# CPSC5210 Fall 2018 Midterm A

Jamion Williams

October 9, 2018

## 1

**(15 points) Given an recurrence algorithm whose time complexity is recursively formulated as $T(n) = 4T\frac{n}{2} + 5n^2$...**

### 1.1 (5 points)Use the master theorem to find out its asymptotic time complexity.

REMEMBERING that the Master Theorem dictates that

$$T(n) = aT(\frac{n}{b}) + f(n)$$

...we can take the recursive problem

$$T(n) = 4T\frac{n}{2} + 5n^2$$

...and assign $a = 4$, $b = 2$, and, remembering that we ignore constants within the $f(n)$ of the Master Theorem, $f(n) = n^2$.

AND FURTHER remembering that the Master Theorem compares $n^{log_b\ a}$ against $f(n)$ as follows:

CASE 1) If $n^{log_b\ a}$ is polynomially larger than $f(n)$, then $T(n) = \theta(n^{log_b\ a})$

CASE 2) If $n^{log_b\ a}$ is equal to $f(n)$, then $T(n) = \theta(f(n)\ lg\ n)$

CASE 3) If $f(n)$ is polynomially larger than $n^{log_b\ a}$, then $T(n) = \theta(f(n))$

...we can see that:

$$n^{log_b\ a} \stackrel{?}{=} f(n)$$

$$n^{log_2\ 4} \stackrel{?}{=} n^2$$

$$\therefore n^2 = n^2$$

THEREFORE, our CASE TWO applies, meaning that the aymptotic tight bounds of $T(n) = 4T\frac{n}{2} + 5n^2$ are:

$$f(n) = \theta(n^2\ log_2\ n)$$

### 1.2 (10 points) Show the analytical time complexity using term-elimination method.

USING the elimination method, we aim to drive towards $T(1)$ by making the left hand side of the equation cancel out a portion of the right hand side. Therefore, the starting equation

$$T(n) = 4T\frac{n}{2} + 5n^2$$

...in which, again–considering the nature of Time Complexity–we can ignore the constant accompanying $f(n)$, therefore

$$T(n) = 4T\frac{n}{2} + n^2$$

...has iterations as follows:

$$\begin{aligned}
iteration\ 0\ &-\ [T(n) = 4T\frac{n}{2} + n^2]4^0 \\
iteration\ 1\ &-\ [T(\frac{n}{2^1}) = 4T\frac{n}{2^2} + (\frac{n}{2^1})^2]4^1 \\
iteration\ 2\ &-\ [T(\frac{n}{2^2}) = 4T\frac{n}{2^3} + (\frac{n}{2^2})^2]4^2 \\
&... \\
iteration\ k-1\ &-\ [T(\frac{n}{2^{k-1}}) = 4T\frac{n}{2^k} + (\frac{n}{2^{k-1}})^2]4^{k-1} \\
iteration\ k\ &-\ T(1) = 1
\end{aligned}$$

...where the $T$ term on the left hand side of each iteration cancels the $T$ term on the preceding iteration's right hand side. Keeping this in mind, we can now add up the non-eliminated terms to either discover a summation or reduce to closed-form, as follows:

$$n^2 + 4^1(\frac{n}{2^1})^2 + 4^2(\frac{n}{2^2})^2 +\ ...\ + 4^{k-1}(\frac{n}{2^{k-1}})^2$$

...which can be reduced to

$$n^2 + n^2 + n^2 +\ ...\ n^2$$

...in which the initial term $n^2$ is repeated $k$ times, therefore:

$$n^2 * k$$

...because we can assume, without loss of generality, that $n = 2^k$ and $k = log_2\ n$, we can close to an Analytical Time Complexity of:

$$n^2\ log\ n$$

Therefore:

$$f(n) \in O(n^2\ log\ n)$$

## 2

(20 points) Considering the following Java method:

```
public static void foo(int n, char A, char B, char C) {
     if (n<=0) return; // primitive operation
     foo(n-2, A, C, B);
     System.out.println('n=' + n); // primitive operation
     foo(n-2, B, A, C);
}
```

