Computer Systems Tutorial

November 10^{th} 2023

Exercise 1. Determine the complexity of the following pieces of code in terms of big-$\mathcal{O}$ notation. For this, derive an expression to denote the number of steps which are needed to compute the pieces and identify the dominant term.

```
if (n == 1) {
    return;
}

for (int i = 0; i < n - 1; i++) {

    for (int j = 0; j < n - i - 1; j++) {

        if (arr[j] > arr[j + 1]) {
        // assume it takes
        // constant number of steps
            swap(arr[j], arr[j + 1]);
        }
}
```

(a)

```
for (int i = 0; i < n - 1; i++) {

    for (int j = 0; j < n - i - 1; j++) {

        if (arr[j] > arr[j + 1]) {
        // assume it takes
        // constant number of steps
            swap(arr[j], arr[j + 1]);
        }
    }
}

for (int i = 0; i < n; i++) {

    for (int j = 1; j < n; j = j*2) {

        k = k + j;

    }
}
```

(b)

```
    for (int i = 0; i < n; i++) {

    // pow(x,y) raises x to the power of y
        for (int j = 0; j < pow(2,i); j++) {

            k = k * j;

    }
```

(c)

```
int fibonacci(n) {

    if (n <= 2) return 1;

    return fibonacci(n-1) + fibonacci(n-2);
}
```

(d)

Solution

a We need to compute the number of steps the program piece makes for the input depending on n. This will give us a formula that expresses the number of steps and by identifying the dominant term in the formula we will come up with the complexity in terms of big-$\mathcal{O}$. As we are interested in the worst-case complexity, we can skip lines 1-3 (or assume that only 1 step is needed to compute the *if* statement). Similarly, we can assume that only 1 step is needed to compute the expression in lines 9-13.

The number of steps is determined by the number of iterations of the innermost loop and by how many times the whole loop is invoked. We will proceed by looking at how many times the inner loop executes for each value of i (because the number of iterations is bounded by $n-i-1$). And, as we assumed above the loop performs only one step at each iteration.

- For $i = 0$ the possible values of j are $[0, \ldots, n-1)$ (the right number is not included as we have a strict $<$). We got $n-1$ by plugging $i = 0$ into $n-i-1 = n-0-1 = n-1$. Hence, we have $n-1$ iterations (for j from 0 to $n-2$ because $n-1$ is not included) of the inner loop for $i = 0$.
- For $i = 1$ the possible values of j are $[0, \ldots, (n-1-1)) = [0, \ldots, (n-2))$ and hence $n-2$ steps.
- Similarly, for $i = 2$ we have $n-3$ steps and so on.
- We end when $i = n-2$ and then the possible values of j are $[0, \ldots, (n-(n-2)-1)) = [0, \ldots, 1)$ and we have only one step.

So far we found the number of steps for each value of i and to find the total number of steps we need to sum them for all i. We have a sum $1 + 2 + \ldots n-2 + n-1$ which is an arithmetic progression[1]. The sum is equal to $\frac{1+(n-1)}{2} \times (n-1) = \frac{n}{2} \times (n-1) = \frac{1}{2}n^2 - \frac{1}{2}n$. This formula represents the total number of steps the two loops take. We can add an extra 1 to this formula to account for the first 3 lines and end up with $\frac{1}{2}n^2 - \frac{1}{2}n + 1$. The dominant term here is n^2 and hence the complexity is $\mathcal{O}(n^2)$.

b Recall that the number of steps for lines 1-11 is $\frac{1}{2}n^2 - \frac{1}{2}n$ and we only need to find the number of steps for lines 13-20. The number of iterations in the inner loops does not depend on i so the total number of steps would be the number of steps the inner loop does times the number of steps the inner loop is executed which is given by `i = 0; i < n`, i.e. it is executed n times. The inner loop does $log_2(n)$ iterations because j goes from 1 to $n-1$ (actually we need to round this logarithm up, but it does not matter that much) and is doubled every iteration, and the number of times 2 should be doubled to reach n is exactly $log_2(n)$ as $2^{log_2(n)} = n$ (and since we start from 1 and reach $n-1$ the number of iterations is the same). We can consider a couple of examples.

- $n = 4$ then $log_2(4) = 2$ and possible values of j are $1, 2$, i.e. exactly 2 of them.
- $n = 16$ then $log_2(16) = 4$ and possible values of j are $1, 2, 4, 8$, i.e. exactly 4 of them.

Hence, we have $log_2(n)$ iterations for each value of i of which there are n. And in total it gives us $nlog_2(n)$ steps. Recalling the number of steps for lines 1 - 11 we get the formula $\frac{1}{2}n^2 - \frac{1}{2}n + nlog_2(n)$ and n^2 is the dominant term (since $n^2 > nlog_2(n)$ as $log_2(n) < n$). Ultimately, the complexity is $\mathcal{O}(n^2)$.

