A Query-Efficient Algorithm for Optimal Trade Execution under Cardinality Constraints via Lagrangian Relaxation

Benedict Presley - 13523067
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jalan Ganesha 10 Bandung
E-mail: presleybenedict04@gmail.com , 13523067@std.stei.itb.ac.id

***Abstract*—Optimal trade execution under cardinality constraints, which limit the number of trades within a given period, presents a complex combinatorial challenge. This paper introduces a query-efficient algorithm to find the maximum achievable profit from exactly k non-intersecting trades within a specified time window. The method reframes the problem using Lagrangian Relaxation, which transforms the hard constraint on the number of trades into a penalized, unconstrained problem by adding a cost (λ) for each trade. To solve this relaxed problem efficiently across many queries, the approach employs a Segment Tree data structure combined with Dynamic Programming.**

***Keywords*— Optimal Trade Execution, Cardinality Constraints, Lagrangian Relaxation, Segment Tree, Dynamic Programming**

I. Introduction

Optimal trade execution is about smartly buying or selling large amounts of assets in financial markets. The goal is to do this in a way that costs the least money and causes the least disruption to the market. In today's fast-paced, computer-driven, high-frequency trading world, being able to trade well and quickly has a big impact on how much profit a company makes and how much risk it faces. A major challenge comes from practical rules that limit how trades can be made. These are called cardinality constraints. They limit the number of separate trades, orders, or specific times you're allowed to trade within a certain period. Such limits are common because of regulatory rules, operational difficulties, or simply to avoid being noticed by other traders. These constraints transform the optimization problem into a more complex combinatorial problem.

Common ways to handle optimal trade execution often use methods like heuristics. While these methods work for some problems, they can become less optimal or challenging when dealing with huge amounts of data, long trading periods, or when cardinality constraints are directly included. The sheer number of possible choices for picking the best trading opportunities makes it hard to quickly get answers to new questions about the best way to trade. For example, if we need to repeatedly figure out the best strategy for different periods or different numbers of allowed trades (like asking, "what's the largest sum of sums for k non-overlapping parts of a segment?"), we need a really fast method.

This paper focuses on making optimal trade execution under cardinality constraints much faster when you need to ask many questions. We suggest a new method that uses Lagrangian Relaxation. This technique helps turn a difficult problem with constraints into a simpler one. Lagrangian relaxation lets us break down the big, complex problem into smaller parts that are easier to solve. By combining this relaxation method with a smart way to answer queries quickly, our approach greatly reduces the time it takes to get answers for different trading scenarios and limits, such as different trading periods ($[l \dots r]$) and the maximum number of allowed trades (k).

II. Theoretical Basis

A. Divide and Conquer

Divide and Conquer is a problem-solving paradigm that involves breaking down a large, complex problem into smaller, more manageable pieces. The core idea is to solve these smaller pieces independently and then combine their solutions to get the solution for the original big problem. Think of it like organizing a very large project: instead of one person trying to do everything, you break it into smaller tasks, assign them to different teams, and then bring all the completed parts together at the end.

The Divide and Conquer strategy generally follows three steps:

1. Divide: The original problem is split into two or more smaller sub-problems. These sub-problems are usually similar in type to the original problem but are simpler to solve because they are smaller.
2. Conquer: Each of these smaller sub-problems is then solved. If a sub-problem is still too big, this step is repeated recursively (the Divide and Conquer process is applied again to the sub-problem). If a sub-problem is small enough, it is solved directly and simply.
3. Combine: The solutions obtained from the individual sub-problems are then put together to form the final solution to the original, larger problem. This combining step is crucial, as it ensures that the work done on the smaller pieces correctly contributes to the overall answer.

The main advantage of Divide and Conquer is that it often leads to efficient algorithms and can make very complicated problems easier to understand and tackle.

B. *Dynamic Programming*

Dynamic Programming (DP) is problem solving paradigm. This paradigm is used in problem that can be broken down into simpler sub-problems. While it shares some similarities with Divide and Conquer, its main strength lies in handling overlapping sub-problems. This means that when we break down a big problem, we might find ourselves trying to solve the exact same smaller problem multiple times. DP avoids this wasted effort.

There are two core principles in Dynamic Programming. A problem must satisfy both principles for Dynamic Programming to apply.

1. Optimal Substructure: This means that the best solution to the overall problem can be found by combining the best solutions to its smaller sub-problems. For example, if we want to find the shortest path from A to C, and it goes through B, then the path from A to B must also be the shortest path to B.
2. Overlapping Sub-problems: This is the key distinguishing feature. In many problems, if we try to solve them using a simple recursive (repeated) approach, we end up re-calculating the answers to the same smaller problems again and again. Dynamic Programming tackles this by solving each unique sub-problem only once and then storing its answer. This stored answer can then be looked up and reused whenever that specific sub-problem comes up again.

