

Warm-up

Problem 1. Suppose we implement a stack using a singly linked list. What would be the complexity of the push and pop operations? Try to be as efficient as possible.

Solution 1. To push an element we add it at the beginning of the list. To pop an element we delete and return the first element of the list. Both operations take $O(1)$ time.

Problem 2. Suppose we implement a queue using a singly linked list. What would be the complexity of the enqueue and dequeue operations? Try to be as efficient as possible.

Solution 2. To enqueue an element we add it at the end of the list. To dequeue an element we remove it from the front of the list. To keep the enqueue operations efficient, we can keep a pointer to the last element of the list, so that both operations take $O(1)$.

Problem solving

Problem 3. Given a singly linked list, we would like to traverse the elements of the list in reverse order.

- Design an algorithm that uses $O(1)$ extra space. What is the best time complexity you can get?
- Design an algorithm that uses $O(\sqrt{n})$ extra space. What is the best time complexity you can get?

You are not allowed to modify the list, but you are allowed to use position/cursors to move around the list.

Solution 3.

- We keep a cursor that initially points to the last element of the list. We iteratively scan the list until we find the node before the cursor, visit the element at the cursor, and update the cursor to the previous node. The time complexity is $\Theta(n + n - 1 + \dots + 1) = \Theta(n^2)$.

- b) We store \sqrt{n} cursors evenly spaced along the list. We traverse the span between two of these cursors using the previous strategy. Each of these segments is \sqrt{n} long, so each takes $\Theta(\sqrt{n}^2) = \Theta(n)$ time. There are \sqrt{n} many such segments, so the total time is $\Theta(n^{3/2})$.

We can even do better if we are more aggressive on the data that we store. To scan between two cursors, we can traverse the chunk in the forward direction storing all the elements on a stack of size \sqrt{n} . Then we can use the smaller stack to traverse that segment in reverse order in $O(\sqrt{n})$ time. In this way, the space is still $O(\sqrt{n})$ and the overall time is $O(n)$.

Problem 4. Suppose that we use a dynamic array implementation of a stack where each time we run out of space we increase the capacity of the array by $\lceil \sqrt{N} \rceil$. In other words, if the array has capacity N and becomes full, we replace it with another one of capacity $N + \lceil \sqrt{N} \rceil$.

Show that performing n push operations takes $\Theta(n^{3/2})$ time.

Solution 4. Each push operation that does not increase the capacity of the array can be performed in $O(1)$ time, so those contribute only $O(n)$. If a push operation triggers an increase in capacity we need to spend $\Theta(N)$ time, where N is the capacity of the array at that time. The idea is to spread this work among integers in the range $[N, \dots, N + \lceil \sqrt{N} \rceil]$, so that each gets assigned $\sqrt{N} + 1$ work.

Let N' be the final capacity of the array after n push operations. Since $N' = \Theta(n)$, each integer in $[1, \dots, N' + \sqrt{N'}]$ gets charged at most $\sqrt{N'} + 1 = O(\sqrt{n})$ and so the total amount of work is at most $O(n^{3/2})$. On the other hand, every integer after $N'/2 + \sqrt{N'}$ gets charged at least $\sqrt{N'} + 1$, so total amount of work is at least $\Omega(n^{3/2})$. Therefore, the total work is $\Theta(n^{3/2})$.

Problem 5. Consider the problem of given an integer n , generating all possible permutations of the numbers $\{1, 2, \dots, n\}$. Provide a recursive algorithm for this problem.

Solution 5. The helper function outputs permutations of the input array A that have the first i elements fixed.

The correctness of the algorithm hinges on the fact that while the helper function and its many recursive calls may modify the array during their execution, when a call to a helper function finally returns, the input array is always restored the state it was in when the call started executing.

```

1  def permutations-recursive(n):
2      def helper(A, i):
3          if i == n:
4              print(A)
5          for j in [i, n):
6              swap A[i] and A[j]
7              helper(A, i+1)
8              swap A[i] and A[j]
9      A ← array([1, 2, ..., n])
10     helper(A, 0)

```

Problem 6. Consider the problem of given an integer n , generating all possible permutations of the numbers $\{1, 2, \dots, n\}$. Provide a non-recursive algorithm for this problem using a stack.

Solution 6. For the non-recursive version we simulate the calls to the helper function with a stack. We use the tuple (c, i, j) to denote stages of a call. The tuple $(\text{"start"}, i, j)$ corresponds to the start of the for loop for some choice of (i, j) and the tuple $(\text{"finish"}, i, j)$ to the part of the body of the for loop after the recursive call to HELPER.

```

1  def permutations(n):
2      A ← array([1, 2, ..., n])
3      S ← a stack with the tuple ("start", 0, 0)
4      while S is not empty:
5          c, i, j ← S.pop()
6          if c == "start"
7              if i == n
8                  print(A)
9              else
10                 A[i], A[j] ← A[j], A[i]
11                 S.push("finish", i, j)
12                 S.push("start", i+1, i+1)
13             else if c == "finish"
14                 A[i], A[j] ← A[j], A[i]
15                 if j < n-1
16                     S.push('start', i, j+1)

```

Problem 7. Using only two stacks, provide an implementation of a queue. Analyze the time complexity of enqueue and dequeue operations.

Solution 7. The simplest solution is to push elements as they arrive into the first stack. When we are required to carry out a dequeue operation, we transfer all the elements to the second stack, pop to later return the element on the top of the second stack, and then transfer back all the remaining elements to the first stack.

This strategy works because when we transfer the elements from one stack to

the next, we reverse the order of the elements. Before we transfer things, the most recent element to be queued is at the top of the first stack. After we transfer we have the oldest element queued at the top of the second stack. Finally, when we transfer the elements back to the first, we go back to the original stack order.

If the queue holds n elements each enqueue operation takes $O(1)$ time and each dequeue takes $O(n)$ time since we need to transfer all n elements twice.