INSTITUTE OF TECHNOLOGY

Worst Case Run-Time Analysis of Population day Operation

Group work: Siyuan Zhou, Feng Yang

April 25, 2015

1 INTRODUCTION

In this document is a detailed *worst case run-time analysis* of the `day()` operation of the `Population` class implemented in lecture. For convenience the relevant code for this operation is included here with numbered lines:

```
        public void day() {
1               mergeSort(genomes, 0, genomes.length - 1, new Genome[genomes.length]);
2               mostFit = genomes[0];
3               int i = numGenomes / 2;
4               int j = numGenomes / 2;
5               while (i++ < numGenomes - 1) {
6                       Random random = new Random();
7                       Genome genomeSelect = genomes[random.nextInt(j)];
8                       int methodSelected = random.nextInt(2);
9                       if (methodSelected == 0) {
10                              Genome genomeAdd = new Genome(genomeSelect);
11                              genomeAdd.mutate();
12                              genomes[i] = genomeAdd;
                        } else {
13                              Genome genomeAdd = new Genome(genomeSelect);
14                              Genome genomeSlected2 = genomes[random.nextInt(j)];
15                              genomeAdd.crossover(genomeSlected2);
16                              genomeAdd.mutate();
```

```
                    genomes[i] = genomeAdd;
                }
            }
    }

    private void mergeSort(Genome[] genomes, int first, int last,
                Genome[] genomeTemp) {
            if (first < last) {
                int mid = (first + last) / 2;
                mergeSort(genomes, first, mid, genomeTemp);
                mergeSort(genomes, mid + 1, last, genomeTemp);
                mergeArray(genomes, first, mid, last, genomeTemp);
            }
    }

    private void mergeArray(Genome[] genomes, int first, int mid, int last,
                Genome[] genomeTemp) {
            int i = first, j = mid + 1, m = mid, n = last, k = 0;
            while (i <= m && j <= n) {
                if (genomes[i].getFitness() <= genomes[j].getFitness())
                    genomeTemp[k++] = genomes[i++];
                else
                    genomeTemp[k++] = genomes[j++];
            }
            while (i <= m)
                genomeTemp[k++] = genomes[i++];
            while (j <= n)
                genomeTemp[k++] = genomes[j++];
            for (i = 0; i < k; i++)
                genomes[first + i] = genomeTemp[i];
    }

    public Integer fitness() {
            int n = string.length(), m = targetName.length();
            int l = n < m ? n : m;
            int fitnessTemp = Math.abs(n - m) * 2;
            for (int i = 0; i < l; i++) {
                if (string.charAt(i) != targetName.charAt(i))
                    fitnessTemp = fitnessTemp + 1;
            }
            this.fitness=fitnessTemp;
            this.calculateFitnessflag=true;
            return fitnessTemp;
    }
```

```java
public void crossover(Genome other) {
        Random random = new Random();
        int otherStringLength = other.getString().length();
        for (int i = 0, len = string.length(); i < len; i++) {
                int selectedString = random.nextInt(2);
                if (selectedString == 0) {
                        continue;
                } else {
                        if (i < otherStringLength) {
                                string.setCharAt(i, other.getString().charAt(i))
                        } else {
                                string.delete(i, string.length());
                                break;
                        }
                }
        }
    }

public void mutate() {
        Random random = new Random();
        double happenRate = random.nextDouble();
        if (happenRate <= mutationRate) {
                addCharRandomly();
        }
        if ((happenRate = random.nextDouble()) <= mutationRate) {
                deleteCharRandomly();
        }
        for (int i = 0, len = string.length(); i < len; i++) {
                if ((happenRate = random.nextDouble()) <= mutationRate) {
                        replaceCharRandomely(i);
                }
        }
    }

private void addCharRandomly() {
        Random random = new Random();
        int charIndex = random.nextInt(29);
        int addPosition = random.nextInt(string.length() + 1);
        if (addPosition < string.length()) {
                string.insert(addPosition, Genome.characters[charIndex]);
        } else {
                string.append(Genome.characters[charIndex]);
        }
        }

    private void deleteCharRandomly() {
67              if (string.length() == 1)
68                      return;
69              Random random = new Random();
70              int delPos = random.nextInt(string.length());
71              string.deleteCharAt(delPos);
        }

    private void replaceCharRandomely(int replacePos) {
72              if (replacePos >= string.length())
73                      return;
74              Random random = new Random();
75              int charIndex = random.nextInt(29);
76              string.setCharAt(replacePos, Genome.characters[charIndex]);
        }
```

2 Analysis

2.1 Line-By-Line

In this section we will look at each line and provide a precise estimate for the number of operations carried out on that line. For reference we will count all declarations, assignments, integer and Boolean arithmetic, and similar operations to be single operations. We will use *constants* and *functions* to estimate the number of operations of methods we are unsure of. In these cases we will seek *upper bounds* because we are looking for an expression of the worst case run-time.

