CS310 - Advanced Data Structures and Algorithms

Spring 2014 – Class 12

March 13, 2014

Problem – Making Change

- Task – buy a cup of coffee (say it costs 89 cents).
- You are given an unlimited number of coins of all types (neglect 50 cents and 1 dollar).
- Pay exact change.
- What is the combination of coins you'd use?

1 cent

5 cents

10 cents

25 cents

Greedy Algorithms – Change Making

- Logically, we'd minimize the number of coins.
- Change-making with the fewest number of US coins – have 1, 5, 10, 25 unit coins to work with.
- Clearly we want to mainly use large-value coins to minimize the total number.
- So for 27 cents, clearly we can't do better than $25 + 2(1)$.
- What about 89? Use as many 25s as fit, $89 = 3(25) + 14$, then as many 10s as fit in the remainder: $89 = 3(25) + 1(10) + 4$, no 5's fit, so we have $89 = 3(25) + 1(10) + 4(1)$, 8 coins.

Greedy Algorithms

- A greedy person grabs everything they can as soon as possible.
- Similarly a greedy algorithm makes locally optimized decisions that appear to be the best thing to do at each step.
- Example: Change-making greedy algorithm for “change” amount, given many coins of each size:
  - Loop until change == 0:
  - Find largest-valued coin less than change, use it.
  - change = change - coin-value;

Change Making

- The greedy method gives the optimal solution for US coinage.
- With different coinage, the greedy algorithm doesn't always find the optimal solution.
- Example of a coinage with an additional 21 cent piece. Then $63 = 3(21)$, but greedy says use 2 25s, 1 10, and 3 1's, a total of 6 coins.
- The coin values need to be spread out enough to make greedy work.
- But even some spread-out cases don't work. Consider having pennies, dimes and quarters, but no nickels.
- Then 30 by greedy uses 1 quarter and 5 pennies, ignoring the best solution of 3 dimes.

(Very bad) Recursive Solution

Example: change for 63 cents with coins $= \{25, 10, 5, 1, 21\}$ no order required in array.

```
makeChange(63)
minCoins = 63
loop over j from 1 to 63/2 = 31
    thisCoins = makeChange(j) + makeChange(63-j)
```

Lots and lots of redundant calls!

(Very bad) Recursive Solution

$T(n) = T(n\text{-}1) + T(n\text{-}2) + T(n\text{-}3) + \ldots + T(n/2) + \ldots$

worse than $T(n) = T(n\text{-}1) + T(n\text{-}2)$ the famous Fibonacci sequence discussed on pg. 242 (and hw1). Fibonacci is exponential, so this certainly is.

Better Idea

- We know we have 1,5,10,21 and 25.
- Therefore, the optimal solution must be the minimum of the following:
  - 1 (A 1 cent) + optimal solution for 62.
  - 1 (A 5 cent) + optimal solution for 58.
  - 1 (A 10 cent) + optimal solution for 53.
  - 1 (A 21 cent) + optimal solution for 42.
  - 1 (A 25 cent) + optimal solution for 38.
  - This reduces the number of recursive calls drastically.
- Naive implementation still makes lots of redundant calls.

Dynamic Programming Implementation

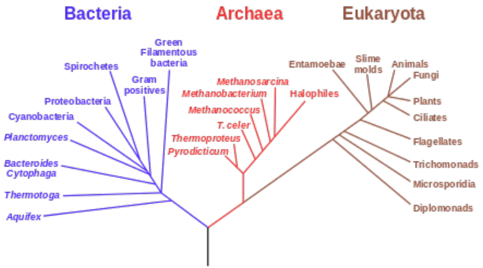
- Idea – instead of performing the same calculation over and over again, save pre-calculated results to an array.
- The answer to a large change depends only on results of smaller calculations, so we can calculate the optimal answer for all the smaller change and save it to an array.
- Then go over the array and minimize on:
  - $change(K) = \min\{change(K - n) + 1\}$
  - For all n types of coins
- Runtime – $O(N * K)$.

