Homework 1

CS302 - Fall 2025

(You should try to solve the problem by yourself first and then compare your solution with the one from AI tools)

Question 1: Fill the parentheses with the shortest answer

(a) $n^3 + 2n^2 + 7n + 1 = \Theta(n^3)$

(b) $n + \log_2 n = \Theta(n)$

(c) $n^2 + 2^n = \Theta(2^n)$

(d) $\log_2 n + \log_3 n = \Theta(\log n)$ the largest will dominate

(e) $2^n + n! + n^2 = \Theta(n!)$

(f) $2^n + n^n + n! = \Theta(n^n)$

Question 2: Mark True/False for each of the following statements

(a) $n^3 + 10n^2 + 7n + 1 = \Theta(n^2)$ F. n^3 is dominated

(b) $\log_2 n + n\log_2 n = \Theta(n)$ F. logn is dominated.

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

(d) $n + \log_2 n = \Omega(n)$ T. Bounded below constant time

(e) $2^n + n! + n^2 = O(2^n)$ F. n! will out grow 2^n eventually.

(f) $2^n + n^n + n! = \Omega(n!)$ T

Question 3: The pseudocode of the insertion sort algorithm is given below

```
Algorithm 1: Insertion sort
1  Input: An unsorted array A
2  Output: A sorted array in an ascending order

3  n = length of A
4  for i = 1 → n-1 do
5      for j = i-1 down to 0 do
6          if A[j+1] < A[j] then
7              swap A[j+1] with A[j]
8          else
9              break
10 return A
```

← algorithms swap leftwards until the order is restored

is sorted → So the invariant holds for next iteration

(a) Explain briefly why this algorithm is correct (i.e. the algorithm always returns a sorted array from a given input array).

b4 the 1st outer iteration i = 1, the subarray A[0..0] is trivally sorted.
A[i] inside the loop moves leftward by swapping until it's = A[0...i-1]. If A[i] < A[0...i-1] → swaps leftwards till order is restored; otherwise break. When swapping done, A[0,..i] is sorted → invariant hold for next i+1. When finish with i = n → A[0,...n-1] is sorted → returns a sorted array!

(b) Which is the appropriate asymptotic notation (i.e. O, Ω, or Θ) to show the number of times that the for loop (line 4) is executed? Show the number of times with this appropriate asymptotic notation.

for i = 1 → n-1 do : n-1 times

→ $n-1 = \Theta(n)$, equivalent to $O(n)$, $\Omega(n)$ → so $\Theta(n)$ is the tight bound. Θ will make it's best to make sure the loop always return exactly n-1 iterations

Answer in the next page

(c) The same question as in (b) but for the comparison operation (line 6)

(d) The same question as in (b) but for the swapping operation (line 7).

(e) ***(Optional)*** The key point of insertion sort is determining how to insert A[i] into the subarray A[1,...,i−1], which is already sorted (lines 5-9). Since this subarray is sorted, we can use binary search to find the correct position and then insert A[i] into this position. The binary search step only costs O(logn), while the insertion step costs O(1). Therefore, the entire insertion sort algorithm would only cost O(nlogn), which is as efficient as merge sort or quick sort. Is this correct? Justify your argument for when A is an array and when A is a linked list.

Question 4: We would like to compare four sorting algorithms: selection sort, insertion sort, merge sort, and quick sort. Your tasks are as follows:

(a) Implement all four algorithms in Python.
(b) Generate randomized number lists of lengths 1,000, 10,000, and 100,000.
(c) Measure the running time of all four algorithms for these lists.
(d) Attach screenshots of your code and the running times

Question 5: We need to find the maximum element and the minimum element of an array. The pseudocode of a divide-and-conquer algorithm is given below

Algorithm 4: Divide-and-conquer algorithm to find min and max

```
1  function divide-and-conquer-min-max(A)
2      n = length of A
3      if n == 1 then
4          return A[0], A[0]
5      if n == 2 then
6          if A[0] < A[1] then
7              return A[0], A[1]
8          else
9              return A[1], A[0]
10     mid = ceiling(n/4)
11     m1,M1 = divide-and-conquer-min-max(A[0, ..., 2*mid - 1])
12     m2,M2 = divide-and-conquer-min-max(A[2*mid, ..., n-1])
13     return min{m1, m2}, max{M1, M2}
```

