CSE103 Introduction to Algorithms

Midterm Exam

May 6th, 2021

You have 1 hour and 30 minutes.
No computer, laptop, phone or any Internet-connected device authorized.
You may use the supplementary material on recurrences.
Each question/subquestion is independent of the others and may be answered in any order.
The maximum score for the exam is 50 points. The number of points awarded by fully answering each question is written next to the question/subquestion itself.

Exercise 1: asymptotic notation (12 points)

True or false? Please give a brief justification of your answer.

1. **(2 points)** $n^2 + 10n + 6 = O(n^3)$
2. **(2 points)** $2^n + 42n^{1000} = \Theta(2^n)$
3. **(2 points)** $4\sqrt[4]{n} + \log n = O(\log n)$
4. **(2 points)** $37n^4 + n + 15 = \Omega(n^2)$
5. **(2 points)** if $f(n) = n2^{O(n)}$, then $\log f(n) = O(n)$
6. **(2 points)** $2^{\frac{n \log n}{2}} = O(2^n)$

Solution.

1. True, n^2 grows less quickly than n^3.
2. True: $2^n + 42n^{1000}$ is obviously bounded below by 2^n, and 2^n eventually bounds above any polynomial.
3. False: $\sqrt[4]{n}$ grows more quickly that $\log n$.
4. True: n^2 grows less quickly than n^4.
5. True: $f(n) = n2^{O(n)}$ means that there exist $m \geq 0$ and $c > 0$ such that, for all $n \geq m$, $f(n) \leq n2^{cn}$. Since log is monotonic, this implies that, for all $n \geq m$, $\log f(n) \leq \log(n2^{cn}) = \log n + \log 2^{cn} = \log n + cn \leq (1+c)n$, which proves that $\log f(n) = O(n)$.
6. False: see Exercise 1.8 of TD3.

Exercise 2: recurrences (12 points)

Give the best asymptotic upper bound you can to the following recurrences (where, in all cases, $T(0) = 1$). Please justify your answer.

1. **(3 points)** $T(n) = 2T\left(\frac{n}{3}\right) + 5n + \log n$
2. **(3 points)** $T(n) = 2T\left(\frac{n}{2}\right) + 2^n$
3. **(3 points)** $T(n) = 4T\left(\frac{n}{2}\right) + \frac{n^2}{\sin n + 2}$
4. **(3 points)** $T(n) = T\left(\frac{2n}{3}\right) + 1$

Solution.

1. Since $5n + \log n = \Theta(n)$, we apply the Master Theorem with $a = 2$, $b = 3$ and $c = 1$, and since $a < b^c$, we get $T(n) = \Theta(n)$.

2. The Master Theorem does not apply, so we try with the recursion tree method. It is a full binary tree, and very regular:
   - level 0 (the root) costs 2^n;
   - level 1 has 2 nodes costing $2^{\frac{n}{2}}$;
   - level 4 has 4 nodes costing $2^{\frac{n}{4}}$, and so on.

   So level k has 2^k nodes costing $2^{\frac{n}{2^k}}$ each. The cost of level k is therefore $2^k 2^{\frac{n}{2^k}} = 2^{k+\frac{n}{2^k}}$. The series for $n \to \infty$ does not converge, so we need to estimate the number of levels. This is very easy: the depth of the tree is $\log n$ because it is a full binary tree. So we have

$$T(n) \leq \sum_{k=0}^{\log n} 2^{k+\frac{n}{2^k}}$$

   From this, we may infer several upper bounds on $T(n)$, depending on how tightly we approximate the above sum. A first, very rough bound may be found as follows (knowing that $\frac{n}{2^k} \leq n$ for all k):

$$\sum_{k=0}^{\log n} 2^{k+\frac{n}{2^k}} \leq \sum_{k=0}^{\log n} 2^{k+n} \leq \sum_{k=0}^{\log n} 2^{\log n + n} = (\log n + 1) n 2^n \leq 4^n,$$

   for n large enough. So we may try with proving that $T(n) = O(g(n))$ where $g(n) := 4^n$. The induction step works easily:

$$T(n) = 2\,T\left(\frac{n}{2}\right) + 2^n \leq 2 \cdot 4^{\frac{n}{2}} + 2^n = 3 \cdot 2^n \leq 4^n$$

   which holds as long as $n \geq 3$. For the base case, let us find the value of $T(3)$:

$$\begin{aligned} T(0) &= 1 \\ T(1) &= 2T(0) + 2^1 = 4 \\ T(2) &= 2T(1) + 2^2 = 12 \\ T(3) &= 2T(1) + 2^3 = 16, \end{aligned}$$

   so we do have $T(3) \leq 4^3$, which proves that $T(n) = O(4^n)$ with constants $m = 3$ and $c = 1$.

