University of Zurich[UZH]

Solution

Department of Informatics
Database Technology Group
Prof. Dr. M. Böhlen

Informatics II
Exercise 11 / **Solution**

May 12, 2020

Dynamic programming

Task 1. Imagine several houses built in a circle. Festive illumination needs to be arranged for Christmas and each house has a given number of light bulbs to use for the event. In order to increase the effect of the illuminations, `only houses which are not next to each other (non-neighbor) shall participate`.

Given an array with the number of light bulbs per house, calculate the highest possible number of light bulbs which can be simultaneously lit up in the night. **Example:** Given the array lightBulbs[ ] $= \{17, 5, 10, 18, 22\}$, it would be optimal if the house 0 (with 17 light bulbs) and house 3 (with 18 light bulbs) would participate. Houses 0 and 4 cannot be combined since they are neighbors due to the circular structure.

Solve the illumination problem using (a) recursion, (b) memoization and (c) dynamic programming. Create a C program including the following functions:

```c
void copyArray(int source[], int dest[], int n) {
    int i;
    for(i = 0; i < n; i++) {
        dest[i] = source[i];
    }
}

/* First and last house are neighbors, so we calculate the result two times: */
/* Once with the first house not included and once with the last house not included */
void prepareHouseArrays(int lightBulbs[], int woFirstHouse[], int woLastHouse[], int n) {
    copyArray(lightBulbs, woFirstHouse, n);
    copyArray(lightBulbs, woLastHouse, n);

    woFirstHouse[0] = 0;
    woLastHouse[n - 1] = 0;
}

int max(int x, int y) {
    return x > y ? x : y;
}
```

- `int maxIlluminationRecursive(int lightBulbs[ ], int n)` which solves the problem recursively.

```
int maxIlluminationRecursiveCalculation(int lightBulbs[], int n) {
    if(n == 0) {
        return 0;
    }

    int i;
    int currentBulbs = 0;

    for(i = 0; i < n; i++) {
        if(i == 0) {
            currentBulbs = lightBulbs[0];
        } else if(i == 1) {
            currentBulbs = max(lightBulbs[0], lightBulbs[1]);
        } else { /* i>1 => more than 3 houses => Neigbors present. */
            currentBulbs = max(currentBulbs, lightBulbs[i] +
                               maxIlluminationRecursiveCalculation(lightBulbs, (i - 2) + 1));
        }
    }

    return currentBulbs;
}

int maxIlluminationRecursive(int lightBulbs[], int n) {
    int bulbsWithoutFirstHouse[n];
    int bulbsWithoutLastHouse[n];

    prepareHouseArrays(lightBulbs, bulbsWithoutFirstHouse, bulbsWithoutLastHouse, n);

    int resultWOFirst = maxIlluminationRecursiveCalculation(bulbsWithoutFirstHouse, n);
    int resultWOLast = maxIlluminationRecursiveCalculation(bulbsWithoutLastHouse, n);

    return max(resultWOFirst, resultWOLast);
}
```

- `int maxIlluminationMemoized(int lightBulbs[ ], int n, int m[])` which solves the problem using memoization. The array m[n] is used in order to store the intermediate results.

```
void prepareMemoizationArray(int lightBulbs[], int m[], int n) {
    if(n >=1) {
        m[0] = lightBulbs[0];
    }
    if(n >=2) {
        m[1] = max(lightBulbs[0], lightBulbs[1]);
    }

    int i;
    for(i = 2; i < n; i++) {
        m[i] = -1;
    }
}
```

```
int maxIlluminationMemoizedCalculation(int lightBulbs[], int n, int m[ ]) {

    if(n == 0) {
        return 0;
    }

    if(m[n - 1] >= 0) {
        return m[n - 1];
    }

    int currentBulbs = 0;
    if(n > 2) { /* Always the case because we prepared m */
        int i;

        for(i = 2; i < n; i++) {
            currentBulbs = max(currentBulbs, lightBulbs[i] +
                                    maxIlluminationMemoizedCalculation(lightBulbs, (i - 2) + 1, m));
        }
    }
    m[n - 1] = currentBulbs;
    return currentBulbs;
}

int maxIlluminationMemoized(int lightBulbs[], int n, int m[]) {
    int bulbsWithoutFirstHouse[n];
    int bulbsWithoutLastHouse[n];

    prepareHouseArrays(lightBulbs, bulbsWithoutFirstHouse, bulbsWithoutLastHouse, n);

    prepareMemoizationArray(bulbsWithoutFirstHouse, m, n);
    int resultWOFirst = maxIlluminationMemoizedCalculation(bulbsWithoutFirstHouse, n, m);

    prepareMemoizationArray(bulbsWithoutLastHouse, m, n);
    int resultWOLast = maxIlluminationMemoizedCalculation(bulbsWithoutLastHouse, n, m);

    return max(resultWOFirst, resultWOLast);
}
```

