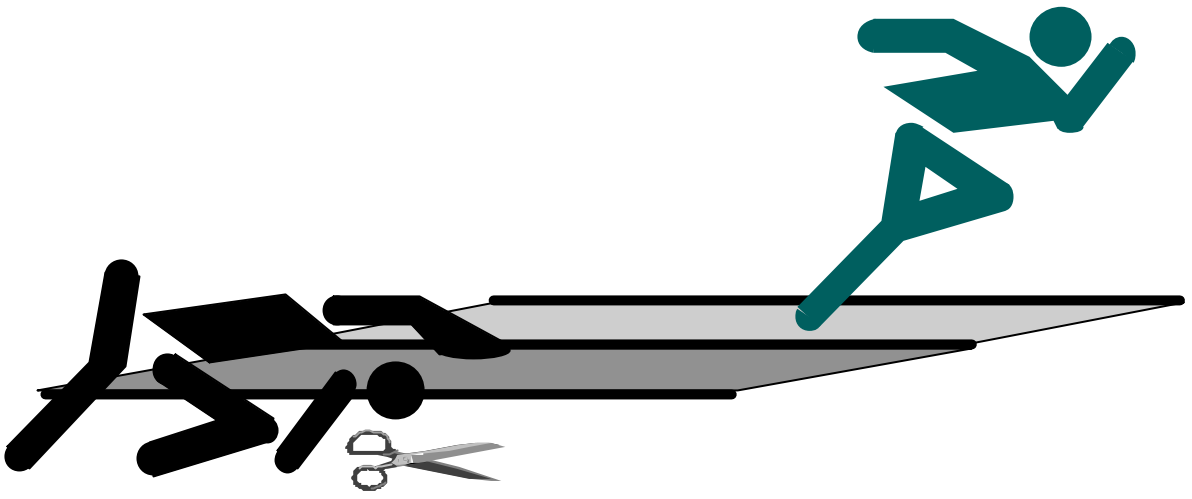
4 Divide y vencerás

4 *El esquema divide y vencerás: Introducción*

- Técnica de diseño de algoritmos "divide y vencerás":
  - descomponer el ejemplar a resolver en un cierto número de subejemplares más pequeños del mismo problema;
  - resolver independientemente cada subejemplar;
  - combinar los resultados obtenidos para construir la solución del ejemplar original.

  aplicar esta técnica recursivamente



4 El esquema divide y vencerás: Introducción

- Esquema genérico:

```
función divide_y_vencerás(x:tx) devuelve ty
variables x1,…,xk:tx; y1,…,yk:ty
principio
  si x es suficientemente simple
    entonces devuelve solución_simple(x)
    sino
      descomponer x en x1,…,xk;
      para i:=1 hasta k hacer
        yi:=divide_y_vencerás(xi)
      fpara;
      devuelve combinar(y1,…,yk)
  fsi
fin
```

- Si k=1, el esquema anterior se llama **técnica de reducción**.

4 El esquema divide y vencerás: Introducción

- Sobre el coste computacional:
  - Sea un algoritmo A que emplea un tiempo cuadrático.
  - Sea c una constante tal que una implementación particular de A emplea un tiempo

    $$t_A(n) \leq cn^2$$

    para un ejemplar de tamaño n.
  - Supongamos que A se puede descomponer en tres subejemplares de tamaño $\lceil n/2 \rceil$, resolverlos y combinar su resultados para resolver A.
  - Sea d una constante tal que el tiempo empleado en la descomposición y combinación es $t(n) \leq dn$.

4 El esquema divide y vencerás: Introducción

– La utilización de esta partición da lugar a un nuevo algoritmo *B* con un tiempo:

$$t_B(n) = 3t_A\left(\lceil n/2 \rceil\right) + t(n)$$
$$\leq 3c\left(\frac{n+1}{2}\right)^2 + dn$$
$$= \frac{3}{4}cn^2 + \left(\frac{3}{2}c + d\right)n + \frac{3}{4}c$$

– Luego *B* es un 25% más rápido que *A*.

– Si se aplica a *B* la misma técnica de descomposición se obtiene un algoritmo *C* con tiempo:

$$t_C(n) = \begin{cases} t_A(n) & \text{si } n \leq n_0 \\ 3t_C\left(\lceil n/2 \rceil\right) + t(n) & \text{en otro caso} \end{cases}$$

– Donde $t_A(n)$ es el tiempo del algoritmo simple, n_0 es el **umbral** por encima del cual el algoritmo se usa recursivamente y $t(n)$ el tiempo de la descomposición y combinación.

– La solución de la ecuación en recurrencias da un tiempo para el algoritmo *C* de $n^{\log 3} \approx n^{1,59}$.

4 El esquema divide y vencerás: Introducción

- Recordar…

  una de las ecuaciones en recurrencias especialmente útil para el análisis de algoritmos de divide y vencerás:

  – La solución de la ecuación

$$T(n) = aT(n/b) + \Theta(n^k \log^p n)$$

con $a \geq 1$, $b > 1$, $p \geq 0$ es

$$T(n) \in \begin{cases} O(n^{\log_b a}), & \text{si } a > b^k \\ O(n^k \log^{p+1} n), & \text{si } a = b^k \\ O(n^k \log^p n), & \text{si } a < b^k \end{cases}$$

4 El esquema divide y vencerás: Introducción

- Sobre el cálculo del umbral óptimo:
  - Es un problema complejo.
  - Para un mismo problema, varía con la implementación concreta y varía con el valor de n.
  - Puede estimarse **empíricamente** (tablas,…)...
  - ... o **teóricamente**: calculando el n para el que la solución recursiva y la solución simple tienen igual coste; por ejemplo, si $t_A(n)=n^2$ y $t(n)=16n$:

$$t_A(n) = 3t_A(\lceil n/2 \rceil) + t(n)$$

da un valor de $n_0 \approx 64$ (ignorando los redondeos).

