CMSC 441: Unit 1 Homework Solutions

September 13, 2017

(A.1-1) Split-up the sum and apply the formula for an arithmetic series:

$$\begin{aligned}\sum_{k=1}^{n} 2k-1 &= 2\sum_{k=1}^{n} k - \sum_{k=1}^{n} 1 \\ &= 2\frac{n(n+1)}{2} - n \\ &= n(n+1) - n \\ &= n^2\end{aligned}$$

(A.1-6) The left hand side of the equation is to be interpreted as a sum of n functions $g_k(i)$, the k^{th} of which is in $O(f_k(i))$. Therefore, for each $k = 1, 2, \ldots, n$, there exists positive constants c_k and $n_{0,k}$ such that if $n \geq n_{0,k}$ then

$$g_k(i) \leq c_k f_k(i).$$

Let n_0 be the maximum of the $n_{0,k}$, $k = 1, 2, \ldots, n$ so that if $n \geq n_0$, all n of the inequalities above are satisfied. Further, let c the maximum of c_k for $k = 1, 2, \ldots, n$. Then

$$\sum_{k=1}^{n} g_k(i) \leq \sum_{k=1}^{n} c_k f_k(i) \leq c\sum_{k=1}^{n} f_k(i).$$

Define $f(i) = \sum_{k=1}^{n} f_k(i)$; then the above reduces to

$$\sum_{k=1}^{n} g_k(i) \leq cf(i),$$

showing that $\sum_{k=1}^{n} g_k(i) = O(f(i))$ and therefore

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

(2.1-3) Pseudocode:

LINEAR-SEARCH(A, v)

```
1  n = A.length
2  r = NIL
3  for i = 1 to n
4       if A[i] == v
5            r = i
6  return r
```

Invariant: r is NIL unless v is contained in $A[1\,..\,(i-1)]$, in which case $A[r] = v$.

Note: By convention, $A[i\,..\,j] = \emptyset$ if $j < i$. In particular, $A[1\,..\,0] = \emptyset$.

Initialization: $i = 1$, but before the test of $i > n$, $A[1\,..\,(i-1)] = A[1\,..\,0]$ which is empty, so v can not be in $A[1\,..\,(i-1)]$ and r is set to NIL, so the invariant holds.

Maintenance: if $A[i] = v$, then r is set to i; if $A[i] \neq v$ and v is not in $A[1\,..\,(i-1)]$, then r is still NIL. So, at the beginning of the next iteration i is incremented to $i+1$ and $r = \text{NIL}$ if v is not in $A[1\,..\,i]$ or $A[r] = v$.

Termination: The loop terminates with $i = n+1$, which case r is NIL if v does not occur in $A[1\,..\,n]$; otherwise, $A[r] = v$.

(2.1-4) Pseudocode:

ADD(A, B, n)

```
1  C is a new array of length n + 1
2  C[1] = (A[1] + B[1]) mod 2
3  carry = (A[1] + B[1]) div 2
4  for i = 2 to n
5       C[i] = (A[i] + B[i] + carry) mod 2
6       carry = (A[i] + B[i] + carry) div 2
7  C[n + 1] = carry
8  return C
```

Invariant: At the start of the i^{th} iteration, $C[1\,..\,(i-1)] = (A[1\,..\,(i-1)] + B[1\,..\,(i-1)])$ mod 2^{i-1}. The variable $carry = 1$ if $(A[1\,..\,(i-1)] + B[1\,..\,(i-1)])$ div 2^{i-1} is one; otherwise $carry = 0$.

Initialization: $i = 2$, $C[1] = (A[1] + B[1])$ mod 2^1, and $carry = (A[1] + B[1])$ div 2^1. Therefore, the invariant holds.