[1] `https://en.wikipedia.org/wiki/Arithmetic_progression`

c To find the number of steps for this piece we need to essentially follow the same routine as in (a). That is, for each value of i we need to find the number of iterations the inner loop does.

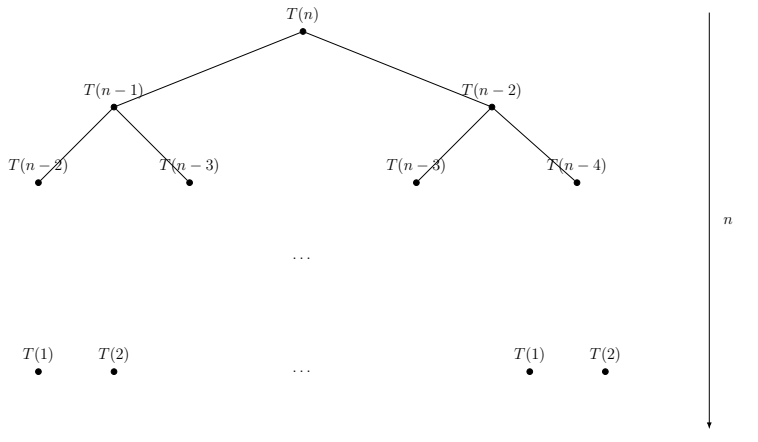
- $i = 0$. Then, j ranges in $[0, \ldots, 2^0) = [0, 1)$, i.e. the loop does only 1 iteration and 1 step (we assume the line 6 takes only 1 step as constants do not affect the overall complexity).
- $i = 1$. Then j ranges in $[0, \ldots, 2^1) = [0, 2)$, i.e. 2 steps.
- $i = 2$. Then j ranges in $[0, \ldots, 2^2) = [0, 4)$, i.e. 4 or 2^2 steps.
- $i = 3$. Then j ranges in $[0, \ldots, 2^3) = [0, 8)$, i.e. 8 or 2^3 steps.
- $i = n - 1$. Then j ranges in $[0, \ldots, 2^(n - 1))$, i.e. $2^(n - 1)$ steps.

Essentially, for each i the inner loop does 2^i steps. We need to find the sum of 2^i for all i from 0 to $n - 1$ to get the total number of steps both loops do. This is $2^0 + 2^1 + 2^2 + \ldots + 2^{n-1}$. The sum is equal to $2^n - 1$ because in binary the sum can be represented as n ones: $\underbrace{11\ldots11}_{n}$ (recall that number 7 in binary is 111 because $7 = 2^0 + 2^1 + 2^2 = 1 + 2 + 4$). Finally $\underbrace{11\ldots11}_{n} = 2^n - 1$, once again, $7 = 2^3 - 1 = 8 - 1$. Alternatively, one could use the formula for geometric series. Thus, the sum is equal to $2^n - 1$ where the dominant term is 2^n and the complexity is $\mathcal{O}(2^n)$. **There was a discussion at the tutorial that we can over approximate the number of iterations the inner loop does with 2^{n-1} for each i (as we saw above the number of iterations for each i is indeed always less than or equal to 2^{n-1}). And then argue that the total number of steps is $n2^{n-1}$. While this might be a reasonable thing to do under some circumstances, we are interested in the tightest upper bound and $2^n - 1$ is a more tight upper bound than $n2^{n-1}$ because for sufficiently large n, $n2^{n-1} > 2^n - 1$. You can see an example below in Figure 2.**

d This example is different from the others because it uses recursion rather than loops. So we need to somehow express the complexity recursively. Let's denote the number of steps the program does for n as input as $T(n)$. Then,

$$T(n) = T(n - 1) + T(n - 2) + 1$$

Where +1 is because we need to perform the addition and $T(n-1)+T(n-2)$ is because to compute $T(n)$ we first need to perform some steps in computing `fibonacci(n-1)` and `fibonacci(n-2)`. We also can note that $T(1) = T(2) = 1$ because we just return 1 in these cases. To find the number of steps for $T(n)$ we can unfold the recurrence. Unfolding the recurrence gives us the following tree (it essentially represents the tree of function calls).



