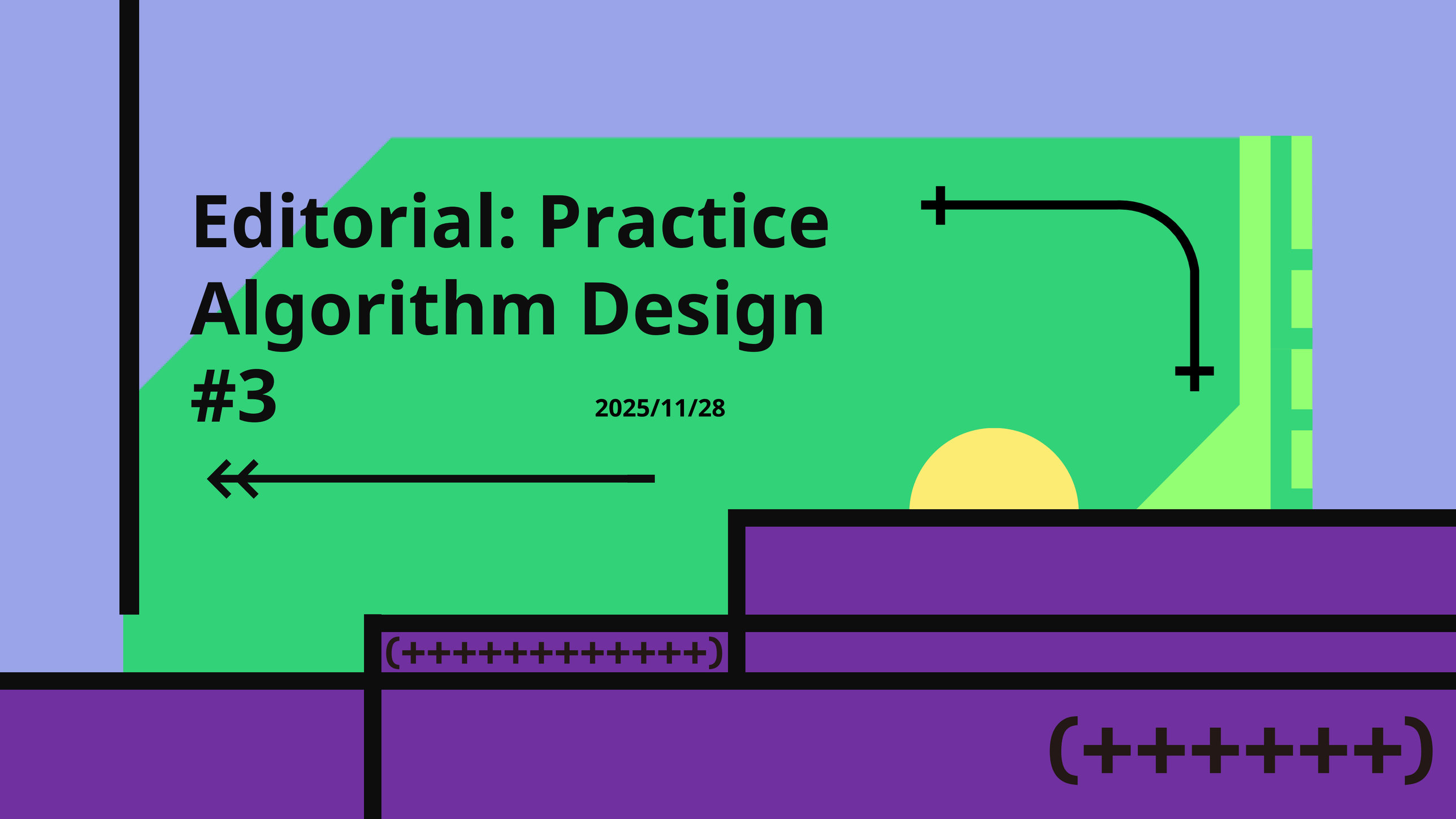
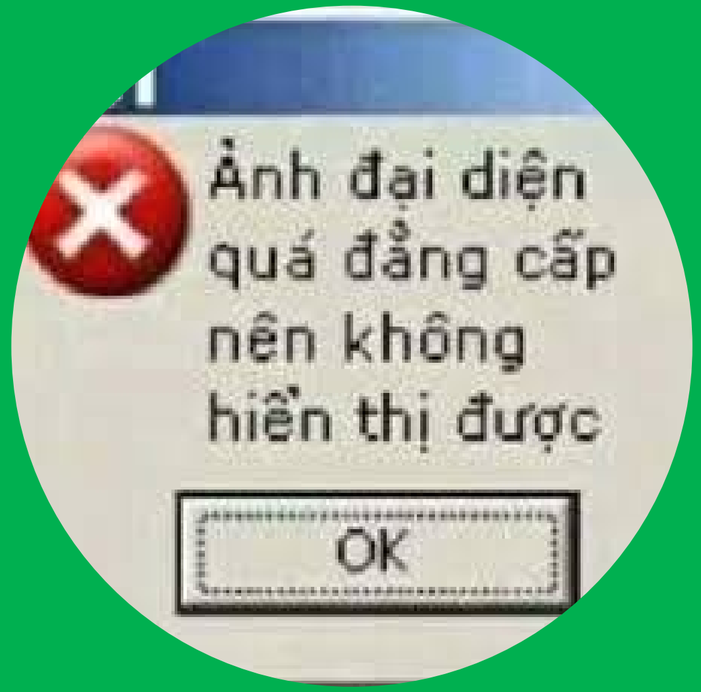
Editorial: Practice Algorithm Design #3

