Global Predecessor Indexing: Linear-Time Weighted Job Scheduling via Efficient Sorting

Amit Joshi
amitjoshi2724@gmail.com
amit.joshiusa@gmail.com
Independent Researcher

Abstract

We improve the solution to the weighted job scheduling problem. While the classical dynamic programming (DP) solution runs in $O(n \log(n))$ time due to comparison-based sorting and per-job binary search [1], we show that the binary search bottleneck can be eliminated. In its place, we introduce a novel $O(n)$ multi-phase preprocessing technique called *Global Predecessor Indexing (GPI)*, which computes the latest non-overlapping job (i.e., the predecessor) for all jobs, enabling direct use in the classical DP recurrence. When combined with linear-time sorting, this yields a complete $O(n)$ solution. Even with comparison-based sorting, GPI significantly outperforms the classical solution in practice by avoiding the binary search bottleneck.

Keywords: Weighted Job Scheduling, Interval Scheduling, Dynamic Programming, Linear Sorting, Two Pointers

1. Introduction

The weighted job scheduling problem is a classic combinatorial optimization task arising in job scheduling, resource allocation, network planning, and more. Each job has a start time, end time, and weight (or value), and the goal is to select a subset of non-overlapping jobs that maximizes total weight.

For example, consider a queue of CPU jobs with integer timestamps and rewards (weights). Selecting the optimal non-overlapping subset becomes crucial for maximizing system value. This formulation also appears in compiler design, backup planning, and dynamic programming in bioinformatics.

The classical DP solution runs in $O(n \log(n))$ time for n jobs. It first sorts jobs by end time and then, for each job i, performs a $O(\log(n))$ binary search to find the latest job that ends before i starts, known as its predecessor $p(i)$ [1] and accesses this index during DP. This is efficient in general, but in settings where $O(n)$ sorting is achievable, the $O(\log(n))$ overhead becomes unnecessary. The classical DP has two bottlenecks: comparison-based sorting and repeated binary searches, yielding $O(n \log(n))$ total time.

In this work, we show that all predecessors can be computed in $O(n)$ time after sorting. We introduce a linear-time preprocessing technique called *Global Predecessor Indexing (GPI)*, which uses linear sorting and then computes all predecessor indices in a single $O(n)$ pass. The final DP step then runs in $O(n)$ time using these predecessor indices. We also discuss the parallelizability of our approach. To our knowledge, this is the first $O(n)$ algorithm for weighted job scheduling when job times admit linear-time sorting, e.g., bounded integers or smooth distributions.

2. Background

2.1. Classical DP Framework

Each job i has a start time s_i, end time e_i, and weight w_i. Two jobs i and k are compatible if their intervals do not overlap, i.e., if $e_i \leq s_k$ or $e_k \leq s_i$. All jobs in the chosen subset must be pairwise compatible, and our aim is to find the optimal subset to maximize the sum of its intervals' weights.

The classical DP algorithm sorts jobs by increasing end time and computes an array `dp`, where `dp[i]` stores the maximum achievable total weight using at most the first i jobs [1]. For each job i, it finds the latest (in order of increasing end time) job k such that $e_k \leq s_i$, defining $p(i) = k$. This is done with binary search over the sorted end times in $O(\log(n))$ time. Over all n jobs, this yields $O(n \log(n))$ time complexity.

The DP recurrence is:

$$\texttt{dp[i]} = \max(\texttt{dp[i-1]}, w_i + \texttt{dp}[p(i)])$$

where $p(i)$ is the predecessor index of job i. If no predecessor exists for a job i (i.e., if no job ends at or before s_i), we define $p(i) = 0$. The base case is `dp[0]` $= 0$, corresponding to an empty job set. Once the `dp` array is computed, the actual set of jobs comprising the optimal solution can be recovered in $O(n)$ time via standard backtracking. Starting from $i = n$, we compare `dp[i]` to `dp[i-1]`: if they are equal, job i was excluded; otherwise, job i was included, and we continue the trace from its predecessor $p(i)$. The selected jobs can be stored in reverse and then returned in sorted order.

