CSE103 Introduction to Algorithms

Practice Exam for Midterm

April 29th, 2021

You have 1 hour and 30 minutes.[a]
No computer, laptop, phone or any Internet-connected device authorized.[b]
You may use the supplementary material on recurrences.
Each question/subquestion is independent of the others and may be answered in any order.[c]
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.

[a] Well, in reality you have 2 hours, but at the midterm you'll have only 1 hour and 30 minutes.
[b] Well, of course everything is allowed now, but it won't be at the midterm.
[c] Except 4.2, which depends on 4.1.

Exercise 1: asymptotic notation (12 points)

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

1. **(2 points)** $n^2 + 10n + 6 = \Omega(n)$
2. **(2 points)** $20\sqrt{n} + \log n = \Theta(\sqrt{n})$
3. **(2 points)** if $f(n) = O(n)$, then $2n \cdot f(n) = O(n^2)$
4. **(2 points)** $2n + 4\log^{10} n = O(\log^{11} n)$
5. **(2 points)** $2n \log^{17} n = O(n^2)$
6. **(2 points)** $3n \log n = \Theta(n^2)$

Solution.

1. True, n^2 grows more quickly than n.
2. True, $\log n$ is negligible with respect to $\sqrt{n}$.
3. True: since f is $O(n)$, we know that there exist $c > 0$ and $m \geq 0$ such that, for all $n \geq m$, $f(n) \leq cn$. Then, for all $n \geq m$, $2n\, f(n) \leq 2ncn = 2cn^2$, which proves the claim.
4. False: n grows more quickly than $\log^\alpha n$, for any α.
5. True: we certainly have $\log^{17} n = O(n)$, so we apply point 3 (or we prove it independently).
6. False: although $3n \log n = O(n^2)$ (same reasoning as point 5), n^2 grows more quickly than $3n \log n$, so $3n \log n \neq \Omega(n^2)$.

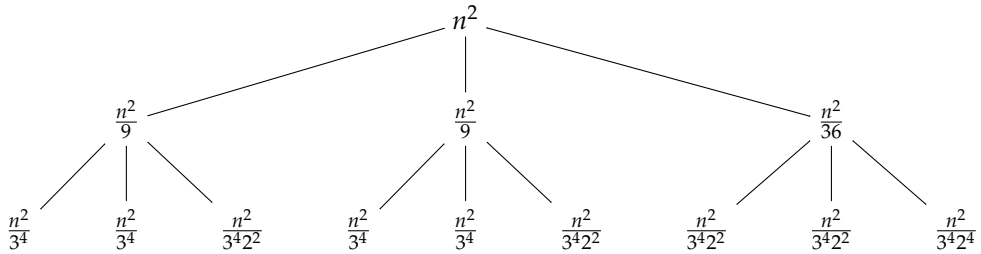
Exercise 2: recurrences (12 points)

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)** $T(n) = 2\,T\left(\frac{n}{3}\right) + T\left(\frac{n}{6}\right) + n^2$
2. **(3 points)** $T(n) = 5T\left(\frac{n}{2}\right) + 10n + 3n^2$
3. **(3 points)** $T(n) = 2T\left(\frac{n}{8}\right) + \frac{1}{n+1} + \sqrt[3]{n}$
4. **(3 points)** $T(n) = 3T\left(\sqrt{n}\right) + \log n$ (for this one, $T(1) = 1$ must also be assumed).

Solution.

1. We apply the recursion tree method. The first three levels of the recursion tree are



The sums of the first three levels are

$$\begin{aligned}
&\text{level 0:} \quad n^2 \\
&\text{level 1:} \quad \frac{n^2}{9} + \frac{n^2}{9} + \frac{n^2}{36} = \frac{4+4+1}{36} n^2 = \frac{9}{36} n^2 = \frac{n^2}{4} \\
&\text{level 2:} \quad \frac{n^2}{3^4} + \frac{n^2}{3^4} + \frac{n^2}{3^4 2^2} + \frac{n^2}{3^4} + \frac{n^2}{3^4} + \frac{n^2}{3^4 2^2} + \frac{n^2}{3^4 2^2} + \frac{n^2}{3^4 2^2} + \frac{n^2}{3^4 2^4} \\
&\qquad = \frac{2^4 + 2^4 + 2^2 + 2^4 + 2^4 + 2^2 + 2^2 + 2^2 + 1}{3^4 2^4} n^2 = \frac{2^6 + 2^4 + 1}{3^4 2^4} n^2 \\
&\qquad = \frac{3^4}{3^4 2^4} n^2 = \frac{n^2}{16}
\end{aligned}$$

