Uppsala Universitet

Algorithms and Data Structures II (1DL231), 2017

Assignment 1

Författare:
Team 13
Daniel Ågstrand
Linnea Andersson

UPPSALA UNIVERSITET

17. november 2017

Problem 1: The Birthday Present Problem

This problem is divided into two problems and solved with two different functions. The first function takes in in a list of products and integer that represents your budget. Then the function checks if you can buyproducts in different combinations, without receiving any change. If this is possible then the function returns true otherwise it returns false. The other functions also checks if you can buyproducts, without receiving any change. But instead of returning true or false, it returns a list with all products used to use up your budget. This is achieved by implementing dynamic programing.

a.) Recursive equation

The recursive equation becomes:

$$BP(P,n,t) = \begin{cases} True & t = 0 \\ False & t = 0, n \leq 0 \\ BP(P,n-1,t) & P[n-1] > t \\ BP(P,n-1,t) \quad \bigcup \quad BP(P,n-1,t-P[n-1]) & otherwise \end{cases}$$

As we can see in the recurrence relation, if we enter the last instance of the relation we get two recursion calls. In the subtrees of these calls, we can get the same recursion call in both trees if for instance $t - P[n-1] = t - P[n-2] - P[n-3]$. Thus we get overlapping subproblems and dynamic programming is applicable to make the algorithm more efficient.

b.) Recursive function with dynamic programming

The code is included in Listing 1 and 2. The $birthday\_present(P, n, t)$ function checks if $t \leq 0$ and in that case returns false. Otherwise it initiates the dynamic programming matrix A to save boolean values for all instances of $birthday\_present(P, n, t)$. It then calls function $birthday\_present\_aux(P, n, t)$ which is the recursive function. Before we enter into a recursion, we check if we have already made this recursion call before. If we have, we do not need to enter a recursion but simply collect the boolean value from A.

```
def birthday_present(P, n, t):
    '''
    Sig:  int[0..n-1], int, int --> Boolean
    Pre:  Elements in P must be > 0
    Post: Returns true if there exist elements in P
          whose sum is t, otherwise false
    Example: P = [2, 32, 234, 35, 12332, 1, 7, 56]
             birthday_present(P, len(P), 299) = True
             birthday_present(P, len(P), 11) = False
    '''
    # Initialize the dynamic programming matrix, A
    # Type: Boolean[0..n][0..t]
    global A
    if t <= 0:
        return False
    A = [[None for i in range(t + 1)] for j in range(n + 1)]
    return birthday_present_aux(P, n, t)
```

Listing 1: Code for Birthday Present

```
def birthday_present_aux(P, n, t):
    '''
    Sig:  int[0..n-1], int, int --> Boolean
    Pre:  Elements in P must be > 0
    Post: Returns true if there exist elements in P
          whose sum is t, otherwise false
    Var:  n-1, t or t-P[n-1]
    Example: P = [2, 32, 234, 35, 12332, 1, 7, 56]
             birthday_present(P, len(P), 299) = True
             birthday_present(P, len(P), 11) = False
    '''
    if n == 0 and t == 0:
        A[n-1][t] = True
        return A[n-1][t]
    if n == 0:
        return False
    elif t == 0:
        A[n-1][t] = True
        return A[n-1][t]
    elif P[n-1] > t:
      if A[n-1][t] is None:
        A[n-1][t] = birthday_present_aux(P, n-1, t)
      return A[n-1][t]
    else:
      if A[n-1][t] is None:
        A[n-1][t] = birthday_present_aux(P, n-1, t)
      if A[n-1][t-P[n-1]] is None:
        A[n-1][t-P[n-1]] = birthday_present_aux(P,n-1,t-P[n-1])
    return A[n-1][t] or A[n-1][t-P[n-1]]
```
Listing 2: Code for Birthday Present

c.) Function to return the subset

The code is included in Listing 3. In $birthday\_present\_subset(P, n, t)$ we first initialize the dynamic programming matrix A and the return list R, which is used to save the elements we would like to return as we find them.

