1. (25 points) Design a decrease-and-conquer algorithm for efficiently computing $\lfloor \lg n \rfloor$ and express it in pseudocode. Determine its time efficiency.

Solution:
We need to find an $m$ such that $2^m \leq n < 2^{m+1}$, then $m = \lfloor \lg n \rfloor$. We shall decrease $n$ by a factor of 2 at each step. Either of the following two algorithms would do.

**Algorithm 1:** Computing $\lfloor \lg n \rfloor$ by decrease and conquer

| input : An integer $n \geq 1$ |
| output: $\lfloor \lg n \rfloor$ |
| $m \leftarrow 0$ |
| while $n > 1$ do |
| $n \leftarrow \lfloor \frac{n}{2} \rfloor$ |
| $m \leftarrow m + 1$ |
| end |
| return $m$ |

**Algorithm 2:** $FloorOfLgOf(n)$

| input : An integer $n \geq 1$ |
| output: $\lfloor \lg n \rfloor$ |
| if $n = 1$ then |
| return 0 |
| else |
| return $FloorOfLgOf \left( \lfloor \frac{n}{2} \rfloor \right) + 1$ |
| end |

Both of the algorithms make $m = \lfloor \lg n \rfloor$ divisions. So, the efficiency is in $\Theta(\lg n)$.

2. (25 points) You have 9 gigabytes of data in a file on the disk. You are allowed to use only 1 gigabyte of RAM. How can you efficiently sort the data in the file given that a disk-access is very costly compared to a RAM-access. Describe your idea with as much specificity as you can. Assume 1 gigabyte = 1000 megabytes. **Hint:** Can you utilize the ideas of mergesort?

Solution:
Divide the 9 gigabytes file into 9 equal parts. Sort each part in-memory by an in-place sorting algorithm like heapsort and write them back to the disk. (The few extra variables used by heapsort can be kept in registers.) Now we need to sort these 9 sorted files. Chop each of these 9 files into 10 equal pieces. Each such piece is 100 megabytes in size. Load 1 such piece from each of the 9 parts into memory (RAM) and allocate the remaining 100 megabytes as the output buffer. Do a 9-way merge on the 9 pieces in-memory and write the sorted output in the output buffer. If the output buffer is full, write its content back to the disk. If one of the 9 pieces gets consumed, load one other piece from the corresponding part (on disk) and so on.

3. (25 points) We can multiply two $2 \times 2$ matrices to get their product matrix in the following way

$$
\begin{bmatrix}
a & b \\
c & d
\end{bmatrix} \cdot \begin{bmatrix}
A & C \\
B & D
\end{bmatrix} = \begin{bmatrix}
aA + bB & w + v + (a + b - c - d)D \\
w + u + d(B + C - A - D) & w + u + v
\end{bmatrix}
$$

where $u = (c - a)(C - D), v = (c + d)(C - A), w = aA + (c + d - a)(A + D - C)$.

We could generalize the above idea for efficiently multiplying two $n \times n$ matrices where $n$ is a power of 2, like we did for Strassen’s algorithm in the class. **Hint:** Do not forget to reuse computations.
(a) Set up the recurrence relation for the number of multiplications including the base case. Solve it.

**Solution:**

To calculate \( u \) we need 1 multiplication, for \( v \) we need 1 multiplication, and for \( w \) we need \( 1 + 1 = 2 \) multiplications. In order to calculate \( C_{00} \) we need just 1 multiplications as we can reuse \( aA \) from \( w \)'s calculation. For \( C_{01} \) and \( C_{10} \) we need 1 multiplication for each of them and for \( C_{11} \) we do not need any multiplication. So, in total there are 7 multiplications. The recurrence relation is

\[
M(n) = \begin{cases}
7 \cdot M\left(\frac{n}{2}\right) & \text{when } n > 1 \\
1 & \text{when } n = 1
\end{cases}
\]

We cannot use the Master theorem to solve the recurrence relation, as \( f(n) = 0 \not\in \Theta(n^d) \) for \( d \geq 0 \).

Let’s use back substitution assuming \( n = 2^m \),

\[
M(n) = 7 \cdot M\left(\frac{n}{2}\right) = 7^2 \cdot M\left(\frac{n}{2^2}\right) = \ldots = 7^m \cdot M\left(\frac{n}{2^m}\right) = 7^m = 7^{\log n} = n^{\log 7}
\]

So, \( M(n) \in \Theta(n^{\log 7}) \).

(b) Set up the recurrence relation for the number of additions/subtractions including the base case. Solve it.

