1 Complexity

1. (*10 Points*) State the precise mathematical definitions for $O(g(n))$, $\Omega(g(n))$, and $\Theta(g(n))$.

   Solution:

   $f(n)$ is $O(g(n))$ if there exist positive numbers c and N such that $0 \leq f(n) \leq cg(n)$ for all $n \geq N$.

   $f(n)$ is $\Omega(g(n))$ if there exist positive numbers c and N such that $f(n) \geq cg(n) \geq 0$ for all $n \geq N$.

   $f(n)$ is $\Theta(g(n))$ if there exist positive numbers c_1, c_2, and N such that $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$ for all $n \geq N$.

2. (*10 Points*) Using the definition of $O(g(n))$, prove that $2n^3 - n^2 + 4n + 7$ is $O(n^3)$.

   Solution: Need to find constants $c, n_0 > 0$ such that $2n^3 - n^2 + 4n + 7 \leq n^3$ for all $n \geq n_0$. In other words, $2 - \frac{1}{n} + 4\frac{1}{n^2} + 7\frac{1}{n^3} \leq c$. Since $\frac{1}{n}$ is decreasing we can choose $c = 14$ and $n_0 = 1$. Then we have $2 - \frac{1}{n} + 4\frac{1}{n^2} + 7\frac{1}{n^3} \leq 2 + \frac{1}{n} + 4\frac{1}{n^2} + 7\frac{1}{n^3} \leq 14 = c$.

3. (*10 Points*) Using the definition of $O(g(n))$, prove that 2^n is $O(n!)$.

   Solution: Need to find constants $c, n_0 > 0$ such that $2^n \leq cn!$ for all $n \geq n_0$.

   $$2^n = 2 \cdot 2 \cdot 2 \cdot 2 \cdots 2 \leq 2 \cdot 2 \cdot 3 \cdot 4 \cdots n = 2 \cdot 1 \cdot 2 \cdot 3 \cdot 4 \cdots n = 2n!$$

   So we can let $c = 2$ and $n_0 = 1$.

4. (*20 Points*) Complexity Analysis with the Master Theorem:

Theorem 4.1 (Master theorem)
Let $a > 0$ and $b > 1$ be constants, and let $f(n)$ be a driving function that is defined and nonnegative on all sufficiently large reals. Define the recurrence $T(n)$ on $n \in \mathbb{N}$ by

$$T(n) = aT(n/b) + f(n)\,,$$

where $aT(n/b)$ actually means $a'T(\lfloor n/b \rfloor) + a''T(\lceil n/b \rceil)$ for some constants $a' \geq 0$ and $a'' \geq 0$ satisfying $a = a' + a''$. Then the asymptotic behavior of $T(n)$ can be characterized as follows:

1. If there exists a constant $\epsilon > 0$ such that $f(n) = O(n^{\log_b a - \epsilon})$, then $T(n) = \Theta(n^{\log_b a})$.
2. If there exists a constant $k \geq 0$ such that $f(n) = \Theta(n^{\log_b a} \lg^k n)$, then $T(n) = \Theta(n^{\log_b a} \lg^{k+1} n)$.
3. If there exists a constant $\epsilon > 0$ such that $f(n) = \Omega(n^{\log_b a + \epsilon})$, and if $f(n)$ additionally satisfies the ***regularity condition*** $af(n/b) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n, then $T(n) = \Theta(f(n))$. ∎

(a) Use the Master Theorem to find the complexity of a function that takes an array of size n and does two recursive calls. The first recursive call uses the first quarter of the array and the second call uses the last quarter of the array. The other work done in the function is a single loop that multiplies every third entry by 2. Simplify all logarithms.

Solution: In this case $a = 2$, $b = 4$, and $f(n) = n/3$.

- Is there an $\epsilon > 0$ with $n/3 = O(n^{\log_4(2) - \epsilon}) = O(n^{1/2 - \epsilon})$? No, since $1/2 - \epsilon < 1$.
- Is there a $k \geq 0$ with $n/3 = \Theta(n^{\log_4(2)} \lg^k(n)) = \Theta(n^{1/2} \lg^k(n))$? No, any positive power of n will overtake any power of a logarithm.
- Is there an $\epsilon > 0$ with $n/3 = \Omega(n^{\log_4(2) + \epsilon}) = \Omega(n^{1/2 + \epsilon})$? Yes, let $\epsilon = 1/4$. Also, $af(n/b) = 2((n/4)/3) = n/6$, so if we let $c = 0.9 < 1$, we have $cf(n) = 0.9 \cdot n/3 = 0.3n > n/6$. So the regularity condition is satisfied and the complexity of this function is $\Theta(n)$.

