Data structures and Algorithms
Recursive algorithm

Pham Quang Dung

Hanoi, 2012

Outline

Recursively defined functions

Recursive function $f(n)$ is specified based on an integer number $n \geq 0$ and the following schema

- Basic step: Specify $f(0)$
- Recursive step: Specify $f(n+1)$ depending on $f(k), \forall k = 0, \ldots, n$

Example

- $f(0) = 3$
- $f(n+1) = 2f(n) + 3, n > 0$

Example

- $f(0) = 1$
- $f(n+1) = (n+1) \times f(n), n > 0$

Recursively defined sets

- Basic step: define a basic set
- Recursive step: specify rules for generating the set from existing sets

Example

- Basic step: 3 is an element of the set S
- Recursive step: if $x \in S$ and $y \in S$ then $x + y \in S$

Example

formula

- Basic step: if x is a variable or constant, then x is a formula
- Recursive step: if f and g are formula, then $(f + g), (f - g), (f * g), (f/g), (f^g)$ are formula

Outline

Recursive algorithm

- An algorithm calls it-self with smaller input
- Suitable to process recursively defined objects
- High level programming languages allow users to design recursive functions and procedures

Algorithm 1: RecursiveAlgo(*input*)

```
if input has smallest size then
    process basic step;
else
    RecursiveAlgo(input with smaller size);
    Combine results of subproblems for obtaining the result R;
    return R
```

Outline

Compute n!

```
int fact(int n){
if(n==0)
    return 1;
else
    return n*fact(n-1);
}
```

Listing 1: Compute n!

Compute Fibonacci number

```
int FibRec(int n){
  if(n<=1)
    return 1;
  else
    return FibRec(n-1) + FibRec(n-2);
}
```

Listing 2: Compute the n^{th} fibonacci number

Compute $\binom{n}{k}$

```
int C(int n, int k){
  if(k==0 || k==n)
    return 1;
  else
    return C(n-1,k-1) + C(n-1,k);
}
```

Listing 3: Compute the n^{th} fibonacci number

Binary search

- An array of numbers $A[0..n-1]$ with $A[i] \leq A[i+1], \forall i = 0, \ldots, n-2$
- Given a value X, find the position i such that $A[i] = X$

```
int bSearch(int* A, int start, int finish, int X){
  if(start==finish){
    if(A[start]==X)    return start;
    else   return -1;/* not found*/
  }else{
    int mid = (start+finish)/2;
    if(A[mid]==X) return mid;
    else if(X < A[mid]) return bSearch(A,start, mid-1,X);
    else return bSearch(A,mid+1,finish ,X);
  }
}
```

Listing 4: Binary search

Hanoi tower

```
void move(int n, char start, char finish, char spare){
  if(n==1){
    printf("Move disk from %c to %c\n",start,finish);
  }else{
    move(n-1,start,spare,finish);
    move(1,start,finish,spare);
    move(n-1,spare,finish,start);
  }
}

int main(){
  int n = 5;
  move(n,'A','B','C');
  return 0;
}
```

Listing 5: Hanoi tower

Outline

Recursive algorithms analysis

Algorithm 2: D-and-C(n)

```
if n ≤ n_0 then
    Process directly;
else
    Divide the problem into a subproblems with input size n/b;
    foreach subproblem do
        D-and-C(n/b);
    Combine solutions of a subproblems for obtaining the result of initial problem;
```

- $T(n)$: time complexity of the problem with input size n
- Division takes $D(n)$
- Combination takes $C(n)$
- Recursive definition of $T(n)$

$$T(n) = \begin{cases} \Theta(1), & n \le n_0 \\ aT(n/b) + D(n) + C(n), & n > n_0 \end{cases}$$

Recursive algorithms analysis

Algorithm 3: D-and-C(n)

```
if n ≤ n_0 then
    Process directly;
else
    Divide the problem into a subproblems with input size n/b;
    foreach subproblem do
        D-and-C(n/b);
    Combine solutions of a subproblems for obtaining the result of initial problem;
```

- $T(n)$: time complexity of the problem with input size n
- Division takes $D(n)$
- Combination takes $C(n)$
- Recursive definition of $T(n)$

$$T(n) = \begin{cases} \Theta(1), & n \leq n_0 \\ aT(n/b) + D(n) + C(n), & n > n_0 \end{cases}$$

Master theorem

$T(n) = aT(n/b) + cn^k$ with $a \geq 1, b > 1, c > 0$ are constant

- If $a > b^k$, then $T(n) = \Theta(n^{log_b a})$
- If $a = b^k$, then $T(n) = \Theta(n^k logn)$ with $log\, n = log_2 n$
- If $a < b^k$, then $T(n) = \Theta(n^k)$

