The University of Michigan
Electrical Engineering & Computer Science
EECS 281: Data Structures and Algorithms

Midterm Exam Written Questions
— Additional Practice —

INSTRUCTIONS: This document contains several written questions to help you prepare for the midterm. The STL and previous exam questions will be available on Gradescope for you to submit your answers and get automatic feedback. The starter files for these questions can be found on Canvas. Good luck!

List of Practice Questions

Question 1	Reverse a Linked List
Question 2	Remove Duplicates from Sorted Linked List
Question 3	Previous Greater Element
Question 4	Merging Intervals
Question 5	List Addition
Question 6	Missing Pair
Question 7	Merging Lists
Question 8	Shifting Zeros
Question 9	Warmer Temperatures
Question 10	2-D Matrix Search
Question 11	Rotated Vector
Question 12	Pair Whose Sum is Closest to Target
Question 13	Social Networking
Question 14	Adding Parentheses to Balance String
Question 15	Design Browser Tab History

STL Algorithm Library Questions

Question 16	`std::unique_copy()`
Question 17	`std::set_difference()`
Question 18	`std::minmax_element()`

Previous Exam Questions

W18 Question 27	Heap Flattening
S18 Question 26	Median Finding
F18 Question 26	Minimal Sorting
W19 Question 26	You're Uniquely Different
F19 Question 26	Market Maker
W20 Question 27	Sorting Student IDs
S20 Question 26	Inverting Parts of a Separated Vector
F20 Question 26	Summing Remnant Values

1. Reverse a Linked List

You are given a singly-linked list, where each `Node` is defined as follows:

```
struct Node {
   int val;
   Node *next;
   Node() : val{0}, next{nullptr} {}
   Node(int x) : val{x}, next{nullptr} {}
   Node(int x, Node *next_in) : val{x}, next{next_in} {}
};
```

Write a program that reverses this singly-linked list. Return the new head node after you're done.

Example: Given the head node of the following list

```
1->2->3->4->nullptr
```

You would return the following list

```
4->3->2->1->nullptr
```

Complexity: $O(n)$ time and $O(1)$ auxiliary space, where n is the length of the list.

Implementation: Implement your solution in the space below. You may **NOT** use anything from the STL. Line limit: 15 lines of code.

```
Node * reverse_list(Node *head) {
  Node *prev = nullptr;
  while (head != nullptr) {
    Node *next = head->next;
    head->next = prev;
    prev = head;
    head = next;
  }
  return prev;
};
```

2. Remove Duplicates from Sorted Linked List

You are given a sorted linked list, where each `Node` is defined as follows:

```
struct Node {
   int val;
   Node *next;
   Node() : val{0}, next{nullptr} {}
   Node(int x) : val{x}, next{nullptr} {}
   Node(int x, Node *next_in) : val{x}, next{next_in} {}
};
```

Write a function that deletes all duplicates in the list so that each element ends up only appearing once.

Example: After passing the list `280->280->281->370->370` into the function, the final list should be `280->281->370`.

Complexity: $O(n)$ time and $O(1)$ auxiliary space, where n is the length of the list.

Implementation: Implement your solution in the space below. You may use anything from the STL. Line limit: 15 lines of code.

```
Node * remove_duplicates(Node *head) {
  Node *curr = head;
  while (curr && curr->next) {
    if (curr->val == curr->next->val) {
      Node *victim = curr->next;
      curr->next = curr->next->next;
      delete victim;
    }
    else {
      curr = curr->next;
    }
  }
  return head;
}
```

3. Previous Greater Element

You are given a non-empty vector of distinct elements, and you want to return a vector that stores the previous greater element that exists before each index. If no previous greater element exists, `-1` is stored.

Example 1: Given `vec = [11, 16, 15, 13]`, you would return `[-1, -1, 16, 15]`.

Example 2: Given `vec = [19, 18, 12, 14, 13]`, you would return `[-1, 19, 18, 18, 14]`.

Complexity: $O(n)$ time and $O(n)$ auxiliary space, where n is the length of the vector.

Implementation: Implement your solution in the space below. You may use anything from the STL. Line limit: 15 lines of code.

```
vector<int> prev_greatest_element(vector<int> &vec) {
  stack<int> s;
  s.push(vec[0]);
  vector<int> res = {-1};
  for (size_t i = 1; i < vec.size(); ++i) {
    while (!s.empty() && s.top() < vec[i])
      s.pop();
    s.empty() ? res.push_back(-1) : res.push_back(s.top());
    s.push(vec[i]);
  }
  return res;
}
```

4. Merging Intervals

You are given a collection of intervals. Write a function that merges all of the overlapping intervals.

Example 1: Given `vec = [[1, 3], [2, 6], [8, 10], [15, 18]]`, you would return `[[1, 6], [8, 10], [15, 18]]`.

Example 2: Given `vec = [[4, 5], [1, 4]]`, you would return `[[1, 5]]`.

Complexity: $O(n\log(n))$ time and $O(n)$ auxiliary memory, where n is the length of the vector of intervals.

Implementation: Implement your solution in the space below. You may use anything from the STL. Line limit: 20 lines of code.

```
struct Interval {
  int start;
  int end;
};

vector<Interval> merge_intervals(vector<Interval> &vec) {
  if (vec.empty()) {
    return vector<Interval>{};
  }
  vector<Interval> result;
  sort(vec.begin(), vec.end(), [](Interval a, Interval b) {
    return a.start < b.start;
  });
  result.push_back(vec.front());
  for (size_t i = 1; i < vec.size(); ++i) {
    if (result.back().end < vec[i].start) {
      result.push_back(vec[i]);
    }
    else {
      result.back().end = max(result.back().end, vec[i].end);
    }
  }
  return result;
}
```

5. List Addition

You are given two non-empty linked lists representing two non-negative integers. The most significant digit comes *first* and each of their nodes contains a single digit. Add the two numbers and return the result as a linked list. You may assume the two numbers do not contain any leading 0's except the number 0 itself. The structure of a `Node` is provided below:

```
struct Node {
   int val;
   Node *next;
   Node() : val{0}, next{nullptr} {}
   Node(int x) : val{x}, next{nullptr} {}
   Node(int x, Node *next_in) : val{x}, next{next_in} {}
};
```

Example: Given the following two lists:

```
1->9->4->7->nullptr
9->3->9->nullptr
```

you would return the list `2->8->8->6->nullptr`.

