Estructuras de datos y algoritmos: recursión y arboles

Dr. J.B. Hayet

CENTRO DE INVESTIGACIÓN EN MATEMÁTICAS

Noviembre 2009

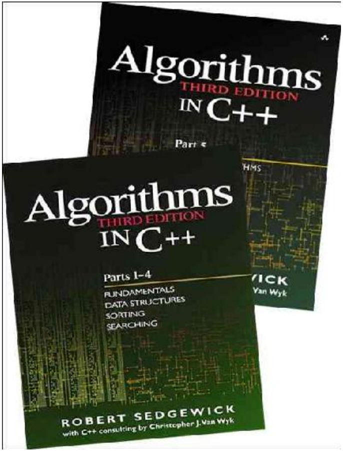
Outline

1. Arboles y recursión

2. Divide and conquer

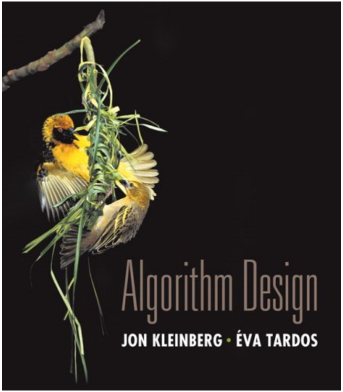
Referencias

- *Algorithms in C++*, R. Sedgewick (Part I y II)
- *Algorithms design*, K. Kleinberg y E. Tardos
- *Introduction to Algorithms*, T.H. Cormen, C.E. Leiserson, R.L. Rivest and C. Stein
- *The art of computer programming*, D. Knuth



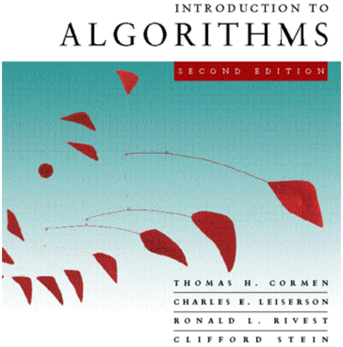
Referencias

- *Algorithms in C++*, R. Sedgewick (Part I y II)
- *Algorithms design*, K. Kleinberg y E. Tardos
- *Introduction to Algorithms*, T.H. Cormen, C.E. Leiserson, R.L. Rivest and C. Stein
- *The art of computer programming*, D. Knuth



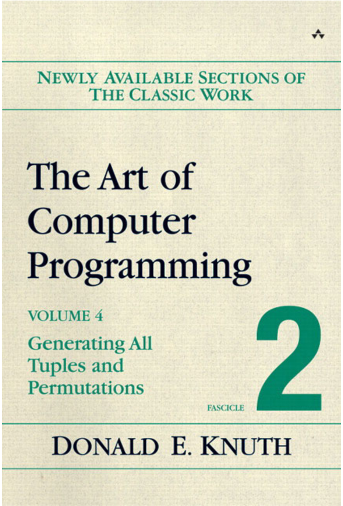
Referencias

- *Algorithms in C++*, R. Sedgewick (Part I y II)
- *Algorithms design*, K. Kleinberg y E. Tardos
- *Introduction to Algorithms*, T.H. Cormen, C.E. Leiserson, R.L. Rivest and C. Stein
- *The art of computer programming*, D. Knuth



Referencias

- *Algorithms in C++*, R. Sedgewick (Part I y II)
- *Algorithms design*, K. Kleinberg y E. Tardos
- *Introduction to Algorithms*, T.H. Cormen, C.E. Leiserson, R.L. Rivest and C. Stein
- *The art of computer programming*, D. Knuth



Tipos de datos abstractos

En esas ultimas sesiones, nos vamos a alejar de la “técnica” de la programación y interesarnos más profundamente a la estructuración de los problemas:

Tipos de datos abstractos: especificaciones matemáticas, lógicas de un conjunto de datos y de las operaciones que se pueden efectuar sobre estos datos; típicamente, corresponden a una especie de contrato que implementa una forma especial de estructuras de datos

Tipos de datos abstractos

- Ladrillos elementales para la concepción de algoritmos en una concepción ascendente
- Separan la especificación de la implementación practica
- Independiente de un lenguaje
- Modularidad
  - El usuario no sabe el costo del uso (como fue la implementación?) pero puede suponer que es una implementación "optima"
  - El conceptor no sabe cuál será el uso: ¿a qué grado de generalidad pararse?