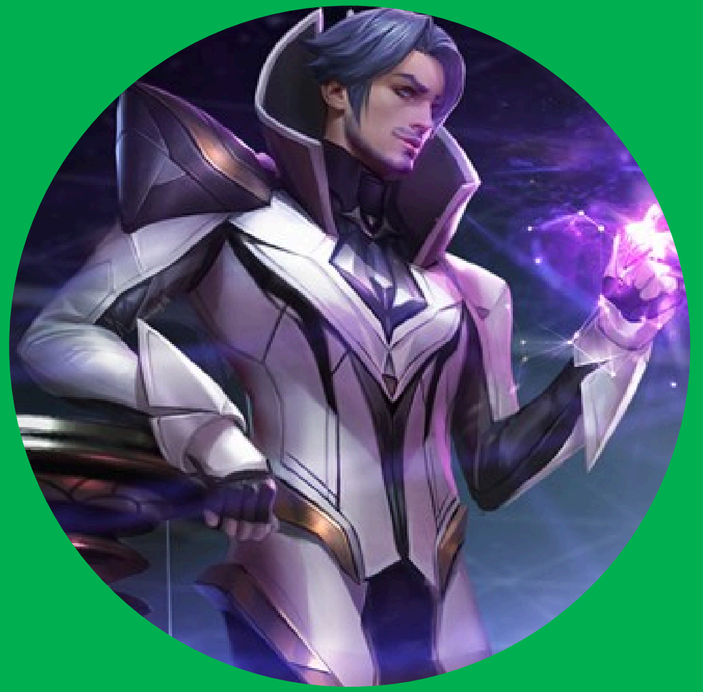
2025/11/28



Meet The Group

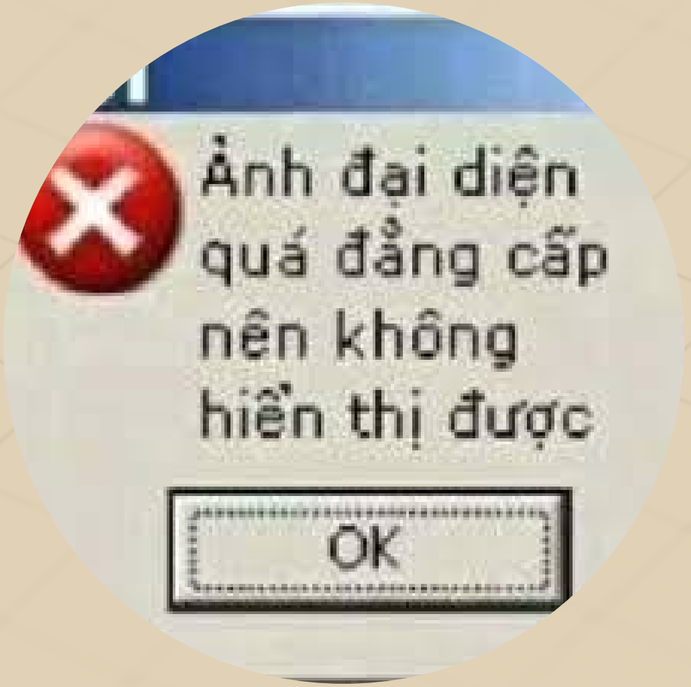


Le Quang Trung



Nguyen Hong Phuc

Meet The Group



Le Quang Trung

Nguyen Hong Phuc

CONTENTS

PART -

(01)

Problem 1: Time-Travel Query

P1: Snapshot at Start

Problem Restatement 1

Given N transactions and M tuples, each tuple exists between the commit time of its creation and deletion. For each transaction, list all tuples alive exactly at its start time.

Output Requirements 2

For each query, output the count and sorted ids of alive tuples. It is guaranteed the total output across all queries is not exceed 5e6.

Constraints 3

The constraints include N, M ≤ 969696, and all timestamps are distinct integers up to 1e12.

(++++++++++++)

How to check if a tuple j is visible to a Transaction T?

Commited Before Transaction T

commit[create_tx[j]] < start[T]

1

Removed After Transaction T

Commit[remove_tx[j]] > start[T]

2

Subtask 1: Small N,M≤9696

Coordinate Compression

Compress all timestamps to reduce the range of values, making the problem more manageable for small N and M.

Brute Force Approach

For each query time s, scan through every tuple and test the condition for visibility.

Complexity

The complexity of this approach is O(M) per query, resulting in a total complexity of O(N M), which is feasible for small N and M.

Result Output

Output the sorted list of tuple ids for each query. This brute force method is acceptable due to the small size constraints.

(++++++++++++)

Which *implementation* is true?

```python
def can_see(T, j, start, commit, create_tx, delete_tx):
    t = start[T]
    c = create_tx[j]
    if commit[c] > t:
        return False
    d = delete_tx[j]
    if d != -1 and commit[d] <= t:
        return False
    return True
```

```python
def can_see(T, j, start, commit, create_tx, delete_tx):
    t = start[T]
    c = create_tx[j]
    if commit[c] > t:
        return False
    d = delete_tx[j]
    if d != -1 and commit[d] > t:
        return False
    return True
```

Subtask 2: Interval Stabbing

Interval Modeling

Model each tuple as an interval [create_commit, delete_commit). For each query time, return the intervals that contain it using a sweep line or offline sorting approach.