### 2.1 Describe the analytical time complexity T(n) of this method (show your work)

In the above (which is a slightly modified Hanoi Tower!), line 2,

```
     if (n<=0) return; // primitive operation
```

...while a primitive operation, is only executed at the conclusion of the sequence, making it our constant, 1. The remaining lines,

```
     foo(n-2, A, C, B);
     System.out.println('n=' + n); // primitive operation
     foo(n-2, B, A, C);
```

...will execute $2T(n-2)$ primitive operations with every iteration (i.e., by calling foo() twice, which will iterate down and each call their own primitive operation). Therefore, the Recursive Time Complexity of this method is

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

...using the Elimination Method, we aim to drive towards $T(1)$ by making the left hand side of the equation cancel out a portion of the right hand side. Therefore, the starting equation has iterations as follows:

$$\begin{aligned}
\textit{iteration } 0 \; - \;& [T(n-2^0) = 2T(n-2^1)+1]2^0 \\
\textit{iteration } 1 \; - \;& [T(n-2^1) = 2T(n-2^2)+1]2^1 \\
\textit{iteration } 2 \; - \;& [T(n-2^2) = 2T(n-2^3)+1]2^2 \\
& ... \\
\textit{iteration } k-1 \; - \;& [T(n-2^{k-1}) = 2T(n-2^k)+1]2^{k-1} \\
\textit{iteration } k \; - \;& T(1) = 1
\end{aligned}$$

...where the $T$ term on the left hand side of each iteration cancels the $T$ term on the preceding iteration's right hand side. Keeping this in mind, we can now add up the non-eliminated terms to either discover a summation or reduce to closed-form, as follows:

$$1 + 2^0 + 2^1 + 2^2 + ... + 2^k$$

...which is a common geometric series resulting in the summation

$$\sum_{k=0}^{n-1} a^k = \frac{a^{n+1}-1}{a-1}$$

which can be reduced to

$$\sum_{k=0}^{n-1} 2^k = \frac{2^{n+1}-1}{2-1}$$

$$\therefore \sum_{k=0}^{n-1} 2^k = 2^{n+1}-1$$

...and, because we can assume, without loss of generality, that $n = 2^k$, we get our final Analytical Time Complexity of

$$\sum_{k=0}^{n-1} n = 2^{n+1}-1$$

## 2.2 Use proof-by-induction to demonstrate the analytical time complexity

BY INDUCTION, we can prove

$$2^0 + 2^1 + 2^2 + ... + 2^k = \frac{2^{n+1} - 1}{2 - 1}$$

and other geometric series of the form

$$1 + a^1 + a^2 + a^3 + ... + a^n = \frac{a^{n+1}}{a - 1}$$

or

$$a^0 + a^1 + a^2 + ... + a^n = \frac{a^{n+1}}{a - 1}$$

as follows. FIRSTLY we assume an Inductive Case, or instance. Assuming that $n = 0$, we can prove by substitution that

$$a^0 = \frac{a^{0+1} - 1}{a - 1}$$
$$a^0 = \frac{a - 1}{a - 1}$$
$$1 = 1$$

ONCE the above is proven, we aim to utilize our inductive hypothesis by proving that, when $n = k$,

$$a^0 + a^1 + a^2 + ... + a^k + a^{k+1} = \frac{a^{k+1}}{a - 1}$$

Since all cases up to $a^{k+1}$ have been proven inductively, we can substitute everything up to that term as follows, when $n = k + 1$:

$$\frac{a^{k+1} - 1}{a - 1} + a^{k+1} = \frac{a^{k+1+1} - 1}{a - 1}$$
$$\therefore \frac{(a^{k+1} - 1) + (a^{k+2} + a^{k+1})}{a - 1} = \frac{a^{k+2} - 1}{a - 1}$$
$$\therefore \frac{-1 + a^{k+2}}{a - 1} = \frac{a^{k+2} - 1}{a - 1}$$
$$\therefore 1 = 1$$

## 3

**(5 points) Assuming you have a task to SORT the telephone numbers of a major carrier, what sorting algorithm will you use? Please implement it using pseudo-code or JAVA and then justify the analytical time complexity.**

### 3.1 A Choice

If given the choice of sorting algorithms for a set of phone numbers, I would choose a Linear Sort such as Radix Sorting for the job, for the following reasons:

1) By not being a comparison sort, we already outperform most sorting mechanisms, getting very close to a big-Oh of $O(n)$ (i.e. approximately $O(4n)$). Removing the comparison and memory complexities thereof deeply simplify our work.

