

Análisis de complejidad de algoritmos I

Ariel Bendersky¹
Nicolás D'Ippolito¹

¹Departamento de Computación, FCEyN,
Universidad de Buenos Aires, Buenos Aires, Argentina

Algoritmos y Estructuras de Datos II
Segundo cuatrimestre de 2019

(2) Comparemos dos algoritmos

- Tenemos un arreglo de naturales positivos de m posiciones.

(2) Comparemos dos algoritmos

- Tenemos un arreglo de naturales positivos de m posiciones.
- Queremos implementar la función `Alternar(nat x)` que si encuentra a x lo elimina y si no lo agrega.

(2) Comparemos dos algoritmos

- Tenemos un arreglo de naturales positivos de m posiciones.
- Queremos implementar la función Alternar(nat x) que si encuentra a x lo elimina y si no lo agrega.
- Vamos a proponer dos algoritmos. En ambos vamos a llamar n a la cantidad de elementos que están presentes en el arreglo.

(2) Comparemos dos algoritmos

- Tenemos un arreglo de naturales positivos de m posiciones.
- Queremos implementar la función Alternar(nat x) que si encuentra a x lo elimina y si no lo agrega.
- Vamos a proponer dos algoritmos. En ambos vamos a llamar n a la cantidad de elementos que están presentes en el arreglo.
 - ① Recorro el arreglo de la posición 1 a la n , si el elemento no está, lo agrego en la posición $n + 1$. Si está, comprimo el arreglo moviendo todos los elementos una posición “hacia atrás”.

(2) Comparemos dos algoritmos

- Tenemos un arreglo de naturales positivos de m posiciones.
- Queremos implementar la función Alternar(nat x) que si encuentra a x lo elimina y si no lo agrega.
- Vamos a proponer dos algoritmos. En ambos vamos a llamar n a la cantidad de elementos que están presentes en el arreglo.
 - 1 Recorro el arreglo de la posición 1 a la n , si el elemento no está, lo agrego en la posición $n + 1$. Si está, comprimo el arreglo moviendo todos los elementos una posición “hacia atrás”.
 - 2 Recorro el arreglo desde la posición 1 hasta haber pasado n elementos no anulados. Si no está, lo agrego en una de las posiciones libres. Si está, marco la posición como anulada pero no comprimo.

(2) Comparemos dos algoritmos

- Tenemos un arreglo de naturales positivos de m posiciones.
- Queremos implementar la función Alternar(nat x) que si encuentra a x lo elimina y si no lo agrega.
- Vamos a proponer dos algoritmos. En ambos vamos a llamar n a la cantidad de elementos que están presentes en el arreglo.
 - 1 Recorro el arreglo de la posición 1 a la n , si el elemento no está, lo agrego en la posición $n + 1$. Si está, comprimo el arreglo moviendo todos los elementos una posición “hacia atrás”.
 - 2 Recorro el arreglo desde la posición 1 hasta haber pasado n elementos no anulados. Si no está, lo agrego en una de las posiciones libres. Si está, marco la posición como anulada pero no comprimo.
- ¿Cuál de los dos es mejor algoritmo? Con más precisión aún: ¿cuál tarda menos?

(3) Comparando algoritmos

- Hay varias formas de categorizar los algoritmos para tratar de responder a la pregunta de cuál es mejor.

(3) Comparando algoritmos

- Hay varias formas de categorizar los algoritmos para tratar de responder a la pregunta de cuál es mejor.
 - Claridad.

(3) Comparando algoritmos

- Hay varias formas de categorizar los algoritmos para tratar de responder a la pregunta de cuál es mejor.
 - Claridad.
 - Facilidad de programación.

(3) Comparando algoritmos

- Hay varias formas de categorizar los algoritmos para tratar de responder a la pregunta de cuál es mejor.
 - Claridad.
 - Facilidad de programación.
 - Cuán paralelizable es.

(3) Comparando algoritmos

- Hay varias formas de categorizar los algoritmos para tratar de responder a la pregunta de cuál es mejor.
 - Claridad.
 - Facilidad de programación.
 - Cuán paralelizable es.
 - Tiempo de ejecución.

(3) Comparando algoritmos

- Hay varias formas de categorizar los algoritmos para tratar de responder a la pregunta de cuál es mejor.
 - Claridad.
 - Facilidad de programación.
 - Cuán paralelizable es.
 - Tiempo de ejecución.
 - Necesidad de almacenamiento (memoria).

