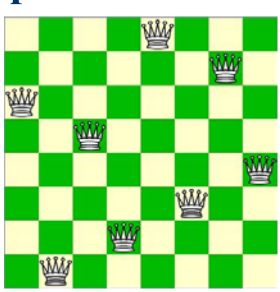
El Arte de la Programación Rápida

Backtracking

Problema ejemplo

- Vamos a analizar la técnica de backtracking a través de un ejemplo.
- El **problema de las 8 reinas**



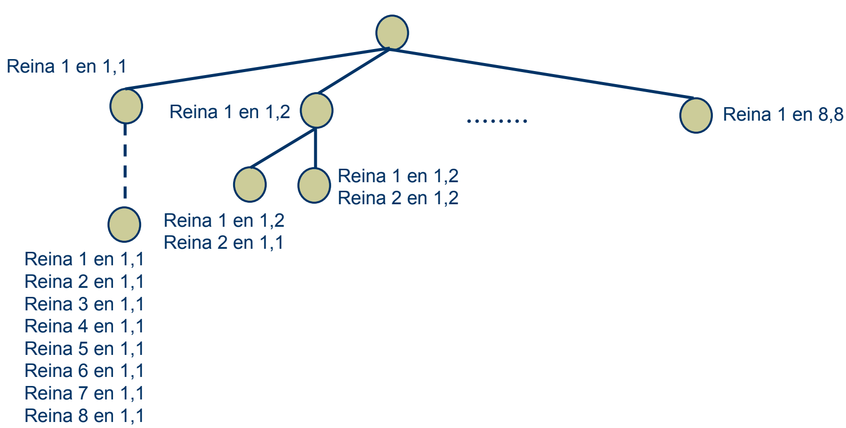
Cómo ubicar 8 reinas en un tablero sin que se "amenacen" la una a la otra?

Generalizable a N reinas

Backtracking = Fuerza bruta

- Método sistemático que itera a través de todas las combinaciones posibles del espacio de búsqueda.
- Es una técnica general que debe ser adaptada para cada aplicación particular.
- Siempre puede encontrar todas las soluciones existentes
  - El **problema** es el **tiempo** que toma en hacerlo.

Representación gráfica



Backtracking: modelamiento

- En general representamos la solución como un vector a = (a_1,a_2,...,a_n), donde:
  - Cada a_i es seleccionado de un conjunto S_i.
- Puede representar, por ejemplo:
  - Una situación donde a_i es el *i-ésimo* elemento de una permutación.
  - Un subconjunto (a_i es verdadero **sii** el *i-ésimo* elemento del universo pertenece al conjunto.
  - Secuencia de movimientos en un juego o camino en un grafo, donde a_i contiene el *i-ésimo* evento en la secuencia.

Backtracking: algoritmo

- En cada paso:
  - Empezamos desde una solución parcial a = (a_1, a_2,...,a_k); (al principio, vacía).
  - Tratamos de extenderla agregando otro elemento al final.
  - Después, comprobamos si tenemos una solución.
    - Si la tenemos, podemos imprimirla, guardarla, contarla, etc.
  - Si no es solución, comprobamos si la solución parcial es todavía extensible para completar una solución
    - Si lo es, nos llamamos recursivamente y continuamos
  - Si no es extensible, borramos el último elemento de a e intentamos otra posibilidad para esa solución, si hay.

Backtracking: algoritmo

```
backtrack(int a[], int k, data input) {
   int c[MAX_CANDIDATES];
   int ncandidates;
   int i;

   if(is_a_solution(a,k,input)){
        process_solution(a,k,input);
   else {
        k = k+1;
        ncandidates=construct_candidates(a,k,input,c);
        for (i=0;i<ncandidates;i++){
               a[k] = c[i];
               backtrack(a,k,input);
               if (finished) return;
        }
   }
}
```


Backtracking: especializando

- `is_a_solution(a,k,input)`
  - Verifica si tenemos una solución. El parámetro `input` sirve para dar información de contexto
    - Por ejemplo, tamaño de la solución buscada
- `construct_candidates(a,k,input,c)`
  - Llena un vector c con los posibles candidatos (hijos del nodo actual en el árbol), retornando el número de candidatos.
- `process_solution(a,k)`
  - Imprime, cuenta o hace lo necesario con la solución encontrada.

Backtracking: construcción de subconjuntos

- Podemos construir los 2^n subconjuntos de n elementos iterando sobre los 2^n posibles vectores de longitud *n*
  - Cada posición tiene un `true` o `false`.
  - Es decir, usando la notación general, S_k=(`true,false`).

Construcción de subconjuntos: algoritmo

```
is_a_solution(int a[],int k, int n){
   return (k == n);
}

int construct_candidates(int a[],int k,int n, int c[]){
   c[0] = TRUE;
   c[1] = FALSE;
   return 2;
}
process_solution(int a[], int k){
   int i;
   printf("{");
   for (i=1; i<=k; i++)
        if(a[i] == TRUE) printf(" %d",i);
   printf(" }\n");
}
```

Construcción de subconjuntos: algoritmo

```
Generate_subsets(int n){
   int a[NMAX];

   backtrack(a,0,n);
}

Salida:

{ 1 2 3 }
{ 1 2 }
{ 1 3 }
{ 1 }
{ 2 3 }
{ 2 }
{ 3 }
{ }
```


Backtracking: construcción de permutaciones

- Ahora el candidato para la próxima "movida" depende de los valores de la solución parcial.
  - Para evitar repetir elementos, debemos verificar que el *i-ésimo* elemento es distinto de los anteriores.
  - Utilizando la notación, `Sk = {1,...,n} - a`
  - `a` es una solución siempre que `k=n`.