There are two common ways to implement Dynamic Programming:

1. Memoization (Top-down): This is a recursive approach. We start trying to solve the big problem. When we need the solution to a sub-problem, we first check if we've already solved it and stored its answer. If yes, we use the stored answer. If no, we solve it, store the result, and then use it.
2. Tabulation (Bottom-up): This is an iterative approach. Instead of starting from the top (the big problem), we start by solving the smallest possible sub-problems first. We then build up solutions for progressively larger sub-problems by using the already computed answers of the smaller ones, typically filling out a table or array.

By avoiding repeated calculations, Dynamic Programming can significantly improve the speed of algorithms, often turning problems that would otherwise take an impossibly long time (exponential complexity) into problems that can be solved in a reasonable amount of time (polynomial complexity).

C. *Binary Search*

Binary Search is a highly efficient algorithm used to locate a specific item within a sorted list or array. Its efficiency significantly surpasses that of sequential item-by-item checking, particularly for very long lists. We can visualize its operation by considering the process of finding a specific word in a thick dictionary: rather than commencing from the first page and progressively flipping through each, one would typically open to an approximate middle, ascertain whether the target word precedes or follows that point, and consequently narrow the search to one half of the remaining dictionary. Binary Search operates on this fundamental principle of progressive elimination.

The general procedure is as follows:

1. Find the middle
   We begin by examining the element positioned at the midpoint of our sorted list.
2. Comparison
   The midpoint element is then compared against the specific item we are seeking, referred to as our "target."
3. Search Space Reduction (Implicit Divide and Conquer)
   If the midpoint element precisely match our target, the search concludes successfully. If our target is smaller than the midpoint element, we deduce that it must reside exclusively within the first half of the list (owing to the list's sorted property). Consequently, the latter half of the list can be disregarded. Conversely, if our target is larger than the midpoint element, it must be located within the second half of the list. The first half is then excluded from further consideration.
4. Iteration
   Steps 1-3 are iteratively applied to the remaining half of the list. This continuous halving of the search space persists until either the target is identified or the search space is entirely depleted, indicating the item's absence from the list.

A critical prerequisite for the correct functioning of Binary Search is that the list must be sorted. If the list is not sorted, Binary Search will not yield accurate results. Due to its consistent division of the search space by half, Binary Search demonstrates exceptional speed, even when dealing with large datasets. Its time complexity is characterized as logarithmic $O(\log N)$, signifying that it requires a relatively small number of operations for substantial amounts of data.

Binary search can also be conceptualized in terms of functions. If we have a sorted list, it can be viewed as representing a monotonic function $f(x)$, where x denotes the index and $f(x)$ corresponds to the value at that index. A monotonic function is defined as one that consistently either increases, decreases, or remains constant. When we perform a binary search for a "target" value in a sorted list, we are essentially endeavoring to determine an index x such that $f(x)$ is equals to target. Even if the precise target value is not present, Binary Search retains the capability to pinpoint the interval where the function crosses a specific threshold or where the target value would logically reside. This functional interpretation proves particularly valuable when employing Binary Search to ascertain optimal values in problems where an objective function or a constraint function exhibits monotonicity.

D. *Segment Tree*

A Segment Tree is a special data structure that helps us efficiently store and query information about ranges or "segments" within an array. It's particularly useful when we need to quickly ask questions like "What is the sum of numbers from index X to index Y?" or "What is the largest value in the range from X to Y?" and also quickly update individual values in the array.

The way a Segment Tree is built and works is a classic example of the Divide and Conquer principle in action:

- Tree Structure
  A Segment Tree is typically a binary tree. Each "node" (or box) in this tree represents a specific range (or segment) of the original array.
- Building the Tree (Divide)
  The "root" node at the very top of the tree represents the entire array. This root node then divides its range into two halves, and each half becomes the range for its two "child" nodes. This division process continues recursively until we reach the "leaf" nodes at the bottom of the tree. Each leaf node represents a single element from the original array.
- Storing Information (Conquer/Combine)
  Each internal node (a node that has children) stores some aggregated information about the range it represents. This information is typically combined from the information stored in its children. For example, if a node represents the sum of a range, its value would be the sum of the sums from its left and right child nodes. This is the "combine" part of Divide and Conquer.

Once built, a Segment Tree allows for two main types of operations:

- Querying a Range: To find information about a specific range (e.g., sum from l to r), we initiate a query starting from the root node of the Segment Tree. Each node in the tree covers a certain range of the original array. During the query process, we compare the range covered by the current node with the range we are interested in (the query range, say $[Q_L, Q_R]$). There are three possible scenarios for each node we visit:
  - No Overlap: If the node's range falls completely outside the query range (e.g., the node covers indices before Q_L or after Q_R), this node and its children are irrelevant to our query. We simply stop traversing this branch and return an identity value.
  - Complete Overlap: If the node's range is entirely contained within our query range (its start and end indices are both within $[Q_L, Q_R]$), then the aggregated information already stored in this node is exactly what we need for this part of the query. We return the value stored in this node directly, without needing to look at its children.
  - Partial Overlap: If the node's range partially overlaps with our query range (meaning some part of its range is inside, and some part is outside, or it spans across the query range), we cannot use the node's stored value directly. In this case, we recursively call the query function on both its left child and its right child. Once we get the results from both children, we combine them according to the type of query. This recursive process efficiently collects all relevant parts from the tree.