Complexity: $O(n)$ time and $O(n)$ auxiliary space, where n is the combined length of the lists.

Implementation: Implement your solution in the space below. You may use anything from the STL. Line limit: 30 lines of code.

```
Node * add_lists(Node *list1, Node *list2) {
  stack<int> stack1, stack2;
  while (list1) {
    stack1.push(list1->val);
    list1 = list1->next;
  }
  while (list2) {
    stack2.push(list2->val);
    list2 = list2->next;
  }
  int sum = 0;
  Node *result = new Node(0);
  while (!stack1.empty() || !stack2.empty()) {
    if (!stack1.empty()) {
      sum += stack1.top();
      stack1.pop();
    }
    if (!stack2.empty()) {
      sum += stack2.top();
      stack2.pop();
    }
    result->val = sum % 10;
    Node *head = new Node(sum / 10);
    head->next = result;
    result = head;
    sum /= 10;
  }
  return result->val == 0 ? result->next : result;
}
```

6. Single Element in Sorted Array

You are given a sorted array consisting of only integers where every element appears exactly twice, except for one element which appears exactly once. Write a function that returns the single element.

Example: Given `vec = [1, 1, 2, 3, 3, 4, 4]`, you would return `2`.

Complexity: $O(\log(n))$ time and $O(1)$ auxiliary space, where n is the length of the vector.

Implementation: Implement your solution in the space below. You may use anything from the STL. Line limit: 15 lines of code.

```
int find_single_element(vector<int> &vec) {
  int left = 0;
  int right = vec.size() - 1;
  while (left < right) {
    int mid = (left + right) / 2;
    if ((mid % 2 == 0 && nums[mid] == nums[mid + 1]) ||
        (mid % 2 == 1 && nums[mid] == nums[mid - 1])) {
      left = mid + 1;
    }
    else {
      right = mid;
    }
  }
  return nums[left];
}
```

7. Merging Lists

You are given k sorted linked lists. Write a program that merges all k lists into a single *sorted* list. The structure of a `Node` is provided below:

```
struct Node {
   int val;
   Node *next;
   Node() : val{0}, next{nullptr} {}
   Node(int x) : val{x}, next{nullptr} {}
   Node(int x, Node *next_in) : val{x}, next{next_in} {}
};
```

Example: Given the following four lists:

```
1->5->7->nullptr
4->9->nullptr
3->6->8->10->nullptr
2->nullptr
```

you would return the list `1->2->3->4->5->6->7->8->9->10->nullptr`.

Complexity: $O(nk\log(k))$ time and $O(n)$ auxiliary space, where n is the length of the longest list.

Implementation: Implement your solution in the space below. You may use anything from the STL. Line limit: 25 lines of code.

```
// Solution #1
struct ListCompare {
  bool operator()(const Node *l1, const Node *l2) const {
    return l1->val > l2->val;
  }
};

Node * merge_lists(vector<Node *> &lists) {
  priority_queue<Node *, vector<Node *>, ListCompare> pq;
  for (Node *l : lists)
    if (l) pq.push(l);
  if (pq.empty())
    return nullptr;
  Node *result = pq.top();
  pq.pop();
  Node *current = result;
  if (current->next)
    pq.push(current->next);
  while (!pq.empty()) {
    current->next = pq.top();
    pq.pop();
    current = current->next;
    if (current->next)
```

```
      pq.push(current->next);
  }
  return result;
}

// Solution #2
Node * merge_two_lists(Node *l1, Node *l2) {
  if (!l1) return l2;
  if (!l2) return l1;
  if (l1->val < l2->val) {
    l1->next = merge_two_lists(l1->next, l2);
    return l1;
  }
  else {
    l2->next = merge_two_lists(l1, l2->next);
    return l2;
  }
}

Node * merge_lists(vector<Node *> &lists) {
  deque<Node *> dq{lists.begin(), lists.end()};
  while (dq.size() > 1) {
    Node *first = dq.front();
    dq.pop_front();
    Node *second = dq.front();
    dq.pop_front();
    dq.push_back(merge_two_lists(first, second));
  }
  return dq.front();
}
```

8. Shifting Zeros

You are given a vector of integers, `vec`, and you are told to implement a function that moves all elements with a value of 0 to the end of the vector *while maintaining the relative order of the non-zero elements*.

Example: Given the initial vector `[0, 1, 0, 4, 3]`, you should rearrange the contents of the vector so that the final ordering of elements is `[1, 4, 3, 0, 0]`.

Complexity: $O(n)$ time, $O(1)$ auxiliary space, where n is the length of the vector.

Implementation: Implement your solution in the space below. You may use anything from the STL. Line limit: 15 lines of code.

```
void shift_zeros(vector<int> &vec) {
  size_t shifted_index = 0;
  for (size_t index = 0; index < vec.size(); ++index) {
    if (vec[index] != 0) {
      vec[shifted_index++] = vec[index];
    }
  }
  for (; shifted_index < vec.size(); ++shifted_index) {
    vec[shifted_index] = 0;
  }
}
```

9. Warmer Temperatures

You are given a vector of integers, `temps`, that stores the daily temperature forecasts for the next few days. Write a program that, for each index of the input vector, stores the number of days you need to wait for a warmer temperature. If there is no future day where this is possible, a value of 0 should be stored.

