

Homework 1

Due February 7

This is the last homework on which the following information will appear, but remember that it applies to every homework:

- Homeworks are due at 11:59 PM on the date specified. Homeworks are to be placed in the CS 16 handin bin, located by the glass doors on the second floor.
- Please staple your homeworks and put your name and login on each page.
- Please ensure that when prompted to provide pseudocode you follow proper pseudocode formatting guidelines. See the relevant links in the Information section of the website. Commenting of your pseudocode is *strongly* encouraged.
- If you use L^AT_EX, use the `newalg` or `algorithmic` packages (<http://www.cs.brown.edu/system/software/latex/packages.html>) to format your pseudocode.
- When writing pseudocode, you may use any algorithms in the lecture slides or course texts for which there is pseudocode provided. If you do so, please cite the specific lecture slide or page from the book where the algorithm is described.
- Credit for problems comes in part from the simplicity of your answers; insanely long answers lose credit.
- Use pictures to illustrate your ideas.
- Write neatly. Hard-to-read homework gets no credit. If you scratch things out, rewrite and hand in a clean copy.

This is a collaborative homework.

Problem 1.1

Suppose an initially empty stack S has performed a total of 25 `push` operations, 12 `top` operations, and 10 `pop` operations. Three of the `pop` operations generated `EmptyStackExceptions`, which were caught and ignored. Is this enough information to determine the current size of S ? If so, what is it? If not, why not?

Solution: Yes. 18. The initial size of S is zero. Each `push` operation adds one to the size and each successful `pop` operation subtracts one from the size. The `top` operations do not affect the size. Thus, the final size is $0 + 25 - (10 - 3) = 18$.

Problem 1.2

Suppose you have a stack S containing n elements and a queue Q that is initially empty. Describe how you can use Q to scan S to see if it contains a certain element x , with the constraint that your algorithm must finish with S in the same state that it started in, i.e., it must contain the same elements in the same order. You may not use any other data structures – only S and Q and a constant number of reference variables.

Solution: You can reverse the elements in a stack S using a queue Q as follows:

Algorithm 1.2.1 ReverseStackWithQueue(S, Q)

```
while !S.isEmpty() do
  Q.enqueue(S.pop())
end while
while !Q.isEmpty() do
  S.push(Q.dequeue())
end while
```

Thus, we can solve the problem at hand by reversing the stack twice, checking for the element x during one of the reversals. The full solution would look like this:

Algorithm 1.2.2 StackSearchWithQueue(S, Q, x)

```
found ← false
while !S.isEmpty() do
  y ← S.pop()
  if x = y then
    found ← true
  end if
  Q.enqueue(y)
end while
while !Q.isEmpty() do
  S.push(Q.dequeue())
end while
ReverseStackWithQueue(S, Q)
return found
```

Problem 1.3

- (a) Describe a recursive algorithm for enumerating all permutations of the numbers $\{1, 2, \dots, n\}$. For our purposes, a permutation will be an array of length n containing each integer $1 \dots n$ exactly once.

Solution: Here are two approaches:

- To produce the permutations of a set of size $N > 1$, for each element x in the set, recursively calculate the permutations of the rest of the set and append x to each of them. Base case: The set of one element has only one permutation.
 - Consider a helper function h , which takes as input a list of integers L and an integer t . Let $|L|$ denote the length of L . If $|L| = t$, output L (it is a complete permutation). Otherwise, for $i = 0$ to $|L|$ make a copy of L called M , insert into M an element $|L| + 1$ as the i th element, and make the recursive call $h(M, t)$.
Begin the process by calling $h([], n)$, where $[]$ denotes the empty list.
- (b) Describe a non-recursive algorithm for enumerating all permutations of the numbers $\{1, 2, \dots, n\}$. Use only one queue and one stack. You may use control-flow and loop constructs, and allocate as many arrays as you want.

Solution: This algorithm is essentially the same as the second algorithm for part (a), but using a stack instead of recursion.

Algorithm 1.3.1 Permutations(n)

```
let  $A$  be an empty queue
let  $S$  be an empty stack
push an array of length 0 onto  $S$ 
while  $L \leftarrow S.pop()$  do
   $l \leftarrow L.length()$ 
  if  $l = n$  then
     $A.enqueue(L)$ 
  else
    for  $i = 0$  to  $l$  do
      let  $M$  be an array of length  $l + 1$ 
      for  $j = 0$  to  $i - 1$  do
         $M[j] \leftarrow L[j]$ 
      end for
       $M[i] \leftarrow l + 1$ 
      for  $j = i + 1$  to  $l$  do
         $M[j] \leftarrow L[j - 1]$ 
      end for
       $S.push(M)$ 
    end for
  end if
end while
return  $A$ 
```

Problem 1.4

Forty-eight notation is an unambiguous way of writing an arithmetic expression (consisting of integers and the binary operations $+$, $-$, $*$, and $/$) without

parentheses. It is defined so that if “ $(exp_1) \text{ op } (exp_2)$ ” is a normal fully parenthesized expression whose operation is op , then the forty-eight version of this is “ $pexp_1, pexp_2, \text{ op}$ ”, where $pexp_1$ is the forty-eight version of exp_1 and $pexp_2$ is the forty-eight version of exp_2 . The forty-eight version of a single number is just the number itself. So, for example, the forty-eight version of “ $((5+2)*(8-3))/4$ ” is “5, 2, +, 8, 3, -, *, 4, /”. Describe a non-recursive algorithm for evaluating an expression in forty-eight notation. You are given on operations `getNextToken()` which reads the expression from left to right, providing a single integer or operation each time it is called. Hint: there is a data structure that will be particularly useful.

