Data Structure and Algorithm

Homework #1 Solution

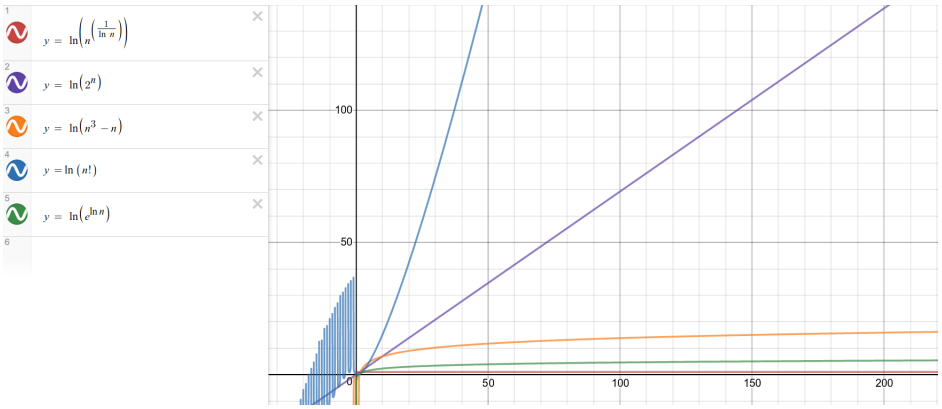
TA email: dsa1@csie.ntu.edu.tw

Problem 1. Asymptotic Notation (25%)

1.1. (9%)

(5%) $n!, 2^n, n^3 - n, e^{\ln n}, n^{\frac{1}{\ln n}}$

(4%) Plot(log scale)



1.2. (6%)

Apply Stirling's approximation (equation 3.18 in the textbook).

$n! = \sqrt{2\pi n}(\frac{n}{e})^n(1 + \Theta(\frac{1}{n}))$

- (3%) Prove $n! = \omega(2^n)$

  $\lim_{n\to\infty} \frac{2^n}{n!} = \lim_{n\to\infty} \frac{1}{\sqrt{2\pi n}(1 + \Theta(\frac{1}{n}))}(\frac{2e}{n})^n \leq \lim_{n\to\infty}(\frac{2e}{n})^n = 0$

  $\implies n! = \omega(2^n)$

- (3%) Prove $n! = o(n^n)$

  $\lim_{n\to\infty} \frac{n^n}{n!} = \lim_{n\to\infty} \frac{1}{\sqrt{2\pi n}(1 + \Theta(\frac{1}{n}))}e^n \geq \lim_{n\to\infty} \frac{e^n}{\sqrt{n}} = \infty$

  $\implies n! = o(n^n)$

1.3. (10%)

(a) (2%)

$f(n) = O(g(n))$

$\leftrightarrow$ We can find a $c > 0$ and an n_0 s.t. $f(n) \leq c \cdot g(n)$ when $n \geq n_0$

$\leftrightarrow$ We can find a $c' > 0$ and an n_1 s.t. $g(n) \geq c' \cdot f(n)$ when $n \geq n_1$ ($c' = \frac{1}{c}$ and $n_1 = n_0$)

$\leftrightarrow g(n) = \Omega(f(n))$

(b) (2%)

$f(n) = \Theta(g(n))$

$\leftrightarrow$ We can find a $c_1 > 0$, a $c_2 > 0$ and an n_0 s.t. $c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$ when $n \geq n_0$

$\leftrightarrow$ We can find a $c_1 > 0$, a $c_2 > 0$ and an n_0 s.t. $f(n) \geq c_1 \cdot g(n)$ when $n \geq n_0$ and $f(n) \leq c_2 \cdot g(n)$ when $n \geq n_0$

$\leftrightarrow f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$

(c) (3%)

$f(n) = O(g(n)) \rightarrow f(n) \cdot g(n) = O((g(n))^2)$

$\because g(n)$ is an asymptotically positive function

$\therefore$We can find an n_0 s.t. $g(n) > 0$ when $n \geq n_0$

$\because f(n) = O(g(n))$

$\therefore$We can find a c and an n_1 s.t. $f(n) \leq c \cdot g(n)$ when $n \geq n_1$

$\rightarrow f(n) \cdot g(n) \leq c \cdot g(n) \cdot g(n)$ when $n \geq n_2$ ($n_2 = \max\{n_0, n_1\}$)

