

CSC 427: Data Structures and Algorithm Analysis

Fall 2004

Problem-solving approaches

- divide & conquer
- greedy
- backtracking
- dynamic programming

1

Divide & Conquer

RECALL: the divide & conquer approach tackles a complex problem by breaking it into smaller pieces, solving each piece, and combining them into an overall solution

- e.g., merge sort divided the list into halves, conquered (sorted) each half, then merged the results
- e.g., to count number of nodes in a binary tree, break into counting the nodes in each subtree (which are smaller), then adding the results + 1

divide & conquer is applicable when a problem can naturally be divided into independent pieces

sometimes, the pieces to be conquered can be handled in sequence

- i.e., arrive at a solution by making a sequence of choices/actions
- in these situations, we can consider specialized classes of algorithms
 - greedy algorithms
 - backtracking algorithms

2

Greedy algorithms

the greedy approach to problem solving involves making a sequence of choices/actions, each of which simply looks best at the moment

local view: choose the locally optimal option
hopefully, a sequence of locally optimal solutions leads to a globally optimal solution

example: optimal change

- given a monetary amount, make change using the fewest coins possible

amount = 16¢ coins?

amount = 96¢ coins?

3

Example: greedy change

while the amount remaining is not 0:

- select the largest coin that is \leq the amount remaining
- add a coin of that type to the change
- subtract the value of that coin from the amount remaining

e.g., $96¢ = 50¢ + 25¢ + 10¢ + 10¢ + 1¢$

will this greedy algorithm always yield the optimal solution?

for U.S. currency, the answer is YES

for arbitrary coin sets, the answer is NO

- suppose the U.S. Treasury added a 12¢ coin

GREEDY: $16¢ = 12¢ + 1¢ + 1¢ + 1¢ + 1¢$ (5 coins)

OPTIMAL: $16¢ = 10¢ + 5¢ + 1¢$ (3 coins)

4

Greed is good?

IMPORTANT: the greedy approach is not applicable to all problems

- but when applicable, it is very effective (no planning or coordination necessary)

example: job scheduling

- suppose you have a collection of jobs to execute and know their lengths
- want to schedule the jobs so as to *minimize* waiting time

Job 1:	5 minutes	Schedule 1-2-3: $0 + 5 + 15 = 20$ minutes waiting
Job 2:	10 minutes	Schedule 3-2-1: $0 + 4 + 14 = 18$ minutes waiting
Job 3:	4 minutes	Schedule 3-1-2: $0 + 4 + 9 = 13$ minutes waiting

GREEDY ALGORITHM: do the shortest job first

i.e., while there are still jobs to execute, schedule the shortest remaining job

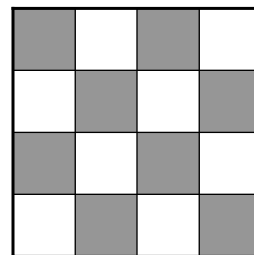
does the greedy algorithm guarantee the optimal schedule? efficiency?

5

Example: N-queens problem

given an $N \times N$ chess board, place a queen on each row so that no queen is in jeopardy

GREEDY algorithm: start with first row, find a valid position in current row, place a queen in that position then move on to the next row



since queen placements are not independent, locally choices do not necessarily lead to a global solution

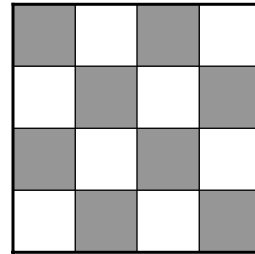
GREEDY does not work – need a more wholistic approach

6

Generate & test

we could take an extreme approach to the N-queens problem

- systematically generate every possible arrangement
- test each one to see if it is a valid solution



this will work (in theory), but the size of the search space may be prohibitive

4x4 board → $16 \cdot 15 \cdot 14 \cdot 13 = 43,680$ arrangements

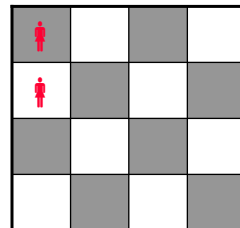
8x8 board → $64 \cdot 63 \cdot 62 \cdot 61 \cdot 60 \cdot 59 \cdot 58 \cdot 57 = 178,462,987,637,760$