(a) Count **exactly** the number of comparisons (line 6) of this algorithm

(b) Is it faster than the naive algorithm that we usually use? If yes, then where do we save the comparisons?

c/ if $A[j+1] < A[j]$

- worst case : require comparing with previous elements for each of the insertion
  $\rightarrow 1 + 2 + \ldots + (n-1) = \frac{n(n-1)}{2} = \theta(n^2)$
- best case : sorted already. only need 1 comparision per iteration
  $\rightarrow n-1 = \theta(n)$
  $\rightarrow$ worst case : $\theta(n^2)$
- average case : random. $A[i]$ is inserted half way back
  $\rightarrow \frac{i}{2}$ comparision per iteration
  $\rightarrow \frac{1}{2} \cdot \frac{n(n-1)}{2} = \theta(n^2)$

d/ $A[j+1]$ with $A[j]$

worst case : descending order $\rightarrow$ need to swap each new element thru the whole sorted prefix.

Swaps: $1 + 2 + \ldots + (n-1) = \theta(n^2)$

best case : sorted already. $\rightarrow$ no need to swap.

swaps = 0

$\rightarrow$ Numer of swaps is $\Theta(n^2)$ for the worst case.

e/ <u>Hypothesis</u> : Using binary search $(O(n \log n))$ + constant insertion $(O(1)) \rightarrow$ the whole algorithm $= O(n \log n)$

- When A is an array
  $\rightarrow$ the claim is incorrect.
  While binary search can find the position in $O(\log n)$, inserting into the array requires shifting elements to make room, shifting costs $O(n)$ in the worst case $\rightarrow$ overall time conplexity remains $O(n^2)$
- When A is a linked list:
  $\rightarrow$ the claim is incorrect
  Insertion can be done in $O(1)$ once the correct position is found. But we could not index in constant time. To access the middle element we must traverse sequentially, which already costs $O(n)$.
  Thus even with a linked list, the search step is $O(n)$ and total complexity $= O(n^2)$

```python
#Sorting Algorithms Comparison

import random, time

# selection sort

def selection_sort(arr):
    a = arr[:]
    n = len(a)
    for i in range(n - 1):
        min_index = i
        for j in range(i + 1, n):
            if a[j] < a[min_index]:
                min_index = j
        a[i], a[min_index] = a[min_index], a[i]   # do the swap after scanning
    return a

# insertion sort

def insertion_sort(arr):
    a = arr[:]
    for i in range(1, len(a)):
        key = a[i]
        j = i - 1
        while j >= 0 and a[j] > key:
            a[j + 1] = a[j]
            j -= 1
        a[j + 1] = key
    return a
```

```python
# merge sort

def merge_sort(arr):
    if len(arr) <= 1:
        return arr
    mid = len(arr) // 2
    left = merge_sort(arr[:mid])
    right = merge_sort(arr[mid:])
    return merge(left, right)

def merge(left, right):
    result = []
    i = j = 0
    while i < len(left) and j < len(right):
        if left[i] <= right[j]:
            result.append(left[i]); i += 1
        else:
            result.append(right[j]); j += 1
    result.extend(left[i:])
    result.extend(right[j:])
    return result
```

```python
# quick sort

def quick_sort(arr):
    if len(arr) <= 1:
        return arr
    pivot = arr[len(arr)//2]
    left  = [x for x in arr if x < pivot]
    mid   = [x for x in arr if x == pivot]
    right = [x for x in arr if x > pivot]
    return quick_sort(left) + mid + quick_sort(right)

def generating_lists():
    sizes = [1000, 10000, 100000]
    return {n: [random.randint(0, 10**6) for _ in range(n)] for n in sizes}

def measure_time(func, arr, repeats=1):
    start = time.perf_counter()
    for _ in range(repeats):
        func(arr)
    end = time.perf_counter()
    return (end - start) / repeats

data_sets = generating_lists()

algorithms = {
    "Selection sort": selection_sort,
    "Insertion sort": insertion_sort,
    "Merge sort": merge_sort,
    "Quick sort": quick_sort
}

for n, arr in data_sets.items():
    print(f"/List size = {n}")
    for name, func in algorithms.items():

        t = measure_time(func, arr)
        print(f"{name}: {t:.5f} seconds")
```

```
/List size = 1000
Selection sort: 0.02601 seconds
Insertion sort: 0.03800 seconds
Merge sort: 0.00241 seconds
Quick sort: 0.00194 seconds
/List size = 10000
Selection sort: 3.56740 seconds
Insertion sort: 3.18870 seconds
Merge sort: 0.07846 seconds
Quick sort: 0.04361 seconds
/List size = 100000
```

