Introducción

IIC2283

¿A qué nos dedicamos en ciencia de la computación?

Construcción de algoritmos

¿A qué nos dedicamos en ciencia de la computación?

Construcción de algoritmos

- Diseño de algoritmos eficientes para resolver distintos tipos de problemas

¿A qué nos dedicamos en ciencia de la computación?

Construcción de algoritmos

- Diseño de algoritmos eficientes para resolver distintos tipos de problemas
- Construcción de cotas inferiores para el tiempo (u otro recurso relevante como espacio) necesario para solucionar un problema

¿A qué nos dedicamos en ciencia de la computación?

Construcción de algoritmos

- Diseño de algoritmos eficientes para resolver distintos tipos de problemas
- Construcción de cotas inferiores para el tiempo (u otro recurso relevante como espacio) necesario para solucionar un problema
  - Demostración de que hay problemas que no pueden ser resueltos de manera eficiente

¿A qué nos dedicamos en ciencia de la computación?

Construcción de algoritmos

- Diseño de algoritmos eficientes para resolver distintos tipos de problemas
- Construcción de cotas inferiores para el tiempo (u otro recurso relevante como espacio) necesario para solucionar un problema
  - Demostración de que hay problemas que no pueden ser resueltos de manera eficiente
- Implementación eficiente de los algoritmos diseñados en un modelo de computación

Un ejemplo fundamental

¿Es el siguiente número un primo?

5943632362508814456797384433006100442712303295066940614569354936549874999082678378231629906729379134167935471382621311620276545251597436711454168850267595109678077983960376792735878876067066338864239372227790339203350191408856924700453890622245349547304896138668552188577288047417779378700982792791819866553113608966810109430765067528429902116607213626746562172730714525439765422832045628189761714003

Un ejemplo fundamental

¿Y este otro número es primo?

3535588911865602415079554504431000263505208333291756909315030
0691037865228817153549425096333001396540623829655658480790004
9168946066529591899268842681841264312295940086851989180358466
9131979917796954324319424965331526025027230674464475670993233
3816841127167397732293501581464489649729208070784814646152966
8102096255734221522359602690212915438950894346502358310134422
7385618229213567413216451747789163148428889293747005855150511
1093864750291988918772395766020579586089685266938407069378266
7566775279430718312544618295454956836631931739908235538749599
9487230769334744346813430561559045156991978005711474172160477
3783655804311797807355647117962261735664667387340020887101122
9285301762159583258409784425931812461681678714227281236534677
1199832023109969143877030654186623320201933709744657744458355
967694968192132697333

La complejidad de primalidad

¿Puede un algoritmo verificar si un número es primo revisando sus posibles divisores?

- Hagamos unos cálculos simples

La complejidad de primalidad

¿Puede un algoritmo verificar si un número es primo revisando sus posibles divisores?

- Hagamos unos cálculos simples

Y ahora ejecutemos en Python un algoritmo para verificar si un número es primo.

La complejidad de primalidad

¿Puede un algoritmo verificar si un número es primo revisando sus posibles divisores?

- Hagamos unos cálculos simples

Y ahora ejecutemos en Python un algoritmo para verificar si un número es primo.

- ¿Cómo funciona este algoritmo?

La complejidad de primalidad

¿Puede un algoritmo verificar si un número es primo revisando sus posibles divisores?

- Hagamos unos cálculos simples

Y ahora ejecutemos en Python un algoritmo para verificar si un número es primo.

- ¿Cómo funciona este algoritmo?

El test de primalidad usado es el resultado de una combinación de técnicas modernas para el diseño de algoritmos.

La complejidad de primalidad

¿Puede un algoritmo verificar si un número es primo revisando sus posibles divisores?

- Hagamos unos cálculos simples

Y ahora ejecutemos en Python un algoritmo para verificar si un número es primo.

- ¿Cómo funciona este algoritmo?

El test de primalidad usado es el resultado de una combinación de técnicas modernas para el diseño de algoritmos.

- En particular el algoritmo es aleatorizado: hay una probabilidad de error de a lo más $\frac{1}{2^{100}} \approx 7.9 \times 10^{-31}$

El objetivo de este curso

Introducir técnicas tanto para el diseño como para el análisis de la complejidad computacional de un algoritmo

El objetivo de este curso

Introducir técnicas tanto para el diseño como para el análisis de la complejidad computacional de un algoritmo

- Técnicas básicas y avanzadas

El objetivo de este curso

Introducir técnicas tanto para el diseño como para el análisis de la complejidad computacional de un algoritmo

- Técnicas básicas y avanzadas

Se dará énfasis a:

- La compresión del modelo computacional sobre el cual se diseña y analiza un algoritmo
- El uso de ejemplos de distintas áreas para mostrar las potencialidades de las técnicas estudiadas

Algunas consideraciones importantes

Al diseñar un algoritmo debemos considerar el modelo de computación sobre el cual será implementado.

- ¿Qué operaciones podemos realizar en el modelo?

Algunas consideraciones importantes

Al diseñar un algoritmo debemos considerar el modelo de computación sobre el cual será implementado.

- ¿Qué operaciones podemos realizar en el modelo?

Al analizar la complejidad computacional de un algoritmo también debemos considerar el modelo de computación.

- ¿Qué operaciones consideramos al analizar la complejidad de un algoritmo?

Algunas consideraciones importantes

En general, el análisis de la complejidad de un algoritmo se realiza considerando un tipo particular de entradas.

- El peor caso es muy utilizado, pero también podemos considerar el caso promedio

Algunas consideraciones importantes

En general, el análisis de la complejidad de un algoritmo se realiza considerando un tipo particular de entradas.

