CSCI-1200 Data Structures
Test 2 — Practice Problem Solutions

1 Tracking Hotel Occupancy [/ 25]

The `Hotel` class tracks the occupancy of rooms on each floor of a hotel as guests check in and check out. First, let's review the class declaration as written in the header file:

```
class Hotel {
public:
  Hotel();
  // *** PART 1: 'MUST-HAVE' PROTOTYPES ***
  // ACCESSOR
  int num_available_rooms() const;
  // MODIFIERS
  void add_floor(int num_rooms);
  void check_in(int floor, int room);
  void check_out(int floor, int room);
private:
  // REPRESENTATION
  int num_floors;
  int* num_rooms_per_floor;
  bool** occupancy_per_floor;
};
```

And a sample usage of the Hotel class:

```
  Hotel bellvue;
  bellvue.add_floor(5);  bellvue.add_floor(5);  bellvue.add_floor(5);  bellvue.add_floor(2);
  assert (bellvue.num_available_rooms() == 17);
  bellvue.check_in(0,3);  bellvue.check_in(2,3);  bellvue.check_in(3,1);
  assert (bellvue.num_available_rooms() == 14);
  Hotel ambassador(bellvue);
  assert (ambassador.num_available_rooms() == 17);
  ambassador.check_in(0,3);
  assert (ambassador.num_available_rooms() == 16);
```

An empty hotel is created (the foundation is laid), then floors are added to the hotel from the ground up. When the hotel is finished and open for business, guests will "check in" to a specific room on a specific floor. Note that the `check_in` function includes lots of error checking:

```
void Hotel::check_in(int floor, int room) {
  assert (floor >= 0 && floor < num_floors);
  assert (room >= 0 && room < num_rooms_per_floor[floor]);
  assert (occupancy_per_floor[floor][room] == false);
  occupancy_per_floor[floor][room] = true;
}
```

The `check_out` function is similar. The `num_available_rooms` function counts and returns the number of rooms in the building that are not currently occupied.

```
int Hotel::num_available_rooms() const {
  int answer = 0;
  for (int i = 0; i < num_floors; i++) {
    for (int j = 0; j < num_rooms_per_floor[i]; j++) {
      if (occupancy_per_floor[i][j] == false) { answer++; }
    }
  }
  return answer;
}
```

1.1 Dynamically-Allocated Classes “Must-Have” Function Prototypes [/ 5]

First, fill in the marked spot of the class declaration with the prototypes of the 3 key functions that must be defined for all classes with dynamically-allocated memory. If these functions are not implemented by the class designer, the compiler will provide a default version, which is almost certainly unacceptable!

Solution:

```
  Hotel(const Hotel &h);
  const Hotel& operator=(const Hotel &h);
  ~Hotel();
```

1.2 Implementing the Hotel Destructor [/ 8]

Next implement the destructor, as it would appear in the implementation file (`hotel.cpp`). The destructor should make sure the `Hotel` object is demolished in an orderly fashion and no garbage (a.k.a. dynamically-allocated memory) is left behind (a.k.a. a memory leak).

Solution:

```
Hotel::~Hotel() {
  for (int i = 0; i < num_floors; i++) {
    delete [] occupancy_per_floor[i];
  }
  delete [] occupancy_per_floor;
  delete [] num_rooms_per_floor;
}
```

1.3 Implementing the Copy Constructor [/ 12]

Finally implement the copy constructor as it would appear in the `hotel.cpp` file. Study carefully the example usage on the first page of this problem. The structure of the provided hotel is copied including all of the floorplans. Thus, the new hotel has the same capacity as the provided model. However, any hotel guests are *not* copied!. The new hotel begins with zero occupancy (fully available).

Solution:

```
Hotel::Hotel(const Hotel &h) {
  num_floors = h.num_floors;
  num_rooms_per_floor = new int[num_floors];
  occupancy_per_floor = new bool*[num_floors];
  for (int i = 0; i < num_floors; i++) {
    int num_rooms = h.num_rooms_per_floor[i];
    num_rooms_per_floor[i] = num_rooms;
    occupancy_per_floor[i] = new bool[num_rooms];
    for (int j = 0; j < num_rooms; j++) {
      occupancy_per_floor[i][j] = false;
    }
  }
}
```

2 Scrubbing Restaurant Reviews [/ 16]

Write a function `ScrubReviews` that takes in two arguments: `reviews`, an STL list of STL vectors of STL strings, and `word`, an STL string. The words of the review written by each restaurant patron are stored as a vector of strings. Edit `reviews` to completely remove all reviews that contain the targeted `word` (e.g., because a review with the word “`overcooked`” is probably a negative review).

