# *CSCI 411 - Advanced Algorithms - Assignment 1*

1) **Asymptotic ordering (smallest → largest)**

**Θ(1)**:
2, 52! *(both constants; no growth)*

**Θ(ln n)**:
ln(n^2) *(= 2 ln n)*

**Θ((ln n)^2)**:
ln^2(n)

**Θ(n^α) (polynomial, α=1/5 then 1 then 3)**:
n^{1/5} < n < n^3

**Θ(n log n)**:
n ln(n), $\log_2((4n)^n)$ *(since $\log_2((4n)^n) = n \cdot \log_2(4n) = 2n + n \cdot \log_2 n = \Theta(n \log n)$)*

**Exponentials**:
(3/2)^n < 2^n

**Super-exponential**:
n!

**Exact f1=O(f2),f2=O(f3),…" order:**

***ORDER (smallest → largest):***

***1) 2***

***2) 52!***

***3) ln(n^2)***

***4) (ln n)^2***

***5) n^(1/5)***

***6) n***

***7) n ln n***

***8) log_2((4n)^n)***________Evaluating:= n*log_2(4n) = 2n + n*log_2 n = Θ(n log n)

***9) n^3***

***10) (3/2)^n***

***11) 2^n***

***12) n!***

**Grouping Equivalent:**

{2, 52!} < ln(n^2) < ln^2(n) < n^{1/5} < n < {n ln n, log2((4n)^n)} < n^3 < (3/2)^n < 2^n < n!

**2)**

**a)**

```
QuickSortPivotLast(A):
   # A is a list of real numbers
   if length(A) ≤ 1 then
      return A
   p ← A[last]                # pivot = last element
   L ← empty list             # elements ≤ p
   R ← empty list             # elements > p
   for i ← 0 to length(A)−2 do # all except the last element
      e ← A[i]
      if e ≤ p then
         append e to L
      else
         append e to R
   L' ← QuickSortPivotLast(L)  # recursively sort L
   R' ← QuickSortPivotLast(R)  # recursively sort R
   return concatenate(L', [p], R')
```

```
"""
QuickSortPivotLast(A):
    # A is a list of real numbers
    if length(A) ≤ 1 then
        return A

    p ← A[last]                  # pivot = last element
    L ← empty list               # elements ≤ p
    R ← empty list               # elements > p

    for i ← 0 to length(A)-2 do # all except the last element
        e ← A[i]
        if e ≤ p then
            append e to L
        else
            append e to R

    L′ ← QuickSortPivotLast(L)  # recursively sort L
    R′ ← QuickSortPivotLast(R)  # recursively sort R

    return concatenate(L′, [p], R′)
```

b)

Worst case: Θ(n^2).

, Explanation:

This is the algorithm:

Given a list A:

1) If len(A) ≤ 1, return A (already sorted).

2) Let p = last element of A (the pivot).

3) Scan all other elements once:

   - If e ≤ p, put e into L

   - If e > p, put e into R

4) Recursively sort L and R to get L' and R'

5) Return L' + [p] + R'

“Worst case” is an input that makes the algorithm do as much work as possible.

With this pivot rule (“always use the last element” and send $e \le p$ to L), the worst case is when the partition is *maximally unbalanced*: one side has $n-1$ elements and the other has 0.

Which means:

If the array is:

- Sorted ascending: $[a1 \le a2 \le \ldots \le an]$
  Pivot $p = an$ (the maximum). Every other element is $\le p \rightarrow$ all $n-1$ go to L, $R = \emptyset$.
- Sorted descending: pivot is the minimum. Every other element goes to R $\rightarrow L = \emptyset$, $R = n-1$.
- All equal: every element satisfies $e \le p \rightarrow$ all $n-1$ go to L ($R = \emptyset$).

Because the pivot choice never improves, each recursive step reduces the problem size by only 1 ($n \rightarrow n-1 \rightarrow n-2 \rightarrow \ldots \rightarrow 1$).

Each level of recursion still scans its whole sublist once, so:

Total work = $n + (n-1) + (n-2) + \ldots + 1 = n(n+1)/2 \approx (1/2)\cdot n^2 \rightarrow \Theta(n^2)$

**Recurrence (worst case):**

Base: $T(0) = T(1) = \Theta(1)$,

- Worst case partition: sizes $(n-1, 0)$
- One linear scan per call: $\Theta(n)$

$T(n) = T(n-1) + T(0) + \Theta(n) = T(n-1) + \Theta(n)$,

where $\Theta(n)$ is the linear work to scan/partition (and concatenate).