2) Because the range of each digit is fixed (0-9), we are perfectly set up for a Linear Sort, and, because the range is that of whole numbers, we do not have to make any other mathematical operations to get the numbers prepared (as we might have to do to scale down numbers for a Bucket Sort).

### 3.2 A Pseudo-Implementation via Pseudo-Code

Below is rough pseudocode.

```
radixSort(array[], int numOfDistinctDigits) {
   // an array for the number of distinct values
   int[] buckets = new int[numOfDistinctDigits];

   // an array for the result set
   int[] temp = new int[array.length];

   // use a counter to determine where in the new array the old value should go
   for (i = 0, i < numOfDistinctDigits, i++)
      buckets[i] = count of i existing in array[]

   // populate the new array with the sorted data
   for (j = 0, j < temp.length, j++) {
      temp[j] = number indicated by value pair in buckets[];
      running total of value pair in buckets++;
   }

   print temp[]
}
```

### 3.3 A Justification

BECAUSE the running time of the Linear Radix Sort approaches $O(n)$ with a stable set of digits (in this case, 10), we outperform all other sorting algorithms, especially any comparison algorithm.

## 4

**(15 points) Devise a heap-sorting-based algorithm for finding the k smallest elements of an unsorted set of n integers. The corresponding analytical time-complexity should also be provided. (Show your work; the time complexity for heap-building should also be included )**

Considering that every Heap sort must have comparable Heapify() and BuildHeap() functions doing the recursive work of sorting and building the heap, which would look like the below (in pseudo-code):

---

```
heapify(array[], int n) {
   int L = Left(n);
   int R = Right(n);
   if (L <= array.length && array[L] > array[n])
      large = L;
   else
      large = n;
   if (r <= array.length && array[R] > array[large])
      large = R;
   if (large != n)
      switch(array, n, large);
      heapify(array, large);
}

buildHeap(array[]) {
   for (int n=floor(array.length / 2), n > 0, n--) {
      heapify(array, n);
   }
}

findMin(sortedArray[], int k) {
   return newArray[k - 1] = sortedArray[0, k -1];
}

void main() {
   int[] array = {9,2,4,0,3,6,2,4};
   int k=3;
   buildHeap(array[]);
   print(findMin(array[], k));
)
```

...which would return

```
newArray = {9,2,4}
```

It is widely accepted that the running time of a heapify() function is $f(n) \in O(log_2\ n)$, and, since the heapify() function is called $n$ times, the big-Oh is $f(n) \in O(n\ log\ n)$. We get this because, looking at a heap tree, the worst case scenario is that the bottom "rung" of the tree is only half full, with pointers to nulls for the remainder of the tree. In this case, we get a Recursive Time Complexity of

$$T(n) = 2T\frac{n}{2} + n$$

...where $n$ represents the operations inherent within the rest of the method. Remembering that the Master Theorem dictates that

$$T(n) = aT(\frac{n}{b}) + f(n)$$

...we can take the recursive problem

$$T(n) = 2T\frac{n}{2} + n$$

...and assign $a = 2$, $b = 2$, and $f(n) = n$.