4.1 La búsqueda dicotómica

- Problema:
  - Dado $T[1..n]$ vector ordenado crecientemente y dado x, encontrar x en T, si es que está.
  - Formalmente: encontrar un natural i tal que $0 \le i \le n$ y $T[i] \le x < T[i+1]$, con el convenio de que $T[0]=-\infty$ y $T[n+1]=+\infty$ (centinelas).
  - Solución secuencial evidente:

```
función secuencial(T:vector[1..n]de dato;
                   x:dato) devuelve 0..n
variables i:0..n+1; b:booleano
principio
  i:=1; b:=falso;
  mq (i≤n) and not b hacer
    si T[i]>x
      entonces b:=verdad
      sino i:=i+1
    fsi
  fmq;
  devuelve i-1
fin
```

Coste caso promedio y caso peor: $\Theta(n)$

4.4 La búsqueda dicotómica

- Búsqueda dicotómica (algoritmo de reducción, puesto que k=1):

```
función dicotómica(T:vector[1..n]de dato;
                   x:dato) devuelve 0..n
principio
  si (n=0) or (x<T[1]) entonces devuelve 0
    sino devuelve dicotRec(T,1,n,x)
  fsi
fin
```

```
función dicotRec(T:vector[1..n]de dato;
                 i,d:1..n;
                 x:dato) devuelve 1..n
{búsqueda dicotómica de x en T[i..d];
 se supone que T[i]≤x<T[d+1] e i≤d}
variable k:1..n
principio
  si i=d entonces devuelve i
    sino
      k:=(i+d+1) div 2;
      si x<T[k]
        ent devuelve dicotRec(T,i,k-1,x)
        sino devuelve dicotRec(T,k,d,x)
      fsi
  fsi
fin
```

4.4 La búsqueda dicotómica

- Versión iterativa:

```
función dicotIter(T:vector[1..n]de dato;
                  x:dato) devuelve 0..n
variables i,d,k:1..n
principio
  si (n=0) or (x<T[1])
    entonces devuelve 0
    sino
      i:=1; d:=n;
      mq i<d hacer
        {T[i]≤x<T[d+1]}
        k:=(i+d+1) div 2;
        si x<T[k]
          entonces d:=k-1
          sino i:=k
        fsi
      fmq;
      devuelve i
  fsi
fin
```

Coste: $\Theta(\log n)$, en todos los casos.

4.4 La búsqueda dicotómica

- Mejora aparente:

```
función dicotIter2(T:vector[1..n]de dato;
                   x:dato) devuelve 0..n
variable i,d,k:1..n
principio
  si (n=0) or (x<T[1])
    entonces devuelve 0
    sino
      i:=1; d:=n;
      mq i<d hacer
        {T[i]≤x<T[d+1]}
        k:=(i+d) div 2;
        si x<T[k]
          entonces d:=k-1
          sino
            si x≥T[k+1]
              entonces i:=k+1
              sino i:=k; d:=k
            fsi
        fsi
      fmq;
      devuelve i
  fsi
fin
```

La búsqueda dicotómica

- La mejora es sólo aparente:
  - el primer algoritmo realiza siempre $\Theta(\log n)$ iteraciones mientras que el segundo puede parar antes;
  - cálculo del **número medio de iteraciones** suponiendo que:
    - todos los elementos de T son distintos,
    - x está en T,
    - cada posición tiene igual probabilidad de contener a x

    [BB90, pp. 116-119]

$$Iter_1(n) \cong Iter_2(n) + \frac{3}{2}$$

  - Es decir, el primer algoritmo realiza, en media, una iteración y media más que el segundo.
  - Pero, cada iteración del segundo es ligeramente más costosa que la del primero.

  Son difícilmente comparables.

  Si n es suficientemente grande, el primer algoritmo es mejor que el segundo.

La ordenación por fusión

- Dado un vector $T[1..n]$ de n elementos dotados de una relación de orden total, se trata de ordenar de forma creciente esos elementos.
- Técnica de divide y vencerás más sencilla:
  - dividir el vector en dos mitades;
  - ordenar esas dos mitades recursivamente;
  - fusionarlas en un solo vector ordenado.

```
algoritmo ordFusión(e/s T:vect[1..n]de dato)
variables U:vect[1..(n div 2)]de dato;
          V:vect[1..((n+1) div 2)]de dato
principio
  si n es pequeño
    entonces ordInserción(T)
    sino
      U:=T[1..(n div 2)];
      V:=T[(n div 2 + 1)..n];
      ordFusión(U);
      ordFusión(V);
      fusión(T,U,V)
  fsi
fin
```

La ordenación por fusión

- Coste de ejecución:
  - la descomposición de T en U y V precisa un tiempo lineal;
  - la fusión de U y V también necesita un tiempo lineal;
  - por tanto, si $t(n)$ es el tiempo de ejecución del algoritmo para ordenar n elementos,

$$t(n) \in t(\lfloor n/2 \rfloor) + t(\lceil n/2 \rceil) + \Theta(n)$$

  es decir, $t(n) \in \Theta(n \log n)$

- Además:

  La fusión de $T[1..k]$ y $T[k+1..n]$ (ya ordenados) en el mismo $T[1..n]$ puede hacerse en tiempo lineal utilizando sólo una cantidad fija de variables auxiliares (independiente de n).

  Ejercicio (véase la solución del ejercicio 18 de la sección 5.2.4 de [Knu73]).

4 El algoritmo de ordenación de Hoare

C.A.R. Hoare: "Quicksort",
Computer Journal, 5(1), pp. 10-15, 1962.

```
algoritmo ordRápida(e/s T:vect[1..n]de dato;
                    ent iz,de:1..n)
{Ordenación de las componentes iz..de de T.}
variable me:1..n
principio
  si de-iz es pequeño
    entonces ordInserción(T,iz,de)
    sino
      divide(T,iz,de,me);
      {iz≤k<me⇒T[k]≤T[me] ∧ me<k≤de⇒T[k]>T[me]}
      ordRápida(T,iz,me-1);
      ordRápida(T,me+1,de)
  fsi
fin
```

4 El algoritmo de ordenación de Hoare

```
algoritmo divide(e/s T:vect[1..n]de dato;
                 ent iz,de:1..n;
                 sal me:1..n)
{Permuta los elementos iz..de de T de forma que:
   iz≤me≤de,
   ∀k t.q. iz≤k<me: T[k]≤p,
   T[me]=p,
   ∀k t.q. me<k≤de: T[k]>p,
   p se llama pivote y es, por ejemplo, el valor
   inicial de T[iz]. }
variables p:dato; k:1..n
principio
  p:=T[iz];
  k:=iz; me:=de+1;
  repetir k:=k+1 hasta que (T[k]>p)or(k≥de);
  repetir me:=me-1 hasta que (T[me]≤p);
  mq k<me hacer
    intercambiar(T[k],T[me]);
    repetir k:=k+1 hasta que T[k]>p;
    repetir me:=me-1 hasta que T[me]≤p
  fmq;
  intercambiar(T[iz],T[me])
fin
```