7

Backtracking

if we were smart, we could greatly reduce the search space

- e.g., any board arrangement with a queen at (1,1) and (2,1) is invalid
- no point in looking at the other queens, so can eliminate 12 boards from consideration



backtracking is a smart way of doing generate & test

- view a solution as a sequence of choices/actions
- when presented with a choice, pick one (similar to GREEDY)
- however, reserve the right to change your mind and backtrack to a previous choice (unlike GREEDY)
- you must remember alternatives:
if a choice does not lead to a solution, back up and try an alternative
- eventually, backtracking will find a solution or exhaust all alternatives

8

Chessboard class

similar to `Pixel`, we could define a class hierarchy for chess pieces

- `ChessPiece` is an abstract class that specifies the common behaviors of pieces
- `Queen`, `Knight`, `Pawn`, ... are derived from `ChessPiece` and implement specific behaviors

```
#include "ChessPiece.h"

class ChessBoard
{
public:
    ChessBoard(int size = 8); // constructs size-by-size board
    ChessPiece * getPiece(int row, int col) const; // returns piece at (row,col)
    void removePiece(int row, int col); // removes piece at (row,col)
    void putPiece(int row, int col, ChessPiece * p); // places a piece, e.g., a queen,
                                                    // at (row,col)
    bool inJeopardy(int row, int col) const; // returns true if (row,col) is
                                                    // under attack by any piece
    void display() const; // displays the board
    int numPieces() const ; // returns number of pieces on board
    int size() const; // returns the board size
private:
    int pieceCount; // keeps track of # of pieces
    Matrix<ChessPiece *> board; // represents the square board
};
```

9

Backtracking N-queens

```
bool PlaceQueens(ChessBoard & board, int row)
// precondition : row < board.Size()
// postcondition: places queens on board starting at row
{
    if (row > board.size()) {
        return true;
    }
    else {
        for (int col = 0; col < board.size(); col++) {
            if (!board.inJeopardy(row, col)) {
                board.PutPiece(row, col, new Queen());

                if (PlaceQueens(board, row+1)) {
                    return true;
                }
                else {
                    board.RemovePiece(row, col);
                }
            }
        }
        return false;
    }
}
```

function is called with starting row number (initially 0)

BASE CASE: if all queens have been placed, then done.

OTHERWISE: try placing queen in the row and recurse to place the rest

note: if recursion fails, must remove the queen in order to backtrack

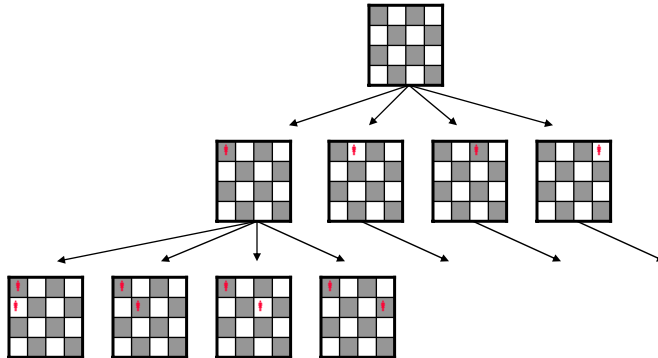
10

Why does backtracking work?

backtracking burns no bridges – all choices are reversible

think of the search space as a tree

- root is the initial state of the problem (e.g., empty board)
- at each step, multiple choices lead to a branching of the tree
- solution is a sequence of choices (path) that leads from start state to a goal state



11

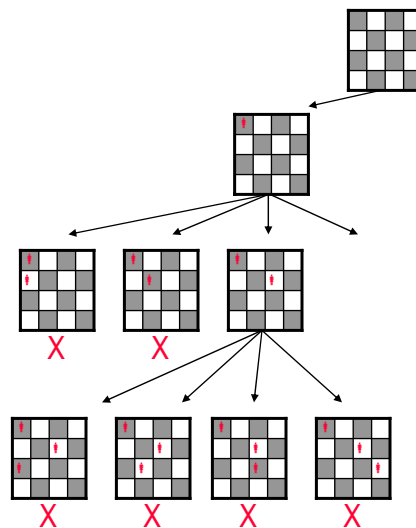
backtracking vs. generate & test

backtracking provides a systematic way of trying all paths (sequences of choices) until a solution is found

