Reverse Linked List - LeetCode 206.

[206. Reverse Linked List](#_4wmn29l6xxha)

[Solution 1 - Recursive Thinking - O(n^2) time and O(n) stack space](#_fpl6k0xeeils)

[Solution 2 - Recursive Thinking - O(n^2) time and O(n) stack space](#_gkgux32f8g01)

[Solution 3 - Recursive - O(n) time and O(n) stack space](#_q7cxkwwzdy5x)

[Solution 4 - Recursive - O(n) time and O(n) stack space](#_j0kofccjeugm)

[Solution 5 - Recursive with f(1..n-1) - O(n^2) time and O(n) stack space](#_crsir5wz969t)

[Solution 6 - Recursive with Preprocessing - O(n) time and O(n) stack space](#_xdxapbwrb8dk)

[Solution 7 - Recursive with Piggybacking - O(n) time and O(n) stack space](#_i3psoderg9xk)

[Solution 8 - Bottom-up - O(n) time and O(1) extra space](#_xqkbdjju0m60)

[Solution 9 - Stack-based reversing - O(n) time and O(1) extra space](#_ttiny7oqav3q)

# [206. Reverse Linked List](https://leetcode.com/problems/reverse-linked-list)

/\*\*

\* Definition for singly-linked list.

\* struct ListNode {

\* int val;

\* ListNode \*next;

\* ListNode() : val(0), next(nullptr) {}

\* ListNode(int x) : val(x), next(nullptr) {}

\* ListNode(int x, ListNode \*next) : val(x), next(next) {}

\* };

\*/

## Solution 1 - Recursive Thinking - O(n^2) time and O(n) stack space

// O(n^2) time and O(n) stack space

class Solution1 {

public:

ListNode\* reverseList(ListNode\* head) { //f(n), f(1..n)

if(head == nullptr) return head;

auto reversed = reverseList(head->next); //f(n-1), f(2..n)

reversed = append(reversed, head);

return reversed;

}

private:

ListNode\* append(ListNode\* list, ListNode\* node) {

node->next = nullptr;

auto ptr = list;

if(ptr == nullptr) return node;

while(ptr->next != nullptr) ptr = ptr->next;

ptr->next = node;

return list;

}

};

## Solution 2 - Recursive Thinking - O(n^2) time and O(n) stack space

// O(n^2) time and O(n) stack space

class Solution2 {

public:

ListNode\* reverseList(ListNode\* head) {

if(head == nullptr) return head;

if(head->next == nullptr) return head;

auto reversed = reverseList(head->next);

append(reversed, head);

return reversed;

}

private:

void append(ListNode\* list, ListNode\* node) {

assert(list);

node->next = nullptr;

auto ptr = list;

while(ptr->next != nullptr) ptr = ptr->next;

ptr->next = node;

}

};

## Solution 3 - Recursive - O(n) time and O(n) stack space

// O(n) time and O(n) stack space

class Solution3 {

public:

ListNode\* reverseList(ListNode\* head) {

if(head == nullptr) return head;

auto reversed = reverseList(head->next);

auto tail = head->next;

head->next = nullptr;

if(tail == nullptr) {

return head;

} else {

tail->next = head;

return reversed;

}

}

};

## Solution 4 - Recursive - O(n) time and O(n) stack space

// O(n) time and O(n) stack space

class Solution4 {

public:

ListNode\* reverseList(ListNode\* head) {

if(head == nullptr) return head;

if(head->next == nullptr) return head;

auto reversed = reverseList(head->next);

auto tail = head->next;

tail->next = head;

head->next = nullptr;

return reversed;

}

};

## Solution 5 - Recursive with f(1..n-1) - O(n^2) time and O(n) stack space

// O(n^2) time and O(n) stack space

class Solution5 {

public:

ListNode\* reverseList(ListNode\* head) {

if(head == nullptr) return head;

int n = length(head);

return reverse(head, n);

}

private:

ListNode\* reverse(ListNode\* head, int n) { // f(1..n)

if(n == 1) {

head->next = nullptr;

return head;

}

auto nthNode = get\_nth(head, n);

auto reversed = reverse(head, n-1); // f(1..n-1)

nthNode->next = reversed;

return nthNode;

}

ListNode\* get\_nth(ListNode\* head, int n) {

assert(head);

while(--n) head = head->next;

return head;

}

int length(ListNode\* head) {

int n = 0;

while(head) {

head = head->next;

n++;

}

return n;

}

};

## Solution 6 - Recursive with Preprocessing - O(n) time and O(n) stack space

// O(n) time and O(n) extra space with O(n) stack space

// Preprocess and store ith node for each i from 1 to n

class Solution6 {

public:

ListNode\* reverseList(ListNode\* head) {

return nullptr;

}

};

## Solution 7 - Recursive with Piggybacking - O(n) time and O(n) stack space

// O(n) time and O(n) stack space

// Piggybacking

class Solution7 {

public:

ListNode\* reverseList(ListNode\* head) {

if(head == nullptr) return head;

int n = length(head);

return reverse(head, n).first;

}

private:

pair<ListNode\*, ListNode\*> reverse(ListNode\* head, int n) {

if(n == 1) {

auto second = head->next;

head->next = nullptr;

return {head, second};

} else {

// auto nthNode = get\_nth(head, n);

auto reversed = reverse(head, n-1);

auto nthNode = reversed.second;

auto second = nthNode->next;

nthNode->next = reversed.first;

return {nthNode, second};

}

}

ListNode\* get\_nth(ListNode\* head, int n) {

assert(head);

while(--n) head = head->next;

return head;

}

int length(ListNode\* head) {

int n = 0;

while(head) {

head = head->next;

n++;

}

return n;

}

};

## Solution 8 - Bottom-up - O(n) time and O(1) extra space

// O(n) time and O(1) extra space

class Solution8 {

public:

ListNode\* reverseList(ListNode\* head) {

if(head == nullptr) return head;

auto reversed = head;

head = head->next;

reversed->next = nullptr;

while(head) {

auto second = head;

head = head->next;

second->next = reversed;

reversed = second;

}

return reversed;

}

};

## Solution 9 - Stack-based reversing - O(n) time and O(1) extra space

// O(n) time and O(1) extra space

// Move elements from one stack to another

class Solution9 {

public:

ListNode\* reverseList(ListNode\* head) {

auto stk1 = head;

ListNode\* stk2 = nullptr;

while(stk1 != nullptr) {

// pop top of stk1

auto elem = stk1;

stk1 = stk1->next;

// push elem onto stk2

elem->next = stk2;

stk2 = elem;

}

return stk2;

}

};