  This recursive process ensures that we only visit the necessary nodes to cover the desired query range, leading to very fast query times ($O(\log N)$ time).
- Updating an Element: If we change a value in the original array, the Segment Tree can be updated quickly. We simply find the leaf node corresponding to the changed element and then update the values of all its parent nodes up to the root ($O(\log N)$ time).

Furthermore, the Segment Tree can be extended to support a wide variety of aggregate queries beyond simple sums or maximums, such as range minimum, range XOR, or even more complex operations like matrix multiplication over ranges. Advanced variations also exist, such as Persistent Segment Trees, which allow us to query past versions of the array efficiently, or Lazy Propagation, which optimizes updates over large ranges.

E. *Convex and Concave Functions*

In mathematics, a function is considered convex if, for any two points on its graph, the line segment connecting these two points lies entirely on or above the graph of the function. Imagine drawing a curve: if we pick any two points on that curve and draw a straight line between them, and that line never goes below the curve, then the curve represents a convex function.

More formally, a function $f(x)$ is convex if for any two points x_1 and x_2 in its domain, and for any value α where $0 \leq \alpha \leq 1$, the following condition holds

$$f(\alpha x_1 + (1-\alpha)x_2) \leq \alpha f(x_1) + (1-\alpha) f(x_2)$$

This inequality means that the function's value at any point along the line segment between x_1 and x_2 is less than or equal to the value of the line connecting $f(x_1)$ and $f(x_2)$.

The opposite of a convex function is a concave function. A function $g(x)$ is concave if the line segment connecting any two points on its graph lies entirely on or below the graph. This means for any two points x_1 and x_2 in its domain, and for any value α where $0 \leq \alpha \leq 1$, the following holds

$$g(\alpha x_1 + (1-\alpha)x_2) \geq \alpha g(x_1) + (1-\alpha)g(x_2)$$

Maximizing a concave function is mathematically equivalent to minimizing its negative (which would be a convex function).

Key properties of both convex and concave functions that make them important in optimization include:

- Unique Minimum/Maximum
  For a convex function, any local minimum is also a global minimum. Similarly, for a concave function, any local maximum is also a global maximum. This is a very powerful property, as it means we don't have to worry about getting stuck in "bad" local solutions when trying to find the absolute best (global) solution.
- Easy to Optimize
  Algorithms for minimizing convex functions or maximizing concave functions are generally much more efficient and reliable than those for non-convex or non-concave functions. Many optimization techniques, including those related to subgradient methods used in Lagrangian Relaxation, rely on these well-behaved properties.

F. *Lagrangian Relaxation*

Lagrangian Relaxation is an advanced mathematical technique used to solve very difficult optimization problems, especially those that include "hard" constraints. A hard constraint is a rule that must be followed, and if it's violated, the solution is not valid. Often, these hard constraints make a problem incredibly complex to solve directly.

The fundamental idea behind Lagrangian Relaxation is as follows:

- Transforming the Problem
  Instead of strictly enforcing a hard constraint, we "relax" it. This means we move that constraint out of the strict rules section and instead incorporate it into the main objective function (the formula we are trying to maximize or minimize).
- The Penalty Factor (Lagrangian Multiplier)
  When a constraint is relaxed and moved into the objective function, it's typically multiplied by a non-negative value called a Lagrangian multiplier (often denoted by the Greek letter λ). This multiplier acts as a "penalty coefficient." If the solution violates the relaxed constraint, the penalty term in the objective function becomes large, making the solution less desirable. Conversely, if the solution satisfies the constraint well, the penalty is small or zero.
- Creating an Easier Subproblem
  By doing this, the original, difficult problem is transformed into a "Lagrangian subproblem." This new subproblem is usually much easier to solve because it has fewer or simpler constraints. For instance, a problem that was hard due to a single complex constraint might become a set of independent, easily solvable parts once that constraint is relaxed.
- Providing a Bound
  The optimal solution found for the easier Lagrangian subproblem provides a valuable "bound" for the original, difficult problem. For minimization problems, the Lagrangian subproblem's optimal value will always be less than or equal to the original problem's optimal value (a lower bound). For maximization problems, it will be greater than or equal to (an upper bound).
- Dual Problem
  A significant advantage in Lagrangian Relaxation is that the Lagrangian dual problem (which involves finding the best λ) has favorable mathematical properties. The dual problem is always concave for maximization or convex for minimization, regardless of whether the original problem has these properties. This makes the dual problem much easier to solve efficiently using standard optimization methods.
- Iterative Adjustment
  To find the best possible bound (the "tightest" one), the Lagrangian multipliers (λ values) are often adjusted iteratively. Methods like subgradient optimization are used to systematically change λ values, guiding the solution towards one that best satisfies the original, relaxed constraint. In cases where the dual function has specific properties (such as convexity or concavity), Binary Search can also be employed to efficiently find the optimal multiplier values, significantly reducing the computational effort required for convergence.