Example 1: Given the following vector:

```
[55, 62, 46, 52, 51, 50, 51, 53, 63]
```

you would return the vector

```
[1, 7, 1, 4, 3, 1, 1, 1, 0]
```

since you would need to wait 1 day on day 0 for a warmer temperature (55 $\rightarrow$ 62), 7 days on day 1 for a warmer temperature (62 $\rightarrow$ 63), and so on.

Example 2: Given the following vector:

```
[74, 74, 73, 75, 74, 73, 72, 71, 70]
```

you would return the vector

```
[3, 2, 1, 0, 0, 0, 0, 0, 0]
```

Your program should run in $O(n)$ time. You may use extra space to store your output, but the rest of your program must use $O(1)$ auxiliary space.

Implementation: Implement your solution in the space below. You may use anything from the STL. Line limit: 20 lines of code.

```
vector<int> warmer_temperatures(vector<int> &temps) {
  vector<int> res(temps.size());
  for (int i = int(temps.size()) - 1; i >= 0; --i) {
    int j = i + 1;
    while (j < temps.size() && temps[j] <= temps[i]) {
      if (res[j] > 0)
        j = res[j] + j;
      else
        j = temps.size();
    }
    if (j < temps.size())
      res[i] = j - i;
  }
  return res;
}
```

10. 2-D Matrix Search

You are given a $m \times n$ matrix in the form of a vector of vectors that has the following properties:

- integers in each row are sorted in ascending order from left to right
- integers in each column are sorted in ascending order from top to bottom

Write a function that searches for a value in this matrix and returns whether the element can be found.

Example: Given the following matrix:

```
[ [ 1,  4,  7, 11, 15],
  [ 2,  5,  8, 12, 19],
  [ 3,  6,  9, 16, 22],
  [10, 13, 14, 17, 24],
  [18, 21, 23, 26, 30]  ]
```

and a target value of 5, you would return `true`. Given a target value of 20, you would return `false`.

Complexity: $O(m + n)$ time, $O(1)$ auxiliary space.

Implementation: Implement your solution in the space below. You may use anything from the STL. Line limit: 20 lines of code.

```
bool matrix_search(vector<vector<int>> &matrix, int target) {
  int numRows = matrix.size();
  if (numRows == 0)
    return false;
  int numCols = matrix[0].size();
  int currRow = 0;
  int currCol = numCols - 1;
  while (currRow < numRows && currCol >= 0) {
    if (matrix[currRow][currCol] == target)
      return true;
    else if (matrix[currRow][currCol] > target)
      --currCol;
    else
      ++currRow;
  }
  return false;
}
```

11. Rotated Vector

Suppose you are given an vector of integers that is sorted in ascending order. However, this vector has been rotated at some pivot unknown to you beforehand.

For instance, the vector `[1, 2, 3, 4, 5]` may be given to you in the form `[3, 4, 5, 1, 2]`, where the vector is rotated at 3. You may assume that no duplicate exists in this array.

Write a function that finds the minimum element in the vector.

Example: Given input `[3, 4, 5, 1, 2]`, you would return 1.

Complexity: $O(\log(n))$ time, $O(1)$ auxiliary space, where n is the length of the vector.

Implementation: Implement your solution in the space below. You may use anything from the STL. Line limit: 15 lines of code.

```
int find_rotated_minimum(vector<int> &vec) {
  int left = 0;
  int right = vec.size() - 1;
  while (left < right) {
    int mid = left + (right - left) / 2;
    if (vec[mid] > vec[right]) {
      start = mid + 1;
    }
    else {
      right = mid;
    }
  }
  return vec[left];
}
```

12. Pair Whose Sum is Closest to Target

You are given a vector of integers and a target value k. Write a function that returns the pair of elements whose sum is closest to k. The smaller element should go first in the pair that is returned.

Example: Given the input vector `[29, 30, 40, 10, 28, 22]` and a target of $k = 54$, you would return the pair `[22, 30]`.

Complexity: $O(n \log(n))$ time, $O(\log(n))$ auxiliary space, where n is the length of the vector.

Implementation: Implement your solution in the space below. You may use anything from the STL. Line limit: 20 lines of code.

```
pair<int, int> closest_sum_to_k(vector<int> &vec, int k) {
  pair<int, int> idx;
  int left = 0, right = vec.size() - 1, best = INT_MAX;
  sort(vec.begin(), vec.end());
  while (left < right) {
    int curr = abs(vec[left] + vec[right] - k);
    if (curr < best) {
      idx.first = left;
      idx.second = right;
      best = curr;
    }
    if (vec[left] + vec[right] > k) {
      --right;
    }
    else {
      ++left;
    }
  }
  return { vec[idx.first], vec[idx.second] };
}
```

13. Social Networking

The EECS 281 staff is testing a brand new social media site, *Fee's Book*, to promote social interaction among EECS students! Suppose there are a total of n students that are invited to the site, and each student is assigned a unique integer ID from 0 to $n-1$. You are given the number of students in the network, `num_students`, and a vector of activity logs, where each log stores an integer timestamp and the IDs of two students on the site:

```
struct Log {
  int timestamp;
  int id_A;
  int id_B;
};
```

Each `Log` object represents the time when students `id_A` and `id_B` became friends. The timestamp is represented in YYYYMMDD format (e.g. February 26, 2020 is represented as the integer 20200226).[*] Friendship is symmetric: if A is friends with B, then B is friends with A.

Let's say that person A is *acquainted* with person B if A and B are in the same friend group; that is, if A is friends with B, or A is a friend of someone acquainted with B. Implement the `earliestAcquaintance()` function, which returns the earliest timestamp for which every person in the network is acquainted with every other person. Return `-1` if there is no such earliest time.

