Parte de Algoritmos, de la asignatura de Programación
Máster de Bioinformática

Aleatoriedad y algoritmos aleatorios

Domingo Giménez Cánovas

Departamento de Informática y Sistemas
Universidad de Murcia
http://dis.um.es/~domingo/algbio.html
domingo@um.es

Aleatoriedad y algoritmos aleatorios

Contenido

- Está en la naturaleza (mutación).
- Generación de valores aleatorios se puede usar para simulación.
- Son necesarias herramientas computacionales para generar valores aleatorios.
- Algoritmos aleatorios realizan operaciones o toman decisiones distintas de forma aleatoria,
  - se pueden utilizar cuando explorar todas las posibilidades no es posible
  - o para tomar una decisión cuando no se sabe cuál es la mejor

- Los lenguajes de programación tienen funciones para generar números aleatorios.
- En realidad son seudoaleatorios, pues generan una secuencia de números, siempre la misma.
- Empezando por una posición distinta (usando una semilla) se puede simular que es aleatoria.
- La semilla se puede establecer con un evento externo, por ejemplo dependiendo de la hora.

Ejemplo 7.1 de PERL

→ Ejecutar el programa.
Escribe historias combinando de forma aleatoria partes distintas:

- Almacena una serie de *nombres*, *verbos* y *preposiciones*
- genera aleatoriamente un *nombre*, un *verbo*, un *nombre* y una *preposicion*, y las concatena formando una historia.
- Genera seis historias y pregunta si queremos continuar.
  Utiliza el bucle do ... `until`,
  y se acaba cuando lo escrito en la entrada estándar (`STDIN`) es, desde el principio (`^`), una serie de espacios en blanco (`\s*`), seguido de (`q`), sin diferenciar mayúsculas y minúsculas (`i`).

Ejemplo 7.1 de PERL (continuación)

→ Localizar `srand` en el programa.
Se genera una semilla para la generación aleatoria con la función `srand`:

- `srand(numero)` generará los números aleatorios en las sucesivas llamadas empezando en la posición de la secuencia aleatoria indicada por `numero`.
- Con `srand(time|$$)` se genera la semilla de forma aleatoria:
  `time` devuelve un número que representa el tiempo
  y con `$$` se concatena (operador `.`) con un número asociado al programa que se está ejecutando.

Ejemplo 7.1 de PERL (fin)

→ Localizar `rand` en el programa.

Con sucesivas llamadas a la función `rand` se generan números aleatorios.

Por ejemplo, `$verbs[int(rand(scalar @verbs))]` se evalúa de dentro a fuera:

- `@verbs` es un array
- `scalar @verbs` devuelve que el array tiene 7 elementos
- `rand(scalar @verbs)` devuelve un número real entre 0 y 7
- con `int(rand(scalar @verbs))` se toma como entero
- y `$verbs[int(rand(scalar @verbs))]` devuelve el elemento (`$`) en esa posición del array `verbo`.

y se puede escribir también `$verbs[int rand scalar @verbs]` o `$verbs[rand @verbs]`

Ejemplo 7.1 de PERL (modificación)

→ Modificar el ejemplo 7.1 para que no se generen historias donde los dos nombres coincidan.

¿Otros cambios?

Ejemplo 7.2 de PERL

Usa funciones para:

- Seleccionar una posición aleatoria en una cadena de ADN,
- seleccionar aleatoriamente un nucleótido
- y sustituirlo en esa posición de la cadena.

Ejemplo 7.2: Posición aleatoria en ADN

```
sub randomposition {
    my($string) = @_;
    return int rand length $string;
}
```

Usa `length` para obtener la longitud del `string` que se le pasa, y devuelve una posición aleatoria en él.

Ejemplo 7.2: Gererar un nucleótido

```
sub randomnucleotide {
    my(@nucs) = @_;
    return $nucs[rand @nucs];
}
```

Si se han guardado las cuatro posibilidades en

```
my @nucleotides = ('A', 'C', 'G', 'T');
```

devuelve uno de los cuatro caracteres.