Algorithm Recap

Build Timeline Events

Convert all create/commit/delete/query actions into timestamped events and sort them chronologically.

Sweep Through Events

Traverse events in time order, inserting or removing tuples from the active set as they become visible or invisible.

Snapshot for Each Query

Whenever we encounter a query event, take a snapshot of the current active tuples and record the result.

Algorithm Recap

```
events = []

# tuple birth
for j in range(1, M + 1):
    c = create_tx[j]
    if commit[c] != -1:
        events.append((commit[c], 0, j))

# tuple death
for j in range(1, M + 1):
    d = delete_tx[j]
    if d != -1 and commit[d] != -1:
        events.append((commit[d], 2, j))

# queries
for T in range(1, N + 1):
    events.append((start[T], 1, T))

# ensure: birth → query → death
events.sort()
```

```
# ensure: birth → query → death
events.sort()

active = []              # sorted list of alive
answers = [[] for _ in range(N + 1)]

for time, typ, idx in events:
    if typ == 0:
        # tuple j becomes visible
        j = idx
        bisect.insort(active, j)

    elif typ == 2:
        # tuple j becomes invisible
        j = idx
        pos = bisect.bisect_left(active, j)
        if pos < len(active) and active[pos]
            active.pop(pos)

    else:
        # query: snapshot of current DB state
        T = idx
        answers[T] = active.copy()
```

PART -

(02)

Operating System Memory Partitioning

P2: Minimize Max Quadrant

1

Problem Restatement

Given a grid of size H×W with N occupied frames, draw one horizontal and one vertical cut to split the plane into four quadrants. The goal is to minimize the maximum number of occupied frames in any quadrant.

2

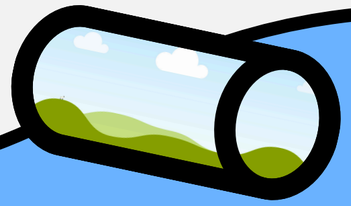
Output Requirements

The output is the minimum possible value of the maximum number of occupied frames in any quadrant after making the cuts.

3

Constraints

The constraints include H, W ≤ 1e12 and N ≤ 969696, making brute force approaches infeasible for large N.



Subtask 1: N <= 9696

Coordinate Compression

Compress coordinates to reduce the problem size. This allows us to efficiently compute quadrant counts.

Prefix Sums

Use 2D prefix sums to compute the number of occupied frames in each quadrant for every possible cut.

Complexity

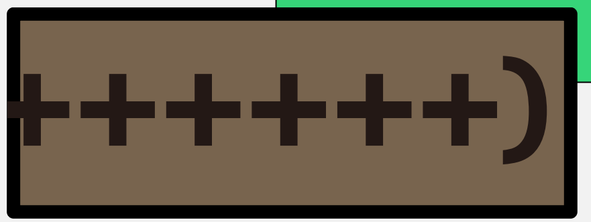
The complexity of this approach is $O(N^2)$.

Result Extraction

Iterate over all possible cuts, compute the maximum quadrant count for each, and keep track of the minimum value found.

Algorithm Recap

```
for i in range(sizeX + 1):
    for j in range(sizeY + 1):
        ans = min(ans, f(i, j))
```