   Another possibility is to take the better estimate $g(n) = n \log(n) 2^n$, which we obtained above before approximating everything to 4^n. This works too.

   Another, more refined bound is obtained by observing that

$$\sum_{k=0}^{\log n} 2^{k+\frac{n}{2^k}} \leq \sum_{k=0}^{\log n} 2^{k+n} = 2^n \sum_{k=0}^{\log n} 2^k = 2^n \frac{1 - 2^{\log n}}{1 - 2} = 2^n (2^{\log n} - 1) = 2^n (n-1).$$

   Let $g(n) := n2^n$ and let us try and prove that, for all n large enough, $T(n) \leq g(n)$ (that is, we are trying to use $c = 1$ as the big O constant). Let us start with the induction step:

$$T(n) = 2\,T\left(\frac{n}{2}\right) + 2^n \leq 2\frac{n}{2} 2^{\frac{n}{2}} + 2^n = n 2^{\frac{n}{2}} + 2^n = n 2^{-\frac{n}{2}} 2^n + 2^n.$$

   Since $2^{-\frac{n}{2}}$ goes to zero very quickly, we certainly have $n 2^{-\frac{n}{2}} \leq \frac{n}{2}$ for all n large enough: in fact, it is easy to see that this holds for all $n \geq 2$. If $n \geq 2$, we also have $\frac{n}{2} \geq 1$, hence $2^n \leq \frac{n}{2} 2^n$. Therefore, for all $n \geq 2$, we have

$$T(n) \leq n 2^{-\frac{n}{2}} 2^n + 2^n \leq \frac{n}{2} 2^n + \frac{n}{2} 2^n = n 2^n = g(n),$$

as desired. The first few values of $T(n)$ and $g(n)$ are:

$$\begin{aligned} T(0) &= 1 & g(0) &= 0 \\ T(1) &= 2T(0) + 2^1 = 4 & g(1) &= 2 \\ T(2) &= 2T(1) + 2^2 = 12 & g(2) &= 8 \\ T(3) &= 2T(1) + 2^3 = 16 & g(3) &= 24, \end{aligned}$$

so we see that taking $m = 3$ works for proving $T(n) = O(n2^n)$.

None of the above upper bounds is tight: in fact, it is possible to prove that $T(n) = \Theta(2^n)$. It is obvious that $T(n) = \Omega(2^n)$ (it is the root of the tree). The intuition for the upper bound is that, unlike most cases we've seen, in which the costs of the nodes of the recursion tree are polynomial in n, the weights here are exponential, and $2^{\frac{n}{2^k}}$ grows strictly more slowly than 2^n, for all $k > 0$ (technically, $2^{\frac{n}{2^k}} = o(2^n)$ for all $k > 0$). Therefore, *all* the costs in the tree are dominated by the root! This is *not* true when the costs are polynomial, because they are all of the same degree.

To confirm this intuition, we may show in the usual way that, for n large enough and for some $c > 0$ to be determined, $T(n) \leq c2^n$. Let us try the induction step, to see if we can determine c:

$$T(n) = 2\,T\left(\frac{n}{2}\right) + 2^n \leq 2c2^{\frac{n}{2}} + 2^n = (c2^{1-\frac{n}{2}} + 1)2^n.$$

If we manage to find c such that $c2^{1-\frac{n}{2}} + 1 \leq c$, we're happy. Solving the inequality, we get

$$c \geq \frac{1}{1 - 2^{1-\frac{n}{2}}}.$$

The quantity to the right of the inequality tends to 1 from above, and in fact, for all $n \geq 4$, we have

$$1 < \frac{1}{1 - 2^{1-\frac{n}{2}}} \leq 2.$$

So, if we take $c := 2$ and n big enough (at least $n \geq 4$), it will work! For the base case, we check a few further values of $T(n)$, and see that $T(4) = 40$ and $T(5) = 56$, whereas $2 \cdot 2^5 = 64$, so we may take $m := 5$ (this is fine because we said above that any $m \geq 4$ would be fine for the induction) and we have shown that $T(n) = O(2^n)$. This, combined with the obvious lower bound pointed out above, gives $T(n) = \Theta(2^n)$. If you managed to prove this, congratulations, you will get extra points!