Algorithm 1 Classical DP Weighted Job Scheduling

Require: Array `jobs` of n jobs
Initialize `end_ordered` $\leftarrow$ `jobs` sorted by increasing e_i
function FindPredecessorBinarySearch(i)
    Binary search for largest $k < i$ such that $e_k \leq s_i$
    return k or 0 if no such k exists
end function
Initialize `dp` $\leftarrow [0, 0, \ldots, 0]$
for $i = 1$ to n **do**
    $k \leftarrow$ FindPredecessorBinarySearch(i)
    `dp[i]` $\leftarrow \max($`dp[i-1]`$, w_i +$ `dp[k]`$)$
end for
return `dp[n]`

2.2. Linear Sorting Algorithms

Standard comparison-based sorting algorithms run in $O(n \log(n))$ time. However, in many practical cases, job times can be sorted in linear time such as when they are bounded integers or drawn from smooth distributions. In such scenarios, specialized linear-time sorting algorithms can achieve $O(n)$ time in the worst case or expectation. We briefly describe several below.

Counting Sort [2] runs in $O(n + K)$ time, where K is the largest input value. It counts occurrences of each value and reconstructs the sorted array stably. It works best when $K = O(n)$, and is inefficient when the input domain is large or sparse.

Radix Sort [2] achieves $O(d(n + b))$ time, where d is the number of digits and b is the base. It uses Counting Sort as a stable subroutine per digit. Radix Sort behaves linearly when d is small and can also handle fixed-width decimals.

Bucket Sort [3] divides input into n evenly spaced buckets, sorts each bucket (using a comparison-based method), and concatenates the results. It has expected $O(n)$ time for roughly uniform inputs, but degrades to $O(n \log(n))$ or worse under skew. It is a good choice when $K \gg n$ or the input range is sparse.

Spreadsort [4] is a hybrid algorithm with expected $O(n)$ time. It uses high-order bits or leading digits to partition inputs, recurses on dense buckets, and falls back to comparison-based sorting for small buckets. Unlike Bucket Sort, it adapts to non-uniform inputs and is suitable for many real-world distributions.

3. Our Approach

We present a linear-time algorithm for the weighted job scheduling problem assuming job start and end times admit approximately linear-time sorting. The overall algorithm runs in $O(n)$ time and is simple, cache-friendly (array-based), and practical for large-scale instances. To overcome the bottleneck of repeated binary searches to identify predecessors, we introduce our multi-phase preprocessing pipeline called *Global Predecessor Indexing (GPI)* and compute all predecessor indices in a single $O(n)$ pass.

In the first phase, we sort the jobs by increasing end time using a suitable linear-time sort and expand each job state to include its ordering within this sort. In the second phase, we linearly sort again by increasing start time on the array with the expanded jobs. We use both orderings to find all predecessor indices with a clever two-pointer $O(n)$ algorithm in the third phase.

Finally, we use these precomputed predecessor indices with the standard DP from Section 2 to compute the final numeric answer: the maximum achievable weight from a subset of non-overlapping job intervals. This eliminates the $O(\log(n))$ binary search overhead with the classical DP, while using the same recurrence.

3.1. Sorting Jobs by End Time, Reordering, and Job State Expansion

We first sort jobs by increasing end time, and ties may be broken arbitrarily. Then, we reorder the given jobs by this sorted order. In other words, j_1 is the first job in this order, j_2 is next, and so on. We refer to index i as the "end-order" of j_i. By design, $\forall i < n \;:\; e_i \leq e_{i+1}$. As we are reordering, the initial ordering may be discarded.

Lastly, we append each job state j_i with its end-order.

$$j_i \leftarrow (s_i, e_i, w_i, i)$$

And we now refer to the array $[j_1, \ldots, j_n]$ as the `end_ordered` array. For clarity, we also use "job i" to refer to j_i, the job with end-order i. Additionally, $p(i)$ refers to the end-order of the predecessor job of j_i.

