Chris Gregg

Section Handout #9 Solutions

If you have any questions about the solutions to the problems in this handout, feel free to reach out to your section leader, Aaron, or Chris for more information.

1. Dijkstra and A*

Dijkstra

```
A to B: {A, B}, cost = 4
A to C: {A, C}, cost = 5
A to D: {A, C, G, D}, cost = 9
A to E: {A, E}, cost = 1
A to F: {A, B, F}, cost = 6
A to G: {A, C, G}, cost = 8
```

A*

```
A to G: {A, C, G}, cost = 8
```

2. Good Burger

To conserve space, new lines are indicated using a "/" between lines.

```
/* a) */
L1 / L2 / B1

/* b) */
L2

/* c) */
COMPILER ERROR

/* d) */
L1 / H2 / L2 / B1

/* e) */
H2 / L2

/* f) */
M3 / L1 / H2 / L2 / B1

/* g) */
COMPILER ERROR

/* h) */
L1 / H2 / L2 / B1

/* i) */
H2 / L2

/* j) */
H2 / L2

/* k) */
B3

/* l) */
COMPILER ERROR

/* m) */
L1 / L2 / B1
(cast doesn't change behavior)

/* n) */
B3
(cast makes it compile)

/* o) */
CRASH
(cast too far down)

/* p) */
COMPILER ERROR

/* q) */
M4

/* r) */
M4
```

3. It's Time To Meet the Muppets

Output for `Waldorf *`

```
Kermit::fozzie
Kermit::rowlf
Statler::misspiggy
Statler:rowlf
Waldorf::animal
Waldorf:rowlf
```

Output for `Gonzo *`

```
Kermit::fozzie
Kermit::rowlf
Gonzo::misspiggy
Kermit::beaker
Gonzo::animal
Gonzo::rowlf
```

4. Append (Linked Lists)

```
void append(ListNode *&list1, ListNode *&list2) {
  if (list1 == nullptr) {
    list1 = list2;
  } else {
    ListNode *current = list1;
    while (current->next != nullptr) {
      current = current->next;
    }
    current->next = list2;
  }
  list2 = nullptr;
}
```

5. Transferring Evens (Linked Lists)

```
ListNode *transferEvens(ListNode *&list1) {
  ListNode *list2 = nullptr;
  if (list1 != nullptr) {
    list2 = list1;
    list1 = list1->next;
    ListNode *current = list1;
    ListNode* list2Last = list2;
    while (current != nullptr && current->next != nullptr) {
      list2Last->next = current->next;
      list2Last = current->next;
      current->next = current->next->next;
      current = current->next;
    }
    list2Last->next = nullptr;
  }
  return list2;
}
```

6. Hacking and Cracking (Recursive Backtracking)

```
string findPassword(string soFar, int maxLength) {
    if (login(soFar)) return soFar;
    if (soFar.size() == maxLength) return "";

    for (char c = 'a'; c <= 'z'; c++) {
        if (findPassword(soFar + c, maxLength) != "") {
            return password;
        }
    }

    for (char c= 'A'; c <= 'Z'; c++) {
        if (findPassword(soFar + c, maxLength) != "") {
            return password;
        }
    }
    return "";
}
```

```
string crack(int maxLength) {
    if (maxLength < 0) {
        throw maxLength;
    } else if (maxLength == 0) {
        return "";
    }

    return findPassword("", maxLength);
}
```

7. ReCuReNCe (Recursive Backtracking)

```
bool isElementSpellable(string word, Lexicon &symbols) {
  if (word.length() == 0) return true;

  for (string symbol : symbols) {
    if (startsWith(word, symbol) &&
      isElementSpellable(word.substr(symbol.length()), symbols)) {
      return true;
    }
  }
  return false;
}
```

8. Limit Leaves (Trees)

```
void limitLeaves(TreeNode *&node, int n) {
  if (node == nullptr) return;

  limitLeaves(node->left, n);
  limitLeaves(node->right, n);

  if (node->left == nullptr && node->right == nullptr) {
    if (node->data <= n) {
      delete node;
      node = nullptr;
    }
  }
}
```

9. Child Swap (Trees)

```
void swapChildrenAtLevelHelper(BinaryTreeNode *root, int k) {
  if (root == nullptr) return;
  if (k == 1) {
    BinaryTreeNode *temp = root->left;
    root->left = root->right;
    root->right = temp;
    return;
  }

  swapChildrenAtLevelHelper(root->left, k - 1);
  swapChildrenAtLevelHelper(root->right, k - 1);
}
```

```
void swapChildrenAtLevel(int k) {
    if (k <= 0) throw k;
    swapChildrenAtLevelHelper(root, k);
}
```