s] = pos
        s += 1
      else:
        pos = suma(S[t], (1, 1))   #pos = S[t] + 1.0
        t += 1
      res.append(pos)

    return res

def suma(a, b):
    """a + b; tanto a como b son pares <numerador, denominador>"""
    c = gcd(a[1], b[1])
    return (a[0]*b[1]//c + b[0]*a[1]//c, a[1]*b[1]//c)
```

- Análisis rápido de complejidad temporal.
- Comportamiento real: parece cuadrático o similar

2 Notación O – repaso (20 mins)

- $f, g\colon \mathbb{N} \to \mathbb{R}_{\geq 0}$ funciones de "medida"
- $g \in O(f)$ si y sólo si existen c, n_0 tal que $g(n) \leq cf(n)$ para todo $n \geq n_0$
  - ▷ equivalentemente $\lim_{n\to\infty} g(n)/f(n) < \infty$
- $g \in \Omega(f)$ si y sólo si $f \in O(g)$
- $\Theta(f) = O(f) \cap \Omega(f)$
- Observación: O y Ω definen el mismo preorden $\precsim$[1], donde $f \precsim g$ si y sólo si $f \in O(g)$ (resp. $g \in \Omega(f)$). La relación de equivalencia $\sim$ de $\precsim$ es Θ: $f \sim g$ si y sólo si $f \in \Theta(g)$.
- El preorden estricto $\prec$ inducido por $\precsim$ se corresponde con o y ω:
  - ▷ $g \in o(f)$ si y sólo si $\lim_{n\to\infty} g(n)/f(n) = 0$
  - ▷ $g \in \omega(f)$ si y sólo si $f \in \omega(g)$.
- Abusos de notación:
  - ▷ usamos $=$ en lugar de $\in$ ($=$ no es una equivalencia; es más parecido a $\precsim$). Ej: $n \to n = O(n \to n^2)$, $n \to n \log n = o(n \to n^2)$.

[1] Un preorden es una relación $\precsim$ que es reflexiva y transitiva. Su *relación de equivalencia* $\sim$ satisface $a \sim b$ si y solo si $a \precsim b$ y $b \precsim a$. El *preorden estricto* es $\prec = \precsim \setminus \sim$, i.e., $a \prec b$ si y sólo si $a \precsim b$ y $b \not\precsim a$.

▷ expresión sobre variable en lugar de función en $O(\bullet)$. Ej: $n = O(n^2)$, $n \log n = o(n^2)$

▷ uso de O en expresiones algebraicas: $f(O(g_1), \ldots, O(g_k)) = O(f(c_1 g_1), \ldots, f(c_k g_k))$ para algunas constantes desconocidas $c_1, \ldots, c_k$. Ej: $O(n)+O(n^2) = O(n^2)$, $\sum_{i=1}^{n} O(n) = O(n^2)$, $n^{O(1)} = \bigcup_{k=1}^{\infty} O(n^k)$.

▷ uso coloquial de O para transformar una expresion en funcion. Ej: $|f(s)| = \Theta(|s|)$ para $s \in C$.

3 Concepto de algoritmo y complejidad (60 minutos)

• Informalmente, un algoritmo es una "descripción" de un proceso mecánico que transforma un input en un output. (Formalmente, esperar a LyC).

• Tanto el algoritmo, como el input y el output son "cadenas".

• Alfabeto: conjunto de símbolos. Usamos $\Sigma_n = \{0, \ldots, n-1\}$.

• Cadena (sobre Σ_n): secuencia de símbolos; podría ser infinita. Σ_n^* es el conjunto de cadenas.

• Lenguaje: subconjunto de cadenas. Un lenguaje finito puede tener cadenas infinitas y un lenguaje sin cadenas infinitas puede ser infinito.

• Observación: todo lenguaje no unario L puede ser codificado en binario a través de una biyección f tal que $|f(s)| = \Theta(|s|)$ para todo $s \in L$. Por eso, es irrelevante el alfabeto del input, output y algoritmo, mientras sea no unario.