(3) Comparando algoritmos

- Hay varias formas de categorizar los algoritmos para tratar de responder a la pregunta de cuál es mejor.
 - Claridad.
 - Facilidad de programación.
 - Cuán paralelizable es.
 - Tiempo de ejecución.
 - Necesidad de almacenamiento (memoria).
- Muy probablemente para ciertos criterios algunos sean mejores y otros peores. No nos vamos a preocupar en esta materia por cómo balancear esos criterios.

(3) Comparando algoritmos

- Hay varias formas de categorizar los algoritmos para tratar de responder a la pregunta de cuál es mejor.
 - Claridad.
 - Facilidad de programación.
 - Cuán paralelizable es.
 - Tiempo de ejecución.
 - Necesidad de almacenamiento (memoria).
- Muy probablemente para ciertos criterios algunos sean mejores y otros peores. No nos vamos a preocupar en esta materia por cómo balancear esos criterios.
- Sí nos vamos a preocupar por los dos últimos.

(4) Carreras de caballos

- En la cancha, ¿se ven los pingos?

(4) Carreras de caballos

- En la cancha, ¿se ven los pingos?
- ¿A quién le apostarían? ¿A “Adorado Cris” o a “Precioso Chico”?

(4) Carreras de caballos

- En la cancha, ¿se ven los pingos?
- ¿A quién le apostarían? ¿A “Adorado Cris” o a “Precioso Chico”?
- No vamos a elegir por el nombre.

(4) Carreras de caballos

- En la cancha, ¿se ven los pingos?
- ¿A quién le apostarían? ¿A “Adorado Cris” o a “Precioso Chico”?
- No vamos a elegir por el nombre.
- Podríamos ponerlos a correr una vez y decidir en base a eso.

(4) Carreras de caballos

- En la cancha, ¿se ven los pingos?
- ¿A quién le apostarían? ¿A “Adorado Cris” o a “Precioso Chico”?
- No vamos a elegir por el nombre.
- Podríamos ponerlos a correr una vez y decidir en base a eso.
- ¿Nos da garantías sobre el próximo encuentro?

(4) Carreras de caballos

- En la cancha, ¿se ven los pingos?
- ¿A quién le apostarían? ¿A “Adorado Cris” o a “Precioso Chico”?
- No vamos a elegir por el nombre.
- Podríamos ponerlos a correr una vez y decidir en base a eso.
- ¿Nos da garantías sobre el próximo encuentro?

(4) Carreras de caballos

- En la cancha, ¿se ven los pingos?
- ¿A quién le apostarían? ¿A “Adorado Cris” o a “Precioso Chico”?
- No vamos a elegir por el nombre.
- Podríamos ponerlos a correr una vez y decidir en base a eso.
- ¿Nos da garantías sobre el próximo encuentro? Sólo si pudiésemos repetir las condiciones.

(4) Carreras de caballos

- En la cancha, ¿se ven los pingos?
- ¿A quién le apostarían? ¿A “Adorado Cris” o a “Precioso Chico”?
- No vamos a elegir por el nombre.
- Podríamos ponerlos a correr una vez y decidir en base a eso.
- ¿Nos da garantías sobre el próximo encuentro? Sólo si pudiésemos repetir las condiciones.
- El buen apostador hace un análisis teórico: quiénes fueron los padres, quién es el jockey, el stud, etc.

(4) Carreras de caballos

- En la cancha, ¿se ven los pingos?
- ¿A quién le apostarían? ¿A “Adorado Cris” o a “Precioso Chico”?
- No vamos a elegir por el nombre.
- Podríamos ponerlos a correr una vez y decidir en base a eso.
- ¿Nos da garantías sobre el próximo encuentro? Sólo si pudiésemos repetir las condiciones.
- El buen apostador hace un análisis teórico: quiénes fueron los padres, quién es el jockey, el stud, etc.
- Con los algoritmos también tenemos la alternativa empírica (que presenta los mismos problemas) y la teórica (que depende mucho menos del azar que en el caso de los caballos).

(5) Análisis teórico de algoritmos

- Algunas observaciones previas.

(5) Análisis teórico de algoritmos


- Algunas observaciones previas.
- El interés de comparar algoritmos surge para problemas grandes. Para problemas suficientemente chicos no suele ser tan importante qué algoritmo se utiliza.

(5) Análisis teórico de algoritmos

- Algunas observaciones previas.
- El interés de comparar algoritmos surge para problemas grandes. Para problemas suficientemente chicos no suele ser tan importante qué algoritmo se utiliza.
- Ahora bien, ¿qué significa *grande* o *chico*? Debemos dar alguna medida del tamaño del problema.

(5) Análisis teórico de algoritmos

- Algunas observaciones previas.
- El interés de comparar algoritmos surge para problemas grandes. Para problemas suficientemente chicos no suele ser tan importante qué algoritmo se utiliza.
- Ahora bien, ¿qué significa *grande* o *chico*? Debemos dar alguna medida del tamaño del problema.

