Computational complexity (exercises)

— P.04 —

Summary:

- Paper and pencil exercises (with solutions and computer verification)
- Extra problems (without solutions)
- Empirical study of the computational complexity of three algorithms
- Formal and empirical computational complexity of several algorithms

Paper and pencil exercises (part 1a, code)

Give a formula for the value returned by each of the following functions, and give their running time in Big Theta notation. Write a program to confirm your results (you can find these functions in the file `functions.c`, stored in the archive `P04.tgz`).

- ```
  int f1(int n)
  {
    int i,r = 0;

    for(i = 1;i <= n;i++)
      r += 1;
    return r;
  }
  ```

- ```
  int f2(int n)
  {
    int i,j,r = 0;

    for(i = 1;i <= n;i++)
      for(j = 1;j <= i;j++)
        r += 1;
    return r;
  }
  ```

- ```
  int f3(int n)
  {
    int i,j,r = 0;

    for(i = 1;i <= n;i++)
      for(j = 1;j <= n;j++)
        r += 1;
    return r;
  }
  ```

- ```
  int f4(int n)
  {
    int i,r = 0;

    for(i = 1;i <= n;i++)
      r += i;
    return r;
  }
  ```

- ```
  int f5(int n)
  {
    int i,j,r = 0;

    for(i = 1;i <= n;i++)
      for(j = i;j <= n;j++)
        r += 1;
    return r;
  }
  ```

- ```
  int f6(int n)
  {
    int i,j,r = 0;

    for(i = 1;i <= n;i++)
      for(j = 1;j <= i;j++)
        r += j;
    return r;
  }
  ```

Paper and pencil exercises (part 1b, solutions)

Let $r(n)$ be the value returned by the function and let $t(n)$ be the corresponding number of iterations of the body of the inner loop. Then, for $n > 0$,

- for the `f1` function,

$$t_1(n) = r_1(n) = \sum_{i=1}^{n} 1 = n.$$

  In this case we have $t_1(n) = \Theta(n)$.

- for the `f2` function,

$$t_2(n) = r_2(n) = \sum_{i=1}^{n}\Big(\sum_{j=1}^{i} 1\Big) = \sum_{i=1}^{n} i = \frac{n(n+1)}{2}.$$

  In this case we have $t_2(n) = \Theta(n^2)$.

- for the `f3` function,

$$t_3(n) = r_3(n) = \sum_{i=1}^{n}\Big(\sum_{j=1}^{n} 1\Big) = \sum_{i=1}^{n} n = n\sum_{i=1}^{n} 1 = n^2.$$

  Since $t_3(n) = r_3(n)$ we have $t_3(n) = \Theta(n^2)$.

- for the `f4` function,

$$r_4(n) = \sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

  and $t_4(n) = r_1(n)$.

- for the `f5` function,

$$\begin{aligned} t_5(n) = r_5(n) &= \sum_{i=1}^{n}\Big(\sum_{j=i}^{n} 1\Big) = \sum_{i=1}^{n}(n-i+1) \\ &= (n+1)\sum_{i=1}^{n} 1 - \sum_{i=1}^{n} i = \frac{n(n+1)}{2}. \end{aligned}$$

  In this case we have $t_5(n) = \Theta(n^2)$.

- for the `f6` function,

$$\begin{aligned} r_6(n) &= \sum_{i=1}^{n}\Big(\sum_{j=1}^{i} j\Big) = \sum_{i=1}^{n} \frac{i(i+1)}{2} \\ &= \frac{1}{2}\frac{n(n+1)(2n+1)}{6} + \frac{1}{2}\frac{n(n+1)}{2} \\ &= \frac{n(n+1)}{12}(2n+1+3) = \frac{n(n+1)(n+2)}{6} \end{aligned}$$

  and $t_6(n) = r_2(n)$. Note that $r_6(n) = \Theta(n^3)$.

For $n \leqslant 0$, in all these case we have $r(n) = t(n) = 0$.

Paper and pencil exercises (part 2a, some problems)

Each one of the statements

- S1: $f(n) = O\big(g(n)\big)$,
- S2: $f(n) = \Omega\big(g(n)\big)$,
- S3: $f(n) = \Theta\big(g(n)\big)$,

can be either true of false for each of the following pairs of functions. Determine which ones are true and explain why.