This results in:

$T(n) = \Theta(n) + \Theta(n-1) + \ldots + \Theta(1) = \Theta(n^2)$.

Therefore, the worst-case asymptotic runtime is $\Theta(n^2)$.

2c)

Let $T(n)$ be the runtime on an input of size n.

Base cases:

$T(0) = \Theta(1)$

$T(1) = \Theta(1)$

For $n \geq 2$:

# remove pivot (1 element), split the other n−1 elements evenly, and do a linear scan

$$T(n) = T(\lfloor(n-1)/2\rfloor) + T(\lceil(n-1)/2\rceil) + \Theta(n)$$

Equal-sizes shorthand (common, when rounding is ignored):

$$T(n) = 2 \cdot T((n-1)/2) + \Theta(n)$$

Standard simplified form (asymptotically equivalent):

$$T(n) = 2 \cdot T(n/2) + \Theta(n)$$

2d)

$T(n) = 2\ T(n/2) + \Theta(n)$

We compare to the form: $T(n) = a\ T(n/b) + f(n)$

Parameters:

$a = 2,\ \ b = 2,\ \ f(n) = \Theta(n)$

Critical exponent:

$n^{\log_b a} = n^{\log_2 2} = n^1 = n$

Comparison:

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

This matches **Case 2** of the Master Theorem (the "balanced" case):

If $f(n) = \Theta(n^{\log_b a} \cdot \log^k n)$ with $k = 0$,

then $T(n) = \Theta(n^{\log_b a} \cdot \log^{k+1} n) = \Theta(n \cdot \log n)$.

Conclusion:

The asymptotic runtime is:

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

3)

a) A, B and C:

**Edge u → v**: "u is **at least as good** as v head-to-head" (u has beaten v ≥ v has beaten u).

**Weakly dominates** u; v: there's a **path** from u to v (maybe via other players). So, u is **not worse than** v by chaining comparisons.

This is like a strongly connected component such that group players who can reach each other both ways like from u to v and v to u.

**Set A:**

A is the set of players u such that (1) if v ; u, then u ; v and (2) there is some player w
such that u ; w but w 6 ; u.

This set is a group such that no outside group players can reach, which means there are no arrows into the group and the group has at least one outgoing arrow.

**Set B:**

**B is the set of players u such that (1) if u ; v, then v ; u and (2) there is some player w**
**such that w ; u but u 6 ; w**

Players in a **group that cannot reach any outside group** (no outgoing arrows), **and some outside group reaches them** (has an incoming arrow).

**Set C:**

**C is the set of all players not included in A or B**

Everyone not in A or B. This includes groups with **both** incoming and outgoing arrows (true "middle"), and groups with **no arrows at all**

**Finding the size:**

**1) Draw arrows between players**

**For every pair (u, v):**

- **Draw an arrow u → v if u has at least as many wins vs v as v has vs u.**
  **(So u is *at least as good* as v head-to-head.)**

**2) Make "groups" of tied/cyclic players**

**If we can travel by arrows both ways between two players (u reaches v *and* v reaches u), put them in the same group.**

- **A group can be one player (if no two-way reach) or multiple players (if there's a loop/tie among them).**

**3) Look only between groups**

**Now we treat each group like one super-player.**
**Draw an arrow Group X → Group Y if any player in X had an arrow to any player in Y.**

**4) Classifying groups, then counting players**

- **Top group (goes to A):**
  **No arrows coming in from other groups, and at least one arrow going out to another group.**
  **→ All players in these groups are in A.**
- **Bottom group (goes to B):**
  **At least one arrow coming in, and no arrows going out to other groups.**
  **→ All players in these groups are in B.**
- **Everything else (goes to C):**
  **Groups that have both in and out arrows (they're in the middle), or no arrows at all (isolated tie-group).**
  **→ All players in these groups are in C.**

**Finally:**

- **$|A|$ = add up the number of players inside all top groups.**
- **$|B|$ = add up the number of players inside all bottom groups.**
- **$|C|$ = everyone else = total players − $|A|$ − $|B|$.**

<u>**An example to show how I approached it:**</u>

Players: P1, P2, P3, P4, P5

Arrows (from head-to-head records):

- P1 ↔ P2 (two-way arrows)
- P1 → P3, P2 → P3
- P1 → P4, P2 → P4
- P4 ↔ P5

Step 2 (make groups):

- Group G1 = {P1, P2} (they reach each other both ways)
- Group G2 = {P3}
- Group G3 = {P4, P5} (they reach each other both ways)

Step 3 (between groups):

- From G1 to G2 (because P1 or P2 had arrows to P3)
- From G1 to G3 (because P1 or P2 had arrows to P4; P4 ties with P5 so it counts as the same group)
- No arrows from G2 to anywhere; no arrows from G3 to anywhere.

