Homework Assignment 1

CS/CE 412/471 Algorithms: Design and Analysis, Spring 2025

5 problems, 75 points

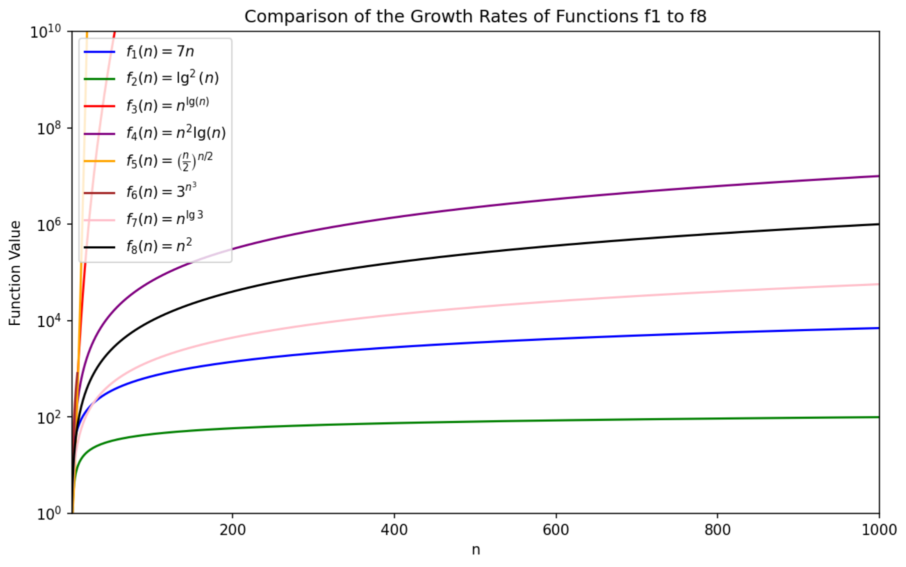
1. $\boxed{\text{10 points}}$ Sort the order of growth of the following functions and justify your solution.

$$f_1(n) = 7n, \quad f_2(n) = \lg n^{\lg n}, \quad f_3(n) = n^{\lg n}, \quad f_4(n) = n^2 \lg n$$

$$f_5(n) = \left(\frac{n}{2}\right)^{\frac{n}{2}}, \quad f_6(n) = 3^{n^3}, \quad f_7(n) = n^{\lg 3}, \quad f_8(n) = n^2$$

If some have the same asymptotic growth, then be sure to indicate that. As usual, lg indicates a base of 2.

Solution:



Explanation:
The graph compares the growth rates of eight functions with varied asymptotic growth, shown on a logarithmic scale to better illustrate their differences. Logarithmic growth is extremely slow, therefore $f_2(n) = \lg^2(n)$ grows the slowest. $f_1(n) = 7n$ represents linear growth, which grows faster than logarithmic functions but slower than quadratic and exponential growth. Finally, $f_3(n) = n\lg(n)$ shows linearithmic growth, which is quicker than linear growth but slower than quadratic functions. The quadratic function ($f_8(n) = n^2$) develops faster than linear and linearithmic functions, but slower than exponential growth. $f_4(n) = n^2\lg(n)$ combines quadratic and logarithmic growth, resulting in quicker growth than pure quadratic functions, but slower than exponential ones. $f_7(n) = n^{\lg 3}$ is faster than quadratic functions but slower than exponential functions like $f_5(n)$ and $f_6(n)$. $f_5(n) = \left(\frac{n}{2}\right).^{n/2}$ is an exponential growth function that increases very quickly, while $f_6(n) = 3^{n^3}$ is the quickest and exhibits super-exponential growth. In conclusion, $f_2(n)$ has the slowest growth, while $f_6(n)$ has the fastest, with each function expanding at varying rates in between.

$$f_2(n) < f_1(n) < f_7(n) < f_8(n) < f_4(n) < f_3(n) < f_5(n) < f_6(n)$$

2. Prove or disprove each of the following:

   (a) $\boxed{\text{5 points}}$ $f(n) = \Theta\left(f\left(\frac{n}{4}\right)\right)$