 A los efectos del análisis de algoritmos tomaremos como medida del problema al tamaño de la entrada, concepto que precisaremos más adelante.

(5) Análisis teórico de algoritmos

- Algunas observaciones previas.
- El interés de comparar algoritmos surge para problemas grandes. Para problemas suficientemente chicos no suele ser tan importante qué algoritmo se utiliza.
- Ahora bien, ¿qué significa *grande* o *chico*? Debemos dar alguna medida del tamaño del problema.

⚠ A los efectos del análisis de algoritmos tomaremos como medida del problema **al tamaño de la entrada**, concepto que precisaremos más adelante.

⚠ Dado que para distintos problemas el concepto de *grande* varía, realizaremos un análisis asintótico: si n es el tamaño de la entrada, nos preguntaremos cómo se comporta el algoritmo cuando $n \rightarrow \infty$.

(5) Análisis teórico de algoritmos

- Algunas observaciones previas.
- El interés de comparar algoritmos surge para problemas grandes. Para problemas suficientemente chicos no suele ser tan importante qué algoritmo se utiliza.
- Ahora bien, ¿qué significa *grande* o *chico*? Debemos dar alguna medida del tamaño del problema.

⚠ A los efectos del análisis de algoritmos tomaremos como medida del problema **al tamaño de la entrada**, concepto que precisaremos más adelante.

⚠ Dado que para distintos problemas el concepto de *grande* varía, realizaremos un **análisis asintótico**: si n es el tamaño de la entrada, nos preguntaremos **cómo se comporta el algoritmo cuando $n \rightarrow \infty$** .

⚠ Para no comparar peras con manzanas, el análisis se hará sobre un **modelo de máquina (o modelo de cómputo)** de referencia.

(6) Características del modelo de cómputo

- Utilizaremos un modelo de referencia basado en una arquitectura “tradicional”. I.e, estilo Von Neumann.

(6) Características del modelo de cómputo

- Utilizaremos un modelo de referencia basado en una arquitectura “tradicional”. I.e, estilo Von Neumann.
- Medida de tiempo: número de pasos o instrucciones que ejecuta la máquina de referencia.

(6) Características del modelo de cómputo

- Utilizaremos un modelo de referencia basado en una arquitectura “tradicional”. I.e, estilo Von Neumann.
- Medida de tiempo: número de pasos o instrucciones que ejecuta la máquina de referencia.
- Medida de espacio: cantidad de posiciones de memoria en la máquina de referencia.

(7) Contando operaciones

- Ahora bien, dado un algoritmo en particular, ¿cómo sabemos cuánto tarda?

(7) Contando operaciones

- Ahora bien, dado un algoritmo en particular, ¿cómo sabemos cuánto tarda?
- Contaremos la cantidad de operaciones elementales (OE).

(7) Contando operaciones

- Ahora bien, dado un algoritmo en particular, ¿cómo sabemos cuánto tarda?
- Contaremos la cantidad de **operaciones elementales** (OE).
- ¿Qué es una OE?

(7) Contando operaciones

- Ahora bien, dado un algoritmo en particular, ¿cómo sabemos cuánto tarda?
- Contaremos la cantidad de **operaciones elementales** (OE).
- ¿Qué es una OE?
 - Aquélla que el procesador realiza en una cantidad de tiempo acotada por una constante (que no depende del tamaño de la entrada).

(7) Contando operaciones

- Ahora bien, dado un algoritmo en particular, ¿cómo sabemos cuánto tarda?
- Contaremos la cantidad de **operaciones elementales** (OE).
- ¿Qué es una OE?
 - Aquélla que el procesador realiza en una cantidad de tiempo acotada por una constante (que no depende del tamaño de la entrada).
 - Típicamente: operaciones aritméticas básicas, comparaciones lógicas, transferencia de control, asignaciones de variables de tipos básicos, etc.

(7) Contando operaciones

- Ahora bien, dado un algoritmo en particular, ¿cómo sabemos cuánto tarda?
- Contaremos la cantidad de **operaciones elementales** (OE).
- ¿Qué es una OE?
 - Aquélla que el procesador realiza en una cantidad de tiempo acotada por una constante (que no depende del tamaño de la entrada).
 - Típicamente: operaciones aritméticas básicas, comparaciones lógicas, transferencia de control, asignaciones de variables de tipos básicos, etc.
- Llamaremos $t(a, d)$ al número de OE del algoritmo a para el conjunto de datos de entrada d . Abusaremos la notación omitiendo alguno de los parámetros, según convenga.


