Parte de Algoritmos
de la asignatura de Programación
Master de Bioinformática

Avance rápido

Web asignatura: http://dis.um.es/~domingo/algbio.html

E-mail profesor: domingo@um.es

Transparencias preparadas a partir de las del curso de Algoritmos y Estructuras de Datos II, del Grado de Ingeniería Informática y An Introduction to Bioinformatics Algorithms

Método general

- Los algoritmos **voraces**, **ávidos** o de **avance rápido** (**greedy**) se utilizan normalmente en problemas de optimización, donde una solución está formada por un conjunto de elementos entre un conjunto de candidatos (con un orden determinado o no).

- El **algoritmo voraz** funciona por pasos:
  - Partimos de una solución vacía.
  - En cada paso se escoge el siguiente elemento para añadir a la solución, entre los candidatos.
  - Una vez tomada esta decisión no se podrá deshacer.
  - El algoritmo acabará cuando el conjunto de elementos seleccionados constituya una solución.

Método general

- **Esquema general de un algoritmo voraz:**

voraz (C: conjunto_candidatos; var S: conjunto_solución);
S = Ø
mientras (C ≠ Ø) y no **solución** (S) hacer
    x = **seleccionar** (C)
    C = C **-** {x}
    si **factible** (S ∪ {x}) entonces
    S = S ∪ {x}
finmientras
si no **solución** (S) entonces
    devolver "No se puede encontrar una solución"

- En cada paso tenemos los siguientes conjuntos:
  - **Candidatos seleccionados** para la solución S.
  - **Candidatos** seleccionados pero **rechazados** luego.
  - **Candidatos pendientes** de seleccionar C.

Método general

- **Funciones:**
  - **solución (S).** Comprueba si un conjunto de candidatos es una solución (independientemente de que sea óptima o no).
  - **seleccionar (C)**. Devuelve el elemento más “prometedor” del conjunto de candidatos pendientes (no seleccionados ni rechazados).
  - **factible (C)**. Indica si a partir del conjunto de candidatos C es posible construir una solución (posiblemente añadiendo otros elementos).
  - **Insertar** un elemento en la solución. Además de la inserción, puede ser necesario hacer otras cosas.
  - Función **objetivo (S)**. Dada una solución devuelve el coste asociado a la misma (resultado del problema de optimización).

Problema del cambio de monedas

Disponemos de monedas de distintos valores: de 1, 2, 5, 10, 20 y 50 céntimos de euro, y de 1 y 2 euros.

Supondremos una cantidad ilimitada de cada una.

Construir un algoritmo que dada una cantidad **P** devuelva esa cantidad con monedas de estos tipos, usando un número mínimo de monedas.

P. ej.: para devolver 3.89 €: 1 monedas de 2€, 1 moneda de 1€, 1 moneda de 50 c€, 1 moneda de 20 c€, 3 monedas de 5 c€ y 2 monedas de 2 c€.

- Podemos aplicar la técnica **voraz**: en cada paso añadir una moneda nueva a la solución actual, hasta que el valor llegue a **P**.

Problema del cambio de monedas

- **Candidatos iniciales**: todos los tipos de monedas disponibles. **Solución**: conjunto de monedas que suman la cantidad **P**.
- Una solución será de la forma $\mathbf{(x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8)}$, donde $\mathbf{x_i}$ es el número de monedas de tipo **i**. Suponemos que la moneda **i** vale $\mathbf{c_i}$.
- **Funciones**:
  - **solución**. El valor actual será solución si $\sum x_i \cdot c_i = P$
  - **objetivo**. La función a minimizar es $\sum x_i$, el número de monedas resultante.
  - **seleccionar**. Elegir en cada paso la moneda de valor más alto posible, pero menor que el valor que queda por devolver.
  - **factible**. Valdrá siempre verdad.

Problema del cambio de monedas

- Para este sistema monetario encuentra siempre la solución óptima.

- Puede no encontrar la solución óptima: supongamos que tenemos monedas de 100, 90 y 1. Queremos devolver 180.
  - Algoritmo voraz: 1 moneda de 100 y 80 monedas de 1: total 81 monedas.
  - Solución óptima: 2 monedas de 90: total 2 monedas.

- Puede haber solución y no encontrarla.

- Puede no haber solución y no lo detecta.