   Solution: False. Let $f(n) = 2^n$, then $f(n/4) = 2^{n/4}$. Now to determine whether $f(n)$ and $f(n/4)$ grow at the same asymptotic rate, we compare their ratio.
   If $f(n) = \Theta(f(n/4))$, then their growth rates must be within constant factors which

means,

$$\lim_{n\to\infty} \frac{f(n)}{f(n/4)} \neq 0 \text{ and } \neq \infty.$$

Now, we'll compute the ratio,

$$\lim_{n\to\infty} \frac{f(n)}{f(n/4)} = \lim_{n\to\infty} \frac{2^n}{2^{n/4}}.$$

We'll simplify the above ratio using exponent properties (i.e. $\frac{a^m}{a^n} = a^{m-n}$),

$$\lim_{n\to\infty} 2^{n-\frac{n}{4}} = \lim_{n\to\infty} 2^{3n/4} = \infty.$$

Since the ratio tends to ∞, which means

$$f(n) = \Omega(f(n/4))$$

where $f(n)$ grows much faster than $f(n/4)$, this doesn't satisfy Big-Theta relation. Therefore, $f(n) \neq \Theta(f(n/4))$.

(b) 5 points $2^n = o(2^{\frac{n}{2}})$ (i.e. $2^n = O(2^{\frac{n}{2}})$ and $2^n \neq \Theta(2^{\frac{n}{2}})$)

Solution:

Definition of Little-O Notation (o)

A function $f(n)$ is said to be **little-o** of $g(n)$, written as:

$$f(n) = o(g(n))$$

if and only if:

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$$

False. Now to prove this, we'll do as follows,

$$= \lim_{n\to\infty} \frac{2^n}{2^{\frac{n}{2}}}$$
$$= \lim_{n\to\infty} 2^{n-\frac{n}{2}}$$
$$= \lim_{n\to\infty} 2^{\frac{n}{2}} = \infty$$

For $2^n = o(2^{\frac{n}{2}})$ to be true, 2^n must grow strictly slower than $2^{\frac{n}{2}}$. However, we've shown that 2^n grows strictly faster than $2^{\frac{n}{2}}$. Thus, $2^n \neq o(2^{\frac{n}{2}})$.

References

[1] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Fourth Edition.*

(c) $\boxed{\text{5 points}}$ $\Theta(f(n)) + \Theta(g(n)) = \Theta(f(n) + g(n))$

Solution:

Definition of Big-Theta Notation (Θ)

For a given function $g(n)$, we denote by $\Theta(g(n))$ the set of functions:

$$\Theta(g(n)) = \{f(n) \mid \text{there exist positive constants } c_1, c_2, \text{ and } n_0 \text{ such that}\}$$

$$0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n), \quad \text{for all } n \geq n_0$$

True. Let $p(n) = \Theta(f(n))$ and $q(n) = \Theta(g(n))$. By the definition of Big-Theta, we know there exist constants c_1, c_2, d_1, d_2, and sufficiently large values n_p and n_q such that

$$0 \leq c_1 f(n) \leq p(n) \leq c_2 f(n), \quad \forall n \geq n_p$$

$$0 \leq d_1 g(n) \leq q(n) \leq d_2 g(n), \quad \forall n \geq n_q$$

Now, let's add the above two inequalities

$$c_1 f(n) + d_1 g(n) \leq p(n) + q(n) \leq c_2 f(n) + d_2 g(n)$$

For large n, the sum $f(n) + g(n)$ will be dominated by the function that grows faster. Thus, the upper bound and lower bound can be expressed as follows,

$$c_2 f(n) + d_2 g(n) \leq \max(c_2, d_2)(f(n) + g(n)) \tag{1}$$

and

$$c_1 f(n) + d_1 g(n) \geq \min(c_1, d_1)(f(n) + g(n)) \tag{2}$$

Thus, by equations (1) and (2) we get,

$$\min(c_1, d_1)(f(n) + g(n)) \leq p(n) + q(n) \leq \max(c_2, d_2)(f(n) + g(n))$$