(7) Contando operaciones

- Ahora bien, dado un algoritmo en particular, ¿cómo sabemos cuánto tarda?
- Contaremos la cantidad de **operaciones elementales** (OE).
- ¿Qué es una OE?
 - Aquélla que el procesador realiza en una cantidad de tiempo acotada por una constante (que no depende del tamaño de la entrada).
 - Típicamente: operaciones aritméticas básicas, comparaciones lógicas, transferencia de control, asignaciones de variables de tipos básicos, etc.
- Llamaremos $t(a, d)$ al número de OE del algoritmo a para el conjunto de datos de entrada d . Abusaremos la notación omitiendo alguno de los parámetros, según convenga.

 ¿Cómo se calcula?

(7) Contando operaciones


- Ahora bien, dado un algoritmo en particular, ¿cómo sabemos cuánto tarda?
- Contaremos la cantidad de **operaciones elementales** (OE).
- ¿Qué es una OE?
 - Aquélla que el procesador realiza en una cantidad de tiempo acotada por una constante (que no depende del tamaño de la entrada).
 - Típicamente: operaciones aritméticas básicas, comparaciones lógicas, transferencia de control, asignaciones de variables de tipos básicos, etc.
- Llamaremos $t(a, d)$ al número de OE del algoritmo a para el conjunto de datos de entrada d . Abusaremos la notación omitiendo alguno de los parámetros, según convenga.

 ¿Cómo se calcula?

- Vamos a definir que $t(OE) = 1$.

(7) Contando operaciones

- Ahora bien, dado un algoritmo en particular, ¿cómo sabemos cuánto tarda?
- Contaremos la cantidad de **operaciones elementales** (OE).
- ¿Qué es una OE?
 - Aquélla que el procesador realiza en una cantidad de tiempo acotada por una constante (que no depende del tamaño de la entrada).
 - Típicamente: operaciones aritméticas básicas, comparaciones lógicas, transferencia de control, asignaciones de variables de tipos básicos, etc.
- Llamaremos $t(a, d)$ al número de OE del algoritmo a para el conjunto de datos de entrada d . Abusaremos la notación omitiendo alguno de los parámetros, según convenga.

 ¿Cómo se calcula?

- Vamos a definir que $t(OE) = 1$.
- Además, $t(o_1; o_2) = t(o_1) + t(o_2)$ (el tiempo de la ejecución secuencial es la suma de los tiempos).

(8) Características del modelo de cómputo

- ¿Y si en la supercomputadora X que usa memoria de última generación y sale primera dos veces en el Top-500?


(8) Características del modelo de cómputo


- ¿Y si en la supercomputadora X que usa memoria de última generación y sale primera dos veces en el Top-500?

⚠ Si sólo es más rápida, nada cambia (lo veremos en la próxima clase).

(8) Características del modelo de cómputo


- ¿Y si en la supercomputadora X que usa memoria de última generación y sale primera dos veces en el Top-500?

 Si sólo es más rápida, nada cambia (lo veremos en la próxima clase).

 Si cambia drásticamente el costo relativo de lo que consideramos OE, podría haber una diferencia. El análisis asintótico de programas, al ser teórico, no tiene en cuenta las “situaciones particulares”. Es una (muy útil) aproximación. De todas maneras, la teoría necesaria para contemplar mejor estas situaciones se presenta en Algo III.

(9) Contando operaciones

- Ahora bien, dado un algoritmo en particular, ¿cómo sabemos cuánto tarda?
- Contaremos la cantidad de **operaciones elementales** (OE).
- ¿Qué es una OE?
 - Aquélla que el procesador realiza en una cantidad de tiempo acotada por una constante (que no depende del tamaño de la entrada).
 - Típicamente: operaciones aritméticas básicas, comparaciones lógicas, transferencia de control, asignaciones de variables de tipos básicos, etc.
- Llamaremos $t(a, d)$ al número de OE del algoritmo a para el conjunto de datos de entrada d . Abusaremos la notación omitiendo alguno de los parámetros, según convenga.

 ¿Cómo se calcula?

- Vamos a definir que $t(OE) = 1$.
- Además, $t(o_1; o_2) = t(o_1) + t(o_2)$ (el tiempo de la ejecución secuencial es la suma de los tiempos).

(10) Contando operaciones

- Veamos un caso de búsqueda secuencial:

```
i:= 1; encontré:= false;  
while (not(encontré))  
    if (A[i]==x) then encontré:= true;  
    i++;
```

(10) Contando operaciones

- Veamos un caso de búsqueda secuencial:

```
i := 1; encontré := false;  
while (not(encontré))  
    if (A[i] == x) then encontré := true;  
    i++;
```

- ¿Cuánto tarda?