For each line we will state the number of operations on that line and justify it in a few points. Let n be the number of genomes and m be the length of the longest Genome. Let $c_0$be the upper bounds on the running times of get random number from `random.nextInt(i)`. Let c_1 be the upper bounds on the running times of create a new `Genome` object. Let c_2 be the upper bounds on the running times of `StringBuilder.length()`. Let c_3 be the upper bounds on the running times of `Math.abs()`. Let c_4 be the upper bounds on the running times of `StringBuilder.charAt()`. Let c_5 be the upper bounds on the running times of `StringBuilder.delete()`. Let c_6 be the upper bounds on the running times of `StringBuilder.insert()`. Let c_7 be the upper bounds on the running times of `StringBuilder.append()`.

1. This line has a call to mergeSort. The analysis below shows the cost of mergeSort is $n \cdot (1 + logn)$.

2. This line has an assignment and get i^{th} value from array. The cost is 2.

3. This line has an assignment and a division. The cost is 2.

4. This line has an assignment and a division. The cost is 2.
5. A comparison, an addition and a subtraction. The cost is 3.
6. An assignment. The cost is 1.
7. An assignment, get i^{th} value from array, and a call to get random integer. The cost is $2 + c_0$.
8. An assignment and a call to get random integer. The cost is $1 + c_0$.
9. A boolean comparison. The cost is 1.
10. An assignment and a call to create a Genome object. The cost is $1 + c_1$.
11. A call to mutate method in Genome class. The analysis below shows the cost is c_m
12. An assignment and get i^{th} value from array. The cost is 2.
13. An assignment and a call to create a Genome object. The cost is $1 + c_1$.
14. An assignment, get i^{th} value from array, and a call to get random integer. The cost is $2 + c_0$.
15. A call to crossover method in Genome class. The analysis below shows the cost is c_c.
16. A call to mutate method in Genome class. The analysis below shows the cost is c_m.
17. An assignment and get i^{th} value from array. The cost is 2.
18. A comparison. The cost is 1.
19. An assignment, an addition and a division. The cost is 3.
20. Recursion method. The analysis is below.
21. Recursion method. The analysis is below.
22. Recursion method. The analysis is below.
23. Five assignment and an addition. The cost is 6.
24. Two comparison. The cost is 2.
25. An comparison, get i^{th} value from array twice, and two calls to getFitness method in Genome class. The analysis below shows the cost is $2 + c_g$.
26. An assignment, two addition, and get i^{th} value from array twice. The cost is 5.
27. An assignment, two addition, and get i^{th} value from array twice. The cost is 5.
28. An comparison. The cost is 1.

29. An assignment, two addition, and get i^{th} value from array twice. The cost is 5.
30. A comparison. The cost is 1.
31. An assignment, two addition, and get i^{th} value from array twice. The cost is 5.
32. An assignment, a comparison and an addition. The cost is 3.
33. An assignment, two addition, and get i^{th} value from array. The cost is 4.
34. Two assignments, two calls to get length of string. The cost is $2 + c_2$.
35. An assignment, a comparison. The cost is 2.
36. An assignment, a subtraction, a multiplication, and a call to Math.abs(). The cost is $3 + c_3$.
37. An assignment, a comparison, and an addition. The cost is 3.
38. A comparison and two calls to String.charAt() method. The cost is $1 + c_4$.
39. An addition. The cost is 1.
40. An assignment. The cost is 1.
41. An assignment. The cost is 1.
42. A return. The cost is 1.
43. An assignment. The cost is 1.
44. An assignment, a call to getString method in Genome class, and a call to length methd. The cost is $2 + c_2$.
45. Two assignments, a comparison and an addition. The cost is 4.
46. An assignment, a call to get random number. The cost is $1 + c_0$.
47. An comparison. The cost is 1.
48. Continue. No cost.
49. A comparison. The cost is 1.
50. A call to String.setCharAt method, a call to String.charAt method, and a call to getString method in Genome class. The cost is $1 + c_4$.
51. A call to String.length method, and a call to StringBuilder.delete method. The cost is $c_2 + c_5$.
52. An assignment. Cost is 1.