We guess that the sum of level k is

$$S_k = \left(\frac{1}{4}\right)^k n^2.$$

We therefore have

$$T(n) < \sum_{k=0}^{\infty} S_k = \sum_{k=0}^{\infty} \left(\frac{1}{4}\right)^k n^2 = n^2 \sum_{k=0}^{\infty} \left(\frac{1}{4}\right)^k = n^2 \frac{1}{1-\frac{1}{4}} = \frac{4}{3} n^2.$$

We try and take $c := \frac{4}{3}$ as the big O constant. It works for the inductive case:

$$T(n) = 2\,T\Big(\frac{n}{3}\Big) + T\Big(\frac{n}{6}\Big) + n^2 \leq 2\frac{4}{3}\frac{n^2}{9} + \frac{4}{3}\frac{n^2}{36} + n^2 = \frac{8+1+36}{36} n^2 = \frac{27}{36} n^2 = \frac{4}{3} n^2.$$

For the base case, if we start trying the first few values of $T(n)$, we see that it never seems to work with the constant $\frac{4}{3}$ (we always have $T(n) > \frac{4}{3}n^2$ at least up to $n = 6$). But this is not a problem, because we may take a constant bigger than $\frac{4}{3}$: indeed, the inductive case tells us that, for n large enough, $T(n) \leq \frac{4}{3}n^2$, and we certainly have, for example, that $\frac{4}{3}n^2 \leq 4n^2$ for all n. This is easily works for the base case, because

$$T(1) = 2T(0) + T(0) + 1 = 2 \cdot 1 + 1 + 1 = 4 \leq 4 \cdot 1^2.$$

So we may take $c = 4$ and $m = 1$ as our constants proving that $T(n) = O(n^2)$. Since the root of the recursion tree is n^2, we actually have $T(n) = \Theta(n^2)$.

Another approach is to ignore the $\frac{4}{3}$ and keep the constant as a generic c. The inductive case gives us

$$\begin{aligned}
T(n) &= 2\,T\Big(\frac{n}{3}\Big) + T\Big(\frac{n}{6}\Big) + n^2 \leq 2c\frac{n^2}{9} + c\frac{n^2}{36} + n^2 = \frac{8c+c}{36} n^2 + n^2 = \\
&= \frac{c}{4} n^2 + n^2 = cn^2 - \left(\frac{3}{4}c - 1\right) n^2.
\end{aligned}$$

The last term is bounded by cn^2 as soon as

$$\frac{3}{4}c - 1 \geq 0$$

(because n^2 is always non-negative). This holds as long as $c \geq \frac{4}{3}$, putting us exactly in the situation above: we try the first few values of $T(n)$, we see that $T(1) = 4$, and since $4 > \frac{4}{3}$, we may take directly that as the constant.

2. Notice that $10n + 3n^2 = \Theta(n^2)$, so we may apply the Master Theorem with $a = 5$, $b = 2$ and $c = 2$, and since $a > b^c$, we have $T(n) = \Theta(n^{\log 5})$.

3. Notice that $\frac{1}{n+1} + \sqrt[3]{n} = \Theta(n^{\frac{1}{3}})$, so we may apply the Master Theorem with $a = 2$, $b = 8$ and $c = \frac{1}{3}$, and since $a = b^c$, we have $T(n) = \Theta(\sqrt[3]{n} \log n)$.

4. We apply a change of variables: we know that

$$T(2^m) = 3\,T\left(2^{\frac{m}{2}}\right) + m,$$

so if we let $S(m) := T(2^m)$, we have that S satisfies the recurrence

$$S(m) = 3\,S\left(\frac{m}{2}\right) + m.$$

This may be solved by applying the Master Theorem with $a = 3$, $b = 2$ and $c = 1$, which gives us $S(m) = \Theta(m^{\log 3})$, and so

$$T(n) = S(\log n) = \Theta(\log^{\log 3} n).$$

Exercise 3: two sorting algorithms (13 points)