Maintenance: At the start of the iteration, $C[1\,..\,(i-1)]$ contains the low $i-1$ bits of $A[1\,..\,(i-1) + B[1\,..\,(i-1)]$ and $carry$ contains the carry bit. $C[i]$ is set to the i^{th} bit of A plus the i^{th} bit of B, plus the carry from the sum of the low $i-1$ bits, so $C[1\,..\,i]$ now contains the low i bits of $A[1\,..\,i] + B[1\,..\,i]$. Moreover, $carry$ contains the carry bit from the addition of $A[1\,..\,i]$ and $B[1\,..\,i]$.

Termination: The loop terminates with $i = n+1$, so $C[1\,..\,n]$ contains the low n bits of $A[1\,..\,n] + B[1\,..\,n]$ and *carry* contains the carry bit, which is saved in $C[n+1]$. Therefore, C contains the full $n+1$-bit sum of A and B.

(3.1-1) We need to show that there exists positive constants c_1, c_2, and n_0 such that $n > n_0$ implies that

$$c_1(f(n) + g(n)) \leq \max(f(n), g(n)) \leq c_2(f(n) + g(n)).$$

First, since $f(n)$ and $g(n)$ are asymptotically non-negative, assume n_0 is large enough that $f(n)$ and $g(n)$ are non-negative for $n > n_0$. Now, by definition, $\max(f(n), g(n))$ is greater than or equal to both $f(n)$ and $g(n)$ and therefore is greater than or equal to their average, $\frac{1}{2}(f(n)+g(n))$, so we may choose $c_1 = \frac{1}{2}$. Since for any particular value of n, $\max(f(n), g(n))$ is equal to either $f(n)$ or $g(n)$, it is less than or equal to the sum $f(n) + g(n)$, and so we may choose $c_2 = 1$. We conclude that $\max(f(n), g(n))$ is in $\Theta(f(n) + g(n))$.

(3.1-3) Big-Oh notation gives an upper bound on running time, not a lower bound. If the speaker means that the running time of Algorithm A is bounded below by cn^2, then they would say that it is at best $\Omega(n^2)$; if on the other hand, they mean to state an upper bound, it would make sense to say that the running time of algorithm A is *at most* $O(n^2)$.

(3.1-4) $2^{n+1} = 2 \cdot 2^n$, so it is $\Theta(2^n)$ with constants $c_1 = c_2 = 2$. 2^{2n} is *not* $O(2^n)$. Suppose it were; then there would exists positive constants c and n_0 such that $n > n_0$ implies $2^{2n} < c2^n$. Suppose $n > n_0$, then dividing by 2^n gives

$$\frac{2^{2n}}{2^n} = 2^n < c,$$

which is a contradiction since the exponential function is unbounded.

(3-2)

	A	B	O	o	Ω	ω	Θ	Comment
a.	$\lg^k n$	n^ϵ	yes	yes	no	no	no	See p. 57
b.	n^k	c^n	yes	yes	no	no	no	See p. 55, eq. 3.10
c.	$\sqrt{n}$	$n^{\sin n}$	no	no	no	no	no	$n^{\sin n}$ oscillates between $1/n$ and n.
d.	2^n	$2^{n/2}$	no	no	yes	yes	no	$\lim_{n\to\infty} \frac{2^n}{2^{n/2}} = \infty$.
e.	$n^{\lg c}$	$c^{\lg n}$	yes	no	yes	no	yes	$n^{\lg c} = c^{\lg n}$
f.	$\lg n!$	$\lg n^n$	yes	no	yes	no	yes	See p. 58, eq. 3.19