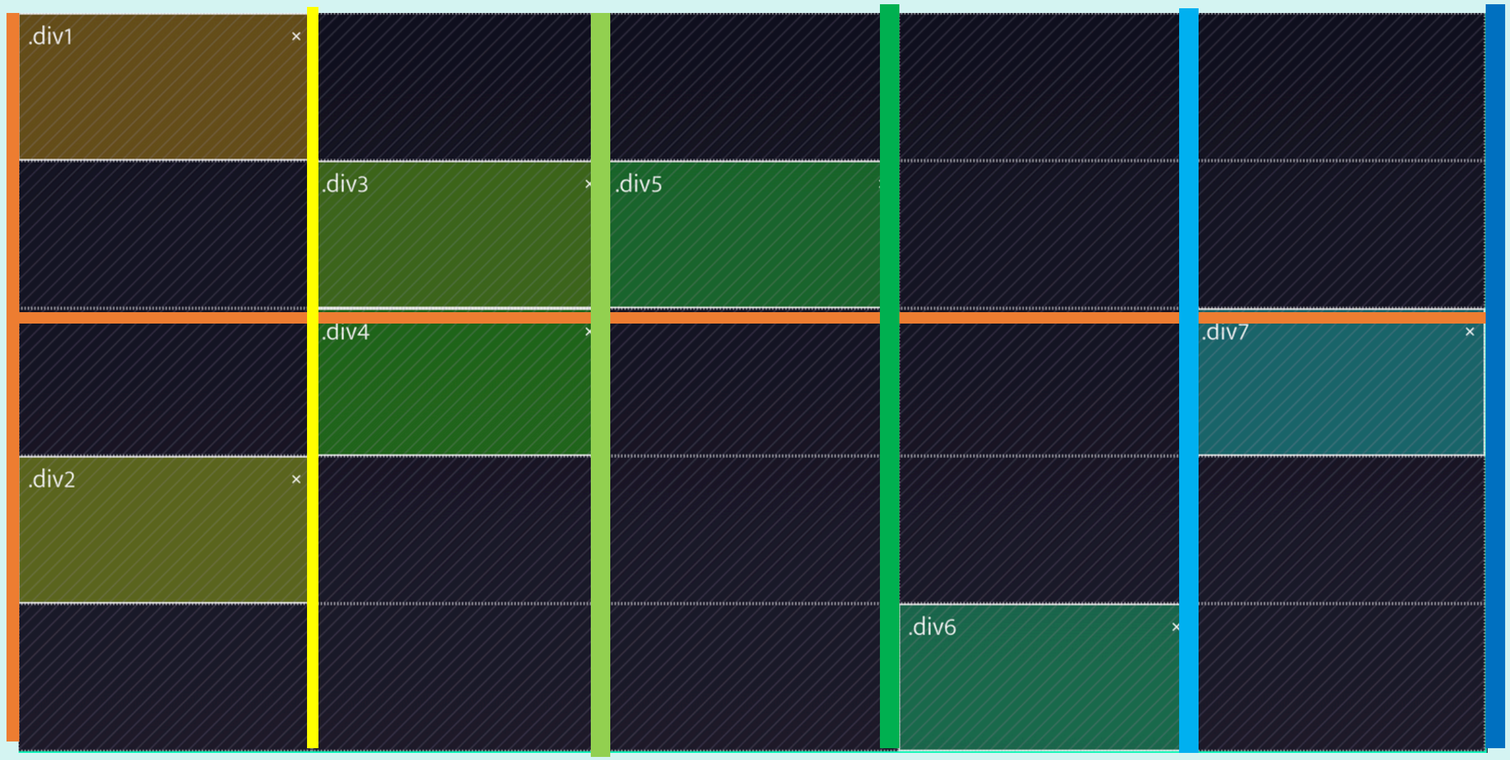


Subtask 2: N <= 363636

Cost Function

Special things about the structure of the cost function f(x, y) = max(quadrant counts) of 4 regions induced by row x and column y?

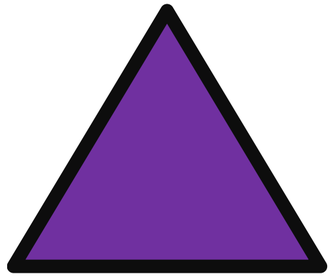
Cost Function Analysis



Algorithm Recap

Sweep Line

Sweep through rows, maintaining two segment trees, one for the upper, and one for the lower part.



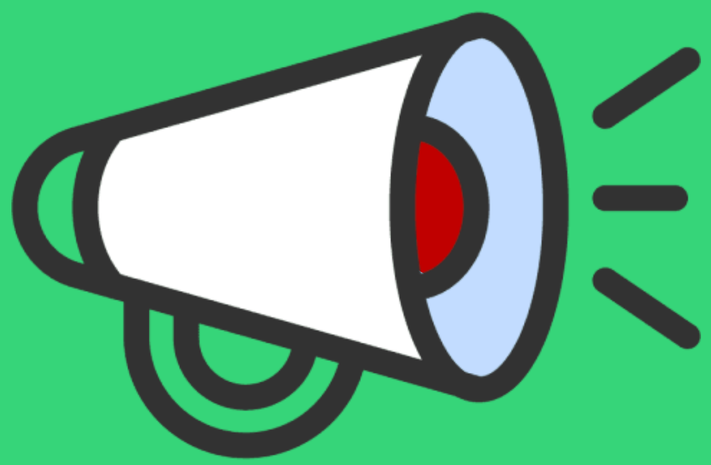
Binary Search

For each row, use binary search to find the optimal point and update answer.

What is the Time and Space Complexity?

Algorithm Recap

```
# points is a list of (i, j)
for i, j in points:
    lower.add(j, 1)

ans = n

for i in range(sizeX + 1):
    for j in rows[i]:
        upper.add(j, 1)
        lower.add(j, -1)

    l = 0
    r = sizeY
    pos = 0

    while l <= r:
        mb = (l + r) // 2
        q = get(i, mb)   # q is a list/tuple of 4 values

        if max(q[0], q[1]) <= max(q[2], q[3]):
            pos = mb
            l = mb + 1
        else:
            r = mb - 1

    ans = min(ans, f(i, pos))
    ans = min(ans, f(i, pos + 1))

print(ans)
```



Subtask 3: N <= 969696

What is Walk On Segment Tree?

This is a technique for navigating a segment tree.
→ Its primary goal is to find the first / last index in the array that satisfies a condition, using the information stored in each node.

Core Idea

Suppose we want the first index i such that: f(a[i]) = true
Algorithm :

- Start at the root.
- At each node:
- If the left child already contains a valid answer → go left.
- Otherwise → go right (and update any required residual values).
- When you reach a leaf, that index is the answer.

