Assignment (part 1)

Question 1:

Completed by Tze Keat 2301221 and Yew Chong 2301219

Context:
In Singapore, all males have to serve a mandatory national service. After completing their basic military training (BMT), they will be posted to a vocation such as infantry, armour, or other specialities. However, many, such as myself, were not given the choice of vocation we had opted for. This is especially disappointing when we have the necessary skillset for a certain vocation but are not given the chance to use it for our services.
If servicemen are given a role they are more suited for and a role they want, the military can spend less time on foundational training and focus more on operational training. For instance, engineering students would have had hands-on training on tools and be more familiar with tools in a workshop as compared to a business student, which made them a more preferred choice for the support crew.
By choosing the right person for the job, it can also reduce the chance of injury since it avoids soldiers deemed unsuitable for the role, for instance, a PES A in infantry should perform better and be healthier when compared to a PES B4 status.

Assigning servicemen to the right place that suits them may also boost morale and make their 2 years in national service a much more relevant place for them.

Why this matters:

- From a serviceman's point of view, their criteria can be:
    - The role they are interested in, the Distance to the camp, and opportunities to train overseas
- From a SAF point of view:
    - PES Status (Health), Relevant degree/diploma, Manpower needs

How the algorithm works

- Two Parties: Serviceman (S) and Vocations (V)
- Capacity: Each vocation has a capacity of cap[v]

<u>Pseudocode:</u>

```
Put all servicemen into a free queue. Track each serviceman's "next vocation to propose to".
While queue is not empty
Pop S, if S has no more vocation it is unmatched
    Let V be S next choice,  S propose to V
If assigned[v] < cap[v] accept S
Else, in V find the worst candidate (W)
If V prefer S over W, swap S and W then requeue W, else continue
```

Constraint and limitation:
In a real-world context, there may be hard constraints such as a Person of the same PES status does not equate to incapable or capable of doing certain things. Which can result in a serviceman not suitable for a certain role (i.e. excuse physical activity for a PES B due to injury). In this case the algorithm is incapable of determining impossible combinations and letting the user know this.

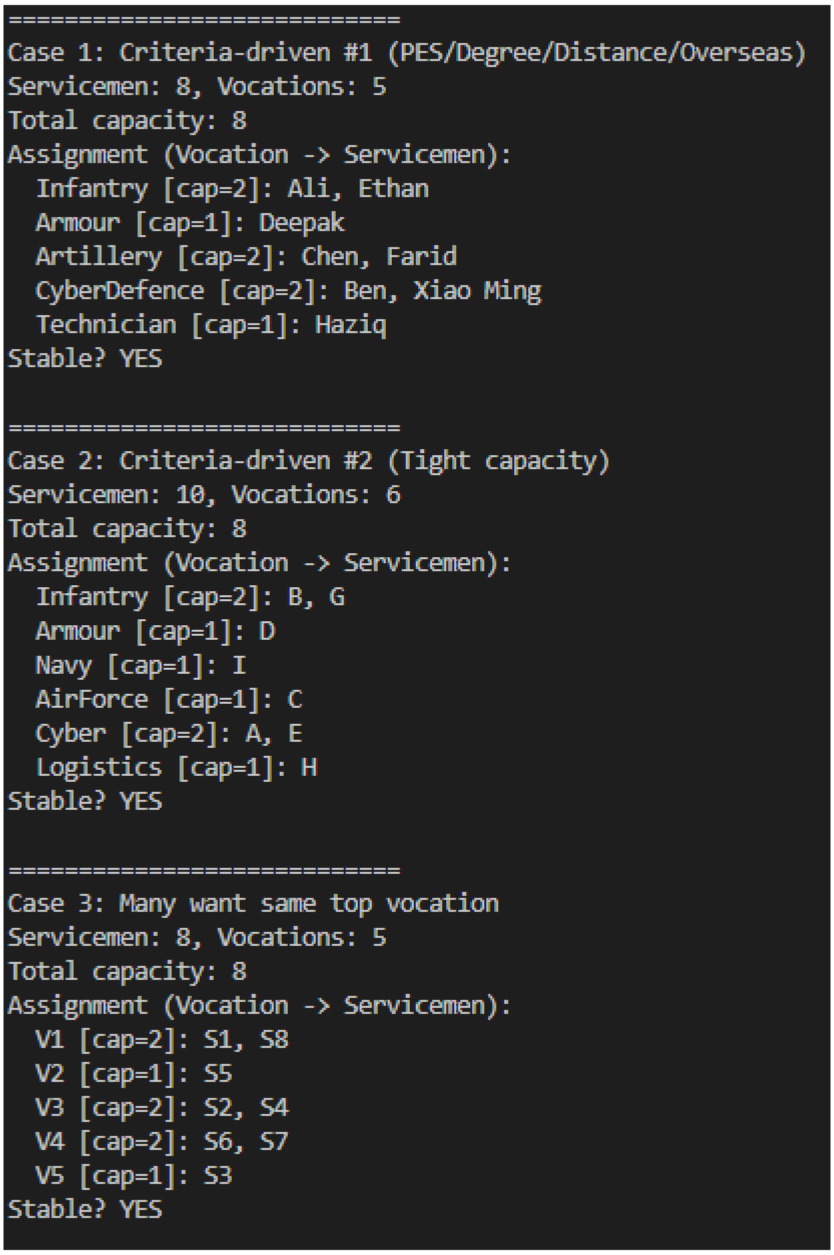
Analyze efficiency:
This algorithm is similar to Gale-Shapley's algorithm but there is a difference as our algorithm performs many to one matching instead of one to one.

The algorithm has a best-case of O(N * V), where N = number of people on one side and V is the number of vocations. This happens when everyone is matched to the vocation they want without being reshuffed.
The worst case is O(N*V*logC) where C is the capacity of the vocation. In this case, servicemen are being shuffled around all the time and the log C comes from maintaining the priority queue of figuring out which serviceman should first be popped from the vocation queue, based on least preferred.

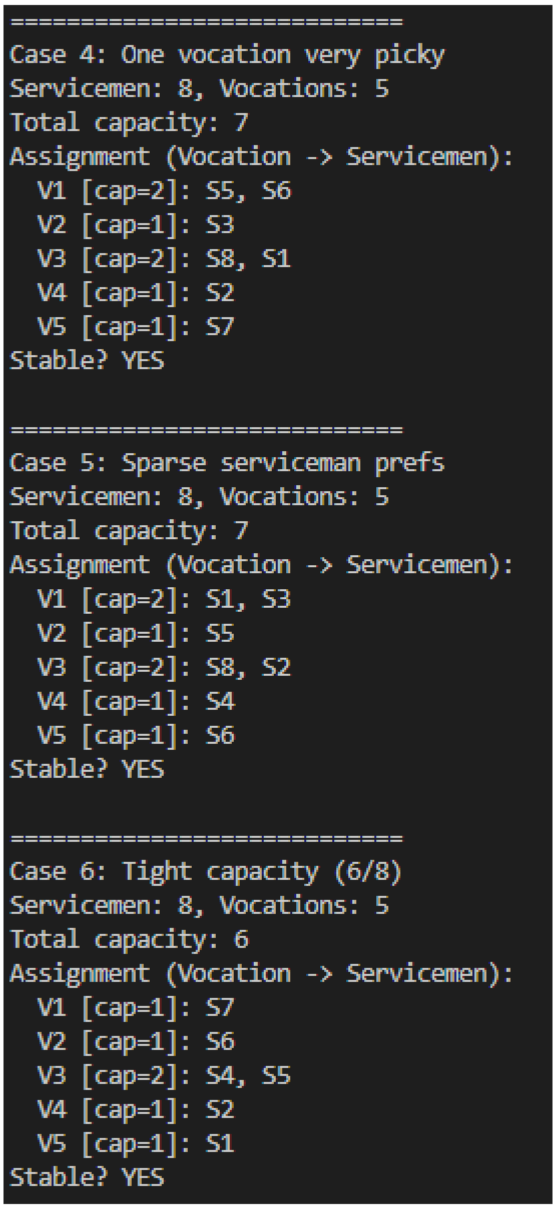
How to use the program
Use the g++ compiler with the command g++ -std=c++17 -O2 -Wall -Wextra Ass1_Q1.cpp -o assignment, then followed by running the program using ./assignment. No parameters into the command line are required.

