Assignment 2 of Algorithm

Zuyao Chen 201728008629002

1. Largest Divisible Subset

a).
- **Def.** $OPT(i)$ = length of the optimal subset of distinct integers $a_1, a_2, ..., a_i$.
- **Given.** a set of distinct positive integers $A_n = \{a_1, a_2, ..., a_n\}$.
- **Goal.** $OPT(n)$ and extract optimal subset S.

To compute $OPT(i)$,

- if $i = 1$, $A_i = \{a_1\}$,$OPT(1) = 1$;
- for $i > 1$, firstly we sort the original array in ascending order, then scan over the array from $j = 1$ to $j = i - 1$,hence

$$OPT(i) = \max \begin{cases} OPT(j) + 1, & a_i \% a_j = 0 \text{ and } \forall m, j < m < i, a_i \% a_m \neq 0 \\ OPT(i-1) \end{cases}$$

b). **pseudo-code:**

Algorithm 1 find the largest divisible subset of distinct positive integers

Input: A set of distinct positive integers $a_1, a_2, ..., a_n$
Output: the largest divisible subset S

function LARGEST-DIVISIBLE($a_1, a_2, ..., a_n$)
    if $n = 1$ **then return** a_1
    end if
    sort original array in ascending order
    initialize elements of $len[n], pre[n]$ with 1 ,$S = \{\}$
    for $i = 2$ to n **do**
        for $j = 1$ to $i - 1$ **do**
            if $a[i] \% a[j] = 0$ and $len[i] < len[j] + 1$ **then**
                $len[i] = len[j] + 1, pre[i] = j$
            end if
        end for
    end for
    $[max\_len, max\_index] = \max(len[])$
    initialize $index[max\_len] = \{0, 0, ..., 0, max\_index\}$
    for $i = max\_len - 1$ to 0 **do**
        $index[i] = pre[max\_index], max\_index = index[i]$
    end for
    $S = S \cup a_{index[k]}, k = 1$ to max_len
    return S
end function

c). **correctness:**

Let S_i be the optimal subset of $A_i = \{a_1, a_2, ..., a_i\}$.

- if $i = 1, A_i = \{a_1\}$, then $OPT(i) = 1$,$S_i = \{a_1\}$
- if $i = k > 1, A_k = \{a_1, a_2, ..., a_k\}$ is arranged in ascending order.
  i. if $a_k \in S_k$,we initialize $S_k = \{a_k\}$, scan over the array from $j = 1$ to $k - 1$,if $a_k\%a_j = 0$, add a_j to S_k, finally we can get the S_k. Find the maximum j subjected to $a_k\%a_j = 0$, then remove all a_m($a_k\%a_m \neq 0(j < m < k)$), S_k should be a subset of $\{a_1, a_2, ..., a_j\} \cup \{a_k\}$, hence len$(S_k) = OPT(j) + 1$;
  ii. if $a_k \notin S_k$, that means it does not matter to remove a_k,thus len$(S_k) =$ len$(S_{k-1}) = OPT(k - 1)$

  thus

$$OPT(i) = \max \begin{cases} OPT(j) + 1, & a_k\%a_j = 0 \text{ and } \forall m, j < m < i, a_i\%a_m \neq 0 \\ OPT(i - 1) \end{cases}$$

In the pseudo-code, we use a small trick that only when $len[j] + 1 > len[i]$, $len[i]$ is updated and the position j is recorded,then find the maximum element of $len[]$.In another words, $len[i]$ is the length of the subset S'_i ended with a_i, $pre[i]$ is the end position of subset $S'_i - \{a_i\}$ ($a_i\%a_{pre[i]} = 0$ and $\forall k, pre[i] < k < i, a_i\%a_k \neq 0$). There must exists a subset ended with max(S_i),if we find i_0 that $len[i_0]$ is the maximum in $len[]$,the subset S'_{i_0} is the optimal subset S_i.

d). **complexity:**

- sorting the array costs $O(nlogn)$ using quick-sort;
- traversal needs $\frac{n(n-1)}{2}$ steps,thus it costs $O(n^2)$;
- tracing back to extract optimal subset costs $O(n)$

thus the time complexity of algorithm is $O(n^2)$.

2. Money robbing

i. case 1

a). **description:**

- **Def.** $OPT(i)$ = sum of optimal subset of a list of non-negative integers.
- **Given.** a list of non-negative integers $A_n = \{a_1, a_2, ..., a_n\}$.
- **Goal.** $OPT(n)$.