- worst case: exhaustively tries all paths, traversing the entire search space

backtracking is different from generate & test in that choices are made sequentially

- earlier choices constrain later ones
- can avoid searching entire branches

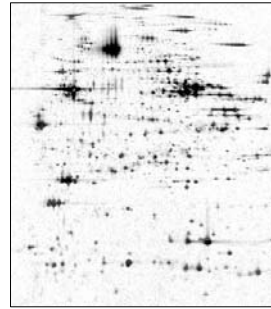


12

Example: blob count

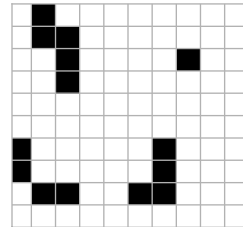
application: 2-D gel electrophoresis

- biologists use electrophoresis to produce a gel image of cellular material
- each "blob" (contiguous collection of dark pixels) represents a protein
- identify proteins by matching the blobs up with another known gel image



we would like to identify each blob, its location and size

- location is highest & leftmost pixel in the blob
- size is the number of contiguous pixels in the blob
- in this small image:
 - pixel[0][1]: size 5
 - pixel[2][7]: size 1
 - pixel[6][0]: size 4
 - pixel[6][6]: size 4



13

Gel class

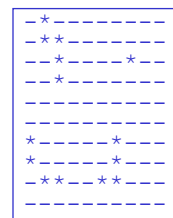
```
class Gel{
public:
    Gel(); // constructor (0-sized Gel)
    void Read(istream & myin = cin); // read Gel from file
    void Write(ostream & myout = cout); // write Gel to file
    void DisplayBlobs(); // list all blobs with sizes

private:
    Matrix<char> gel; // 2-d array of "pixels"

    int BlobCount(int row, int col); // recursively counts size of blob
};
```

for simplicity, a gel image is stored internally as a Matrix of chars

- DisplayBlobs traverses the gel, pixel-by-pixel
 - if a blob is encountered, expand in all directions (i.e., backtrack) to explore the entire blob



14

Gel methods

```
void Gel::DisplayBlobs()
// precondition: displays location (top-left) and size of blobs
{
    for (int r = 0; r < gel.rows(); r++) {
        for (int c = 0; c < gel.cols(); c++) {
            if (gel[r][c] == '*') {
                cout << "Blob at [" << r << "]" << c
                    << " : size " << BlobCount(r, c) << endl;
            }
        }
    }
}

int Gel::BlobCount(int row, int col)
// precondition : 0 <= row < gel.rows(), 0 <= col < gel.cols()
// precondition: recursively counts (and erases) blob connected
// to specified row & column
{
    if (row < 0 || row >= gel.rows() ||
        col < 0 || col >= gel.cols() || gel[row][col] != '*') {
        return 0;
    }
    else {
        gel[row][col] = 'O';

        return 1 + BlobCount(row-1, col-1)
            + BlobCount(row-1, col)
            + BlobCount(row-1, col+1)
            + BlobCount(row, col-1)
            + BlobCount(row, col+1)
            + BlobCount(row+1, col-1)
            + BlobCount(row+1, col)
            + BlobCount(row+1, col+1);
    }
}
```

DisplayBlobs traverses the image, checks each pixel for a blob

BlobCount uses backtracking to expand in all directions once a blob is found

note: each pixel is "erased" after it is processed to avoid double-counting (& infinite recursion)

WHAT HAPPENS TO THE IMAGE WHEN DONE?

15

HW 6: backtracking Boggle

```
bool BoggleBoard::contains(string word)
{
    for (int r = 1; r <= BOARD_SIZE; r++) {
        for (int c = 1; c <= BOARD_SIZE; c++) {
            if (contains(word, r, c)) {
                return true;
            }
        }
    }
    return false;
}

bool BoggleBoard::contains(string word, int r, int c)
{
    if (word == "") {
        return true;
    }
    else if (board[r][c] != word[0]) {
        return false;
    }
    else {
        char safe = board[r][c];
        board[r][c] = '*';
        string rest = word.substr(1, word.length()-1);
        bool result = (contains(rest, r-1, c-1) || contains(rest, r-1, c) ||
            contains(rest, r-1, c+1) || contains(rest, r, c-1) ||
            contains(rest, r, c+1) || contains(rest, r+1, c-1) ||
            contains(rest, r+1, c) || contains(rest, r+1, c+1));
        board[r][c] = safe;
        return result;
    }
}
```

main function checks each letter in the board to see if the word starts there – calls recursive helper function