The program will automatically run all test cases at once with inputs that are already built into the program.

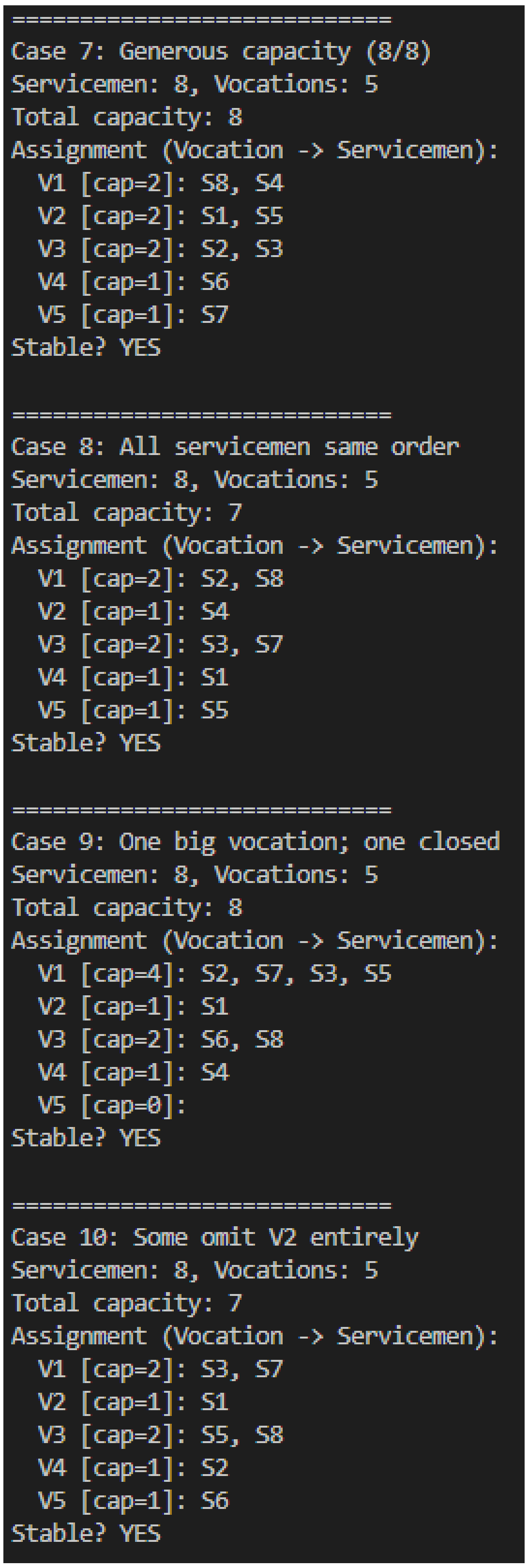
Files used: Ass1_Q1.cpp
Testing result:
Each comes with the case of how we test, follow by X number of serviceman and X number of vocations
And total capacity, then we print out the data on who is assigned to where, you can refer to the test case code functions to view more information



```
============================
Case 1: Criteria-driven #1 (PES/Degree/Distance/Overseas)
Servicemen: 8, Vocations: 5
Total capacity: 8
Assignment (Vocation -> Servicemen):
  Infantry [cap=2]: Ali, Ethan
  Armour [cap=1]: Deepak
  Artillery [cap=2]: Chen, Farid
  CyberDefence [cap=2]: Ben, Xiao Ming
  Technician [cap=1]: Haziq
Stable? YES

============================
Case 2: Criteria-driven #2 (Tight capacity)
Servicemen: 10, Vocations: 6
Total capacity: 8
Assignment (Vocation -> Servicemen):
  Infantry [cap=2]: B, G
  Armour [cap=1]: D
  Navy [cap=1]: I
  AirForce [cap=1]: C
  Cyber [cap=2]: A, E
  Logistics [cap=1]: H
Stable? YES

============================
Case 3: Many want same top vocation
Servicemen: 8, Vocations: 5
Total capacity: 8
Assignment (Vocation -> Servicemen):
  V1 [cap=2]: S1, S8
  V2 [cap=1]: S5
  V3 [cap=2]: S2, S4
  V4 [cap=2]: S6, S7
  V5 [cap=1]: S3
Stable? YES
```



```
==============================
Case 4: One vocation very picky
Servicemen: 8, Vocations: 5
Total capacity: 7
Assignment (Vocation -> Servicemen):
  V1 [cap=2]: S5, S6
  V2 [cap=1]: S3
  V3 [cap=2]: S8, S1
  V4 [cap=1]: S2
  V5 [cap=1]: S7
Stable? YES

==============================
Case 5: Sparse serviceman prefs
Servicemen: 8, Vocations: 5
Total capacity: 7
Assignment (Vocation -> Servicemen):
  V1 [cap=2]: S1, S3
  V2 [cap=1]: S5
  V3 [cap=2]: S8, S2
  V4 [cap=1]: S4
  V5 [cap=1]: S6
Stable? YES

==============================
Case 6: Tight capacity (6/8)
Servicemen: 8, Vocations: 5
Total capacity: 6
Assignment (Vocation -> Servicemen):
  V1 [cap=1]: S7
  V2 [cap=1]: S6
  V3 [cap=2]: S4, S5
  V4 [cap=1]: S2
  V5 [cap=1]: S1
Stable? YES
```



```
==============================
Case 7: Generous capacity (8/8)
Servicemen: 8, Vocations: 5
Total capacity: 8
Assignment (Vocation -> Servicemen):
  V1 [cap=2]: S8, S4
  V2 [cap=2]: S1, S5
  V3 [cap=2]: S2, S3
  V4 [cap=1]: S6
  V5 [cap=1]: S7
Stable? YES

==============================
Case 8: All servicemen same order
Servicemen: 8, Vocations: 5
Total capacity: 7
Assignment (Vocation -> Servicemen):
  V1 [cap=2]: S2, S8
  V2 [cap=1]: S4
  V3 [cap=2]: S3, S7
  V4 [cap=1]: S1
  V5 [cap=1]: S5
Stable? YES

==============================
Case 9: One big vocation; one closed
Servicemen: 8, Vocations: 5
Total capacity: 8
Assignment (Vocation -> Servicemen):
  V1 [cap=4]: S2, S7, S3, S5
  V2 [cap=1]: S1
  V3 [cap=2]: S6, S8
  V4 [cap=1]: S4
  V5 [cap=0]:
Stable? YES

==============================
Case 10: Some omit V2 entirely
Servicemen: 8, Vocations: 5
Total capacity: 7
Assignment (Vocation -> Servicemen):
  V1 [cap=2]: S3, S7
  V2 [cap=1]: S1
  V3 [cap=2]: S5, S8
  V4 [cap=1]: S2
  V5 [cap=1]: S6
Stable? YES
```

Question 2:
Completed by Bryan Ang 2301397 (part a) and Ryan Cheong 2301236 (part b)

a) Algorithm Order of Growth Types:
1. Constant - O(1)
What it means: The algorithm takes the same amount of time regardless of input size.
Characteristics:
- Time stays exactly the same whether you have 10 items or 10 million items
- The most efficient type possible
- No loops that depend on input size
- Direct access operations
Real-world example: Looking up a word in a dictionary if someone tells you the exact page number. Whether the dictionary has 100 or 1000 pages, you go directly to that page.

