Algorithmic Complexity of Summation Codes

Muneeb Khan

November 26, 2016

Sum1

```
1  double sum1(std::vector<double>& v)
2  {
3      if (v.empty()) {
4          return 0.0;
5      }
6      for(size_t i = 0; i < v.size() - 1; ++i) {
7          std::sort(v.begin()+i, v.end());
8          v[i+1] += v[i];
9      }
10     return v.back();
11 }
```

The big-O complexity of the *sort* function (used at line 7 in the code) is O($n \cdot logn$), where n is the number of elements sorted in the vector v. Note that the number of elements in vector v that are sorted shrinks as the number of iterations (of loop at line 6) progress. The total cost of this loop can be shown as the sum of the cost of the *sort* function across the iterations of the loop as follows

$n \cdot log(n) + (n-1) \cdot log(n-1) + (n-2) \cdot log(n-2) + ... + 2 \cdot log(2) + 1 \cdot log(1)$

To compute the worst case complexity for this loop we argue the following

$n \cdot log(n) + (n-1) \cdot log(n-1) + (n-2) \cdot log(n-2) + ... + 2 \cdot log(2) + 1 \cdot log(1)$

$\leq$

$n \cdot log(n) + n \cdot log(n) + n \cdot log(n) + ... + n \cdot log(n) + n \cdot log(n)$

The upper bound computed above resolves to the following

$n \cdot n \cdot log(n) = n^2 \cdot log(n)$

So the worst-case complexity for the function *sum1* is O($\boldsymbol{n^2 \cdot log(n)}$)

NOTE It is understood here that the last iteration where *n=1* will not happen as the loop stops at the condition *v.size() - 1*. This means that the cost $1 \cdot log(1)$ will not be incurred and should not actually be included. However, this does not affect the overall derivation of the complexity. Same applies to the solutions of *sum2* and *sum3* functions that follow.

Sum2

```
1  double sum2(std::vector<double>& v)
2  {
3      if (v.empty()) {
4          return 0.0;
5      }
6      for(size_t i = 0; i < v.size() - 1; ++i) {
7          std::partial_sort(v.begin() + i, v.begin() + i + 2, v
               .end());
8          v[i+1] += v[i];
9      }
10     return v.back();
11 }
```

The complexity of the *partial_sort(first, middle, last)* function (used at line 7 in the code) is O($n \cdot logm$), where n is the number of elements sorted between *first* and *last* elements and m is the number of elements between *first* and *middle* elements. From the code we see that in this case m=2. This implies that the complexity of the *partial_sort* function in this case resolves to O(n) ($n \cdot log2$ = n). Now, as the number of iterations progress (in loop at line 6) n – number of elements between *first* and *last* – decreases.

The total cost of this loop can be shown as the sum of the cost of the *partial_sort* function across the iterations of the loop as follows

$n + n - 1 + n - 2 + ... + 3 + 2 + 1 = \sum_{i=1}^{n} i$

The summation derived above ($\sum_{i=1}^{n} i$) resolves to $\frac{n^2+n}{2}$. So the worst-case complexity for the function *sum2* is O(n^2)

Sum3

```
1  double sum3(std::vector<double>& v)
2  {
3      std::multiset set(v.begin(), v.end());
4      while (set.size() > 1) {
5          std::multiset<double>::const_iterator itA = set.begin
               ();
6          std::multiset<double>::const_iterator itB = ++set.
               begin();
7          double c = *itA + *itB;
8          set.erase(itA, itB);
9          set.insert(c);
10     }
11     return !set.empty() ? *set.begin()
12                         : 0.0;
13 }
```

The complexity of the *erase* (at line 8) function is constant whereas the cost of *insert* (at line 9) function is O($\boldsymbol{logn}$) which makes the latter more interesting to study the growth of the function's complexity. Here, n is the number of elements in the set at the time of invoking the *insert* function. Note that as the number of iterations progress (in loop at line 4) n – number of elements in the set – decreases.

The total cost of this loop can be shown as the sum of the cost of the *insert* function across the iterations of the loop as follows

$$\boldsymbol{log(n) + log(n-1) + log(n-2) + ... + log(2) + log(1)}$$

To compute the worst case complexity for this loop we argue the following

$$\boldsymbol{log(n) + log(n-1) + log(n-2) + ... + log(2) + log(1)} \leq \boldsymbol{log(n) + log(n) + log(n) + ... + log(n) + log(n)}$$

The upper bound computed above resolves to the worst-case complexity of O($\boldsymbol{n \cdot log(n)}$)