Since this is the definition of Big-Theta notation, so we conclude that,

$$p(n) + q(n) = \Theta(f(n) + g(n))$$

Therefore, we have proved that

$$\Theta(f(n)) + \Theta(g(n)) = \Theta(f(n) + g(n))$$

References

[1] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Fourth Edition.*

3. You are given 2 problems below. You need to design an efficient algorithm for each of them. Specifically,

   i. Clearly state the problem. For example, see how the sorting problem is stated on the first page of Chapter 1 of CLRS.

   ii. Describe your algorithm. Follow the pseudocode conventions listed in Section 2.1 of CLRS. *Do not* use programming language features, e.g. `list.append()`.

   iii. Illustrate, in any suitable manner, e.g. as in Section 2.1 in CLRS, a run of your algorithm on a sample input.

   iv. Describe the best and worst case for your algorithm.

   v. Describe an input, if any, on which your algorithm may not work correctly.

   vi. Provide a time complexity analysis of your algorithm in terms of input size n using asymptotic notation.

   (a) $\boxed{\text{10 points}}$ Given a number, x, and an array, A, containing n numbers, decide if the array contains two numbers a and b such that $a + b = x$.

Solution:

i. Problem Definition
Input: A number x and $A = [a_1, a_2, .., a_n]$ containing n numbers
Output: A Boolean value that indicates whether there exist two numbers a and b in the array such that $a + b = x$

ii. Algorithm

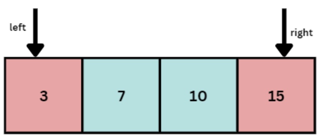
```
Find-Pair-With-Sum(A, x)
   MERGE-SORT(A, 1,n)
   left = 1
   right = n
   while $left < right$
   current_sum = A[left] + A[right]
   if current_sum = x
      return True
   elseif current_sum ¡ x
      left = left + 1
   else
      right = right - 1
   end if
   return False

MERGE(A, start, middle, end)
   $n_1 = middle - start + 1$
   $n_2 = end - middle$
   Create arrays $L[1 \ldots n_1]$ and $R[1 \ldots n_2]$
   for $i = 1$ to $n_1$ do
      $L[i] = A[start + i - 1]$
   end for
   for $j = 1$ to $n_2$ do
```

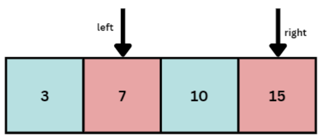
```
        $R[j] = A[middle + j]$
    **end for**
    $i = 1, j = 1, k = start$
    **while** $i \leq n_1$ **and** $j \leq n_2$ do
        **if** $L[i] \leq R[j]$
            $A[k] = L[i]$
            $i = i + 1$
        **else**
            $A[k] = R[j]$
            $j = j + 1$
        $k = k + 1$
    **end while**
    **while** $i \leq n_1$ do
        $A[k] = L[i]$
        $i = i + 1$
        $k = k + 1$
    **end while**
    **while** $j \leq n_2$ do
        $A[k] = R[j]$
        $j = j + 1$
        $k = k + 1$
    **end while**

**MERGE-SORT**(A, start, end)
    **if** $start < end$
        middle = $\lfloor \frac{start+end}{2} \rfloor$
        **MERGE-SORT**(A, start, middle)
        **MERGE-SORT**(A, middle + 1, end)
        **MERGE**(A, start, middle, end)
    **end if**
```

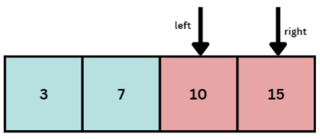
iii. Sample Input Dry Run



(a)
Current Sum: 3 + 15 = 18
Value of X: 25



(b)
Current Sum: 7 + 15 = 22
Value of X: 25



(c)
Current Sum: 10 + 15 = 25
Value of X: 25

iv. Best and Worst Case of Algorithm

   Best Case: $O(1)$, when pair whose sum equals the target is found on the first check. For example, the left pointer begins at 3 and the right pointer begins at 15 if the array is [3, 7, 10, 15] and the target sum is 17. The program finds the pair in the first check since their sum is 17.