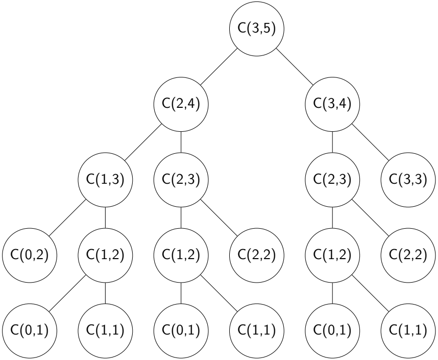
Example

- $T(n) = 3T(n/4) + cn^2 \Rightarrow T(n) = \Theta(n^2)$
- $T(n) = 2T(n/2) + n^{0.5} \Rightarrow T(n) = \Theta(n)$
- $T(n) = 16T(n/4) + n \Rightarrow T(n) = \Theta(n^2)$
- $T(n) = T(3n/7) + 1 \Rightarrow T(n) = \Theta(logn)$

Outline

Recursive algorithms with memory

- In many cases, the identical subproblems appear very often
- Use memory for storing results of subproblems solved to avoid resolving them



Recursive algorithms with memory

```
long D[100][100];
long C(int k, int n){
  if(k == 0 || k == n) D[k][n] = 1;
  else{
    if(D[k][n] <= 0)
      D[k][n] = C(k-1,n-1) + C(k,n-1);
  }
  return D[k][n];
}
```

Listing 6: recursive algorithms with memory

Outline

Introduction

- List all configurations satisfying some given constraints
  - permutations
  - subsets of a given set
  - etc.
- $A_1, \ldots, A_n$ are finite sets and $X = \{(a_1, \ldots, a_n) \mid a_i \in A_i, \forall 1 \leq i \leq n\}$
- $\mathcal{P}$ is a property on X
- Generate all configurations $(a_1, \ldots, a_n)$ having $\mathcal{P}$

Introduction

- In many cases, listing is a final way for solving some combinatorial problems
- Two popular methods
  - Generating method (not consider)
  - BackTracking algorithm

BackTracking algorithm

Construct elements of the configuration step-by-step

- Initialization: Constructed configuration is null: ()
- Step 1:
  - Compute (base on $\mathcal{P}$) a set S_1 of candidates for the first position of the configuration under construction
  - Select an item of S_1 and put it in the first position

At Step k: Suppose we have partial configuration $a_1, \ldots, a_{k-1}$

- Compute (base on $\mathcal{P}$) a set S_k of candidates for the k^{th} position of the configuration under construction
  - If $S_k \neq \emptyset$, then select an item of S_k and put it in the k^{th} position and obtain $(a_1, \ldots, a_{k-1}, a_k)$
    - If $k = n$, then process the complete configuration $a_1, \ldots, a_n)$
    - Otherwise, construct the $k+1^{th}$ element of the partial configuration in the same schema
  - If $S_k = \emptyset$, then backtrack for trying another item a'_{k-1} for the $k-1^{th}$ position
    - If a'_{k-1} exists, then put it in the $k-1^{th}$ position
    - Otherwise, backtrack for trying another item for the $k-2^{th}$ position, ...

BackTracking algorithm

Algorithm 4: BackTracking(k)

```
Construct a candidate set S_k;
foreach y ∈ S_k do
    a_k ← y;
    if (a_1, ..., a_k) is a complete configuration then
        ProcessConfiguration(a_1, ..., a_k);
    else
        BackTracking(k + 1);
```

Algorithm 5: Main()

```
BackTracking(1);
```

BackTracking algorithm - binary sequence

- A configuration is represented by $b_1, b_2, \ldots, b_n$
- Candidates for b_i is $\{0, 1\}$

BackTracking algorithm - binary sequence

```
void BackTracking(int k){
  for(int i = 0; i <= 1; i++){
    b[k] = i;
    if(k == n)
      printConfiguration();
    else
      BackTracking(k+1);
  }
}

void main(){
  BackTracking(1);
}
```

BackTracking algorithm - combination

- A configuration is represented by $(c_1, c_2, \ldots, c_k)$
  - dummy $c_0 = 1$
  - Candidates for c_i being aware of $\langle c_1, c_2, \ldots, c_{i-1} \rangle$:
    $c_{i-1} + 1 \leq c_i \leq n - k + i, \forall i = 1, 2, \ldots, k$

BackTracking algorithm - combination

```
void BackTracking(int i){
  for(int j = c[i-1]+1; j <= n-k+i; j++){
    c[i] = j;
    if(i == k){
     printConfiguration();
    }else
      BackTracking(i+1);
  }
}

void main(){
  c[0] = 0;
  BackTracking(1);
}
```