53. An assignment and a call to get random number. Cost is $1 + c_0$.

54. A comparison. Cost is 1.

55. A call to addCharRandomly method in Genome class. The analysis below shows the cost is c_a.

56. An assignment, a comparison and a call to get a random number. Cost is $2 + c_0$.

57. A call to deleteCharRandomly. The analysis below shows the cost is c_d

58. Two assignments, a comparison, an addition, and a call to String.length method. The cost is $4 + c_2$.

59. An assignment, a comparison and a call to get a random number. Cost is $2 + c_0$.

60. A call to replaceCharRandomly method in Genome class. The analysis below shows the cost is c_r

61. An assignment. Cost is 1.

62. An assignment and a call to get random number. Cost is $1 + c_0$.

63. An assignment, an addition, a call to String.length method and a call to get random number. Cost is $2 + c_0 + c_2$.

64. A comparison and a call to String.length method. Cost is $1 + c_2$.

65. A call to StringBuilder.insert method. The cost is c_6.

66. A call to StringBuilder.append method. The cost is c_7.

67. A call to StringBuilder.length method and a comparison. The cost is $1 + c_2$.

68. return. no cost.

69. An assignment. Cost is 1.

70. An assignment, a call to get random number. The cost is $1 + c_0$.

71. A call to StringBuilder.delete method. The cost is c_6.

72. A call to StringBuilder.length method and a comparison. The cost is $1 + c_2$.

73. return. no cost.

74. An assignment. Cost is 1.

75. An assignment, a call to get random number. The cost is $1 + c_0$.

76. A call to StringBuilder.setCharAt method. The cost is c_4.

2.2 Loops

There are four `while` loops in this method consisting of lines 5-17, 24-27, 28-29, and 30-31. There are also four `for` loops in this method consisting of lines 32-33, 37-39, 46-51, and 58-60. There are recursion structure in the method consisting of lines 18-33.

The loop on line 5-17 begins with the position equal to $n-1$ where n is the number of the genomes in the Population class. The worst case of loop ends when it goes over n/2 times. Let $f(n_1)$ be a function expressing the total cost of the `while` loop on line 5-17. We can express this sum as:

$$\begin{aligned} f(n_1) &= \sum_{i=n/2}^{n-1}(16+2\cdot c_0+c_1+2\cdot c_m+c_c) \\ &= c_{x1}\cdot n \end{aligned}$$

Let c_{x1}replace other constant.

The recursion and loop on line 18-33 is considered as a popular data structure type– merge sort. Merge sort is $O(n\log n)$ which n is the number of genomes. Here is the simple indication.

T(n) = 2T(n/2) + $O(n)$

T(n/2) = 2T(n/4) + $O(n)$

T(n/4) = 2T(n/8) + $O(n)$

......

T(1) = $O(1)$

By induction, we have: T(n) = $n\cdot(1+logn)$ which is $O(n\log n)$.

The for loop on line 37-39 begins with 0 and ends with the length of Genome. The worse case is when the Genome is the longest one. Let $f(n_2)$ be a function expressing the total cost of the `for` loop on line 37-39. We can express this sum as:

$$\begin{aligned} f(n_2) &= \sum_{i=0}^{m-1}(5+c_4) \\ &= c_{y1}\cdot m \end{aligned}$$

Let c_{y1} replace other constant.

The for loop on line 45-51 begins with 0 and ends with the length of Genome. The worse case is when the Genome is the longest one. Let $f(n_3)$ be a function expressing the total cost of the `for` loop on line 45-51. We can express this sum as:

$$\begin{aligned} f(n_3) &= \sum_{i=0}^{m-1}(8+c_0+c_2+c_4+c_5) \\ &= c_{y2}\cdot m \end{aligned}$$

Let c_{y2} replace other constant.

The for loop on line 58-60 begins with 0 and ends with the length of Genome. The worse case is when the Genome is the longest one. Let $f(n_4)$ be a function expressing the total cost of the `for` loop on line 58-60. We can express this sum as:

$$\begin{aligned} f(n_4) &= \sum_{i=0}^{m-1} (6 + c_2 + c_4) \\ &= c_{y3} \cdot m \end{aligned}$$

Let c_{y3} replace other constant.

2.3 Total Cost

The total cost of the `day` method is $g(n) = T(n) + f(n_1) + f(n_2) + f(n_3) + f(n_4) = c_{x1} \cdot n + c_{y1} \cdot m + c_{y2} \cdot + c_{y3} \cdot m + n \cdot (1 + logn)$. Simplifying and substituting $a = c_{x1}$ and $b = c_{y1} + c_{y2} + c_{y3}$ we have $g(n) = a \cdot n + b \cdot m$.

We can see that $g(n) \in \mathrm{O}(n)$(when n is much greater than m) or $g(n) \in \mathrm{O}(m)$(when m is much greater than n). No matter in which condition, g(n) takes linear times operation.