Complexity

O(logN) per query

Algorithm Recap

```
id = 1, l = 1, r = sizeY, pos = 0
countLeftUpper = 0, countLeftLower = 0, countRightUpper = 0, countRightLower = 0

while l <= r:
    mb = (l + r) // 2

    r1 = countLeftUpper + (1 if l == r else upper.tree[id * 2])
    r2 = countLeftLower + (1 if l == r else lower.tree[id * 2])
    r3 = countRightUpper + (0 if l == r else upper.tree[id * 2 + 1])
    r4 = countRightLower + (0 if l == r else lower.tree[id * 2 + 1])

    if max(r1, r2) <= max(r3, r4):
        pos = mb
        if l == r:
            break

        countLeftUpper += upper.tree[id * 2]
        countLeftLower += lower.tree[id * 2]
        id = id * 2 + 1
        l = mb + 1
    else:
        if l == r:
            break

        countRightUpper += upper.tree[id * 2 + 1]
        countRightLower += lower.tree[id * 2 + 1]
        id = id * 2
        r = mb
```

Final Complexity

Time Complexity

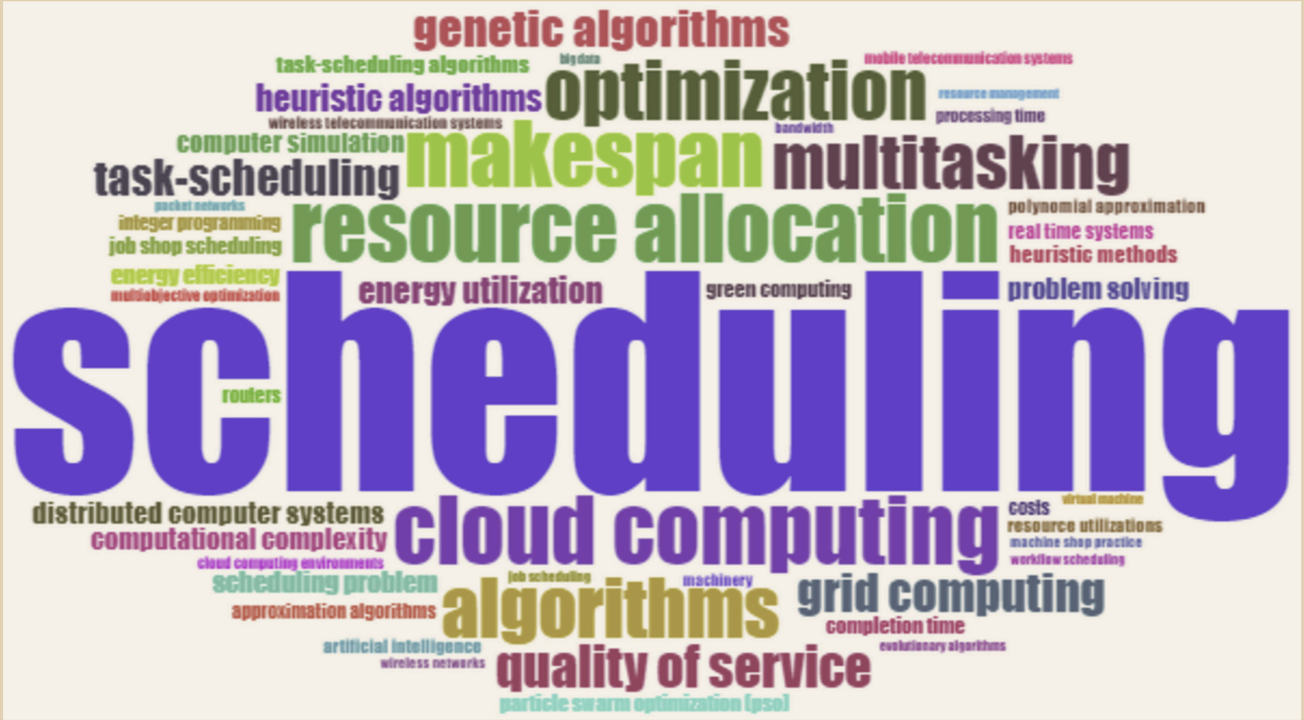
The overall time complexity is O(N log N) for coordinate compression and Segment Tree operations (update, query + walk).

Memory Usage

The space complexity is O(N) for points, row vector, and two Segment Tree.