- `int maxIlluminationDynamic(int lightBulbs[ ], int n)` which solves the problem with the help of dynamic programming.

```
int maxIlluminationDynamicCalculation(int lightBulbs[], int n) {
    if(n == 0) {
        return 0;
    }

    int m[n];
    prepareMemoizationArray(lightBulbs, m, n);
    int i;
    for(i = 2; i < n; i++) {
        int currentBulbs = -1;
        int j;
        for(j = 2; j <= i; j++) {
            currentBulbs = max(currentBulbs, lightBulbs[j] + m[(j - 2)]);
```

```c
        }
        m[i] = currentBulbs;
    }

    return m[n - 1];
}

int maxIlluminationDynamic(int lightBulbs[], int n) {
    int bulbsWithoutFirstHouse[n];
    int bulbsWithoutLastHouse[n];

    prepareHouseArrays(lightBulbs, bulbsWithoutFirstHouse, bulbsWithoutLastHouse, n);

    int resultWOFirst = maxIlluminationDynamicCalculation(bulbsWithoutFirstHouse, n);
    int resultWOLast = maxIlluminationDynamicCalculation(bulbsWithoutLastHouse, n);

    return max(resultWOFirst, resultWOLast);
}
```

Print the results of your functions for each of the following light bulbs-arrays: `[ 11, 4, 3, 6, 8, 9 ]`, `[ 4, 4, 4, 4, 4 ]` and `[ 13, 15, 31, 21, 9, 12, 44, 32, 12, 43, 22, 9, 11, 32, 26, 22, 21, 3, 4, 29]`.

```c
    int testArrayA[] = {11, 4, 3, 6, 8, 9};
    int testArrayB[] = {4, 4, 4, 4, 4};
    int testArrayC[] = {13, 15, 31, 21, 9, 12, 44, 32, 12, 43, 22, 9, 11, 32, 26, 22, 21, 3, 4, 29};
    int mA[A_SIZE];
    int mB[B_SIZE];
    int mC[C_SIZE];

    printf("maxIlluminationRecursive:\n");
    printf("Array␣A:␣%d\n", maxIlluminationRecursive(testArrayA, A_SIZE));
    printf("Array␣B:␣%d\n", maxIlluminationRecursive(testArrayB, B_SIZE));
    printf("Array␣C:␣%d\n", maxIlluminationRecursive(testArrayC, C_SIZE));
    printf("\n");
    printf("maxIlluminationMemoized:\n");
    printf("Array␣A:␣%d\n", maxIlluminationMemoized(testArrayA, A_SIZE, mA));
    printf("Array␣B:␣%d\n", maxIlluminationMemoized(testArrayB, B_SIZE, mB));
    printf("Array␣C:␣%d\n", maxIlluminationMemoized(testArrayC, C_SIZE, mC));
    printf("\n");
    printf("maxIlluminationDynamic:\n");
    printf("Array␣A:␣%d\n", maxIlluminationDynamic(testArrayA, A_SIZE));
    printf("Array␣B:␣%d\n", maxIlluminationDynamic(testArrayB, B_SIZE));
    printf("Array␣C:␣%d\n", maxIlluminationDynamic(testArrayC, C_SIZE));
```