The following general-purpose, comparison-based sorting algorithm is known as *odd-even sort*:

```
def oddEvenSort(l):
    isSorted = False
    c = 0
    while (not isSorted and c < len(l)):
        isSorted = True
        c += 1
        for i in range(0, len(l), 2):
            if i+1 < len(l) and l[i] > l[i+1]:
                l[i], l[i+1] = l[i+1], l[i]
                isSorted = False
        for i in range(1, len(l), 2):
            if i+1 < len(l) and l[i] > l[i+1]:
                l[i], l[i+1] = l[i+1], l[i]
                isSorted = False
```

The following algorithm is not comparison-based and works assuming that the input list `l` contains non-negative integers:

```
def timeSort(l):
    n = len(l)
    r = []
    t = [0] * n
    for i in range(n):
        t[i] = l[i]
```

```
    while len(r) < n:
        for i in range(n):
            if t[i] == 0:
                r.append(l[i])
            t[i] -= 1
    return r
```

It is called *time sort* because it acts as if each element were a "countdown" clock: each iteration decreases all clocks by 1, and when a clock reaches 0, its corresponding element is appended to the output.

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

1. **(2 points)** Which of the two above sorting algorithms is in place?
2. **(3 points)** How does odd-even sort compare to insertion sort? Justify your answer by analyzing the asymptotic complexity (big O) of the function `oddEvenSort`.
3. **(3 points)** What is the asymptotic complexity (big O) of executing `oddEvenSort(l)` on a list `l` of length n which is already sorted?
4. **(5 points)** Suppose that we are interested in sorting lists of arbitrary length n containing integers between 0 and 99. Which algorithm is asymptotically faster, merge sort or time sort? How does time sort compare to counting sort? Justify your answer by analyzing the complexity (big O) of the function `timeSort`.

Solution.

1. Odd-even sort is obviously in place: it only swaps the elements of `l`. Time sort is *not* in place, because it uses an auxiliary list `t` of variable length and builds the sorted list into a new list `r`.

2. Let n be the length of the input list `l`. Structurally, odd-even sort consists of a `while` loop containing two `for` loops, each executing instructions of cost $O(1)$. Both `for` loops are bounded by n, so their cost is $O(n)$. They are not nested, so their total cost is still $O(n)$. In the worst case, the `while` loop is executed n times, so the overall complexity of odd-even sort is $O(n^2)$. It is therefore asymptotically equivalent to insertion sort.

3. If the input is already sorted, the condition guarding the two `if` statements never becomes true, so `isSorted` is `True` at the end of the first execution of the `while` loop, which is therefore executed only once. The complexity is thus given by the two `for` loops and is $O(n)$.

4. Let n be the length of the input list `l`. The function `timeSort` consists of a `for` loop bounded by n, followed by a `while` loop containing another `for` loop bounded by n. The complexity of the first `for` loop (which is $O(n)$) is therefore dominated by the complexity of the `while` loop, so we may concentrate on the latter. In particular, we need to determine how many times the body of the `while` loop is executed. We see that the list `r` grows every time one of the elements of `t` reaches `0`; when every element of `t` is below or equal to `0`, `r` will be as long as `l` and we will exit the `while` loop. Now, each time we execute the `while` loop, *every* element of `t` is decreased by `1`. Since the elements of `t` are set to be initially equal to those of the input `l`, and since these are, by hypothesis, all at most equal to `99`, we have that the `while` loop will be executed at most 99 times. Since 99 is a constant (it does not depend on n), it is absorbed in the asymptotic notation and the complexity of time sort is just $O(n)$ (the complexity of the `for` loop inside the `while`).

   Therefore, for the task at hand (sorting lists containing integers between 0 and 99), time sort is asymptotically faster than merge sort. In fact, it is asymptotically equivalent to counting sort, of which it is basically a variant (both time sort and counting sort are based on the same principle).

Exercise 4: Maximum sublist (13 points)

Consider the following problem:

MAX SUBLIST
input: a list `l` of $n > 0$ integers (which may be negative);
output: two integers $0 \leq$ `i` $<$ `j` $\leq n$ maximizing the number `sum(l[i:j])`.

For example, on input `l = [0, -7, 6, 4, 6, 2, 4, -6, -7, -3]`, the pair `(2,7)` is a correct output, because `l[2:7] = [6, 4, 6, 2, 4]` has maximal sum (this is clear in this example because all negative numbers are avoided by this sublist).