2. Logarithmic - O(log n)
What it means: Time increases slowly as input size increases. When input doubles, time increases by just one unit.
Characteristics:
- Very efficient for large datasets
- Often involves dividing the problem in half repeatedly
- Common in search algorithms
- Growth pattern: 1→2→3→4 as input goes 2→4→8→16
Real-world example: Finding a word in a dictionary by opening to the middle, then middle of remaining half, etc. Each step eliminates half the remaining pages.

3. Linear - O(n)
What it means: Time increases proportionally with input size. Double the input = double the time.
Characteristics:
- Very predictable and common
- Usually involves examining each item once
- Forms a straight line on a graph
- Growth pattern: 1→2→4→8 as input goes 1→2→4→8
Real-world example: Reading every page of a book. Twice as many pages = twice as much reading time.

4. Linearithmic - O(n log n)
What it means: Combines linear and logarithmic growth. More than linear but less than quadratic.
Characteristics:
- Common in efficient sorting algorithms
- Better than quadratic, worse than linear
- Often involves divide-and-conquer approaches
- Growth pattern: 1→4→12→32 as input goes 1→2→4→8
Real-world example: Organizing a deck of cards by repeatedly splitting it into smaller piles, sorting each pile, then merging them back together.

5. Quadratic - O(n^2)
What it means: Time increases with the square of the input size. Double the input = four times the time.
Characteristics:
- Growth becomes very steep with large inputs
- Often involves nested loops
- Each item compared with every other item

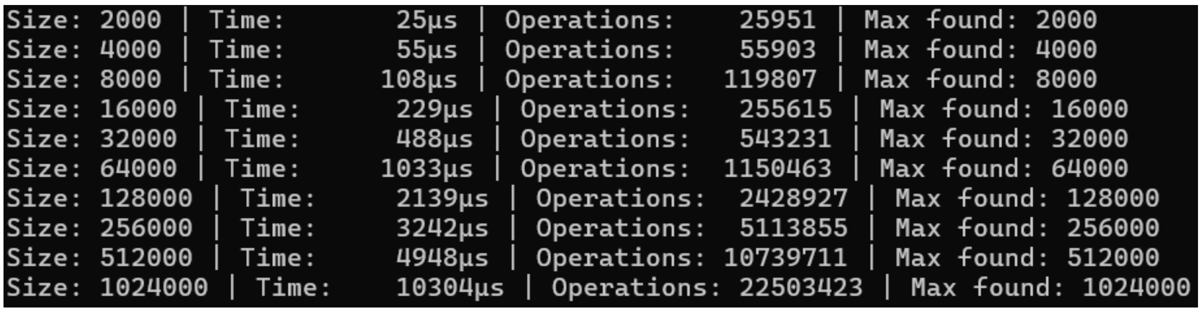
- Growth pattern: 1→4→16→64 as input goes 1→2→4→8
Real-world example: Comparing every person in a room with every other person for a group photo arrangement. In a room of 10 people, it's 100 comparisons. With 20 people, it's 400 comparisons.

6. Exponential - $O(2^n)$
What it means: Time doubles with each additional input item.
Characteristics:
- Becomes impractical very quickly
- Often seen in brute-force solutions
- Each new input item doubles the work
- Growth pattern: 1→2→4→8→16→32 as input goes 1→2→3→4→5→6
Real-world example: Trying every possible combination of a password. Each additional character position doubles the number of combinations to try.

7. Factorial - O(n!)
What it means: Time increases by multiplying all numbers from 1 to n.
Characteristics:
- The worst practical complexity
- Becomes impossible even for small inputs
- Growth pattern: 1→2→6→24→120→720 as input goes 1→2→3→4→5→6
Real-world example: Trying every possible arrangement of people in a line. With 3 people: 6 arrangements. With 10 people: 3,628,800 arrangements.

Algorithm Complexity Analysis: Linearithmic and Factorial Growth Patterns
Linearithmic O(n log n): Divide and Conquer Max Finder
How it demonstrates O(n log n):
Divide Phase: The array is split in half recursively, creating log n levels of recursion
Conquer Phase: At each level, process the entire array section (artificial work for demonstration)
Combine Phase: Merge results from both halves

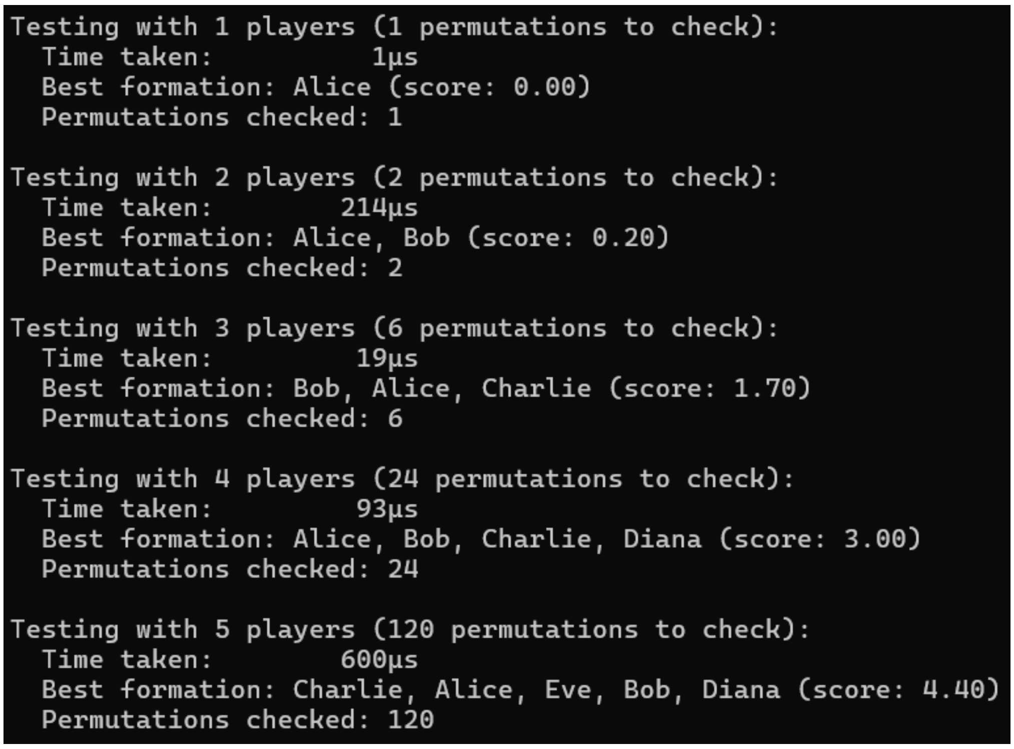
Why it's O(n log n):
Depth: log n recursive levels (each level cuts problem size in half)
Work per level: n elements are processed at each level
Total: n work × log n levels = n log n

Test results (complexity_demo.cpp):

```
Size: 2000 | Time:       25µs | Operations:    25951 | Max found: 2000
Size: 4000 | Time:       55µs | Operations:    55903 | Max found: 4000
Size: 8000 | Time:      108µs | Operations:   119807 | Max found: 8000
Size: 16000 | Time:      229µs | Operations:   255615 | Max found: 16000
Size: 32000 | Time:      488µs | Operations:   543231 | Max found: 32000
Size: 64000 | Time:     1033µs | Operations:  1150463 | Max found: 64000
Size: 128000 | Time:     2139µs | Operations:  2428927 | Max found: 128000
Size: 256000 | Time:     3242µs | Operations:  5113855 | Max found: 256000
Size: 512000 | Time:     4948µs | Operations: 10739711 | Max found: 512000
Size: 1024000 | Time:    10304µs | Operations: 22503423 | Max found: 1024000
```
Factorial O(n!): Team Formation Optimizer
How it demonstrates O(n!):
Permutation Generation: Creates every possible arrangement of players
Evaluation: For each arrangement, calculates a "chemistry score"