Solution:

```
void ScrubReviews(std::list<std::vector<std::string> > &reviews, const std::string &word) {
  for (std::list<std::vector<std::string> >::iterator itr = reviews.begin();
       itr != reviews.end(); /* itr incremented later */) {
    bool erased = false;
    for (int i = 0; i < itr->size(); i++) {
      if ((*itr)[i] == word) {
        itr = reviews.erase(itr);
        erased = true;
        break;
      }
    }
    if (!erased) itr++;
  }
}
```

3 Global Order Notation [/ 16]

For each programming task described below, choose the letter that best describes the order notation of a straightforward implementation of an algorithm to solve the problem. *Hint: Each letter will be used exactly once.*

A) O(n^2)
B) O(log n)
C) O(1)
D) O(n!)
E) O(n log n)
F) O(n)
G) O($\sqrt{n}$)
H) other

Solution: G For a large city, with a high density population stretching equally far in all directions, with streets laid out in a square grid (equally spaced streets running north-south and east-west), that has a total of n city blocks, what is the average distance between any two blocks in the city?

Solution: C If you have n U.S. dollars, what is the running time of the algorithm to calculate the equivalent amount of Swiss Francs you should receive at the currency exchange?

Solution: B Using a German→English dictionary with n words, translate a short dinner menu to find the vegetarian entrees.

Solution: E Given an English→German dictionary with n English words mapped to the corresponding n German words (alphabetized by English word), create a German→English dictionary with the same English & German words, this time alphabetized by the German words.

Solution: D Given a set of n popular tourist destinations in Switzerland, enumerate all possible orderings or sequences in which the tourists may visit all of these destinations.

Solution: A At a busy European airport with n planes currently flying in close proximity to the airport, find the pair of airplanes that are nearest to each other (to ensure safe separation distances are maintained and avoid collisions).

Solution: H The Center for Disease Control (CDC) is interested in modeling the interactions between the traveling public and the spread of disease. Assuming we start with 1 infected person, and every person takes one plane flight per day, where they sit next to 2 people (middle seat!), how many frequent flyers will be potentially infected after n days?

Solution: F At the end of a busy day of shopping a souvenir shop clerk performs an inventory to confirm that the difference in the stock on the store shelves from opening to closing is equal to the money in the cash register plus credit card receipts.

4 Twirling Links [/27]

In this problem you will write the `twirl` function, which takes in as arguments a pointer to the head of a doubly-linked chain of `Node` objects and an integer value that is found in one of the nodes in the middle of the chain. The `twirl` function first finds the node containing the input value, then swaps the nodes that are before and after that node, and finally removes the node containing the input value.

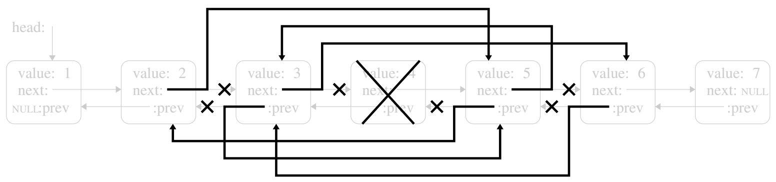
```
class Node {
public:
  int value;
  Node* next;
  Node* prev;
};
```

For example, let's make a chain containing the numbers: 1 2 3 4 5 6 7, in that order, and have the `head` variable point to the `Node` containing the value 1. Now, the function call `twirl(head, 4)` edits the chain so that afterward, the nodes accessed starting at the head are: 1 2 5 3 6 7. Note: this function should edit the pointers in the original `Node`s and should not edit any values or create new `Node`s. This problem does not involve the STL `list` class, or iterators.

4.1 Diagramming Memory [/7]

Below is the state of memory just before calling `twirl(head,4)` . Modify this diagram to show the result of this call. Be as neat as possible in editing the diagram so we can give you full credit for your solution.

Solution:



4.2 Recursive Output [/5]

Write a simple *recursive* function `PrintData` that takes in a single `Node` pointer to the `head` element and outputs the sequence of values to `std::cout`.