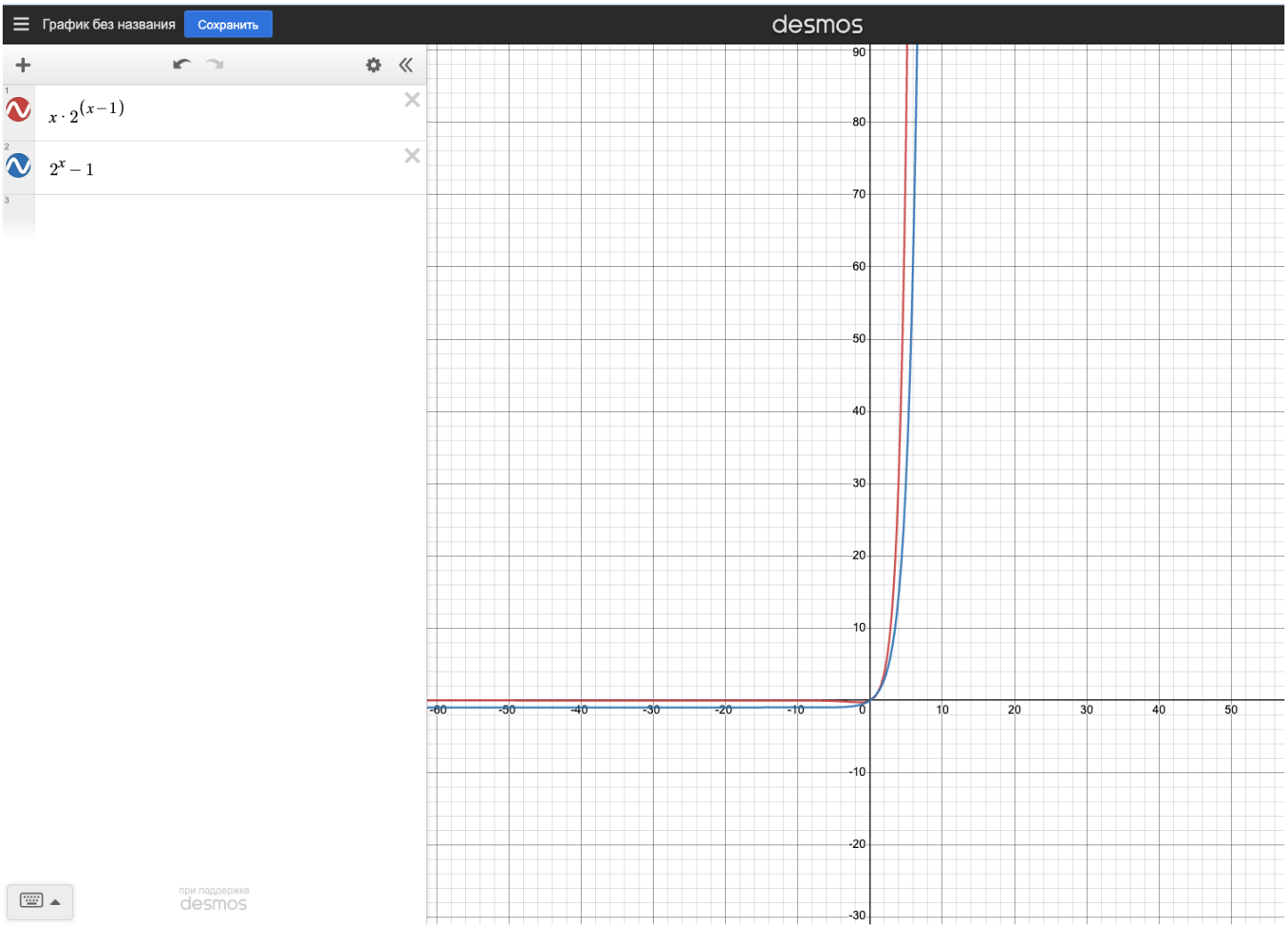


Figure 2: $n2^{n-1} > 2^n - 1$

The depth of the tree is n (you can check that by following the leftmost path from $T(n) to T(n)$) and at each level $i = [0, \ldots, n-1]$ we have 2^i nodes. Each of the leaf nodes takes 1 step to compute. All the others take the number of steps their children do and another 1 extra step. So to calculate the total number of steps we need to find the number of nodes in this tree. Because at each level we have 2^i nodes and i goes from $0 \to n-1$ we have $2^n - 1$ steps and the complexity is $\mathcal{O}(2^n)$.

Exercise 2. Using Shunting yard algorithm build an RPN representation of the following arithmetic expression.

$$3 + 4 \times (2 - 1 \times 2)$$

Then, evaluate the expression using stack.