(4.1-3) My Python implementation, running on my Macbook, crosses-over when the input array length is around 65. For small arrays, e.g. length 10, brute

force is the clear winner. Modifying the divide-and-conquer implementation to call the brute force algorithm for small problems results in an implementation that is nearly as fast as brute force for small problems but has the asymptotic performance of the divide-and-conquer solution.

```
#!/usr/bin/env python

import time


# Maximum Subarray - linear-time algorithm

def msa_lin(x):

    # Initialize with MSA and max prefix equal x[0]

    max_i = 0
    max_j = 0
    max_sa = x[0]
    max_pi = 0
    max_p = x[0]

    # Iterate over x[1], x[2], ..., x[n-1]

    for i in range(1, len(x)):

        val = x[i]

        # If x[i] <= 0, only need to update max prefix

        if val <= 0:
            max_p = max_p + val

        # Otherwise, if x[i] > 0, we need to determine if there
        # is a new MSA ending at x[i]

        else:

            # If max prefix is positive, check if x[i] + max prefix
            # is the new MSA

            if max_p > 0: # x[i] > 0 and max prefix > 0

                if val + max_p > max_sa:  # new msa
                    max_i = max_pi
                    max_j = i
```

```
                    max_sa = val + max_p

                # update max prefix in either case

                max_p = val + max_p

            else:  # val > 0, max_p <= 0

                if val > max_sa:  # new msa
                    max_i = i
                    max_j = i
                    max_sa = val

                # since max prefix was negative and x[i] > 0, set
                # beginning of max prefix to i and set the value
                # of the max prefix to x[i]

                max_pi = i
                max_p = val

    return max_sa, max_i, max_j


# Maximum Subarray - brute-force algorithm

def msa_bf(x):
    n = len(x)
    max_sum = x[0]
    range_sum = x[0]
    max_i = 0
    max_j = 0
    for i in range(1, n):
        range_sum = range_sum + x[i]
        subrange_sum = range_sum
        for j in range(i+1):
            if subrange_sum > max_sum:
                max_sum = subrange_sum
                max_i = i
                max_j = j
            subrange_sum = subrange_sum - x[j]
    return max_sum, max_j, max_i


#
# Maximum Subarray - divide-and-conquer algorithm
```

```
def msa_dac(x):

# Basic divide-and-conquer
    i, j, max =  find_max_subarray(x, 0, len(x)-1)
    return max, i, j

# Divide-and-conquer w/brute force for small problems
#       small_problem_threshold = 65
#       if len(x) < small_problem_threshold:
#           return msa_bf(x)
#       else:
#           i, j, max =  find_max_subarray(x, 0, len(x)-1)
#           return max, i, j

def find_max_subarray(x, low, high):
    if high == low:
        return low, high, x[low]
    else:
        mid = (low + high) / 2
        left_low, left_high, left_sum = \
            find_max_subarray(x, low, mid)
        right_low, right_high, right_sum = \
            find_max_subarray(x, mid+1, high)
        cross_low, cross_high, cross_sum = \
            find_max_crossing_subarray(x, low, mid, high)
        if left_sum >= right_sum and left_sum >= cross_sum:
            return left_low, left_high, left_sum
        elif right_sum >= left_sum and right_sum >= cross_sum:
            return right_low, right_high, right_sum
        else:
            return cross_low, cross_high, cross_sum


def find_max_crossing_subarray(x, low, mid, high):
    left_sum = float("-inf")
    sum = 0
    for i in range(mid, low-1, -1):
        sum = sum + x[i]
        if sum > left_sum:
            left_sum = sum
            max_left = i
    right_sum = float("-inf")
    sum = 0
    for j in range(mid+1, high+1):
        sum = sum + x[j]
        if sum > right_sum:
```

```
            right_sum = sum
            max_right = j
    return max_left, max_right, left_sum + right_sum


if __name__ == "__main__":

    # Datat used in textbook
    # x = [13, -3, -25, 20, -3, -16, -23, 18, 20, -7, 12, -5, -22, 15, -4, 7]

    data_size = 65
    num_tests = 1000

    x = range(1,data_size)

    t1 = time.time()
    for i in range(num_tests):
        msa_bf(x)
    t = time.time() - t1
    print "bf_timing:__" + str(t/num_tests)

    t1 = time.time()
    for i in range(num_tests):
        msa_dac(x)
    t = time.time() - t1
    print "dac_timing:_" + str(t/num_tests)

    t1 = time.time()
    for i in range(num_tests):
        msa_lin(x)
    t = time.time() - t1
    print "lin_timing:_" + str(t/num_tests)
```

(4.3-1) Use induction to prove that $T(n) = T(n-1) + n$ is $O(n^2)$.