Example 1: Given `num_students = 5` and `friendships = [{20200229, 2, 3}, {20200227, 1, 4}, {20200303, 0, 3}, {20200228, 0, 4}, {20200301, 1, 2}, {20200226, 0, 2}]`, you would return `20200229`, since that is the first time stamp for which all 5 students are acquainted (on the 29th, person 0 is direct friends with 2 and 4, acquainted to 1 via 4, and acquainted to 3 via 2).

Example 2: Given `num_students = 5` and `friendships = [{20200304, 1, 4}, {20200307, 0, 2}, {20200306, 3, 4}]`, you would return -1. In this example, person 0 and person 2 are in a separate friend group and are never acquainted with 1, 3, or 4.

Complexity: $O(n\log(n))$ time, $O(\log(n)+s)$ auxiliary space, where n represents the number of logs in the vector, and s represents the number of students.

Implementation: Implement your solution on the back of this page. You may use anything from the STL. Line limit: 30 lines of code.

Hint: You may want to use additional helper functions to solve this problem.

[*]This allows you to compare timestamps by directly comparing the integers themselves - you don't need to parse anything!

Implement your solution to "Social Networking" below:

```
struct LogComp {
  bool operator() (const Log& lhs, const Log& rhs) const {
    return lhs.timestamp < rhs.timestamp;
  }
};

// find the representative (using path compression, needed for
// amortized constant time)
int find(vector<int>& reps, int id) {
  if (id == reps[id]) return id;
  reps[id] = find(reps, reps[id]);
  return reps[id];
}

int earliest_acquaintance(vector<Log> &friendships, int num_students) {
  LogComp comp;
  sort(friendships.begin(), friendships.end(), comp);
  vector<int> reps(num_students);
  int count = 0;
  for (int i = 0; i < num_students; ++i) {
    // set each student to its own representative (can also use
    // std::iota to save lines)
    reps[i] = i;
  }
  for (Log& f : friendships) {
    int rep_A = find(reps, f.id_A);
    int rep_B = find(reps, f.id_B);
    if (rep_A != rep_B) {
      reps[rep_B] = rep_A;
      ++count;
    }
    // done when n - 1 connections are made
    if (count == num_students - 1) {
       return f.timestamp;
    }
  }
  return -1;
}
```

14. Adding Parentheses to Balance String

You are given a string `str` that consists of the characters `'('` and `')'`. Write a function that returns the minimum number of parentheses (either `'('` or `')'`) that need to be added to the string so that the resulting string is balanced.

Example 1: Given the input string `"())"`, you would return `1`, since only 1 new parenthesis needs to be added to balance this string (adding `'('` at the beginning).

Example 2: Given the input string `"()))(("`, you would return `4`, since a minimum of 4 new parentheses are needed to balance the string: `"()`**`(`**`)`**`(`**`)((`**`))`**`"`).

Complexity: $O(n)$ time, $O(1)$ auxiliary space, where n is the length of the string.

Implementation: Implement your solution in the space below. You may use anything from the STL. Line limit: 15 lines of code.

```
int min_add_to_make_string_valid(string str) {
  int left = 0, right = 0;
  for (char c : str) {
    if (c == '(') {
      ++right;
    }
    else if (right > 0) {
      --right;
    }
    else {
      ++left;
    }
  }
  return left + right;
}
```

15. Design Browser Tab History

In this question, you will be implementing a class that simulates a browser tab. You will start on the given `homepage` (initialized in the constructor), and you can visit different `url` strings. You are also able to go back or move forward in the URL history a given number of `steps`.

Implement the following `BrowserTab` class. Feel free to add any member variables that you may find useful.

```
class BrowserTab {
  // Add any member variables that you may find useful here.
  stack<string> h_back, h_forward;
  string current;
public:
  // This constructor inits the BrowserTab with the homepage of browser.
  BrowserTab(string homepage) {
    current = homepage;
  }
  // This function visits the given url from the current page.
  // Calling this function clears up all the forward history.
  void visit(string url) {
    h_forward = stack<string>();
    h_back.push(current);
    current = url;
  }
  // This function moves back in history a total of "steps" steps (or as far
  // back as possible if the number of steps specified exceeds the history
  // length). Return the url you end on after moving back in history.
  string back(int steps) {
     while (--steps >= 0 && !h_back.empty()) {
       h_forward.push(current);
       current = h_back.top();
       h_back.pop();
     }
     return current;
  }
  // This function moves forward in history a total of "steps" steps (or as
  // far forward as possible if the number of steps exceeds the history
  // length). Return the url you end on after moving forward in history.
  string forward(int steps) {
    while (--steps >= 0 && !h_forward.empty()) {
      h_back.push(current);
      current = h_forward.top();
      h_forward.pop();
    }
    return current;
  }
};
```

STL Practice Question 1: `std::unique_copy()`

Implement STL's `unique_copy()` function according to its official interface and description.

```
template <class ForwardIterator, class OutputIterator>
OutputIterator unique_copy(ForwardIterator first, ForwardIterator last,
                           OutputIterator result);
```

Quoting from `http://www.sgi.com/tech/stl/unique_copy.html`:

`unique_copy()` copies elements from the range `[first, last)` to a range beginning with `result`, except that in a consecutive group of duplicate elements only the first one is copied. The return value is the end of the range to which the elements are copied.

Complexity: Linear. For elements in the ranges, exactly `last - first` applications of `operator==()` and at most `last - first` assignments.

