A Touch of Complexity Theory

CS 1025 Computer Science Fundamentals I

Stephen M. Watt

University of Western Ontario

Objectives

- Understand how running time or space used
  is a function of the problem size.
- Learn that this can be analyzed.
- See how to use this analysis to choose better algorithms.

Why Do We Double the Size?

```
class DataSet {
      private double[] data  = new double[20];
      private int      nused = 0;

      public void addValue(double val) {
             if (nused == data.length) {
                    double[] newData = new double[2*data.length];
                    for (int i = 0; i < data.length; i++)
                           newData[i] = data[i];
                    data = newData;
             }
              data[nused++] = val;
      }

      // All the rest is the same .....
   }
```

Suppose We Grow One at a Time

- Time to add the first is 1.
- Time to add the second is 2 (copy 1, add 1).
- ...
- Time to add the k-th is k (copy k-1, add 1).

- Time to add all of the first k is

$$T(k) = 1 + 2 + \ldots + k = \mathbf{(k^2 + k)/2}$$

Time to Add the First k, with Doubling

- Time to add the first is 1.
- Time to add the second is 1.
- ...
- Time to add the 20^{th} is 1.
- *Must double size, however, for 21. This involves copying 20.* Time to add the 21^{st} is 21 = 20 + 1.
- Time to add the 22^{nd} is 1.
- ...
- Time to add the 40^{th} is 1
- Time to add the 41^{st} is 41.
- Time to add the 42^{nd} is 1
- ...

Time to Add the First k Elements (contd)

- Time to add the first k is

  T(k) = 1 + 1 + 1 + ... + 1   (k of them) +
  + 20 + 40 + 80 + 160 + ...+ 20*2^n,   for max n such that 20*2^n < k
  = k + 20 × [1 + 2 + 4 + ... + 2^n] = k + 20 × [2^(n+1) – 1]

  Note n is the integer such that   20*2^n < k ≤ 20*2^(n+1)
  so   log[2](k/20) -1 ≤ n < log[2](k/20).

  Therefore, using the green inequality in the red expression,

  k + 20×[2^ (log[2](k/20) – 1+1) – 1] ≤ T(k) < k + 20×[2^ (log[2](k/20) +1) – 1]
  k + 20 × [k/20 – 1] ≤ T(k) < k + 20×[k/20 × 2 – 1]
  2k – 20 ≤ T(k) < 3k – 20

- That is rather a lot of algebra, but it shows the time to add the first k elements is proportional to k.

Doesn’t This Waste Space?

- Potentially *about half* the space is wasted?

  E.g. After enlarging from 20 to 21, have 19 unused slots.

- That is indeed the *worst case* behaviour.

- What is the behaviour *on average?*

Average Space Use

- Expect about 75 % used. Why?

- Suppose we have just doubled the size going from k to 2k to add element k+1.

  Then averaging over all cases k+1 to 2k we have

  once k+1 out of 2k are full
  once k+2 out of 2k are full
  ...
  once 2k out of 2k are full.

  => an average of 1.5k/2k = ¾ full for these cases.

  Same for the previous bunch (from k/2 to k),
  and the bunch before that (from k/4 to k/2), ...

  => On average ¾ of slots are full and ¼ are “wasted”.

Conclusion

- By doubling instead of adding one at a time, we waste on average k/4 space.
- By doubling instead of adding one at a time, we take time proportional to k instead of k^2.
- We can figure out how our programs behave by mathematical analysis.