Tipos de datos abstractos

Ya vimos varios ejemplos de TDAs, que pueden tener varias implementaciones:

- pila
- cola
- cola de prioridad

Cada uno puede corresponder a una solución algorítmica a un problema dado. En las clases que vienen, veremos otros ADTs, en particular arboles, arboles binarios, diccionarios...

Outline

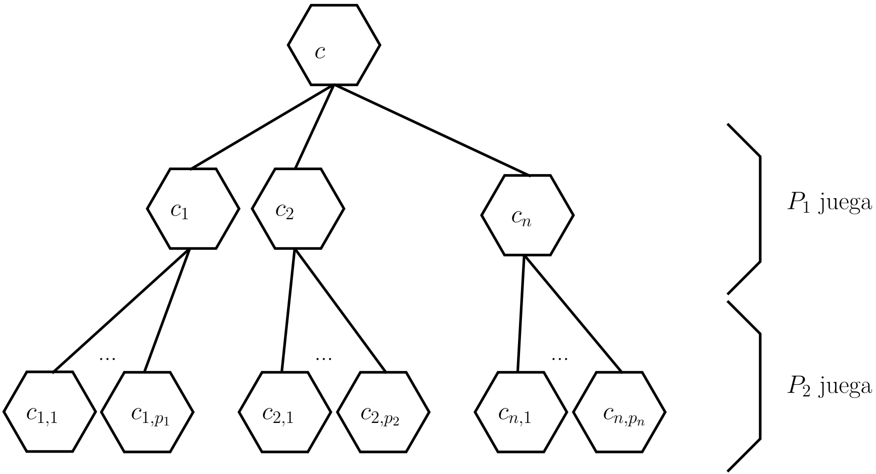
1. Arboles y recursión

2. Divide and conquer

Recursión

- Concepto muy clásico en computación y matemática; un programa recursivo es uno que se llama a si mismo
- Concepto intrínsecamente ligado al de árbol: la estructura de las llamadas al programa es la de un árbol, cada llamada (nodo padre) llamando sí misma una o mas llamadas al mismo programa (nodos hijos)
- Ver los cerillos. . .

Recursión



Equivalentemente...

```
// Compute best turn
int computeBestTurnMiniMax(const configuration *s,
                           turn *best,
                           double *valmax) {
  //
  int nTurns = computePossibleTurnNumber(s);
  if (nTurns<=1) {
    *valmax = evaluation(s);
    return 0;
  }
  // Turns
  turn *turns  = NULL;
  computePossibleTurns(s,&turns,&nTurns);
  double *vals = (double *)malloc(nTurns*sizeof(double));

  for (int i=0;i<nTurns;i++) {
    // Next configuration
    configuration snext = nextConfiguration(s,&turns[i]);
    // Compute recursively the best turn
    computeBestTurnMiniMax(&snext, best,&vals[i]);
  }
  ...
```

Equivalentemente. . .

```
// I am playing : at this step I want to maximise my gains
if (s->player==0) {
  // Search the max
  *valmax = -1.0;
  for (int k=0;k<nTurns;k++) {
    if (vals[k]>*valmax) {
      *valmax = vals[k];
      *best   = turns[k];
    }
  }
}
// He is playing : he will minimize my gains
else {
  // Search the min
  *valmax = 10.0;
  for (int k=0;k<nTurns;k++) {
    if (vals[k]<*valmax) {
      *valmax = vals[k];
      *best   = turns[k];
    }
  }
}
free(vals);
```

La recursión mas vieja del mundo

```
int gcd(int m, int n) {
    if (n == 0) return m;
    return gcd(n, m % n);
}
```

Basada en el hecho que si $m > n$,

$$m = kn + m\%n$$

Un divisor común a m y n tiene que dividir $m\%n$

Una recursión clásica

```
int factorial(int N) {
    if (N == 0) return 1;
    return N*factorial(N-1);
}
```

Equivalente a un ciclo, ¿no?

```
for (int i=1;i<=N;i++) fac *= i;
```

- En general, las funciones recursivas se pueden escribir como ciclos y viceversa.
- Las recursiones permiten escribir el algoritmo de “bonita” manera
- Puede haber un costo adicional muy grande al usar funciones recursivas, por parte de la recursión intrínsecamente (ex: Fibonacci) o de los mecanismos informáticos (llamadas a funciones)

Una recursión problemática

```
int puzzle(int N) {
    if (N == 1) return 1;
    if (N % 2 == 0)
        return puzzle(N/2);
    else return puzzle(3*N+1);
  }
```

Comportamiento?