For example, if you executed this code:

```
int data[6] = {1, 3, 3, 1, 1, 0}, output[6];
unique_copy(data, data + 6, output);
```

You would have this in array `output`:

1	3	1	0		

Implementation: Use the back of this page as a working area, then rewrite **neatly** on the front. Limit: 15 lines of code (points deducted if longer). You may **NOT** use other STL algorithms/functions.

```
template <class ForwardIterator, class OutputIterator>
OutputIterator unique_copy(ForwardIterator first, ForwardIterator last,
                           OutputIterator result) {
  if (first == last)
    return result;

  *result++ = *first; // establish a starting point
  ForwardIterator prev = first++;

  while (first != last) {
    if (!(*first == *prev)) // *first != *prev
      *result++ = *first;
    prev = first++;
  } // while
  return result;
}
```

STL Practice Question 2: `std::set_difference()`

Implement STL's `set_difference()` function according to its official interface and description.

```
template <class ForwardIterator1, class ForwardIterator2,
          class OutputIterator, class Compare>
OutputIterator set_difference(ForwardIterator1 first1, ForwardIterator1 last1,
                              ForwardIterator2 first2, ForwardIterator2 last2,
                              OutputIterator result, Compare comp);
```

The `set_difference()` function constructs a sorted range beginning in the location pointed to by `result` with the set difference of the sorted range `[first1,last1)` with respect to the sorted range `[first2,last2)`. The difference of two sets is formed by the elements that are present in the first set, but not in the second one. The elements copied by the function always come from the first range, in the same order. For containers supporting multiple occurrences of a value, the difference includes as many occurrences of a given value as in the first range, minus the number of matching elements in the second, preserving order. The elements are compared using the comparator `comp`. Two elements, `a` and `b`, are considered equivalent if `!comp(a,b) && !comp(b,a)`. For example, given the vectors

```
first = [5, 10, 15, 20, 25]
second = [10, 20, 30, 40, 50]
```

the set difference constructed at `result` is `[5, 15, 25]` (elements in the first range not in second).

Implement the function below. You may **NOT** use any STL algorithms/functions. The function returns an iterator to the end of the constructed range. The program must run in linear time, and you are limited to 15 lines of code.

```
template <class ForwardIterator1, class ForwardIterator2,
          class OutputIterator, class Compare>
OutputIterator set_difference(ForwardIterator1 first1, ForwardIterator1 last1,
                              ForwardIterator2 first2, ForwardIterator2 last2,
                              OutputIterator result, Compare comp) {
  while (first1 != last1 && first2 != last2) {
    if (comp(*first1, *first2))
      *result++ = *first1++;
    else if (comp(*first2, *first1))
      ++first2;
    else {
      ++first1;
      ++first2;
    }
  }
  while (first1 != last1)
    *result++ = *first1++;
  return result;
}
```

STL Practice Question 3: `std::minmax_element()`

Implement STL's `minmax_element()` function according to its official interface and description.

```
template <class ForwardIterator, class Compare>
pair<ForwardIterator, ForwardIterator> minmax_element(ForwardIterator first,
                                        ForwardIterator last, Compare comp);
```

The `minmax_element()` function returns a pair with an iterator pointing to the element with the smallest value in the range `[first, last)` as the `first` element, and an iterator pointing to the largest value in the range as the `second` element. The comparisons are performed using the `comp` comparator. If more than one equivalent element has the smallest value, the first iterator points to the *first* of such elements. If more than one equivalent element has the largest value, the second iterator points to the *last* of such elements.

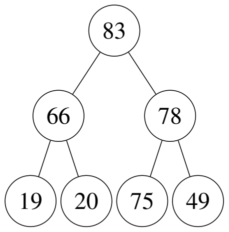
For example, given the vector `vec = [3, 7, 6, 9, 5, 8, 2, 4]`, running the function with `vec.begin()` as the first argument, `vec.end()` as the second argument, and `operator<` as the comparator, a pair would be returned with an iterator to `2` as the first element and an iterator to `9` as the second element.

You may **NOT** use any STL algorithms/functions. The program must run in linear time, and you are limited to 15 lines of code.

```
template <class ForwardIterator, class Compare>
pair<ForwardIterator, ForwardIterator> minmax_element(ForwardIterator first,
                                        ForwardIterator last, Compare comp) {
  ForwardIterator min = first, max = first;
  for (; first != last; ++first) {
    if (comp(*first, *min)) {
      min = first;
    }
    if (!comp(*first, *max)) {
      max = first;
    }
  }
  return {min, max};
}
```

Winter 2018 Midterm Exam Question 27: Heap Flattening

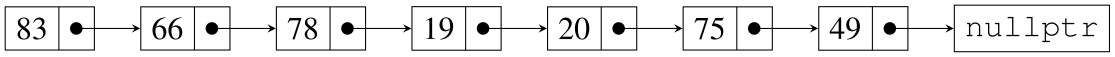
Given a pointer-based binary heap, flatten it into a **singly**-linked list. The binary heap passed in (as a `Node *`) must be modified directly. DO NOT create or destroy any `Node`s. The left child should appear in the linked list before the right child. An example is shown below:



Each `Node` is defined as:

```
struct Node {
   int  val;
   Node *left;
   Node *right;
};
```

Turns into:



The `right` pointer within each `Node` must be used as “next”, the `left` must be set to `nullptr`.

Complexity: $O(n)$ time, up to $O(n)$ additional space BUT you cannot create or delete `Nodes`.

Implementation: Write your code neatly in the space below. Limit: 18 lines of code (points deducted if longer). You **MAY** use anything from the STL.

```
void linearize(Node *binary_heap) {
  queue<Node *> q;
  q.push(binary_heap);
  while (!q.empty()) {
    Node *temp = q.front();
    q.pop();
    if (temp) {
      q.push(temp->left);
      q.push(temp->right);
      temp->left = nullptr;
      temp->right = q.front();
    }
  }
}
```

Spring 2018 Midterm Exam Question 26: Median Finding

Suppose that you are given an array of `double` values. You want to find the median element quickly, and you decide that you can make use of quicksort's `partition()` function. However, you don't need to fully sort the array, and want to do less work than that. The prototype for partitioning the array is:

```
// Partition the array a from the left index to right.
// The left index is inclusive, right is exclusive.
// Returns the index of the pivot element.
int partition(double a[], int left, int right);
```

When there are an odd number of elements, this will be sufficient. However, when there are an even number of elements, you *might* find this function useful (but it is not necessary):

```
// Find the smallest element of array a from the left index to right.
// The left index is inclusive, right is exclusive.
// Assumes the range [left, right) contains at least one element.
// Returns the minimum value in that range.
double find_min(const double a[], int left, int right);
```

Note: DO NOT write the `partition()` or `find_min()` functions, only use them as needed.