$f(n)$	$g(n)$
$\log n^2$	$\log n + 2$
$\sqrt{n}$	$\log n^2$
$n\sqrt{n}$	n^2
$n \log n$	n
$n \log n$	$10n \log n + n$
$n \log n$	$n^2 + n \log n$

List the functions given below from lowest to highest order. Mark the functions with the same order.

20	$\log n$	$\log \log n$	1.000001^n	2^n	$0.1n \log n$
n	3^n	$\frac{n}{\log n}$	$n \log n + 100$	2^{n+10}	$n + \frac{100}{n}$
$n!$	$n + 10^9$	$n^2 + n\sqrt{n} \log n$	$\log^2 n$	n^2	$\log n + 10 \log \log n$

Paper and pencil exercises (part 2b, solutions)

$f(n)$	$g(n)$	comparison to perform	true statements
$\log n^2$	$\log n + 2$	$\log n$ compared to $\log n$, tie	S1, S2, and S3
$\sqrt{n}$	$\log n^2$	$n^{1/2}$ compared to $\log n$, $f(n)$ wins	S2
$n\sqrt{n}$	n^2	$n^{3/2}$ compared to n^2, $f(n)$ loses	S1
$n \log n$	n	$n \log n$ compared to n, $f(n)$ wins	S2
$n \log n$	$10n \log n + n$	$n \log n$ compared to $n \log n$, tie	S1, S2, and S3
$n \log n$	$n^2 + n \log n$	$n \log n$ compared to n^2, $f(n)$ loses	S1

rank	function(s)
1	20
2	$\log \log n$
3	$\log n$, $\log n + 10 \log \log n$
4	$\log^2 n$
5	$\frac{n}{\log n}$
6	n, $n + \frac{100}{n}$, $n + 10^9$
7	$0.1n \log n$, $n \log n + 100$
8	n^2, $n^2 + n\sqrt{n} \log n$
9	1.000001^n
10	2^n, 2^{n+10}
11	3^n
12	$n!$

(Things in red are not significant)

(S1 is like $\leqslant$)

(S2 is like $\geqslant$)

(S3 is like $=$)

Extra problems

Give a formula for the value returned by each of the following functions (if applicable), and give their running time in Big Theta notation. Write a program to confirm your results (you can find these functions in the file `functions_extra.c`, stored in the archive `P04.tgz`).

- ```
  int g1(int n)
  {
    int i,j,r = 0;

    for(i = 0;i <= n;i++)
      for(j = i;j >= 0;j--)
        r += i - j;
    return r;
  }
  ```
- ```
  int g2(int n)
  {
    int r = 0;

    for(int i = 0;i < 2 * n;i += 2)
      for(int j = i;j <= 2 * n;j += 2)
        r += j;
    return r;
  }
  ```
- ```
  void g3(int n,int *a)
  {
    for(int i = 1;i <= n;i++)
      for(int j = i;j <= n;j += i)
        a[j] = i;
  }
  ```
- ```
  int g4(int n)
  {
    int r = 0;

    for(int i = 1;i <= n;i *= 2)
      r += i;
    return r;
  }
  ```

Computational complexity of three algorithms

Given a sequence of n distinct integers $a_1, a_2, \ldots, a_n$, our task is to determine all pairs (a_i, a_j) such that $a_i + a_j$ is equal to a given v. The program `find_pairs.c` solves this problem in three different ways.
Study, compile, and run the program. What can we say about the time and space complexities of each of the three algorithms coded in the program? (In the space complexity do not take into account the space needed to store the algorithm's input.) Which of the three algorithms has a better computational complexity? Which one uses less extra space?

Auxiliary information:

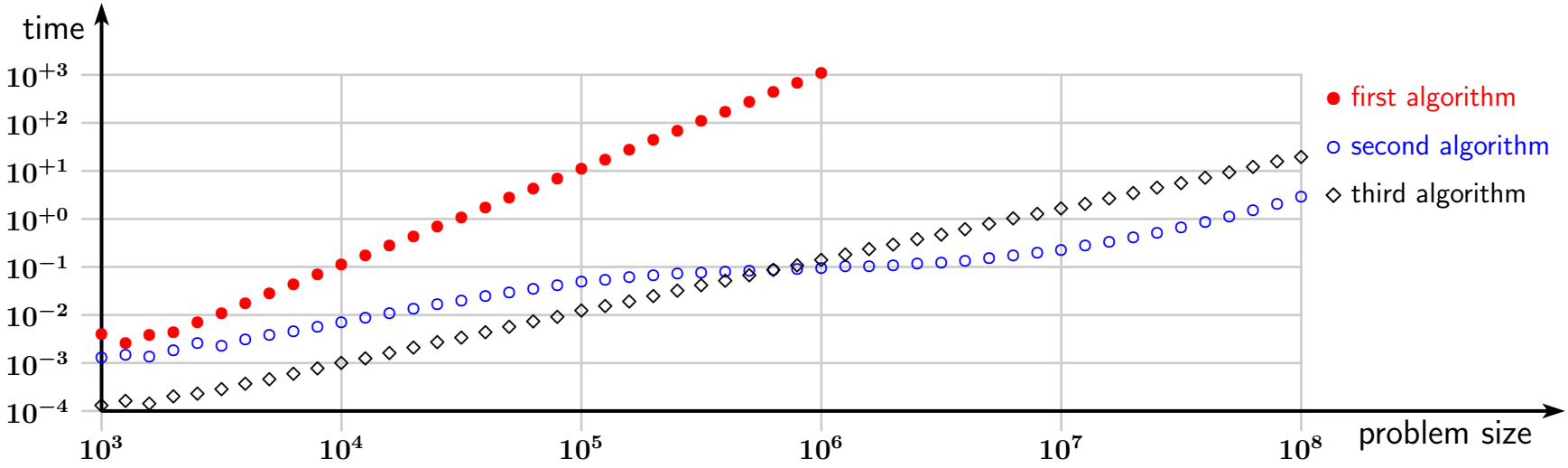
- In the three algorithms (a_i, a_j) is considered to be the same as (a_j, a_i), so only one of the two is printed. It is also assumed that in a pair $i \neq j$.
- The second algorithm requires non-negative integers.
- The `qsort` function sorts an array using a user supplied comparison function. [Study how this is done!] It has an average case complexity of $O(n \log n)$ and a worst case complexity of $O(n^2)$. It can be replaced by another sorting routine that has a worst case complexity of $O(n \log n)$. So, in this problem assume that sorting can be done in $O(n \log n)$.
- The `calloc` function allocates a memory region with a size (number of bytes) that is the product of its two arguments, fills that region with zeros, and returns a pointer to its starting location.
- The `malloc` function allocates a memory region with a size that is given in its only argument and returns a pointer to its starting location. The initial contents of the memory are arbitrary.
- The `free` function frees (deallocates) a memory region previously allocated by either the `calloc` or `malloc` functions.

Computational complexity of three algorithms (solution)

The first algorithm uses two nested `for` loops to iterate over all pairs of indices of the input array. Is has a time complexity of $O(n^2)$ and a space complexity of $O(1)$. The algorithm can be modified to work with real numbers.

For a sum value of `v` (an unsigned integer), the second algorithm uses a temporary array with `v+1` elements to record the unsigned integers that appear in the input array. It then computes `v-a[i]` for all elements of the input array and checks if the difference is marked or not in the temporary array. It has a time complexity of $O(n)$ and a space complexity of $O(v)$. The algorithm cannot be modified to work with real numbers.

The third algorithm sorts the input array and then makes a single pass (one index going up and another index going down) through the array. It has a time complexity of $O(n \log n)$, due to the sorting routine, and a space complexity of $O(n)$, to store the sorted array. The algorithm can be modified to work with real numbers.



Formal and empirical computational complexity of several algorithms

Extract the file `examples.c` from the archive `P04.tgz`. This file contains the code of all functions described in the T.04 lecture. Redoing the formal analysis done in the T lecture, determine the computational complexity of each one of the functions. Confirm your results by adding code to each of the functions to count (and print at the end) the number of times that the body of the innermost loop is executed.

Note that in C it is possible to write code like this (in C++ that is not possible):

```
for(n = 1;n <= 10;n++)
{
  double A[n][n]; // inside a code block non static arrays do NOT need to have a size defined at compile time!
  int i,j;

  for(i = 0;i < n;i++)
    for(j = 0;j < n;j++)
      A[i][j] = (double)rand() / (double)RAND_MAX; // one way to get uniformly distributed pseudo-random numbers
  //
  // put more stuff here, such as a call to an O(n^2) or an O(n^3) function
  //
}
```

For each interesting case (say, one for each computational complexity), make a log-log graph of the number of times the inner loop was executed versus the problem size.