Ejemplo 7.2: Cambiar una posición en la cadena

```
sub mutate {
    my($dna) = @_;
    my(@nucleotides) = ('A', 'C', 'G', 'T');
# Pick a random position in the DNA
    my($position) = randomposition($dna);
# Pick a random nucleotide
    my($newbase) = randomnucleotide(@nucleotides);
# Insert the random nucleotide into the random position in the DNA.
# The substr arguments mean the following:
# In the string $dna at position $position change 1 character to
# the string in $newbase
    substr($dna,$position,1,$newbase);
    return $dna;
}
```

→ Explicar el funcionamiento de la rutina a partir de los comentarios.

→ Ejecutar y entender el funcionamiento del programa completo.

Ejemplo 7.2 de PERL (modificación)

→ Modificar el ejemplo 7.2:

- Para no cambiar un nucleótido por sí mismo.
- Para cambiar un nucleótido por entre 1 y 4 nucleótidos. El número se genera aleatoriamente y los caracteres también.

Ejemplo 7.3 de PERL

Con técnica de diseño *top-down* (en el caso anterior se ha usado *bottom-up*):

```
@random_DNA = make_random_DNA_set( $minimum_length,
$maximum_length, $size_of_set );
```

donde:

```
repeat $size_of_set times:
    $length = random number between minimum and maximum length
    $dna = make_random_DNA ( $length );
    add $dna to @set
return @set
```

etcétera, se llega al ejemplo 7.3

Ejemplo 7.3 de PERL

→ Identificar la forma en que se ha diseñado el ejemplo 7.3.

→ Identificar las funciones reutilizadas.

→ Identificar nuevos operadores o funciones.

Ejemplo 7.3 de PERL (modificación)

→ Modificar el ejemplo 7.3:

- Para que las cadenas que se generen tengan un número par de caracteres.
- Para que en una cadena el número de caracteres de un tipo no exceda en más de uno el número de caracteres de otro tipo.
- Para que en una cadena no se repita tres veces consecutivas en mismo carácter.

Ejemplo 7.4 de PERL

Se trata de encontrar el porcentaje de coincidencia entre pares de cadenas de ADN de la misma longitud.

→ Implementar un programa que lo haga

→ y, después, compararlo con el ejemplo 7.4 del libro.

Esquema

Para ordenar un array s:
Se selecciona un elemento e del array, y se divide el array en dos subarrays con:

- s_{izq} con elementos menores o iguales a e
- s_{der} con elementos mayores o iguales a e
- y se ordenan recursivamente s_{izq} y s_{der}

Esquema con datos en array

```
quicksort(s,izq,der)
    if length(s)!=1
        p=pivotar(s,izq,der)
        quicksort(s,izq,p-1)
        quicksort(s,p+1,der)
```

donde `pivotar` deja un elemento e en su posición p una vez se hubiera ordenado el array,
deja a la izquierda (entre izq y $p - 1$) elementos menores o iguales a e,
y a la izquierda (entre $p + 1$ y der) elementos mayores o iguales a e.

Pivotaje

```
sub pivotar {
    my ($array, $low, $high) = @_;
    my $x = $$array[$high];
    my $i = $low - 1;
    for my $j ($low .. $high - 1) {
        if ($$array[$j] <= $x) {
            $i++;
            @$array[$i, $j] = @$array[$j, $i];
        }
    }
    $i++;
    @$array[$i, $high] = @$array[$high, $i];
    return $i;
}
```

Tomado de internet...

→ entender lo que hace y hacerlo funcionar.

Tiempo de ejecución

- → Obtener el tiempo de ejecución para entradas ordenadas o inversamente ordenadas, para arrays de muchos elementos.
- El problema es que ese es el caso más defavorable (volveremos sobre esto en el análisis de algoritmos).
- Si se selecciona el pivote de forma aleatoria no tenemos ese caso más desfavorable.
- → Programarlo así y comparar el tiempo de ejecución con la versión anterior.
- → Buscar formas distintas de medir el tiempo de ejecución.

- En el capítulo 12 del libro ALG hay un par de ejemplos de algoritmos aleatorios en bioinformática, pero quedan fuera de los conceptos básicos que estamos viendo en esta introducción a la programación.
- En el libro de Brassard y Bratley hay un capítulo de algoritmos aleatorios (no de bioinformática).