Fucs and his server

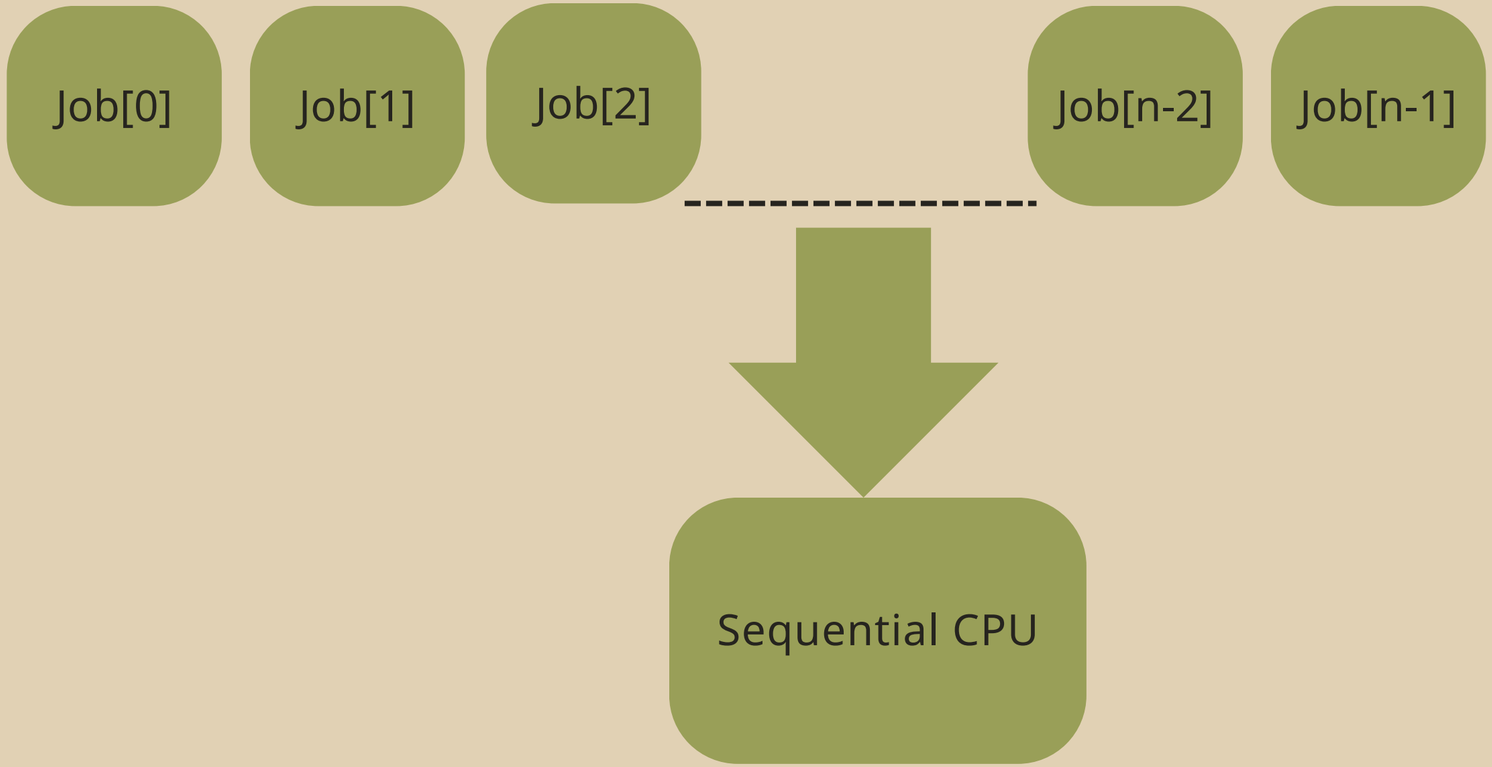
Scheduling Optimization



Presenter: Fucs

MAXIMIZE PROFIT

Attribute of Job[i]
r[i] – runtime: how long it takes
t[i] – timeout: the latest time to finish; if exceeded, the task fails
p[i] – profit: the reward if finished on time
ATOMIC !!!



SAMPLE CASE

Input:

Job	Runtime	Time out	Profit
1	5	8	200
2	2	4	100
3	1	3	50
4	2	3	10

Output:

Profit: 350

Order: 3 2 1

Subtasks

	Require	Input	Output
Subtask1	Find exact solution	**N <= 20 and r[i], t[i], p[i] <= 1e5**	Total Profit and Schedule
Subtask2	Find approximate solution	**N <= 500 and r[i], t[i], p[i] <= 1e12**	

