CS106B
Spring 2017

Chris Gregg

Section Handout #4 Solutions

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

1. Insertion Sort

after pass 1	`{17, 29,  3, 94, 46,  8, -4, 12}`
after pass 2	`{ 3, 17, 29, 94, 46,  8, -4, 12}`
after pass 3	`{ 3, 17, 29, 94, 46,  8, -4, 12}`

2. Merge Sort

after split 1	`{29, 17, 3, 94} {46, 8, -4, 12}`
after split 2	`{29, 17} {3, 94} {46, 8} {-4, 12}`
after split 3	`{29} {17} {3} {94} {46} {8} {-4} {12}`
after merge 1	`{17, 29} {3, 94} {8, 46} {-4, 12}`
after merge 2	`{3, 17, 29, 94} {-4, 8, 12, 46}`
after merge 3	`{-4, 3, 8, 12, 17, 29, 46, 94}`

3. It Was The Best of Cases, It Was The Worst of Cases
Quicksort performs worst when the vector is already sorted (or in reverse sorted order). But why is that? More generally, quicksort performs the worst when *the pivot is the largest or smallest value in the vector*. This is because quicksort works by dividing the problem into smaller pieces (elements less than the pivot, and elements greater than the pivot). If the pivot is already the largest or smallest element, then the problem isn't smaller. A sorted or reverse sorted vector guarantees that the pivot chosen will always be the smallest or largest element in the vector.

On the other hand, quicksort performs best when *the pivot is the median value in the vector*. Intuitively, this is because when the pivot is the median element, it evenly splits the remaining elements for the two recursive calls.

Note that, as we discussed in lecture, the best and worst case performance can very depending on the algorithm you choose to select the pivot. The code we showed always picked the first element in the Vector, but we also suggested that you pick a random element as the pivot. Even if you randomize the pivot, there's no way to avoid the worst case behavior, since you might still randomly pick the elements in sorted (or reverse sorted) order.

4. Reciprocate and Divide

```
void Fraction::reciprocal() {
  int tempDenom = denom;
  denom = num;
  num = tempDenom;
}

void Fraction::divide(Fraction other) {
  mult(Fraction(other.getDenom(), other.getNum()));
}
```

5. First Date (Class)

```
// .h file
class Date {
public:
  Date(int m, int d);

  int getDay();
  int getMonth();
  int daysInMonth();
  void nextDay();
private:
  int m;
  int d;
};
```

```
// .cpp file
Date::Date(int month, int day) {
  m = month;
  d = day;
}

int Date::getDay() {
  return d;
}

int Date::getMonth() {
  return m;
}

int Date::daysInMonth() {
  switch (m) {
    case 2:
      return 28;
    case 1:
    case 3:
    case 5:
    case 7:
    case 8:
    case 10:
    case 12:
      return 31;
    default:
      return 30;
  }
}

void Date::nextDay() {
  d += 1;
  if (d > daysInMonth()) {
    m = ((m + 1) % 12);
    d = 1;
  }
}
```

6. Circle (Class) of Life

```
// .h file
class Circle {
public:
  Circle(double radius);

  double area();
  double circumference();
  double getRadius();
  string toString();
private:
  double r;
};
```

```
// .cpp file
Circle::Circle(double radius) {
  r = radius;
}

double Circle::area() {
  return PI * r * r;
}

double Circle::circumference() {
  return 2 * PI * r;
}

doube Circle::getRadius() {
  return r;
}

string Circle::toString() {
  return string("Circle{radius=") + realToString(r) +
    string("}");
}
```

7. Align DNA Strands

```
int alignStrands(string &one, string &two, Map<string, int> &cache) {
  if (one.empty()) return -2 * two.length();
  if (two.empty()) return -2 * one.length();

  string key = one + ":" + two;
  if (cache.containsKey(key)) return cache[key];

  if (one[0] == two[0]) { // two leading bases match
    int score = alignStrands(one.substr(1), two.substr(1), cache) + 1;
    cache[key] = score;
    return cache[key];
  }

  int first = alignStrands(one, two.substr(1), cache) - 2; // insert the space in one
  int second = alignStrands(one.substr(1), two, cache) - 2; // insert the space in two
  int third = alignStrands(one.substr(1), two.substr(1), cache) - 1; // don't insert space
  cache[key] = max(first, max(second, third));
  return cache[key];
}

int alignStrands(const string& one, const string& two) {
  Map<string, int> cache;
  return alignStrands(one, two, cache);
}
```

8. Sorting in O(n) Time

```
void linearSort(Vector<int> &vec, int k) {
  Vector<int> sorted(k + 1, 0); // Vector with 0-k (inclusive) elements, initialized to 0

  // create a histogram of the values in vec
  for (int i = 0; i < vec.size(); i++) {
    int elem = vec[i];
    sorted[elem] += 1;
  }

  // transform the histogram back into the individual values
  int counter = 0; // index at which to insert next element
  for (int i = 0; i < sorted.size(); i++) {
    for (int j = 0; j < sorted[i]; j++) {
      vec[counter++] = i;
    }
  }
}
```

This is a variation of a sorting algorithm called counting sort. Unlike many of the algorithms we looked at in class, it doesn't require comparing two or more elements, which is why it runs faster than O(n log n) – you'll learn more about this if you go on to study algorithms. Why might you not want to use counting sort? Think of circumstances where the limitations of counting sort would prevent its use.