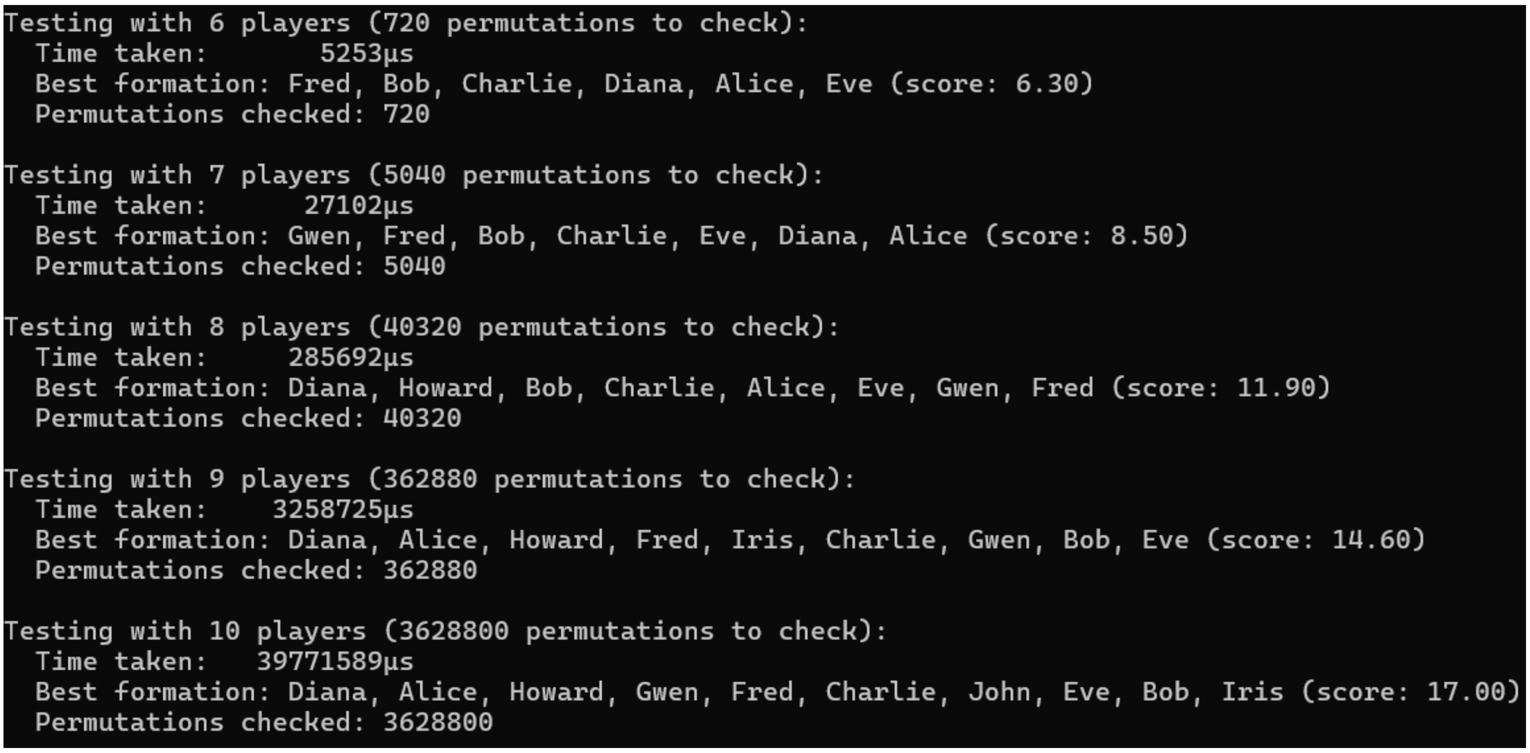
Selection: Finds the best formation from all possibilities

Why it's O(n!):
Number of arrangements: n! (factorial) possible ways to arrange n players
Work per arrangement: O(n) to calculate chemistry between players
Total: n! arrangements × n work = O(n!)

Test results (complexity_demo.cpp):



```
Testing with 1 players (1 permutations to check):
  Time taken:          1μs
  Best formation: Alice (score: 0.00)
  Permutations checked: 1

Testing with 2 players (2 permutations to check):
  Time taken:        214μs
  Best formation: Alice, Bob (score: 0.20)
  Permutations checked: 2

Testing with 3 players (6 permutations to check):
  Time taken:         19μs
  Best formation: Bob, Alice, Charlie (score: 1.70)
  Permutations checked: 6

Testing with 4 players (24 permutations to check):
  Time taken:         93μs
  Best formation: Alice, Bob, Charlie, Diana (score: 3.00)
  Permutations checked: 24

Testing with 5 players (120 permutations to check):
  Time taken:        600μs
  Best formation: Charlie, Alice, Eve, Bob, Diana (score: 4.40)
  Permutations checked: 120
```



```
Testing with 6 players (720 permutations to check):
  Time taken:       5253μs
  Best formation: Fred, Bob, Charlie, Diana, Alice, Eve (score: 6.30)
  Permutations checked: 720

Testing with 7 players (5040 permutations to check):
  Time taken:      27102μs
  Best formation: Gwen, Fred, Bob, Charlie, Eve, Diana, Alice (score: 8.50)
  Permutations checked: 5040

Testing with 8 players (40320 permutations to check):
  Time taken:     285692μs
  Best formation: Diana, Howard, Bob, Charlie, Alice, Eve, Gwen, Fred (score: 11.90)
  Permutations checked: 40320

Testing with 9 players (362880 permutations to check):
  Time taken:    3258725μs
  Best formation: Diana, Alice, Howard, Fred, Iris, Charlie, Gwen, Bob, Eve (score: 14.60)
  Permutations checked: 362880

Testing with 10 players (3628800 permutations to check):
  Time taken:   39771589μs
  Best formation: Diana, Alice, Howard, Gwen, Fred, Charlie, John, Eve, Bob, Iris (score: 17.00)
  Permutations checked: 3628800
```

b) **Recurrence Problem: T(n) = 4T(n/2) + n**

Problem Background: Signal Interference Analysis in a Singaporean Urban Grid
The Problem: In a densely populated country like Singapore, telecommunication companies (telcos) face the challenge of managing cellular signal interference. When too many cell towers are close together, or when signals reflect off dense HDB blocks and skyscrapers, their signals can interfere with each other, leading to dropped calls and slow data speeds. We need an algorithm to model and quantify this interference across a given district.
The Model: We can model a district, for example, Punggol Digital District, as a square grid of size n×n, where n is a power of two. Each cell (i,j) in the grid holds a value representing the base signal strength.
The Goal: Our algorithm will calculate a total "interference score" for the entire n×n grid. This score is derived from two sources:

1. **Sub-Region Interference:** The interference that occurs within smaller, localized areas.
2. **Cross-Border Interference:** The interference that occurs specifically along the dividing lines between these sub-regions, which is often the most critical area for analysis.

Practicality: This model is practical because it allows telco engineers to identify "hotspots" of high interference. By recursively analyzing smaller and smaller quadrants, they can pinpoint specific areas that require adjustments, such as changing a tower's signal direction or reducing its power. This divide-and-conquer approach is computationally efficient for analyzing large urban maps.