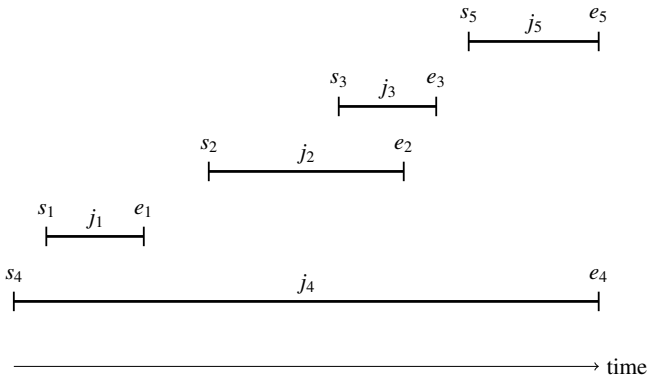


Figure 1: Example job intervals reordered by end time.

3.2. Sorting Jobs by Start Time

Unlike classical DP, we sort once again by start time into a new array since we will need both orderings for the next phase. Similarly, we refer to a job's index in the start time ordering as its "start-order." By sorting again, we implicitly define a permutation π over $\{1, \ldots, n\}$ which maps from a job's start-order to its end-order. By design, $\forall i < n \;:\; s_{\pi(i)} \leq s_{\pi(i+1)}$. We refer to $[j_{\pi(1)}, \ldots, j_{\pi(n)}]$ as the `start_ordered` array. By construction, $\pi(i)$ is the fourth component of the i^{th} job in the `start_ordered` array—due to the job state expansion from Section 3.1) For clarity, we also use "job $\pi(i)$" to refer to $j_{\pi(i)}$, the job with start-order i.

start-order i	end-order $\pi(i)$
1	4
2	1
3	2
4	3
5	5

Table 1: Permutation $\pi(i)$ mapping start-order to end-order of the jobs from Figure 1, created by sorting again by increasing start time.

3.3. Precomputing Predecessor Indices in $O(n)$ time

Next, we describe our two-pointer single $O(n)$ pass to compute all predecessors, which avoids the repeated binary searches from the classical DP solution. We store predecessor indices in an array aligned with end-order, where the i^{th} entry of `p` holds the predecessor $p(i)$ for job i.

We find the predecessor for each job in decreasing start-order, i.e., find the predecessor for job $\pi(n)$, then job $\pi(n-1)$, down to job $\pi(1)$. For this, we use `start_index` to traverse backward through `start_ordered` in our outer loop. We also have an inner loop index, `end_index`, which traverses backward through `end_ordered`.

Before the outer loop begins, we use a single binary search on `end_ordered` to initialize `end_index` with $p(j_{\pi(n)})$: the predecessor index of job $\pi(n)$. Since no job starts later than job $\pi(n)$, the highest end-order of any predecessor is $p(j_{\pi(n)})$.

Within the outer loop, the inner loop decrements `end_index` until we find a job that ends no later than job π(`start_index`) starts. Since we index *backwards* through `end_ordered`, once we have found such a job, we are guaranteed to have found $p(j_{\pi(\texttt{start\_index})})$: the job with the highest end-order that ends before job π(`start_index`) starts. If `end_index` hits 0, then any remaining jobs with unassigned predecessors do not have valid predecessors since they start before any other job ends and we set their entries in `p` to 0. Once the outer loop finishes, all predecessors have been correctly assigned.

Algorithm 2 Precompute Predecessors

Require: Array `end_ordered` of n jobs. Appears as $[j_1, \ldots, j_n]$ where each (s_i, e_i, w_i, i) sorted by increasing e_i.
Require: Array `start_ordered` of n jobs. Appears as $[j_{\pi(1)}, \ldots, j_{\pi(n)}]$ where each $(s_{\pi(i)}, e_{\pi(i)}, w_{\pi(i)}, \pi(i))$ sorted by increasing $s_{\pi(i)}$

```
function EndsNoLaterThanStarts(k, i)
    Check if j_k ends no later than j_i starts
    return true if e_k ≤ s_i, false otherwise
end function
function FindPredecessor(i)
    Binary search over end_ordered for largest k < i such that e_k ≤ s_i
    return k or 0 if no such k exists
end function
function ComputePredecessors(n)
    Initialize p[1...n] ← [0...0]                ▷ predecessor array
    end_index ← FindPredecessor(π(n))
    for start_index = n to 1 do
        while end_index ≥ 1 and
        (not EndsNoLaterThanStarts(end_index,
        π(start_index))) do
            end_index ← end_index − 1
        end while
        if end_index = 0 then                    ▷ Remaining jobs with start-order ≤ start_index do not have predecessors, so their entries in p remain 0
            break
        end if
        p[π(start_index)] ← end_index
    end for
    return p
end function
```