4 *El algoritmo de ordenación de Hoare*

- Coste en el peor caso: **cuadrático**

  (si la elección del pivote no es adecuada, los subejemplares no tienen tamaño parecido…)

- Tiempo medio: $O(n \log n)$

  Dem.: [BB90, pp. 123-124]

  (la constante multiplicativa es menor que en los otros algoritmos de ordenación que hemos visto)

4 Algoritmos de selección y de búsqueda de la mediana

- Dado un vector $T[1..n]$ de enteros, la **mediana** de T es un elemento m de T que tiene tantos elementos menores como mayores que él en T:

$$\left|\{i \in [1..n] \mid T[i] < m\}\right| < n/2, \quad \text{y}$$

$$\left|\{i \in [1..n] \mid T[i] > m\}\right| \leq n/2$$

(nótese que la definición formal es más general: n puede ser par y además puede haber elementos repetidos, incluidaPrimera solución:

- Ordenar T y extraer el elemento $\lceil n/2 \rceil$-ésimo.
- Tiempo $\Theta(n \log n)$.

4 Algoritmos de selección y de búsqueda de la mediana

- Problema más general: la **selección** (o cálculo de los **estadísticos de orden**).

  Dado un vector T de n elementos y $1 \le k \le n$, el ***k*-ésimo menor** elemento de T es aquel elemento m de T que satisface:

$$\left|\{i \in [1..n] \mid T[i] < m\}\right| < k, \quad \text{y}$$

$$\left|\{i \in [1..n] \mid T[i] \le m\}\right| \ge k$$

  Es decir, el elemento en posición k si T estuviese ordenado crecientemente.

- Por tanto, la mediana de T es el $\lceil n/2 \rceil$-ésimo menor elemento de T.

- Solución del problema: inspirada en el algoritmo de ordenación de Hoare (pero ahora sólo se resuelve uno de los subejemplares).

4 Algoritmos de selección y de búsqueda de la mediana

```
función seleccionar(T:vector[1..n]de dato;
                    k:1..n) devuelve dato
principio
  devuelve selRec(T,1,n,k)
fin
```

```
función selRec(T:vector[1..n]de dato;
               i,d,k:1..n) devuelve dato
principio
  si d-i es pequeño entonces
    ordInserción(T,i,d); devuelve T[k]
  sino
    divide(T,i,d,m);
    si m-i+1≥k
      entonces selRec(T,i,m,k)
      sino selRec(T,m+1,d,k-(m-i+1))
    fsi
  fsi
fin
```

```
algoritmo divide(e/s T:vect[1..n]de dato;
                 ent i,d:1..n; sal m:1..n)
{Permuta los elementos i..d de T de forma que:
 i≤m≤d; ∀k t.q. i≤k<m: T[k]≤p; T[m]=p;
 ∀k t.q. m<k≤d: T[k]>p;
 p es, por ejemplo, el valor inicial de T[i]. }
```

4 Algoritmos de selección y de búsqueda de la mediana

- Como en el método de ordenación de Hoare, una elección desafortunada del pivote conduce en el **caso peor** a un tiempo **cuadrático**.

- No obstante, el **coste promedio es lineal**.

- Existe una mejora que permite un coste lineal también en el peor caso, pero, en la práctica, se comporta mejor el algoritmo anterior [CLR90].

4 Multiplicación de enteros grandes

- Coste de realizar las operaciones elementales de suma y multiplicación:
  - Es razonable considerarlo constante si los operandos son directamente manipulables por el hardware, es decir, no son muy grandes.
  - Si se necesitan enteros muy grandes, hay que implementar por software las operaciones.

- Enteros muy grandes:
  - **Representación** de un entero de *n* cifras: puede hacerse en un vector con un espacio en $O(n)$ bits.
  - Operaciones de **suma**, **resta**: pueden hacerse en **tiempo lineal**.
  - Operaciones de **multiplicación** y **división** entera **por potencias positivas de 10**: pueden hacerse en **tiempo lineal** (desplazamientos de las cifras).
  - Operación de **multiplicación** con el **algoritmo clásico** (o con el de multiplicación rusa): **tiempo cuadrático**.

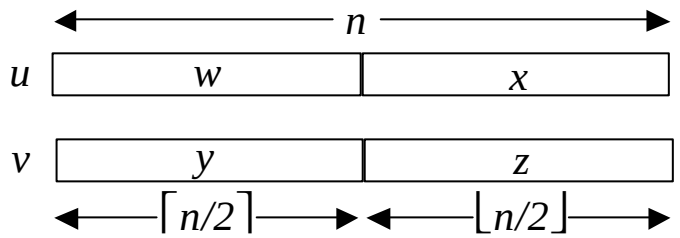
4 Multiplicación de enteros grandes

A. Karatsuba e Y. Ofman:
"Multiplication of multidigit numbers on automata",
Dokl. Akad. Nauk SSSR, 145, pp. 293-294, 1962.

- Técnica de divide y vencerás para la multiplicación de enteros muy grandes:
  - Sean u y v dos enteros de n cifras.
  - Se descomponen en mitades de igual tamaño:

$$\left.\begin{aligned} u &= 10^s w + x \\ v &= 10^s y + z \end{aligned}\right\} \text{ con } \quad 0 \le x < 10^s, \quad 0 \le z < 10^s, \text{ y } \quad s = \lfloor n/2 \rfloor$$

  - Por tanto, w e y tienen $\lceil n/2 \rceil$ cifras



  - El producto es:

$$uv = 10^{2s} wy + 10^s (wz + xy) + xz$$

4 Multiplicación de enteros grandes

```
función mult(u,v:granEnt) devuelve granEnt
variables n,s:entero; w,x,y,z:granEnt
principio
  n:=máx(tamaño(u),tamaño(v));
  si n es pequeño
    entonces devuelve multClásica(u,v)
    sino
      s:=n div 2;
      w:=u div 10^s; x:=u mod 10^s;
      y:=v div 10^s; z:=v mod 10^s;
      devuelve mult(w,y)*10^2s
               +(mult(w,z)+mult(x.y))*10^s
               +mult(x,z)
  fsi
fin
```

4 Multiplicación de enteros grandes

- Coste temporal:
  - Las sumas, multiplicaciones por potencias de 10 y divisiones por potencias de 10: tiempo lineal.
  - Operación módulo una potencia de 10: tiempo lineal (puede hacerse con una división, una multiplicación y una resta).
  - Por tanto, si $t(n)$ es el tiempo del algoritmo:

$$t(n) = 3t\left(\lceil n/2 \rceil\right) + t\left(\lfloor n/2 \rfloor\right) + \Theta(n)$$

Si n se supone potencia de 2,

$$t(n) = 4t\left(n/2\right) + \Theta(n)$$

La solución de esta recurrencia es:

$$t(n) \in O(n^2)$$

¿¡Y no hemos ganado nada!?

