

Diseño de Algoritmos - Algoritmos II

Nazareno Aguirre, Sonia Permigiani, Gastón Scilingo,
Simón Gutiérrez

Departamento de Computación

Facultad de Ciencias Exactas, Físico-Químicas y Naturales
Universidad Nacional de Río Cuarto

Clase 1(b): Introducción al Diseño de Algoritmos

Un Ejemplo: Cálculo del Máximo Común Divisor

A modo de ejemplo, podemos considerar el problema de calcular el máximo común divisor, y un algoritmo básico para resolverlo.

Problema: Dados dos números naturales positivos x e y , calcular el mayor número natural d que divide a x e y .

Algoritmo:

Paso 1. Obtener r , el mínimo entre x e y

Paso 2. Si r divide a x e y , retornar r .

Paso 3. Sino, decrementar r en 1 y volver al paso 2.

La Noción de Algoritmo

Existen muchas definiciones (algunas no muy precisas) de la noción de algoritmo. Una de ellas, que se ajusta a nuestro uso de esta noción, es la siguiente:

un algoritmo es una secuencia finita de instrucciones precisas (i.e., no ambiguas), usadas con el objetivo de resolver un problema.

Con frecuencia, el problema puede expresarse en términos de entradas (parámetros del problema), y la solución del problema como la obtención de una salida, el resultado de la resolución del problema.

La precisión de las instrucciones está relacionada a que éstas deben poder ser llevadas a cabo por un individuo o un dispositivo (por ejemplo, una computadora).

Un Ejemplo: Cálculo del Máximo Común Divisor

Como programadores, estamos acostumbrados a describir algoritmos mediante código o pseudo-código:

```
r ← MIN(x,y)
while x mod r != 0 or y mod r != 0 do
    begin
        r ← r-1
    end
return r
```

Soluciones Alternativas

Los problemas que admiten soluciones algorítmicas suelen poder resolverse de muchas formas alternativas. Por ejemplo, el problema de calcular el máximo común divisor entre dos números puede resolverse de otras maneras:

Algoritmo de Euclides

Paso 1. Si $y = 0$, retornar x como resultado y terminar. Caso contrario, continuar con el paso 2.

Paso 2. Dividir x por y , obtener el resto r de la división. Continuar con el paso 3.

Paso 3. Volver al paso 1 para calcular el máximo común divisor, esta vez de y y r .

Basado en Factorización de Enteros

Paso 1. Encontrar la factorización de x en términos de números primos.

Paso 2. Encontrar la factorización de y en términos de números primos.

Paso 3. Tomar los factores primos comunes a x e y

Paso 4. Calcular el producto de todos los factores primos comunes a x e y , y retornar esto como resultado

Abstracción y Refinamiento en la Descripción de Algoritmos

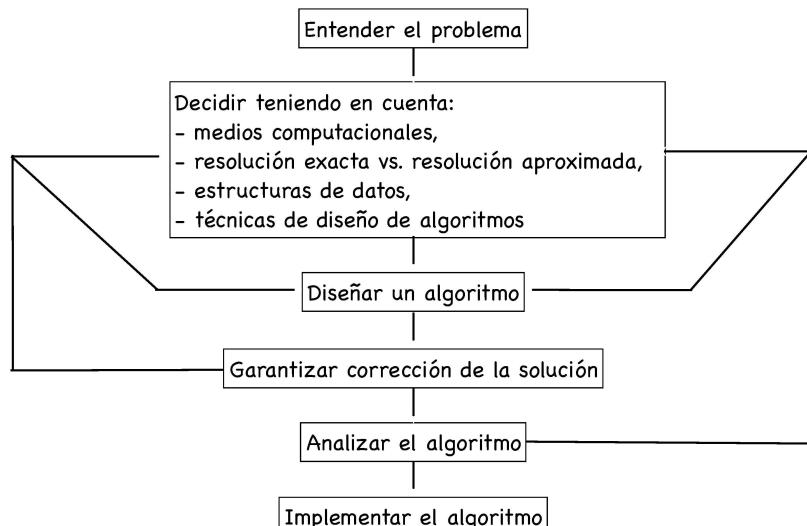
La descripción del proceso para computar el máximo común divisor basado en factorización de enteros constituye la descripción de un algoritmo sólo si contamos con un algoritmo para computar la factorización en términos de números primos. Es decir, constituye la descripción de un proceso abstracto, que necesita ser refinado para poder ser considerado un algoritmo.