• Los algoritmos son cadenas de un lenguaje de programación que definen un *modelo de cómputo*. Ejemplos[2]:

Máquina de Turing: algoritmo = tabla de transiciones de estados.

Máquina RAM: algoritmo = cadena binaria de un programa en lenguaje ensamblador (Orga 1).

Lenguaje imperativo: algoritmo = secuencia de instrucciones que empieza en algún procedimiento principal. En lenguajes estructurados, puede estar acompañado de descripciones de estructuras de datos.

Lenguaje funcional: una función principal que puede depender de funciones auxiliares.

Lenguaje de objetos: entorno de objetos que intercambian mensajes.

Etc.

• El input y el output tambien son cadenas. Sin embargo, hay una visión dual de acuerdo a si lo interpreta la máquina o el programador.

▷ Para la máquina, el input es una cualquier cadena y cualquier cadena puede ser input. Con esta postura, un algoritmo computa una función $f\colon \Sigma_2^* \to \Sigma_2^*$. **La máquina no entiende de semántica.**

▷ Para el programador, sólo algunas cadenas particulares *representan* inputs validos (recordar invariante de representación) y, más aún, el mismo input puede esta representado por más de una cadena (recordar función de abstracción). Al programador le interesa el valor representado y no la forma en que se representa. Para el programador, un algoritmo es una función $f\colon \mathcal{I} \to \mathcal{O}$ donde $\mathcal{I}$ es el conjunto de *instancias* y $\mathcal{O}$ es el dominio de los outputs. **El programador le da semántica a los bits.**

• El proceso mecánico descrito por un algoritmo se ejecuta con la repetición de un mecanismo. Qué mecanismo se repite depende del modelo de cómputo. Empero, podemos definir el tiempo como la cantidad de veces que se repite el mecanismo. De la misma forma podemos medir otros recursos, como el espacio, que también dependen de cómo se defina la computadora.

[2]El autor de estas notas es consciente de que los estudiantes no conocen todos estos lenguajes; la idea es mostrar que no hay un único modelo de cómputo. Por otra parte, en rigor los lenguajes no son máquinas ni modelos de cómputo donde se mida la complejidad, pero sirve para ejemplificar el punto.

Máquina de Turing: mecanismo = transición de estados

Máquina RAM: mecanismo = ciclo fetch-decode-excecute (Orga 1)

Lenguaje imperativo: mecanismo = operación elemental

Cálculo λ: mecanismo = reducción de un símbolo no terminal.

Lenguaje de objetos: mecanismo = envio de un mensaje.

• El consumo de un recurso se mide con una función. Recordando que desde el punto de vista de la máquina todo string binario es un input, la función de consumo de recurso es una función $t\colon \Sigma_2^* \to \mathbb{N}$. Es importante notar que desde el punto de vista del programador, una instancia puede representarse con distintos strings: e.g., el par $(\{1,2,3\},1)$ se podria representar (con ciertas abstracciones) como 123|1 o como 321|1. Esto indica que podemos tener distintos costos para la misma instancia abstracta. Por ejemplo, al buscar 1 en $\{1,2,3\}$ podemos tener $t(123|1) = 1$ y $t(321|1) = 3$.

• Las funciones de complejidad agrupan los costos de todos los inputs del mismo tamaño. Son funciones $T\colon \mathbb{N} \to \mathbb{N}$.

Peor caso: $T(n) = \max\{t(s) \mid s \in \Sigma_2^* \text{ con } |s| = n\}$

Caso promedio: $T(n) = E\{t(s) \mid s \in \Sigma_2^* \text{ con } |s| = n\}$. En este caso, t se interpreta como una variable aleatoria que indica el tiempo del input s.

• Análisis empírico: ejecutar el algoritmo sobre distintos inputs significativos para medir su complejidad.