(10) Contando operaciones

- Veamos un caso de búsqueda secuencial:

```
i := 1; encontré := false;  
while (not(encontré))  
    if (A[i] == x) then encontré := true;  
    i++;
```

- ¿Cuánto tarda?


 Otra suposición que haremos (por ahora) es que nos interesa el peor caso.

(10) Contando operaciones

- Veamos un caso de búsqueda secuencial:

```
i := 1; encontré := false;  
while (not(encontré))  
    if (A[i] == x) then encontré := true;  
    i++;
```

- ¿Cuánto tarda?

 Otra suposición que haremos (por ahora) es que nos interesa el **peor caso**.

 Estructuras de control:

(10) Contando operaciones

- Veamos un caso de búsqueda secuencial:

```
i := 1; encontré := false;  
while (not(encontré))  
    if (A[i] == x) then encontré := true;  
    i++;
```

- ¿Cuánto tarda?

⚠ Otra suposición que haremos (por ahora) es que nos interesa el **peor caso**.

⚠ Estructuras de control:

- $t(\text{case } E : v_1 : C_1; \dots; v_k : C_k) = t(E) + k + \text{máx}(t(C_1), \dots, t(C_k)).$

(10) Contando operaciones

- Veamos un caso de búsqueda secuencial:

```
i := 1; encontré := false;  
while (not(encontré))  
    if (A[i] == x) then encontré := true;  
    i++;
```

- ¿Cuánto tarda?

⚠ Otra suposición que haremos (por ahora) es que nos interesa el **peor caso**.

⚠ Estructuras de control:

- $t(\text{case } E : v_1 : C_1; \dots; v_k : C_k) = t(E) + k + \text{máx}(t(C_1), \dots, t(C_k)).$
- Un if es un case con dos condiciones.

(10) Contando operaciones

- Veamos un caso de búsqueda secuencial:

```
i := 1; encontré := false;  
while (not(encontré))  
    if (A[i] == x) then encontré := true;  
    i++;
```

- ¿Cuánto tarda?

⚠ Otra suposición que haremos (por ahora) es que nos interesa el **peor caso**.

⚠ Estructuras de control:

- $t(\text{case } E : v_1 : C_1; \dots; v_k : C_k) = t(E) + k + \max(t(C_1), \dots, t(C_k)).$
- Un if es un case con dos condiciones.
- $t(\text{while } G \text{ do } C \text{ od}) = \sum_{i=1 \dots k} t(G_i) + t(C_i),$ donde k es la cantidad de iteraciones.

(10) Contando operaciones

- Veamos un caso de búsqueda secuencial:

```
i := 1; encontré := false;  
while (not(encontré))  
    if (A[i]==x) then encontré := true;  
    i++;
```

- ¿Cuánto tarda?

⚠ Otra suposición que haremos (por ahora) es que nos interesa el **peor caso**.

⚠ Estructuras de control:

- $t(\text{case } E : v_1 : C_1; \dots; v_k : C_k) = t(E) + k + \max(t(C_1), \dots, t(C_k)).$
- Un if es un case con dos condiciones.
- $t(\text{while } G \text{ do } C \text{ od}) = \sum_{i=1 \dots k} t(G_i) + t(C_i)$, donde k es la cantidad de iteraciones.

⚠ La expresión es equivalente a $k \times (t(G) + t(C))$ sólo si G y C no varían en cada iteración.

(11) Contando operaciones (cont.)

- Más reglas para el cálculo de OE:

(11) Contando operaciones (cont.)

- Más reglas para el cálculo de OE:
 - Para calcular el resto de las formas de repetición alcanzar con llevarlas la forma de while.

(11) Contando operaciones (cont.)

- Más reglas para el cálculo de OE:
 - Para calcular el resto de las formas de repetición alcanzar con llevarlas la forma de while.
 - $t(F(p_1, \dots, p_k)) = 1 + t(p_1) + \dots + t(p_k) + t(F)$ (es decir, la llamada a función, la evaluación de los parámetros y luego el tiempo de la función propiamente dicha).

(11) Contando operaciones (cont.)

- Más reglas para el cálculo de OE:
 - Para calcular el resto de las formas de repetición alcanzar con llevarlas la forma de while.
 - $t(F(p_1, \dots, p_k)) = 1 + t(p_1) + \dots + t(p_k) + t(F)$ (es decir, la llamada a función, la evaluación de los parámetros y luego el tiempo de la función propiamente dicha).
 - $t(p_i)$ es 1 si p_i se trata de un puntero o un tipo básico. Si es un arreglo o un *record*, contaremos la cantidad de elementos básicos que contenga.