Para calcular la factorización en términos de números primos de un entero, podríamos por ejemplo utilizar un algoritmo basado en la Criba de Eratóstenes:

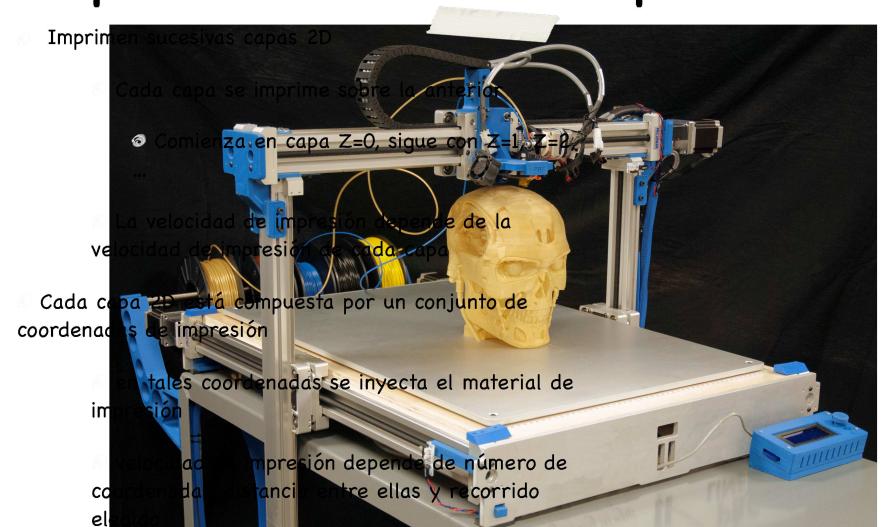
```
Input: Entero  $n \geq 2$ . Output: Lista de primos menores o iguales que  $n$ 
for  $p \leftarrow 2$  to  $n$  do  $A[p] \leftarrow p$ 
for  $p \leftarrow 2$  to  $\lfloor \sqrt{n} \rfloor$  do
    if  $A[p] \neq 0$  //  $p$  no ha sido eliminado previamente de la lista
         $j \leftarrow p * p$ 
        while  $j \leq n$  do
             $A[j] \leftarrow 0$  // marcar el elemento como eliminado
             $j \leftarrow j + p$ 
```

Transformar el proceso/ algoritmo original con una descripción más detallada de las tareas que la componen se denomina refinamiento

El Proceso de Diseño y Análisis de Algoritmos



Ejemplo: Recorrido de impresión en una impresora 3D



Ejemplo: Recorrido de impresión en una impresora 3D

A modo de ejemplo, ilustrando las diferentes cuestiones a tener en cuenta en la resolución algorítmica de problemas, consideremos el siguiente problema:

Dado un conjunto P de puntos del plano, elegir una forma de recorrerlos de manera que la distancia global recorrida sea la mínima