4 Multiplicación de enteros grandes

– Truco: calcular wy, $wz+xy$, y xz haciendo menos de cuatro multiplicaciones.

Teniendo en cuenta que:

$$r = (w + x)(y + z) = wy + (wz + xy) + xz$$

```
función mult2(u,v:granEnt) devuelve granEnt
variables n,s:entero; w,x,y,z,r,p,q:granEnt
principio
  n:=máx(tamaño(u),tamaño(v));
  si n es pequeño
    entonces devuelve multClásica(u,v)
    sino
      s:=n div 2;
      w:=u div 10^s; x:=u mod 10^s;
      y:=v div 10^s; z:=v mod 10^s;
      r:=mult2(w+x,y+z);
      p:=mult2(w,y); q:=mult2(x,z);
      devuelve p*10^2s+(r-p-q)*10^s+q
  fsi
fin
```

4 Multiplicación de enteros grandes

- Eficiencia temporal de la versión 2:
  - Teniendo en cuenta que $w+x$ e $y+z$ pueden necesitar $1+\lceil n/2 \rceil$ cifras,

    $$t_2(n) \in t_2(\lfloor n/2 \rfloor) + t_2(\lceil n/2 \rceil) + t_2(1+\lceil n/2 \rceil) + \Theta(n)$$

    Y por tanto,

    $$t_2(n) \in \Theta(n^{\log 3}) \in O(n^{1,59})$$

  - Debido a la constante multiplicativa, el algoritmo sólo es interesante en la práctica para *n* grande.
  - Una buena implementación no debe usar la base 10 sino la base más grande posible que permita multiplicar dos “cifras” (de esa base) directamente en hardware.

4 Multiplicación de enteros grandes

– La diferencia entre los órdenes n^2 y $n^{1,59}$ es menos espectacular que la existente entre n^2 y $n\log n$.

– Ejemplo realizado en un computador en base 2^{20}:

(computador con palabras de 60 dígitos, que permite multiplicar directamente números de 20 dígitos binarios)

- Multiplicación de números de unas 602 cifras (base decimal):

  algoritmo clásico: 400 milisegundos

  algoritmo mult2: 300 milisegundos

- Multiplicaciones de números de unas 6000 cifras (base decimal):

  algoritmo clásico: 40 segundos

  algoritmo mult2: 15 segundos

– Descomponiendo los operandos en más de dos partes se puede lograr la multiplicación de dos enteros en un tiempo del orden de n^{α}, para cualquier $\alpha>1$.

– Descomponiendo los operandos en $n^{1/2}$ partes y usando la transformada de Fourier, se pueden multiplicar dos enteros de n cifras en un tiempo $O(n \log n \log\log n)$.

4 Potenciación de enteros

- Problema:

  Dados los enteros positivos *a*, *n* y *z*, se trata de calcular a^n mod *z*.

- Solución ingenua:

```
función pot0(a,n,z:entero) devuelve entero
{Devuelve aⁿ mod z.}
variables r,i:entero
principio
  r:=1;
  para i:=1 hasta n hacer
    r:=r*a
  fpara;
  devuelve r mod z
fin
```

  - Si se quita la operación `mod z` final, sirve para calcular a^n (no modular).
  - Si "`r:=r*a`" se considera "operación elemental", el coste está en $\Theta(n)$.
    Pero NO es así (p.e., 15^{17} no cabe en 8 bytes).

4 Potenciación de enteros

- Inconvenientes de esta solución:

  - Su coste (el bucle se ejecuta n veces).

    Si se quiere aplicar a un entero a grande (de m cifras), el coste es prohibitivo.

    - $\Theta(m^2n^2)$, si se usa el algoritmo clásico de multiplicar.

      nº cifras(r_1)=m (r_1 denota r tras el 1er paso)

      nº cifras(r_{i+1})=$m+m_i \Rightarrow$ nº cifras(r_{i+1})=$i\ m$

      $$\text{coste} = \sum_{i=1}^{n-1} M(m, i\ m)$$

      Método clásico de multiplicar

      $$\Rightarrow \text{coste} = \sum_{i=1}^{n-1} c\ m\ i\ m = cm^2 \sum_{i=1}^{n-1} i = cm^2n^2$$

    - $\Theta(m^{\log 3}n^2)$, si se usa el algoritmo de divide y vencerás [BB96].

4.4 Potenciación de enteros

- Existe un algoritmo más eficiente para la potenciación de enteros.

  Por ejemplo, con dos multiplicaciones y elevar al cuadrado cuatro veces se obtiene:

$$x^{25} = \left(\left(\left(x^2 x \right)^2 \right)^2 \right)^2 x$$

  Nótese que reemplazando en

$$x^{25} = \left(\left(\left(x^2 \times x \right)^2 \times 1 \right)^2 \times 1 \right)^2 \times x$$

  cada *x* por un 1 y cada 1 por un 0, se obtiene la secuencia de bits 11001 (expresión binaria de 25).