Base case: We can assume $T(n)$ is bounded by a contstant for small n. In particular, $T(1) \leq c = c \cdot 1^2$.

Inductive Hypothesis: Let $n > 1$ and assume that, for all integers k, $1 \leq k < n$,

$$T(k) \leq ck^2.$$

Inductive step: Show that $T(n) \leq cn^2$.

$$\begin{aligned} T(n) &= T(n-1) + n \\ &\leq c(n-1)^2 + n \end{aligned}$$

$$\leq \quad cn^2 \text{ so long as } c > 1/2.$$

To see that the last inequality is true, observe that

$$c(n-1)^2 + n = cn^2 - 2cn + c + n = cn^2 + (1-2c)n + c \leq cn^2$$

so long as $c > 1/2$ and $n > c/(2c-1)$.

(4.3-2) Since the recurrence relation divides the data size by two with each recursive call, and the cost at each level is constant, it is a good guess that the running time is $O(\lg n)$.

We need to show that $T(n) \leq c \lg n$ for some constant c and $n \geq n_0$. When $n = 1$, $\lg n = 0$, so we can't satisfy the inequality in this case. However, for $n = 2$, $\lg n = 1$, and so we can choose $c \geq T(2)$.

Now, suppose that for some $n > 2$ we have that for all k, $2 \leq k < n$, $T(k) \leq c \lg k$. Furthermore, suppose n is even. Then

$$\begin{aligned} T(n) &= T(\lceil n/2 \rceil) + 1 \\ &= T(n/2) + 1 \\ &\leq c \lg(n/2) + 1 \\ &= c \lg n - c \lg 2 + 1 \\ &= c \lg n - c + 1 \end{aligned}$$

So long as $c \geq 1$, we can conclude that $T(n) \leq c \lg n$. If n is odd, then

$$\begin{aligned} T(n) &= T(\lceil n/2 \rceil) + 1 \\ &= T((n+1)/2) + 1 \\ &\leq c \lg((n+1)/2) + 1 \\ &= c \lg(n+1) - c \lg 2 + 1 \end{aligned}$$

What can be do about the $\lg(n+1)$ term? It turns out that by choosing n sufficiently large, we can make $\lg(n+1) < \lg n + \epsilon$ for any $0 < \epsilon < 1$. To see this, consider the sequence of equivalent inequalities:

$$\begin{aligned} \lg(n+1) &< \lg n + \epsilon \\ \lg((n+1)/n) &< \epsilon \\ (n+1)/n &< 2^\epsilon. \end{aligned}$$

We have that $\epsilon > 0$, so $2^\epsilon > 1$; since $\lim_{n\to\infty}(n+1)/n = 1$, we can make $(n+1)/n$ smaller than 2^ϵ by choosing n sufficiently large. This gives us

$$\begin{aligned} T(n) &\leq c \lg(n+1) - c \lg 2 + 1 \\ &\leq c(\lg n + \epsilon) - c + 1 \\ &= c \lg n - (c - \epsilon) + 1 \\ &\leq c \lg n \end{aligned}$$

so long as $c - \epsilon > 1$ or $c > 1 + \epsilon$.

We have shown that $T(n) \leq c \lg n$ and so $T(n) = O(\lg n)$.