AND FURTHER remembering that the Master Theorem compares $n^{log_b\ a}$ against $f(n)$ as follows:

CASE 1) If $n^{log_b\ a}$ is polynomially larger than $f(n)$, then $T(n) = \theta(n^{log_b\ a})$

CASE 2) If $n^{log_b\ a}$ is equal to $f(n)$, then $T(n) = \theta(f(n)\ lg\ n)$

CASE 3) If $f(n)$ is polynomially larger than $n^{log_b\ a}$, then $T(n) = \theta(f(n))$

...we can see that:

$$n^{log_b\ a} \overset{?}{=} f(n)$$

$$n^{log_2\ 2} \overset{?}{=} n$$

$$\therefore n = n$$

THEREFORE, our CASE TWO applies, meaning that the aymptotic tight bounds of $T(n) = 2T\frac{n}{2} + n$ are:

$$f(n) = \theta(log_2\ n)$$

...which, when combined with the actual *build* of the heap, which runs in $O(n)$ time, we get a combined Time Complexity of

$$f(n) = \theta(n\ log\ n)$$

## 5

**(15 points) given a quick-sorting algorithm being implemented and runs on a four-processor parallel computer system, describe the corresponding best-case analytical time complexity. The time latency caused by inter-processor communication and synchronization are negligible. (Show your work).**

CONSIDERING that the Best-Case Analytical Time Complexity of a Quicksort is

$$T(n) = 2T(\frac{n}{2}) + n$$

...the introduction of a four-processor effectively divides the coefficient of the $T$ term by 4, meaning that we get a revised parallel Recursive Time Complexity of

$$T(n) = \frac{1}{2}T(\frac{n}{2}) + n$$

...using the Elimination Method, we aim to drive towards $T(1)$ by making the left hand side of the equation cancel out a portion of the right hand side. Therefore, the starting equation has iterations as follows:

$$\begin{gathered}
iteration\ 0\ -\ \ [T(n) = 0.5T(\frac{n}{2}) + n]0.5^0 \\
iteration\ 1\ -\ \ [T(\frac{n}{2^1} = 0.5T(\frac{n}{2^2}) + \frac{n}{2^1}]0.5^1 \\
iteration\ 2\ -\ \ [T(\frac{n}{2^2} = 0.5T(\frac{n}{2^3}) + \frac{n}{2^2}]0.5^2 \\
... \\
iteration\ k-1\ -\ \ [T(\frac{n}{2^{k-1}} = 0.5T(\frac{n}{2^k}) + \frac{n}{2^{k-1}}]0.5^{k-1} \\
iteration\ k\ -\ \ T(1) = 1
\end{gathered}$$

...where the $T$ term on the left hand side of each iteration cancels the $T$ term on the preceding iteration's right hand side. Keeping this in mind, we can now add up the non-eliminated terms to either discover a summation or reduce to closed-form, as follows:

$$n + 0.5^1(\frac{n}{2^1}) + 0.5^2(\frac{n}{2^2}) + ... + 0.5^{k-1}(\frac{n}{2^{k-1}})$$

or

$$n + \frac{1}{4}n + \frac{1}{16}n + \frac{1}{64}n + ... + 4^{1-k}n$$

Remembering that, without loss of generality, we can assume that $k = log\ n$, our Analytical Time Complexity becomes

$$4^{1-k}n$$
$$4^{1-log\ n}n$$

We can further determine the tight asymptotic bounds by remembering that the Master Theorem dictates that

$$T(n) = aT(\frac{n}{b}) + f(n)$$

From which we can take the recursive problem

$$T(n) = \frac{1}{2}T(\frac{n}{2}) + n$$

...and assign $a = 0.5$, $b = 2$, and $f(n) = n$.

AND FURTHER remembering that the Master Theorem compares $n^{log_b\ a}$ against $f(n)$ as follows:

CASE 1) If $n^{log_b\ a}$ is polynomially larger than $f(n)$, then $T(n) = \theta(n^{log_b\ a})$

CASE 2) If $n^{log_b\ a}$ is equal to $f(n)$, then $T(n) = \theta(f(n)\ lg\ n)$

CASE 3) If $f(n)$ is polynomially larger than $n^{log_b\ a}$, then $T(n) = \theta(f(n))$