To compute $OPT(i)$,

- if $i = 1, OPT(i) = a_1$;
- if $i = 2, OPT(i) = \max(a_1, a_2)$;

- if $i > 2$,

$$OPT(i) = \max \begin{cases} OPT(i-1) \\ OPT(i-2) + a_i \end{cases}$$

b). **pseudo-code:**

Algorithm 2 find the optimal subset with maximum sum of a list of positive integers

Input: A set of positive integers $A_n = \{a_1, a_2, ..., a_n\}$
Output: maximum sum of subset S,any two elements of S are not adjacent in A_n

function MONEY-ROBBING($a_1, a_2, ..., a_n$)
  if $n = 0$(A_n is NULL) **then**
    return 0
  else if $n \leq 2$ **then**
    return max(A_n)
  end if
  initialize $sum[n] = \{a_1, \max(a_1, a_2), 0, 0, ...\}$
  for $i = 3$ to n **do**
    $sum[i] = \max(sum[i-1], sum[i-2] + a_i)$
  end for
  return $sum[n]$
end function

c). **correctness:**

- if $n = 1, A_n = \{a_1\}$,then $OPT(n) = a_1$;
- if $n = 2, A_n = \{a_1, a_2\}$,then $OPT(n) = \max(a_1, a_2)$;
- if $n > 2$, let $k = n - 1$,$A_k = \{a_1, a_2, ..., a_k\}$, $OPT(k) = \text{sum}(S_k)$,S_k be the optimal subset of A_k. $A_{k+1} = \{a_1, a_2, ..., a_k, a_{k+1}\}$,
  i. if $a_{k+1} \in S_{k+1}$,then $a_k \notin S_{k+1}$, $S_{k+1} = S_{k-1} \cup \{a_{k+1}\}$, thus $OPT(k+1) = OPT(k-1) + a_{k+1}$;
  ii. if $a_{k+1} \notin S_{k+1}$, $S_{k+1} = S_k$,hence $OPT(k+1) = OPT(k)$.

  in summary, if $n > 2$

$$OPT(n) = \max \begin{cases} OPT(n-1) \\ OPT(n-2) + a_n \end{cases}$$

$OPT(n)$ can be solved recursively from $OPT(1)$,the algorithm is correct .

d). **complexity:**
Obviously it costs $O(n)$ time.

ii. case 2 – all houses are arranged in a circle

Under this circumstance, let S'_k be the optimal subset of $A'_k = \{a_1, a_2, ..., a_k\}$ (a_1, a_k are adjacent), $OPT'(k)$ be the sum of S'_k,$OPT(k)$ be the sum in the condition above.

- if $a_1 \in S'_k$,then $a_k \notin S'_k$, $OPT'(k) = OPT(k-1)$;
- if $a_1 \notin S'_k$, then a_k may belong to S'_k,just let $a_1 = 0$, thus $OPT'(k) = OPT(k)$

hence,

$$OPT'(n) = \begin{cases} \max(A'_k), & \text{if } n <= 3 \\ \max \begin{cases} OPT(n-1) \\ OPT(n), \text{let } a_1 = 0 \end{cases} & otherwise \end{cases}$$

3. Palindrome Partition

- **Def.** $OPT(s, i, j)$ = the minimum cuts need for a palindrome partitioning of string $s[i \cdots j]$
- **Given.** a string $S_n = \{s_1 s_2 \cdots s_n\}$
- **Goal.** $OPT(S_n, 1, n)$

there are several conditions ,

- if $i = j$ then $OPT(s, i, j) = 0$
- if $s[i \cdots j]$ is a palindrome, then $OPT(s, i, j) = 0$
- if none of the above conditions is true, then

$$OPT(s, i, j) = \min\{OPT(s, i, k) + 1 + OPT(s, k+1, j)\}, k = i, \cdots, j-1$$

We first determine whether $s_L[i \cdots j]$ is a palindrome, here s_L stands for the substrings s_L (L from 2 to n) .