Algorithm Design and Implementation
The algorithm, calculate_interference, uses a divide-and-conquer strategy that perfectly matches the recurrence relation T(n)=4T(n/2)+n.

Algorithm Steps:

1. **Divide:** The n×n grid is divided into four equal sub-grids (quadrants) of size (n/2)×(n/2). These are the top-left, top-right, bottom-left, and bottom-right quadrants.
2. **Conquer:** The algorithm recursively calls itself on each of these four sub-grids. This step corresponds to the $4 \cdot T(n/2)$ part of the recurrence. It calculates the interference score that originates from within each smaller quadrant.
3. **Combine & Work:** After the recursive calls return the scores from the sub-grids, the algorithm performs additional work at the current level. It calculates the "cross-border" interference by iterating along the central row and central column that separate the four quadrants. This linear scan takes O(n) time and corresponds to the +n part of the recurrence. The final score is the sum of the four sub-grid scores plus this new cross-border score.

Test Results (Ass1_Q2b.cpp)
```
Grid size n = 2
Total Interference Score: 180
Total Operations: 9
-------------------------
Grid size n = 4
Total Interference Score: 584
Total Operations: 45
-------------------------
Grid size n = 8
Total Interference Score: 2783
Total Operations: 197
-------------------------
```

```
Grid size n = 16
Total Interference Score: 14380
Total Operations: 821
-------------------------
Grid size n = 32
Total Interference Score: 61394
Total Operations: 3349
-------------------------
```

From the output, we can see the number of operations grows very quickly. For n=2, it's 9. For n=4 (doubling the input), it jumps to 57 (more than 4x). For n=8, it's 321. This growth is much faster than linear (O(n)) or linearithmic (O(nlogn)) and is characteristic of a quadratic (O(n^2)) complexity, which we will now prove.

Complexity Derivation

1. Master Method

The Master Method provides a straightforward way to solve recurrence relations of the form T(n)=aT(n/b)+f(n).

1. Identify parameters:
    - a=4 (number of subproblems)
    - b=2 (factor by which input size is reduced)
    - f(n)=n (cost of work done outside recursive calls)
2. Compare f(n) with $n^{log_b a}$:
    - Calculate $n^{log_b a}$ = $n^{log_2 4}$=n^2.
3. Apply the correct case:
    - We compare f(n)=n with n^2.
    - Clearly, n=O($n^{2-\epsilon}$) for ϵ = 1.
    - This fits Case 1 of the Master Theorem, which states that if f(n)=O($n^{log_b a-\epsilon}$) for some constant ϵ > 0, then T(n)=Θ($n^{log_b a}$).
4. Conclusion:
    - The time complexity is T(n)=Θ(n^2).

2. Recursion-Tree Method

This method visualizes the cost at each level of recursion.

- Level 0 (Root): One problem of size n. The cost of work is n.
- Level 1: Four subproblems, each of size n/2. The cost of work at this level is 4×(n/2)=2n.
- Level 2: Sixteen subproblems (4^2), each of size n/4. The cost is 16×(n/4)=4n.
- Level i: There are 4^i nodes, each for a problem of size n/2^i. The total cost at level i is 4^i×(n/2^i)=n$\cdot$ 2^i.

The tree depth is $log_2 n$. The total cost is the sum of costs at all levels, plus the cost of the leaf nodes.

- Sum of non-leaf levels:
  $(i=0)\sum(log_2 n-1)(n \cdot 2^i) = n(i=0)\sum(log_2 n-1)2^i$
  This is a geometric series which sums to $n \cdot (2^{log_2 n}-1)/(2-1) = n \cdot (n-1) = n^2 - n$.
- Cost of leaf nodes: The number of leaves is $4 log_2 n = (2^2)\log 2n = (2^{log_2 n})^2 = n^2$. Each leaf corresponds to a base case T(1) which takes constant time, so the total cost at the leaves is $\Theta(n^2)$.
- Total Cost: (Cost of non-leaf levels) + (Cost of leaf nodes) = $(n^2-n)+\Theta(n^2)=\Theta(n^2)$.

The cost is dominated by the vast number of leaf nodes, confirming $T(n)=\Theta(n^2)$.

3. Substitution Method

We guess the solution is $T(n)=O(n^2)$ and prove it by induction.

1. Guess: Assume $T(n) \le cn^2$ for some constant $c>0$.
2. Inductive Hypothesis: Assume the guess holds for all values smaller than n, specifically for n/2. So, $T(n/2) \le c(n/2)^2 = cn^2/4$.
3. Substitution: Substitute the hypothesis into the recurrence:
   $T(n)=4T(n/2)+n$
   $T(n) \le 4(cn^2/4)+n$
   $T(n) \le cn^2+n$
   This result, cn^2+n, is not $\le cn^2$. The proof fails. We must strengthen the hypothesis by subtracting a lower-order term.
4. Strengthened Guess: Assume $T(n) \le cn^2 - dn$ for constants $c,d>0$.
   - New Hypothesis: $T(n/2) \le c(n/2)^2 - d(n/2) = cn^2/4 - dn/2$.
   - New Substitution:
     $T(n) \le 4(cn^2/4 - dn/2)+n$
     $T(n) \le cn^2 - 2dn + n$
     $T(n) \le cn^2 - (2d-1)n$
   - We need to show this is less than or equal to our guess, cn^2-dn:
     $cn^2-(2d-1)n \le cn^2-dn$
     $-(2d-1)n \le -dn$
     $(2d-1)n \ge dn$
     $2d-1 \ge d$
     $d \ge 1$
5. Conclusion: The inductive step holds as long as we choose a constant $d \ge 1$. We can also choose a large enough c to handle the base cases. Therefore, we have proven that $T(n)=O(n^2)$.

Question 3:
Completed by Halis Ilyasa Bin Amat Sarijan 2301333 (part a) and Chew Bangxin Steven 2303348 (part b)

a) Stability in a sorting algorithm refers to the relative order of elements with equal keys being preserved. For example, if we sort people by age, and two people are both 25 years old, then in a stable sort they will appear in the same order as in the input list.

When sorting data on more than one field (e.g. name, age, department), it is important that the order of sorting is retained. For example, if employees are first sorted by department and then sorted by name, a stable sort will keep employees within the same department grouped together while still arranging them alphabetically. Without stability, the naming sort could scramble the department grouping, making the overall order meaningless. This shows that stability is essential when multiple sorting operations are layered on the same dataset, as it guarantees that the results of previous sorts remain intact.

Name	**Department**
Charlie	Sales
Alice	Engineering
Bob	Sales
Diana	Engineering

Input List

Name	**Department**
Alice	Engineering
Bob	Sales
Charlie	Sales
Diana	Engineering

Sort by name first (secondary key)

Name	**Department**
Alice	Engineering
Diana	Engineering
Bob	Sales
Charlie	Sales

Sort by department (primary key)

Stability is also important when a dataset already has a meaningful pre-existing order that must be preserved. For example, student records may initially be ordered by student ID but later need to be sorted by grade. A stable sort will ensure that students with the same grade remain in their original ID order, while an unstable sort could shuffle them randomly, creating confusion and inconsistency. This shows that stability is crucial for maintaining existing order within data, ensuring that information remains reliable and easy to interpret after sorting.

Student ID	**Grade**
001	B
002	A
003	B
004	A

Original order

Student ID	**Grade**
002	A
004	A
001	B
003	B

After sorting by grade

We will use counting sort for stable sort and selection sort for unstable sort. Counting sort preserves the relative order of elements with equal keys by placing them into the output array in a controlled sequence, while selection sort may swap equal elements during its process, potentially changing their original order.

Counting Sort counts the number of occurrences of each key, then computes cumulative counts, and places each element into the correct position. It does not compare elements, which is why it's very fast for integers or small discrete ranges.