Lagrangian Relaxation is highly effective for problems where direct methods are too slow. It offers a way to find good approximate solutions and provides a mathematical guarantee (the bound) on how far off these approximations might be from the true optimal answer.

III. Problem Definition

A. *Problem Definition*

In the high-frequency trading (HFT) domain, a crucial task is to strategically execute large trades or manage a series of trading opportunities over specific time periods. We can imagine the market's behavior or our trading system's potential at discrete time intervals as a sequence of numbers. Each number a_i in our array A represents the expected profit or loss if we initiate a trade or observe a market signal at that precise moment. A "segment" of this array, say from time l to time r, signifies a particular window or horizon during which we are interested in making trades.

A key challenge in HFT is that we often face limitations on how many separate trading actions we can take within a given window. These "cardinality constraints" are important for several reasons: they help manage transaction costs, comply with regulatory rules on the number of messages sent to exchanges, reduce our overall impact on the market, and avoid revealing our strategy to other traders. Therefore, for any given trading window (from time l to r), we are not simply looking for the most profitable trades, but specifically the most profitable set of exactly k distinct, non-overlapping trading actions. Each trading action might span a short period (a subsegment of time) and its total value is the sum of all individual profit/loss values within that subsegment. Our goal is to find these k actions such that their combined profit is maximized.

Formally, we define the problem as follows,

We are given an array $A = [a_1, a_2, \dots, a_N]$ of N real numbers. A segment of the array, denoted as $[l \dots r]$, refers to the contiguous subarray $a_l, a_{l+1}, \dots, a_r$, where $1 \leq l \leq r \leq n$.

We are presented with a series of q independent queries, Each query is defined by an ordered triple (l, r, k), where:

- $1 \leq l \leq r \leq n$: This defines the start and end indices of the specific segment of interest within the array A.
- $1 \leq k \leq r - l + 1$: This specifies the exact number of non-empty, non-intersecting subsegments that must be selected from the segment $[l \dots r]$.

For each query (l, r, k), our objective is to determine the largest possible sum achievable by selecting precisely k non-empty, non-intersecting subsegments from the array segment $[l \dots r]$. The sum we aim to maximize is the sum of the sums of elements within these k chosen subsegments.

Mathematically, for a given query (l, r, k), we seek to maximize:

$$\sum_{j=1}^{k} \sum_{i=s_j}^{e_j} a_i$$

subject to the following conditions:

- Each selected subsegment $[s_j \dots e_j]$ must be non-empty, i.e., $s_j \leq e_j$.
- All selected subsegments must be contained within the query segment $[l \dots r]$, i.e., $l \leq s_j \leq e_j \leq r$ for all $j = 1, \dots, k$.
- The selected subsegments must be non-intersecting. This implies that for any two distinct subsegments $[s_j \dots e_j]$ and $[s_p \dots e_p]$ with $j \neq p$, their intervals do not overlap. Formally, either $e_j < s_p$ or $e_p < s_j$.

  Without loss of generality, if we order the subsegments by their starting indices ($s_1 \leq s_2 \leq \dots \leq s_k$), then this condition simplifies to $e_j < s_{j+1}$ for all $j = 1, \dots, k-1$.

IV. Solution

To address the problem of finding the optimal sum of k non-intersecting subsegments within a given range $[l \dots r]$, we will use an algorithm that combines Lagrangian Relaxation, Dynamic Programming, Segment Tree, and Parametric Search.

A. *Lagrangian Relaxation Reformulation*

Let $F(k)$ denote the maximum achievable sum using exactly k non-empty, non-intersecting subsegments within a query range $[l \dots r]$. A fundamental property of this function is its concavity. That is, the marginal gain from selecting an additional subsegment is non-increasing: $F(k) - F(k-1) \geq F(k+1) - F(k)$. This is due to that fact that we will always choose segments with larger sum first before choosing those with smaller sum. This property is key to our approach.

Instead of directly solving for a specific k, we use Lagrangian relaxation to transform the constrainted optimization problem into an unconstrained one. We introduce a Lagrange multiplier, λ, which can be interpreted a penalty or cost for each subsegment selected. The objective function becomes:

$$G(\lambda) = \max_S \left(\sum_{s \in S} sum(s) - \lambda |S| \right)$$

where S is any set of non-empty, non-intersecting subsegments. This can be expressed in terms of $F(k)$ as

$$G(\lambda) = \max_{i \geq 0} F(i) - \lambda i$$

Maximizing $G(\lambda)$ is equivalent to finding the maximum-weight set of subsegments where each segment incurs a fixed cost λ. The function that maps the penalty λ to the optimal number of segments chosen for $G(\lambda)$, let's call it $c(\lambda)$, is monotonically non-increasing. A higher penalty λ will lead to selecting fewer segments. This monotonicity allows us to use binary search on the value of λ to find a penalty that encourage the selection of exactly k subsegments.

B. *Dynamic Programming on Segment Tree*

To compute $G(\lambda)$ for any given range $[l \dots r]$ efficiently, we employ dynamic programming (DP). To avoid re-computing for each query, the DP is pre-calculated and stored in a segment tree data structure built over the initial array A.