Job i (end-order i)	Predecessor index $p(i)$
1	0
2	1
3	1
4	0
5	3

Table 2: Predecessor indices $p(i)$ corresponding to the intervals in Figure 1.

Time Complexity

We analyze the time complexity of Algorithm 2: Precompute Predecessors. First, we perform a single binary search to find $p(\pi(n))$ which takes $O(\log(n))$ time. Next, although the `while` loop is nested within the `for` loop, the crucial observations are that:

- `start_index` decrements monotonically and visits each index from $[1 \ldots n]$ at most once, and so the number of iterations of the outer loop is at most n.
- `end_index` is initialized before the outer loop, decrements monotonically, and visits each index from $[0 \ldots n]$

at most once. The *total* number of iterations of the inner loop across the function is at most n.

Each iteration of the inner loop performs a fixed number of $O(1)$ operations: evaluating EndsNoLaterThanStarts (two array accesses and a comparison) and decrementing `end_index`. Each iteration of the outer loop decrementing `start_index`) also performs a fixed number of $O(1)$ operations: evaluating the comparison on line 16 and the array assignment on line 19. All of these operations—including arithmetic updates, logical comparisons, and array accesses or assignments—are primitive and take constant time, i.e., $O(1)$. Since the total number of `start_index` and `end_index` decrements is bounded by $2n$, and each decrement involves only $O(1)$ work, the total runtime remains $O(n)$.

Correctness

We prove correctness of our novel preprocessing step via a Loop Invariant over the outer (**for**) loop on line 12.

Loop Invariant. At the beginning of line 16 during iteration `start_index` $= i$, `end_index` is the predecessor of job $\pi($`start_index`$) = i$. Although the loop begins at line 12, this invariant is defined to hold specifically at line 16, just before the predecessor for job $\pi($`start_index`$)$ is assigned on line 19.

Base Case. Before the first iteration of line 16, FindPredecessor uses binary search to initialize `end_index` with $p(j_\pi(n))$. Since `start_index` $= n$ in the first iteration and by the correctness of the classical DP, the loop invariant holds true.

Maintenance. Assume the invariant holds at the start of line 16 during iteration `start_index` $= i$. Hence, `end_index` equals $p(\pi($`start_index`$))$. Now, to show that in the next iteration, `start_index` $= (i-1)$ the invariant also holds true. First, we prove that $p(\pi(i-1)) \leq p(\pi(i))$:

Proof. By way of contradiction, assume the opposite, so $p(\pi(i-1)) > p(\pi(i))$. By definition of predecessor, job $p(\pi(i-1))$ ends no later than job $\pi(i-1)$ starts. By construction, job $\pi(i-1)$ starts no later than job $\pi(i)$ starts. Then, by transitivity, job $p(\pi(i-1))$ ends no later than job $\pi(i)$ starts.

Due to our assumption for contradiction that $p(\pi(i-1)) > p(\pi(i))$, $p(\pi(i))$ can not be the predecessor for job $\pi(i)$ since there is a higher end-order job, job $p(\pi(i-1))$, that ends no later than job $\pi(i)$ starts. However, $p(\pi(i))$ is indeed defined to be the predecessor for job $\pi(i)$. □

Now, we know that the correct `end_index` for iteration `start_index` $= (i-1)$ occurs at or before the correct `end_index` from iteration `start_index` $= i$. As we are simply doing a linear search backwards through `end_ordered`, we are guaranteed to find the job with the highest end-order that ends no later than job $\pi(i-1)$ starts: its predecessor. Lines 13-15 decrement `end_index` until the invariant is restored or maintained, so that when control reaches line 16, the invariant still holds.