5/

Question 5: We need to find the maximum element and the minimum element of an array. The pseudocode of a divide-and-conquer algorithm is given below

Algorithm 4: Divide-and-conquer algorithm to find min and max

```
1  function divide-and-conquer-min-max(A)
2      n = length of A
3      if n == 1 then
4          return A[0], A[0]
5      if n == 2 then
6          if A[0] < A[1] then
7              return A[0], A[1]
8          else
9              return A[1], A[0]
10     mid = ceiling(n/4)
11     m1,M1 = divide-and-conquer-min-max(A[0, ..., 2*mid - 1])
12     m2,M2 = divide-and-conquer-min-max(A[2*mid, ..., n-1])
13     return min{m1, m2}, max{M1, M2}
```

(a) Count **exactly** the number of comparisons (line 6) of this algorithm

(b) Is it faster than the naive algorithm that we usually use? If yes, then where do we save the comparisons?

a/ Denote $T(n)$ = number of comparisons for array of size n

if A[0] < A[1] then

- Base case: $n = 1 \rightarrow$ no comparisons. $T(1) = 0$
- Base case: $n = 2 \rightarrow$ exactly 1 comparison $T(2) = 1$
- Recursive case $n > 2$ / Array is splitted into 2 halves

Each recursive call returns (min, max) for its half

$\rightarrow$ 2 comparisons $\left[\begin{array}{l} \min\{m1, m2\} \rightarrow 1 \text{ comparison} \\ \max\{M_1, M_2\} \rightarrow 1 \text{ comparison} \end{array}\right.$

$n = 1 : T(1) = 0$

$n = 2 : T(2) = 1$

$n > 2 : T(n) = T(2 \times mid) + T(n - 2 \times mid) + 2$

since $mid = ceil(n/4)$ (line 10) $\rightarrow mid \times 2 = \frac{n}{2}$

$\rightarrow$ split in half $\rightarrow T(n) = T(\frac{n}{2}) + T(\frac{n}{2}) + 2 = 2T(\frac{n}{2}) + 2$

$\rightarrow T(n) = 2.(2T.\frac{n}{4} + 2) + 2 = 4T.\frac{n}{4} + 6$

$\rightarrow T(n) = 8T.\frac{n}{8} + 14 \rightarrow T(n) = 2^i T\left(\frac{n}{2^i}\right) + 2(2^i - 1)$

Base case when $T(2) = 1$

$\rightarrow \frac{n}{2^i} = 2$

$n = 2^{i+1}$

let $n = 2^k$

$2^k = 2^{i+1}$

$\rightarrow k = i + 1$ (1)

plus (1) into $T(n)$

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

As $2^{k-1} = \frac{n}{2}$

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

$\rightarrow$ For general $n \rightarrow T(n) = \lceil \frac{3n}{2} \rceil - 2$

b/ Yes. it is faster because it's avoiding double the work.

For naive algorithm / the min value and max value will cost $n-1$ comparisons for each.

$\rightarrow$ total for naive is $(n-1) + (n-1) = 2(n-1)$

$$2(n-1) > \frac{3n}{2} \quad (\forall n > 0)$$

$\rightarrow$ divide and conquer also is more efficient in terms of number of comparisons.

$$2n - \frac{3}{2}n = \frac{1}{2}n$$

$\rightarrow$ divide and conquer save $\approx \frac{n}{2}$ comparisons.