   Worst Case: $O(n)$, when the pair is not found, and the algorithm has to traverse the entire array. For example, There are no pairs in the array [3, 7, 10, 15] with a needed sum of 100. In order to verify every possible combination without discovering a pair that matches, the algorithm will shift the pointersuntil they meet.

v. When algorithm will not work?
   As long as the array is sorted and the target sum falls within the range of possible sums of two different elements, the algorithm will work properly in almost every case. The method will return 'False' if the expected total is too big or too little to be generated by any pair of elements.

vi. Time Complexity Analysis

   Sorting the Array will take $O(nlogn)$ time. Since each pointer moves no more than n times, the **two-pointer approach** runs in $O(n)$ time. The sorting stage thus dominates the entire time complexity, which is **O(nlogn)**.

References

1. https://www.geeksforgeeks.org/check-if-pair-with-given-sum-exists-in-array/

(b) $\boxed{\text{10 points}}$ Given k sorted lists containing a total of n numbers, combine them to create a single sorted list of the n numbers.

Solution:

i. Problem Definition

Input: A collection of k sorted lists $\langle L_1, L_2, \ldots, L_k \rangle$, where each list $L_i = \langle l_{i1}, l_{i2}, \ldots, l_{in_i} \rangle$ contains n_i elements sorted in non-decreasing order, and $n = n_1 + n_2 + \cdots + n_k$ represents the total number of elements across all lists.

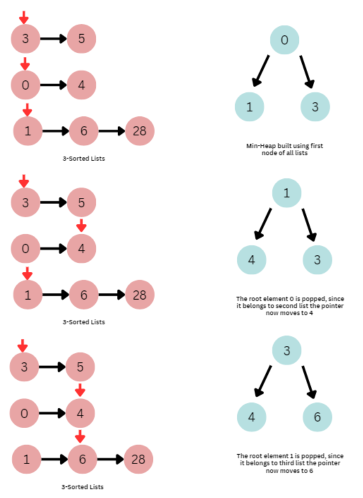
Output: A single sorted list $L = \langle l_1, l_2, \ldots, l_n \rangle$, where L contains all elements from the input lists $L_1, L_2, \ldots, L_k$ in non-decreasing order, such that:

$$l_1 \leq l_2 \leq \cdots \leq l_n$$

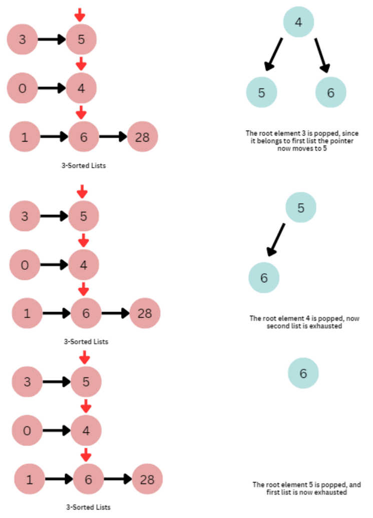
ii. Algorithm

```
Merge-Sorted-Lists($\langle L_1, L_2, \ldots, L_k \rangle$)
  min_heap = []
  result = []
  for $i = 1$ to $k$
    if $L_i \neq []$
      push $(L[i][0], i, 0)$ into min_heap
  end if
  end for
  while min_heap $\neq$ []
    (val, list_idx, ele_idx) = pop from min_heap
    append val to result
    if $ele\_idx + 1 \leq$ length($L[list\_idx]$)
      next_val = $L[list\_idx][ele\_idx + 1]$
      push $(next\_val, list\_idx, ele\_idx + 1)$ into min_heap
    end if
  end while
  return result
```

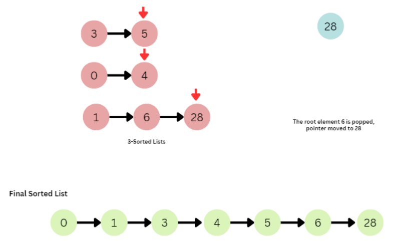
iii. Sample Input Dry Run


3-Sorted Lists
Min-Heap built using first node of all lists
3-Sorted Lists
The root element 0 is popped, since it belongs to second list the pointer now moves to 4
3-Sorted Lists
The root element 1 is popped, since it belongs to third list the pointer now moves to 6

1


3-Sorted Lists
The root element 3 is popped, since it belongs to first list the pointer now moves to 5
3-Sorted Lists
The root element 4 is popped, now second list is exhausted
3-Sorted Lists
The root element 5 is popped, and first list is now exhausted

2



3

iv. Best and Worst Case of Algorithm

Best Case: $O(klogk)$, where all lists have their initial elements in ascending order and after the initial insertion of the first entries, no additional heap operations are needed. For instance, given the lists $\langle[1,3],[2,4],[5,6]\rangle$, the method will efficiently merge the first entries (1, 2, 5) from the min heap into the final sorted list without requiring any additional heap a restructuring following the initial pop.