- Mecanismo de terminación
- Las llamadas recursivas se deben de hacer sobre valores, conjuntos de datos "mas chicos" que los de la entrada para asegurar una convergencia hacia el caso de terminación (o sea para poder hacer pruebas inductivas)

Otro ejemplo

```
char *a; int i;
int eval() { int x = 0;
    while (a[i] == ' ') i++;
    if (a[i] == '+')
      { i++; return eval() + eval(); }
    if (a[i] == '*')
      { i++; return eval() * eval(); }
    while ((a[i] >= '0') && (a[i] <= '9'))
      x = 10*x + (a[i++]-'0');
    return x;
  }
```

Que hace el programa?

Mas generalmente, pruebas por inducción, recurrencia

Listas ligadas

Un caso de problema intrínsecamente recursivo:

- Listas ligadas
- Arboles

Generalmente estructuras construidas expresando hijos/siguientes en función del nodo corriente y apuntadores, y que implican funciones de recorrido recursivas

```
void traverse(link h, void (*visit)(link)) {
    if (h == 0) return;
    (*visit)visit(h);
    traverse(h->next, visit);
  }
```

Outline

1 Arboles y recursión

2 Divide and conquer

Divide and Conquer

Una primera clase de algoritmos recursivos: los de Divide And Conquer que consisten en separar la resolución de un problema en la resolución de varios sub-problemas "mas fáciles" e independientes, suponiendo que se tiene un operador que permite calcular el resultado global a partir de los resultados de las sub-estancias.

```
Item max(Item a[], int l, int r) {
    if (l == r) return a[l];
    int m = (l+r)/2;
    Item u = max(a, l, m);
    Item v = max(a, m+1, r);
    if (u > v) return u; else return v;
  }
```

Interesante solo si es mas eficiente que la versión iterativa...

Divide and Conquer

Propiedad: una función recursiva que divide un problema de tamaño N en dos problemas independientes no vacíos se llama a sí misma menos de N veces.

Se prueba facilmente que con $T_1 = 0$, y si se divide el problema en dos partes sumando a N

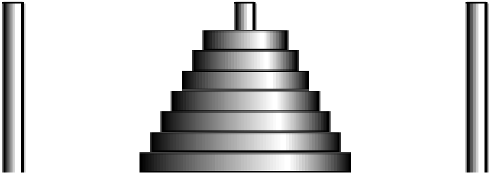
$$T_N = T_k + T_{N-k} + 1$$

y de la misma manera si se divide en partes sumando a $< N$.
En cuanto a la complejidad algorítmica, depende:

- de las operaciones dentro la función
- de si los sub-conjuntos hacen un overlap del conjunto inicial

Divide and Conquer: las torres de Hanoi

- Discos de tamaño decreciente en pila sobre un palo
- No se puede poner un disco de tamaño mas grande arriba de un disco de tamaño mas chico
- Hay tres palos, como pasar la pila de un palo al de su derecha?



Divide and Conquer: las torres de Hanoi

Intuición, programa recursivo:

```
void hanoi(int N, int d) {
    if (N == 0) return;
    hanoi(N-1, -d);
    shift(N, d);
    hanoi(N-1, -d);
  }
```

Se mueve las torres de arriba hacia la dirección opuesta a la que queremos ir, se mueve el disco de abajo por la buena dirección y se mueve de nuevo la torre movida de un desplazamiento por la dirección opuesta ala deseada (circularidad)

Divide and Conquer: las torres de Hanoi

Complejidad: se resuelve el problema de N instancias como 2 resoluciones de problemas a $N-1$ instancias. Ademas $T_1 = 1\ldots$

$$T_N = 2T_{N-1} + 1$$

que lleva fácilmente (recurrencia) a $T_N = 2^N - 1$

Divide and Conquer: un problema similar

Dibujar una regla graduada, con marcas grandes cada unidad, marcas mas pequeñas cada media unidad, marcas menos pequeñas cada cuarto de unidad...

```
void rule(int l, int r, int h) {
    int m = (l+r)/2;
    if (h > 0) {
        rule(l, m, h-1);
        mark(m, h);
        rule(m, r, h-1);
    }
}
```

Ejemplo: que hace *rule*$(0, 8, 3)$? Estructura muy similar al de las torres de Hanoi!