▷ Ventajas: análisis de casos usuales; más fácil que calcular la esperanza del tiempo.

▷ Desventajas: hay que programar previamente; tiempo de ejecución no depreciable; no dice nada de casos no ejecutados; overfitting.

Intervalo (10 mins)

4 Modelo de cómputo para AED3 (50 mins)

4.1 Maquina RAM con palabras de tamaño w (30 mins)

• Introducido en [6] y usado como estándar en papers de algoritmia ya que modela "bien" las computadoras reales. En términos de [5] es un "modelo razonable".

• Ejecuta un único programa sobre un único input. Hay que reiniciarla con otro programa o input.[3]

• Memoria de programa y de entrada de solo lectura; memoria de output de solo escritura. La memoria de entrada esta formada por a lo sumo 2^w palabras de w bits cada una.

• $O(1)$ registros, cada uno conteniendo w bits.

• Memoria de trabajo: a lo sumo 2^w celdas indexadas, cada una de los cuales puede contener palabras de hasta w bits.

• Mecanismo fetch-decode-execute hasta instruccion halt.

• Set de instrucciones:

▷ Asumimos que cualquier operación sobre los registros está implementada, requiriendo una sola repetición de fetch-decode-execute.[4] Ergo, cualquier operación sobre una cantidad finita de argumentos de tamaño $O(w)$ se ejecuta en $O(1)$ tiempo.

[3] En cualquier caso, se podría ejecutar un sistema operativo.

[4] Esto no es del todo real, pero alcanza para nuestros propósitos.

▷ Operaciones de salto para implementar while, if, etc. Todas cuestan $O(1)$ tiempo.

▷ Operaciones de load y store en memoria en $O(1)$ tiempo. Todas estas operaciones leen la dirección de la celda de trabajo desde un registro o desde el input. Por ello la memoria está acotada a 2^w celdas: con w bits no podemos identificar más de 2^w celdas.

- Costo temporal: cantidad de repeticiones de fetch-decode-execute.
- Costo espacial: cantidad de celdas de la memoria de trabajo usadas.
- Tamaño de la entrada: wx donde x es el costo espacial.
- Análisis asintótico: a medida que crece el tamaño n del input también crece w ya que $n < 2^w$, lo que permite el análisis cuando n tiende a infinito. Esto equivale a tener que cambiar una máquina cuando no tiene suficiente memoria y al hecho de que las máquinas reales cada vez tienen más capacidad de cómputo. Ergo, w **no es una constante**[5].
- Que podamos ejecutar una instrucción en $O(1)$ tiempo sobre una cantidad no constante w de memoria es algo que resulta físicamente imposible cuando $w \to \infty$. Sin embargo, esta máquina modela la existencia del paralelismo de las máquinas reales y el hecho de que la cantidad de bits de la arquitectura crece con el paso del tiempo. Muchos algoritmos interesantes explotan este paralelismo y queremos modelarlos.
- Nota: los modelos logarítmico y uniforme simplifican la restricción de memoria finita.

4.2 Lenguajes imperativos en arquitecturas de w bits (10 mins)

- Usamos pseudocódigo, C++, Python, etc, y suponemos que los programas se compilan a máquina RAM con palabras de w bits.
- Cada instrucción de alto nivel genera una cantidad de instrucciones de máquina RAM. Aquellas que trabajan con $O(1)$ argumentos de tamaño $O(w)$ generan $O(1)$ instrucciones RAM y se llaman *operaciones elementales*.
- Una instrucción de alto nivel puede requerir $\omega(1)$ instrucciones RAM. Para calcular el costo de la instrucción necesitamos conocer su implementación.

  ▷ Las llamadas a funcion de C++ pueden requerir copias.

  ▷ Las operaciones aritméticas de Python pueden trabajar sobre números grandes.
- Estructuras de datos. Necesitamos conocer su implementación para saber qué costo tiene cada operación y cuál es el tamaño que ocupan.
- Vamos a hacer suposiciones razonables de las implementaciones ([5]).