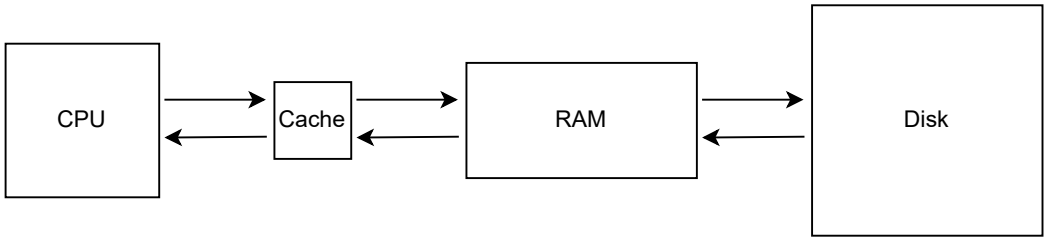
Solution The algorithm can be found here: `https://en.wikipedia.org/wiki/Shunting_yard_algorithm` and some other examples of its usage. I will present the trace of the algorithm in Table 1 for the example above.

You can try to evaluate the expression in the last row using stack to see if it evaluates to 3.

Step	Output	Stack	Input	Comment
1	Empty	Empty	$3 + 4 \times (2 - 1 \times 2)$	Initially both the stack and the output are empty
2	3	Empty	$+4 \times (2 - 1 \times 2)$	A number from the input goes right to the output
3	3	$+$	$4 \times (2 - 1 \times 2)$	Operand goes to the stack (the top of the stack is on the right)
4	3 4	$+$	$\times(2 - 1 \times 2)$	
5	3 4	$+ \times$	$(2 - 1 \times 2)$	
6	3 4	$+ \times ($	$2 - 1 \times 2)$	Note that left parenthesis also goes onto the stack
7	3 4 2	$+ \times ($	$-1 \times 2)$	
8	3 4 2	$+ \times (-$	$1 \times 2)$	
9	3 4 2 1	$+ \times (-$	$\times 2)$	
10	3 4 2 1	$+ \times (- \times$	$2)$	
11	3 4 2 1 2	$+ \times (- \times$	$)$	
12	3 4 2 1 2 $\times$ $-$	$+ \times$	Empty	When a right parenthesis is encountered, the stack is popped to the output until the matching left parenthesis is found, which is discarded from the stack (not put to the output)
13	3 4 2 1 2 $\times$ $-$ $\times$ $+$	Empty	Empty	The rest of the stack is popped to the output

Table 1: Run of Shunting yard algorithm

Exercise 3. A computer system has a cache, main memory, and a disk used for virtual memory. If a referenced word is in the cache, 20ns are required to access it. If it is in the main memory but not in cache, 60ns are needed to load it into cache (this includes the time to originally check the cache), and then the reference is started again. If the word is not in main memory, 12ms are required to fetch the word from disk, followed by 60ns to copy it to the cache, and then the reference is started again. The cache hit-ratio is 0.9 and the main memory hit-ratio is 0.6. What is the average time in ns required to access a referenced word on this system?



Solution. To find the average time we need to multiply the time to take some data from the particular memory by the probability that this particular memory is accessed. This is essentially the expected value of the access time: `https://en.wikipedia.org/wiki/Expected_value`. You can think of it as a dice with 6 sides. The expected value of a single dice roll is $1 \cdot \frac{1}{6} + 2 \cdot \frac{1}{6} + \ldots + 6 \cdot \frac{1}{6} = 3.5$ i.e. the sum of values on the dice side multiplied by the probability to roll this value. The probabilities and times can be found in Table 2. The average is then $20 \cdot 0.9 + 80 \cdot 0.06 + 12000080 \cdot 0.04 = 480026$ ns.

Exercise 4. Predict the Output of the following program.

```c
int main(){
    int x = 1;
    if (fork() == 0) {
        printf("Child has: %d", ++x);
    } else {
        printf("Parent has: %d", --x);
    }
    wait(NULL);
    return 0;
}
```

Solution. `fork()` creates a child process that has its own copy of variable `x` and each process can modify only its copy. Thus, the possible outputs are

```
Child has: 2 //++1 = 2
Parent has: 0 // --1 = 0
```

and

```
Parent has: 0 //--1 = 0
Child has: 2 // ++1 = 2
```

Because processes can execute in any order, for example, the parent process time quantum could expire at line 3 and the child process will then print first.

`wait(NULL)` is needed for the parent process to wait until all its children exit. We are checking the output of `fork()` to check in which process we are at the moment: for the parent process the function returns the child's process id and for the child, it returns 0.

Location	Probability	Time to access in ns	Comment
Cache	0.9	20ns	
RAM	0.6×0.1	20ns + 60ns	We access RAM only if the data is not in Cache which has probability 1 - 0.9. It takes 60ns to fetch from RAM to Cache and another 20ns to fetch from Cache
Disk	0.4×0.1	20ns + 60ns + 12000000ns	We access Disk only if the data is not in Cache and not in RAM which has probability of 1 - 0.9 and 1 - 0.6 respectively. Also note that 1ms = 1000000ns.

Table 2: Probabilities and times