Then we iterate $A[i, j]$ for all values of i in $(0, n)$ and j in $(0, t)$ using double for loops in a bottom-up manner. When the matrix A is filled with booleans, we use it to search for the elements in P we would like to include in the return subset. Starting at index t we scan through the column for increasing n until we reach True. If $A[i, t]$ has a True value, it means we should include this $P[i]$ in the return subset. Next we scan the column $t - P[i]$ for increasing n, and when $A[j, t - P[i]]$ is True we include $P[j]$ in the return subset. We keep doing this until $t = 0$, at which point we have found a complete subset whose sum is t.

```
def birthday_present_subset(P, n, t):
    '''
    Sig:  int[0..n-1], int, int --> int[0..m]
    Pre:  Elements in P must be > 0
    Post: Returns a list of elements whose sum is t
    Example: P = [2, 32, 234, 35, 12332, 1, 7, 56]
             birthday_present_subset(P, len(P), 299) = [56, 7, 234, 2]
             birthday_present_subset(P, len(P), 11) = []
    '''
    # Initialize return list, R
    # Type: int[0..n]
    R = []
    # Initialize the dynamic programming matrix, A
    # Type: Boolean[0..n][0..t]
    A = [[None for i in range(t + 1)] for j in range(n + 1)]
    if t <= 0:
        return R

    for i in range(t+1):
        #Variant: (t+1) - i
        for j in range(n+1):
            #Variant: (n+1) - j
            if i == 0 and j == 0:
                A[j][i] = True
            elif i == 0:
                A[j][i] = True
            elif j == 0:
                A[j][i] = False
            elif P[j-1] > i:
                A[j][i] = A[j-1][i]
            else:
                A[j][i] = A[j-1][i] or A[j-1][i-P[j-1]]
    if A[n][t] == False:
        return R
    while True:
        #Variant: none
        i = 0
        while A[i][t - sum(R)] == False:
            #Variant: A[i][t - sum(R)] - i
            i = i + 1
        R.append(P[i-1])
        if t == sum(R):
            return R
```
Listing 3: Code for Birthday Present Subset

d.) Complexity analysis of the subset function

Our algorithm will scan though all possible values of n between and t in every case. The complexity of each instance of the inner loop is $O(1)$ since we are only assigning a value to $A[i, j]$ in every instance. The total complexity of the inner loop is therefore $O(n + 1) = O(n)$. The outer loop then has complexity $O(n)$ for every instance. The total complexity of the two loops is therefore $O(n) \cdot O(t + 1) = O(n \cdot t)$. The final while loop will scan a maximum of n times if the first element of P is to be included in P', so its complexity is $O(n)$.

The total complexity of the algorithm is therefore $O(n \cdot t) + O(n) = O(n \cdot t)$.

Problem 2: Integer Sort

The function Integer Sort takes in a list with unsorted integers, A, and an other integer that should be equal to the inputs lists biggest element, k. The first thing the function should do is to create a new list, Y, that have the length of k + 1, with just zeros in it. The Y-list then adds the number of times a integer n shows up in list-A, in position Y[n]. After that the function goes in to a loop, length of Y times and checks if an element is bigger than zero and if it is equal t times the loop has run. In that case we put an integer with the value of the index of the Y-list in the A-list, t times where x is the value of the element in Y[index]. Then the function returns the sorted A-list.

a.) Implementation

```
def integer_sort(A, k):
    '''
    Sig:  int array[1..n], int -> int array[1..n]
    Pre:  k has to be bigger or equal to zero, max(A) <= k
    Post: A is an increasingly sorted list
    Example: integer_sort([5, 3, 6, 7, 12, 3, 6, 1, 4, 7]), 12) =
                [1, 3, 3, 4, 5, 6, 6, 7, 7, 12]
    '''
    # Initialize the help list, Y
    # Type: int[0] * (k + 1)
    Y = ([0] * (k+1))

    for i in range(len(A)):
        #Variant: len(A) - i
        for x in range(len(Y)):
            #Variant: len(Ý) - x
            if A[i] == x:
                Y[x] = Y[x] + 1

    i = 0
    for x in range(len(Y)):
        #Variant: len(Y) - x
        for t in range(len(A)):
            #Variant: len(A) - t
            if Y[x] == t and t>0:
                for j in range(t):
                    #Variant: t - j
                    A[i] = x
                    i = i + 1
        return A
```

Listing 4: Code for Integer Sort

b.) Complexity analysis

We have a double for loop and a triple for loop. As for the double for loop (starting at row 13 in Listing 4) the inner loop has complexity $O(1)$ for every instance and will run at most $(k+1)$ times, so the complexity of the inner loop is $O(k+1) = O(k)$. The outer loop then has complexity $O(k) \cdot O(|A|+1) = O(k \cdot |A|)$.

As for the triple for loop (starting at row 21 in Listing 4) the innermost loop will run $|A|$ times every time since we have to fill A with all its elements. When A is filled we never enter the third for loop again. So we can just add the complexity of the third for loop to the complexity of the double for loop, which is $O(k \cdot |A|)$ by the same reasoning as the first double for loop. So the total complexity for the triple for loop is $O(k \cdot |A|) + O(|A|) = O(k \cdot |A|)$.

