ECE 264 Advanced C Programming

Contents

1 Complexity

What is the complexity of sorting using linked lists? Let's consider three scenarios:

- The input values are already sorted from the smallest to the largest.
- The input values are already sorted from the largest to the smallest.
- The input values are not sorted and the order is random.

In the first scenario, when a new value is added, it is larger than all values already in the list. Therefore, every time a number is inserted, we have to go through all existing nodes. Suppose there are n numbers to be inserted. When we insert i^{th} number ($1 \leq i \leq n$), we have to go through $i-1$ nodes. The total number of nodes we have visited after inserting n numbers is

$$\sum_{i=1}^{n}(i-1) = \frac{n(n-1)}{2}. \tag{1}$$

This is $\approx n^2$.

In the second scenario, when a new value is inserted, it is always smaller than any existing value in the list. Therefore, this new value is always inserted at the beginning of the list. Only one node is checked for each insertion and the complexity is $\approx n$.

In the third scenario, we do not know how many nodes to check. On average, we need to check half way of the nodes already in the list and the complexity is

$$\sum_{i=1}^{n} \frac{i-1}{2} = \frac{n(n-1)}{4}. \tag{2}$$

This is also $\approx n^2$. Hence, the complexity is $\approx n^2$ using a linked list for sorting.

2 Recursion and Complexity

Why do we care about complexity? There are several reasons. (1) We write computer programs to compute and to computer fast. Many programs have time constraints. For example, weather forecast has to predict a storm before the storm arrives. Banking programs have to finish all transactions before the next business day starts. If you do on-line shopping, you certainly want to find the items within a few seconds. If you are designing an autonomous vehicle, you need to recognize an obstacle before a collision. All these examples require fast computer programs. (2) Complexity is important but most students have not seen it. This may be the first time for most of you to learn complexity analysis. (3) The most important reason is that you can **easily** write programs that are terribly slow.

We can use recursion to implement binary search and can eliminate about half of the elements in one step. We divide the problem into two parts with the confidence that one half can be completely ignored.

Sometimes, we may use recursion inefficiently. Fibonacci numbers are an example.

$$f(n) = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ f(n-1) + f(n-2) & \text{if } n > 1. \end{cases} \tag{3}$$

and a simple, straightforward implementation

```
#include <stdlib.h>
#include <stdio.h>
unsigned int Fibonacci(unsigned int n)
{
  if (n == 0) { return 0; }
  if (n == 1) { return 1; }
  return (Fibonacci(n - 1) + Fibonacci(n - 2));
```

```
}

int main(int argc, char * argv[])
{
  unsigned int n;
  if (argc < 2)
    {
      printf("need a number\n");
      return -1;
    }
  n = strtol(argv[1], (char **)NULL, 10);
  printf("f(%d) = %d\n", n, Fibonacci(n));
  return 0;
}
```

Let's consider calling $f(5)$. This call is computed by calling $f(4) + f(3)$. The former is computed by calling $f(3) + f(2)$. Therefore, we can write

$f(5) = f(4) + f(3) = (f(3) + f(2)) + f(3) = 2f(3) + f(2)$.

We can continue to compute

$f(3) = f(2) + f(1)$

and

$f(5) = 2f(3) + f(2) = 2(f(2) + f(1)) + f(2) = 3f(2) + 2f(1) = 3(f(1) + f(0)) + 2f(1) = 5f(1) + 3f(0)$.

What is the problem? We are calling $f(2)$ three times and each time it calls $f(1) + f(0)$. Let's see how many times $f(i)$ is called for computing $f(n)$ when $0 < i < n$.

```
/* fabonacci.c */
#include <stdlib.h>
#include <stdio.h>
const int maxN = 50;
int * callCounter;
unsigned int Fibonacci(unsigned int n)
{
  callCounter[n] ++;
  if (n == 0) { return 0; }
  if (n == 1) { return 1; }
  return (Fibonacci(n - 1) + Fibonacci(n - 2));
}
```

```
int main(int argc, char * argv[])
{
  int cnt;
  callCounter = malloc(maxN * sizeof(int));
  for (cnt = 0; cnt < maxN; cnt ++)
    {
      callCounter[cnt] = 0;
    }
  Fibonacci(maxN - 1);
  for (cnt = 0; cnt < maxN; cnt ++)
    {
      printf("callCounter[%2d] = %d\n",
             cnt, callCounter[cnt]);
    }
  free (callCounter);
  return 0;
}
/*
  callCounter[ 0] = 2584
  callCounter[ 1] = 4181
  callCounter[ 2] = 2584
  callCounter[ 3] = 1597
  callCounter[ 4] = 987
  callCounter[ 5] = 610
  callCounter[ 6] = 377
  callCounter[ 7] = 233
  callCounter[ 8] = 144
  callCounter[ 9] = 89
  callCounter[10] = 55
  callCounter[11] = 34
  callCounter[12] = 21
  callCounter[13] = 13
  callCounter[14] = 8
  callCounter[15] = 5
  callCounter[16] = 3
  callCounter[17] = 2
  callCounter[18] = 1
  callCounter[19] = 1
*/
```