3. We have that, for all $n \geq 0$, $1 \leq \sin n + 2 \leq 3$, so for all $n \geq 0$, $\frac{1}{3}n^2 \leq \frac{n^2}{\sin n + 2} \leq n^2$, which shows that $\frac{n^2}{\sin n+2} = \Theta(n^2)$. We may therefore apply the Master Theorem with $a = 4$, $b = 2$ and $c = 2$, and since $a = b^c$, we get $T(n) = \Theta(n^2 \log n)$.

4. Obviously, for all $n > 0$, $1 = n^0$ so we may apply the master theorem with $a = 1$, $b = \frac{3}{2}$ and $c = 0$, and since $a = b^c$, we get $T(n) = \Theta(\log n)$.

Exercise 3: a weird sorting algorithm (13 points)

Believe it or not, the following function sorts a list `l`:

```
def weirdSort(l):
    n = len(l)
    r = l
    if n == 2 and l[0] >= l[1]:
        r = [l[1],l[0]]
    elif n > 2:
        a = n // 3
        b = 2*a + (n % 3)
        r1 = weirdSort(l[0:b]) + l[b:n]
        r2 = r1[0:a] + weirdSort(r1[a:n])
        r = weirdSort(r2[0:b]) + r2[b:n]
    return r
```

Please answer the following questions, justifying your answer in each case:

1. **(8 points)** Is this algorithm more or less efficient than insertion sort? Justify your answer by a complexity analysis using one of the techniques you have learned.
2. **(5 points)** Suppose that the above function is modified by replacing the `if` condition at the third line with

   ```
   n == 2 and l[0][0] >= l[1][0]
   ```

   In this way, the function operates on lists of lists, and sorts them according to the value of the first element of the inner lists. For example, `weirdSort([[1,4],[2,3],[0,2]])` will return `[[0,2],[1,4],[2,3]]`.
   Recall that a sorting algorithm is *stable* if it preserves the relative position of elements with identical sorting key (in this case, the sorting key is the value of the first element of the inner lists). For example, when sorting `[[0,4],[2,3],[0,2]]`, a stable sorting algorithm must return `[[0,4],[0,2],[2,3]]`, because `[0,4]` comes before `[0,2]` in the original list, whereas an unstable sorting algorithm may return `[[0,2],[0,4],[2,3]]`, which is equally well sorted, but the position of the elements `[0,4]` and `[0,2]` has been swapped.
   Show, by means of an example, that `weirdSort` as modified above is *not* stable. Is it possible to modify it so that it becomes stable? How?

Solution.

1. By inspecting the code, we see that `weirdSort(l)` works as follows (we denote the length of `l` by n):
   - if $n \leq 1$, it returns `l` itself;
   - if $n = 2$, it swaps the two elements of `l` (if necessary) and returns the result;
   - if $n \geq 3$ or greater, the function calls itself three times:
     - the first time on a list of length `b`;
     - the second time on a list of length $n -$ `a`;
     - the third time on a list of length `b`.

   From the code, we see that `a` $= \frac{n}{3}$, so $n -$ `a` $= \frac{2}{3}n$, whereas `b` is equal to 2`a` plus a constant (at most equal to 2), so we may say that `b` $= \frac{2}{3}n$ as well. For the rest, the algorithm does only basic arithmetic (costing $\Theta(1)$) and slicing and concatenating lists of length n (costing $\Theta(n)$), so the total cost outside of the recursive calls is $\Theta(n)$.

   Therefore, the running time $T(n)$ of `weirdSort` obeys the recurrence

   $$T(n) = 3\,T\left(\frac{2}{3}n\right) + \Theta(n),$$

   which may be solved using the Master Theorem: $a = 3$, $b = \frac{3}{2}$ and $c = 1$, and since $a > b^c$, we have $T(n) = \Theta(n^{\log_{\frac{3}{2}} 3})$.

   Now, since $\left(\frac{3}{2}\right)^2 = \frac{9}{4} = 2 + \frac{1}{4} < 3$, we have that $\log_{\frac{3}{2}} 3 > 2$. Therefore, `weirdSort` is asymptotically *worse* than insertion sort (which is $\Theta(n^2)$).

2. An example showing that `weirdSort` is not stable is `[[0,1],[0,2]]`: when we execute the modified version of `weirdSort` with this input, the `if` condition is true (the length is 2 and the sorting

keys are equal), so the function swaps the two elements of the list and returns `[[0,2],[0,1]]`, whereas a stable sorting algorithm would have returned the list unchanged.