COUNTING-SORT(A, B, k)

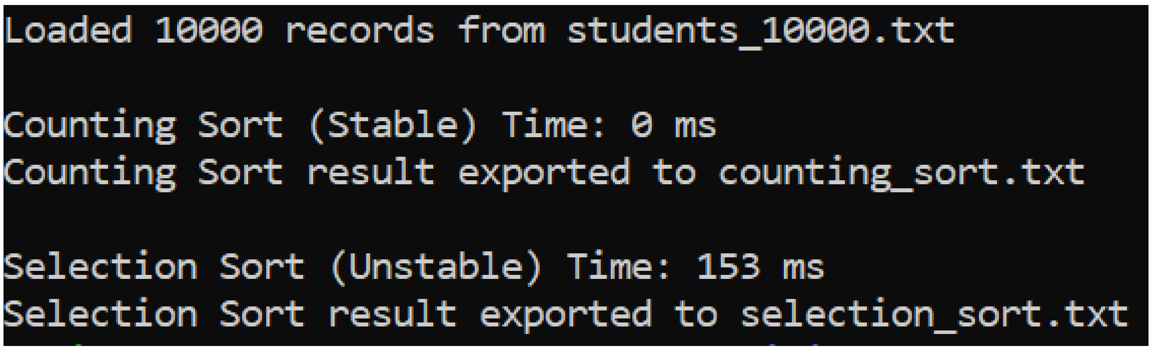
```
1. let C[0..k] be a new array initialized to 0
2. for j = 1 to n
       C[A[j]] = C[A[j]] + 1
3. for i = 1 to k
       C[i] = C[i] + C[i-1]
4. for j = n to 1
       B[C[A[j]]] = A[j]
       C[A[j]] = C[A[j]] - 1
```

Total time complexity: O(n+k)

Selection Sort repeatedly finds the minimum element from the unsorted part and swaps it with the first unsorted element. This process continues until the array is sorted.

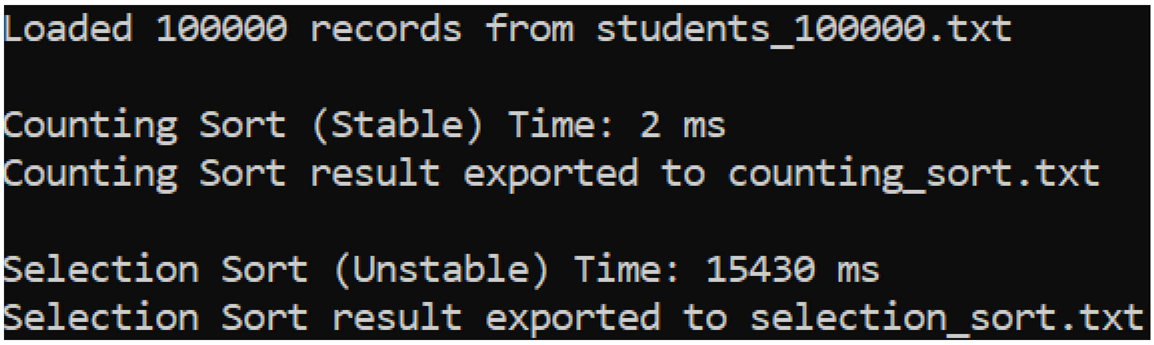
```
SELECTION-SORT(A)
1. let n = length(A)
2. for i = 0 to n-2
       minIndex = i
3.     for j = i+1 to n-1
           if A[j] < A[minIndex]
               minIndex = j
4.     swap A[i] with A[minIndex]
```

Total time complexity: $O(n^2)$



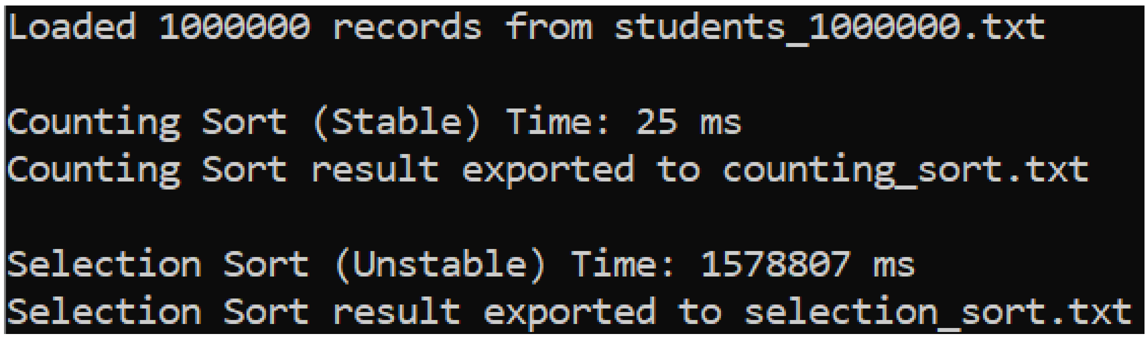
```
Loaded 10000 records from students_10000.txt

Counting Sort (Stable) Time: 0 ms
Counting Sort result exported to counting_sort.txt

Selection Sort (Unstable) Time: 153 ms
Selection Sort result exported to selection_sort.txt
```

Counting sort at 10k dataset: <0.001s
Selection sort at 10k dataset: 0.153s



```
Loaded 100000 records from students_100000.txt

Counting Sort (Stable) Time: 2 ms
Counting Sort result exported to counting_sort.txt

Selection Sort (Unstable) Time: 15430 ms
Selection Sort result exported to selection_sort.txt
```

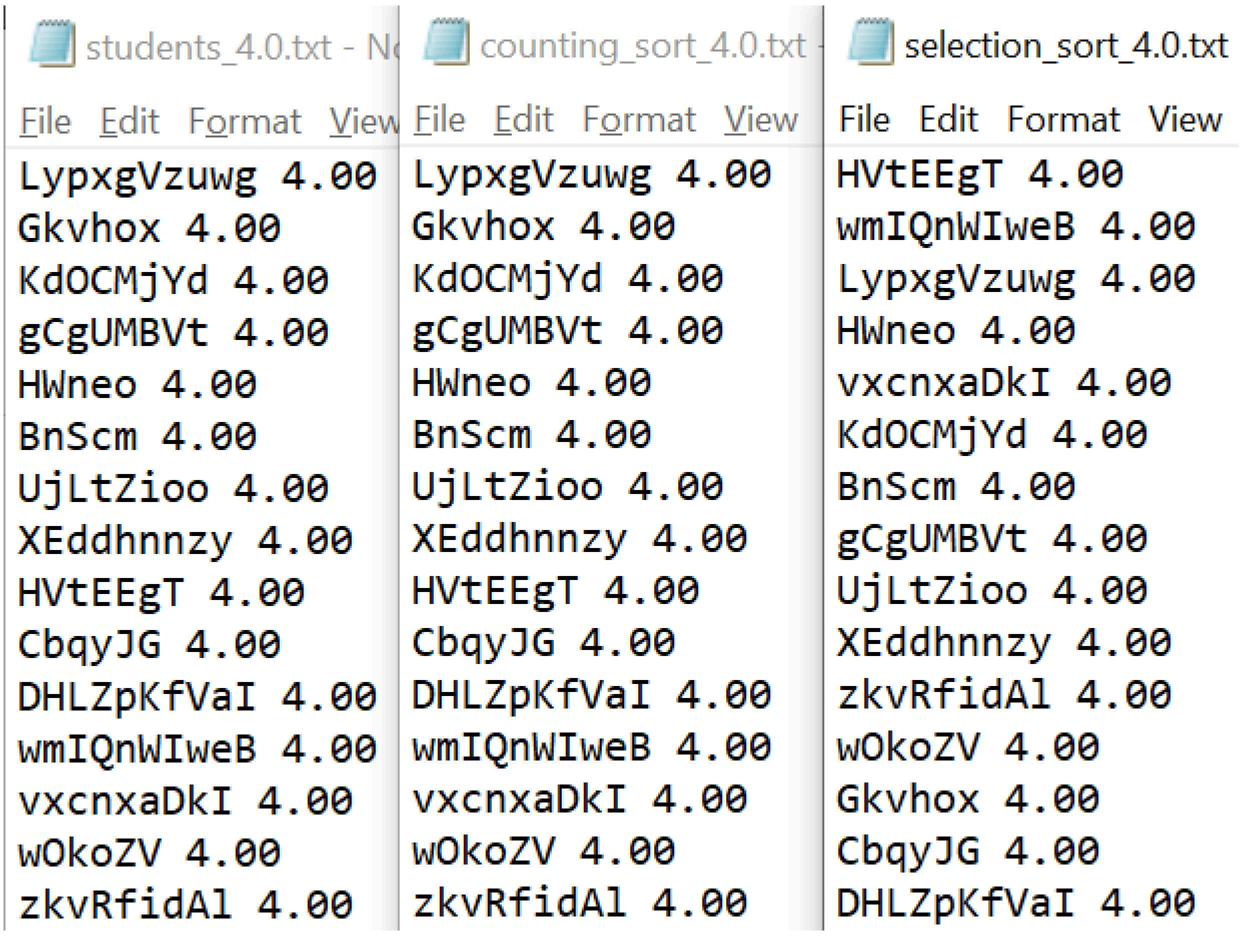
Counting sort at 100k dataset: 0.002s
Selection sort 100k dataset: 15.43s



```
Loaded 1000000 records from students_1000000.txt

Counting Sort (Stable) Time: 25 ms
Counting Sort result exported to counting_sort.txt

Selection Sort (Unstable) Time: 1578807 ms
Selection Sort result exported to selection_sort.txt
```