  En otras palabras,

$$x^{25} = x^{24} x; \quad x^{24} = \left(x^{12} \right)^2; \quad \text{etc.}$$

4 Potenciación de enteros

- Un algoritmo de eficiencia aceptable:

```
función pot(a,n,z:entero) devuelve entero
{Devuelve a^n mod z.}
variable r:entero
principio
  si n=0
    entonces devuelve 1
    sino
      si n es impar
        entonces
          r:=pot(a,n-1,z);
          devuelve a*r mod z
        sino
          r:=pot(a,n/2,z);
          devuelve r*r mod z
      fsi
  fsi
fin
```

(Es un algoritmo de reducción)

4 Potenciación de enteros

- Sobre la eficiencia de este algoritmo:
  - Tiempo $O(h(n) \times M(z))$
    - $h(n)$ = nº de multiplicaciones módulo z.
    - $M(z)$ = cota del tiempo para multiplicar dos enteros menores que z y calcular el módulo.
  - Es claro que:

$$h(n) = \begin{cases} 0 & \text{si } n = 0 \\ 1 + h(n-1) & \text{si } n \text{ es impar} \\ 1 + h(n/2) & \text{en otro caso} \end{cases}$$

Y, por tanto, $h(n)$ es logarítmico en n.

4.4 Potenciación de enteros

- Una versión iterativa:

```
función potIter(a,n,z:entero) devuelve entero
{Devuelve a^n mod z.}
variable i,x,r:entero
principio
  i:=n; x:=a; r:=1;
  mq i>0 hacer
    si i es impar entonces r:=r*x mod z fsi;
    x:=x*x mod z;
    i:=i div 2
  fmq;
  devuelve r
fin
```

4 Potenciación de enteros

- Comentarios finales:

  - Ni la versión recursiva ni la iterativa del algoritmo de potenciación presentadas aquí son las óptimas, pero ambas son razonablemente buenas.

  - Suprimiendo de ambos algoritmos los cálculos "mod *z*", se obtienen algoritmos para elevar un entero cualquiera *a* a otro cualquiera *n*.

    Su eficiencia depende, obviamente, del algoritmo que se utilice para multiplicar enteros grandes.

  - Si *a*, *n* y *z* son números de 200 cifras:

    - el algoritmo `potIter` puede calcular a^n mod *z* en **menos de 1 segundo**,

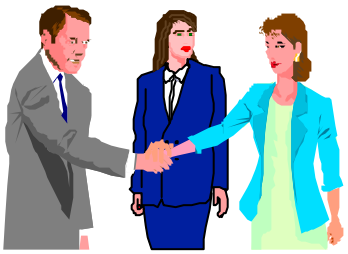
    - el algoritmo ingenuo `pot0` requeriría del orden de $\mathbf{10^{179}}$ **veces la edad del Universo**.

    → Suponiendo que dos enteros de 200 cifras se multiplican en 1 milisegundo.

Introducción a la criptografía

- El problema:

  Eva fue un día al teatro con su vecina Alicia y allí le presentó a ésta a su amigo Roberto.



  Por razones que no vienen al caso, Alicia decide enviar a Roberto un mensaje secreto.

  Desea hacerlo en una conversación telefónica, pero ésta puede ser escuchada por Eva.

  Eva puede escuchar la información que circula por la línea pero no puede introducir mensajes ni modificar los que circulan.

  Por supuesto, Alicia no desea que Eva tenga acceso a sus secretos.

  Se trata de encontrar un método que permita a Alicia conseguir su objetivo.

Introducción a la criptografía

- La solución clásica (¡usada durante siglos!):
  - Alicia y Roberto, previamente y en secreto, se han puesto de acuerdo en una clave *k*.
  - Usando la clave *k*, Alicia puede encriptar su mensaje *m*, obteniendo el mensaje cifrado *c*.
  - Roberto, al recibir el mensaje *c*, utiliza la clave *k* para descifrarlo y obtener el mensaje original *m*.
  - Si Eva intercepta *c*, como no conoce la clave *k*, no podrá reconstruir el mensaje *m*.
- Problema actual: telecomunicaciones
  - Dos ciudadanos necesitan enviarse un mensaje cifrado mediante correo electrónico por internet pero no pueden verse previamente en secreto para acordar una clave *k*.

  ¡Este problema no se resolvió hasta los años 70!

4A Introducción a la criptografía

- Solución:
  - Un **sistema criptográfico de clave pública**.
  - El más conocido: el sistema RSA.

R.L. Rivest, A. Shamir y L.M. Adleman:
"A method for obtaining digital signatures and public-key cryptosystems",
Communications of the ACM, 21(2), pp. 120-126, 1978.

  - Publicado previamente en:

M. Gardner: "Mathematical games: A new kind of cipher that would take millions of years to break",
Scientific American, 237(2), pp. 120-124, 1977.

  - Planteó un problema cuya solución estimó que necesitaría 2 millones de veces la edad del Universo de cálculo ininterrumpido del mejor computador de aquel momento.

4A Introducción a la criptografía

💻 p, q: n^{os} primos de 100 cifras
💻 $z=pq$
💻 $\phi=(p-1)(q-1)$
💻 n: $1 \le n \le z-1$, $\text{mcd}(n,\phi)=1$
💻 s: $1 \le s \le z-1$: $ns \bmod \phi = 1$ } (*)
(s es único, ¡me lo guardo!)

(*) ¿Cómo?

☎ z, n

☺ m: mensaje en texto
💻 a: codificado (ASCII)
($0 \le a \le z-1$)
💻 c: $c=a^n \bmod z$

☎ c

💻 a: $a=c^s \bmod z$ (**)
💻 m: mensaje (ASCII)
☺ Yo también…

☹ z, n, c…

(**) ¿Por qué?

Introducción a la criptografía

- Cálculo de n:
  - Se genera un valor de n aleatoriamente en $(1,z\text{-}1)$.
  - Después se intenta calcular s tal que $1 \leq s \leq z\text{-}1$ y $ns \bmod \phi = 1$;
    - la teoría de números asegura que si existe es único y si no existe es porque n no verifica $\text{mcd}(n,\phi)=1$.
  - Si no se encuentra tal s es que n no es adecuado ($\text{mcd}(n,\phi)\neq 1$), se vuelve a generar otro n; etc.

4A Introducción a la criptografía

- Cálculo de *s*:
  - **Lema**. Sean $u \geq v > 0$ y $d = \text{mcd}(u,v)$. Entonces existen a y b tales que $au+bv=d$.

    Dem.: Si $u=v$ entonces $d=v$, $a=0$ y $b=1$.

    Si $u>v$, sea $w=u \bmod v$.

    En particular $w<v$.

    Se tiene que $d=\text{mcd}(v,w)$.

    Por inducción, existen a' y b' tales que $a'v+b'w=d$.