(11) Contando operaciones (cont.)

- Más reglas para el cálculo de OE:
 - Para calcular el resto de las formas de repetición alcanzar con llevarlas la forma de while.
 - $t(F(p_1, \dots, p_k)) = 1 + t(p_1) + \dots + t(p_k) + t(F)$ (es decir, la llamada a función, la evaluación de los parámetros y luego el tiempo de la función propiamente dicha).
 - $t(p_i)$ es 1 si p_i se trata de un puntero o un tipo básico. Si es un arreglo o un *record*, contaremos la cantidad de elementos básicos que contenga.
 - El tiempo de ejecución de las llamadas a procedimientos recursivos va a dar lugar a ecuaciones en recurrencia, que veremos posteriormente.

(12) Complejidad temporal

- Si queremos predecir, no nos va a interesar tanto la complejidad para una instancia en particular, si no más bien para una clase de instancias.

(12) Complejidad temporal

- Si queremos predecir, no nos va a interesar tanto la complejidad para una instancia en particular, si no más bien para una clase de instancias.
- ¿Cómo armaremos las clases? Por tamaño de la entrada.

(12) Complejidad temporal

- Si queremos predecir, no nos va a interesar tanto la complejidad para una instancia en particular, si no más bien para una clase de instancias.
- ¿Cómo armaremos las clases? Por tamaño de la entrada.
- Modificando un poco la notación, la forma de expresar el costo en tiempo del algoritmo para una entrada de tamaño n será $T(n)$.

(12) Complejidad temporal

- Si queremos predecir, no nos va a interesar tanto la complejidad para una instancia en particular, si no más bien para una clase de instancias.
- ¿Cómo armaremos las clases? Por tamaño de la entrada.
- Modificando un poco la notación, la forma de expresar el **costo en tiempo** del algoritmo para una entrada de tamaño n será $T(n)$.
- Pero distintas instancias, aún coincidiendo en tamaño, pueden hacer que el algoritmo se comporte de maneras muy diferentes.

(12) Complejidad temporal

- Si queremos predecir, no nos va a interesar tanto la complejidad para una instancia en particular, si no más bien para una clase de instancias.
- ¿Cómo armaremos las clases? Por tamaño de la entrada.
- Modificando un poco la notación, la forma de expresar el **costo en tiempo** del algoritmo para una entrada de tamaño n será $T(n)$.
- Pero distintas instancias, aún coincidiendo en tamaño, pueden hacer que el algoritmo se comporte de maneras muy diferentes.

 Por eso, distinguiremos mejor caso, peor caso y caso promedio.

(13) Peor caso

- Llamemos I al conjunto de instancias posibles de nuestro algoritmo (variaciones de los parámetros de entrada).

(13) Peor caso


- Llamemos I al conjunto de instancias posibles de nuestro algoritmo (variaciones de los parámetros de entrada).
- $T_{peor}(n) = \max_{i \in I \mid |i|=n} (t(i))$

(13) Peor caso

- Llamemos I al conjunto de instancias posibles de nuestro algoritmo (variaciones de los parámetros de entrada).
- $T_{\text{peor}}(n) = \max_{i \in I \mid |i|=n} (t(i))$
- Intuitivamente, $T_{\text{peor}}(n)$ es el mayor tiempo que puede tomar el algoritmo sobre una instancia de tamaño n .

(13) Peor caso

- Llamemos I al conjunto de instancias posibles de nuestro algoritmo (variaciones de los parámetros de entrada).
- $T_{\text{peor}}(n) = \max_{i \in I \mid |i|=n} (t(i))$
- Intuitivamente, $T_{\text{peor}}(n)$ es el mayor tiempo que puede tomar el algoritmo sobre una instancia de tamaño n .

 La ventaja de la complejidad de peor caso es que sabemos que el algoritmo no se va a comportar peor que eso. Es decir, tenemos una garantía sobre su comportamiento.

(14) Mejor caso

- La definición de $T_{\text{mejor}}(n)$ es análoga.

(14) Mejor caso

- La definición de $T_{\text{mejor}}(n)$ es análoga.
- No es demasiado interesante.

(15) Caso promedio

- ¿Cómo se imaginan la definición de $T_{prom}(n)$?

(15) Caso promedio

- ¿Cómo se imaginan la definición de $T_{prom}(n)$?
- Idea intuitiva: tiempo “promedio”, o “esperado” sobre instancias “típicas” (suponiendo que tales cosas existan...).