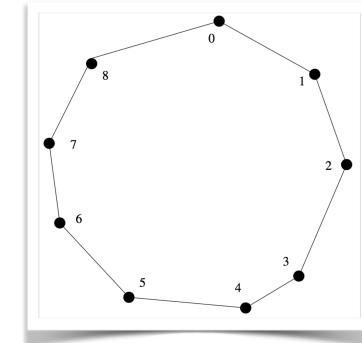
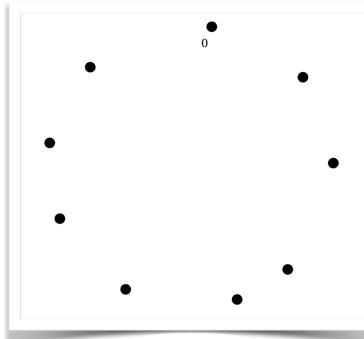
Algunas preguntas que se pueden hacer para comprender mejor el problema son las siguientes:

¿hay algún punto de inicio específico? NO

¿cómo son los movimientos en el plano? En línea recta del origen al destino

¿hay algún punto final específico? Misma posición que punto de salida

Ejemplos de Conjuntos de Puntos y Recorridos Esperados



Recorrido Óptimo de Impresión

Primera Propuesta

Una propuesta para resolver el problema del recorrido óptimo de impresión es elegir siempre el punto vecino no visitado más próximo al punto actual.

RecorridoOptimo(P : Cto. de n puntos en el plano)

 elegir un punto p_0 de P como inicial

 visitar p_0

$i := 1$

while $i < n$ **do**

begin

 elegir punto p_i no visitado más próximo a $p(i-1)$

 visitar p_i

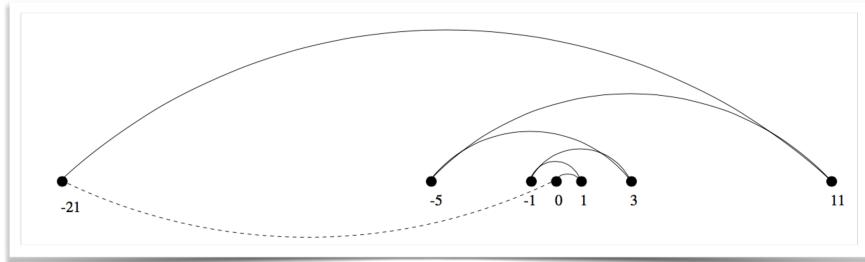
$i := i + 1$

end

 retornar a p_0 (desde $p(n-1)$)

¿Resuelve el problema correctamente?

Mala Instancia de Vecino más Próximo



Recorrido Óptimo de Impresión

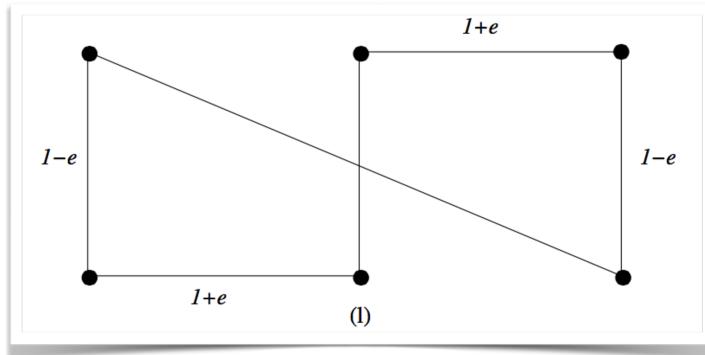
Segunda Propuesta

Una segunda propuesta para resolver el problema del recorrido óptimo de impresión es armar el recorrido en base a pares de puntos más próximos entre sí.

```
RecorridoOptimo'(P: Cto. de n puntos en el plano)
  for i := 1 to n-1 do
    begin
      d := infinity
      for (s,t) en cadenas de vértices distintas do
        begin
          if dist(s,t) <= d then
            begin
              d := dist(s,t)
              sm := s
              tm := t
            end
        end
      conectar s y t con un arco
      conectar las dos puntas con un arco
```

¿Resuelve el problema correctamente?