...we can see that:

$$n^{log_b\ a} \overset{?}{=} f(n)$$
$$n^{log_2\ 0.5} \overset{?}{=} n$$
$$n^{-1} \overset{?}{=} n$$
$$\therefore \frac{1}{n} < n$$

THEREFORE, our CASE THREE applies, meaning that the aymptotic tight bounds of $T(n) = \frac{1}{2}T(\frac{n}{2}) + n$ are:

$$f(n) = \theta(n)$$

## 6

**(20 points) You are asked to modify the textbook version of Merge-Sort on page 31 into a new algorithm of Merge-Triplet-Sort. In Merge-Triplet-Sort, an input array A will be split into three equal-sized sub-arrays L, M, and R that stands for left, middle, and right.**

## 6.1

**(10 points) Please revise the textbook pseudocodes of Merge-Sort() and Merge() functions for your Merge-Triplet-Sort algorithm.**

Below is the *original* version of the MERGE and MERGE-SORT pseudocode functions:

---

```
MERGE(A, p, q, r) {
   n_1 = q - p + 1
   n_2 = r - q
   let L[1...n_1 + 1] and R[1...n_2 + 1] be new arrays
   for i = 1 to n_1
      L[i] = A[p + i - 1]
   for j = 1 to n_2
      R[j] = A[q + j]
   L[n_1 + 1] = infinity
   R[n_2 + 1] = infinity
   i = 1
   j = 1
   for k = p to r
      if L[i] leq R[j]
         A[k] = L[i]
         i = i + 1
      else A[k] = R[j]
         j = j + 1
}

MERGE-SORT(A, p, r) {
   if p < r
      q = floor((p + r)/2)
      MERGE-SORT(A, p, q)
      MERGE-SORT(A, q + 1, r)
      MERGE(A, p, q, r)
}
```

---

...of which a *modified* version as instructed is below:

---

```
JAMION-MERGE(A, p, q, r, t) {
   n_1 = q - p + 1
   n_2 = r - q + 1
   n_3 = t - r
   let L[1...n_1 + 1], R[1...n_2 + 1], and M[1...n_3 + 1] be new arrays
   for i = 1 to n_1
      L[i] = A[p + i - 1]
   for j = 1 to n_2
```

```
    M[j] = A[q + j - 1]
  for k = 1 to n_3
    R[k] = A[r + k - 1]
  L[n_1 + 1] = infinity
  M[n_2 + 1] = infinity
  R[n_3 + 1] = infinity
  i = 1
  j = 1
  k = 1
  for m = p to r
    if L[i] leq R[j]
      A[m] = L[i]
      i = i + 1
    else if M[j] leq R[k]
      A[m] = M[j]
      j = j + 1
    else A[m] = R[k]
      k = k + 1
}

MERGE-TRIPLET-SORT(A, p, r) {
  if p < r
    q = floor((p + r)/3)
    MERGE-TRIPLET-SORT(A, p, q)
    MERGE-TRIPLET-SORT(A, q + 1, r)
    MERGE-TRIPLET-SORT(A, r + 1, t)
    JAMION-MERGE(A, p, q, r, t)
}
```

---

### 6.2

**(10 points) Please find the analytical time complexity of Merge-Triplet-Sort using elimination method. You must show your procedure.**

The original Recursive Time Complexity of the textbook MERGE-SORT algorithm is

$$T(n) = 2T(\frac{n}{2}) + n$$

...with consideration that the new algorithm, instead of splitting the problem into *two*, now recursively splits the problem into *three*, we have a revised Recursive Time Complexity of

$$T(n) = 3T(\frac{n}{3}) + n$$

...using the Elimination Method, we aim to drive towards $T(1)$ by making the left hand side of the equation cancel out a portion of the right hand side. Therefore, the starting equation has iterations as follows:

$$\begin{gathered}
iteration\ 0\ -\ \ [T(n) = 3T(\frac{n}{3}) + n]3^0 \\
iteration\ 1\ -\ \ [T(\frac{n}{3^1} = 3T(\frac{n}{3^2}) + \frac{n}{3^1}]3^1 \\
iteration\ 2\ -\ \ [T(\frac{n}{3^2} = 3T(\frac{n}{3^3}) + \frac{n}{3^2}]3^2 \\
... \\
iteration\ k-1\ -\ \ [T(\frac{n}{3^{k-1}} = 3T(\frac{n}{3^k}) + \frac{n}{3^{k-1}}]3^{k-1} \\
iteration\ k\ -\ \ T(1) = 1
\end{gathered}$$