(b) For the merge sort, what are the values of a, b, and $f(n)$? Use these and the Master Theorem to derive the complexity of the merge sort. Simplify all logarithms.

Solution: For the merge sort, $a = 2$, $b = 2$, and $f(n) = n$.

- Is there an $\epsilon > 0$ with $n = O(n^{\log_2(2) - \epsilon}) = O(n^{1 - \epsilon})$? No, since $1 - \epsilon < 1$.
- Is there a $k \geq 0$ with $n = \Theta(n^{\log_2(2)} \lg^k(n)) = \Theta(n \lg^k(n))$? Yes, let $k = 0$. Hence the complexity is $\Theta(n^{\log_2(2)} \lg^{k+1}(n)) = \Theta(n \lg(n))$.

5. **Short Answer:** (*25 Points*)

   (a) What are the criteria for a Red-Black tree?
   Solution:
      i. Every node is either red or black.
      ii. The root is black.
      iii. Every leaf (NIL) is black.
      iv. If a node is red, then both its children are black.
      v. For each node, all simple paths from the node to descendant leaves contain the same number of black nodes.

   (b) What are the criteria for an AVL tree?
   Solution: An AVL tree (originally called an admissible tree) is one in which the height of the left and right subtrees of every node differ by at most one.

   (c) What are the height bounds for a Red-Black tree?
   Solution: $\lg(n+1) \leq h \leq 2\lg(n+1)$

   (d) What are the height bounds for an AVL tree, assuming all nodes have counts of 1?
   Solution: $\lg(n+1) \leq h \leq 1.44\lg(n+2) - 0.328$

   (e) What is the complexity of node insertion into a Red-Black tree?
   Solution: $O(\lg(n))$

   (f) What is the complexity of the deletion of a node from a Red-Black tree?
   Solution: $O(\lg(n))$

   (g) What is the complexity of the depth-first search/traversal?
   Solution: $O(|V| + |E|)$

   (h) What is the complexity of the breadth-first search/traversal?
   Solution: $O(|V| + |E|)$

   (i) What is the complexity of Dijkstra's algorithm for finding the shortest path from one vertex to all the other vertices in a graph?
   Solution: $O(|V|^2)$

   (j) What is the complexity of Ford's algorithm for finding the shortest path from one vertex to all the other vertices in a graph?
   Solution: $O(|V||E|)$

   (k) What is the complexity of Kruskal's algorithm for finding a minimal spanning tree for a graph?
   Solution: $O(|E|\lg(|V|))$

   (l) What is the complexity of Dijkstra's algorithm for finding a minimal spanning tree for a graph?
   Solution: $O(|E||V|)$

   (m) What is the complexity of the Ford-Fulkerson algorithm for finding the maximum flow through a network?
   Solution: $O(|V||E|^2)$

   (n) What is the complexity of the Quick Sort in the best and worst cases?
   Solution: Best is $O(n\lg(n))$ and worst is $O(n^2)$.

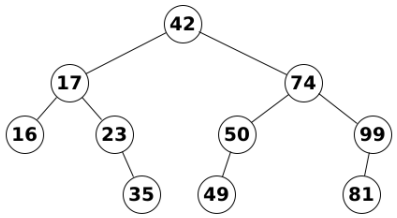
   (o) What is the complexity of the Heap Sort in the best and worst cases?
   Solution: Both are $O(n\lg(n))$

   (p) What is the complexity of the Merge Sort in the best and worst cases?
   Solution: Both are $O(n\lg(n))$

   (q) Which sorts, that we discussed in class and lab, run in linear time?
   Solution: Count, Radix, and Bucket.

2 Algorithms

1. (*20 Points*) The following exercises deal with the following AVL tree.