Divide and Conquer: un problema similar

- En el caso del max, problema lineal en el tamaño de los inputs
- En el caso de Hanoi o de las marcas, problema lineal en el tamaño del output (pero exponencial en el tamaño de los inputs; pero queríamos de todos modos 2^N marcas, no ?)
- Un algoritmo iterativo simple en el caso de las marcas de la regla?

Divide and Conquer: un problema similar

Notar que la estructura del problema es la de los múltiples de las potencias de 2 dentro de números a N bits:

0 0 0 0 1

0 0 0 1 0

0 0 0 1 1

0 0 1 0 0

0 0 1 0 1

0 0 1 1 0

0 0 1 1 1

0 1 0 0 0

0 1 0 0 1

Un algoritmo muy simple: contar los ceros consecutivos a partir del bit de peso mas chico!

Divide and Conquer: un problema similar

De la misma observación se puede deducir un algoritmo iterativo para las torres de Hanoi (p.e. mover una torre de N elementos a la derecha): alternar hasta la meta

1. mover el disco mas chico hacia la derecha si N impar (derecha si N par)
2. efectuar el único movimiento posible que no involucra a este mismo disco mas chico

se empieza Y se acaba por un movimiento de disco mas chico

Prueba: por recurrencia !

Divide and Conquer: un problema similar

Usar las potencias de 2:

```
void rule(int l, int r, int h) {
    for (int t = 1, j = 1; t <= h; j += j, t++)
      for (int i = 0; l+j+i <= r; i += j+j)
        mark(l+j+i, t);
  }
```

Implementación *bottom-up*

Divide and Conquer: un problema similar

Las diferentes maneras de resolver el problema de dibujo de marcas finalmente solo se distinguen en cuanto al orden de hacer los dibujos, y, al fin y al cabo, todos los dibujos están hechos:

- El programa bottom-up recorre el árbol nivel por nivel
- El programa inicial hace un recorrido in-orden: recorre la rama izquierda, hace la marca y recorre la rama derecha
- Se puede proponer un algoritmo que haga un recorrido pre-orden (marcar y luego ocuparse de las dos mitades)
- El orden puede importar o no, dependiendo del problema

Divide and Conquer

- Método *bottom-up*: *combine and conquer*, ya que combina los resultados obtenidos para instancias de dificultad trivial entre niveles mas y mas difíciles
- Extensión a fractales: fractal de Koch (sigue linear en el numero efectivo de segmentos que se obtiene pero exponencial en la profundidad del árbol)
- Otros ejemplos de Divide And Conquer: búsqueda binaria y mergeSort (complejidad?)

Master theorem

Una receta de cocina para determinar el comportamiento asintótico de secuencias (T_n) satisfaciendo:

$$T_n = aT_{\lfloor n/b \rfloor} + f_n,$$

donde $a \geq 1$ y la secuencia (f_n) es también dada
Denominado también Master Theorem

Master theorem

- Si $f_n = \Omega(n^{\log_b a+\varepsilon})$ para algún $\varepsilon > 0$ entonces

$$T_n = \Theta(f_n)$$

con la condición que $af_{\frac{n}{b}} < cf_n$ par algún $c < 1$

- Si las dos secuencias (f_n) y (T_n) son con valores estrictamente positivos y $f_n = \Theta(n^{\log_b a} \log^k n)$ para algún k entonces

$$T_n = \Theta(n^{\log_b a} \log_b^{k+1} n)$$

- Si $f_n = O(n^{\log_b a-\varepsilon})$ para algún $\varepsilon > 0$ entonces

$$T_n = \Theta(n^{\log_b a})$$

Master theorem

Ejemplos:

1. $T_n = T_{n/2} + 1$
2. $T_n = T_{n/3} + n$
3. $T_n = 2T_{n^{\frac{1}{2}}} + \log n$