4.3 Lenguaje coloquial (10 mins)

- A pesar de la ambigüedad, lo vamos a usar y hay que ser riguroso.
- Sirve para garantizar corrección, pero abstrae detalles de implementación importantes para calcular complejidad.
- Hay que describir la implementación con suficiente detalle para poder calcular la complejidad, sabiendo que contamos con el soporte de los lenguajes de programación y la máquina RAM.
- Toda afirmación acerca de una implementación debería poder garantizarse en el modelo formal.

[5] En la práctica w funciona como una constante (la de la arquitectura).

5 Ejemplos de implementaciones razonables (30 mins)

• Es imposible describir toda la implementación hasta el nivel RAM para cualquier algoritmo → implementaciones razonables.

• Mostramos algunos ejemplos para entender cuál es el costo de un algoritmo y cómo medir la complejidad en función del tamaño de entrada.

5.1 Naturales grandes

• En nuestra máquina RAM contamos con 2^w celdas de memoria, cada una con w bits, que nos dan un total de $w2^w$ bits de almacenamiento. El máximo valor representable en esta máquina no supera a $M = 2^{w2^w}$. En consecuencia, cualquier valor $x \leq M$ representable consume $n = \lceil \log x \rceil \leq \log M = w2^w$ bits (es decir, no consume más que toda la memoria, lo cual tiene sentido). Medido en cantidad de celdas, x requiere $\lceil n/w \rceil \leq 2^w$ celdas (nuevamente, no consume más celdas que las disponibles). Para representar a x podemos usar 1 celda para almacenar la longitud n/w de x (medida en celdas), más una secuencia contigua de n/w celdas con los bits de x. Notar que, con nuestra representación, el máximo valor representable realmente es $2^{w2^{w-1}}$, porque necesitamos al menos 1 celda para saber cuánto ocupa el número. Ejemplo: $25 = 11001_2$ en una máquina con $w = 3$ se representa en 3 registros; el primero tiene el valor $2 = 010_2$ que es el tamaño de 25 y luego 2 registros que mantienen 011_2 y 001_2.

• En máquinas con memoria infinita (e.g., máquinas de Turing que se ven en LyC), se pueden implementar usando una secuencia contigua (array) de celdas. Para poder distinguir la última celda, cada digito 0 del número se reemplaza por dos dígitos, un cero seguido de un no-cero, y al final de la secuencia se escribe 00. Ejemplo, el numero $1|0|2|3|0|8_{10}$ en base 10 se podría codificar como $1|01|2|3|01|8|00_{10}$, consumiendo 4 símbolos adicionales.

• En cualquiera de las representaciones anteriores, x se codifica con $\Theta(\log x)$ celdas, lo cual es óptimo salvo un factor constante.

• Suma: sumar las celdas, teniendo cuidado del carry. Dados $x, y \in \mathbb{N}$, tenemos que el costo $t((x, y))$ de sumar el par (x, y) es $\Theta(\log x + \log y)$. Luego, la complejidad temporal $T(n + m)$ de sumar un valor de tamaño n con otro de tamaño m es $O(n + m)$, lo que resulta en un algoritmo lineal.

• Multiplicación: algoritmo por definicion → $T(n+m) = \Theta(n2^m)$; algoritmo primaria → $T(n+m) = \Theta(nm)$; algoritmo Karatsuba $T(n+m) = \Theta((n+m)^{\log_2 3})$. Nosotros vamos a considerar que se puede multiplicar en tiempo $T(n + m) = \Theta(n + m)$, aunque en una máquina real sea un poco mayor que lineal el costo.

5.2 Listas

• Puntero a nodo, donde nodo es un par con un valor y un puntero a nodo. Se reserva una dirección de memoria especial (null) para indicar el fin de la lista. Luego, el tamaño de una lista de n valores $e_1, \ldots, e_n$ es $\Theta(n)$ para los punteros y $\Theta(\sum_{i=1}^{n} |e_i|)$ para los valores.