BASE CASES: empty string or first letter doesn't match

OTHERWISE: mark the letter so can't use again, recurse on rest of word (from neighbor letters), restore letter before done

16

Recent programming contest problem

November 13, 2004

ACM North Central North America Regional Programming Contest

Problem 1

Problem 1: Breaking a Dollar

Using only the U. S. coins worth 1, 5, 10, 25, 50, and 100 cents, there are exactly 293 ways in which one U. S. dollar can be represented. Canada has no coin with a value of 50 cents, so there are only 243 ways in which one Canadian dollar can be represented. Suppose you are given a new set of denominations for the coins (each of which we will assume represents some integral number of cents less than or equal to 100, but greater than 0). In how many ways could 100 cents be represented?

Input

The input will contain multiple cases. The input for each case will begin with an integer N (at least 1, but no more than 10) that indicates the number of unique coin denominations. By *unique* it is meant that there will not be two (or more) different coins with the same value. The value of N will be followed by N integers giving the denominations of the coins.

Input for the last case will be followed by a single integer -1.

Output

For each case, display the case number (they start with 1 and increase sequentially) and the number of different combinations of those coins that total 100 cents. Separate the output for consecutive cases with a blank line.

Sample Input

```
6 1 5 10 25 50 100
5 1 5 10 25 100
-1
```

Output for the Sample Input

```
Case 1: 293 combinations of coins
Case 2: 243 combinations of coins
```

17

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;
```

```
int GetChange(int amount, vector<int> coins, int maxCoinIndex);
```

```
int main()
```

```
{
    int numCoins, caseNum = 1;
    while (cin >> numCoins && numCoins != -1) {
        vector<int> coins; int c;
        for (int i = 0; i < numCoins; i++) {
            cin >> c;
            coins.push_back(c);
        }
        sort(coins.begin(), coins.end());

        int count = GetChange(100, coins, coins.size()-1);

        if (caseNum > 1) cout << endl;
        cout << "Case " << caseNum++ << ": " << count << " combinations of coins" << endl;
    }
    return 0;
}
```

Divide & conquer solution

count how many ways to get 100 using all possible coins

```
int GetChange(int amount, vector<int> coins, int maxCoinIndex)
```

```
{
    if (maxCoinIndex < 0 || amount < 0) {
        return 0;
    }
    else if (amount == 0) {
        return 1;
    }
    else {
        return GetChange(amount-coins[maxCoinIndex], coins, maxCoinIndex) +
            GetChange(amount, coins, maxCoinIndex-1);
    }
}
```

recursion: count how many ways using a largest coin + how many ways not using a largest coin

18

Caching version

```
int main()
{
    . . .

    int remember[101][11];

    for (int r = 0; r < 101; r++) {
        for (int c = 0; c < 11; c++) {
            remember[r][c] = -1;
        }
    }

    int count = GetChange(100, coins, coins.size()-1);

    . . .
}

int GetChange(int amount, vector<int> coins, int maxCoinIndex)
{
    if (remember[amount][maxCoinIndex] == -1) {
        if (maxCoinIndex < 0 || amount < 0) {
            remember[amount][maxCoinIndex] = 0;
        }
        else if (amount == 0) {
            remember[amount][maxCoinIndex] = 1;
        }
        else {
            remember[amount][maxCoinIndex] =
                GetChange(amount-coins[maxCoinIndex], coins, maxCoinIndex) +
                GetChange(amount, coins, maxCoinIndex-1);
        }
        return remember[amount][maxCoinIndex];
    }
}
```

with caching, even worst
case answer is immediate:
6,292,069 combinations

21

Dynamic programming

caching solutions to smaller problems, building up to the goal is known as *dynamic programming*

- applicable to same types of problems as divide & conquer
- bottom-up, as opposed to divide & conquer which is top-down
- usually more effective than top-down if the parts are not completely independent (thus leading to redundancy)

silly Fibonacci example: top-down divide & conquer

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

smarter Fibonacci : bottom-up dynamic programming