Algorithm 3 find the minimum cuts need for palindrome partitioning

Input: A string $S = \{s_1 s_2 \cdots s_n\}$
Output: the minimum cuts need for S

function MINI-PARTITION($S = \{s_1 s_2 \cdots s_n\}$)
    set each elements of $C[n]$ with 0, each elements of $P[n][n]$ with $True$
    for L from 2 to n **do**
        for i from 1 to $n - L + 1$ **do**
            $j = i + L - 1$
            if $s[i] = s[j]$ and $L = 2$ **then**
                $P[i][j] = True$
            else if $s[i] = s[j]$ and $L > 2$ **then**
                $P[i][j] = P[i+1]P[j-1]$
            else
                $P[i][j] = False$
            end if
        end for
    end for
    for k from 1 to n **do**
        if $P[1][k] = True$ **then**
            $C[k] = 0$
        else
            $C[k] = inf$
            for l from 1 to $k - 1$ **do**
                if $P[l+1][k] = True$ and $C[k] > 1 + C[l]$ **then**
                    $C[k] = 1 + C[l]$
                end if
            end for
        end if
    end for
    return $C[n]$
end function

time complexity: $O(n^2)$

4. Decoding

a). **description:**

- **Def.** $OPT(i)$ = the number of ways decoding a message $S_i = \{s_1 s_2 ... s_i\}$.
- **Given.** an encoded message $S_n = \{s_1 s_2 s_3 ... s_n\}$.
- **Goal.** $OPT(n)$.

the all cases are showed below,

i. If S_i is NULL or started with '0'($s_1 =$ '0'), it makes no sense,thus $OPT(i) = 0$;

ii. If $i = 1$ and $s_1 \neq$ '0', $OPT(i) = 1$;

iii. If $i > 1$ and $s_1 \neq$ '0', define $OPT(0) = 1$. Consider $S_{k-1} = \{s_1 s_2 ... s_{k-1}\}$ and $S_k = \{s_1 s_2 ... s_{k-1} s_k\}$, $k > 1$. suppose that S_{k-1} is valid (no isolated '0' in the decoded message),

- if $s_{k-1} =$ '0' and $s_k =$'0' ,S_k is invalid,$OPT(k) = 0$.
- if $s_{k-1} =$ '0' and $s_k \neq$'0' , s_k can not be combined with s_{k-1}, thus $OPT(k) = OPT(k-1)$.
- if '$s_{k-1}s_k$'> 26, s_k should be isolated, $OPT(k) = OPT(k-1)$.
- if '$s_{k-1}s_k \leq 26$, if $s_k =$'0',that means $s_{k-1}s_k$ should be regarded as a number,hence $OPT(k) = OPT(k-2)$;if $s_k \neq$'0', the result can be regarded as a combination of two part: one part is just like decoding S_{k-1} (s_k is isolated),another part is regarding $s_{k-1}s_k$ as a number,then there has no $s_{k-2}s_{k-1}$ and s_{k-1} is isolated in decoding S_{k-1}, therefore $OPT(k) = OPT(k-1) + OPT(k-2)$.

In summary,to compute $OPT(n)$ (define $OPT(0) = 1$),

$$OPT(n) = \begin{cases} 0, & \text{if } S_n \text{ is NULL or } s_1 = \text{'0' or '00' in } S_n \\ 1, & \text{else if } n = 1 \\ OPT(n-1), & \text{else if } n \geq 2, \text{'}s_{n-1}s_n\text{'} > 26 \text{ or } s_{n-1} = \text{'0'} \\ OPT(n-2), & \text{else if } n \geq 2, \text{'}s_{n-1}s_n\text{'} \leq 26 \text{ and } s_n = \text{'0'} \\ OPT(n-1) + OPT(n-2), & \text{else if } n \geq 2, \text{'}s_{n-1}s_n\text{'} \leq 26 \text{ and } s_n \neq \text{'0'} \end{cases}$$

b). **pseudo-code:**

Algorithm 4 find the number of ways decoding a message

Input: an encoded message $S_n = \{s_1 s_2 ... s_n\}$
Output: the number of ways decoding the message

```
function NUM-DECODINGS($S_n$)
    if $S_n$ is invalid then
        return 0
    end if
    if $n = 1$ then
        return 1
    end if
    initialize $M[n+1] = \{1, 1, 0, 0...\}$
    for $i = 2$ to $n$ do
        if $s_{i-1}s_i > 26$ or $s_{i-1} =$ '0' then
            if $s_i =$ '0' then
                return 0
            end if
            $M[i+1] = M[i]$
        else
            if $s_i =$'0' then
                $M[i+1] = M[i-1]$
            else
                $M[i+1] = M[i] + M[i-1]$
            end if
        end if
    end for
    return $M[n+1]$
end function
```

c). **correctness:**
the detail description is showed above,$OPT(n)$ can be solved from state $OPT(1)$,and all conditions are considered,thus it's a correct algorithm.

d). **complexity:**
Obviously it costs $O(n)$ time.

5. Longest Consecutive Subsequence

6. Maximum length

Code is written using C++ in accessory file 'max_length.cpp'. Here is a test example ,