$\rightarrow f(n) \cdot g(n) = O((g(n))^2)$

$f(n) \cdot g(n) = O((g(n))^2) \rightarrow f(n) = O(g(n))$

$\because g(n)$ is an asymptotically positive function

$\therefore$We can find an n_0 s.t. $g(n) > 0$ when $n \geq n_0$

$\because f(n) \cdot g(n) = O((g(n))^2)$

$\therefore$We can find a c and an n_1 s.t. $f(n) \cdot g(n) \leq c \cdot g(n) \cdot g(n)$

$\rightarrow f(n) \leq c \cdot g(n)$ when $n \geq n_2$ ($n_2 = \max\{n_0, n_1\}$)

$\rightarrow f(n) = O(g(n))$

(d) (3%)

$f(n) = O(g(n)) \rightarrow (f(n))^2 = O((g(n))^2)$

$\because f(n)$ is an asymptotically positive function.

$\therefore$We can find an n_0 s.t. $f(n) > 0$ when $n \geq n_0$

$\because g(n)$ is an asymptotically positive function.

$\therefore$We can find an n_1 s.t. $g(n) > 0$ when $n \geq n_1$

$\because f(n) = O(g(n))$

$\therefore$ We can find a c and an n_2 s.t. $f(n) \leq c \cdot g(n)$ when $n \geq n_2$

$\rightarrow f(n) \cdot f(n) \leq c \cdot f(n) \cdot g(n) \leq c \cdot (c \cdot g(n)) \cdot g(n)$ when $n \geq n_3$ ($n_3 = \max\{n_0, n_1, n_2\}$)

$\rightarrow (f(n))^2 = O((g(n))^2)$

$(f(n))^2 = O((g(n))^2) \rightarrow f(n) = O(g(n))$

$\because f(n)$ is an asymptotically positive function.

$\therefore$We can find an n_0 s.t. $f(n) > 0$ when $n \geq n_0$

$\because g(n)$ is an asymptotically positive function.

$\therefore$We can find an n_1 s.t. $g(n) > 0$ when $n \geq n_1$

$\because (f(n))^2 = O((g(n))^2)$

$\therefore$We can find a c and an n_2 s.t. $f(n)^2 \leq c \cdot g(n)^2$ when $n \geq n_2$

$\rightarrow \sqrt{f(n)^2} \leq \sqrt{c \cdot g(n)^2}$ when $n \geq n_3$ ($n_3 = \max\{n_0, n_1, n_2\}$)

$\rightarrow$We can find a c' and an n_3 s.t. $f(n) \leq c' \cdot g(n)$ when $n \geq n_3$ ($c' = \sqrt{c}$)

$\rightarrow f(n) = O(g(n))$

Problem 2. Time and Space Complexities (15%)

2.1. (4%)

(1) Time complexity

Binary_Search : O($N \log N$). Since the outer for loop must run N times and the inner one will run $logN$ times.

Count_Search : O(N). Since the *malloc*() function takes O(1) and the for loop will run N times.

(2) Space complexity of *Count_Search*

O(max$\{K, k\}$), except the space cost of A, the B array takes max$\{K, k\} + 1 =$ O(max$\{K, k\}$).

(3) *Binary_Search* : Higher time-complexity, but less space used.

or

Count_Search : Lower time-complexity, but more space needed.

2.2.

(a) (3%)

```
M = 0;
for (i = 0; i < N-1; i++)
    for (j = i+1; j < N; j++)
        if (A[i] + A[j] == k)
            M++;
```

(b) (3%)

```
sort(A);
M = 0;
for (i = 0; i < N-1; i++) {
    left = i+1 , right = N-1;
    while ( left <= right ) {
        mid = ( left + right ) / 2;
        if ( A [ mid ] == k-A[i] ) {
            M++;
            break;
        }
        else if ( A [ mid ] < k-A[i] ) left = mid + 1;
        else if ( A [ mid ] > k-A[i] ) right = mid - 1;
    }
}
```

2.3. (5%)

```
for (i = 0; i < N-2; i++) {
    left = i+1;
    right = N-1
    while(left < right) {
        if (A[left] + A[right] == k-A[i]) return true;
        else if (A[left] + A[right] > k-A[i]) right--;
        else if (A[left] + A[right] < k-A[i]) left++;
    }
}
```

The worst case happens when the last three elements are the answers. In such case, the for loop runs $(N-3)$ times and each iteration contains a while loop that runs $(N-i)$ times. Therefore, the time-complexity is $O(n^2)$.