```
int Fib(int n)
{
    int prev = 1, current = 1;
    for (int i = 1; i < n; i++) {
        int next = prev + current;
        prev = current;
        current = next;
    }
    return current;
}
```

22

Example: binary coefficient

binomial coefficient $C(n, k)$ is relevant to a large number of problems

- the number of ways you can select k lottery balls out of n
- the number of ways to get k heads when a coin is tossed n times,
- the number of birth orders possible in a family of n children where k are sons
- the number of acyclic paths connecting 2 corners of an $k \times (n-k)$ grid

- the coefficient of the $x^k y^{n-k}$ term in the polynomial expansion of $(x + y)^n$
- the entry at the n th row and k th column of Pascal's triangle

$$C(n, k) \equiv \binom{n}{k} \equiv \frac{n!}{k!(n-k)!}$$

23

Example: binary coefficient

while easy to define, a binomial coefficient is difficult to compute

e.g. 6 number lottery with 49 balls $\rightarrow 49!/6!43!$

$49! = 608,281,864,034,267,560,872,252,163,321,295,376,887,552,831,379,210,240,000,000,000$

could try to get fancy by canceling terms from numerator & denominator

- can still end up with individual terms that exceed integer limits

a computationally easier approach makes use of the following recursive relationship

$$\binom{n}{k} \equiv \binom{n-1}{k-1} + \binom{n-1}{k}$$

e.g., to select 6 lottery balls out of 49, partition into:

selections that include 1
(must select 5 out of remaining 48)

+

selections that don't include 1
(must select 6 out of remaining 48)

24

Example: binomial coefficient

could use straight divide&conquer to compute based on this relation

```
int Binomial(int n, int k)
// Assumes: 0 < k <= n
// Returns: n choose k (using divide-and-conquer approach)
{
    if (k == 0 || n == k) {
        return 1;
    }
    else {
        return Binomial(n-1, k-1) + Binomial(n-1, k);
    }
}
```

however, this will take a long time or exceed memory due to redundant work

$$\binom{47}{4} + \binom{48}{5} + \binom{49}{6} + \binom{47}{5} + \binom{48}{6} + \binom{47}{6}$$

25

Dynamic programming solution

could use caching to store answers, or build a solution entirely from the bottom-up (starting at base cases)

	0	1	2	3	...	k
0	1					
1	1	1				
2	1		1			
3	1			1		
...	
n	1					1

```
int Binomial(int n, int k)
// Assumes: 0 < k <= n
// Returns: n choose k (using dynamic programming approach)
{
    if (n < 2) {
        return 1;
    }
    else {
        vector< vector<int> > bin(n+1); // CONSTRUCT A TABLE TO STORE
        for (int i = 0; i <= n; i++) { // COMPUTED VALUES
            bin[i].resize(k+1);

            for (int j = 0; j <= i && j <= k; j++) {
                if (j == 0 || j == i) {
                    bin[i][j] = 1; // ENTER 1 IF BASE CASE
                }
                else {
                    bin[i][j] = bin[i-1][j-1] + bin[i-1][j]; // OTHERWISE, USE FORMULA
                }
            }
        }
        return bin[n][k]; // ANSWER IS AT bin[n][k]
    }
}
```

26

World series puzzle

Consider the following puzzle:

At the start of the world series (best-of-7), you must pick the team you want to win and then bet on games so that

- if your team wins the series, you win exactly \$1,000
- if your team loses the series, you lose exactly \$1,000

You may bet different amounts on different games, and can even bet \$0 if you wish.

QUESTION: how much should you bet on the first game?

27

Algorithmic approaches summary

divide & conquer: tackles a complex problem by breaking it into smaller pieces, solving each piece, and combining them into an overall solution

- often involves recursion, if the pieces are similar in form to the original problem
- applicable for any application that can be divided into independent parts

greedy: involves making a sequence of choices/actions, each of which simply looks best at the moment

- applicable when a solution is a sequence of moves & perfect knowledge is available

backtracking: involves making a sequence of choices/actions (similar to greedy), but stores alternatives so that they can be attempted if the current choices lead to failure

- more costly in terms of time and memory than greedy, but general-purpose
- worst case: will have to try every alternative and exhaust the search space

dynamic: bottom-up implementation of divide & conquer – start with the base cases and build up to the desired solution, caching results to avoid redundant computation

- usually more effective than top-down recursion if the parts are not completely independent (thus leading to redundancy)
- note: can approximate using top-down recursion with caching

28