Implementation: Write your solution on the next page. You can use the area below (and on the next page) to work out your solution. Make your solution as efficient as possible; points will be deducted for unnecessary complexity. Limit: 20 lines of code. Points will be deducted from longer solutions. You may **NOT** use any STL algorithms/functions.

Hint: Think recursion! If you need a helper function, just write it below the given function.

Implement your solution to "Median Finding" below:

Solution 1 (Recursive):

```
double find_median(double a[], int left, int right) {
  int middle = (left + right - 1) / 2;
  if ((right - left) % 2 == 0) {
    double med1 = find_median_helper(a, left, right, middle);
    double med2 = find_median_helper(a, left, right, middle + 1);
    return (med1 + med2) / 2;
  }
  return find_median_helper(a, left, right, middle);
}

double find_median_helper(double a[], int left, int right, int middle) {
  int pivot = partition(a, left, right);
  if (pivot < middle)
    return find_median_helper(a, pivot + 1, right, middle);
  else if (pivot > middle)
    return find_median_helper(a, left, pivot, middle);
  return a[middle];
}
```

Solution 2 (Iterative):

```
double find_median(double a[], int left, int right) {
  int middle = (left + right - 1) / 2;
  int pleft = left, pright = right;
  while (true) {
    int pivot = partition(a, pleft, pright);
    if (pivot == middle) {
      double median = a[middle];
      if ((right - left) % 2 == 0)
        median = (a[middle] + find_min(a, middle + 1, right)) / 2;
      return median;
    }
    else if (pivot < middle)
      pleft = pivot + 1;
    else
      pright = pivot;
  }
}
```

Fall 2018 Midterm Exam Question 26: Minimal Sorting

Given a vector of integers (of size n, $n > 1$), find the smallest section of the vector such that sorting that section will make the entire vector sorted (in increasing order). You should display the first and last indices that need to be sorted (BOTH inclusive, using 0-based indexing). You can assume that your function WILL be called with a vector containing exactly one such section.

For example, given the input vector: `{ 1, 2, 5, 7, 3, 6, 4, 8 }`

Your output should be: `Sort from index 2 to index 6`

Hint: You can use `INT_MAX` and `INT_MIN` for the largest and smallest values that can fit inside a variable of type `int`.

Requirements: Your solution runtime must be no worse than $O(n)$ time. You may use up to $O(1)$ auxiliary space.

Implementation: Write your code neatly in the space below. Limit: 20 lines of code (points deducted if longer). You may **NOT** use any STL algorithms, functions or containers (except the provided `vector`).

```
void find_subarray(const vector<int> &v) {
  int left_idx = -1, right_idx = -1;
  int max_so_far = INT_MIN, min_so_far = INT_MAX;
  for (int i = 0; i < v.size(); i++) {
    if (max_so_far < v[i])
      max_so_far = v[i];
    if (v[i] < max_so_far)
      right_idx = i;
  } // for
  for (int i = (v.size()-1); i >= 0; i--) {
    if (min_so_far > v[i])
      min_so_far = v[i];
    if (v[i] > min_so_far)
      left_idx = i;
  } // for
  cout << "Sort from index " << left_idx << " to index " << right_idx << endl;
} // find_sub_array()
```

Winter 2019 Midterm Exam Question 26: You're Uniquely Different

Given two vectors, `a` and `b`, return a vector of all unique elements that are **NOT** members of both `a` and `b`. If there are duplicate elements within one vector that do not appear in the other, you may select an arbitrary element from the duplicates to include (i.e., you may pick the element to include; see the example with `8` in vector `b` below). Elements in the vectors may be compared using the < and == operators. You may return the output in any order.

Example: Given

```
vector<int> a = {2, 1, 0, 5, 3, 7};
vector<int> b = {0, 5, 8, 0, 8};
```

calling `unique_difference(a, b)` will return a vector containing `{1, 2, 3, 7, 8}` (in any order).

Complexity: $O(n \log(n))$ time and $O(\log(n))$ space (in addition to memory for the returned vector), where `n = a.size() + b.size()`.

Implementation: Write your code neatly in the space below. Limit: 30 lines of code (points deducted if longer). You **MAY** use any STL algorithms/functions.

Solution 1 (STL):

```
template <class T>
vector<T> unique_difference(vector<T> a, vector<T> b) {
  vector<T> out;
  auto out_end = back_inserter(out);

  sort(a.begin(), a.end());
  sort(b.begin(), b.end());
  auto a_end = unique(a.begin(), a.end());
  auto b_end = unique(b.begin(), b.end());

  set_symmetric_difference(a.begin(), a_end, b.begin(), b_end, out_end);
  return out;
}
```