Worst Case: $O(nlogk)$, where k is the number of lists and n is the total number of entries in all the lists. When the algorithm must push and pop each element into the heap, this occurs. Since we have n elements, the total complexity will be $O(nlogk)$, and each insertion or removal will take the heap $O(logk)$ time. For instance, the heap will need to be adjusted repeatedly when each element is processed if the input lists are lengthy and unsorted, like $\langle[10,20,30],[5,15],[7,17,27]\rangle$.

v. When algorithm will not work?
Even if one or more of the lists are empty, the algorithm will continue to work properly for all valid inputs. In order to efficiently merge non-empty lists into a single sorted list, it will make use of the min-heap. It will carefully avoid empty lists because it makes no assumptions about whether any list is non-empty. Therefore, as long as the input lists are sorted and the approach is applied appropriately, there are no particular inputs where this algorithm would fail.

vi. Time Complexity Analysis

It takes $O(logk)$ time for each operation (including insertion and deletion) on the min-heap. The overall time complexity for these operations is **O(nlogk)** as we execute them n times (one for each element in the lists).

References

1. `https://algo.monster/liteproblems/23`

4. For each of the following recurrences, derive a solution to the recurrence (show your working) and compare your solution with that obtained through the master theorem (show how it applies).

(a) 5 points $T(n) = 2T\left(\frac{n}{4}\right) + 1$

Solution:

Method 1:- Expansion Method

$$T(n) = 2T(n/4) + 1 \tag{1}$$

Finding $T(n/4)$,

$$T(n/4) = 2T(n/16) + 1 \tag{2}$$

Now substituting equation (2) into (1),

$$T(n) = 2(2T(n/16) + 1) + 1$$

$$T(n) = 2^2T(n/16) + 2 + 1 \tag{3}$$

Now we find $T(n/16)$,

$$T(n/16) = 2T(n/64) + 1 \tag{4}$$

Again we'll substitute equation (4) into (3) because of which we get,

$$T(n) = 2^2(2T(n/64) + 1) + 2 + 1$$

$$T(n) = 2^3T(n/64) + 4 + 2 + 1$$

We'll then generalize after k steps,

$$T(n) = 2^k T(n/4^k) + \sum_{i=0}^{k-1} 2^i$$

since the base case is $n/4^k = 1$ we'll find value of k as follows,

$$n/4^k = 1$$
$$n = 4^k$$
$$\log_4 n = k \log_4 4$$
$$k = \log_4 n$$

The sum of the series is

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

Thus,

$$T(n) = 2^{\log_4 n} T(n/4^{\log_4 n}) + 2^{\log_4 n} - 1$$

The $2^{\log_4 n}$ can be simplified using logarithmic base change rule,

$$2^{\log_4 n} = n^{\log_4 2}$$
$$2^{\log_4 n} = n^{\frac{1}{2}}$$
$$T(n) = n^{\frac{1}{2}} T(1) + n^{\frac{1}{2}} - 1$$

So we get,

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

Method 2:- Master Theorem

The given recurrence is in the form:

$$T(n) = aT(n/b) + f(n)$$

where $a = 2$, $b = 4$, and $f(n) = 1$.

The important step is to compute watershed function $n^{\log_b a}$ we'll plug in our values to get,

$$n^{\log_4 2} = n^{\frac{1}{2}}$$

Now we'll compare $f(n) = 1 = O(1)$ with $n^{\frac{1}{2}}$. As we can see that $f(n)$ is smaller than $n^{1/2}$, **Case 1** of the Master Theorem will get applied here.

