Sergio E. Garcia Tapia
Algorithms by Sedgewick and Wayne (4th edition) [SW11]
September 30th, 2024

1.4: Analysis of Algorithms

Exercise 1. Show that the number of different triples that can be chosen from n items is precisely $n(n-1)(n-2)/6$. *Hint*: Use mathematical induction or a counting argument.

Solution.

Proof. This is the problem of choosing a combination of 3 out of n, which is given by $\binom{n}{3}$, and

$$\binom{n}{3} = \frac{n!}{3!(n-3)!} = \frac{n \cdot (n-1) \cdot (n-2) \cdot (n-3)!}{3!(n-3)!} = \frac{n(n-1)(n-2)}{6}$$

□

Exercise 2. Modify `ThreeSum` to work properly even when the `int` values are so large that adding two of them might cause integer overflow.

Solution. There are two cases when it comes to overflow:

(i) *Positive overflow.* The sum exceeds `Integer.MAX_VALUE`. If two terms sum to `Integer.MAX_VALUE + 1`, overflow occurs, and the value wraps around to `Integer.MIN_VALUE`. Thus, if $a + b =$ `Integer.MAX_VALUE + 1` and $c =$ `Integer.MIN_VALUE`, then we have a valid sum. However, if $a + b$ sums to anything larger, then no value of c will do because c cannot be smaller than `Integer.MIN_VALUE`.

(ii) *Negative overflow.* The sum of two negative numbers a and b, yielding a value below `Integer.MIN_VALUE`. In this case, it's impossible to have $a + b = c$ for any 32-bit two's complement integer c.

See the `com.segarciat.algs4.ch1.sec4.ex02.ThreeSum` class.

Exercise 3. Modify `DoublingTest` to use `StdDraw` to produce plots like the standard and log-log plots in the text, rescaling as necessary so that the plot always fills a substantial portion of the window.

Solution. See my `com.segarciat.algs4.ch1.sec4.ex03.DoublingTest` class.

Exercise 4. Develop a table like the one on page 181 for `TwoSum`.

Solution. The `TwoSum` program referenced is:

```
public class TwoSum {
   public static int count(int[] a)
   {  // Count pairs that sum to 0;
      // A: The entire method body
      int n = a.length;
      int count = 0;
      // B: The outermost for loop and its body (not including the statement i
         = 0)
      for (int i = 0; i < n; i++)
        // C: The innermost for loop and its body (not including the
           statement j = i + 1)
        for (int j = i + 1; j < n; j++)
          if (a[i] + a[j] == 0)
            // D: The number of times the if block is executed
            count++;
      return count;
    }
}
```

The table in 181 is used to analyze the running time of by keeping track of the frequency of each statement block, that is, the number of times the block is executed:

Statement block	Time in seconds	Frequency	Total time
D	t_0	x (depends on input)	t_0x
C	t_1	$\binom{n}{2}$	$t_1\binom{n}{2}$
B	t_2	n	t_2n
A	t_3	1	t_3

Therefore, the grand total is:

$$
\begin{aligned}
t_1\binom{n}{2} + t_2n + t_3 + t_0x &= t_1\frac{n(n-1)}{2} + t_2n + t_3 + t_0x \\
&= \frac{t_1}{2}n^2 + \left(t_2 - \frac{t_1}{2}\right)n + t_3 + t_0x
\end{aligned}
$$

The tilde approximation (assuming x is small) is

$$\sim \left(\frac{t_1}{2}\right)n^2$$

Hence the order of growth is n^2.

Exercise 5. Give tilde approximations for the following quantities:

(a) $n + 1$

(b) $1 + 1/n$

(c) $(1 + 1/n)(1 + 2/n)$

(d) $(2n^3 - 15n^2)$

(e) $\lg(2n)/\lg n$

(f) $\lg(n^2 + 1)/\lg n$

(g) $n^{100}/2^n$

Solution. The definition of tilde approximation given on page 179 of [SW11] says $\sim f(n)$ represents a function that, when divided by $f(n)$, approaches 1 as n grows.

(a) $\sim n$

(b) ~ 1, since $1/n$ approaches 0 as n grows, and hence $1+1/n$ approaches 1 as n grows.

(c) ~ 1, similar to (b).

(d) $\sim 2n^3$

(e) Since $\lg(2n) = \lg(2) + \lg(n)$, this means $\lg(2n)/\lg n = \lg(2)/\lg(n) + 1$. Hence, this is again ~ 1.

(f) Since $\lg(n^2+1) \approx \lg(n^2)$, and $\lg(n^2)/\lg(n) = 2\lg(n)/\lg(n)$, we conclude that this is ~ 2.

(g) Here we can just say $\sim (n^{100}2^{-n})$.

Exercise 6. Give the order of growth (as a function of n) of the running times of each of the following code fragments:

(a)

```
int sum = 0;
for (int k = n; k > 0; k /= 2)
   for (int i = 0; i < k; i++)
      sum++;
```

(b)

```
int sum = 0;
for (int i = 1; i < n; i *= 2)
   for (int j = 0; j < i; j++)
      sum++;
```

(c)

```
int sum = 0;
for (int i = 1; i < n; i *= 2)
   for (int j = 0; j < n; j++)
      sum++;
```

Solution. Let $m = \lfloor \lg n \rfloor + 1$. Then m is the number of bits needed to represent n in binary, and is the frequency of execution of the outer loop. In particular, $2^{m-1} \leq n < 2^m$.