BackTracking algorithm - permutation

- A configuration: $p_1, p_2, \ldots, p_k$
- Candidates for p_i being aware of $\langle p_1, p_2, \ldots, p_{i-1} \rangle$: $\{1, 2, \ldots, n\} \setminus \{p_1, p_2, \ldots, p_{i-1}\}$
- Use an array of booleans for making values used $b_1, b_2, \ldots, b_n$
  - $b_v = 1$, if value v is already used (apprear in $p_1, p_2, \ldots, p_{i-1}$)
  - $b_v = 0$, otherwise

```
void BackTracking(int k){
  for(int i = 1; i <= n; i++){
    if(!b[i]){
      p[k] = i;
      b[i] = 1;
      if(k == n){
        printConfiguration();
      }else
        BackTracking(k+1);
      b[i] = 0;
    }
  }
}

void main(){
  for(int i = 1; i <= n; i++)
    b[i] = 0;
  BackTracking(1);
}
```

Solve the linear equations in a set of positive integers

$$a_1x_1 + a_2x_2 + \cdots + a_nx_n = M$$

where $(a_i)_{1\leq i\leq n}$ and M are positive integers

- Partial solution $(x_1, x_2, \ldots, x_{k-1})$
- $m = \sum_{i=1}^{k-1} a_ix_i$
- $A = \sum_{i=k+1}^{n} a_i$
- $\overline{M} = M - m - A$
- Candidates of x_k is $\{v \in \mathbb{Z} \mid 1 \leq v \leq \frac{\overline{M}}{a_k}\}$

BackTracking algorithm - Linear integer equation

```
void TRY(int k){ // try a value for variable x[k]
  for(int val = 1; val <= (M-m-A)/a[k]; val++){
    x[k] = val;
    m = m + a[k]*x[k];
    A = A - a[k];
    if(k == n){
      if(m==M)
      printSolution();
    }else
      TRY(k+1);
    m = m - a[k]*x[k];
    A = A + a[k];
  }
}
int main(int argc, char** argv){
  m = 0;
  A = 0;
  for(int i = 2; i <= n; i++)
    A = A + a[i];
  TRY(1);
}
```

BackTracking algorithm - n-queens problem

- Problem: Place n queens on a chess board such that no two queens attack each other
- Solution model: $(x_1, x_2, \ldots, x_n)$ where x_i represents the row on which the queen in column i is located
- Constraints:
  - $x_i \neq x_j, \forall 1 \leq i < j \leq n$
  - $|x_i - x_j| \neq |i - j|, \forall 1 \leq i < j \leq n$

BackTracking algorithm - n-queens problem

```
int x[100];
int n;
int candidate(int k, int v){
  for(int i = 1; i <= k-1; i++)
    if(x[i] == v || abs(x[i]-v)==abs(i-k)) return 0;
  return 1;
}
void BTrack(int k){
  for(int v = 1; v <= n; v++)
    if(candidate(k,v) == 1){
      x[k] = v;
      if(k == n)
        printSolution();
      else
        BTrack(k+1);
    }
}
int main(int argc, char** args){
  n = 8;
  BTrack(1);
}
```

BackTracking algorithm - n-queens problem - refinement

- Use arrays for marking forbidden cells
  - $r[1..n]$: $r[i] =$ false if the cells on row i are forbidden
  - $d_1[1-n..n-1]$: $d_1[q] =$ false if cells (r, c) s.t. $c - r = q$ are forbiden
    - in C++, indices of elements of an array cannot be negative (i.e., indices are 0, 1, ...). Hence making a deplacement: $d_1[q+n-1]$ instead of $d_1[q]$
  - $d_2[2..2n-2]$: $d_2[q]$ =false if cells (r, c) s.t. $r + c = q$ are forbiden

BackTracking algorithm - n-queens problem

```
void BTrack(int i){// try values for x[i]
  for(int val = 1; val <= n; val++){
    if(r[val] == true && d1[i-val+n-1] == true && d2[i+val] == true){
      x[i] = val;
      r[val] = false;// marking forbiden cells
      d1[i-val+n-1] = false;// marking forbiden cells
      d2[i+val] = false;// marking forbiden cells
      if(i == n){
        printSolution();
      }else
        BTrack(i+1);
      r[val] = true;// recovering marking
      d1[i-val+n-1] = true;// recovering marking
      d2[i+val] = true;// recovering marking
    }
  }
}
```

BackTracking algorithm - n-queens problem

```
int main(int argc, char** argv){
  n = atoi(argv[1]);

  for(int i = 1; i <= n; i++)
    r[i] = true;
  for(int i = 0; i <= 2*n; i++){
    d1[i] = true;
    d2[i] = true;
  }

  BTrack(1);
}
```

BackTracking algorithm - Exercises

- Sudoku problem
- Subset Sum problem
- List all the ways to decompose a positive integer N into a sum of positive integers