Master Theorem Case 1

If there exists a constant $\epsilon > 0$ such that:

$$f(n) = O(n^{\log_b a - \epsilon}),$$

then:

$$T(n) = \Theta(n^{\log_b a}).$$

Since $f(n) = O(n^0) = O(n^{\log_4 2-\epsilon})$ for $\epsilon = \frac{1}{2}$, Case 1 applies giving us with

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

Conclusion

The **Expansion Method** gave us an upper bound $O(n^{\frac{1}{2}})$ while **Master Theorem** showed that this bound was not only the upper bound but also the exact growth of $T(n)$ giving us $\Theta(n^{\frac{1}{2}})$. Thus, the final answer should be

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

References

[1] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Fourth Edition.*

(b) $\boxed{\text{5 points}}$ $T(n) = 2T\left(\frac{n}{4}\right) + n$

Solution:

Method 1:- Expansion Method

$$T(n) = 2T(n/4) + n$$

Substituting $T(n/4)$,

$$T(n) = 2(2T(n/16) + n/4) + n$$
$$T(n) = 2^2T(n/16) + 2(n/4) + n$$

Now again substituting $T(n/16)$ and expanding it we get,

$$T(n) = 2^3T(n/64) + 2^2(n/16) + 2(n/4) + n$$

Now we'll generalize after k steps,

$$T(n) = 2^kT(n/4^k) + \sum_{i=0}^{k-1} 2^i(n/4^i)$$

The base case occurs when $n/4^k = 1$, which gives,

$$\frac{n}{4^k} = 1$$
$$n = 4^k$$
$$k = \log_4 n$$

The summation term is,

$$\sum_{i=0}^{k-1} 2^i(n/4^i)$$

$$n\sum_{i=0}^{k-1}(2/4)^i = n\sum_{i=0}^{k-1}(1/2)^i$$

Since this is geometric series we get,

$$\sum_{i=0}^{k-1}(1/2)^i = \frac{1-(1/2)^k}{1-1/2} = 2(1-(1/2)^k)$$

Now when substituting $k = \log_4 n$ we'll get

$$T(n) = 2^k T(n/4^{\log_4 n}) + n(2(1-(1/n^{1/2})))$$

For large k, we approximate the geometric series to the following,

$$\sum_{i=0}^{k-1}(1/2)^i \approx 2$$

Because of which now we'll get,

$$T(n) = 2^k T(n/4^{\log_4 n}) + 2n$$

$$T(n) = 2^k T(1) + 2n$$

Since the first term $2^k T(1)$ is a constant, we drop it in asymptotic analysis,

$$T(n) = O(n)$$

Method 2:- Master Theorem

The given recurrence is in the form:

$$T(n) = aT(n/b) + f(n)$$

where $a = 2$, $b = 4$ and $f(n) = n$.
We'll plug in our values to watershed function $n^{\log_b a}$,

$$n^{\log_b a} = n^{\log_4 2} = n^{1/2}$$

We'll compare $f(n) = n = O(n)$ with $n^{1/2}$. Since $f(n)$ grows faster than $n^{1/2}$, we'll check if **Case 3** of the Master Theorem applies.

Master Theorem Case 3

If there exists a constant $\epsilon > 0$ such that:

$$f(n) = \Omega(n^{\log_b a+\epsilon}),$$

and if $f(n)$ additionally satisfies the ***regularity condition***:

$$af(n/b) \leq cf(n)$$

for some constant $c < 1$ and for all sufficiently large n, then:

$$T(n) = \Theta(f(n)).$$

Since $f(n) = \Omega(n^{\log_4 2+\epsilon})$ for $\epsilon = \frac{1}{2}$, this condition holds true and hence **Case 3** applies.
Now we check it's **regularity condition**,

$$af(n/b) \leq cf(n)$$

$$2f(n/4) \leq cf(n)$$

$$2(n/4) \leq c(n)$$

$$n(1/2) \leq c(n)$$

Since $(1/2)n \leq cn$ for $c = 1/2 < 1$, the regularity condition holds.
Since both conditions for Case 3 are satisfied, we'll conclude that,

$$T(n) = \Theta(f(n)) = \Theta(n)$$

Conclusion

The **Expansion Method** gave us an upper bound $O(n)$ while **Master Theorem** showed that this bound was not only the upper bound but also the exact growth of $T(n)$ giving us $\Theta(n)$. Thus, the final answer should be

$$T(n) = \Theta(n)$$

References

[1] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Fourth Edition.*

(c) $\boxed{\text{5 points}}$ $T(n) = 2T\left(\frac{n}{4}\right) + n^2$

Solution:

Method 1:- Expansion Method

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

Substituting $T(n/4)$,

$$T(n) = 2\left(2T(n/16) + (n/4)^2\right) + n^2$$

$$T(n) = 2^2T(n/16) + 2(n^2/16) + n^2$$

Now again substituting $T(n/16)$ and expanding it to get,

$$T(n) = 2^3T(n/64) + 2(n^2/16 + n^2/4) + n^2$$

Now well generalize after k steps,

$$T(n) = 2^kT(n/4^k) + \sum_{i=0}^{k-1} 2^i(n^2/4^i)$$

Since $n/4^k = 1$ at base case, we get $k = \log_4 n$ and the geometric series sum becomes,

$$\sum_{i=0}^{k-1} 2^i(n^2/4^i) = n^2 \sum_{i=0}^{k-1} \left(\frac{1}{2}\right)^i$$

For large k, we approximate the geometric series to the following,

$$\sum_{i=0}^{k-1}(1/2)^i \approx 2$$

So now we'll get

$$T(n) = 2^kT(1) + 2n^2$$

$$T(n) = 2^{\log_4 n}T(1) + 2n^2$$

$$T(n) = 2^{\log_4 n}T(1) + 2n^2$$

$$T(n) = n^{1/2}T(1) + 2n^2$$

We can say that $n^{1/2}$ is dominated by the $2n^2$ term for large n. Thus, the total complexity becomes,

$$T(n) = O(n^2)$$

Method 2:- Master Theorem

The given recurrence is in the form:

$$T(n) = aT(n/b) + f(n)$$

where $a = 2$, $b = 4$ and $f(n) = n^2$.
We'll plug in our values to watershed function $n^{\log_b a}$,

$$n^{\log_b a} = n^{\log_4 2} = n^{1/2}$$

We'll compare $f(n) = n^2$ with $n^{1/2}$. Since $f(n)$ grows faster than $n^{1/2}$, we'll check if **Case 3** of the Master Theorem applies.

Master Theorem Case 3

If there exists a constant $\epsilon > 0$ such that:

$$f(n) = \Omega(n^{\log_b a+\epsilon}),$$

and if $f(n)$ additionally satisfies the ***regularity condition***:

$$af(n/b) \leq cf(n)$$

for some constant $c < 1$ and for all sufficiently large n, then:

$$T(n) = \Theta(f(n)).$$

Since $f(n) = \Omega(n^{\log_4 2+\epsilon})$, for $\epsilon = 1/2$, this condition holds true hence **Case 3** applies.
Now we check it's **regularity condition**,

$$af(n/b) \leq cf(n)$$
$$2f(n/4) \leq cf(n)$$
$$2(n^2/16) \leq c(n^2)$$
$$n^2(1/8) \leq c(n^2)$$

Since $(1/8)n^2 \leq cn^2$ for $c = 1/8 < 1$, the regularity condition holds.
Since both conditions for Case 3 are satisfied, we'll conclude that,

$$T(n) = \Theta(f(n)) = \Theta(n^2)$$

Conclusion

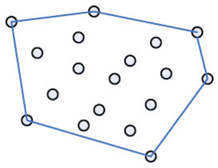
The **Expansion Method** gave us an upper bound $O(n^2)$ while **Master Theorem** showed that this bound was not only the upper bound but also the exact growth of $T(n)$ giving us $\Theta(n^2)$. Thus, the final answer should be

$$T(n) = \Theta(n^2)$$

References

[1] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Fourth Edition.*

5.



The *convex hull* of a set of 2D points is a subset of the points that forms a convex polygon which contains all the other points. It can be described informally as, the shape enclosed by a rubber band stretched around the points.

(a) $\boxed{\text{10 points}}$ Design a divide and conquer algorithm to compute the convex hull of a set of 2D points and describe it using the pseudocode conventions listed in Section 2.1 of CLRS.