**Solution:**

We need 1 subtraction for \( (c - d) \), 1 subtraction for \( (C - D) \), 1 addition for \( (c + d) \), 1 subtraction for \( C - A \), 1 addition for \( (c + d - a) \) [as we can reuse \( c + d \)], and 1 addition for \( (A + D - C) = A - (C - D) \) [as we can just negate \( C - D \)]. Then for \( C_{00} \) we need 1 addition, for \( C_{01} \) we need \( 1 + 1 + 1 = 3 \) additions/subtractions [as we can reuse \( c + d - a \)], for \( C_{10} \) we need \( 1 + 1 + 1 = 3 \) additions/subtractions [as we can reuse \( A + D - C \)], and for \( C_{11} \) we need just 1 addition [as we can reuse \( w + u \) from \( C_{10} \)]. So, in total there are 15 additions/subtractions. The recurrence relation is,

\[
A(n) = \begin{cases}
7 \cdot A\left(\frac{n}{2}\right) + 15 \cdot \left(\frac{n}{2}\right)^2 & \text{when } n > 1 \\
0 & \text{when } n = 1
\end{cases}
\]

Here, \( a = 7 \), \( b = 2 \), and \( d = 2 \). As \( b^d = 2^2 = 4 < a \), by the Master theorem \( A(n) \in \Theta(n^{\log 7}) \).

(c) To what asymptotic efficiency class does this algorithm’s complexity belong? Is it better than Strassen’s algorithm? If yes, in what respect? If no, why is it inferior?

**Solution:**

As both \( M(n) \) and \( A(n) \) have the same asymptotic growth rate, the overall asymptotic efficiency is in \( \Theta(n^{\log 7}) \). Though this one has the same asymptotic efficiency as that of Strassen's, it uses smaller number of additions/subtractions. So, this one is more efficient than Strassen’s.

4. (25 points) Consider the following algorithm for transforming an array into a (max) heap:
Algorithm 3: Bottom up heap construction

input : An array \( H[1..n] \) of orderable items
output: A heap \( H[1..n] \)

1. for \( i \leftarrow \left\lceil \frac{n}{2} \right\rceil \) downto 1 do
   2. \( k \leftarrow i \)
   3. \( v \leftarrow H[k] \)
   4. heap \leftarrow false
   5. while not heap and \( 2 \cdot k \leq n \) do
      6. \( j \leftarrow 2 \cdot k \)
      7. if \( j < n \) then
         8. if \( H[j] < H[j + 1] \) then
            9. \( j \leftarrow j + 1 \)
      end
      12. if \( v \geq H[j] \) then
         13. heap \leftarrow true
         15. \( H[k] \leftarrow H[j] \)
         16. \( k \leftarrow j \)
      end
   18. end
   19. \( H[k] \leftarrow v \)
end

(a) What changes would you make to Algorithm 1, in order to use it for transforming an array into a min heap, instead of a max heap. Recall that a min heap is an essentially complete binary tree in which, every key is less than or equal to the keys in its children.

Solution:
On line 8 change “<” into “>” and on line 12 change “≥” into “≤”.

(b) Perform heapsort on the array \( ⟨7, 8, 16, 4, 14⟩ \). Show the states of the array during iterations both for the “heap construction” and the “maximum deletions” stages.

Solution:
For the “heap construction”:

\[ ⟨7, 8, 16, 4, 14⟩ \]
\[ ⟨7, 14, 16, 4, 8⟩ \]
\[ ⟨16, 14, 7, 4, 8⟩ \]

The bold numbers are the keys of the parents.
For the “maximum deletions”:
\[
\langle 16, 14, 7, 4, 8 \rangle \\
\langle 8, 14, 7, 4, 16 \rangle \\
\langle 14, 8, 7, 4, 16 \rangle \\
\langle 4, 8, 7, 14, 16 \rangle \\
\langle 8, 4, 7, 14, 16 \rangle \\
\langle 7, 4, 8, 14, 16 \rangle \\
\langle 4, 7, 8, 14, 16 \rangle 
\]

Here bold numbers belong to the current heap.

1 Master Theorem

For the recurrence relation \( T(n) = aT\left(\frac{n}{b}\right) + f(n) \), if \( f(n) \in \Theta(n^d) \) where \( d \geq 0 \), then
\[
T(n) \in \begin{cases} 
\Theta(n^d) & \text{if } a < b^d \\
\Theta(n^d \log n) & \text{if } a = b^d \\
\Theta(n^{\log_a b}) & \text{if } a > b^d 
\end{cases}
\]

Analogous results hold for the \( O \) and \( \Omega \) notations.

2 Useful formulas

1. \( \sum_{i=1}^{u} 1 = u - l + 1 \)
2. \( \sum_{i=1}^{n} i = \frac{n(n+1)}{2} \)
3. \( \sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6} \)
4. \( \sum_{i=0}^{n} a^i = \frac{a^{n+1} - 1}{a-1} (a \neq 1) \)
5. \( \sum_{i=1}^{n} i2^i = (n-1)2^{n+1} + 2 \)
6. \( \lg n = \frac{\ln n}{\ln 2} \)
7. \( \frac{d}{dn} n^m = mn^{m-1} \)
8. \( \frac{d}{dn} \ln n = \frac{1}{n} \)
9. \( \frac{d}{dn} a^n = (\ln a)a^n \)