Counting sort at 1M dataset: 0.025s
Selection sort at 1M dataset: ~26mins

For a 1 million dataset, counting sort completes in milliseconds because it only counts occurrences and does not compare elements $O(n+k)$, making it extremely efficient for small-range keys. Selection sort, on the other hand, performs $O(n^2)$ comparisons and swaps. This translates to about 10^{12} operations, which would take hours to complete. This illustrates why counting sort is practical for large datasets while selection sort is suitable only for smaller datasets.


students_4.0.txt - N
File Edit Format View

```
LypxgVzuwg 4.00
Gkvhox 4.00
KdOCMjYd 4.00
gCgUMBVt 4.00
HWneo 4.00
BnScm 4.00
UjLtZioo 4.00
XEddhnnzy 4.00
HVtEEgT 4.00
CbqyJG 4.00
DHLZpKfVaI 4.00
wmIQnWIweB 4.00
vxcnxaDkI 4.00
wOkoZV 4.00
zkvRfidAl 4.00
```

counting_sort_4.0.txt
File Edit Format View

```
LypxgVzuwg 4.00
Gkvhox 4.00
KdOCMjYd 4.00
gCgUMBVt 4.00
HWneo 4.00
BnScm 4.00
UjLtZioo 4.00
XEddhnnzy 4.00
HVtEEgT 4.00
CbqyJG 4.00
DHLZpKfVaI 4.00
wmIQnWIweB 4.00
vxcnxaDkI 4.00
wOkoZV 4.00
zkvRfidAl 4.00
```

selection_sort_4.0.txt
File Edit Format View

```
HVtEEgT 4.00
wmIQnWIweB 4.00
LypxgVzuwg 4.00
HWneo 4.00
vxcnxaDkI 4.00
KdOCMjYd 4.00
BnScm 4.00
gCgUMBVt 4.00
UjLtZioo 4.00
XEddhnnzy 4.00
zkvRfidAl 4.00
wOkoZV 4.00
Gkvhox 4.00
CbqyJG 4.00
DHLZpKfVaI 4.00
```

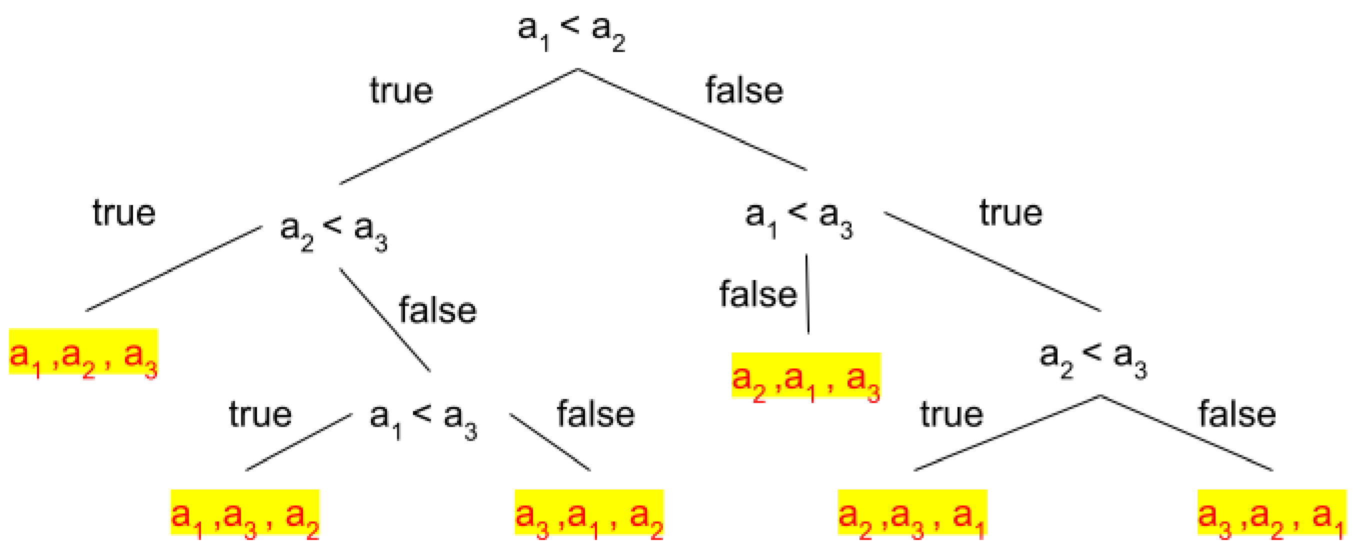
Students with 4.0 GPA filtered. Counting sort retains the original order from the input file whilst selection sort does not retain the original order.

Counting Sort is stable because it places elements into the output array according to their cumulative counts, iterating from the end of the input array. This ensures that elements with the same key appear in the same relative order as in the original dataset. Selection Sort is unstable because it repeatedly swaps the minimum element with the current position in the array, which can change the order of elements with equal keys. Thus, this shows that counting sort is more preferable for order retention over selection sort.

b) Most of the sorting algorithms have a O(n^2) complexity. The best algorithm has achieved a O(n lg n) complexity. It is shown that sorting algorithm can never be better than O(n lg n).

This statement is true if the algorithm uses comparison-based sorting where the relative order of the elements is only known through the evaluation of $a_i < a_j$. Suppose an unsorted array of n elements, the number of permutations in random order that the array can be represented is n!. Each comparison operation evaluates the pairwise ordering and is asymptotically O(1). To evaluate the entire sample space of orderings which is n!, the minimum number of comparisons, represented by the variable c, needed would be $2^c \geq n!$. By taking log_2, the expression evaluates to $c \geq log_2 n!$. Using the stirling approximation, $log_2(\sqrt{2\pi n}(\frac{n}{e})^n) = nlog_2 n - nlog_2 e + \frac{1}{2}log_2(2\pi n) = O(nlog_2 n)$. $c \geq nlog_2 n$. Thus, the worst case performance requires a comparison complexity of at least $O(nlog_2 n)$.