- El peor caso es muy utilizado, pero también podemos considerar el caso promedio

Al estudiar un problema debemos tener en cuenta que cotas inferiores se puede demostrar para su complejidad

- Estas cotas inferiores dependen del modelo de computación considerado

Algunas consideraciones importantes

Debemos considerar modelos de computación que representen el funcionamiento de una arquitectura de computadores.

- Por ejemplo, debemos consider acceso directo a los datos y la diferencia de costo entre el uso de memoria principal y secundaria

Algunas consideraciones importantes

Debemos considerar modelos de computación que representen el funcionamiento de una arquitectura de computadores.

- Por ejemplo, debemos consider acceso directo a los datos y la diferencia de costo entre el uso de memoria principal y secundaria

Vamos a considerar un ejemplo que nos servirán para ilustrar los puntos anteriores: ordenación

Ordenación de una lista

El siguiente es una algoritmo clásico para ordenar una lista L de números enteros (de menor a mayor).

InsertionSort($L[1 \dots n]$: lista de números enteros)
    for $i := 2$ **to** n **do**
        $j := i - 1$
        while $j \geq 1$ **and** $L[j] > L[j+1]$ **do**
            $aux := L[j]$
            $L[j] := L[j+1]$
            $L[j+1] := aux$
            $j := j - 1$
    return L

Análisis de la complejidad de insertion sort

Consideramos la comparación como la operación a contar, la cual tiene costo 1.

Análisis de la complejidad de insertion sort

Consideramos la comparación como la operación a contar, la cual tiene costo 1.

Dada una lista L con n números enteros.

Análisis de la complejidad de insertion sort

Consideramos la comparación como la operación a contar, la cual tiene costo 1.

Dada una lista L con n números enteros.

- ¿Cuál es el peor caso del algoritmo?

Análisis de la complejidad de insertion sort

Consideramos la comparación como la operación a contar, la cual tiene costo 1.

Dada una lista L con n números enteros.

- ¿Cuál es el peor caso del algoritmo?
- ¿Cuantas comparaciones realiza el algoritmo en el peor caso, como una función de n?

Análisis de la complejidad de insertion sort

Consideramos la comparación como la operación a contar, la cual tiene costo 1.

Dada una lista L con n números enteros.

- ¿Cuál es el peor caso del algoritmo?
- ¿Cuantas comparaciones realiza el algoritmo en el peor caso, como una función de n?

¿Qué otras operaciones son realizadas por el algoritmo? ¿Cambia el tiempo de ejecución del algoritmo si las consideramos?

Una versión mejorada de insertion sort

Suponiendo que $L[1 \cdots (i-1)]$ ya ha sido ordenada (de menor a mayor), el paso básico del algoritmo es encontrar la posición donde debería ser ubicado $L[i]$

- Además el algoritmo debe colocar $L[i]$ en esta posición

Una versión mejorada de insertion sort

Suponiendo que $L[1 \cdots (i-1)]$ ya ha sido ordenada (de menor a mayor), el paso básico del algoritmo es encontrar la posición donde debería ser ubicado $L[i]$

- Además el algoritmo debe colocar $L[i]$ en esta posición

Para disminuir el tiempo de ejecución del algoritmo podríamos utilizar búsqueda binaria para encontrar la posición correcta para $L[i]$

Una versión mejorada de insertion sort

Consideramos nuevamente la comparación como la operación a contar.

Una versión mejorada de insertion sort

Consideramos nuevamente la comparación como la operación a contar.

Dado que búsqueda binaria realiza $O(\log_2(i))$ comparaciones para encontrar la posición correcta para $L[i]$, el algoritmo mejorado es de orden $O(n \cdot \log_2(n))$

Una versión mejorada de insertion sort

Consideramos nuevamente la comparación como la operación a contar.

Dado que búsqueda binaria realiza $O(\log_2(i))$ comparaciones para encontrar la posición correcta para $L[i]$, el algoritmo mejorado es de orden $O(n \cdot \log_2(n))$

- Esto puede ser deducido utilizando lo siguiente:

$$\begin{aligned}
\sum_{i=1}^{n} \log_2(i) &= \log_2\left(\prod_{i=1}^{n} i\right) \\
&= \log_2(n!) \\
&\leq \log_2(n^n) \\
&= n \cdot \log_2(n)
\end{aligned}$$

¿Una versión mejorada de insertion sort?

¿Es más eficiente el algoritmo que utiliza búsqueda binaria?

- ¿Ve algún problema en este algoritmo?

¿Una versión mejorada de insertion sort?

¿Es más eficiente el algoritmo que utiliza búsqueda binaria?

- ¿Ve algún problema en este algoritmo?

Un problema con este algoritmo: ¿Cómo colocar de manera eficiente $L[i]$ en la posición correcta en $L[1 \ldots (i-1)]$?

¿Una versión mejorada de insertion sort?

¿Es más eficiente el algoritmo que utiliza búsqueda binaria?

- ¿Ve algún problema en este algoritmo?

Un problema con este algoritmo: ¿Cómo colocar de manera eficiente $L[i]$ en la posición correcta en $L[1 \ldots (i-1)]$?

- Un algoritmo ingenuo toma tiempo lineal en este paso, por lo que el tiempo total del algoritmo sería $O(n^2)$, el mismo orden que para insertion sort

¿Una versión mejorada de insertion sort?

Dos lecciones importantes

- Una operación puede ser cambiada por otra en un algoritmo, esto debe tenerse en cuenta al momento de analizar su complejidad.
- El uso de estructuras de datos es fundamental para implementar de manera eficiente las operaciones requeridas por un algoritmo.