Construcción de permutaciones: algoritmo

```
int construct_candidates(int a[],int k,int n, int c[]){
   int i;
   bool in_perm[NMAX];
   int candidates = 0;

   for(i=1;i<=NMAX;i++) in_perm[i]=FALSE;
   for(i=0;i<k;i++) in_perm[ a[i] ] = TRUE;

   for (i=1; i<=n;i++)
        if(in_perm[i]=FALSE{
               c[candidates] = i;
               candidates = candidates + 1;
        }
   return candidates;
}
```


¿Y las 8 reinas?

- Primera solución:
  - 1 vector para cada reina: $(a_{1,1}, a_{1,2},\ldots a_{64,64})$, donde uno de los $a_{i,j}$ es TRUE y el resto FALSE.
  - 2^{64} vectores posibles para cada reina ($\approx 2*10^{19}$)
- Antes de sacar cuentas, pensemos...
  - PC usan procesadores de ≈ 1Gh (1000 millones de operaciones por segundo)
  - Se puede esperar procesar unos pocos millones de elementos por segundo ($\approx 10^6$)

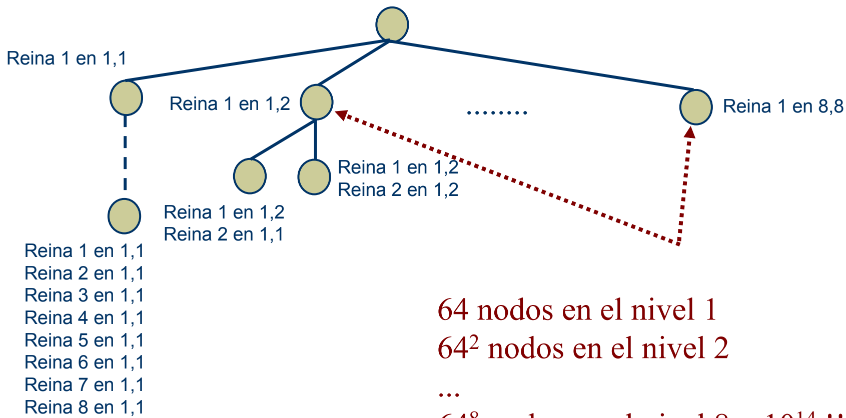
Representación

- Está claro que da lo mismo que la reina en la celda 1,1 sea la reina 1 o la reina x.
- Y está claro que en la solución no pueden haber dos reinas en la misma celda!
- Entonces sólo necesitamos un vector $a = (a_{1,1}, a_{1,2},\ldots a_{64,64})$, donde que $a_{i,j}$ es TRUE significa que hay una reina en esa celda (hay 8 TRUE y el resto FALSE).
- Esto lo podemos resolver con el cálculo de subconjuntos que ya vimos.
- Seguimos necesitando $\approx 2*10^{19}$ vectores.
  - Es decir, en el orden de 10^{13} segundos!!
  - "Sólo" unos.... 300000 años?

Representación y poda

- Podemos considerar que el `i-ésimo` elemento del vector solución nos dice explícitamente dónde está ubicada la reina `i`.
- $(a_1, a_2,\ldots a_8)$, donde cada a_i es un número entre 1 y 64.
- Poda: sólo consideramos candidatos a las celdas no amenazadas

Representación gráfica



64 nodos en el nivel 1

64^2 nodos en el nivel 2

...

64^8 nodos en el nivel 8 $\approx 10^{14}$!!

Métodos de poda

- Es necesario reducir mucho mas el número de nodos a generar.
- Primero podemos quitar simetrías:
  - Ya sabes que las reinas son iguales.
  - Estoy generando la misma solución 8! = 40320 veces.
  - Es fácil evitarlo haciendo que la reina en la posición a_i esté siempre en una celda con número mayor que la está en la posición a_{i-1}.
    - El espacio de búsqueda será de $\approx 4*10^9$

Métodos de poda (II)

- Podemos darnos cuenta que siempre tendremos una y sólo una reina por fila.
- Entonces los candidatos para el reina `i` sólo pueden ser las celdas de la fila `i`.
  - Es decir, indico el número de columna.
- Nuestro espacio de búsqueda ahora es $8^8 \approx 1{,}677 * 10^7$
  - Grande, pero manejable...

Métodos de poda (III)

- Más aún, no puede haber dos reinas en la misma columna...
- Entonces los números de columna de la solución son una permutación de [1..8].
  - Espacio de solución = 8! = 40320

Solución posible a N reinas

```
int construct_candidates(int a[],int k,int n,int c[]){
        int i,j;
        bool legal_move;
        int candidates = 0;
        for (i=1;i<=n;i++){
                legal_move = TRUE;
                for (j=1;j<k;j++){
                    if(abs(k-j) == abs(i-a[j])) legal_move = FALSE;
                    if(i==a[j])legal_move = FALSE;
                }
                if (legal_move == TRUE) c[candidates++] = i;
        }
}
```


Solución posible a N reinas (II)

```
process_solution(int a[], int k){
        solution_count ++;
}

is_a_solution(int a[], int k, int n){
        return (k == n);
}

nqueens(int n){
        int a[NMAX];
        solution_count = 0;
        backtrack(a,0,n);
        printf("n=%d solution count=%d\n", n, solution_count);
}
```