Big O notation Analysis in three simple algorithms

Phuriwat Angkoondittaphong
phuriwat.ang@student.mahidol.ac.th
Mahidol University

September 10, 2021

1 Introduction

Big O notation is mathematics tools to approximate that some particular functions (or sequences) grow or shrink at some rate. For instance, $f(x) = 9x - 4x^2 + x^{10}$ has $\mathcal{O}(f(x)) = \mathcal{O}(x^{10})$ as $x \to \infty$ or $g(x) = 91x^6$ has $\mathcal{O}(g(x)) = \mathcal{O}(x^6)$. Taking only terms that most significant to the whole expression.

Big O notation can be ordered in this way.

$$\mathcal{O}(1) < \mathcal{O}(log_c(x)) < \mathcal{O}(x^c) < \mathcal{O}(c^x) < \mathcal{O}(x!) < \mathcal{O}(x^x)$$

where c $\in \mathbb{R} > 1$.
$\mathcal{O}(1)$ means the whole expression doesn't depends on x (constant, for single variable function).

2 Big O notation on simple algorithms

If we know time used on in a simple fixed-time operation like addition of 32-bit integer, we can approximate time used on the sequences of operation. Here's example code.

```
1 public static int sumOfFirstN(int n){
2     int sum = 0;
3     for(int i = 1; i <= n; i++){
4         sum += i;
5     }
6     return sum;
7 }
```

Assuming adding java's integer, single Boolean arithmetic and assigning single variable use some particular constant time a, b, and c respectively. This example code will add variable sum from $i = 1$ to $i = n$ or write as mathematics expression as $sum = \sum_{i=1}^{n} i$.

Since there is an iteration in this code, some operation that normally use constant time become time-dependent on some variable (in this case is variable n) When running example code above, adding occurs 2n times (line 3 and 4), n times of Boolean arithmetic (line 3), and 1 + 2n assignments occurs (line 1, 2 and 3). We can say that total time used $t_a = 2n \cdot a + n \cdot b + (1 + 2n) \cdot c$ which can be rewrite as

$$t_a = (2a + b + 2c)n + (a + c)$$

If we know a, b, and c, we will be able to approximate time use on this algorithm with any n. We can conclude that times used by this method depends on variable n, since n controls iteration and anything else is constant. Therefore, this method is $\mathcal{O}(n)$.

2.1 Selection Sort

Selection sort is one of the most if not easiest to understand sort out there. The way of thought just "find max of sublist then place it at the end of sublist, do it whole list" Badaboom! Badabang! you got a sort algorithm.

2.1.1 Pseudocode

1. Initiate boundary between unordered and ordered list.
2. Find maximum of unordered list (by check one by one).
3. Send maximum to the end of unordered list (starting point of ordered list).
4. Repeat 2. until unorderded list is empty.

2.1.2 Example

We have an unordered list of numbers

$$l = [5, 2, 3, 9]$$

When we sort this list using selection sort l will change along Table 1.

Iteration	Unordered	ordered
0	[5, 2, 3, 9]	[ ]
1	[5, 2, 3]	[9]
2	[2, 3]	[5, 9]
3	[2]	[3, 5, 9]
4	[ ]	[2, 3, 5, 9]

Table 1: List during Selection sort.

2.1.3 Algorithm Analysis

Finding max in unordered list is $\mathcal{O}(k)$ where k is length of input list and algorithm has to do that exactly n times where n is length of list to be sort. Since, each iteration unordered list is shorter by one, which can write as

$$\mathcal{O}(n) + \mathcal{O}(n-1) + \mathcal{O}(n-2) + ... + \mathcal{O}(n-(n-2)) + \mathcal{O}(n-(n-1)) = \mathcal{O}(n^2)$$

2.1.4 Code

```
1     public static <T extends Comparable<T>> int findMax(List<T> l, int start, int end){
2         int len = end - start;
3         if(len <= 0) {
4             return -1;
5         }
6         int c = start;
7         for(int i = start+1; i < end; i++){
8             if(l.get(c).compareTo(l.get(i)) < 0){
9                 c = i;
10            }
11        }
12        return c;
13    }
14    public static <T extends Comparable<T>> void selectionSort(List<T> l){
15        int len = l.size(); int i, j; T tmp;
16        for(i = len-1; i >= 0; i--){
17            j = findMax(l, 0, i+1);
18            tmp = l.get(j);
19            l.set(j, l.get(i));
20            l.set(i, tmp);
21        }
22    }
```

2.1.5 Utility

Sorting has many utility but I don't know why right now I can't think anything else.

- in statistic when we want to see distribution of data, we can sort it and visualising it as bar chart, or time series.

2.2 Binary Search

Binary Search is require collection to be search in sorted unless it failed. This is because binary search take advantage of comparison. If a object that it try to find is less then the element in the middle, forget the right of the middle element, keep searching on left, since elements in the right side are always bigger than the element in the middle which is surely not the element it try to find.

2.2.1 Pseudocode

1. look at middle of list (starting left bound is 0 and right bound is length of list, middle index can be calcualate by $\lfloor (leftbound + rightbound)/2) \rfloor$.
2. check whether middle element is what we want, if it is, we are done. But if it doesn't compare if middle element is bigger or smaller.
3. If smaller, focusing on left side, change right bound to middle (searching left bound to middle).
4. If larger, focusing on right side, change left bound to middle (searching middle to right bound).
5. Then repeat 1. with new left or right bound.