   (a) Starting with the original tree, draw the AVL tree after 48 has been inserted.
   Solution:

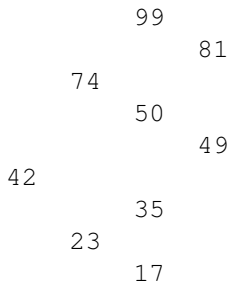
```
          99
              81
     74
          50
              49
          48
42
              35
          23
     17
          16
```

   (b) Starting with the original tree, draw the AVL tree after 33 has been inserted.
   Solution:

```
          99
              81
     74
          50
              49
42
              35
          33
              23
     17
          16
```

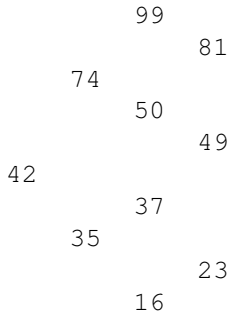
(c) Starting with the original tree, draw the AVL tree after 16 has been removed.

Solution:

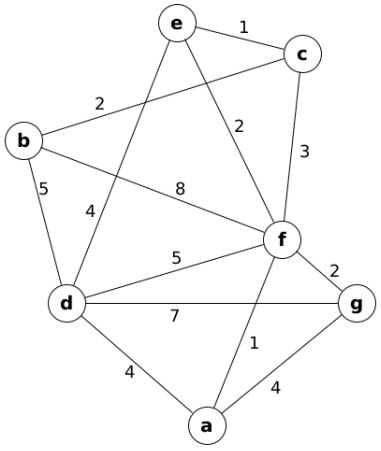


(d) Starting with the original tree, draw the AVL tree after the insertion of 37 and the deletion of 17, in that order.

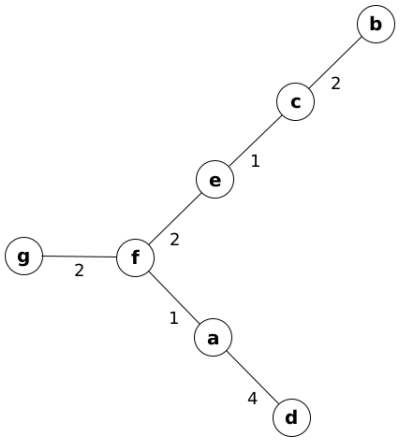
Solution:



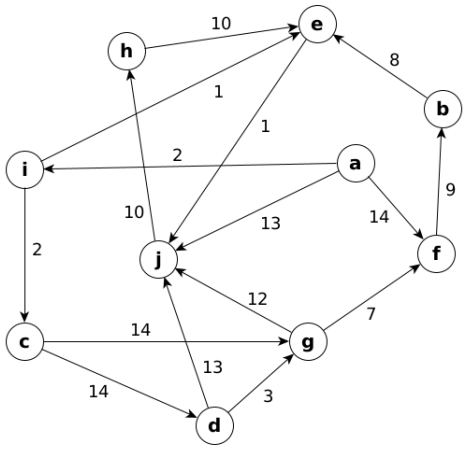
2. (*15 Points*) Given the following weighted graph, use Kruskal's algorithm to find the minimum spanning tree of the graph. Display the final resulting minimum spanning tree. Edges with the same weights are to be processed alphabetically.



Solution:



3. (*15 Points*) Given the following directed weighted graph, use Dijkstra's algorithm to find the shortest path to all vertices from the starting vertex a . Display each iteration, active vertex, and weight label in chart form as was done in the text and in class.



Solution:

```
a 0
b 23
c 4
d 18
e 3
f 14
g 18
h 14
i 2
j 4
```

4. (*5 Points*) The following array will be used as a hash table with open addressing and a linear probe. The hash function will be $h(x) = x \ \% \ sz$ where sz is the size of the array (10 in this case).

   Insert the following values into the hash table in this order: 37, 42, 22, 51, 73, 66, 25, 11.

   Solution:

51
42
22
73
25
66
37
11

5. (*5 Points*) The following array will be used as a hash table with open addressing and a quadratic probe. The hash function will be $h(x) = x \ \% \ sz$ where sz is the size of the array (10 in this case).

   Insert the following values into the hash table in this order: 33, 74, 23, 17, 82, 53, 99, 10.

   Solution:

99
82
23
33
74
10
17
53

6. (*5 Points*) The following array will be used as a hash table with open addressing and double hashing for the probe. The hash function will be $h(x) = x \ \% \ sz$ where sz is the size of the array (10 in this case). The probe hash function is $h_p(x) = (x/100) \ \% \ sz$ where sz is the size of the array (10 in this case).

   Insert the following values into the hash table in this order: 115, 273, 516, 908, 418, 312, 798.