• Podemos dar algoritmos genéricos para implementar funciones sobre listas. Sin embargo, hasta no saber el tipo exacto de la lista, no podemos saber la complejidad de la operación siguiendo la definición que tenemos hasta ahora. Abajo resolvemos este problema.

Intervalo (10 mins)

6 Ejercitación: análisis rápido (y pobre) de algoritmos numéricos (30 mins)

Nota: el objetivo de esta sección es mostrar la importancia de conocer la representación y no hacer un análisis ajustado. Sólo se busca erradicar el pre-concepto de que las operaciones sobre números requieren tiempo constante, como muchas veces se asume en AED2. Para más información, consultar [2] donde se demuestra que si una máquina RAM puede realizar operaciones aritméticas sobre números grandes en $O(1)$ tiempo, entonces se puede resolver en tiempo polinomial cualquier problema NP-completo.

6.1 Primo

- Problema: determinar si un número $x \in \mathbb{N}$ es primo. Analizamos dos casos:
  - ▷ $x < 2^w$, con lo cual entra en un registro.
  - ▷ $2^w \leq x < 2^{w2^w}$, con lo cual x es un número grande, implementado como en la sección anterior.
- Algoritmo simple: verificar si y divide a x para $y \in 2, \ldots, x-1$.
- Costo para la instancia x:
  - ▷ Si $x < 2^w$, entonces $t(x) = \Theta(x)$ porque cada división cuesta $O(1)$.
  - ▷ Sino, $t(x) = O(xd(x))$ y $t(x) = \Omega(x)$, donde $d(x)$ es el máximo costos de verificar si un número divide a x. Se sabe que $d(x) = O(\mathsf{poly}(n/w))$ donde $n = |x|$.
- Complejidad exponencial en ambos casos, donde $n = |x|$:
  - ▷ Si $x < 2^w$, entonces $T(w) = O(2^n)$. Peor caso: $n = w$.
  - ▷ Caso contrario, $T(n) = O(2^n \mathsf{poly}(n/w))$.
- Desde 2002 se conoce un algoritmo polinomial [1].

6.2 Sumatoria

- Problema: sumar una lista $L = [e_1, \ldots, e_n]$ de números, cada uno de los cuales entra en una celda.
- Algoritmo: acumular en una variable a que puede ser un número grande.
- Análisis: en peor caso la i-esima iteración cuesta $O(\log a) = O(\log(i2^w)) = O(w + \log i)$. Entonces, $t(L) = O(nw + n \log n)$. Por máquina RAM, $n < 2^w$, ergo $t(L) = O(nw + n \log 2^w) = O(nw)$.
- Como el tamaño de la entrada es $O(nw)$, el costo es lineal.
- Nota: la cuenta cambia si los números de la lista no entran en una celda.

6.3 Potencia

- Problema: dados $x, y \in \mathbb{N}$, calcular x^y.
- Análisis: el output requiere $\Theta(\log x^y) = \Theta(2^m n)$ bits, donde $|x| = n$ y $|y| = m$. En consecuencia, $m < w$ y, por lo tanto $y < 2^w$. Cualquier algoritmo que escriba esta cantidad de bits requiere tiempo $T(n + w) = \Omega(2^m n)$, lo que exponencial cuando $m = \Omega(w)$.
- Aprendizaje: cuidado con suponer que las operaciones aritméticas cuestan $O(1)$ tiempo. (No está mal, pero cuidado!; ver abajo).

6.4 Fibonacci

- Problema: dado x calcular el x-ésimo número de fibonacci.
- Observación: existen distintos algoritmos para este problema, uno de los cuales es hacer una cuenta.
- Análisis: el output requiere $\Theta(\log \phi^x) = \Theta(x \log \phi)$ bits, con lo cual cualquier algoritmo es exponencial.

