

Assignment 2 of Algorithm

Zuyao Chen 201728008629002

zychen.uestc@gmail.com

1. Largest Divisible Subset

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

Output: the largest divisible subset S

```
1: function LARGEST-DIVISIBLE( $a_1, a_2, \dots, a_n$ )
2:   if  $n = 1$  then return  $a_1$ 
3:   end if
4:   sort original array in ascending order
5:   initialize elements of  $len[n], pre[n]$  with 1,  $S = \{\}$ 
6:   for  $i = 2$  to  $n$  do
7:     for  $j = 1$  to  $i - 1$  do
8:       if  $a[i] \% a[j] = 0$  and  $len[i] < len[j] + 1$  then
9:          $len[i] = len[j] + 1, pre[i] = j$ 
10:      end if
11:    end for
12:  end for
13:   $[max\_len, max\_index] = \max(len[])$ 
14:  initialize  $index[max\_len] = \{0, 0, \dots, 0, max\_index\}$ 
15:  for  $i = max\_len - 1$  to 0 do
16:     $index[i] = pre[max\_index], max\_index = index[i]$ 
17:  end for
18:   $S = S \cup a_{index[k]}, k = 1$  to  $max\_len$ 
19:  return  $S$ 
20: end function
```

c). **correctness:**

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

- if $i = 1$, $A_i = \{a_1\}$, then $OPT(i) = 1, S_i = \{a_1\}$
- if $i = k > 1$, $A_k = \{a_1, a_2, \dots, a_k\}$ 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 $\{a_1, a_2, \dots, a_j\} \cup \{a_k\}$, hence $\text{len}(S_k) = OPT(j) + 1$;
 - ii. if $a_k \notin S_k$, that means it does not matter to remove a_k , thus $\text{len}(S_k) = \text{len}(S_{k-1}) = OPT(k - 1)$

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

d). **complexity:**

- sorting the array costs $O(n \log n)$ 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 $A_n = \{a_1, a_2, \dots, a_n\}$.
- **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 $A_n = \{a_1, a_2, \dots, a_n\}$

Output: maximum sum of subset S , any two elements of S are not adjacent in A_n

```

1: function MONEY-ROBBING( $a_1, a_2, \dots, a_n$ )
2:   if  $n = 0$  ( $A_n$  is NULL) then
3:     return 0
4:   else if  $n \leq 2$  then
5:     return  $\max(A_n)$ 
6:   end if
7:   initialize  $sum[n] = \{a_1, \max(a_1, a_2), 0, 0, \dots\}$ 
8:   for  $i = 3$  to  $n$  do
9:      $sum[i] = \max(sum[i-1], sum[i-2] + a_i)$ 
10:  end for
11:  return  $sum[n]$ 
12: 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 $k = n - 1, A_k = \{a_1, a_2, \dots, a_k\}$, $OPT(k) = \text{sum}(S_k)$, S_k be the optimal subset of A_k . $A_{k+1} = \{a_1, a_2, \dots, a_k, a_{k+1}\}$,
 - 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 $OPT(k+1) = OPT(k-1) + a_{k+1}$;
 - 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 $A'_k = \{a_1, a_2, \dots, a_k\}$ (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,

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

3. Decoding

a). **description:**

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

the all cases are showed below,

- If S_i is NULL or started with '0' ($s_1 = '0'$), it makes no sense, thus $OPT(i) = 0$;
- If $i = 1$ and $s_1 \neq '0'$, $OPT(i) = 1$;
- If $i > 1$ and $s_1 \neq '0'$, define $OPT(0) = 1$. Consider $S_{k-1} = \{s_1 s_2 \dots s_{k-1}\}$ and $S_k = \{s_1 s_2 \dots s_{k-1} s_k\}$, $k > 1$. 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 $OPT(0) = 1$),

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

b). **pseudo-code:**

Algorithm 3 find the number of ways decoding a message

Input: an encoded message $S_n = \{s_1 s_2 \dots s_n\}$

Output: the number of ways decoding the message

```
1: function NUM-DECODINGS( $S_n$ )
2:   if  $S_n$  is invalid then
3:     return 0
4:   end if
5:   if  $n = 1$  then
6:     return 1
7:   end if
8:   initialize  $M[n + 1] = \{1, 1, 0, 0 \dots\}$ 
9:   for  $i = 2$  to  $n$  do
10:    if  $s_{i-1}s_i > 26$  or  $s_{i-1} = '0'$  then
11:      if  $s_i = '0'$  then
12:        return 0
13:      end if
14:       $M[i + 1] = M[i]$ 
15:    else
16:      if  $s_i = '0'$  then
17:         $M[i + 1] = M[i - 1]$ 
18:      else
19:         $M[i + 1] = M[i] + M[i - 1]$ 
20:      end if
21:    end if
22:  end for
23:  return  $M[n + 1]$ 
24: 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.

4. Maximum length

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

```
Joseph_chen@JosephChen-Desktop:~/Documents/algorithms_course/c_plus_plus$ ./max
array:3 5 6 2 5 4 19 5 6 7 1
length of optimal subset:5
```