   Solution:

312
418
273
115
516
908
798

3 Coding

1. (*15 Points*) Write either the Merge Sort or the Quick Sort for a templated array, do only one of these.

   Solution:

```
template <class T>
void merge(T A[], T Temp[], int startA, int startB, int end) {
  int aptr = startA, bptr = startB, i = startA;

  while (aptr < startB && bptr <= end)
    if (A[aptr] < A[bptr])
      Temp[i++] = A[aptr++];
    else
      Temp[i++] = A[bptr++];

  while (aptr < startB)
    Temp[i++] = A[aptr++];

  while (bptr <= end)
    Temp[i++] = A[bptr++];

  for (i = startA; i <= end; i++)
    A[i] = Temp[i];
}

template <class T> void mergeSort(T A[], T Temp[], int start, int end) {
  if (start < end) {
    int mid = (start + end) / 2;
    mergeSort(A, Temp, start, mid);
    mergeSort(A, Temp, mid + 1, end);
    merge(A, Temp, start, mid + 1, end);
  }
}

template <class T> void mergeSort(T A[], int size) {
  T *Temp = new T[size];
  mergeSort(A, Temp, 0, size - 1);
  delete[] Temp;
}

////////////////////////////////////////////////////////////////

template <class T> void quickSort(T A[], int left, int right) {
  int i = left, j = right, mid = (left + right) / 2;
  T pivot = A[mid];

  while (i <= j) {
    while (A[i] < pivot)
      i++;

    while (A[j] > pivot)
      j--;

    if (i <= j) {
      T tmp = A[i];
      A[i] = A[j];
      A[j] = tmp; i++; j--;
    }
  }

  if (left < j)
    quickSort(A, left, j);
  if (i < right)
    quickSort(A, i, right);
}

template <class T> void quickSort(T A[], int size) {
  quickSort(A, 0, size - 1);
}
```

2. (*15 Points*) Write the insert and delete by merge functions for a templated (unbalanced) binary search tree. The delete by merge should merge the subtree from the immediate successor of the deleted node. The following is the class specification you will be working in. Create implementations for all the functions except for the `destroySubTree` function.

   Solution:

```
template <class T>
void BinaryTree<T>::insert(TreeNode *&nodePtr, TreeNode *&newNode) {
  if (nodePtr == nullptr)
    nodePtr = newNode;
  else if (newNode->value < nodePtr->value)
    insert(nodePtr->left, newNode);
  else
    insert(nodePtr->right, newNode);
}

template <class T> void BinaryTree<T>::insertNode(T item) {
  TreeNode *newNode = nullptr;
  newNode = new TreeNode;
  newNode->value = item;
  newNode->left = newNode->right = nullptr;
  insert(root, newNode);
}

//////////////////////////////

template <class T> void BinaryTree<T>::remove(T item) {
  deleteNode(item, root);
}

template <class T> void BinaryTree<T>::deleteNode(T item, TreeNode *&nodePtr) {
  if (item < nodePtr->value)
    deleteNode(item, nodePtr->left);
  else if (item > nodePtr->value)
    deleteNode(item, nodePtr->right);
  else
    makeDeletion(nodePtr);
}

template <class T> void BinaryTree<T>::makeDeletion(TreeNode *&nodePtr) {
  TreeNode *tempNodePtr = nullptr;

  if (nodePtr == nullptr)
    cout << "Cannot delete empty node.\n";
  else if (nodePtr->right == nullptr) {
    tempNodePtr = nodePtr;
    nodePtr = nodePtr->left;
    delete tempNodePtr;
  } else if (nodePtr->left == nullptr) {
    tempNodePtr = nodePtr;
    nodePtr = nodePtr->right;
    delete tempNodePtr;
  }
  else {
    tempNodePtr = nodePtr->right;
    while (tempNodePtr->left)
      tempNodePtr = tempNodePtr->left;
    tempNodePtr->left = nodePtr->left;
    tempNodePtr = nodePtr;
    nodePtr = nodePtr->right;
    delete tempNodePtr;
  }
}
```

3. (*15 Points*) Write the three binary tree traversal functions, inorder, preorder, and postorder. These are to all to include a function pointer parameter that will apply the function to each element of the tree. These traversal functions should fit into the same framework as the given specification in the previous exercise.

   Also write a call to to these functions, as if in the main, that will print the value in the node to the console screen, this parameter function pointer should be done as a lambda expression.