Each node in the segment tree represents a contiguous subsegment of A. For each node, we store DP states that encapsulate the optimal solution for its corresponding range. A critical aspect is handling the merging of subsegments across the boundaries of adjacent nodes. To facilitate this, our DP state must capture whether the endpoints of a node's range are part of an "open" subsegment.

For each node in the segment tree, we maintain a 2×2 matrix of DP information, let's call it $\mathcal{D}_{uv}$. The entry $\mathcal{D}_{uv}$, where $u, v \in \{0, 1\}$, stores the optimal solutions for the node's range under different boundary conditions:

- $u = 0$: The left boundary of the node's range is not covered by a selected subsegment.
- $u = 1$: The left boundary is covered by a subsegment that could potentially merge with a subsegment from an adjacent node to the left.
- $v = 0$: The right boundary is not covered.
- $v = 1$: The right boundary is covered.

Each entry $\mathcal{D}_{uv}$ is a vector representing the concave function $F(i)$. Specifically, it stores the maximum sum for each possible number of subsegments i under the state's boundary conditions.

C. *Segment Tree Merging*

The use of segment tree here is to combine, or merge, the results from two child nodes to computer the result for their parent. Suppose a parent node covers range $[i \dots j]$, and its left and right children cover $[i \dots m]$ and $[m + 1 \dots j]$ respectively. To computer the parent's DP state $\mathcal{D}_{uv}^{parent}$, we combine the children's states. For example:

$$\mathcal{D}_{uv}^{parent} = \max\left(\mathcal{D}_{u0}^{left} \oplus \mathcal{D}_{0v}^{right}, \mathcal{D}_{u1}^{left} \oplus \mathcal{D}_{1v}^{right}\right)$$

Where $\oplus$ denotes the merge operation for the DP vectors. The first term corresponds to the case where no subsegment crosses the midpoint boundary m, while the second corresponds to the case where a subsegment from the left child merges with one from the right child.

A naive merge operation (convolution) would be too slow. However, due to the concavity of the underlying functions, we can perform the merge much more efficiently. Instead of the vectors of values, we can operate on the vectors of their slopes (the differences between consecutive values). Merging two concave vectors is equivalent to merging their sorted slope arrays and reconstructing the result via a cumulative sum. This allows the merge operation for two nodes to be completed in time linear to the sum of their lengths. The entire segment tree can thus be built in $O(N \log N)$ time.

D. *Answering Queries*

For a given penalty λ, we can query segment tree over the range $[l \dots r]$ to find the value of $G(\lambda)$ and the corresponding optimal number of segments $c(\lambda)$. This query takes $O(\log^2 N)$ time.

We perform a binary search on a range of possible values for λ to find the critical penalty, λ_{crit}, that separates the choice of fewer than k segments from the choice of at least k segments. Specifically, we find the smallest integer penalty λ_{crit} such that $c(\lambda_{crit}) \geq k$. From the binary search, we know that for the penalty $\lambda_{crit} - 1$, the optimal number of segments is $\lambda_{prev} < k$.

At this point, we have two anchor points on our concave function F:

- For a penalty of $\lambda_{crit} - 1$, the optimal solution uses c_{prev} segments. The maximum sum is $F(c_{prev}) = G(\lambda_{crit} - 1) + (\lambda_{crit} - 1)c_{prev}$.
- For a penalty of λ_{crit}, the optimal solution uses c_{crit} segments. The maximum sum is $F(c_{crit}) = G(\lambda_{crit}) + \lambda_{crit} c_{crit}$.

Since $F(k)$ is concave and we are seeking $F(k)$ for $c_{prev} < k \leq c_{crit}$, we can exploit the fact that the function is linear between the points on the convex hull defined by these penalties. We can therefore determine the value of $F(k)$ precisely via linear interpolation:

$$F(k) = F(c_{prev}) + \frac{F(c_{crit}) - F(c_{prev})}{c_{crit} - c_{prev}}(k - c_{prev})$$

This combination of techniques provides a query-efficient algorithm. The preprocessing takes $O(N \log N)$, and each query is answered in $O(\log^2 N \log V)$, where V is the range of possible sums.

E. *Focus on Value-Only Optimization*

A notable characteristic of the proposed algorithm is that it is optimized solely for finding the maximum achievable sum, not for reconstructing the set of subsegments that produces this sum. This design choice is fundamental to the method's efficiency.

The primary reason for this lies in the information compression that occurs during the DP state merges within the segment tree. The merge operation on two DP vectors (representing the concave functions from child nodes) combines them by merging their sorted arrays of slopes. While this correctly computes the shape and values of the parent node's concave function, it discards the information about which specific combinations of subsegments from the child nodes gave rise to each optimal value. For any given point on the parent's function, there may have been numerous valid combinations from the children; the algorithm only propagates the maximum sum, not the path taken to achieve it.

Furthermore, the final interpolation step to calculate $F(k)$ is a mathematical abstraction. It leverages the global property of concavity to find a value that lies on a line between two points ($F(c_{prev})$ and $F(c_{crit})$) derived from different penalty values. The resulting value $F(k)$ does not directly correspond to any single set of segments computer during the parametric search.