Solution 2 (Manual):

```
template <class T>
vector<T> unique_difference(vector<T> a, vector<T> b) {
  vector<T> out;

  sort(a.begin(), a.end());
  sort(b.begin(), b.end());

  size_t a_idx = 0, b_idx = 0;

  while (a_idx != a.size() && b_idx != b.size()) {
    if (a[a_idx] < b[b_idx]){
      if(out.empty() || !(a[a_idx] == out.back() ))
        out.push_back(a[a_idx]);
      ++a_idx;
    }
    else if(b[b_idx] < a[a_idx]){
      if(out.empty() || !(b[b_idx] == out.back() ))
        out.push_back(b[b_idx]);
      ++b_idx;
    }
    else {
      T back = a[a_idx];
      while (a[a_idx] == back) { ++a_idx; }
        while (b[b_idx] == back) { ++b_idx; }
    }
  }

  while (a_idx != a.size()) {
    T back = a[a_idx];
    out.push_back(a[a_idx++]);
    while (a[a_idx] == back) {
      ++a_idx;
    }
  }

  while (b_idx != b.size()) {
    T back = b[b_idx];
    out.push_back(b[b_idx++]);
    while (b[b_idx] == back) {
      ++b_idx;
    }
  }

  return out;
}
```

Solution 3 (Binary Search):

```
template <class T>
vector<T> unique_difference(vector<T> a, vector<T> b) {
  vector<T> out;

  sort(a.begin(), a.end());
  sort(b.begin(), b.end());

  for(auto &x : a) {
    if(!binary_search(b.begin(), b.end(), x)) {
      if(out.empty() || !(x == out.back() ))
        out.push_back(x);
    }
  }

  for(auto &x : b) {
    if(!binary_search(a.begin(), a.end(), x)) {
      if(out.empty() || !(x == out.back() ))
        out.push_back(x);
    }
  }

  return out;
}
```

Fall 2019 Midterm Exam Question 26: Market Maker

Assume you are a market maker in an electronic stock exchange. You are given a stream of stock data which includes an identifier (a unique abbreviation that identifies the stock), the highest price that a buyer is willing to pay for the stock, and the lowest price that a seller is willing to sell the stock for. Each stock will only appear **once** in the stream and will have *exactly one buyer and one seller*.

The difference between the buyer and seller prices (i.e. buyer price - seller price) is the amount of money you earn from participating in the transaction. For instance, if a buyer is looking to buy EECS stock at \$10 and a seller is offering EECS stock for \$7, you can make a profit of \$10 - \$7 = \$3 by buying from the seller at \$7 and selling to the buyer for \$10. **If buyer price - seller price for a stock is negative, that stock should be ignored, as you would lose money if you tried to force a transaction!**

You will implement the `max_profit` function (defined on the next page), which takes in a stream of stock information `stock_in` and a positive integer k. **The function returns the maximum profit you can make from trading at most k stocks from the entire stream.** It is possible for fewer than k transactions to occur, since not all stock transactions may produce positive profits (see example 2 on the back). The `stock_in` stream will contain data for a total of n stocks in the following format:

<stock_id1> <buy_price1> <sell_price1> <stock_id2> <buy_price2> <sell_price2> … <stock_id-n> <buy_price-n> <sell_price-n>

Constraints:

- Your solution should run in at most $\Theta(n \log(k))$ time, where n is the number of stocks in the stream. The value of n will NOT be given to you.
- Your solution should use at most $\Theta(k)$ space.
- You may assume that $0 < k \leq n$.
- Your solution should NOT make use of custom-defined structs or classes.

You may use the space below (as well as the previous page) as a working area. Write your code neatly in the space provided on the back of this page. **The back of this page also includes additional examples.**

Hint: *The `stock_in` stream behaves like `cin`, and its contents can be extracted using `operator»`. For example, if you had a string variable named `stock_id` and two double variables named `buy_price` and `sell_price`, you can read the contents of a single stock using the following line:*

```
stock_in » stock_id » buy_price » sell_price;
```

Example 1: Given an integer $k = 3$ and the following `stock_in` stream:

EECS 5.00 2.00 BBB 81.00 79.00 IOE 42.50 42.00 EWRE 20.53 15.53 GGBL 22.15 21.15

You would return the value **10.00**, since that is the maximum profit you can make from trading at most 3 stocks in the stream (EECS for a profit of \$5 - \$2 = \$3, BBB for a profit of \$81 - \$79 = \$2, and EWRE for a profit of \$20.53 - \$15.53 = \$5) $\rightarrow$ \$3 + \$2 + \$5 = \$10.

Example 2: Given an integer $k = 5$ and the following `stock_in` stream:

DOW 14.41 12.41 COOL 45.19 46.83 LBME 16.63 16.61 FXB 3.14 8.11 NAME 12.79 12.00

You would return the value **2.81**, since that is the maximum profit you can make from trading at most 5 stocks in the stream (DOW for a profit of \$14.41 - \$12.41 = \$2, LBME for a profit of \$16.63 - \$16.61 = \$0.02, and NAME for a profit of \$12.79 - \$12.00 = \$0.79) $\rightarrow$ \$2 + \$0.02 + \$0.79 = \$2.81. Even though you are allowed to trade up to 5 stocks, only 3 are traded because the buyer prices of COOL and FXB are lower than their respective seller prices.

Complexity: At most $\Theta(n \log(k))$ time and $\Theta(k)$ space (see "Constraints" on previous page).

Implementation: Write your code neatly in the space below. Limit: 25 lines of code (points deducted if longer). You MAY use anything in the STL.

```cpp
double max_profit(istream &stock_in, int k) {
  string stock_name;
  double buy_price, sell_price, profit = 0;
  priority_queue<double, vector<double>, greater<double>> best_stocks;
  while (stock_in >> stock_name >> buy_price >> sell_price) {
    if (buy_price > sell_price) {
      if (best_stocks.size() < k) {
        best_stocks.push(buy_price - sell_price);
      }
      else if (best_stocks.top() < buy_price - sell_price) {
        best_stocks.pop();
        best_stocks.push(buy_price - sell_price);
      }
    }
  }
  while (!best_stocks.empty()) {
    profit += best_stocks.top();
    best_stocks.pop();
  }
  return profit;
}
```

Winter 2020 Midterm Exam Question 27: Sorting Student IDs

You work at the university registrar, and you have a vector of n student ID numbers that are currently in use at the university. To make assigning new student IDs easier, you always keep your vector of IDs in sorted order. However, a mischievous EECS 281 student hacked into the system last night and shuffled the order of student IDs in your vector!