2.2.2 Example

Define ordered list l as example

$$l = [2, 3, 5, 7, 11, 13, 17, 19, 23, 31, 37]$$

and we want to find whether 11 in this list. Try to walk though above procedures.

Iteration	left	middle	right
0	[2, 3, 5, 7, 11]	13	[17, 19, 23, 31, 37]
1	[2, 3]	5	[7, 11]
2	[7]	11	[ ]

Table 2: Working though binary search pseudocode.

Since middle is equals to 11, we can conclude that 11 is in the list. In the code everything will be index base (not value base like in Table 2) so we will be able to get the index and return it as well (as I write it in code).

2.2.3 Algorithm Analysis

For each iteration, algorithm checks a few of if-statement and divided candidate for searching by half. In the worst case that target is in the first or last index, algorithm at most performs $\log_2(n)$ iterations and no more. Thus, we can say that Binary search has time complexity of

$$\mathcal{O}(\log_2(n))$$

2.2.4 Code

```
    public static <T extends Comparable<T>> int searchIter(List<T> l, T o, int start, int end){
        if(l.isEmpty())
            return -1;
        int cmp, mid, i = 0;
        while(start != end){
            mid = (start + end) / 2;
            cmp = o.compareTo(l.get(mid));
            if(cmp == 0){
                System.out.println("used " + i + " iteration(s)");
                return mid;
            }
            if(cmp < 0){
                end = mid;
            }
            else if(cmp > 0){
                start = mid + 1;
            }
            i += 1;
        }
        return -1;
    }

    public static <T extends Comparable<T>> int searchIter(List<T> l, T o){
        return searchIter(l, o, 0, l.size());
    }
```

2.2.5 Utility

Searching is pretty self-explanatory. When you try to search something on sorted array, binary search is pretty much your first choose.

- Searching keys in database

2.3 ArrayReshape2d

ArrayReshape2d is algorithm change shape of the 2d array without changing its data.
From now on author will write shape of the 2d array as this format (r, c).
where r is number of row,
and c is number of column.

The idea behind is any dimension can be serialize to 1d array by some formula. For 2d array we can map data from 2d to 1d using

$$s = ic + j$$

where
s is index of serialized array
i is index of row of array and $0 \leq i < r$
j is index of column of array and $0 \leq i < c$.

From 1d array turn into any dimension array as well. Formula for 1d to 2d is

$$i = \lfloor s/c \rfloor$$

$$j = s \pmod{c}$$

2.3.1 Pseudocode

1. Create a new array that has dimension of (user provided) (r, c).
2. Initialize counting variable $s = 0$.
3. Traversing old array (for loop i, j).
   (a) While traversing, set new index row is $ri = \frac{s}{c}$ and new index column $ci = s \pmod{c}$.
   (b) Set new array value at (ri, ci) as old array value at (i, j).
   (c) Increment s.

2.3.2 Example

Author writes Array that has shape (4, 3) can be reshape to (6, 2) like example

$$\begin{bmatrix} 0 & 1 & 2 \\ 3 & 4 & 5 \\ 6 & 7 & 8 \\ 9 & 10 & 11 \end{bmatrix} \rightarrow^{\text{after reshape}} \begin{bmatrix} 0 & 1 \\ 2 & 3 \\ 4 & 5 \\ 6 & 7 \\ 8 & 9 \\ 10 & 11 \end{bmatrix}$$

(4, 3) to (3, 4)

$$\begin{bmatrix} 0 & 1 & 2 \\ 3 & 4 & 5 \\ 6 & 7 & 8 \\ 9 & 10 & 11 \end{bmatrix} \rightarrow^{\text{after reshape}} \begin{bmatrix} 0 & 1 & 2 & 3 \\ 4 & 5 & 6 & 7 \\ 8 & 9 & 10 & 11 \end{bmatrix}$$

(2, 3) to (6, 1)

$$\begin{bmatrix} 0 & 1 & 2 \\ 3 & 4 & 5 \end{bmatrix} \rightarrow^{\text{after reshape}} \begin{bmatrix} 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{bmatrix}$$

2.3.3 Algorithm Analysis

This algorithm is just traversing along input array. Number of computation work such as arithmetic and read/write memory increase along number of element of input array. Thus, time complexity of ArrayReshape2d is

$$\mathcal{O}(r \cdot c)$$

Where
r is number of rows in array.
c is number of column in array.

2.3.4 Code

```
    public static int[][] arrayReshape2d(int[][] mat, int r, int c) {
        if(r == -1){
            r = mat.length / c;
        }
        else if(c == -1){
            c = mat.length / r;
        }

        int[][] n = new int[r][c];
        int count = 0;
        int ri, ci;
        for(int i = 0; i < mat.length; i++){
            for(int j = 0; j < mat[0].length; j++){
                ri = count / c;
                ci = count % c;
                n[ri][ci] = mat[i][j];
                count++;
            }
        }
        return n;
    }
```

2.3.5 Utility

- Transforms vector into wanted shape before input into some maths model.

References

https://leetcode.com/problems/reshape-the-matrix/
https://numpy.org/doc/stable/reference/generated/numpy.reshape.html
https://en.wikipedia.org/wiki/Big_O_notation