(15) Caso promedio

- ¿Cómo se imaginan la definición de $T_{prom}(n)$?
- Idea intuitiva: tiempo “promedio”, o “esperado” sobre instancias “típicas” (suponiendo que tales cosas existan...).
- Se define como la esperanza matemática de la variable aleatoria definida por todas las posibles ejecuciones del algoritmo para un tamaño de la entrada dado, multiplicado por las probabilidades de que éstas ocurran para esa entrada.

(15) Caso promedio

- ¿Cómo se imaginan la definición de $T_{prom}(n)$?
- Idea intuitiva: tiempo “promedio”, o “esperado” sobre instancias “típicas” (suponiendo que tales cosas existan...).
- Se define como la esperanza matemática de la variable aleatoria definida por todas las posibles ejecuciones del algoritmo para un tamaño de la entrada dado, multiplicado por las probabilidades de que éstas ocurran para esa entrada.
- $T_{prom}(n) = \sum_{i \in I \mid |i|=n} P(i) \cdot t(i)$

(15) Caso promedio

- ¿Cómo se imaginan la definición de $T_{prom}(n)$?
- Idea intuitiva: tiempo “promedio”, o “esperado” sobre instancias “típicas” (suponiendo que tales cosas existan...).
- Se define como la esperanza matemática de la variable aleatoria definida por todas las posibles ejecuciones del algoritmo para un tamaño de la entrada dado, multiplicado por las probabilidades de que éstas ocurran para esa entrada.
- $T_{prom}(n) = \sum_{i \in I \mid |i|=n} P(i) \cdot t(i)$
- Algunas precauciones:

(15) Caso promedio

- ¿Cómo se imaginan la definición de $T_{prom}(n)$?
- Idea intuitiva: tiempo “promedio”, o “esperado” sobre instancias “típicas” (suponiendo que tales cosas existan...).
- Se define como la esperanza matemática de la variable aleatoria definida por todas las posibles ejecuciones del algoritmo para un tamaño de la entrada dado, multiplicado por las probabilidades de que éstas ocurran para esa entrada.
- $T_{prom}(n) = \sum_{i \in I \mid |i|=n} P(i) \cdot t(i)$
- Algunas precauciones:
 - Requiere conocer la distribución estadística de los datos de entrada: en muchos casos eso no es realista.

(15) Caso promedio

- ¿Cómo se imaginan la definición de $T_{prom}(n)$?
- Idea intuitiva: tiempo “promedio”, o “esperado” sobre instancias “típicas” (suponiendo que tales cosas existan...).
- Se define como la esperanza matemática de la variable aleatoria definida por todas las posibles ejecuciones del algoritmo para un tamaño de la entrada dado, multiplicado por las probabilidades de que éstas ocurran para esa entrada.
- $T_{prom}(n) = \sum_{i \in I \mid |i|=n} P(i) \cdot t(i)$
- Algunas precauciones:
 - Requiere conocer la distribución estadística de los datos de entrada: en muchos casos eso no es realista.
 - En muchos casos la matemática se complica, y se terminan haciendo hipótesis simplificadoras poco realistas.

(15) Caso promedio

- ¿Cómo se imaginan la definición de $T_{prom}(n)$?
- Idea intuitiva: tiempo “promedio”, o “esperado” sobre instancias “típicas” (suponiendo que tales cosas existan...).
- Se define como la esperanza matemática de la variable aleatoria definida por todas las posibles ejecuciones del algoritmo para un tamaño de la entrada dado, multiplicado por las probabilidades de que éstas ocurran para esa entrada.
- $T_{prom}(n) = \sum_{i \in I \mid |i|=n} P(i) \cdot t(i)$
- Algunas precauciones:
 - Requiere conocer la distribución estadística de los datos de entrada: en muchos casos eso no es realista.
 - En muchos casos la matemática se complica, y se terminan haciendo hipótesis simplificadoras poco realistas.
 - Podemos tener algoritmos para los cuales ninguna entrada requiere tiempo medio (por ejemplo, un algoritmo que requiere o bien 1 o bien 100 pasos).

(16) Poniéndolo en práctica

- Recordemos:

```
i:= 1; encontré:= false;  
while (not(encontré))  
  if (A[i]==x) then encontré:= true;  
  i++;
```

(16) Poniéndolo en práctica

- Recordemos:

```
i := 1; encontré := false;  
while (not(encontré))  
  if (A[i] == x) then encontré := true;  
  i++;
```

- ¿Mejor caso?

(16) Poniéndolo en práctica

- Recordemos:

```
i := 1; encontré := false;  
while (not(encontré))  
    if (A[i] == x) then encontré := true;  
    i++;
```

- ¿Mejor caso?
- ¿Peor caso?