Problema del cambio de monedas

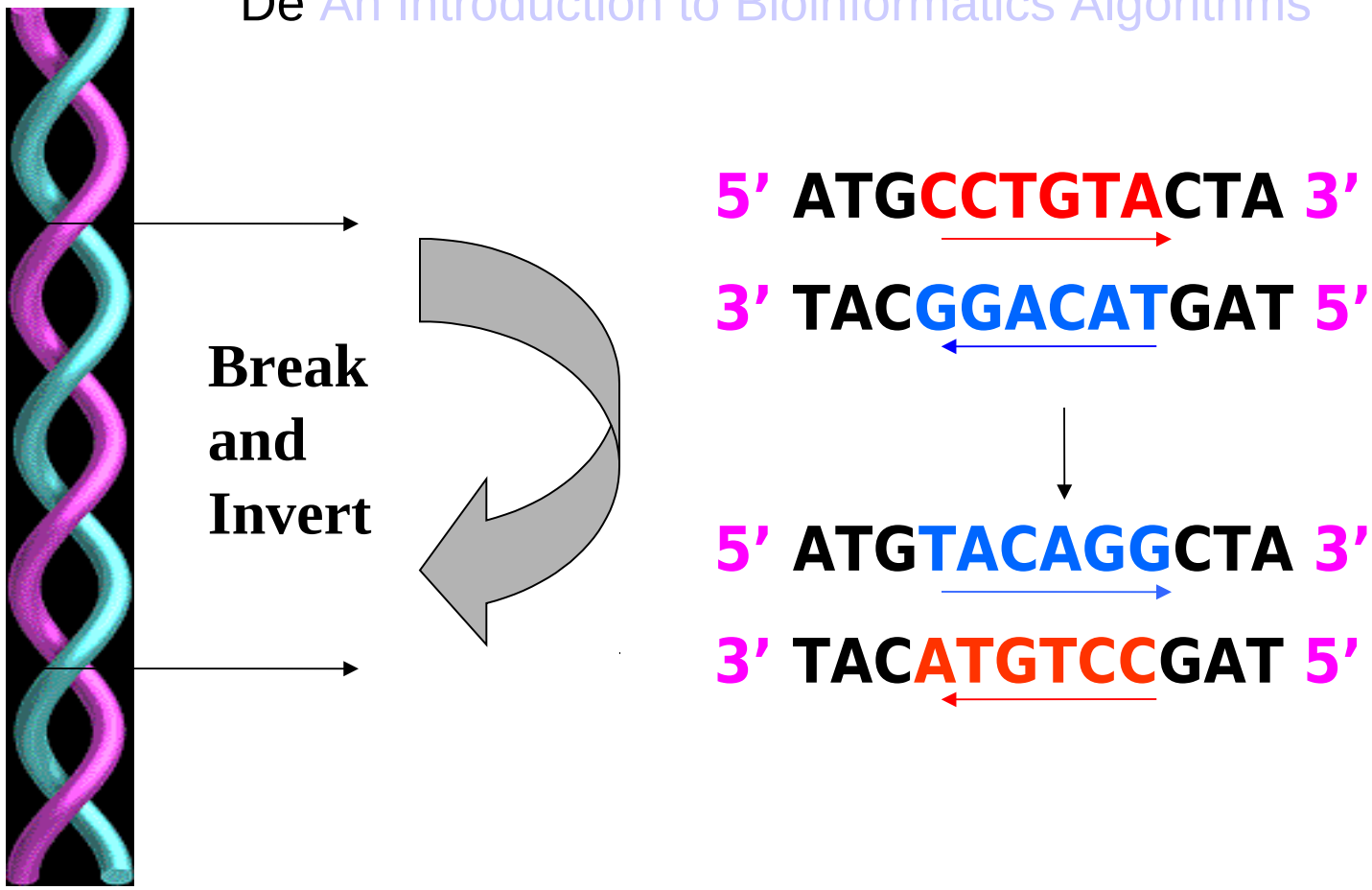
Devolver-cambio (P: integer; C: array [1..N] of integer;
var X: array [1..N] of integer);

```
act = 0
para i = 1,2,...,N
        X[i] = 0
mientras act ≠ P
        j = el mayor elemento de C tal que C[j] ≤ (P - act)
        si j=0 then     { Si no existe ese elemento }
        devolver "No existe solución";
        X[j] = (P - act) div C[j]
        act = act + C[j]*X[j]
```

→ Programar este algoritmo en Perl.

Problema de la distancia de inversión

De An Introduction to Bioinformatics Algorithms



Problema de la distancia de inversión

Puede haber varias transformaciones

π = 1 2 3 4 5 6 7 8

$\rho(3,5)$

1 2 5 4 3 6 7 8

$\rho(5,6)$

1 2 5 4 6 3 7 8

Problema de la distancia de inversión

- Dadas dos permutaciones, encontrar la sucesión más corta de inversiones que transforma una en la otra.

- Entrada: Permutaciones P1 y P2

- Salida: Una serie mínima de inversiones que transforma P1 en P2

- El número de inversiones se llama distancia de inversión entre P1 y P2 (d(P1,P2))

Un caso particular es el Problema de Ordenación por Inversión:
P2 es la identidad (*1 2 … n*)

Problema de ordenación por inversión

Idea de un algoritmo de avance rápido:

- Al ordenar la permutación P1 = 1 2 3 6 4 5, los tres primeros elementos están en orden.
- Llamamos prefijo a la longitud de la parte inicial ya ordenada (*prefix*(P1)).
- Se puede incrementar el valor de *prefix*(P1) en pasos sucesivos.
- El número máximo de pasos es *(n – 1)*

Problema de ordenación por inversión

SimpleReversalSort(π)

1 **for** $i \leftarrow 1$ to $n - 1$
2 $j \leftarrow$ position of element i in π (i.e., $\pi_j = i$)
3 **if** $j \neq i$
4 $\pi \leftarrow \pi * \rho(i, j)$
5 **output** π
6 **if** π is the identity permutation
7 **return**

$\rightarrow$ Programar este algoritmo en Perl.

Se pueden consultar algunas mejoras a este algoritmo y otros algoritmos de avance rápido para problemas de cadenas en
An Introduction to Bioinformatics Algorithms