Solution: Read the expression from left to right. When you read a number, push it onto the stack. When you see an operation, pop two numbers off the stack, apply the operation to them, and push the result onto the stack. When the entire expression has been read, the stack should contain a single number which is the value of the expression.

Problem 1.5

When a share of common stock of some company is sold, the capital gain (or loss) is the difference between the share’s selling price and the price originally paid to buy it. This rule is easy to understand for a single share, but if we sell multiple shares of stock bought over a long period of time, then we must identify the shares actually being sold. A standard accounting principle in such a case is to use a FIFO protocol—the shares sold are the ones that have been held the longest. For example, suppose we buy 100 shares at \$20 each on day 1, 20 shares at \$24 on day 2, 200 shares at \$36 on day 3, and then sell 150 shares on day 4 at \$30 each. The capital gain in this case would be $(100*10) + (20*6) + (30*(-6))$, or \$940. Give the pseudocode for a program that takes as input a sequence of transactions of the form “buy x shares at \$ y each” or “sell x shares at \$ y each,” assuming that the transactions occur on consecutive days and the values x and y are integers. Given this input sequence, the output should be the total capital gain (or loss) for the entire sequence, using the FIFO protocol to identify shares. (The stock of only one company is being bought and sold.)

Solution:

```
let  $Q$  be an empty queue
 $gain \leftarrow 0$ 
while  $t \leftarrow \text{nextTransaction}()$  do
  if  $t$  is of the form “buy  $x$  shares at $ $y$  each” then
    for  $i = 1$  to  $x$  do
       $Q.\text{enqueue}(y)$ 
    end for
  else if  $t$  is of the form “sell  $x$  shares at $ $y$  each” then
    for  $i = 1$  to  $x$  do
       $z \leftarrow Q.\text{dequeue}()$ 
       $gain \leftarrow gain + y - z$ 
    end for
```

```
    end if
  end while
return gain
```

Problem 1.6

Dilbert is responsible for organizing a marathon. Because the best runners get to start first, he needs a way to order people from front to back. Marathoners wear number-tags so that it's easy to tell who finished first. The simple thing to do would be to assign number 1 to the top seeded runner, number 2 to the next, and so on. But because he's Dilbert, he decides to give everyone a BINARY number instead. [include pointer to Wikipedia about binary numbering for those who haven't seen it]. When it comes time to race, he wants everyone to line up in order...but most marathoners don't know binary. So he has them queue up randomly on a long street.

With his megaphone, he says "Everyone whose rightmost digit is a 1, take one step to the right; everyone else take one step to the left." Now he has two queues. The people in the left queue are told "step forward until you're close to the person in front of you." Those in the right queue are told "step BACKWARD until you're close to the person behind you." (The backmost person is told not to move). When everyone's settled, the tail of the left queue is just a little in front of where the head of the right queue is.

(a) Explain why.

Now he tells everyone to take a step to the right or left so that they're once again in a single queue. They repeat the process for the second-to-rightmost digit. And then the third-to-rightmost digit, and so on until the very last digit.

You can hand-simulate the process with a deck of cards. Extract the club suit and shuffle it. Associate to each card its binary representation (A = 0001, 2 = 0010, ...9 = 1001, 10=1010, J=1011, Q=1100, K=1101). [You may also want to just get 13 index cards and write these 13 binary numbers on them!] Shuffle the deck. Holding it in your hand face up, look at the first card. If its last binary digit is "1" (i.e., if it's an odd number), place it face-down in a pile on your right; if it's even, place it face-down in a pile on your left. Repeat until you have no more cards in your hand. Then place the face-down "1" pile atop the face-down "0" pile. Pick up the resulting pile and turn it over so that you can see the cards. Now repeat the process for the 2's place, the 4's place, and the 8's place. When your done, the deck should be sorted from lowest to highest (as viewed from the face-up side).

(b) Carry out this exercise until it works for you. On your handin, write "I did part b."

(c) Explain, as best you can, why the runners (or the cards) are now in order from smallest to largest.

(d) Implement this idea in pseudocode: you have a queue full of 12-bit binary numbers. If n is one of these numbers, then $n.bit(i)$ returns either 0 or 1. The bits are ordered from right to left, so if

$n = 0000\ 0000\ 1101$ then

n.bit(0) = 1
n.bit(1) = 0
n.bit(2) = 1
n.bit(3) = 1
n.bit(4) = 0
n.bit(k) for k = 5, 6, ..., 11 are all zero too.

Write pseudocode for `marathonSort`, where the input is a queue of numbers and the output is a queue containing the same numbers in increasing order.

(e) What's the running time of your algorithm in terms of n ?