(16) Poniéndolo en práctica

- Recordemos:

```
i := 1; encontré := false;  
while (not(encontré))  
  if (A[i] == x) then encontré := true;  
  i++;
```

- ¿Mejor caso?
- ¿Peor caso?
- ¿Caso promedio?

(16) Poniéndolo en práctica

- Recordemos:

```
i := 1; encontré := false;  
while (not(encontré))  
    if (A[i] == x) then encontré := true;  
    i++;
```

- ¿Mejor caso?
- ¿Peor caso?
- ¿Caso promedio?
- ¿Y si estuviera ordenado?

(17) Principio de invarianza

- Dado un algoritmo y dos máquinas (o dos implementaciones) M_1 y M_2 , que tardan $T_1(n)$ y $T_2(n)$ respectivamente sobre entradas de tamaño n , existen una constante real positiva c y un $n_0 \in \mathbb{N}$ tales que
 $(\forall n \geq n_0) \quad T_1(n) \leq cT_2(n)$

(17) Principio de invarianza

- Dado un algoritmo y dos máquinas (o dos implementaciones) M_1 y M_2 , que tardan $T_1(n)$ y $T_2(n)$ respectivamente sobre entradas de tamaño n , existen una constante real positiva c y un $n_0 \in \mathbb{N}$ tales que
$$(\forall n \geq n_0) \quad T_1(n) \leq cT_2(n)$$
- Idea: dos ejecuciones distintas del mismo algoritmo sólo difieren en cuanto a eficiencia en un factor constante para valores de la entradas suficientemente grandes.

(18) Recapitulando...

- Argumentamos a favor de un análisis teórico (en lugar de empírico) como método para comparar algoritmos en base a su tiempo de ejecución.

(18) Recapitulando...

- Argumentamos a favor de un análisis teórico (en lugar de empírico) como método para comparar algoritmos en base a su tiempo de ejecución.
 - El método era asintótico.

(18) Recapitulando...

- Argumentamos a favor de un análisis teórico (en lugar de empírico) como método para comparar algoritmos en base a su tiempo de ejecución.
 - El método era asintótico.
 - Y se basaba en el tamaño de la entrada.

(18) Recapitulando...

- Argumentamos a favor de un análisis teórico (en lugar de empírico) como método para comparar algoritmos en base a su tiempo de ejecución.
 - El método era asintótico.
 - Y se basaba en el tamaño de la entrada.
- Acordamos utilizar como marco de referencia una máquina ideal y dimos un cálculo para establecer el costo de ejecutar cada instrucción.

(18) Recapitulando...

- Argumentamos a favor de un análisis teórico (en lugar de empírico) como método para comparar algoritmos en base a su tiempo de ejecución.
 - El método era asintótico.
 - Y se basaba en el tamaño de la entrada.
- Acordamos utilizar como marco de referencia una máquina ideal y dimos un cálculo para establecer el costo de ejecutar cada instrucción.
- Como nos interesaba hablar de la complejidad de clases de instancias, pasamos a una notación que expresa la complejidad en base al tamaño de la entrada: $T(n)$.

(18) Recapitulando...

- Argumentamos a favor de un análisis teórico (en lugar de empírico) como método para comparar algoritmos en base a su tiempo de ejecución.
 - El método era asintótico.
 - Y se basaba en el tamaño de la entrada.
- Acordamos utilizar como marco de referencia una máquina ideal y dimos un cálculo para establecer el costo de ejecutar cada instrucción.
- Como nos interesaba hablar de la complejidad de **clases** de instancias, pasamos a una notación que expresa la complejidad en base al tamaño de la entrada: $T(n)$.
- **Vimos que debíamos diferenciar al menos los casos promedio, mejor y peor.**

(19) En la próxima clase...

- Veremos una forma de aproximar $T(n)$: la notación “ $O(n)$ ”.

(19) En la próxima clase...

- Veremos una forma de aproximar $T(n)$: la notación “ $O(n)$ ”.
- Miraremos más en detalle qué significa *tamaño de la entrada*.

(19) En la próxima clase...

- Veremos una forma de aproximar $T(n)$: la notación “ $O(n)$ ”.
- Miraremos más en detalle qué significa *tamaño de la entrada*.
- Aportaremos más elementos para comparar la eficiencia de los algoritmos.

- Data Structures and Algorithms. Alfred V. Aho, Jeffrey D. Ullman, John E. Hopcroft. Addison-Wesley.

- Data Structures and Algorithms. Alfred V. Aho, Jeffrey D. Ullman, John E. Hopcroft. Addison-Wesley.
- Introduction to Algorithms, Second Edition. Thomas H. Cormen, Charles E. Leiserson and Ronald L. Rivest. MIT Press, 2001.