7 Algoritmos genéricos, modelo logarítmico y modelo uniforme (30 mins)

- En los algoritmos genéricos no conocemos el tipo de datos y por lo tanto no sabemos el tamaño de la entrada (en bits) ni el costo de algunas operaciones
- Para el cálculo del costo se eligen algunas operaciones significativas (e.g., comparaciones para sorting)
- Para el cálculo del tamaño se considera que cada valor consume 1 unidad de memoria
- Ejemplo: la complejidad de merge-sort es $T(n) = O(n \log n)$ comparaciones, donde n es la cantidad de elementos de la lista.
- Esta misma idea se puede aplicar sobre tipos conocidos y se usa para definir otros modelos de cómputo que son más simples de operar y proveen "buenas" aproximaciones del costo en general, aunque permiten abusos si se usan mal.

  Modelo uniforme: máquina RAM con infinita memoria donde cada celda puede acomodar números arbitrariamente grandes. Cada número consume 1 unidad de espacio y cada operación aritmética cuesta 1 unidad de tiempo, independientemente de la cantidad de bits.

  Modelo logarítmico: máquina RAM con infinita memoria donde cada celda puede acomodar números arbitrariamente grandes. Sin embargo, se mide el consumo de memoria en bits (i.e., n requiere $\log n$ bits) y las operaciones aritméticas habituales (suma, resta, multiplicación, división, etc) que generan números de $\log n$ bits cuestan $\Theta(\log n)$ unidades de tiempo. Este modelo aproxima bastante bien a la máquina RAM de w bits aunque: 1. pierde la noción de paralelismo ya que nunca $\omega(1)$ bits se pueden operar en $O(1)$ tiempo, 2. no está claro que algunas de las operaciones aritméticas puedan implementarse realmente en $\Theta(\log n)$ tiempo.

- El modelo uniforme es más simple, pero da resultados no tan finos. Supongamos que:
  - ▷ el input tiene n valores, cada uno de $O(b)$ bits,
  - ▷ el algoritmo realiza $T_u(n)$ operaciones, cada una de costo $f(b)$ para un input de tamaño b,
  - ▷ todo valor intermedio requiere $O(g(b))$ bits.

Entonces, en términos de la máquina RAM con palabras de b bits, el tamaño de la entrada es $O(nb)$ y su costo temporal es $T(nb) = O(T_u(n) \cdot f(g(b)))$.

- Si por error consideramos que $f(b) = O(1)$ y $g(b) = O(1)$ como en el modelo uniforme, entonces estamos perdiendo el factor $O(f \circ g(b))$ que puede ser mucho mayor a $T_u(n)$. Pero...
- Como corolario 1: si $b < w$, f es polinomial y $g < (nw)^{O(1)}$, entonces $T(nw) = O(T_u(n)(nw)^{O(1)})$. Es decir, si los valores se mantienen polinomiales, al menos no decimos que un algoritmo exponencial es polinomial.
- Como corolario 2: si $b < w$, f es lineal y $g = O(w + \log n)$ (caso común), entonces $T(nw) = O(T_u(n)(w + \log n))$ que, como $n < 2^w$ por restricción de memoria, implica $T(nw) = O(T_u(n)w)$. Es decir, el costo calculado por el modelo uniforme $T(n) = O(T_u(n))$ donde omitimos w coincide con el de la máquina RAM. Este es el caso típico, e.g., sorting, búsqueda binaria, etc.

- Ejemplo: algoritmo de sumatoria de una lista.
  - ▷ Costo en modelo uniforme: $T_u(n) = \Theta(n)$ sumas, i.e., lineal.
  - ▷ Cada suma es lineal, i.e., $f(b) = O(b)$.
  - ▷ Máximo espacio requerido por un valor intermedio (el acumulador): $\Theta(\log(n2^w)) = O(w + \log n)$ bits.
  - ▷ Conclusion: costo en máquina RAM de w bits es $T(nw) = O(nw)$, i.e., lineal.