    Luego $a=b'$ y $b=a'-(u \text{ div } v)b'$.

```
algoritmo totó(ent u,v:entero;
               sal d,a,b:entero)
{Pre: u,v>0} {Post: d=mcd(u,v) ∧ d=au+bv}
variables w,aa,bb:entero
principio
  selección
    u=v: d:=v; a:=0; b:=1;
    u<v: totó(v,u,d,b,a);
    u>v: w:=u mod v; {w<v} {d=mcd(v,w)}
         totó(v,w,d,aa,bb);
         a:=bb; b:=aa-(u div v)*bb
  fselección
fin
```

Introducción a la criptografía

– Por tanto, para calcular *s* tal que *ns* mod ϕ=1:

- *n*, ϕ>0
- mcd(*n*,ϕ)=1

Calcular *s* y *t* tales que *ns*+*t*ϕ=1.

```
totó(n,ϕ,d,s,t);
si d=1
  entonces s es el buscado
  sino n no es 'bueno'
fsi
```

$ns + t\phi = 1 \Rightarrow ns = 1 - t\phi$

$ns \bmod \phi = (1 - t\phi) \bmod \phi = 1 - 0 = 1$

Introducción a la criptografía

- Falta comprobar la corrección del mensaje obtenido por Roberto.
  - **Teorema**. Sean p,q dos números primos, $z=pq$, $\phi=(p-1)(q-1)$ y a tal que $0 \leq a < z$.

    Si $x \bmod \phi = 1$ entonces $a^x \bmod z = a$.
  - Alicia construye su mensaje a, $0 \leq a < z$, y a partir de él, calcula $c = a^n \bmod z$.
  - Roberto calcula

    $c^s \bmod z = (a^n \bmod z)^s \bmod z = a^{ns} \bmod z = a$

    (porque $ns \bmod \phi = 1$).

Introducción a la criptografía

- ¿Qué puede hacer Eva?
  - Conoce z, n y c.
  - Tiene que encontrar a, que es el único número entre 0 y z-1 tal que $c=a^n \bmod z$.
  - Es decir, tiene que calcular la raíz n-ésima de c módulo z.
  - **No se conoce ningún algoritmo eficiente para hacerlo.**

Introducción a la criptografía

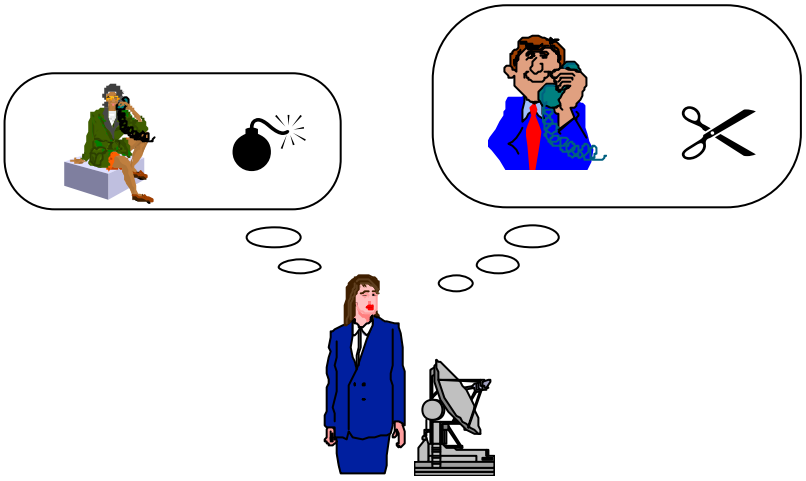
- La mejor solución conocida es **factorizar *z* en *p* y *q***.
  - Después, calcular $\phi=(p-1)(q-1)$, calcular *s*, y calcular *a* igual que Roberto.

- Problema: factorizar un número de 200 cifras *parece* **imposible** con la tecnología actual y el estado actual de la teoría de números.
  - El problema propuesto por Gardner exigía factorizar un número de 129 cifras (Gardner estimó en 2 millones de veces la edad del Universo el tiempo necesario para resolverlo).
  - En 1989, se estimaba que ese mismo problema podría ser resuelto "ya" en unos 250 años–CRAY.

    Si se trata de un número de 200 cifras, la estimación subía a 1 millón de años–CRAY.

Introducción a la criptografía

- ¡Cuidado!
  - En abril de 1994, Atkins, Graff, Lenstra y Leyland han resuelto el problema propuesto por Gardner en 1977 (que exigía factorizar un número de 129 cifras):
    - 8 meses de cálculo, mediante
    - más de 600 computadores de todo el mundo trabajando como una máquina paralela virtual.



4A Multiplicación de matrices

V. Strassen: "Gaussian elimination is not optimal", *Numerische Mathematik*, 13, pp. 354-356, 1969.

- Sean A y B dos matrices $n \times n$ y C su producto.
  - El algoritmo clásico calcula:

$$C_{ij} = \sum_{k=1}^{n} A_{ik} B_{kj}; \quad i, j = 1, \ldots, n$$

    Es decir, precisa un tiempo $\Theta(n^3)$.

- La idea de Strassen es reducir el número de multiplicaciones a costa de aumentar el número de sumas y restas e introducir variables auxiliares (como en la multiplicación de enteros grandes).

4A Multiplicación de matrices

- Veamos el caso 2×2:

$$A = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} \quad \text{y} \quad B = \begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{pmatrix}$$

  - Algoritmo clásico: 8 multiplicaciones y 4 sumas
  - Reducción:

$$\begin{aligned} m_1 &= (a_{21} + a_{22} - a_{11})(b_{22} - b_{12} + b_{11}) \\ m_2 &= a_{11} b_{11} \\ m_3 &= a_{12} b_{21} \\ m_4 &= (a_{11} - a_{21})(b_{22} - b_{12}) \\ m_5 &= (a_{21} + a_{22})(b_{12} - b_{11}) \\ m_6 &= (a_{12} - a_{21} + a_{11} - a_{22}) b_{22} \\ m_7 &= a_{22}(b_{11} + b_{22} - b_{12} - b_{21}) \end{aligned}$$

$$AB = \begin{pmatrix} m_2 + m_3 & m_1 + m_2 + m_5 + m_6 \\ m_1 + m_2 + m_4 - m_7 & m_1 + m_2 + m_4 + m_5 \end{pmatrix}$$

Es decir, 7 multiplicaciones y 24 sumas y restas.