Decision Tree proof
Suppose an unsorted array, n = 3.
A = [a_1, a_2, a_3], where A is unsorted.
Starting with pairwise comparison a_1, a_2.



From the decision tree, we can see that the number of leaf is n! (3! = 6) and that at least $2^c \geq n!$ comparisons are required to sort the array. In this example the minimum number of comparisons is c = 2. Using this equation, we can compute the number of comparisons required to sort an array of n size asymptotically. Since the pairwise comparison operation is linear time complexity, the total time complexity is O(c)O(1).

Computing c(n) the number of comparisons to sort an array of size n

Since $2^c \geq n!$

$$c * log_2(2) \geq log_2 n!$$

Using stirling approximation,

$$log_2 n! = log_2(\sqrt{2\pi n}(\frac{n}{e})^n) = nlog_2 n - nlog_2 e + \frac{1}{2}log_2(2\pi n) = O(nlog_2 n)$$

$$c \geq nlog_2 n$$

$$c(n) \geq O(nlog_2 n)$$

Radix sort

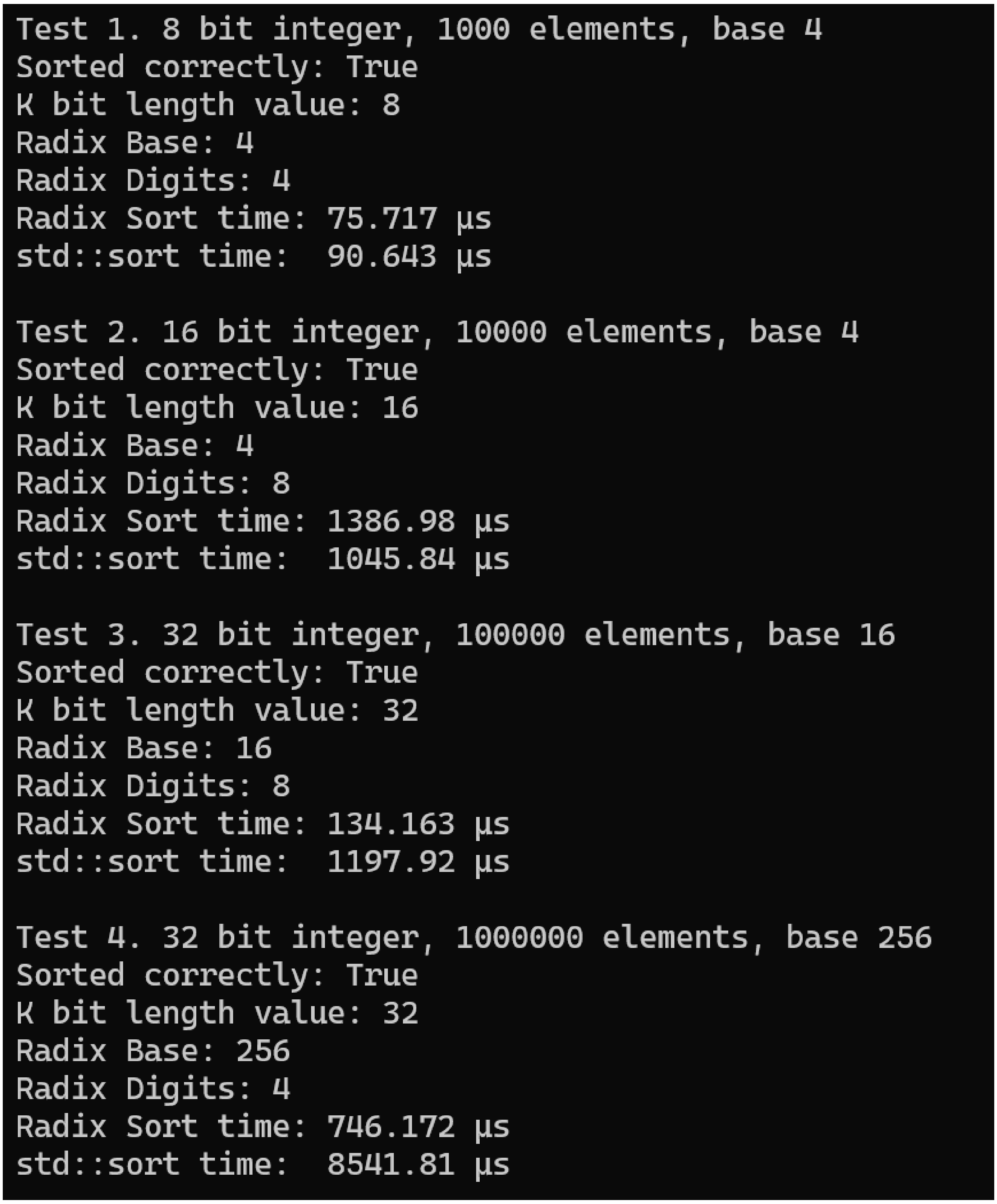
Pseudocode

```
RadixSort(int[] : arr, int : base):
    Max = FindMax(arr)
    Exp = 1
    While Max / Exp > 0:
        CountingSortDigit(arr, Exp, base)
        Exp*=base

CountingSortDigit(int[] : arr, int : Exp, int : base):
    Len = Size(arr)
    For i from 0 to Len:
        Digit = (arr[i] / Exp) mod base
        Count[Digit]++
    For d from 1 to base-1:
        Count[d] = Count[d] + Count[d+1]
    For i from Len-1 down to 0:
        Digit = (arr[i] / Exp) mod base
        Pos = Count[Digit] - 1
        Out[Pos] = arr[i]
        Count[Digit] = Count[Digit] - 1
   arr = Out
```

Limitations

- The program only works for integer
- The program only supports up to 32bit integer as values
- The sorting algorithm is not an inplace sorting algorithm. The space complexity is O(n + base)

Radix sort implementation tests (3b.cpp)



```
Test 1. 8 bit integer, 1000 elements, base 4
Sorted correctly: True
K bit length value: 8
Radix Base: 4
Radix Digits: 4
Radix Sort time: 75.717 µs
std::sort time:  90.643 µs

Test 2. 16 bit integer, 10000 elements, base 4
Sorted correctly: True
K bit length value: 16
Radix Base: 4
Radix Digits: 8
Radix Sort time: 1386.98 µs
std::sort time:  1045.84 µs

Test 3. 32 bit integer, 100000 elements, base 16
Sorted correctly: True
K bit length value: 32
Radix Base: 16
Radix Digits: 8
Radix Sort time: 134.163 µs
std::sort time:  1197.92 µs

Test 4. 32 bit integer, 1000000 elements, base 256
Sorted correctly: True
K bit length value: 32
Radix Base: 256
Radix Digits: 4
Radix Sort time: 746.172 µs
std::sort time:  8541.81 µs
```

Analysis

Radix sort performs with near linear (n) time complexity when the number of digits with respect to the base is small. When compared to std::sort that uses a comparison based algorithm such as quicksort (n lg n) in its implementation, Radix sort performs an order of magnitude faster than comparison based sorting algorithm asymptotically. This is because Radix sort processes a digit at a time using a counting sort algorithm which has a linear time complexity. The counting sort algorithm does not perform any pairwise comparison among elements. Instead it counts the occurrence in a fixed length allowing linear time complexity. Thus, the complexity of Radix sort is O(n * k), where k is the number of digits. If the digits are small, it is able to achieve near linear time complexity asymptotically. However, when the data size is small and digits are high, Radix sort might perform worse than comparison sort as seen in the test data