To reconstruct the actual segments, one would need to store back-pointers or other path-related metadata at every stage of the DP calculation and merging process. This would significantly increase both the memory complexity (from $O(N \log N)$ to potentially $O(N^2)$) and the time complexity of the merge and query operations, thereby negating the high query efficiency that is the central goal of this algorithm.

V. Implementation

A. Constants and Structs

```
#define ll long long
#define ld long double

const ll INF = 1e18;

struct Query {
    int l, r, k;
};

struct Result {
    ld value;
    int count;

    bool operator<(const Result& other) const {
        if (value != other.value) {
            return value < other.value;
        }
        return count < other.count;
    }
};

struct DPState {
    std::vector<ll> f[2][2];
    int len = 0;

    void init_leaf(ll val) {
        len = 1;
        for (int i = 0; i < 2; ++i) {
            for (int j = 0; j < 2; ++j) {
                f[i][j] = std::vector<ll>(2, -INF);
            }
        }
        f[0][0][0] = 0;
        f[0][0][1] = val;
        f[0][1][1] = val;
        f[1][0][1] = val;
        f[1][1][1] = val;
    }
};
```

These three structures work together to manage the flow of information throughout the algorithm.

- A Query object represents the initial problem statement, defining the target segment [l, r] and the required number of subsegments k.
- The DPState is the most complex structure, serving as the memory of the segment tree nodes. It encapsulates the dynamic programming results for a specific range of the array, including all possible maximum sums for any number of subsegments, while also tracking boundary conditions necessary for merging with adjacent nodes.
- A Result object is a lightweight, temporary container used during the parametric search. For a given penalty lambda, it holds the two key pieces of information needed to guide the search: the optimal value of the relaxed objective function and the number of segments used to achieve it.