Dynamic Programming for Change Problem

```
 public static void makeChange( int [] coins, int differentCoins,
int maxChange, int [] coinsUsed, int [] lastCoin )
{
    coinsUsed[ 0 ] = 0; lastCoin[ 0 ] = 1;
    for( int cents = 1; cents <= maxChange; cents++ ) {
        int minCoins = cents;
        int newCoin = 1;
        for( int j = 0; j < differentCoins; j++ ) {
            if(coins[j] > cents) continue; // Cannot use coin j
            if(coinsUsed[ cents - coins[j] ] + 1 < minCoins) {
                minCoins = coinsUsed[ cents - coins[j] ] + 1;
                newCoin = coins[j];
            }
        }
        coinsUsed[ cents ] = minCoins;
        lastCoin[ cents ] = newCoin;
    }
}
```

Application – Sequence Alignment

Phylogenetic Tree of Life




```
A5ASC3.1   14 SIKLWPPSQTTRLLLVERMANNLST..PSIFTRK..YGSLSKEEARENAKQIEEVACSTANQ.....HYEKEPDGDGGSAVQLYAKECSKLILEVLK 101
B4F917.1   13 SIKLWPPSESTRIMLVDRMTNNLST..ESIFSRK..YRLLGKQEAHENAKTIEELCFALADE.....HFREEPDGDGSSAVQLYAKETSKMMLEVLK 100
A9S1V2.1   23 VFKLWPPSQGTREAVRQKMALKLSS..ACFESQS..FARIELADAQEHARAIEEVAFGAAQE......ADSGGDKTGSAVVMVYAKHASKLMLETLR 109
B9GSN7.1   13 SVKLWPPGQSTRLMLVERMTKNFIT..PSFISRK..YGLLSKEEAEEDAKKIEEVAFAAANQ.....HYEKQPDGDGSSAVQIYAKESSRLMLEVLK 100
Q8H056.1   30 SFSIWPPTQRTRDAVVRRLVDTLGG..DTILCKR..YGAVPAADAEPAARGIEAEAFDAAAA..SGEAAATASVEEGIKALQLYSKEVSRRLLDFVK 120
Q0D4Z3.2   44 SLSIWPPSQRTRDAVVRRLVQTLVA..PSILSQR..YGAVPEAEAGRAAAAVEAEAYAAVTES.SSAAAAPASVEDGIEVLQAYSKEVSRRLLELAK 135
B9MVW8.1   56 SFSIWPPTQRTRDAIISRLIETLST..TSVLSKR..YGTIPKEEASEASRRIEEEAFSGAST.......VASSEKDGLEVLQLYSKEISKRMLETVK 141
Q0IYC5.1   29 SFAVWPPTRRTRDAVVRRLVAVLSGDTTTALRKRYRYGAVPAADAERAARAVEAQAFDAASA....SSSSSSSVEDGIETLQLYSREVSNRLLAFVR 121
A9NW46.1   13 SIKLWPPSESTRLMLVERMTDNLSS..VSFFSRK..YGLLSKEEAAENAKRIEETAFLAAND.....HEAKEPNLDDSSVVQFYAREASKLMLEALK 100
Q9C500.1   57 SLRIWPPTQKTRDAVLNRLIETLST..ESILSKR..YGTLKSDDATTVAKLIEEEAYGVASN.......AVSSDDDGIKILELYSKEISKRMLESVK 142
Q2HRI7.1   25 NYSIWPPKQRTRDAVKNRLIETLST..PSVLTKR..YGTMSADEASAAAIQIEDEAFSVANA.......SSSTSNDNVTILEVYSKEISKRMIETVK 110
Q9M7N3.1   28 SFKIWPPTQRTREAVVRRLVETLTS..QSVLSKR..YGVIPEEDATSAARIIEEEAFSVASV.ASAASTGGRPEDEWIEVLHIYSQEIXQRVVESAK 119
Q9M7N6.1   25 SFSIWPPTQRTRDAVINRLIESLST..PSILSKR..YGTLPQDEASETARLIEEEAFAAAGS.......TASDADDGIEILQVYSKEISKRMIDTVK 110
Q9LE82.1   14 SVKMWPPSKSTRLMLVERMTKNITT..PSIFSRK..YGLLSVEEAEQDAKRIEDLAFATANK.....HFQNEPDGDGTSAVHVYAKESSKLMLDVIK 101
Q9M651.2   13 SIKLWPPSLPTRKALIERITNNFSS..KTIFTEK..YGSLTKDQATENAKRIEDIAFSTANQ.....QFEREPDGDGGSAVQLYAKECSKLILEVLK 100
B9R748.1   48 SLSIWPPTQRTRDAVITRLIETLSS..PSVLSKR..YGTISHDEAESAARRIEDEAFGVANT.......ATSAEDDGLEILQLYSKEISRRMLDTVK 133
```