Step 4 (classify):

- G1 has no incoming arrows and does have outgoing → Top group → players {P1, P2} go to A.
- G2 has incoming (from G1) and no outgoing → Bottom group → player {P3} goes to B.
- G3 has incoming (from G1) and no outgoing → Bottom group → players {P4, P5} go to B.

Counts:

- |A| = 2 (P1, P2)
- |B| = 3 (P3, P4, P5)
- |C| = 0 (no middle/isolated groups in this example)

If we added a player P6 with no arrows to or from anyone, they'd form their own group with no in and no out—that group goes to C. Then |C| would be 1.

3b) ***Explaining everything using comments line by line and providing a picture of exact code without comments below explanation:***

```
getSetSizes(G):
   # Input: directed graph G = (V, E)
   # Output: (|A|, |B|, |C|)

   # 1) Building groups (SCCs) and between-group edges.
   (U, F, compId) ← makeSCCs(G)      # U: list of groups, F: edges between groups, compId[v]: group of v
   # Each U[i] is the set/list of original players in group i
   #F: a list of directed edges between groups (pairs (x, y) meaning group x → group y)
   #compId: a map/array so that compId[v] is the index i of the group U[i] that player v belongs to.
   k ← |U|

   # 2) Counting players in each group.
   size[0 to k−1] ← 0 # k is number of groups
   #Making an array called size with one slot per group (indices 0, 1, to k−1).
```

*#Initializing all counts to 0. (So size[u] will store "how many players are in group u".)*

*for each v in V do #going through every player v*

*u ← compId[v] # looking up which group u player v belongs to*

*size[u] ← size[u] + 1 # incrementing the count for group u*

*# 3) Compute in/out-degrees for each group.*

*indeg[0..k−1] ← 0  # making two arrays of length k, filling with zeros.*

*outdeg[0..k−1] ← 0*

*for each (x, y) in F do*

*outdeg[x] ← outdeg[x] + 1 #Increasing outdeg[x] because x → y means x has one more outgoing arrow.*

*indeg[y]  ← indeg[y]  + 1 #Increasing indeg[y] because y has one more incoming arrow.*

*# 4) Classify groups and sum sizes.*

*sizeA ← 0; sizeB ← 0; sizeC ← 0 #counters for sizes of sets A, B, C , Each will hold the number of players in A, B, and C.*

*for u from 0 to k−1 do #Going through each group u*

*#Checking if group u has no incoming edges (indeg[u]=0) and at least one outgoing (outdeg[u]>0) in the group-graph.*

*#That means no outside group can reach u, but u does reach someone → this is a TOP group (belongs to set A).*

*if indeg[u] = 0 and outdeg[u] > 0 then*

*#Adding all players in this group to A's total. (size[u] was computed earlier as the number of players in group u.)*

*sizeA ← sizeA + size[u]      # TOP → A*

*#Otherwise, if there is no outgoing and some incoming, then others reach u but u reaches no one → this is a BOTTOM group (set B).*

*else if outdeg[u] = 0 and indeg[u] > 0 then*

*sizeB ← sizeB + size[u]      # BOTTOM → B*

*#All remaining cases: either both incoming and outgoing (a middle group), or neither incoming nor outgoing (an isolated group).*

*else*

*sizeC ← sizeC + size[u]     # MIDDLE or ISOLATED → C (includes in=out=0)*

*return (sizeA, sizeB, sizeC)*

***clear code:***

```
getSetSizes(G):
    (U, F, compId) ← makeSCCs(G)
    k ← |U|

    size[0..k-1] ← 0
    for each v in V do
        u ← compId[v]
        size[u] ← size[u] + 1

    indeg[0..k-1] ← 0
    outdeg[0..k-1] ← 0
    for each (x, y) in F do
        outdeg[x] ← outdeg[x] + 1
        indeg[y]  ← indeg[y]  + 1

    sizeA ← 0; sizeB ← 0; sizeC ← 0
    for u from 0 to k-1 do
        if indeg[u] = 0 and outdeg[u] > 0 then
            sizeA ← sizeA + size[u]
        else if outdeg[u] = 0 and indeg[u] > 0 then
            sizeB ← sizeB + size[u]
        else
            sizeC ← sizeC + size[u]

    return (sizeA, sizeB, sizeC)
```