   Solution:

```
protected:
  void InOrderRec(Node *, void (*fct)(T &)) const;
  void PreOrderRec(Node *, void (*fct)(T &)) const;
  void PostOrderRec(Node *, void (*fct)(T &)) const;

public:
  void InOrder(void (*fct)(T &)) const { InOrderRec(root, fct); }
  void PreOrder(void (*fct)(T &)) const { PreOrderRec(root, fct); }
  void PostOrder(void (*fct)(T &)) const { PostOrderRec(root, fct); }


template <class T>
void BinaryTree<T>::InOrderRec(TreeNode *nodePtr, void (*fct)(T &)) const {
  if (nodePtr) {
    InOrderRec(nodePtr->left, fct);
    fct(nodePtr->value);
    InOrderRec(nodePtr->right, fct);
  }
}

template <class T>
void BinaryTree<T>::PreOrderRec(TreeNode *nodePtr, void (*fct)(T &)) const {
  if (nodePtr) {
    fct(nodePtr->value);
    PreOrderRec(nodePtr->left, fct);
    PreOrderRec(nodePtr->right, fct);
  }
}

template <class T>
void BinaryTree<T>::PostOrderRec(TreeNode *nodePtr, void (*fct)(T &)) const {
  if (nodePtr) {
    PostOrderRec(nodePtr->left, fct);
    PostOrderRec(nodePtr->right, fct);
    fct(nodePtr->value);
  }
}

//////////////////////////////////////

tree.PreOrder([](int &a) { cout << a << " "; });
tree.InOrder([](int &a) { cout << a << " "; });
tree.PostOrder([](int &a) { cout << a << " "; });
```

4. (*15 Points*) Write either the depth-first or breadth-first search for the templated graph, do only one of these. The functions should be written as if in the main program. Below is the specification to the Graph class that these functions are to work with. You may use the the functionality of this class, any structure from the STL and any functions from the algorithm library. Outside of that, you need to write the full code. The output can simply be a printout of the list of edges to the console screen.

   Solution:

```
template <class T> void depthFirstSearch(Graph<T> &G) {
  vector<T> vlist = G.getVertexList();
  vector<int> num(vlist.size());
  vector<pair<T, T>> Edges;
  int count = 1;

  while (find(num.begin(), num.end(), 0) < num.end()) {
    int pos = find(num.begin(), num.end(), 0) - num.begin();
    DFS(G, num, vlist, pos, count, Edges);
  }
  for (pair<T, T> e : Edges)
    cout << e.first << " - " << e.second << endl;
}

template <class T>
void DFS(Graph<T> &G, vector<int> &num, vector<T> &vlist, int pos, int &count,
         vector<pair<T, T>> &Edges) {
  vector<T> Adj = G.getAdjacentList(vlist[pos]);
  num[pos] = count++;

  for (size_t i = 0; i < Adj.size(); i++) {
    T vert = Adj[i];
    size_t vPos = find(vlist.begin(), vlist.end(), vert) - vlist.begin();
    if (vPos < vlist.size() && num[vPos] == 0) {
      Edges.push_back({vlist[pos], vert});
      DFS(G, num, vlist, vPos, count, Edges);
    }
  }
}

//////////////////////////////////////////

template <class T> void breadthFirstSearch(Graph<T> &G) {
  vector<T> vlist = G.getVertexList();
  vector<int> num(vlist.size());
  vector<pair<T, T>> Edges;
  int count = 1;
  deque<T> queue;

  while (find(num.begin(), num.end(), 0) < num.end()) {
    size_t pos = find(num.begin(), num.end(), 0) - num.begin();
    num[pos] = count++;
    queue.push_back(vlist[pos]);
    while (!queue.empty()) {
      T vert = queue.front();
      queue.pop_front();
      vector<T> Adj = G.getAdjacentList(vert);
      for (size_t i = 0; i < Adj.size(); i++) {
        size_t AdjvPos =
            find(vlist.begin(), vlist.end(), Adj[i]) - vlist.begin();
        if (num[AdjvPos] == 0) {
          num[AdjvPos] = count++;
          queue.push_back(Adj[i]);
          Edges.push_back({vert, Adj[i]});
        }
      }
    }
  }
  for (pair<T, T> e : Edges)
    cout << e.first << " - " << e.second << endl;
}
```