Binomial Coefficients

- Another famous example is the sequence of binomial coefficients
- Can be generated by Pascals triangle
- Each number is the sum of the two closest above it.

$$
\begin{array}{ccccccccc}
 & & & & 1 & & & & \\
 & & & 1 & & 1 & & & \\
 & & 1 & & 2 & & 1 & & \\
 & 1 & & 3 & & 3 & & 1 & \\
1 & & 4 & & 6 & & 4 & & 1 \\
 & 5 & & & & & & &
\end{array}
$$

Binomial Coefficients

- Start a new row with 1's on the edges.
- The row number is N, and the entries are k=0, k=1, ..., k=N across a row, so for example
- C(4,0) = 1, C(4,1) = 4, C(4,2) = 6, C(4,3) = 4, C(4,4) = 1.
- These are the coefficients of the binomial expansion
- $(x+y)^4 = x^4 + 4x^3y + 6x^2y^2 + 4xy^3 + y^4$

Binomial Coefficient and n Choose k

- Also, C(N, k) = number of ways to choose a set of k objects from N
- Ex. C(4, 2) = 6 The 2-sets of 4 numbers are the 6 sets: $\{1, 2\}$, $\{1, 3\}$, $\{1, 4\}$, $\{2, 3\}$, $\{2, 4\}$, $\{3, 4\}$
- Recursion: C(N, k) = C(N-1, k) + C(N-1, k-1) This is just the sum rule of Pascal's triangle.

n Choose k – Rationale

- Base cases: C(N, 0) = 1, C(N, N) = 1
- To choose k objects from N, set one object x aside and find all the ways of choosing k objects from the remaining N-1.
- These are all the sets we want that dont include x, C(N-1, k) in number.
- The sets that do include x also need k-1 other objects from the other N-1, C(N-1, k-1) in number.

Binomial Coefficient – Recursion

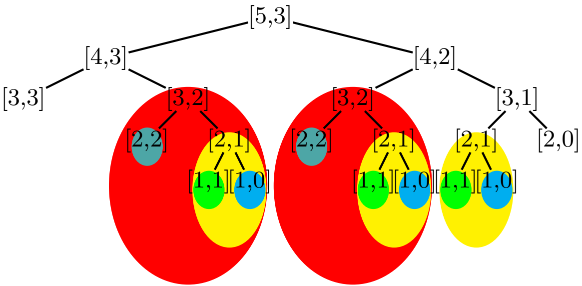
- f we write a recursive function:

```
 combo(N, k):
if (k == 0) return 1
if (k == N) return 1
return combo(N-1, k) + combo(N-1, k-1)
```

- Note the double recursion, without halving the "N" value, so dangerous recursion.

Binomial Coefficient – Recursion

- We get exponential T(N)
- T(N, k) = T(N-1, k) + T(N-1, k-1) – 2 terms in N-1
  = T(N-2, k) + ... 4 terms in N-2
  = ... some of these hit base cases and stop



Efficient Calculation of Binomial Coefficients

- If we save and reuse values, it's much faster.
- In other words, use Pascal's triangle to generate all the coefficients.
- One way: set up a table and use it for each N in turn.

```
 C[1][0] = 1
C[1][1] = 1
for n up to N
    for k up to n
        C[n][ k] = C[n-1][ k] + C[n-1][k-1]
```

- $O(1)$ to fill each spot in NxN array, so $O(N^2)$

Map Approach to Binomial Coefficients

- Another approach: set up Map from (N, k) to value.
- if N and k both ints, long key $=$ N $+$ (long)k$>>$32
- Case of classic dynamic programming, saving partial results along the way.

Map Approach to Binomial Coefficients

```
 combo(N, k):
val = M.get(key(N,k))
if (val != null) return val
if (k == 0) val = 1
if (k == N) val = 1
else val = combo(N-1, k) + combo(N-1, k-1)
M.put(key(N, k), val)
return val
```

once this recursion reaches a cell, fills it in, so work bounded by number of cells below (N, k), which is $< N^2$.