B. Solver

```
class TradeExecutionSolver {
private:
    int n, q;
    int tree_size;
    std::vector<ll> a;
    std::vector<Query> queries;
    std::vector<DPState> seg_tree;
    std::vector<ll> answers;

    std::vector<ll> merge_vectors(const std::vector<ll>& v1, const
std::vector<ll>& v2) {
        if (v1.empty() || v2.empty()) {
            return {};
        }

        std::vector<ll> slopes1, slopes2;
        for (size_t i = 1; i < v1.size(); ++i) slopes1.push_back(v1[i]
- v1[i-1]);
        for (size_t i = 1; i < v2.size(); ++i) slopes2.push_back(v2[i]
- v2[i-1]);

        std::vector<ll> merged_slopes;
        std::merge(slopes1.begin(), slopes1.end(), slopes2.begin(),
slopes2.end(),
                   std::back_inserter(merged_slopes),
std::greater<ll>());

        std::vector<ll> result;
        result.push_back(v1[0] + v2[0]);
        for (ll slope : merged_slopes) {
            result.push_back(result.back() + slope);
        }

        return result;
    }

    DPState merge_nodes(const DPState& left, const DPState& right) {
        if (left.len == 0) return right;
        if (right.len == 0) return left;

        DPState res;
        res.len = left.len + right.len;
        for (int i = 0; i < 2; ++i) {
            for (int j = 0; j < 2; ++j) {
                res.f[i][j] = std::vector<ll>(res.len + 1, -INF);
            }
        }

        for (int i = 0; i < 2; ++i) {
            for (int j = 0; j < 2; ++j) {
                auto    merged_0    =    merge_vectors(left.f[i][0],
right.f[0][j]);
                if (!merged_0.empty()) {
                    for (size_t l = 0; l < merged_0.size(); ++l) {
                        res.f[i][j][l]   =   std::max(res.f[i][j][l],
merged_0[l]);
                    }
                }

                auto    merged_1    =    merge_vectors(left.f[i][1],
right.f[1][j]);
                if (!merged_1.empty()) {
                    for (size_t l = 1; l < merged_1.size(); ++l) {
                        if (l > 0 && merged_1[l] > -INF/2) {
                            res.f[i][j][l-1]                        =
std::max(res.f[i][j][l-1], merged_1[l]);
                        }
                    }
                }
            }
        }
        return res;
    }

    void build() {
        tree_size = 1;
        while (tree_size < n) tree_size *= 2;
        seg_tree.assign(2 * tree_size, DPState());

        for (int i = 0; i < n; ++i) {
            seg_tree[tree_size + i].init_leaf(a[i]);
        }
        for (int i = tree_size - 1; i > 0; --i) {
            seg_tree[i] = merge_nodes(seg_tree[2 * i], seg_tree[2 * i
+ 1]);
        }
    }

    Result   get_optimal_for_lambda(const   std::vector<ll>&   v,   ld
lambda) {
        if (v.empty() || v[0] == -INF) {
            return {-1e18, 0};
        }
        int best_idx = 0;
        ld max_val = -1e18;
        if (v[0] != -INF) {
            max_val = (ld)v[0] - lambda * 0;
        }

        int low = 0, high = v.size() - 1;
        while(high - low >= 3) {
            int m1 = low + (high-low)/3;
            int m2 = high - (high-low)/3;
            ld v1 = (v[m1] == -INF) ? -1e18 : (ld)v[m1] - lambda * m1;
            ld v2 = (v[m2] == -INF) ? -1e18 : (ld)v[m2] - lambda * m2;
            if(v1 < v2) low = m1;
            else high = m2;
        }

        for(int i = low; i <= high; ++i) {
            if (v[i] != -INF) {
                ld current_val = (ld)v[i] - lambda * i;
                if(current_val > max_val) {
                    max_val = current_val;
                    best_idx = i;
                }
            }
        }
        return {max_val, best_idx};
    }

    DPState query_range_for_lambda(int node_idx, int cur_l, int cur_r,
int target_l, int target_r) {
        if (cur_l > target_r || cur_r < target_l) {
            return DPState();
        }
        if (cur_l >= target_l && cur_r <= target_r) {
            return seg_tree[node_idx];
        }

        int mid = cur_l + (cur_r - cur_l) / 2;
        DPState left_res = query_range_for_lambda(2 * node_idx, cur_l,
mid, target_l, target_r);
        DPState right_res = query_range_for_lambda(2 * node_idx + 1,
mid + 1, cur_r, target_l, target_r);

        return merge_nodes(left_res, right_res);
    }

    Result calculate_for_lambda(int l, int r, ld lambda) {
        DPState  query_res_node  =  query_range_for_lambda(1,  0,
tree_size - 1, l - 1, r - 1);
        return      get_optimal_for_lambda(query_res_node.f[0][0],
lambda);
    }

    void solve_query(const Query& q) {
        ld low = -1.5e9, high = 1.5e9;

        for(int i = 0; i < 100; ++i) {
            ld mid = low + (high - low) / 2;
            if (mid == low || mid == high) break;
            Result res = calculate_for_lambda(q.l, q.r, mid);
            if (res.count >= q.k) {
                low = mid;
            } else {
                high = mid;
            }
        }

        Result res_low = calculate_for_lambda(q.l, q.r, low);
        Result res_high = calculate_for_lambda(q.l, q.r, high);

        ll sum_at_low = round(res_low.value + low * res_low.count);
        ll   sum_at_high   =   round(res_high.value   +   high   *
res_high.count);

        ll count_at_low = res_low.count;
        ll count_at_high = res_high.count;

        if (count_at_low == count_at_high) {
            answers.push_back(sum_at_low);
        } else {
            ll  ans  =  sum_at_high  +  (ll)round((ld)(sum_at_low  -
sum_at_high)  *  (ld)(q.k  -  count_at_high)  /  (ld)(count_at_low  -
count_at_high));
            answers.push_back(ans);
        }
    }

public:
    TradeExecutionSolver() = default;

    void run() {
        std::ios_base::sync_with_stdio(0);
        std::cin.tie(0);
        std::cout.tie(0);
        std::cout << std::fixed << std::setprecision(0);

        std::cin >> n >> q;
        a.resize(n);
        for (int i = 0; i < n; ++i) std::cin >> a[i];

        queries.resize(q);
        for (int i = 0; i < q; ++i) {
            std::cin >> queries[i].l >> queries[i].r >> queries[i].k;
        }

        build();
```

```
        for (const auto& query : queries) {
            solve_query(query);
        }

        for (const auto& ans : answers) {
            std::cout << ans << "\n";
        }
    }
};
```

This class acts as the main engine and orchestrator for the entire solution.

Private Member Variables

- int n, q: Store the number of elements in the input array and the number of queries, respectively.
- int tree_size: The base size of the segment tree, calculated as the smallest power of 2 greater than or equal to n. This simplifies the tree's indexing and structure.
- std::vector<ll> a: Stores the input array of profit/loss values.
- std::vector<Query> queries: A vector to hold all the Query objects read from the input.
- std::vector<DPState> seg_tree: The segment tree itself, stored as a flat vector. Its size is 2 * tree_size.
- std::vector<ll> answers: A vector to store the final computed answer for each query.

Private Methods

- build(): Constructs the segment tree iteratively. It first populates the leaf nodes with the initial array values and then works its way up, merging child nodes to create parent nodes until the root is built.
- merge_nodes(...): Takes two DPState objects (from left and right child nodes) and computes the DPState for their parent. It considers all possible ways of combining subsegments, including merging them across the children's boundary.
- merge_vectors(...): An efficient helper function that merges two vectors representing concave functions. It operates on the vectors' slopes, which is much faster than a direct convolution.
- get_optimal_for_lambda(...): Given a DP vector (representing a concave function $F(i)$) and a penalty lambda, this function uses a ternary search to efficiently find the number of segments i that maximizes the expression $F(i) - \lambda i$.
- query_range_for_lambda(...): Performs a standard range query on the segment tree. Given a target range $[l \dots r]$, it traverses the tree and merges the DPStates of the relevant nodes to produce a single DPState for that exact range.
- calculate_for_lambda(...): A convenience wrapper that takes a query range $[l \dots r]$ and a lambda, calls query_range_for_lambda to get the DP state for that range, and then uses get_optimal_for_lambda to return the final Result.
- solve_query(...): This method implements the solution for a single query. It performs the binary search

(parametric search) on lambda to find the critical penalty value. It then calculates the final answer using the linear interpolation method described before.