Solution:

```
void PrintData(Node *head) {
  if (head == NULL) return;
  std::cout << head->value << " ";
  PrintData(head->next);
}
```

4.3 Implementation [/15]

Now implement the function `twirl` for the *general case*, where the input value is stored in a `Node` in the list and that node is not close to the beginning or end of the linked chain.

When implementing the general case, what specific assumptions did you make about the input? Write a few sentences describing these assumptions and/or insert `assert` statements within the code to explicitly check these assumptions. Your code does not need to work for any of the "special cases" or "corner cases".

Solution:

```
void twirl (Node *head, int value) {
  assert (head != NULL);
  // first, find the node that we want to twirl around
  Node *tmp = head;
  while (tmp != NULL && tmp->value != value) {
    tmp = tmp->next;
  }
  assert (tmp->value == value);
  // just make sure we are aren't near either end of the links!
  // (we must have at least 2 nodes before and 2 nodes after)
  assert (tmp->prev != NULL);
  assert (tmp->prev->prev != NULL);
  assert (tmp->next != NULL);
  assert (tmp->next->next != NULL);
  // set up a couple of temporary pointers
  Node *new_prev = tmp->next;
  Node *new_next = tmp->prev;
  // change the 6 links
  new_prev->prev = new_next->prev;
  new_next->next = new_prev->next;
  new_prev->prev->next = new_prev;
  new_next->next->prev = new_prev;
  new_prev->next = new_next;
  new_next->prev = new_prev;
  // delete the twirled node
  delete tmp;
}
```

5 Gin Rummy Iterators [/36]

In this problem we will revisit the simple `PlayingCard` class from Homework 5. However, instead of using a homemade linked list node structure and pointers, we will use the STL `list` container class and iterators. To get started, we initialize the hand variable and print out the contents:

```
std::list<PlayingCard> hand;
/* initialization code omitted */
Print("hand:",hand);
```

For this example, the contents of the `hand` variable result in this output:

```
hand: 4♡ 5♡ 6♡ 5♣ 8♣ 9♣ 7♣ 4♢ 6♢ 7♢ 5♢ 5♠ 7♠
```

In playing a game of Gin Rummy, we need to search the hand for all *sets* and *runs*. A *set* is 3 or 4 cards that have the same face card value but different suits. A *run* is 3 or more cards that have the same suit, but different face cards, specifically forming an unbroken increasing sequence. For this problem, we will specify that Aces (card value == 1) are low, and can only be part of a run with the low cards (e.g., $A\heartsuit, 2\heartsuit, 3\heartsuit$).

```
std::vector<std::list<PlayingCard> > sets;
FindSets(hand,sets);    assert (sets.size() == 2);
Print("sets[0]:",sets[0]);
Print("sets[1]:",sets[1]);
std::vector<std::list<PlayingCard> > runs;
FindRuns(hand,runs);    assert (runs.size() == 3);
Print("runs[0]:",runs[0]);
Print("runs[1]:",runs[1]);
Print("runs[2]:",runs[2]);
```

The code above will result in the following output:

```
sets[0]: 5♡ 5♣ 5♢ 5♠
sets[1]: 7♣ 7♢ 7♠
runs[0]: 4♡ 5♡ 6♡
runs[1]: 7♣ 8♣ 9♣
runs[2]: 4♢ 5♢ 6♢ 7♢
```

In Gin Rummy, we must form sets and/or runs that use up all of the cards in our hand. However, we may not use the same card in multiple sets or runs. Thus, the function `Remove` pulls the cards in a set or run from the hand. The function returns true if it was successful. If one or more of the cards in the set or run is not in the hand anymore, `Remove` does not edit the hand and returns false.

```
bool success = Remove(hand,sets[0]);  assert (success);
Print("after removing sets[0]: ", hand);
success = Remove(hand,runs[0]);  assert (!success);
success = Remove(hand,runs[1]);  assert (success);
Print("after removing runs[1]: ", hand);
success = Remove(hand,sets[1]);  assert (!success);
success = Remove(hand,runs[2]);  assert (!success);
```

The code above results in the following output (and no assertion failures):

```
after removing sets[0]:  4♡ 6♡ 8♣ 9♣ 7♣ 4♢ 6♢ 7♢ 7♠
after removing runs[1]:  4♡ 6♡ 4♢ 6♢ 7♢ 7♠
```

5.1 Implementing `FindRuns` [/20]

First, implement the `FindRuns` function so that it works as presented on the previous page. You may assume that the hand does not contain any duplicate cards, but you may not assume that the hand is sorted. Furthermore, your code should not rearrange the hand or use any sorting algorithm. The `PlayingCard` function has accessor member functions `getSuit` and `getCard` that return the `std::string` and `int` values of the object. Be sure to use const and reference as appropriate.