In order to make `weirdSort` stable, it is enough to replace the conditional at the third line with

```
if n == 2 and l[0][0] > l[1][0]:
```

that is, we test for *strict* inequality. In this way, elements with the same sorting key are never swapped, and they retain their original relative position.

Exercise 4: Russian multiplication (13 points)

The following is an algorithm for multiplying two *strictly positive* integer numbers `x` and `y`:

```
def mul(x, y, s=0):
    if x == 1:
        return s + y
    if x % 2 != 0:
        s += y
    return mul(x//2, y*2, s)
```

Known by some as "Russian multiplication", it is in fact a variant of a very old multiplication algorithm used by the ancient Egyptians. Notice that it only requires knowing how to add, multiply by 2, divide by 2, and recognizing whether a number is even or odd.

We will consider the following cost model, which is realistic when representing arbitrarily big numbers in base 2:

- addition of two n-digit numbers costs $\Theta(n)$;
- multiplying or dividing an arbitrary number by 2 costs $\Theta(1)$;
- verifying whether an arbitrary number is even or odd costs $\Theta(1)$.

Please answer the following questions, justifying your answer in each case:

1. **(5 points)** What is the worst-case asymptotic (big O) complexity of multiplying two n-digit numbers using Russian multiplication?
2. **(3 points)** What is the worst-case asymptotic (big O) complexity of executing `mul(1,y)` with respect to the number n of digits of `y`?
3. **(3 points)** What is the worst-case asymptotic (big O) complexity of executing `mul(x,1)` with respect to the number n of digits of `x`?
4. **(2 points)** The ancient Egyptians did not consider zero to be a number. What happens if the above algorithm is applied to `x = 0`? What about `y = 0`?

Solution.

1. The algorithm works as follows: it keeps track of an incremental sum `s`, and if the first input `x` is 1, it stops immediately returning `s + y`; otherwise, it updates the sum `s` by adding `y` to it only in case `x` is odd, and calls itself on inputs `x//2` and `y*2`. Notice that, albeit recursive, this is *not* a divide and conquer algorithm: if the number of digits of `x` is n, the number of digits of `x//2` is $n-1$, so we are decreasing the input size but not dividing it by something.

   So, on input two positive integers `x` and `y` both of length n, the algorithm will call itself n times before returing, and each time it will perform some operations of cost $O(1)$ and a sum `s + y`, the cost of which is $O(m)$, where m is the maximum of the number of digits of `s` and `y`. For the latter, it has n digits at the beginning and, at each iteration, the number of digits increases by 1 (because the recursive call is on `2*y`), so at any time the number of digits of `y` is bounded by $2n$. The number of digits of `s` increases progressively but, since `s` is always smaller than the result `x * y`, it is also at most $2n$, so $m \leq 2n = O(n)$ and we have that the total cost of each iteration is $O(n)$. Since, as pointed out above, we are doing n iterations, the total cost of Russian multiplication is $O(n^2)$.

2. When we execute Russian multiplication on input `x = 1` and an n-digit number `y`, we return immediately `0 + y`. We may imagine that addition is implemented so that adding zero terminates immediately (elementary school addition has this behavior, for example), so the cost is $O(1)$, *i.e.*, it is independent of `y`. Otherwise, if for some reason addition is "silly", then we would get a cost $O(n)$. Since this was not specified in the statement of the problem, both answers are fine.

3. When we execute Russian multiplication on input an n-digit number `x` and `y = 1`, we do n iterations during which we progressively "rebuild" `x` in `s`, by means of successive additions. The worst case is when `x`$= 2^n - 1$ (*i.e.*, its binary representation conists of n digits `1`). In that case, we perform an addition at every iteration, on numbers with more and more digits (1 digit the first time, 2 digits the second time, and so on), so the complexity is

$$O\left(\sum_{i=1}^{n} i\right) = O(n^2).$$

4. When we execute `mul(0,y)` (which is the same as `mul(0,y,0)`), we do not enter in any of the two `if` statements, and we call again `mul(0,y,0)`, so we enter an infinite loop and never terminate.

   When we execute `mul(x,0)` (which is the same as `mul(x,0,0)`), with `x` having n digits, we either do not change `s`, or we add `0` to it (so `s` stays `0` in both cases), and then we do a recursive call on `mul(x//2,0,0)`, so the length of the first input has decreased by 1. This means that we will do n recursive calls, during which `s` will always be `0`, and in the end, after $O(n)$ time, we will return `0`, which is the expected result.