As you can see, to compute $f(19)$, $f(5)$ is called 610 times. Each time it is computed by calling $f(4)$, $f(3)$, ..., $f(0)$. If you look at the numbers carefully, `callCounter[19]` is

actually $f(1)$, `callCounter[18]` is $f(2)$, `callCounter[17]` is $f(3)$. For any value i, $f(i)$ does not change. Why can't we **compute $f(i)$ once and remember its value**? We do not have to compute $f(i)$ over and over again.

```
/* compare.c */
#include <sys/time.h>
#include <stdlib.h>
#include <stdio.h>
const int maxN = 46;
unsigned int Fibonacci1(unsigned int n)
{
  if (n == 0) { return 0; }
  if (n == 1) { return 1; }
  return (Fibonacci1(n - 1) + Fibonacci1(n - 2));
}

unsigned int Fibonacci2(unsigned int n)
{
  unsigned int * f;
  unsigned int cnt;
  unsigned int result;
  f = malloc((n + 2) * sizeof(unsigned int));
  f[0] = 0;
  f[1] = 1;
  for (cnt = 2; cnt <= n; cnt ++)
    {
      f[cnt] = f[cnt - 1] + f[cnt - 2];
    }
  result = f[n];
  free (f);
  return result;
}

int main(int argc, char * argv[])
{
  int cnt;
  struct timeval t1;
  struct timeval t2;
  int f1;
  int f2;
  for (cnt = 0; cnt < maxN; cnt ++)
    {
      gettimeofday(& t1, NULL);
```

```
      f1 = Fibonacci1(cnt);
      gettimeofday(& t2, NULL);
      printf("n = %2d, F1 %9f sec = %10d",
             cnt,
             (t2.tv_sec - t1.tv_sec) +
             1e-6 * (t2.tv_usec - t1.tv_usec),
             f1);
      gettimeofday(& t1, NULL);
      f2 = Fibonacci2(cnt);
      gettimeofday(& t2, NULL);
      printf(", F2 %9f sec = %10d\n",
             (t2.tv_sec - t1.tv_sec) +
             1e-6 * (t2.tv_usec - t1.tv_usec),
             f2);
    }
  return 0;
}
```

We implement Fibonacci again in `Fibonacci2` using **bottom-up**. The value of $f(i)$ for each i is calculated once only. Is that faster? Let's execute this program and see the difference.

```
/*
n =  0, F1  0.000001 sec =          0, F2  0.000057 sec =          0
n =  1, F1  0.000000 sec =          1, F2  0.000001 sec =          1
n =  2, F1  0.000001 sec =          1, F2  0.000001 sec =          1
n =  3, F1  0.000000 sec =          2, F2  0.000001 sec =          2
n =  4, F1  0.000001 sec =          3, F2  0.000001 sec =          3
n =  5, F1  0.000001 sec =          5, F2  0.000001 sec =          5
n =  6, F1  0.000001 sec =          8, F2  0.000001 sec =          8
n =  7, F1  0.000001 sec =         13, F2  0.000001 sec =         13
n =  8, F1  0.000002 sec =         21, F2  0.000001 sec =         21
n =  9, F1  0.000001 sec =         34, F2  0.000000 sec =         34
n = 10, F1  0.000002 sec =         55, F2  0.000000 sec =         55
n = 11, F1  0.000003 sec =         89, F2  0.000001 sec =         89
n = 12, F1  0.000005 sec =        144, F2  0.000001 sec =        144
n = 13, F1  0.000008 sec =        233, F2  0.000000 sec =        233
n = 14, F1  0.000012 sec =        377, F2  0.000001 sec =        377
n = 15, F1  0.000019 sec =        610, F2  0.000001 sec =        610
n = 16, F1  0.000031 sec =        987, F2  0.000001 sec =        987
n = 17, F1  0.000049 sec =       1597, F2  0.000001 sec =       1597
n = 18, F1  0.000079 sec =       2584, F2  0.000001 sec =       2584
n = 19, F1  0.000135 sec =       4181, F2  0.000001 sec =       4181
n = 20, F1  0.000199 sec =       6765, F2  0.000001 sec =       6765
n = 21, F1  0.000334 sec =      10946, F2  0.000001 sec =      10946
n = 22, F1  0.000540 sec =      17711, F2  0.000001 sec =      17711
n = 23, F1  0.000874 sec =      28657, F2  0.000001 sec =      28657
n = 24, F1  0.001415 sec =      46368, F2  0.000001 sec =      46368
```

```
n = 25, F1  0.002201 sec =      75025, F2  0.000001 sec =      75025
n = 26, F1  0.003608 sec =     121393, F2  0.000001 sec =     121393
n = 27, F1  0.005987 sec =     196418, F2  0.000001 sec =     196418
n = 28, F1  0.009692 sec =     317811, F2  0.000000 sec =     317811
n = 29, F1  0.015671 sec =     514229, F2  0.000001 sec =     514229
n = 30, F1  0.025064 sec =     832040, F2  0.000001 sec =     832040
n = 31, F1  0.040995 sec =    1346269, F2  0.000001 sec =    1346269
n = 32, F1  0.066309 sec =    2178309, F2  0.000001 sec =    2178309
n = 33, F1  0.107114 sec =    3524578, F2  0.000001 sec =    3524578
n = 34, F1  0.172566 sec =    5702887, F2  0.000001 sec =    5702887
n = 35, F1  0.274598 sec =    9227465, F2  0.000001 sec =    9227465
n = 36, F1  0.446309 sec =   14930352, F2  0.000002 sec =   14930352
n = 37, F1  0.716845 sec =   24157817, F2  0.000001 sec =   24157817
n = 38, F1  1.157057 sec =   39088169, F2  0.000001 sec =   39088169
n = 39, F1  1.870791 sec =   63245986, F2  0.000001 sec =   63245986
n = 40, F1  3.022568 sec =  102334155, F2  0.000002 sec =  102334155
n = 41, F1  4.886978 sec =  165580141, F2  0.000001 sec =  165580141
n = 42, F1  7.937471 sec =  267914296, F2  0.000001 sec =  267914296
n = 43, F1 12.791690 sec =  433494437, F2  0.000001 sec =  433494437
n = 44, F1 20.751278 sec =  701408733, F2  0.000001 sec =  701408733
n = 45, F1 33.568536 sec = 1134903170, F2  0.000002 sec = 1134903170
*/
```