Fortunately, the hacker was nice enough to tell you that every ID number in the shuffled vector is **at most** d positions away from its correct sorted position, where d is an integer in the range $[1, n)$. Your goal is to implement a function that can restore the sorted vector of student IDs, given the value of d.

Complexity: $O(n \log(d))$ time and $O(d)$ auxiliary space.

Implementation: You may use anything from the STL. Limit: 20 lines of code (points deducted if longer).

Example: Given $d = 2$ and the following altered vector:

```
ids = [2, 1, 3, 5, 7, 4, 6]
```

the `restore_sorted_IDs()` function should restore `ids` so that its elements are in sorted order:

```
ids = [1, 2, 3, 4, 5, 6, 7]
```

Solution 1 (Priority Queue):

```
void restore_sorted_IDs(vector<int> &ids, size_t d) {
  priority_queue<int, vector<int>, greater<int>>
      pq(ids.begin(), ids.begin() + d + 1);
  int curr_idx = 0;
  for (size_t i = d + 1; i < ids.size(); ++i) {
    ids[curr_idx++] = pq.top();
    pq.pop();
    pq.push(ids[i]);
  }
  while (!pq.empty()) {
    ids[curr_idx++] = pq.top();
    pq.pop();
  }
}
```

Solution 2 (Sliding Window):

```
void restore_sorted_IDs(vector<int> &ids, size_t d) {
  auto front = ids.begin();
  auto back = ids.begin() + 2 * d;
  if (d * 2 < ids.size()) {
    for (int i = 0; i < (ids.size() / d) - 1; ++i) {
      sort(front, back);
      front += d;
      back += d;
    }
  }
  sort(front, ids.end());
}
```

Spring 2020 Midterm Exam Question 26: Inverting Parts of a Separated Vector

We define **inverting** a section of a `vector<int>` of size s as swapping the elements at index i and $s - i - 1$ for each integer i such that $0 \leq i < \frac{s}{2}$. Inverting can be done with the provided `invert()` function in $O(s)$ time and $O(1)$ auxiliary space, where s is the distance between `end` and `begin`. You do **NOT** need to write `invert()`.

```
void invert(vector<int>::iterator begin, vector<int>::iterator end);
```

Implement a function `invert_parts()` that, when given a `vector<int>` and a separating integer, inverts the vector chunk by chunk around the separating integer. There are **NO** restrictions on the size of the input vector, the number of times the separator appears, or the location of the separator(s) within the input.

Example:

```
vector<int> vec = {1, 2, 3, 4, 5, 777, 9, 10, 11, 99, 777, 123, 321};
int separator = 777;
invert_parts(vec, separator);
// vec is now {123, 321, 777, 9, 10, 11, 99, 777, 1, 2, 3, 4, 5}
```

Complexity: $O(n)$ time and $O(1)$ auxiliary space, where n is the size of the given vector.

Implementation: You may use anything from the STL. Limit: 25 lines of code (points deducted if longer).

```
void invert_parts(vector<int> &vec, int separator) {
  invert(vec.begin(), vec.end());
  auto prev = vec.begin();
  for (auto curr = vec.begin(); curr != vec.end(); ++curr) {
    if (*curr == separator) {
      invert(prev, curr);
      prev = curr + 1;
    }
  }
  invert(prev, vec.end());
}
```

Fall 2020 Midterm Exam Question 26: Summing Remnant Values

Suppose you were given a `vector<int>` and a number k, and you consider k or more copies of the same value in a row to constitute a "group". You need to sum up any values that are left over after forming groups; these left over values are the remnants. For example, given $k = 3$ and the following vector:

```
{3, 5, 5, 2, 2, 2, 5, 5, 5, 2, 3, 4, 4, 4, 3}
```

Starting from the left, there is one copy of 3 (not a group, since $k = 3$), followed by two 5's (which are also not a group). These are followed by three 2's, which do form a group, and thus are not remnants. This means we effectively have this remaining:

```
{3, 5, 5, 5, 5, 5, 2, 3, 4, 4, 4, 3}
```

Now there are five copies of the value 5, which do form a group. Thus, the portion remaining to be considered is:

```
{3, 2, 3, 4, 4, 4, 3}
```

The cluster of three 4's is a group, leaving:

```
{3, 2, 3, 3}
```

Even though there are k copies of the value 3 remaining, they are not consecutive, and thus do not form a group. Your function should return the sum of these remnants, which is 11.

You may assume that `seq` is not empty, and that $k > 1$.

Example:

```
vector<int> seq = {3, 5, 5, 2, 2, 2, 5, 5, 5, 2, 3, 4, 4, 4, 3};
int k = 3;
cout << sum_remnant_values(seq, k) << endl; // displays 11
```

Complexity: $O(n)$ time and $O(n)$ auxiliary space, where n is the size of the given vector.

Implementation: You may use anything from the STL. Limit: 30 lines of code (points deducted if longer).

```
int sum_remnant_values(const vector<int> &seq, int k) {
  struct Counter {         // a pair<int, int> works as well
    int value, count;
  };

  vector<Counter> c;
  int sum = 0;

  for (int x : seq) {
    sum += x;
    if (c.empty())
      c.push_back({ x, 1 });
    else if (c.back().value == x)
      ++c.back().count;
    else {
      if (c.back().count >= k) {
        sum -= c.back().value * c.back().count;
        c.pop_back();
      } // if
      if (c.empty() || c.back().value != x)
        c.push_back({ x, 1 });
      else
        ++c.back().count;
      } // else
    } // for

    if (c.back().count >= k)
      sum -= c.back().value * c.back().count;

    return sum;
} // sum_remnant_values()
```