3C) Let:

- **n = |V|** = number of players (vertices) in the original graph
- **m = |E|** = number of head-to-head “at least as many wins” edges
- After grouping ties into SCCs, let **k = |U|** (number of groups) and **f = |F|** (edges between groups).

**Step 1 — Building SCC groups and the between-group graph**

- Work: makeSCCs(G) visits each vertex and edge a constant number of times.
- Time: **O(n + m)**

**Step 2 — Count players in each group**

- Loop over every original vertex once; increase a counter for its group.
- Time: **O(n)**

**Step 3 — Computing "has incoming / has outgoing" for each group**

- Initialized two arrays of length k: **O(k)**
- Scanning each between-group edge (x, y) once; update outdeg[x], indeg[y]: **O(f)**
- Since k ≤ n and f ≤ m, total here is **O(k + f) ⊆ O(n + m)**

**Step 4 — Classify each group and add its size**

- One pass over the k groups to decide A/B/C and sum sizes.
- Time: **O(k) ⊆ O(n)**

**⊆ - subset or equal and the above steps are just a pseudo code simplification to compute TC and**

**Total time** $= O(n + m) + O(n) + O(n + m) + O(n) = O(n + m)$

3d)

rankPlayers(G):

*#makeSCCs(G) returns U: list of SCCs (groups). k is how many groups. F: edges between SCCs (group x → group y).compId[v]: which SCC (index) each original vertex v belongs to*

(U, F, compId) ← makeSCCs(G)

k ← |U|

*members[0..k−1] ← empty lists*

*for each v in V do*

```
    u ← compId[v]
    append v to members[u]
```

*# Making group (members[u]) for each SCC u, and drop each player into the group of their SCC. (Within a group, players are tied, so any order among them is fine.. For each metagraph edge x → y: increase outdeg[x] and indeg[y]. ,Record y in adj[x] so we can traverse later.*

```
  indeg[0..k−1] ← 0
  outdeg[0..k−1] ← 0
  adj[0..k−1] ← empty lists
  for each (x, y) in F do
    outdeg[x] ← outdeg[x] + 1
    indeg[y]  ← indeg[y]  + 1
    append y to adj[x]
```

# **A (top)**: no one points into (indeg=0), but you point to someone (outdeg>0).

#  **B (bottom)**: others point to (indeg>0), but you point to no one (outdeg=0).

# **C (middle/isolated)**: everything else (both in & out, or neither ⇒ isolated).

```
  label[0..k−1]
  for u from 0 to k−1 do
    if indeg[u] = 0 and outdeg[u] > 0 then
      label[u] ← "A"
    else if outdeg[u] = 0 and indeg[u] > 0 then
      label[u] ← "B"
    else
      label[u] ← "C"
```

*# Copying indegrees to indeg2 so we don't lose the original labels info. Starting with all SCCs that have no incoming edges (indeg2=0). Repeatedly taking one, putting it in topo, and "remove" its outgoing edges by decrementing neighbors' indegrees; when a neighbor hits 0, push it.  Result: topo lists SCCs so that every edge goes from earlier to later—perfect for ranking by reachability.*

```
indeg2[0..k−1] ← indeg
 Q ← all u with indeg2[u] = 0
 topo ← empty list
 while Q not empty do
   x ← pop(Q)
   append x to topo
   for each y in adj[x] do
     indeg2[y] ← indeg2[y] − 1
     if indeg2[y] = 0 then push(Q, y)
```

# We pass through the **same topo order three times**: Append all **A** groups (so A's come first, and still respect topo among themselves). Then all **C** groups (middle). Finally all **B** groups (last). When we "append members[u]", we dump the actual player IDs of that SCC into the result.
Inside one SCC, order doesn't matter (they're tied).

```
order ← empty list
for each u in topo do
  if label[u] = "A" then append members[u] to order
for each u in topo do
  if label[u] = "C" then append members[u] to order
for each u in topo do
```

```
    if label[u] = "B" then append members[u] to order

return order
```