In this comparison, the time to compute `Fibonacci2` does not change much as n increases. The time is so short (microsecond), the numbers do not really mean anything. In contrast, the execution time of `Fibonacci1` grows to noticeable values, half a minute when n is 45.

If you want to be a good programmer, you certainly want your programs to produce results fast. **If you are not careful, you can write a program (or a function) that is simple, straightforward, "elegant", and terribly slow.** It is extremely important to understand this concept.

The complexity of `Fibonacci1` is $\approx 2^n$. The following is a sketch of the reasoning. Remember

$$f(n) = f(n-1) + f(n-2) = 2f(n-2) + f(n-3).$$

The problem of `Fibonacci1` is that $f(n-2)$ is computed twice. We further expand the formula:

$$f(n) = 2f(n-2) + f(n-3) = 3f(n-3) + 2f(n-4) = 5f(n-4) + 3f(n-5).$$

`Fibonacci1` computes $f(n-4)$ five times. We can count only four time; this **underestimates** the complexity but it is simpler. Following this procedure, computing $f(n)$ will requires computing $f(n-2k)$ more than 2^k times. Hence, using `Fibonacci1` to compute $f(n)$ requires calling $f(1)$ more than $2^{\frac{n}{2}} = (\sqrt{2})^n$ times. This is an underestimation so `Fibonacci1` takes at least exponential time to compute $f(n)$.

Another way to compute $f(n)$ is using this formula

$$f(n) = \frac{1}{\sqrt{5}}[(\frac{1+\sqrt{5}}{2})^n - (\frac{1-\sqrt{5}}{2})^n]. \tag{4}$$

You can verify that $f(n) = f(n-1) + f(n-2)$.

`Fibonacci2`'s complexity is $\approx n$ because it computes $f(n)$ for each n only once. **It remembers the values so that it does not recompute.**

3 From 2^n to $\lg n$

You can compute Fibonacci numbers in $\approx \lg n$ by thinking of this problem in two ways. First, you can compute a^n in $\lg n$ time by doubling the exponent each time, instead of increasing the exponent by one each time.

$$\begin{aligned} a^1 &= a \\ a^2 &= a \cdot a \\ a^4 &= a^2 \cdot a^2 \\ a^{2k} &= a^k \cdot a^k. \end{aligned} \tag{5}$$

Another solution is to recognize the following relation

$$f(2n) = f(n-1)f(n) + f(n)^2. \tag{6}$$

In one step, we reduce $2n$ to n and $n-1$, namely reducing it by half. Obviously, you should **not** compute $f(n)$ twice in order to obtain $f(n)^2$. Instead, your program should remember $f(n)$ so that computing $f(n)^2$ is a single step using one multiplication. Continue this process to reduce n by half again in another single step.

Why is this important? Why do we need to learn this in a *programming class*? This is not in the textbook. Do we care about mathematics? Yes, if you want to be an excellent engineer. This example shows that you can write code in different ways. Consider these different ways to compute $f(n)$, from 2^n to n to $\lg n$. If n is one thousand, the first will take longer than the history of the universe; the third will take no more than one second.

Why were computers invented? To compute and to compute faster. As more and more information is handled on-line (hundreds of millions of users on facebook.com, for example), your career depends on the ability to design and write fast programs.