Solution:

```
void FindRuns(const std::list<PlayingCard> &hand, std::vector<std::list<PlayingCard> > &sets) {
  // each card in the hand is a candidate for the first card in a sequence
  for (std::list<PlayingCard>::const_iterator itr = hand.begin(); itr != hand.end(); itr++) {
    // first check to see if there is a earlier card to start with
    bool before = false;
    for (std::list<PlayingCard>::const_iterator itr2 = hand.begin(); itr2 != hand.end(); itr2++) {
      if (itr->getSuit() == itr2->getSuit() &&
          itr->getCard() == itr2->getCard()+1) {
        before = true;
      }
    }
    if (before == true) continue;
    // otherwise, start building the sequence
    std::list<PlayingCard> tmp;
    tmp.push_back(*itr);
    while (true) {
      // search for the next card in the sequence
      bool found = false;
      for (std::list<PlayingCard>::const_iterator itr2 = hand.begin(); itr2 != hand.end(); itr2++) {
        if (itr == itr2) continue;
        if (tmp.back().getSuit() == itr2->getSuit() &&
            tmp.back().getCard()+1 == itr2->getCard()) {
          tmp.push_back(*itr2);
          found = true;
        }
      }
      if (!found) break;
    }
    // if the sequence is long enough add it to the set
    if (tmp.size() >= 3 && !before)
      runs.push_back(tmp);
  }
}
```

Assuming the hand contains h cards and the longest run contains r cards, what is the order notation for your solution?

Solution: $O(h*(h+h+...+h)) = O(h*(h*r)) = O(r*h^2)$ **. For each card h in the hand, we search the whole hand h for the next card in the sequence. We do this r times.**

5.2 Implementing `Remove` [/16]

Now, implement the `Remove` function so that it works as presented in the example. Again, you may assume the hand does not contain any duplicate cards, but you may not assume the hand is sorted, nor may you sort the cards. Be sure to use const and reference as appropriate.

Solution:

```
bool Remove(std::list<PlayingCard> &hand, const std::list<PlayingCard> &run_or_set) {
  // first, check to see if every card in the run/set is in the hand
  for (std::list<PlayingCard>::const_iterator itr = run_or_set.begin(); itr != run_or_set.end(); itr++) {
    bool found = false;
    for (std::list<PlayingCard>::const_iterator itr2 = hand.begin(); itr2 != hand.end(); itr2++) {
      if (*itr == *itr2) found = true;
    }
    // return false if one or more is missing
    if (found == false) { return false; }
  }
  // then, repeat the process, but erase the cards one at a time.
  for (std::list<PlayingCard>::const_iterator itr = run_or_set.begin(); itr != run_or_set.end(); itr++) {
    for (std::list<PlayingCard>::iterator itr2 = hand.begin(); itr2 != hand.end(); itr2++) {
      if (*itr == *itr2) {
        hand.erase(itr2);
        break;
      }
    }
  }
  // return true for success
  return true;
}
```

Assuming the hand contains h cards and the set or run contains s cards, what is the order notation for your solution?

Solution: O(s * h). For each element in the set or run s, we must search through all the cards h in the hand to see if it is present. The second nested for loop also has the same running time.

6 Studying the `Vec` Implementation [/34]

In this problem we study the templated `Vec` class that mimics the STL's `vector` container:

```
template <class T> class Vec {
public:
  Vec(unsigned int n, const T& t);
  Vec& operator=(const Vec& v);
  T& operator[] (unsigned int i) { return m_data[i]; }
  /* ADDITIONAL FUNCTIONALITY OMITTED */
private:
  T* m_data;            // Pointer to first location in the allocated array
  unsigned int m_size;  // Number of elements stored in the vector
  unsigned int m_alloc; // Number of array locations allocated,  m_size <= m_alloc
};
```