(a) For each value of `k`, the `i` loop block containing the statement `sum++` will execute `k` times, where $k = \lfloor n/2^j \rfloor$ for $0 \leq j \leq m$. Thus the total number of times that `sum++` is executed is approximately given by

$$\begin{aligned}\sum_{j=0}^{m-1}\left\lfloor\frac{n}{2^j}\right\rfloor &< \sum_{j=0}^{m-1}\left\lfloor\frac{2^m}{2^j}\right\rfloor\\ &= \sum_{j=0}^{m-1}\frac{2^m}{2^j}\\ &= 2^m\sum_{j=0}^{m-1}\frac{1}{2^j}\\ &= 2^m\cdot\left(2-\frac{1}{2^{m-1}}\right)\\ &= 2^{m+1}-2\\ &= 2(2^m-1)\\ &\le 2(2^{\lg n+1}-1)\\ &= 2\cdot(2n-1)\end{aligned}$$

The order of growth is n, or linear.

(b) This is similar to (a), but the analysis is much simpler: `i` takes on the values 2^k for $0 \le k < m$, and the `j` loop executes `i` times for each `i`. Thus the total number of times `sum++` runs is approximately:

$$\begin{aligned}\sum_{k=0}^{m-1}2^k = 2^m - 1 &= 2^{\lg n+1}-1\\ &\le 2^{\lg n+1}-1\\ &= 2n-1\end{aligned}$$

Thus the order of growth is n, or linear.

(c) The outer loop executes m times, so again `i` takes on the values 2^k for $0 \le k < m$. For each `i`, the `j` loops executes `n` times. Thus the total number of times that `sum++` executes is nm times, which is exactly $n(\lfloor\lg n + 1)\rfloor$ times. Hence the order of growth is linearithmic.

Exercise 7. Analyze `ThreeSum` under a cost model that counts arithmetic operations (and comparisons) involving the input numbers.

Solution. The arithmetic operations in `ThreeSum` are all additions. The comparison operations are less than (`<`) and equal to (`==`).

Altogether, there are $n+1$ comparisons from `i < n`, $\binom{n}{2}+1$ comparisons from `j < n` (once for each pair (i,j)), $\binom{n}{3}+1$ comparisons from `k < n`, and $\binom{n}{3}$ comparisons for `a[i] + a[j] + a[k] == 0` (one for each triple (i,j,k)).

There are n additions from `i++`, $\binom{n}{2}$ additions for `j++`, $\binom{n}{3}$ additions for `k++`, $2\cdot\binom{n}{3}$ additions for `a[i] + a[j] + a[k]` (since it involves two additions for each triple), and the number of times `count++` executes is indeterminate since it depends precisely on how many times the control expression of the `if` evaluates to `true`.

Overall, then, the cost is dominated by the statements in the innermost `k` loop. Therefore, we can say that `ThreeSum` uses about $5\binom{n}{3}$ or $\sim \frac{5}{6}n^3$ arithmetic operations (including comparisons), and hence its order of growth is cubic under a cost model that counts arithmetic operations and comparisons.

Exercise 8. Write a program to determine the number of pairs of values in an input file that are equal. If your first try is quadratic, think again and use `Arrays.sort()` to develop a linearithmic solution.

Solution. See the `com.segarciat.algs.ch1.sec4.ex08.EqualNumberPairs` class. It uses `Arrays.sort()`, which has a linearithmic order of growth. Then it uses the fact that equal numbers are adjacent to each other to compute the frequency of occurrence of each number. If a number has a frequency f, then there are $\binom{f}{2}$ equal pairs corresponding to that number. This loop operates in linear time since it performs at most a constant number of operations in each iteration, and it iterates n times, where n is the number of items in the file.

Exercise 9. Give a formula to predict the running time of a program for a problem of size n when doubling experiments have shown that the doubling factor is 2^b and the running time for problems of size n_0 is t.

Solution. Let $T(n)$ be the running time of the program. Since the doubling factor is 2^b, we know that $T(n)$ has an order of growth approximately n^b, as claimed in [SW11] on page 192 (Section 1.4). Hence,

$$\begin{aligned}
T(n) &\sim n^b \\
&= \left(\frac{n}{n_0}n_0\right)^b \\
&= \left(\frac{n}{n_0}\right)^b n_0^b \\
&\sim \left(\frac{n}{n_0}\right)^b T(n_0)
\end{aligned}$$

Since $T(n_0) = t$, we can predict $T(n)$ to be approximately $\left(\frac{n}{n_0}\right)^b t$.

Exercise 10. Modify binary search so that it always returns the element with the smallest index that matches the search element (and still guarantees logarithmic running time).

Solution. See the `com.segarciat.algs4.ch1.sec4.ex10.BinarySearch` class. I implemented an `indexOf()` method that is similar to my `rank()` implementation in Exercise 1.1.29; see `com.segarciat.algs4.ch1.sec1.ex29.BinarySearch`.

In Exercise 1.1.29, the return value is a number that is between `0` and `a.length`, inclusive. There are two cases:

(i) When the `key` exists in the array, the return value `a[lo]` has value `key`. Moreover, `lo` is the smallest index such that `a[lo]` equals `key`.

(ii) When `key` is not in the array, then either `lo` is `a.length` (indicating that `key` exceeds every value in the array) or `lo` is between `0` and `a.length - 1`, but `a[lo]` is not equal to `key`.

My implementation for this exercise adapts the code my `rank()` method in 1.1.29 with these considerations.

Exercise 11. Add an instance method `howMany()` to `StaticSETofInts` (page 99) that finds the number of occurrences of a given key in time proportional to $\log n$ in the worst case.

Solution. My implementation again adapts the code from Exercise 1.1.29, namely the `rank()` and `rankGe()` methods. Both adaptations run in $\log n$ in the worst case. The `howMany()` calls each once, and hence it completes in time proportional to $\log n$ as well.

References

[SW11] Robert Sedgewick and Kevin Wayne. *Algorithms*. 4th ed. Addison-Wesley, 2011. ISBN: 9780321573513.