(4.3-7) Suppose that for $m < n$, $T(m) \leq cn^{\log_3 4}$. Then

$$\begin{aligned} T(n) &\leq 4c(n/3)^{\log_3 4} + n \\ &= 4cn^{\log_3 4}/4 + n \\ &= cn^{\log_3 4} + n. \end{aligned}$$

We are left with a n term, and so the proof will not work. Make a stronger inductive hypothesis by subtracting a linear term,

$$T(m) \leq cm^{\log_3 4} - dm,$$

where c and d are positive constants. Then

$$\begin{aligned} T(n) &\leq 4(c(n/3)^{\log_3 4} - dn) + n \\ &= 4cn^{\log_3 4}/4 - 4dn + n \\ &= cn^{\log_3 4} - (4d - 1)n \\ &\leq cn^{\log_3 4} - dn \end{aligned}$$

so long as $4d - 1 \geq d$, or $d \geq 1/3$.

(4.4-2) Since the coefficient $a = 1$, there is only a single recursive call, and the tree consists of only a trunk. The cost at the top node is n^2, at the second, $(n/2)^2$, etc., and we see that the cost at level j, with $j = 0$ the top node, is $n^2/4^j$. Therefore, the total cost is

$$n^2 + n^2/4 + \cdots + n^2/4^{\lg n} = n^2 \sum_{j=0}^{\lg n} (1/4)^j \leq n^2(4/3)$$

Therefore, the estimated cost is $O(n^2)$.

Base case: this is easy since we can always "bump up" c if we need to. Assume c is large enough that $T(1) \leq c$.

Inductive step: Suppose $T(k) \leq ck^2$ for all $1 \leq k < n$.

$$\begin{aligned} T(n) &= T(n/2) + n^2 \\ &\leq c(n/2)^2 + n^2 \\ &= (c/4)n^2 + n^2 \\ &= (c/4 + 1)n^2. \end{aligned}$$

Is this last term $\leq cn^2$? $c/4 + 1 \leq c$ iff $c \geq 4/3$. If we insist that $c \geq 4/3$, which we are allowed to do, we conclude that $T(n) \leq cn^2$ and $T(n) = O(n^2)$.

(4.4-4) The tree should be a balanced binary tree with cost 1 at each node, and with the bottom level corresponding to $n = 1$, i.e. $T(1)$. Therefore, there are n levels; the first has cost 1, the second has cost 2, the third has cost 4, etc., down to the last level with cost 2^{n-1}. Therefore, the total cost is

$$1 + 2 + 4 + \cdots + 2^{n-1} = \sum_{i=0}^{n-1} 2^i = 2^n - 1.$$

Therefore, the estimated cost is $O(2^n)$.

Base case: $T(1) = 1 \leq 2^1$.

Inductive step: Suppose $T(k) \leq c2^k - d$ for all $1 \leq k < n$. Show that $T(n) \leq c2^n - d$.

$$\begin{aligned} T(n) &= 2T(n-1) + 1 \\ &\leq 2(c2^{n-1} - d) + 1 \\ &= c2^n - 2d + 1 \\ &\leq c2^n - d \end{aligned}$$

so long as $d \geq 1$.

(4.5-1) (a) $T(n) = 2T(n/4) + 1$. $a = 2$, $b = 4$, and $\log_4 2 = 1/2$. $f(n) = 1$ is $O(n^{1/2-\epsilon})$ for $0 < \epsilon \leq 1/2$, so case (i) applies; therefore $T(n) = \Theta(n^{1/2})$.

(b) $T(n) = 2T(n/4) + \sqrt{n}$. $f(n) = \Theta(n^{1/2})$, so by case (ii), $T(n) = \Theta(\sqrt{n} \lg n)$.

(c) $T(n) = 2T(n/4) + n$. $f(n) = n$ is $\Omega(n^{1/2+\epsilon})$ for $0 < \epsilon \leq 1/2$. Also, $2f(n/4) = n/2 \leq (1/2)n$. Therefore, by case (iii), $T(n) = \Theta(n)$.

(d) $T(n) = 2T(n/4) + n^2$. $f(n) = n^2$ is $\Omega(n^{1/2+\epsilon})$ for $0 < \epsilon \leq 3/2$. Also, $2f(n/4) = n^2/8 \leq (1/8)n^2$. Therefore, by case (iii), $T(n) = \Theta(n^2)$.

