Homework 1 - CS 170

Guilherme G. Haetinger
University of California, Berkeley

January 27, 2020

---

1 Study Group

None

2 Course Policies

- **What dates and times are the exams for CS170 this semester?**
  - Midterm 1 → *<2020-02-20 Thu>* 8-10pm
  - Midterm 2 → *<2020-03-17 Tue>* 8-10pm
  - Final → *<2020-05-15 Fri>* 7-10pm

- **Homework is due at 10:00pm, with a late deadline at 10:30pm. At what time do we recommend you have your homework finished?**
  10:00pm.

- **We provide 2 homework drops for cases of emergency or technical issues that may arise due to homework submission. If you miss the Gradescope late deadline (even by a few minutes) and need to submit the homework, what should you do?**
  There is no extensions!

- **What is the primary source of communication for CS170 to reach students? We will emailout all important deadlines through this medium, and you are responsible for checking your emails and reading each announcement fully.**
  Piazza.

- **Read all guidelines, etiquette and policies**
  I have read and understood the course syllabus and policies.
  Guilherme Gomes Haetinger

3 Understanding Academic Dishonesty

- **You ask a friend who took CS 170 previously for their homework solutions, some of which overlap with this semesters problem sets. You look at their solutions, then later writethem down in your own words.**
  Not OK

- **You had 5 midterms on the same day and are behind on your homework. You decide to ask your classmate, whos already done the homework, for help. They tell you how to do the first three problems.**
  Not OK

- **You look up a homework problem online and find the exact solution. You then write it in your words and cite the source.**
  OK

- **You were looking up Dijkstras on the internet, and run into a website with a problemvery similar to one on your homework. You read it, including the solution, and then youclose the website, write up your solution, and cite the website URL in your homework writeup.**
  OK

4 Asymptotic Complexity Comparison

4.1 Order the following functions so that for all i, j, if f_i comes before f_j in the order then $f_i = O(f_j)$. Do not justify your answers.

$f_3, f_7, f_2, f_5, f_9, f_4, f_8, f_6, f_1$

4.2 In each of the following, indicate whether $f = O(g)$,$f = (g)$, or both (in which case $f = \theta(g)$). Briefly justify each of your answers. Recall that in terms of asymptotic growth rate, logarithmic < linear < polynomial < exponential.

4.2.1 $log_3 n = \theta(log_4 n)$

As it is said on the textbook, the base is irrelevant on the big-O notation [DPV(2006), p. 22].

4.2.2 $nlog(n^4) = O(n^2 log(n^3))$

$nlog(n^4) = 4nlogn = O(n^2 log(n^3) = 3n^2 log(n))$

4.2.3 $\sqrt{n} = \Omega((log(n))^3)$

Log functions increase much slower than root operations. By that, we can see how $(log(n))^3 = log(n^{log(n^{log(n)})})$ grows in a slower rate than $\sqrt{n}$.

4.2.4 $n + log(n) = O(n + (log(n))^2)$

The only difference between the two is the squared log, which eventually will make the right-side of the equality will grow faster.

5 Computing Factorials

5.1 N is $log(N)$ bits long (this is how many bits are needed to store a number the size of N). Find an $f(N)$ so that $N!$ is $(f(N))$ bits long. Simplify your answer as much as possible, and give an argument for why it is true

As the book shows, the product of two numbers has at most the sum of its bits [DPV(2006), p. 24]. This means that the size of $N!$ is at most $\sum_{k=0}^{N} log(k)$. Now, by the Stirling's formula, we can say that, $f(n) = log(\sqrt{2\pi n}(\frac{n}{e})^n) \rightarrow size(N!) = \theta(f(n))$.

5.2 Give a simple (naive) algorithm to compute $N!$. You may assume that multiplying two bits together (e.g. 0x0,0x1) takes 1 unit of time.

1. Algorithm

```
def factorial(n)
  result = n
  n -= 1
  while n >= 1
    retult *= n
    n -= 1
  end
  return result
end
```

2. Proof of correctness

   We can see that the base case $n = 1$ works as it will simply ignore the loop. Now, let's say it works for n, if we input $n + 1$, the algorithm will run the loop one extra time, multiplying the value of `factorial(n)` by $n+1$, which is theta definition of the factorial functions, meaning that the functions holds for $n + 1$. By induction, the algorithm is proven to be correct.

3. Runtime Analysis

   The first two operations will always run, so we have 2 as the starting time. They loop will, then, run $n-1$ times, seen as it starts with $n = n-1$ and breaks when $n = 0$. The internal runtime for each loop is 2. This means that the algorithm will have a runtime complexity of $2+2(n-1) = 2+2n+2 = 2n+4$, which is linear.

6 Polynomial Evaluation

6.1 Describe a naive algorithm that, given $[a_0, ..., a_n]$ and x, computes $p(x)$. Give an analysis of its runtime as a function of the degree of the polynomial n.

6.1.1 Algorithm

```
def p(coef, x):
    # coef: list
    # x: list
    result = 0
    for k in range(0, len(coef)):
        result += coef[k] * x**k
    return result
```

6.1.2 Runtime Analysis

We can see that the loop will run n times. Although the sum is of complexity $O(1)$, when website power x to the k, we have k operations. This way, as k increases, the loop's internal complexity increases proportionally. Thus, the final runtime complexity is $\sum_{k=0}^{n}(1+k) = n + \sum_{k=0}^{n} k = n + (\frac{n(n+1)}{2}) = \frac{2n+n(n+1)}{2} = \frac{2n+n^2+n}{2} = \theta(n+n^2)$.

6.2 Describe and analyze an algorithm that evaluates the polynomial using the above expression. The runtime should be a function of n as well.

6.2.1 Algorithm

```
def p(coef, x):
    if len(coef) == 1:
        return coef[0]
    else:
        return coef[0] + x * p(coef[1:], x)
```

6.2.2 Proof of Correctness

Considering the correctness of the presented expression, we can state the following: for the base case $n = 1, [a], x$ as input, we have that the first condition will be true, returning the properly value; Let the algorithm hold for n, if we input $n+1$, theta algorithm is still correct because it simply encapsulates the past execution of n in a another recursion layer that will be called and multiplied by x. Therefore, by induction, the algorithm is correct. I.e. $p([a_0, ..., a_n], x) = a_0 + x(a_1 + x(a_2 + ... + x(a_{n1} + x * a_n)...)) \rightarrow p([a_0, ..., a_{n+1}], x) = a_0 + x(a_1 + x(a_2 + ... + x(a_n + x * a_{n+1})...)) = p([a_0, ..., a_n], x) + a_{n+1} * x$.

6.2.3 Runtime Analysis

We know there will be n recursion calls, in which there will be at most 2 operations of $O(1)$ complexity. This makes the function of linear complexity.

References

[DPV(2006)] DPV. *Algorithms*. 2006.