Question 6: We need to compute the value 2^n for an input integer number n

(a) Develop a naive algorithm running in a linear time complexity O(n).

Algorithm : naive_double (n)
input : an integer
output : value 2^n
if n < 0:
return 1 / naive_double (-n)
if n == 0:
return 1

result = 1
for i from 1 → n :
result = result × 2
return result

(b) Develop a divide-and-conquer algorithm that is faster than O(n).

Algorithm : divide_and_conquer (n)
input : an integer n ≥ 0
output: value 2^n
if n < 0 then
return 1 / divide_and_conquer (-n)
if n == 0 then
return 1
Compute $2^{floor(n/2)}$

half_pow = divide_and_conquer (floor(n/2))
if n is even :
return half_pow × half_pow
else :
return 2 × half_pow × half_pow
→ O(log n). Keep divide n by 2 till reach 0.

Question 7: For a number array A, an inversion is a pair (i, j) such that i < j but A[i] > A[j]. For example, A = [1, 9, 6, 4, 5] has 5 inversions (9, 6), (9, 4), (9, 5), (6, 4), (6, 5).

(a) Develop a naive algorithm running in $O(n^2)$ to count the number of all inversions, n is the length of A.

Algorithms : naive_inver_count (A)
Input: An array A of length n
output: number of inversions in A
count ← 0
for i ← 0 to n − 2 do
for j ← i + 1 to n-1 do
if A[i] > A[j] then
count ← count + 1
return count

My idea is to check every pair i, j where i < j and count if A[i] > A[j]

→ Checking all pairs will cost $O(n^2)$.

(b) Develop a divide-and-conquer algorithm to count faster than $O(n^2)$.

Algorithm : inversion_count (A)
input : array A of length n
output: number of inversions in A
if length (A) ≤ 1 then
return 0, A → sorted already.
mid ← floor (n/2)
l_count, l_sort ← inversion_count (A[0: mid])
r_count, r_sort ← inversion_count (A[mid: n])
split_count, merge ← merge_and_count (l_sort, r_sort)
return l_count + r_count + split_count + merged

Algorithm merge_count (left, right)
input : 2 sorted arrays left and right
output : number of split inversions, merged sorted array

```
i ← 0, j ← 0, count ← 0
merged ← []
while i < length (left) and j < length (right) do
    if left [i] ≤ right[j] then
        append left [i] to merged
        i ← i + 1
    else :
        append right[i] to merged
        j ← j + 1
        count ← count + (length (left) - i)
    // all remaining elements in left from inversions with right [j]
append remaining elements of left [i:] to merged
append remaining elements of right [j:] to merged

return (count, merged)
```

Question 8 (Optional): We are given a number array A, and two numbers min and max. Our goal is to count how many continuous subarrays of A that the sum is in the interval [min, max]. For example, A = [1, -3, 2, 4], min = 1, max = 3 then there are 3 such sub-arrays including [1], [2], and [-3, 2, 4].

(a) Develop a simple algorithm to solve this problem in $O(n^3)$, n is the length of A.

(b) Develop a divide-and-conquer algorithm faster than $O(n^2)$ to solve this problem.

a/ Idea: enumerate all subarrays A[i..j] ($O(n^2)$ choices)
For each, compute the sum by looping over its element
Check whether the subarrays falls inside [min, max]

Algorithm: Count-sub-array
input: array A, integer n minVal, maxVal
Output: Count of subarrays with sum in [minVal, maxVal]

```
n ← length (A)
count ← 0
for i ← 0 to n-1:
    for j ← i to n-1:
        S = 0
        for k = i to j:
            S += A[k]
        if minVal <= S <= maxVal
            count ← +1
return count
```

b/

Question 8 (Optional): We are given a number array A, and two numbers min and max. Our goal is to count how many continuous subarrays of A that the sum is in the interval [min, max]. For example, A = [1, -3, 2, 4], min = 1, max = 3 then there are 3 such sub-arrays including [1], [2], and [-3, 2, 4].

(a) Develop a simple algorithm to solve this problem in $O(n^3)$, n is the length of A.
(b) Develop a divide-and-conquer algorithm faster than $O(n^2)$ to solve this problem.