Hence the total complexity of the algorithm is $O(k \cdot |A|) + O(k \cdot |A|) = O(k \cdot |A|)$.

c.) Complexity analysis when $\mathbf{k = O[|A|]}$

If $k = O(|A|)$ we have $O(k \cdot |A|) = O(|A| \cdot |A|) = O(|A|^2)$.

Problem 3: Binary Multiplication

The function for this problem should take in two lists and then do a binary multiplication operation of the two lists and return a list with the length of the input list A and B's length, that have the result from the multiplication in it.

We solved it by making a function that corrects the length of the lists, so that do lists length always is divisible by 2^n, $n > 1$. Then this two new list, A and B, are used as input for a recursive function. The recursive function divide the input lists into two new lists, one upper and one lower lists. Four recursions are executed, with different combinations of low and high, and A and B as input. After the recursions, the result of the recursion of high A and high B are shifted n times to the left, where n is length of one if the input lists. The result of low A and low B and the result of low A and high B are added in a binary adder. The result from this addition is then shifted $n/2$ to the left. This two result are then bit-added with the result of the low A, low B recursion and this result is return.

The last thing that happens is that the length of the result is compared with the length of the original input list A plus length of the original input list B. If the $difference < 0$ then the result list will perpend 0, $|difference|$ times. Otherwise the result list will be popped, $difference$ times. Then the list will be returned.

a.) Implementation

```python
def shift_left(l, n):
    """
    Sig:    int[0..len-1], int n ==> int[0..((len - 1) + n[0])]
    Pre:    None
    Post:   l is the same as input but with n zeros at the end
    Example:    shift_left([0,1,1], 1) = [0,1,1,0]
    """
    for i in range(n):
        #Variant: n - i
        l.append(0)
    return l
```

Listing 5: Code for Shift Left function

```python
def full_adder(A, B):
    """
    Sig:    int[0..m], int[0..k] ==> int[0..n]
    Pre:    A and B cannot be zero
    Post:   Binary addition of A and B
    Example:    full_adder([1,1,1], [1,0,0]) = [1,0,1,1]
    """
    carry = 0
    alen = len(A)
    blen = len(B)
    if alen != blen:
        n = abs(alen - blen)
        if n > 0:
            if alen  > blen:
                for i in range(n):
                    #Variant: n - i
                    B.insert(0, 0)
            else:
                for i in range(n):
                    #Variant: n - i
                    A.insert(0, 0)

    newlen = len(A)
    # Initialize the return list, C
    # Type: int[0..n]
    C = [0] * newlen
    for i in range(newlen):
        #Variant: newlen - i
        if i == (newlen - 1) and (A[-i - 1] == 1 and
                            B[-i - 1] == 1 and carry == 1):
            C[-i - 1] = 1
            C.insert(0, 1)
        elif i == (newlen - 1) and (A[-i - 1] != B[-i - 1]
                    and carry == 1):
            C[-i - 1] = 0
            C.insert(0, 1)
        elif i == (newlen - 1) and (A[-i - 1] == 1 and B[-i - 1] == 1
                    and carry == 0):
            C[-i - 1] = 0
            C.insert(0, 1)
        else:
            if A[-i - 1] == B[-i - 1] and carry == 0:
                if A[-i - 1] == 0:
                    C[-i - 1] = 0
                else:
                    C[-i - 1] = 0
                    carry = 1
            elif A[-i - 1] == B[-i - 1]:
                if A[-i - 1] == 1:
                    C[-i - 1] = 1
                    carry = 1
                else:
                    C[-i - 1] = 1
                    carry = 0
            else:
                if carry == 1:
                    C[-i - 1] = 0
                else:
                    C[-i - 1] = 1
                    carry = 0
    return C
```
Listing 6: Code for Binary Adder

```
def binary_mult(A, B):
    """
    Sig:      int[0..n-1], int[0..n-1] ==> int[0..2*n-1]
    Pre:      A and B cannot be empty
    Post:     Binary multiplication of A and B
    Example:      binary_mult([0,1,1],[1,0,0]) = [0,0,1,1,0,0]
    """
    if len(A) == 1 and len(B) == 1:
        return [A[0] * B[0]]

    finallen = len(A) + len(B)
    if len(A) != len(B):
        t = abs(len(A) - len(B))
        if len(A) > len(B):
            for i in range(t):
                #Variant: t - i
                B.insert(0, 0)
        else:
            for i in range(t):
                #Variant: t - i
                A.insert(0, 0)

    add = True
    power = 1
    newlen = len(A)
    while add:
        #Variant: add, power
        if newlen == (2 ** power):
            add = False
        if newlen < 2 ** power:
            n = 2 ** power - newlen
            for i in range(n):
                #Variant: n - i
                A.insert(0, 0)
                B.insert(0, 0)
            add = False
        else:
            power += 1

    nlen = len(A)
    # Initialize the return list, temp, from the recursive function binary mult aux(A, B, n)
    # Type: int[0..n]
    print len(A)
    temp = binary_mult_aux(A, B, nlen)
    n = len(temp) - finallen
    if n < 0:
        for i in range(abs(n)):
            #Variant: |n| - i
            temp.insert(0, 0)
    else:
        for i in range(n):
            #Variant: n - i
            temp.remove(0)

    return temp
```