Maximum Contiguous Subsequence Sum

- Given a sequence of integers $(A_1, A_2, ..., A_N)$, possibly negative.
- Identify the subsequence $(A_i, ..., A_j)$ that corresponds to the maximum value of $\sum_i^j A_k$
- Naive approach is cubic (examine all $O(N^2)$ sequences and sum each one).
- Use a divide-and-conquer algorithm.

Divide-and-Conquer Algorithm

- Sample input is {4, -3, 5, -2, -1, 2, 6, -2}
- 3 possible cases:
  1. in the first half
  2. in the second half
  3. begins in the first half and ends in the second half

For case 3: sum = sum 1^{st} + sum 2^{nd}

$\leftarrow \qquad \rightarrow$

First Half				Second Half				
4	–3	5	–2	–1	2	6	–2	Values
4*	0	3	–2	–1	1	7*	5	Running sums

Running sum from the center (*denotes maximum for each half).

figure 7.19

Dividing the maximum contiguous subsequence problem into halves

Divide-and-Conquer Algorithm

- Case 3 is solved in linear time.
- Apply case 3's strategy to solve case 1 and 2
- Summary:
  - Recursively compute the max subsequence sum in the first half
  - Recursively compute the max subsequence sum in the second half
  - Compute, via 2 consecutive loops, the max subsequence sum that begins in the first half but ends in the second half
  - Choose the largest of the 3 sums

Divide-and-Conquer Algorithm

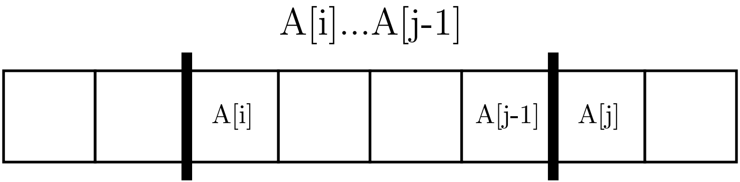
figure 7.20

A divide-and-conquer algorithm for the maximum contiguous subsequence sum problem

```
/**
 * Recursive maximum contiguous subsequence sum algorithm.
 * Finds maximum sum in subarray spanning a[left..right].
 * Does not attempt to maintain actual best sequence.
 */
private static int maxSumRec( int [ ] a, int left, int right )
{
    int maxLeftBorderSum = 0, maxRightBorderSum = 0;
    int leftBorderSum = 0, rightBorderSum = 0;
    int center = ( left + right ) / 2;

    if( left == right )  // Base case
        return a[ left ] > 0 ? a[ left ] : 0;

    int maxLeftSum  = maxSumRec( a, left, center );
    int maxRightSum = maxSumRec( a, center + 1, right );

    for( int i = center; i >= left; i-- )
    {
        leftBorderSum += a[ i ];
        if( leftBorderSum > maxLeftBorderSum )
            maxLeftBorderSum = leftBorderSum;
    }

    for( int i = center + 1; i <= right; i++ )
    {
        rightBorderSum += a[ i ];
        if( rightBorderSum > maxRightBorderSum )
            maxRightBorderSum = rightBorderSum;
    }

    return max3( maxLeftSum, maxRightSum,
                 maxLeftBorderSum + maxRightBorderSum );
}

/**
 * Driver for divide-and-conquer maximum contiguous
 * subsequence sum algorithm.
 */
public static int maxSubsequenceSum( int [ ] a )
{
    return a.length > 0 ? maxSumRec( a, 0, a.length - 1 ) : 0;
}
```

Dynamic Programming Solution

- Let's look at index j.
- The maximum contiguous subsequence ending at j (denoted MaxSum(j)) either extends a previous maximum subsequence (ending at j-1) or starts a new sum.
- The former happens if MaxSum(j-1) is positive.
- The latter happens if MaxSum(j-1) is non-positive.



Dynamic Programming Solution

- Therefore a dynamic programming solution for Max(j) is: $Max(j) = \max\{Max(j-1) + a[j], a[j]\}$ (constant for each j, considering that Max(j-1) was already computed).
- The overall solution to the problem is $\max_j Max(j)$.
- T(1) = O(1) – the maximum sum for a[0] is $\max\{a[0], 0\}$.
- T(N) = T(N-1) + O(1) – maximizing over two O(1) expressions.
- T(N) = O(N).