PROBLEM REDUCTION
Transform and Conquer

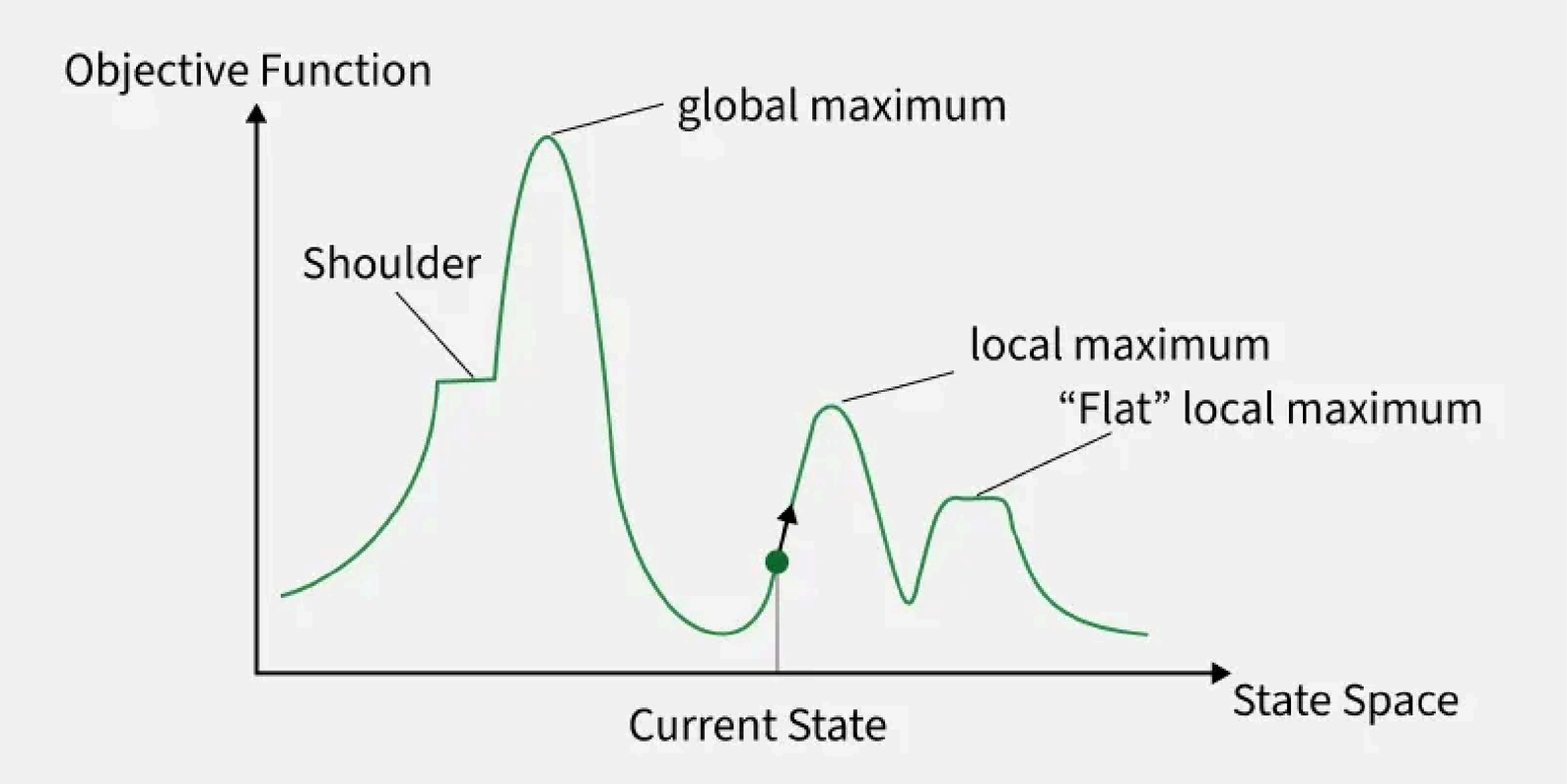
Permutation Problem -> Subset Selection

Subtask 1: N<= 20

Approach	**Dynamic Programming**	**Backtracking**
Idea	Similar to the knapsack problem. Sort tasks by increasing t[i]. **Formula:** $dp[i][j]=max(dp[i][j],dp[i-1][j-r[i]]+p[i])$	Try choosing or not choosing each task (0/1). Sort tasks by increasing t[i].
Complexity	Time: O(N * max(T)) Space: O(N*max(T))	Time: $O(2^N * N)$ Space: O(N)

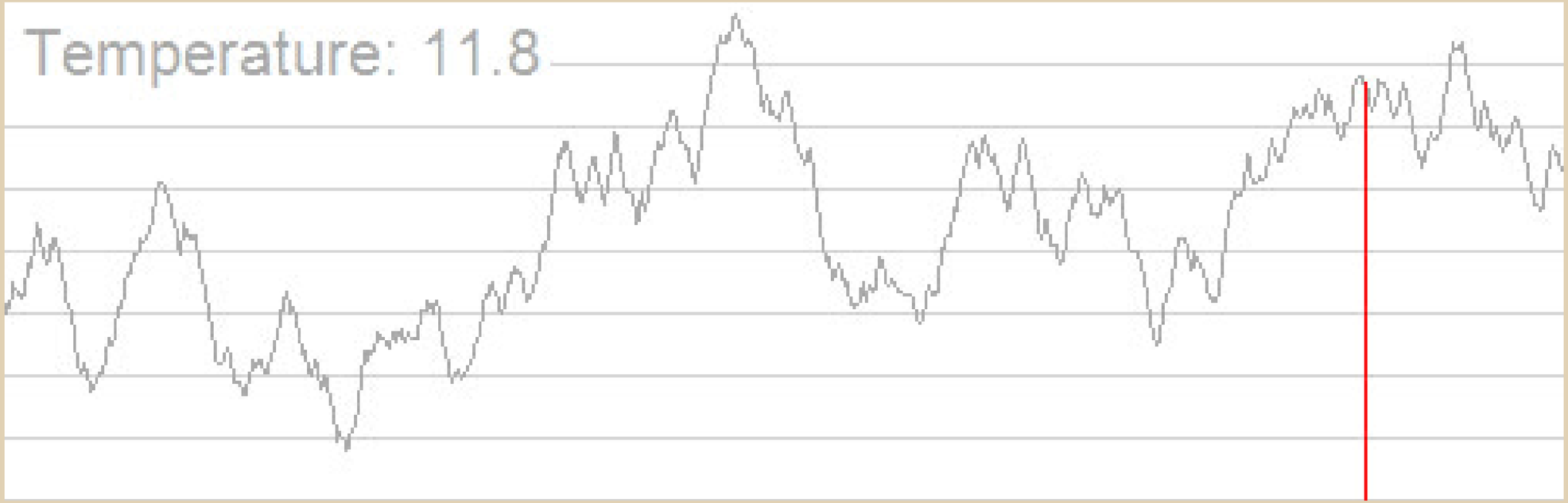
Local Search

Simulated Annealing



Local Search

Simulated Annealing



The Foundation

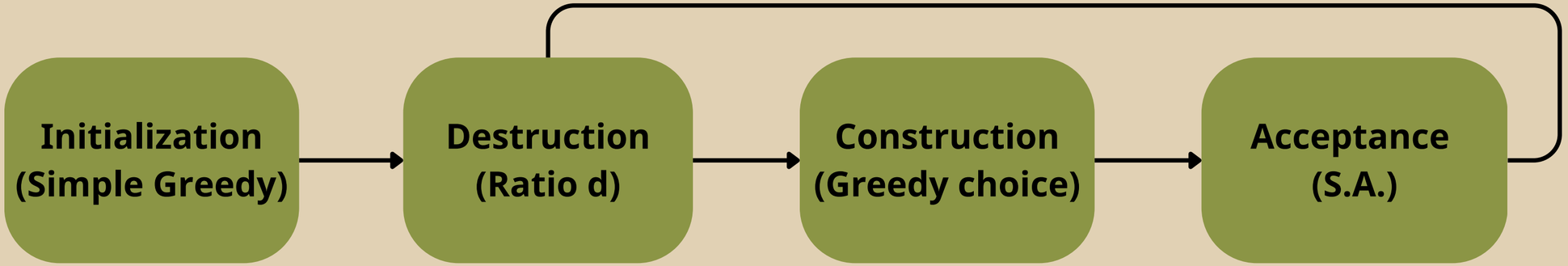
A Simple and Effective Iterated Greedy Algorithm for the Permutation Flowshop Scheduling Problem (2007)