6.1 Assignment Operator Implementation [/5]

Ben Bitdiddle has just finished implementing the assignment operator. He wrote a rather extensive test suite and all of his test cases pass. The memory debuggers, Dr. Memory and Valgrind, also assure him that there are no uninitialized variables, invalid memory accesses, or memory leaks.

```
template <class T> Vec<T>& Vec<T>::operator=(const Vec<T>& v) {
  delete [] m_data;
  this->m_alloc = v.m_alloc;
  this->m_size = v.m_size;
  this->m_data = new T[this->m_alloc];
  for (unsigned int i = 0; i < this->m_size; ++i) {
    this->m_data[i] = v.m_data[i]; }
  return *this;
}
```

However, Alyssa P. Hacker takes a look at Ben's assignment operator code and finds a subtle but important bug. She writes this additional test case:

```
Vec<string> a(10,"hello");
a = a;
std::cout << a[3] << std::endl;
```

What specifically is the problem with Ben's implementation? Write 2-3 concise and well written sentences.

Solution: Ben's implementation does not check for self-assignment. The correct behavior for self-assignment is to do nothing, but the code as written deletes the structure, and then tries to copy the now unreachable data values into a newly created array. Alyssa's new test case might appear to work on some systems, but when run under a memory debugger, this code will have uninitialized memory and or invalid memory access errors.

6.2 Efficiency of the Assignment Operator [/13]

Furthermore, Alyssa points out that Ben's implementation is unnecessarily expensive when the `m_alloc` member variable in the two `Vec` structures is equal, but `m_size` is very small. Alyssa says Ben's version has runtime $O(a+s)$, where a is the size of the array allocations (`m_alloc`) and s is the number of elements actually stored in the input `Vec` object (`m_size`). Re-implement the assignment operator to fix the bug Alyssa found in the previous section *and* improve the inefficiency described above.

Solution:

```
template <class T> Vec<T>& Vec<T>::operator=(const Vec<T>& v) {
  // check for self-assignment!
  if (this != &v) {
    // only reallocate if the two structures have different sizes
    if (m_alloc != v.m_alloc) {
      delete [] m_data;
      this->m_alloc = v.m_alloc;
      this->m_data = new T[this->m_alloc];
    }
    this->m_size = v.m_size;
    // copy the data
    for (unsigned int i = 0; i < this->m_size; ++i) {
      this->m_data[i] = v.m_data[i]; }
  }
  return *this;
}
```

What is the order notation of the new version of the assignment operator when the `m_alloc` member variables in the two `Vec` structures are equal?

Solution: $O(s)$**, because we only have to copy the valid elements.**

6.3 push_front and pop_front for Vec [/16]

Finally, Alyssa wants to add efficient `push_front`/`pop_front` functionality to the `Vec` class. Her idea is to leave extra space in the front of the vector just like we leave extra space at the back of the vector for `push_back`. If there isn't any room at the front of the vector, the allocated array is doubled and the data is copied, leaving the new space at the front, so that a significant number of future `push_front` calls can be executed with O(1) running time.

Here's an example of how she would like the function to work:

```
  Vec<double> a(10, 3.14);
  a.push_front(2.68);
  a.push_front(0.95);
  a.push_front(10.3);
  assert (a.size() == 13);    assert (a[0] == 10.3);      assert (a[1] == 0.95);
  assert (a[2] == 2.68);      assert (a[3] == 3.14);      assert (a[12] == 3.14);
```

In Alyssa's modified representation, she adds one new variable, `m_first`:

```
  T* m_data;            // Pointer to first location in the allocated array
  unsigned int m_size;  // Number of elements stored in the vector
  unsigned int m_alloc; // Number of array locations allocated,  m_size <= m_alloc
  unsigned int m_first; // Index of the first element in the internal array
```

First, re-implement the subscripting operator:

Solution:

```
template <class T>
T& Vec<T>::operator[] (unsigned int i) {
  return m_data[m_first+i];
}
```

And then implement the new `push_front` function:

Solution:

```
template <class T>
void Vec<T>::push_front(const T& val) {
  // if it's the first element, just use push_back
  if (m_alloc == 0) { push_back(val); return; }
  assert (m_alloc > 0);
  if (m_first == 0) {
    // Calculate the new allocation.  Make sure it is at least one.
    m_alloc *= 2;
    assert (m_alloc > 1);
    // Allocate the new array
    T* new_data = new T[ m_alloc ];
    // put the existing data in the back half of the array
    m_first = m_alloc / 2;
    // copy the data
    for (unsigned int i=0; i<m_size; ++i) {
      new_data[m_first+i] = m_data[i]; }
    // delete the old array and reset the pointers
    delete [] m_data;
    m_data = new_data;
  }
  // move the first index back one spot
  m_first--;
  // Add the value at the last location and increment the bound
  m_data[m_first] = val;
  ++ m_size;
}
```

7 Sorted Splicing [/28]

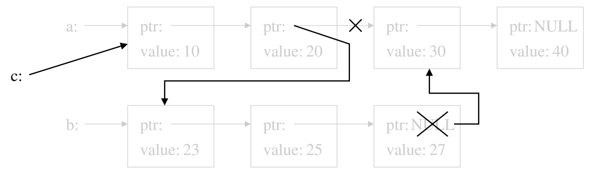
In this problem you will write a `sorted_splice` function that takes in as arguments two pointers to chains of linked `Node`s named *a* and *b* and splices the second chain, *b*, into the right spot in chain *a* so that all the elements are now in proper sorted order. You should assume the input chains *a* and *b* are sorted, there are no duplicate elements in the input chains, no number appears in both *a* and *b*, and no element of *a* belongs between two elements of *b* when properly sorted. The function returns a pointer to the start of the complete chain. Note: This function should edit the `Node`s in the original chains. This problem does not involve the `new` command, the STL `list` class, or iterators.

```
class Node {
public:
  Node* ptr;
  int value;
};
```

7.1 Diagramming Memory [/5]

Below is the state of memory just before calling `c = sorted_splice(a,b)` . Modify this diagram to show the result of this call (all 7 nodes should be accessible from pointer `c`). Be as neat as possible in editing the diagram so we can give you full credit for your solution.

Solution:



7.2 Other Test Cases [/5]

What other *valid* test cases should we try to ensure that our program logic is fully debugged? Just write the numbers (you do not need to draw memory diagrams).

Solution:

a:	1 2 3	b:	4 5 6	output c:	1 2 3 4 5 6
a:	4 5 6	b:	1 2 3	output c:	1 2 3 4 5 6
a:	1 2 3	b:		output c:	1 2 3
a:		b:	4 5 6	output c:	4 5 6

7.3 Implementation [/15]

Now implement the function `sorted_splice`. Think carefully about the corner cases you created on the previous page.

Solution:

```
Node* sorted_splice(Node* a, Node*b) {
  if (a == NULL) return b;
  if (b == NULL) return a;
  Node* answer;
  Node* tmp;
  Node* rest;
  if (b->value < a->value) {
    // handle the case when all of list b comes before the first element of list b
    answer = b;
    tmp = b;
    rest = a;
  } else {
    answer = a;
    // else, use tmp to walk down a until we find the node before the splice
    tmp = a;
    while (tmp->ptr != NULL && tmp->ptr->value < b->value) {
      tmp = tmp->ptr;
    }
    rest = tmp->ptr;
    tmp->ptr = b;
  }
  // walk through the b list until the last node
  while (tmp->ptr != NULL) {
    tmp = tmp->ptr;
  }
  // attach the rest of the first list
  tmp->ptr = rest;
  return answer;
}
```

7.4 Order Notation [/3]

What is the running time of your solution if chain `a` contains n elements and chain `b` contains m elements?

Solution: O(n+m)

8 Order Notation

Match the order notation with each fragment of code.

A) O(n)	B) O(1)	C) O(n^n)	D) O(n^2)
E) O(2^n)	F) O($\log n$)	G) O($n \log n$)	H) O($\sqrt{n}$)

Solution: D

```
vector<int> my_vector;
// my_vector is initialized with n entries
// do not include initialization in performance analysis
for (int i = 0; i < n; i++) {
  my_vector.erase(my_vector.begin());
}
```

Solution: E

```
int foo(int n) {
  if (n == 1 || n == 0) return 1;
  return foo(n-1) + foo(n-2);
}
```

Solution: G

```
int k = 0;
for (int i = 0; i < n; i++) {
  for (int j = 0; j < log(n); j++) {
    k += i*j;
  }
}
```

Solution: B

```
float* my_array = new float[n];
// do not include memory allocation in performance analysis
my_array[n/2] = sqrt(n);
```