...where the $T$ term on the left hand side of each iteration cancels the $T$ term on the preceding iteration's right hand side. Keeping this in mind, we can now add up the non-eliminated terms to either discover a summation or reduce to closed-form, as follows:

$$n + 3^1(\frac{n}{3^1}) + 3^2(\frac{n}{3^2}) + ... + 3^{k-1}(\frac{n}{3^{k-1}})$$

...which reduces to

$$n + n + n + ... + nk$$

...and, because we can safely assume that $n = 2k$ and $k = log_2\ n$, we can reduce to closed-form as

$$n * log_2\ n$$

Therefore, our Analytical Time Complexity is $n\ log\ n$. We can further determine the tight asymptotic bounds by remembering that the Master Theorem dictates that

$$T(n) = aT(\frac{n}{b}) + f(n)$$

From which we can take the recursive problem

$$T(n) = 3T(\frac{n}{3}) + n$$

...and assign $a = 3$, $b = 3$, and $f(n) = n$.

AND FURTHER remembering that the Master Theorem compares $n^{log_b\ a}$ against $f(n)$ as follows:

CASE 1) If $n^{log_b\ a}$ is polynomially larger than $f(n)$, then $T(n) = \theta(n^{log_b\ a})$

CASE 2) If $n^{log_b\ a}$ is equal to $f(n)$, then $T(n) = \theta(f(n)\ lg\ n)$

CASE 3) If $f(n)$ is polynomially larger than $n^{log_b\ a}$, then $T(n) = \theta(f(n))$

...we can see that:

$$n^{log_b\ a} \overset{?}{=} f(n)$$

$$n^{log_3\ 3} \overset{?}{=} n$$

$$n^1 \overset{?}{=} n$$

$$\therefore n = n$$

THEREFORE, our CASE TWO applies, meaning that the aymptotic tight bounds of $T(n) = 3T(\frac{n}{3}) + n$ are:

$$f(n) = \theta(n\ log\ n)$$

## 7

**(10 points) Please modify the pseudocode of Count-Sort algorithm on page 195 in the textbook such that an input array of numbers in the rage of [-n, n] can be sorted.**

Below is the *original* version of the COUNT-SORT pseudocode function:

```
COUNTING-SORT(A, B, k) {
   let C[0...k] be a new array
   for i = 0 to k
      C[i] = 0
   for j = 1 to A.length
      C[A[j]] = C[A[j]] + 1
   // C[i] now contains the number of elements equal to i
   for i = 1 to k
      C[i] = C[i] + C[i - 1]
   // C[i] now contains the number of elements less than or equal to i
   for j = A.length downto 1
      B[C[A[j]]] = A[j]
      C[A[j]] = C[A[j]] - 1
}
```

...of which a *modified* version as instructed is below. Our work is made infinitely easier by virtue of our range, $-n$ to $n$, meaning that the floor of the input has the same absolute value of the ceiling! That is to say that the $abs(-n)$ is assumed to be $n$. Therefore:

```
JAMION-COUNTING-SORT(A, B, k) {
   let C[0...(k * 2)] be a new array
```

```
  for i = 0 to (k * 2)
    C[i] = 0
  for j = 1 to A.length
    C[A[j] + k] = C[A[j] + k] + 1
  // C[i] now contains the number of elements equal to i
  for i = 1 to (k * 2)
    C[i] = C[i] + C[i - 1]
  // C[i] now contains the number of elements less than or equal to i
  for j = A.length downto 1
    B[C[A[j] + k ]] = A[j]
    C[A[j] + k ] = C[A[j] + k] - 1
}
```