Solution:

Algorithm

1. We first sort the points by x-coordinate before callingthe procedure to calculate the convex hull once and for all. It takes $O(nlogn)$ time to complete this step.
2. **Divide Step:** Determine the median x-coordinate of the point. This step should take a fixed amount of time because the input points have already been sorted by x-coordinate. It could occasionally take up to $O(n)$ time, depending on how we implement it.
3. **Conquer Step:** Recursively run the procedure on both halves.
4. **Merge Step:** Combine the two convex hulls that were determined by the conquer step's two recursive calls. The process of merging should take $O(n)$ time.

Pseudocode

```
CONVEX-HULL(P)
  if $|P| \leq 3$
    if $|P| == 3$ and not IS-LEFT-TURN($P[1], P[2], P[3]$)
      swap $P[2]$ and $P[3]$
    return $P$
  end if
  end if
  sort $P$ by x-coordinate
  mid = $\lfloor |P|/2 \rfloor$
  $P_L = P[1 : mid]$
  $P_R = P[mid + 1 : n]$
```

```
    $leftHull$ = CONVEX-HULL($P_L$)
    $rightHull$ = CONVEX-HULL($P_R$)
    **return** MERGE($leftHull$, $rightHull$)

**MERGE**(H_L, H_R)
    $i_{up}$ = rightmost point index in $H_L$
    $j_{up}$ = leftmost point index in $H_R$
    $i_{low} = i_{up}$
    $j_{low} = j_{up}$

    **repeat**
        $saved_i = i_{up}$
        $saved_j = j_{up}$
        **while IS-LEFT-TURN**($H_L[i_{up}], H_R[j_{up}], H_R[(j_{up}+1) \mod |H_R|]$)
            $j_{up} = (j_{up}+1) \mod |H_R|$
        **while** not **IS-LEFT-TURN**($H_L[(i_{up}-1) \mod |H_L|], H_L[i_{up}], H_R[j_{up}]$)
            $i_{up} = (i_{up}-1) \mod |H_L|$
        **end while**
    **end while**
    **until** $i_{up} = saved_i$ **and** $j_{up} = saved_j$

    **repeat**
        $saved_i = i_{low}$
        $saved_j = j_{low}$
        **while IS-RIGHT-TURN**($H_L[i_{low}], H_R[j_{low}], H_R[(j_{low}+1) \mod |H_R|]$)

            $j_{low} = (j_{low}+1) \mod |H_R|$
        **while** not **IS-RIGHT-TURN**($H_L[(i_{low}-1) \mod |H_L|], H_L[i_{low}], H_R[j_{low}]$)

            $i_{low} = (i_{low}-1) \mod |H_L|$
        **end while**
    **end while**
    **until** $i_{low} = saved_i$ **and** $j_{low} = saved_j$

    $merged\_hull$ = empty array
    $k = i_{up}$
    **while** $k \neq i_{low}$
        $[len(merged\_hull):] = H_L[k]$ **// equal to .append functionality**
        $k = (k+1) \mod |H_L|$
    $[len(merged\_hull):] = H_L[i_{low}]$

    $k = j_{low}$
    **while** $k \neq j_{up}$
        $[len(merged\_hull):] = H_R[k]$
        $k = (k+1) \mod |H_R|$
    $[len(merged\_hull):] = H_R[j_{up}]$

    return $merged\_hull$

IS-LEFT-TURN(p, q, r)
    return $(r_y - p_y)(q_x - p_x) > (q_y - p_y)(r_x - p_x)$

IS-RIGHT-TURN(p, q, r)
    return $(r_y - p_y)(q_x - p_x) < (q_y - p_y)(r_x - p_x)$
```

Note: Miss Ayesha Enayat told to apppend this way in pseudocode convention $[len(a) :] = x$ this will be equivalent to $a.append(x)$

References

1. `https://www.geeksforgeeks.org/convex-hull-algorithm/`
2. `https://sahandsaba.com/convex-hull-visualized-using-raphael.html`

(b) $\boxed{\text{5 points}}$ Visualize the running of your algorithm as an animation and submit the animation as an MP4 file.