Task 2. Write in C a function `int lenOfLongestGP(int set[], int n)` that uses dynamic programming to calculate and return **Length of the Longest Geometrix Progression (LLGP)** in a given set of numbers. The common ratio of the Geometric Progression must be an integer. For example in series `[5, 7, 10, 15, 20, 29]`, the LLGP is 3 corresponding to subseries `[5, 10, 20]` with common ratio of 2.

```
void swap(int *xp, int *yp) {
    int temp = *xp;
    *xp = *yp;
    *yp = temp;
}

void sort(int arr[], int n) {
   int i, j;
   for (i = 0; i < n-1; i++)
       for (j = 0; j < n-i-1; j++)
           if (arr[j] > arr[j+1])
              swap(&arr[j], &arr[j+1]);
}

int lenOfLongestGP(int set[], int n) {
    if(n < 2) { return n;}
    if (n == 2) {
        return (set[0] % set[1] == 0 || set[1] % set[0] == 0) + 1;
    }
    sort(set, n);
    // An entry L[i][j] in this table stores LLGP with
    // set[i] and set[j] as first two elements of GP
    // and j > i.
    int L[n][n];
    int llgp = 1;
    int i,j;
    for (i = 0; i < n-1; ++i) {
        if (set[n-1] % set[i] == 0) {
            L[i][n-1] = 2;
            if(L[i][n-1] > llgp) {
                llgp = L[i][n-1];
            }
        } else {
            L[i][n-1] = 1;
        }
    }
    // Consider every element as second element of GP
    for (j = n - 2; j >=1; --j) {
        // i, j, k that form a GP
        int i = j-1, k=j+1;

        while(i>= 0 && k <=n-1) {
            // when do these form a GP? set[i] * set[k] == set[j]^2, k/j = j/i

            if(set[i] * set[k] < set[j]*set[j]) {
                ++k;
            } else if (set[i] * set[k] > set[j]*set[j]) {
```

```
                if (set[j] % set [i] == 0) {
                    L[i][j] = 2;
                    if (L[i][j] > llgp) {
                        llgp = L[i][j];
                    }
                } else {
                    L[i][j] = 1;
                }
                --i;
            }
            //i, j, k is a GP
            else {
                L[i][j] = L[j][k] + 1;
                // Update
                if(L[i][j] > llgp) {
                    llgp = L[i][j];
                }

                --i;
                ++k;
            }
        }

        while (i ≥0) {
            if(set[j] % set[i] == 0) {
                L[i][j] = 2;
                if(L[i][j] > llgp) {
                    llgp = L[i][j];
                }
            } else {
                L[i][j] = 1;
            }
            --i;
        }
    }
    return llgp;
}
```

Task 3. Assume a rectangle of size $n \times m$. Determine the minimum number of squares that is prepared to tile the rectangle. The side length of rectangles must be an integer. Figure show as rectangle of 6×5 that requires tiles to tile it with the minimum number of tiles.

The idea for dynamic programming solution is to recursively split the rectangle of size $(n \times m)$ into either two horizontal rectangles $((n \times h)$ and $(n \times (m-h))$ or two vertical rectangles $((v \times m)$ and $((n-v) \times m)$ where $h = 1, 2, \cdots, m-1$ and $v = 1, 2, \cdots, n-1$. Then finding the minimum squares that fill these splits.

- Assume `MinSquare(n, m)` denotes the minimum squares that is required to tile a rectangle of size $n \times m$. State a recursive definition of `MinSquare(n, m)`.
- To efficiently compute the minimum squares that is required to tile a rectangle of $n \times m$, a dynamic programming solution with 2D array `dp`$[0 \cdots n]$`[`$0 \cdots m$`]`

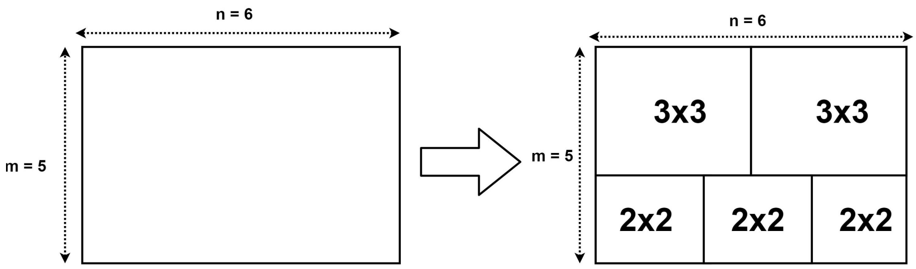

Figure 1: Minimum number of square tiles for a rectangle of size 5×6

can be used. Element `dp[i][j]` is the minimum number of squares that is required to tile a rectangle of size $i \times j$. Complete the 2D array below with the minimum squares required to tile rectangles up to size 4×3.

m \ n	0	1	2	3	4	
	0					
	1	1	2	3	4	
	2	2	1	3	2	
	3	3	3	1	4	

- For a rectangle of size $n \times m$, write a C program that uses a dynamic programming solution to compute the minimum number of squares that is required to tile the rectangle.

```
#define INT_MAX 300
int dp[INT_MAX][INT_MAX];

int minimumSquare(int m, int n) {
    int vertical_min = INT_MAX;
    int horizontal_min = INT_MAX;

    if (m == n)
        return 1;


    if (dp[m][n])
        return dp[m][n];


    for (int i = 1;i≤ m/2;i++) {
        horizontal_min = min(minimumSquare(i, n) +
```

```
                minimumSquare(m−i, n), horizontal_min);
    }

    for (int j = 1;j≤ n/2;j++) {
        vertical_min = min(minimumSquare(m, j) +
                minimumSquare(m, n−j), vertical_min);
    }

    dp[m][n] = min(vertical_min, horizontal_min);

    return dp[m][n];
}
```