Mala Instancia de Pares más Próximos



Recorrido Óptimo de Impresión

Tercera Propuesta

Una tercera propuesta, sin dudas correcta, consiste en intentar todas las permutaciones, y quedarse con la mejor.

```
RecorridoOptimo"(P: Cto. de n puntos en el plano)
  d := infinity
  for cada permutación Pi de P do
    if cost(Pi) <= d then
      begin
        d := cost(Pi)
        Pmin := Pi
      end
    end
  retornar Pmin
```

Desafortunadamente, esta solución es muy costosa en tiempo. Para 20 puntos, tiene que considerar 2432902008176640000 combinaciones.

Resolviendo el Problema de Recorrido Óptimo

El problema de determinar un recorrido óptimo ha sido estudiado durante mucho tiempo.

Desafortunadamente, este problema pertenece a la clase de problemas NP-completos. Para estos problemas, como veremos más adelante en la materia, las únicas soluciones óptimas conocidas son del tipo "probar todas las posibilidades".

En nuestro caso, probar todas las posibilidades consiste en probar con todas las posibles permutaciones de nodos.

No es difícil darse cuenta que este tipo de procedimiento es computacionalmente costoso, con lo cual sólo es viable para conjuntos pequeños de puntos.

Alternativas a “Probar todas las Posibilidades”

La primera pregunta que deberíamos hacernos es:

nuestro programa se usará sólo con conjuntos pequeños?

Si es así, podríamos intentar usar directamente el proceso de “probar todas las posibilidades”. Si esto resulta ser demasiado ineficiente para nuestro caso, tenemos (al menos) las siguientes alternativas:

Buscar información adicional sobre el problema en cuestión. Información adicional sobre el dominio del problema podría evitarnos tener que probar todas las posibilidades.

Buscar una solución buena al problema, aunque no necesariamente óptima. Dada la complejidad del problema original, podríamos quizás conformarnos si consiguiéramos una solución que eficientemente nos brindara un recorrido próximo al óptimo (aunque no fuera necesariamente mínimo). Las dos alternativas vistas al comienzo son ejemplos de estas opciones.

Refinando las Heurísticas para el Recorrido Óptimo

Para conseguir versiones más cercanas a una implementación a partir de las descripciones anteriores para resolver el problema de recorrido óptimo, necesitamos como mínimo resolver los siguientes problemas:

Decidir cómo representar conjuntos (por ejemplo, mediante AVLs o tablas hash).

Decidir cómo representar el hecho de que un punto fue visitado

Implementar la visita de pares de puntos

Implementar la construcción de permutaciones de una secuencia

Aspectos Importantes en el Diseño de Algoritmos

El ejemplo anterior ilustra algunos aspectos importantes del diseño de algoritmos, en particular:

La importancia de la formalización de problemas

La importancia en la elección de los TADs a utilizar para modelar los datos, y las estructuras de datos sobre las cuales éstos se implementan

La importancia de conocer clases de problemas (y su complejidad), además de algoritmos asociados a los principales tipos y estructuras de datos

La importancia de la eficiencia de algoritmos, y la relevancia de esto de acuerdo al problema que se desea resolver

La importancia de conocer y elegir técnicas de diseño de algoritmos que nos ayuden a resolver problemas algorítmicamente.

Resumen de la Clase

Hemos visto que la noción de algoritmo tiene gran relevancia en computación, pues está íntimamente ligada a la programación.

La programación involucra la búsqueda de soluciones algorítmicas para problemas, y por lo tanto conocer técnicas de diseño de tales soluciones, es de fundamental importancia.

La tarea de diseñar algoritmos requiere además la comprensión (y muchas veces formalización) de problemas, y demanda diferentes elecciones, tales como las estructuras de datos sobre las cuales implementar algoritmos, y el grado de exactitud de la solución propuesta (que debe balancearse en muchos casos con la eficiencia de las soluciones alternativas posibles).

Muchos problemas admiten diferentes soluciones algorítmicas, y por lo tanto es útil poder comparar las mismas de acuerdo a su eficiencia, respecto de los recursos necesarios para resolver problemas.