Notar que el número de sumas y restas puede reducirse a 15 utilizando más variables auxiliares para evitar cálculos repetidos, como el de

$$m_1 + m_2 + m_4$$

4A Multiplicación de matrices

- Caso $2n \times 2n$:
  - Si cada elemento de A y B del caso 2×2 lo sustituimos por una matriz $n \times n$, se pueden multiplicar dos matrices $2n \times 2n$ realizando:
    - 7 multiplicaciones de matrices de $n \times n$, y
    - 15 sumas y restas de matrices de $n \times n$.

- Coste:
  - Una suma de matrices $n \times n$ está en $\Theta(n^2)$.
  - Suponer que n es potencia de 2; se obtiene la recurrencia:

$$T(n) = 7T\left(n/2\right) + \Theta(n^2)$$

    Es decir, se pueden multiplicar matrices en tiempo $\Theta(n^{\log 7}) \in O(n^{2,81})$.

- ¿Y si n no es potencia de 2?

Multiplicación de matrices

- Si n no es potencia de 2:
  - Solución más fácil: añadir filas y columnas nulas hasta que lo sea.
  - Solución sugerida por Strassen:
    - Para todo entero n es siempre posible encontrar enteros m y k tales que $n=m2^k$.
    - Por tanto, una matriz $n \times n$ se puede partir en $2^k \times 2^k$ submatrices de $m \times m$.
    - Entonces, se aplica el método de divide y vencerás hasta que hay que multiplicar matrices $m \times m$ y entonces se usa el método clásico.

      (Nótese que si, por ejemplo, n es impar, el método no sirve.)

Multiplicación de matrices

- Comentarios finales sobre el algoritmo de Strassen:
  - Según algunos estudios empíricos y debido a la constante multiplicativa, para que se noten mejoras con respecto al algoritmo clásico, n debe ser superior a 100, y la matriz densa.
  - Es menos estable que el algoritmo clásico (i.e., para errores similares en los datos produce mayores errores en el resultado).
  - Es más difícil de implementar que el algoritmo clásico.
  - Es difícilmente paralelizable, mientras que el clásico puede ser fácilmente paralelizado.
  - El algoritmo clásico precisa un espacio adicional de tamaño constante, mientras que el algoritmo de Strassen precisa un espacio adicional mayor.
  - El algoritmo de Strassen ha tenido una repercusión fundamentalmente teórica (diotambién para otras operaciones basadas en él (inversión de matrices, determinantes,...).
  - Se conocen algoritmos teóricamente mejores (la actual cota superior está en $O(n^{2,376})$).

4.4 Calendario de un campeonato

- El problema:

  En una competición deportiva se enfrentan n participantes.

  Debe confeccionarse un calendario para que cada participante juegue exactamente una vez con cada adversario.

  Además, cada participante debe jugar exactamente un partido diario.

  Supongamos, por simplificar, que $n=2^k$.

4.4 Calendario de un campeonato

- La representación de los datos y de la solución:

  Participantes: 1, 2, …, *n*.

  Cada participante debe saber el orden en el que se enfrenta a los *n*-1 restantes.

  Por tanto, la solución puede representarse en una matriz *n*×(*n*-1).

participantes	días 1	2	3	4	5	6	7
1	2	3	4	5	6	7	8
2	1	4	3	8	5	6	7
3	4	1	2	7	8	5	6
4	3	2	1	6	7	8	5
5	6	7	8	1	2	3	4
6	5	8	7	4	1	2	3
7	8	5	6	3	4	1	2
8	7	6	5	2	3	4	1

  El elemento (*i*,*j*), 1≤*i*≤*n*, 1≤*j*≤*n*-1, contiene el número del participante contra el que el participante *i*-ésimo compite el día *j*-ésimo.

4.4 Calendario de un campeonato

- Solución de fuerza bruta:

1. Se obtiene para cada participante i, $1 \leq i \leq n$, el conjunto $P(i)$ de todas las permutaciones posibles del resto de los participantes $\{1..n\} \setminus \{i\}$.
2. Se completan las filas de la matriz de todas las formas posibles, incluyendo en cada fila i algún elemento de $P(i)$.
3. Se elige cualquiera de las matrices resultantes en la que toda columna j, $1 \leq j \leq n-1$, contiene números distintos (nadie puede competir el mismo día contra dos participantes).

– Cada conjunto $P(i)$ consta de $(n-1)!$ elementos.
– La matriz tiene n filas.

→ Hay $n!$ formas distintas de rellenar la matriz.

¡Es demasiado costoso!

4.4 Calendario de un campeonato

- Una solución mediante la técnica de divide y vencerás (caso $n=2^k$, $k>0$):
  - Caso básico: $n=2$, basta con una competición.
  - Caso a dividir: $n=2^k$, con $k>1$.
    - Se elaboran independientemente dos subcalendarios de 2^{k-1} participantes:

      Uno para los participantes $1..2^{k-1}$, y

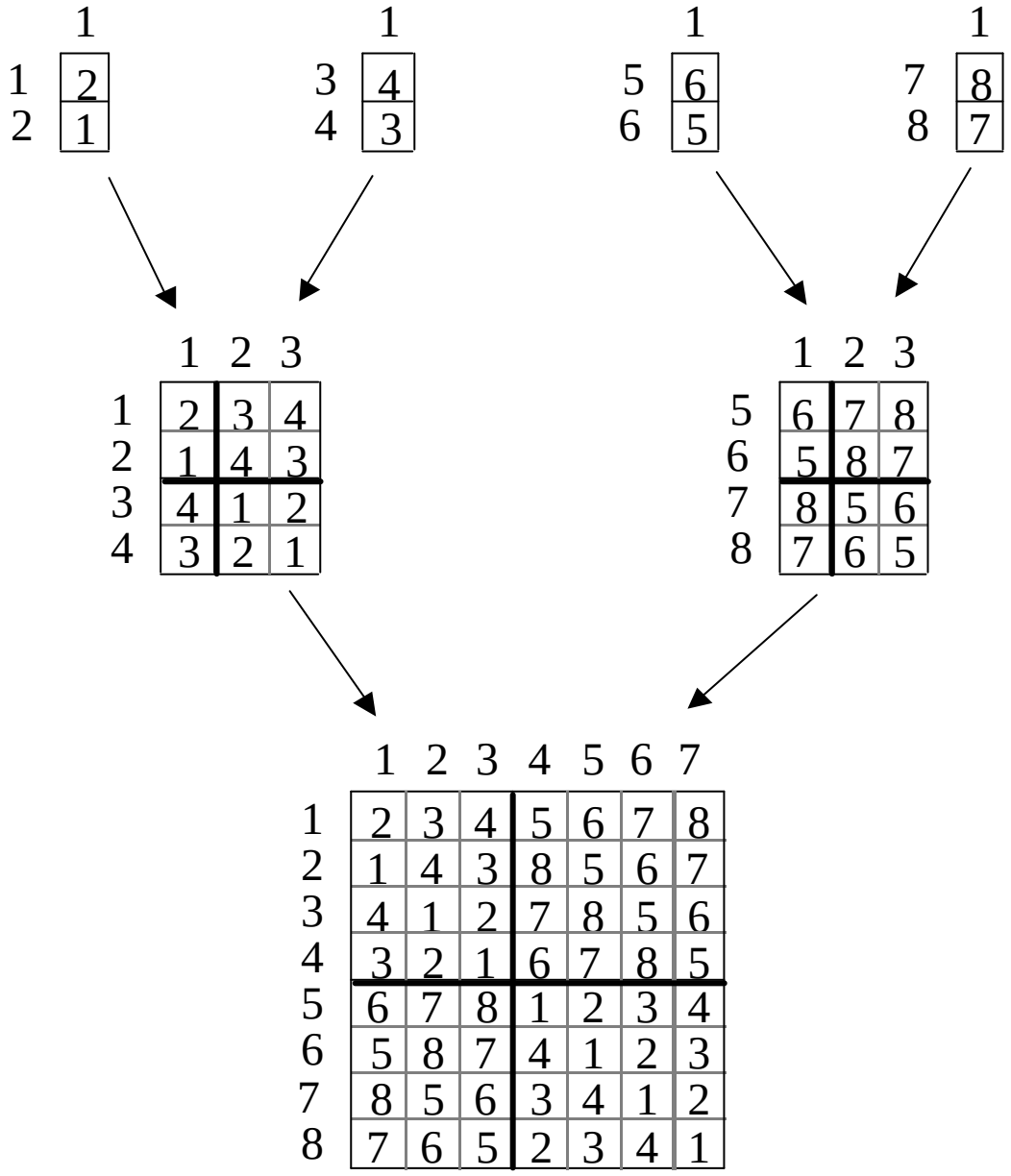
      otro para los participantes $2^{k-1}+1..2^k$.
    - Falta elaborar las competiciones cruzadas entre los participantes de numeración inferior y los de numeración superior.
      - Se completa primero la parte de los participantes de numeración inferior:

        1er participante: compite en días sucesivos con los participantes de numeración superior en orden creciente.

        2º participante: toma la misma secuencia y realiza una permutación cíclica de un participante.

        Se repite el proceso para todos los participantes de numeración inferior.
      - Para los de numeración superior se hace lo análogo con los de inferior.

4.4 Calendario de un campeonato



4 Calendario de un campeonato

```
tipo tabla=vector[1..n,1..n-1] de 1..n
```

```
algoritmo calendario(sal t:tabla)
{Devuelve en t al calendario de competición de
 n participantes, con n=2^k, k>0.}
principio
  formarTabla(t,1,n)
fin
```

```
algoritmo formarTabla(sal t:tabla;
                      ent inf,sup:1..n)
{Forma la parte de tabla correspondiente a los
 enfrentamientos de los participantes inf..sup}
variable medio:1..n
principio
  si inf:=sup-1
    entonces t[inf,1]:=sup; t[sup,1]:=inf
    sino
      medio:=(inf+sup) div 2;
      formarTabla(t,inf,medio);
      formarTabla(t,medio+1,sup);
      completarTabla(t,inf,medio,
                     medio,sup-1,medio+1);
      completarTabla(t,medio+1,sup,
                     medio,sup-1,inf)
  fsi
fin
```

4.4 Calendario de un campeonato

```
algoritmo completarTabla(ent/sal t:tabla;
                         ent eqInf,eqSup,
                             díaInf,díaSup,
                             eqInic:1,,n)
{Rellena t[eqInf..eqSup,díaInf..díaSup] con
 permutaciones cíclicas empezando en eqInic}
principio
  para j:=díaInf hasta díaSup hacer
    t[eqInf,j]:=eqInic+j-díaInf
  fpara;
  para i:=eqInf+1 hasta eqSup hacer
    t[i,díaInf]:=t[i-1,díaSup];
    para j:=díaInf+1 hasta díaSup hacer
      t[i,j]:=t[i-1,j-1]
    fpara
  fpara
fin
```

4.4 Calendario de un campeonato

- Coste temporal:
  - Recurrencia:

$$T(n) = 2T\left(n/2\right) + n^2/4$$

  - Por tanto, $O(n^2)$.

- ¿Y si *n* no es potencia de 2?
  - Supongamos que está **resuelto para *n* par**.
  - **Si *n* es impar** (mayor que 1), no bastan con *n*-1 días para la competición, sino que se necesitan *n*.

    Basta con resolver el problema resultante de **añadir un participante (ficticio) más**: el *n*+1.

    Como *n*+1 es par, podemos calcular el calendario, que consta de *n* días.

    Una vez resuelto, si al participante *i*-ésimo le toca jugar el día *j*-ésimo contra el participante *n*+1 (ficticio), eso significa que *j* es el día de descanso para el participante *i*.

4.4 Calendario de un campeonato

- ¿Y si *n* es par?
  - Si *n* div 2 es par, se puede aplicar el algoritmo "formarTabla" (visto para el caso $n=2^k$).
  - Si *n* div 2 es impar:

    Paso 1:

    - Se calcula la primera parte de la competición para los *n* div 2 participantes de numeración inferior, añadiendo un participante ficticio.
    - Se calcula la primera parte de la competición para los otros *n* div 2 participantes , añadiendo otro participante ficticio.

    Paso 2:

    - El día *j*-ésimo, $1 \le j \le n$ div 2, se hace jugar entre sí a los dos participantes, uno de numeración inferior y otro superior, a los que les había tocado el *j* como día de descanso en el Paso anterior.

    Paso 3:

    - Se completa el resto de la tabla con competiciones cruzadas, de forma parecida a como se hizo en el algoritmo "completarTabla".

4 Calendario de un campeonato