Problem 3. Stack and Queue (20%)

3.1. Yes, it is stack-valid. The operations are as follows:

```
PUSH 1
PUSH 2
PUSH 3
POP
POP
PUSH 4
POP
POP
PUSH 5
POP
```

3.2. (2%)

```
Is_Queue_Valid(A) {
    n = A.size
    for i from 0 to n-1
        if A[i] != i
            return false
        end if
    end for

    return true
}
```

Time Complexity: The for loop takes at most n iterations, and other operations take constant time, so the total complexity is $O(n)$.

Actually you may also simulate and maintain a queue as in the next subproblem, but we believe you are smart enough to come up with this solution :).

3.3. (6%)

```
Is_Stack_Valid(A) {
    n = A.size
    S = new Stack
    num = 1
    for i from 0 to n-1
        while num <= A[i]
            S.push(num)
            num++
        end while

        if A[i] == S.top()
            S.pop()
        else
            return false
        end if
    end for

    return true
}
```

Time Complexity: For the for loop, there are at most n iterations. Inside the for loop, since num will increase by 1 for at most n times, the while loop will run at most n times in total throughout the entire execution of this function. Other operations only take constant time, so the total time complexity is $O(n)$.

3.4. (2%)

The answer is the same as problem 3.2. The time complexity is also $O(n)$.

Space Complexity: O(1), since the extra variables are n and i.

3.5. (6%)

```
Is_Stack_Valid(A) {
    n = A.size
    for i from 0 to n-2
        temp = A[i]
        for j from i+1 to n-1
            if A[j] < A[i]
                if A[j] > temp
                    return false
                end if
                temp = A[j]
            end if
        end for
    end for

    return true
```

```
16 }
```

Brief explanation: since the numbers are pushed into the stack in ascending order, for any $A[i]$ popped at some time, we can infer that all the numbers that are still in the stack and smaller than $A[i]$ will be popped out in descending order.

Time Complexity: $O(n^2)$, since the inner for loop takes at most $(n-1)+(n-2)+\cdots+1 = n(n-1)/2$ iterations, and other operations only take constant time.

Space Complexity: O(1), since the extra variables are n, i, j and *temp*.

Problem 4. Calculator (Programming problem) (20%)

There are two steps to solve this problem.

1. Check whether it's valid arithmetic expression

   Since the definition of valid arithmetic expression $\mathbb{E}$ is defined recursively, we can also check the validity recursively.

   Let the given arithmetic expression be S. If S is a valid arithmetic expression, it should be in the form of $\mathbb{N}$ or $\mathbb{E}$+$\mathbb{E}$ or $\mathbb{E}$-$\mathbb{E}$ or $\mathbb{E}$*$\mathbb{E}$ or ($\mathbb{E}$).

   - If all the characters of S is among `0-9` (specially, if first digit is `0`, its length should be exactly 1), we can directly know that S is in $\mathbb{N}$. Therefore, `S` is a valid arithmetic expression.
   - If the first digit is `0-9` and we have already checked it's not in $\mathbb{N}$, we can know that this arithmetic expression starts with $\mathbb{N}$. Thus, we find the next character not in `0-9`, say i-th character, *i.e.*, S_i. If S_i is `()`, then we have an invalid arithmetic expression. Otherwise, S may be in one of $\mathbb{E}$+$\mathbb{E}$ or $\mathbb{E}$-$\mathbb{E}$ or $\mathbb{E}$*$\mathbb{E}$. Since, we already found that S start with $\mathbb{N}$(Remember to check leading zeros) and followed by one of `+-*`. We just need to check whether those characters after S_i form a valid arithmetic expression.
   - If the first digit is `(`, *i.e.*, the opening parenthesis, we can find the corresponding `)`, closing parenthesis. (If not, S must be invalid.) Suppose the corresponding `)` is at the i-th character. We need to check whether the second to $(i-1)$-th characters form a valid arithmetic expression in order to match ($\mathbb{E}$). If there's any character after i-th one, it should be one of `+-*`. Then, we also need to check whether those characters after $i+1$-th one is a valid arithmetic expression.
   - Otherwise, it must be invalid.