Listing 7: Code for Binary Mult

```
def binary_mult_aux(A, B, n):
    """
    Sig:      int[0..n-1], int[0..n-1] ==> int[0..m]
    Pre:      A and B cannot be empty, A and B have equal length and
                    ((len(A) & len(B)) % 2^k) == 0
    Post:     Binary multiplication of A and B
    Var:      n / 2, A / 2, B / 2
    Example:      binary_mult([0,1,1],[1,0,0]) = [0,0,1,1,0,0]
    """
    if n == 1:
        return [A[0] * B[0]]
    else:
        m = n - n / 2

        ah = A[:n / 2]
        al = A[n / 2:n]
        bh = B[:n / 2]
        bl = B[n / 2:n]

        x1 = binary_mult_aux(ah, bh, m)
        x2 = binary_mult_aux(al, bl, m)
        x3 = binary_mult_aux(al, bh, m)
        x4 = binary_mult_aux(ah, bl, m)

        t1 = shift_left(x1, n)
        t2 = full_adder(x3, x4)
        t3 = shift_left(t2, m)
        t4 = full_adder(t1, t3)

        final = full_adder(t4, x2)

    return final
```

Listing 8: Code for Binary Mult Aux

b.) Recursion tree method

In the worst case we have arrays whose length is not 2^k where $k \in N$. In this case we have to add zeros until the length is 2^k. This will make the arrays longer but if n is the starting length, the final length will be $< 2n$. Thus $|A| = O(n)$. We also have the following complexities:

Finding the next 2^k larger than $|A|$: $O(lg(n))$
Adding zeros in the beginning: $O(n)$
Removing unwanted zeros at the end: $O(n)$

As we will see, the complexity of these operations will be negligible compared to the complexity of the recursion part of the algorithm. The recursion tree will have height $lg_2(n)$ and length $4^{lg_2(n)} = n^2$. At each recursion level we do two left shifts and four additions. The complexities of these are as follows:

Left shifts: $O(n) + O(\frac{n}{2})$
Additions: $O(n) + O(n) + O(n)$

So the work to combine the result at each level is $O(n)$ (for the n at that level). At level k the size of the array is $\frac{n}{2^k}$ and the number of recursion calls are 4^k. So the total work required at level k is $\frac{n}{2^k} \cdot 4^k = n \cdot 2^k$. The total amount of work is:

$\sum_{k=0}^{lg_2 n} n \cdot 2^k = n(\frac{1-2^{lg_2 n-1}}{1-2}) = 2n^2 + 1 = O(n^2)$

So the recursion tree method gives the final complexity of $O(n^2)$.

c.) The substitution method

Our recurrence relation is $T(n) = 4T(\frac{n}{2}) + n$ and our guess at the complexity of $T(n)$ is $O(n^2)$. We have:

$T(n) = 4T(\frac{n}{2}) + n \leq k(\frac{n}{2})^2 + n = k\frac{n^2}{4} + n \leq kn^2$

So we have that $T(n) \leq kn^2$ and thus $T(n) = O(n^2)$

d.) The master theorem

The number of subproblems is constant, the number of subproblems are more than 1, and $f(n) = n$ is a positive polynomial, so we can apply the master theorem[1].

$lg_2 4 = 2$ and $k = 1 < 2$ means that we should apply Case 1 of the master theorem so
$T(n) = \Theta(n^{lg_2 4}) = \Theta(n^2)$

1 Referenser

[1] Brilliant.org. (17/11-2009). 'Master Theorem' [Website]. Downloaded (17/11-2017) from `https://brilliant.org/wiki/master-theorem/`