- **Resumen**: el modelo uniforme da una buena aproximación cuando los valores intermedios consumen más o menos el mismo espacio que los valores del input. Caso contrario, el modelo uniforme sigue siendo útil, pero no hay que olvidar el costo de cada operación. Si no somos cuidadosos, podemos decir que un algoritmo exponencial es lineal.

8 Ejercitación: análisis final del problema motivador (10 mins)

- Modelo uniforme: tenemos $T_u(n) = \Theta(n)$ operaciones aritméticas.
- Modelo logarítmico: las sumas y multiplicaciones son lineales en cantidad de bits.
- Modelo RAM: las sumas son lineales, las multiplicaciones son "casi" lineales.
- En la i-esima operación aritmética se utilizan valores del orden de $\Theta(2^i)$ en peor caso, con lo cual $\Theta(n)$ operaciones se aplican a valores que requieren $\Theta(n)$ bits.
- En consecuencia:
  - ▷ Costo modelo log: $T(n) = \Theta(n^2)$
  - ▷ Costo modelo RAM: $T(n) = \Omega(nf(n))$ donde $f(n)$ es "casi lineal".

9 Conclusión: complejidad in a nutshell

- Vamos a usar el modelo uniforme, teniendo cuidado de que los valores intermedios no crezcan y conociendo el soporte teórico subyacente.
- En caso que los valores crezcan, vamos a tenerlo en cuenta calculando una cota superior del costo de cada operación.
- Vamos a estar particularmente interesados en no afirmar que un algoritmo exponencial es polinomial.
- No vamos a usar el modelo RAM para el análisis porque es molesto, pero hay que entender que este es el modelo que funciona en el análisis experimental (laboratorio).
- Para profundizar conviene cursar LyC. Luego, se puede consultar la siguiente bibliografía que **no forma parte de la materia**. La bibliografía oficial de la materia para este tema se reduce a [5, Secciones 1–3] y, en caso de querer profundizar, [4, Capítulos 2–4].
- Para más información, se puede consultar la siguiente bibliografía.
  - ▷ Funcionamiento de una máquina real: cualquier libro de Organización de Computadoras.
  - ▷ Análisis practico de complejidad: bibliografía obligatoria.
  - ▷ Definición inicial de modelo RAM: [6, Sección 1].
  - ▷ Modelo uniforme y logarítmico: [7, Capítulo 1].
  - ▷ Límites del modelo uniforme: [2, Sección 1]
  - ▷ Lenguajes, máquinas de Turing y complejidad algorítmica: [8].

References

[1] Manindra Agrawal, Neeraj Kayal, and Nitin Saxena. PRIMES is in P. *Ann. of Math. (2)*, 160(2):781–793, 2004.

[2] Alberto Bertoni, Giancarlo Mauri, and Nicoletta Sabadini. A characterization of the class of functions computable in polynomial time on random access machines. In *Proceedings of the Thirteenth Annual ACM Symposium on Theory of Computing*, STOC '81, pages 168–176, New York, NY, USA, 1981. ACM.

[3] Kenneth P. Bogart and Douglas B. West. A short proof that "proper = unit". *Discrete Math.*, 201(1-3):21–23, 1999.

[4] Gilles Brassard and Paul Bratley. *Fundamentals of algorithmics*. Prentice Hall, Inc., Englewood Cliffs, NJ, 1996.

[5] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to algorithms*. MIT Press, Cambridge, MA, third edition, 2009.

[6] Michael L. Fredman and Dan E. Willard. Surpassing the information-theoretic bound with fusion trees. In *Proceedings of the 22nd Annual ACM Symposium on Theory of Computing (Baltimore, MD, 1990)*, volume 47, pages 424–436, 1993.

[7] Dinesh P. Mehta and Sartaj Sahni, editors. *Handbook of data structures and applications*. Chapman & Hall/CRC Computer and Information Science Series. Chapman & Hall/CRC, Boca Raton, FL, 2005.

[8] M. Sipser. *Introduction to the Theory of Computation*. Cengage Learning, 2012.