The following pseudo-code may clarify the statements above:

```
Is_valid_N(S):
  if len(S) == 0:
    return False
  for c in S:
    if c not in [0-9]:
      return False
  if S[0] == '0' and len(S) > 1:
    return False
  return True
Is_valid_E(S):
  if len(S) == 0:
    return False
  if S[0] == '(':
    idx = pair() # index of ')' paired with this '('
    if idx == -1: # Not found
      return False
    if not Is_valid_E(S[1:idx]):
      return False
    if idx + 1 == len(S):
      return True
    if S[idx+1] not in [+-*]:
      return False
    return Is_valid_E(S[idx+2:])
  idx = next() # index of next character not in [0-9]
  if idx == -1: # Not found
    return Is_valid_N(S)
  if S[idx] in [()]:
    return False
  return Is_valid_E(S[:idx]) and Is_valid_E(S[idx+1:])
```

Actually, we don't need to slice out the string and pass it recursively. We just need to pass the start index and end index of the substring currently concerned. (Each string passed to `Is_valid_E` is always a substring of S)

Moreover, we can preprocess the string to find out the paired parentheses with a stack and also precompute the index of the next character not in [0-9]. With these two precomputations, we can check whether the string is a valid arithmetic expression in linear time (i.e. $O(|S|)$).

The pseudo-code for precomputations is as follows:

```
precomputation(S):
  pair = [-1] * len(S)
  next = [-1] * len(S)
  st = Stack()
  for i in 0..len(S)-1:
    if S[i] == '(':
      st.push(i)
    if s[i] == ')':
      if not st.empty():
        pair[st.pop()] = i
  iter = -1
  for i in len(S)-1..0:
    if S[i] not in [0-9]:
      iter = i
    next[i] = iter
```

Also, this precomputation takes linear time (i.e. $O(|S|)$)

2. Evaluate the value of arithmetic expression

   First of all, the given arithmetic expression is actually in the form of in-order expression. If we turn the in-order expression into post-order expression, it will be easier to evaluate. Thus, our goal is to first transform the given infix expression into post-fix one and then evaluate it afterwards. These two steps can be done, each with a stack, but actually we can do the evaluation while transforming. Since the method is well-known, it's not mentioned here.

   Nevertheless, we can do something much easier. Since we already know how to check the arithmetic expression recursively in the previous problem, we can also evaluate the arithmetic expression recursively, which would decrease the complexity of coding.

The pseudo-code is as follows:

```
def priority(op):
  if op in [+-]:
    return 1
  if op in [*]:
    return 2
  return -1
def evaluate(S):
  num = Stack() # Stack for storing numbers
  ope = Stack() # Stack for storing operators
  iter = 0
  while iter < len(S):
    if S[iter] == '(':
      idx = pair[iter]
      num.push(evaluate(S[1:idx-1]))
      iter = idx+1
    else if S[iter] in [0-9]:
      idx = next[iter]
      if idx == -1:
        num.push(to_int(S[iter:]))
        break
      num.push(to_int(S[iter:idx]))
      iter = idx
    else: # it should be one of [+-*]
      while not ope.empty() and \
          priority(ope.top()) >= priority(S[iter]):
        rhs = num.pop()
        lhs = num.pop()
        num.push( calc(lhs, ope.pop(), rhs) )
      ope.push(S[iter])
      iter += 1
  while not ope.empty():
    rhs = num.pop()
    lhs = num.pop()
    num.push( calc(lhs, ope.pop(), rhs) )
  return num.pop()
```

Since we have already checked whether the arithmetic expression is valid, we can assume something and make the code clearer. For example, we can be sure that whenever it reaches `idx = pair[iter]`, it must not be `-1`, and whenever we pop out an element from stack, the stack must contain at least one element.

Actually, the evaluation part can take linear time only (i.e. $O(|S|)$) if implemented carefully. The most confusing and important part is that the given arithmetic expression may contain extremely large integer such as `999...999+999...999` which may exceed the range of 64-bit integer. However, you can still store the number in an integer array and perform the addition and subtraction manually. For example:

```
def add(a, b): # a, b are integer array
  maxlen = max(len(a),len(b))+1
  c = [0] * maxlen
  for i in 0..maxlen-1:
    if i < len(a):
      c[i] += a[i]
    if i < len(b):
      c[i] += b[i]
    if c[i] >= 10: # carry
      c[i+1] += c[i] / 10
      c[i] = c[i] mod 10
  return c
```

In this case, total time complexity will still remain linear. However, for multiplication, naive implementation will take square time, *i.e.*, $O(|S|^2)$. It will take too much time and cause time limit exceeded. Although, there exists some way to compute multiplication in $O(N \lg N)$ (may be introduced later in the class). It's not so difficult and complicated.