The following function `findMax` implements a divide-and-conquer algorithm for MAX SUBLIST, using an auxiliary function `findMaxCross`:

```
def findMaxCross(l):
    n = len(l)
    h = n // 2
    m = sum(l[0:h])
    c = m
    i = 0
    for k in range(h-1):
        c -= l[k]
        if c > m:
            m = c
            i = k + 1
    m = sum(l[h:n])
    c = m
    j = n
    for k in range(n-1,h,-1):
        c -= l[k]
        if c > m:
            m = c
            j = k
    return (i,j)

def findMax(l):
    n = len(l)
    if n == 0:
        return None  # l shouldn't be empty
    if n == 1:
        return (0,1)
    h = n // 2
    (i,j) = findMax(l[0:h])
    m = sum(l[i:j])
    (i2,j2) = findMax(l[h:n])
    i2 += h
    j2 += h
    m2 = sum(l[i2:j2])
    if m2 > m:
        m = m2
        i = i2
        j = j2
    (i2,j2) = findMaxCross(l)
    m2 = sum(l[i2:j2])
    if m2 > m:
        m = m2
        i = i2
        j = j2
```

```
    return (i,j)
```

The idea of the algorithm is simple: the maximum sublist is either a sublist of the first half of the input, or a sublist of the second half of the input, or it crosses the middle of the input list. The first two cases are covered by the recursive calls, the third case is covered by the auxiliary function.

We consider the usual cost model in which every operation, comparison, etc. is $\Theta(1)$, *except* for the operation `sum(l)` and the slicing operation `l[i:j]`, which are $\Theta(n)$ where n is the length of `l`.

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

1. **(5 points)** What is the asymptotic big Θ complexity of `findMaxCross` with respect to the length n of its input list? (Notice the big Θ: your answer must be of the form $\Theta(f(n))$, which means that $f(n)$ is both an asymptotic upper bound (big O) and an asymptotic lower bound (big Ω) to the running time of `findMaxCross`).
2. **(8 points)** What is the worst-case asymptotic (big O) complexity of `findMax` with respect to the length n of its input list?

Solution.

1. Ignoring $\Theta(1)$ operations, `findMaxCross` consists of:
   - slicing the input list (to give a list of length $n/2$), which is of complexity $\Theta(n)$;
   - a `sum` on a list of length $n/2$, and therefore of complexity $\Theta(n)$;
   - a `for` loop bounded by $n/2$, and therefore of complexity $\Theta(n)$;
   - slicing again the input list (to give again a list of length $n/2$), which is of complexity $\Theta(n)$;
   - another `sum` on a list of length $n/2$, and therefore of complexity $\Theta(n)$;
   - another `for` loop bounded by $n/2$, and therefore of complexity $\Theta(n)$.

   One must not be deceived by the instructions of the form `m = sum(l[i:j])`, combining slicing with a sum: these are equivalent to a sequence of instructions

   ```
   l1 = l[i:j]
   m = sum(l1)
   ```

   the complexity of the first instruction is $\Theta(n)$ (if n is the length of `l`), and `l1` is of length bounded by n, so the complexity of the second instruction is again $\Theta(n)$, and the overall complexity is still $\Theta(n)$ (the instructions are one after the other, there is no nesting).

   So, the complexity of `findMaxCross` is $\Theta(n)$. Notice that we are allowed to write Θ (rather than just O) because:
   - for `sum` and slicing, it is a hypothesis of our cost model;
   - for the `for` loops, their execution is unconditional and always of the same length (they are `for` loops!).

2. Let T denote the running time of `findMax`. Ignoring $\Theta(1)$ operations, the execution of `findMax` on a non-singleton list (that is, $n > 1$) consists of:
   - a recursive call to `findMax` on a list of length $n/2$, therefore costing $T(n/2)$;
   - a `sum` operation on a list of length at most n, therefore costing $\Theta(n)$;
   - another recursive call to `findMax` on a list of length $n/2$, therefore costing $T(n/2)$;
   - another `sum` operation on a list of length at most n, therefore costing $\Theta(n)$;
   - a call to `findMaxCross` on a list of length n (`l` itself), which by the previous point we know costs $\Theta(n)$;
   - one last `sum` operation on a list of length at most n, therefore costing $\Theta(n)$.

   The running time of `findMax` thus satisfies the recurrence

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

   which may be solved by applying the Master Theorem with $a = 2$, $b = 1$ and $c = 1$, giving us $T(n) = \Theta(n \log n)$, which is of course also $O(n \log n)$.