Ruben Ruiz & Thomas Stützle

"Iterated Greedy yields better results than complex algorithms like Genetic Algorithms while being much simpler to implement."

Subtask 2: N<= 500

Iterated Greedy



Suitability

Selection & Replacement

DESTRUCTIBLE STRUCTURE

NEED FOR DEEP SEARCH

Subtask 2: N<= 500

Genetic Algorithm

Generations

Population size

Bitstring Encoding

Fitness Function

Crossover

Mutation

Tournament Selection

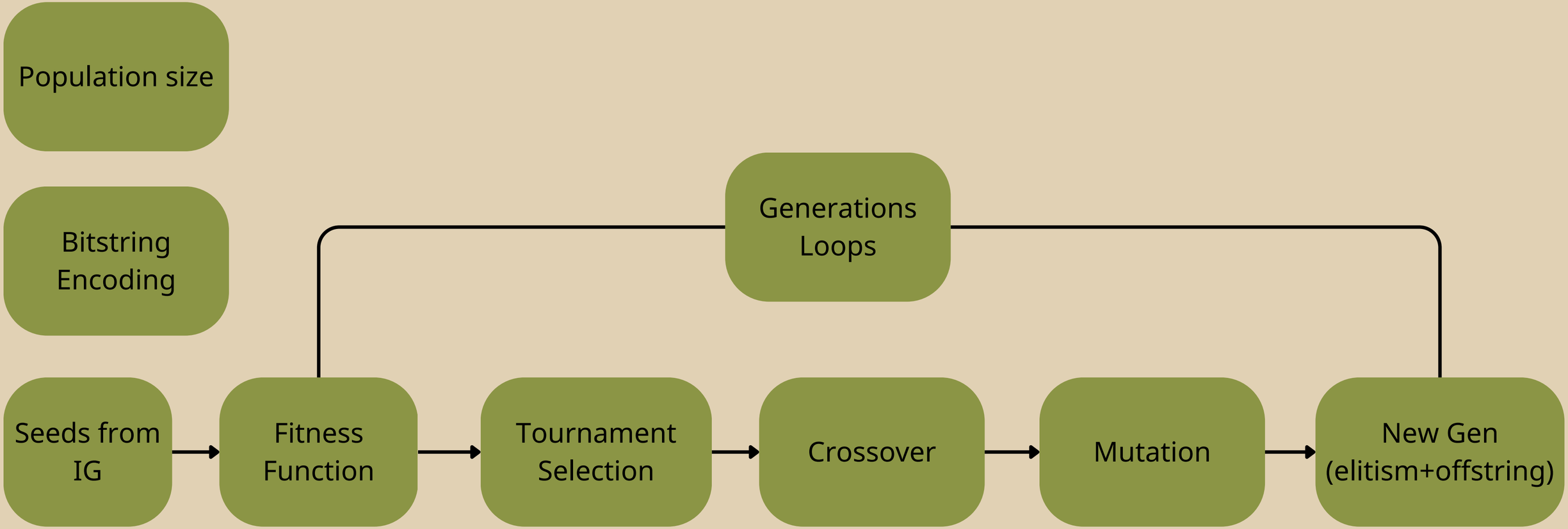
Elitism

Seeds from IG

Subtask 2: N<= 500

Genetic Algorithm

Key components



Prerequisites for Genetic Algorithm?

Conditions to apply Genetic Algorithm

GENETIC REPRESENTATION

FITNESS FUNCTION

EVOLUTIONARY OPERATORS

COMPLEXITY ANALYSIS

Time & Space Constraints

Algorithm	Time Complexity	Space Complexity
Iterated Greedy	$O(M * N^2)$	O(N)
Genetic Algorithm	O(G*P*N)	O(P*N)

EXPERIMENTAL RESULTS

Algorithm	Parameters	% of Best Profit
Iterated Greedy - The best	Time Limited = 10s	Ground Truth
Iterated Greedy 1	**D = 0.05**	**99.98%**
Iterated Greedy 2	D = 0.1	99.82%
Genetic Algorithm 1	P = 70	99.92%
Genetic Algorithm 2	P = 50	99.85%
DP&Backtracking	Exact Solution	TLE

FINAL VERDICT

Iterated Greedy

Best Overall Performance

Genetic Algorithms

Diversity Champion

Dynamic Programming & Backtracking

Exact but Limited

"While Dynamic Programming & Backtracking offers exact solutions for small tasks, it fails at scale. Genetic Algorithms ensure diversity, but Iterated Greedy takes the crown for its balance of speed and optimization on large datasets. "

Thank you

Do you have any questions for us?