We just need to take advantage of **modular arithmetic**! Since the final answer we are concerned with is modulo by some integer, we can use some facts below:

- $(a+b) \mod c \equiv (a \mod c) + (b \mod c)$
- $(a-b) \mod c \equiv (a \mod c) - (b \mod c)$
- $(a \times b) \mod c \equiv (a \mod c) \times (b \mod c)$

Therefore, we just always mod whatever number we get. Then, the number stored will always be in the range of $[0,10^9+6]$, which can easily fit in 32-bit integer.

The function `to_int` in the above code piece can be implemented as follows:

```
def to_int(S):
  ret = 0
  for c in S:
    ret = ( ret * 10 + ( c - '0' ) ) mod 1000000007

  return ret
```

Problem 5. Good String (20%)

Let $S_{i,j}$ $(1 \leq i \leq j \leq |S|)$ denote the substring of S that starts from i-th character and ends at j-th character (both inclusive). For example, if S is `abcde`, then $S_{1,3}$ is `abc` and $S_{2,4}$ is `bcd`.

With a simple observation, one can find that if $S_{i,j}$ is a *good string*, then $S_{i,j+k}$ is also a *good string* for any positive integer k as long as the substring $S_{i,j+k}$ is legal, i.e. $j + k \leq |S|$.

Let $f(i)$ be a function that returns the minimum integer j such that $S_{i,j}$ is a *good string* or returns $|S| + 1$ if such integer j does not exist.

For a fixed integer i, $S_{i,f(i)}, S_{i,f(i)+1}, \ldots S_{i,|S|}$ are all the *good strings* among all substrings of S that starts from i-th character. So the *good value* of string S can be calculate as $\sum_{i=1}^{|S|}(|S|+1-f(i))$.

Now, let's focus on how to calculate all $f(i)$ $(1 \leq i \leq |S|)$ efficiently.

First, let's consider the relation of $f(i)$ and $f(i+1)$. Suppose there is an integer i such that $f(i) > f(i+1)$. That is, $S_{i+1,f(i+1)}$ is a *good string*, which implies $S_{i,f(i+1)}$ is also a *good string*. However, $f(i)$ should be the minimum integer such that $S_{i,f(i)}$ is a *good string*, which conflicts the $S_{i,f(i+1)}$ being a *good string*. Therefore, with proof by contradiction, we know that $\forall 1 \leq i < |S|, f(i) \leq f(i+1)$. Intuitively, when deducting a character from head of $S_{i,f(i)}$, one only needs to find the number of characters that is required to add to the tail of $S_{i+1,f(i)}$ so that it remains to be a *good string*.

With the observations above, the following algorithm could be obtained.

```
1  // C is a mysterious container
2  // S is input string (1-indexed)
3  j = 0
4  for i = 1 to |S|:
5      if i > 1:
6          deduct character S[i] from C
7      while C does NOT contain all lowercase English alphabets:
8          if j == |S|:
9              break
10         j = j + 1
11         add character S[j] to C
12     if C contains all lowercase English alphabets:
13         f(i) is j
14     else:
15         f(i) is |S| + 1
```

To check if C contains all lowercase English alphabets (in Line 7 and 12), one can use an array as counters and update the counters whenever container C is operated. The following algorithm runs in constant time ($O(1)$).

```
counter[26] = {0}
num_of_type = 0

when add character c to C:
    if counter[c - 'a'] == 0:
        num_of_type += 1
    counter[c - 'a'] += 1

when deduct character c from C:
    counter[c - 'a'] -= 1
    if counter[c - 'a'] == 0:
        num_of_type -= 1

//check() returns True if and only if C contains all lowercase
   English alphabets
def check():
    return num_of_type == 26
```

Since the characters are deducted and added in the order $S[1], S[2], S[3], \ldots$, *queue* is a good choice of this mysterious container C.

The total time complexity is linear because there are only $|S|$ characters to be added to the queue.

For implementation, an explicit queue is not required. Using two variables to specify the head and tail on substrings of S would be fine.