Public Methods

- run(): The main public entry point for the class. It orchestrates the entire process:
    - Handles all I/O operations.
    - Initializes the tree_size.
    - Calls build() to construct the segment tree.
    - Loops through each query and calls solve_query() for it.
    - Prints all the stored answers.

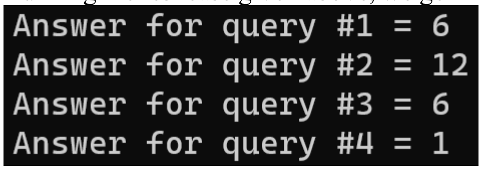
The full code can be accessed here: https://github.com/BP04/Makalah-Strategi-Algoritma.

VI. Test and Results

Given the test below

```
// Number of elements
6
// Array A
3, -5, 4, 2, -7, 6
// Number of queries
4
// Query# 1
1 6 1
// Query #2
1 6 2
// Query #3
2 5 2
// Query #4
2 5 3
```

Running the test case given above, we get



```
Answer for query #1 = 6
Answer for query #2 = 12
Answer for query #3 = 6
Answer for query #4 = 1
```

We can verify the answer to each query by finding the expected answer respectively.

- 1st query
  Choose the range [3 ... 4] with sum 6.
- 2nd query
  Choose the range [3 ... 4] and [6 ... 6] with sum 12.
- 3rd query
- Choose the range [3 ... 3] and [4 ... 4] with sum 6.
- 4th query
- Choose the range [3 ... 3], [4 ... 4], [5 ... 5] with sum 1.

VII. Conclusion

The algorithm presented offers a potent solution to a non-trivial optimization problem, providing a query-efficient method for determining the maximum potential value from a series of trading opportunities under cardinality constraints. While the model makes certain abstractions, its utility in strategy development, evaluation, and even in fields beyond finance is significant.

The algorithm's design makes two important trade-offs that are critical to its performance and practical application.

- Utility of Value-Only Optimization: The algorithm is intentionally designed to optimize for a single value—the maximum possible sum, rather than reconstructing the specific segments that produce this sum. In many HFT contexts, the primary need is for rapid, high-level decision-making. The optimal value serves as a theoretical upper bound for benchmarking live strategies and for quickly assessing whether the potential profit in a given window justifies the associated risks and costs.
- Utility in an Uncertain Market: The model assumes the input array of profits and losses is known beforehand. In a live market, this is not the case. The algorithm's value, therefore, lies not in direct execution but in modeling and backtesting. The input array can represent the forecasted returns from a predictive model. By running this algorithm on historical signal outputs, a firm can quantitatively determine the maximum theoretical profit that signal could have generated, providing an invaluable tool for comparing and refining different predictive models.

The framework of this algorithm is generalizable to a variety of domains beyond finance. At its core, it solves the problem of selecting a fixed number of non-overlapping intervals from a sequence to maximize a cumulative metric.

Future research could extend this work by incorporating more complex, real-world constraints, such as risk limits, transaction costs that are dependent on segment size, or granular liquidity considerations across different time intervals.

In summary, the combination of Lagrangian relaxation with a segment tree data structure provides a method for query-efficient range optimization problem. It transforms a difficult optimization and combinatorial problem into a manageable, queryable form, offering a valuable analytical instrument for assessing the potential of sequential opportunities in a constrained environment.

Appendix

GitHub: https://github.com/BP04/Makalah-Strategi-Algoritma

Acknowledgements

The author sincerely thanks God Almighty for providing the strength and opportunity to complete this paper successfully. The author also extends deep appreciation to Dr. Ir. Rinaldi, M.T., lecturer for IF2211 Algorithmic Strategies, for his guidance, encouragement, and support throughout the semester and while preparing this paper. Additionally, the author thanks Mr. Dewana Gustavus for his help in discussing the algorithm described in this paper.

References

[1] TOKI, "Pemrograman Kompetitif Dasar," OSN TOKI. [Online]. Available: https://osn.toki.id/data/pemrograman-kompetitif-dasar.pdf. [Accessed: Jun. 20, 2025].

[2] A. Laaksonen, *Competitive Programmer's Handbook*. Helsinki, Finland: Springer International Publishing AG, 2017. [Online]. Available: https://cses.fi/book.pdf. [Accessed: Jun. 20, 2025].

[3] S. İ. Birbil, "Lagrangian Relaxation," March 6, 2016. [Online]. Available: https://personal.eur.nl/birbil/bolbilim/teaa/02_Lag_Rel.pdf. [Accessed: Jun. 20, 2025].

PERNYATAAN

Dengan ini saya menyatakan bahwa makalah yang saya tulis ini adalah tulisan saya sendiri, bukan saduran, atau terjemahan dari makalah orang lain, dan bukan plagiasi.

Bandung, 24 Juni 2025

[illegible]

Benedict Presley
13523067