(4.5-3) $T(n) = T(n/2) + \Theta(n)$. We are given that $f(n) = \Theta(n)$. We see that $a = 1$ and $b = 2$, so $\log_b a = \lg 1 = 0$. Since $f(n) = \Theta(n)$, $f(n) \neq \Theta(n^{-\epsilon})$ for any $\epsilon > 0$. Therefore, case 1 of the MT does not apply. However, we see that case 2 does apply, and we conclude that $T(n) = \Theta(n^0 \lg n) = \Theta(\lg n)$.

(4-1)

(a) $a = b = 2$, so $\log_b a = 1$ and $n^{\log_b a} = n$. $f(n) = n^4 = \Omega(n^{1+\epsilon})$. Lastly,

$$af(n/b) = 2\left(\frac{n}{2}\right)^4 = \frac{1}{8}n^4 = \frac{1}{8}f(n),$$

so $T(n) = \Theta(n^4)$ by part 3 of the Master Theorem.

(b) $a = 1$, $b = 10/7$, so $\log_b a = 0$ and $n^{\log_b a} = 1$. $f(n) = n = \Omega(n^{\epsilon})$. Lastly,

$$af(n/b) = \frac{7}{10}n = \frac{7}{10}f(n),$$

so $T(n) = \Theta(n)$ by part 3 of the Master Theorem.

(c) $a = 16$, $b = 4$, so $\log_b a = 2$ and $n^{\log_b a} = n^2$. $f(n) = n^2 = \Theta(n^2)$, so $T(n) = \Theta(n^2 \lg n)$ by part 2 of the Master Theorem.

(d) $a = 7$, $b = 3$, so $\log_b a = \log_3 7 \approx 1.7712$, and $n^{\log_b a} \approx n^{1.7712}$. $f(n) = n^2 = \Omega(n^{1.7712+\epsilon})$. Lastly,

$$7f(n/3) = 7\left(\frac{n}{3}\right)^2 = \frac{7}{9}n^2 = \frac{7}{9}f(n),$$

so $T(n) = \Theta(n^2)$ by part 3 of the Master Theorem.

(e) This is almost the same as (d), except $b = 2$ and $\log_b a \approx 2.8074$ so $n^{\log_b a} \approx n^{2.8074}$. $f(n) = n^2 = O(n^{2.8074-\epsilon})$, so $T(n) = \Theta(n^{\log_2 7})$ by part 1 of the Master Theorem.

(f) $a = 2$, $b = 4$, so $\log_b a = \frac{1}{2}$ and $n^{\log_b a} = n^{1/2} = \sqrt{n}$. $f(n) = \sqrt{n} = \Theta(\sqrt{n})$, so $T(n) = \Theta(\sqrt{n} \lg n)$ by part 2 of the Master Theorem.

(g) We don't use the Master Theorem for this one. Write out a recursion tree: you should have $n/2$ levels with costs $n^2, (n-2)^2, (n-4)^2, \ldots$ ending with either $2^2 = 4$ or $1^2 = 1$ depending on whether n is even or odd, respectively. We see that if n is odd, the running-time is the sum of the squares of the first $n/2$ odd numbers, and if n is even, it is the sum of the squares of the first $n/2$ even numbers. There are closed formed expressions for both of these quantities. If n is even,

$$\sum_{k=1}^{n/2} (2k)^2 = \frac{n(n+1)(n+2)}{24},$$

and if n is odd,

$$\sum_{k=1}^{(n+1)/2} (2k-1)^2 = \frac{n(n+1)(n+2)}{6}.$$

We see that both of these quantities are $\Theta(n^3)$ and so we guess that $T(n) = \Theta(n^3)$.

At this point it is a straight-forward substitution (induction) proof to show that $T(n) = O(n^3)$ and, similarly, $T(n) = \Omega(n^3)$, proving that $T(n) = \Theta(n^3)$.