Termination. When the loop terminates after iteration `start_index` $= 1$, the invariant has held at line 16 in every iteration `start_index` $= i$ that was reached and correctly assigning $p[\pi(i)]$ to `end_index` on line 19. Any outer-loop iterations that were not reached were because `end_index` hit 0, which all entries in `p` were initialized to anyway. Since π is a permutation over $\{1,\ldots,n\}$ and `start_index` looped from n to 1, then `p[i]` was set at most once for each $i \in \{1,\ldots,n\}$ with the correct `end_index`. Thus, `p[i]` correctly stores the predecessor for all jobs i.

Parallelizability

The outer loop can be split across multiple threads. Each thread can handle a continuous segment of `start_index` values in reverse order, say $[u_m,\ldots,u_1]$. The cost is one binary search per thread to initialize `end_index` with the predecessor of job u_m. As each thread can compute the predecessor of many jobs, this is a good tradeoff.

3.4. Dynamic Programming

Assuming the predecessor indices $p(i)$ have already been computed (see Section 3.3) and we have the resulting predecessor lookup table `p`, we now describe the final DP step from Section 2:

$$\texttt{dp[i]} = \max(\texttt{dp[i-1]}, w_i + \texttt{dp[p[i]]})$$

with base case `dp[0]` $= 0$ by convention. During the DP, we can retrieve each job's predecessor in constant time via the lookup table created during GPI, eliminating the need for binary search and providing the main speed-up of this paper. Since all array accesses and maximum computations are $O(1)$, the total time to compute the DP table and the final answer is $O(n)$.

Correctness

As in the classical solution, `dp[i]` holds the maximum achievable total weight among the first i jobs. We prove correctness by induction on i. The base case `dp[0]` $= 0$ is valid as no jobs have been selected. Assume `dp[i-1]` is correct. Then `dp[i]` either inherits the optimal solution for the first $(i-1)$ jobs or takes the optimal solution for the predecessor $p(i)$—`dp[p[i]]`—plus w_i. Since $p(i)$ was correctly computed, and since the recurrence takes the maximum of these two cases, the solution remains optimal. Since `dp[n]` now represents the maximum achievable total weight among the first n jobs, and there are n jobs total, it holds the final answer after the loop ends.

Parallelizability

Since all predecessor relationships are known in advance, the DP step can be parallelized using Kahn's topological sort [5], where each job depends only on its predecessor job's DP value. The effectiveness of this parallelization depends on how often jobs are predecessors to multiple other jobs.

4. Experimental Results

To demonstrate the practical advantage of GPI — our linear-time algorithm — we benchmark it against the classical $O(n \log(n))$ DP solution with binary search. Our experiments are designed to isolate the cost of binary search in the classical method. Although linear sorting is available, we found that classical DP performs best with comparison-based sorting, so we used it to ensure a fair comparison. We release our full algorithm implementation and benchmarking code[1].

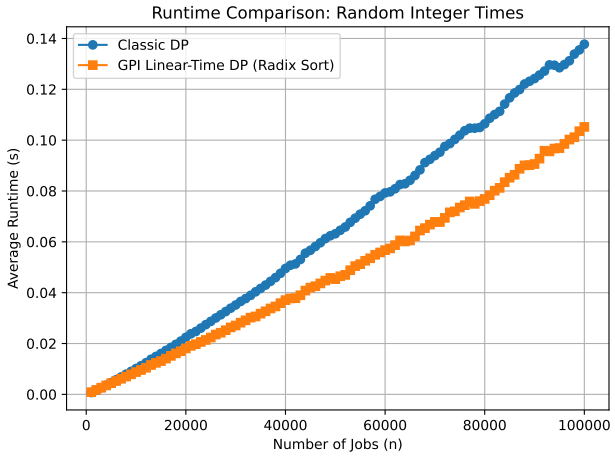
4.1. Benchmark Design

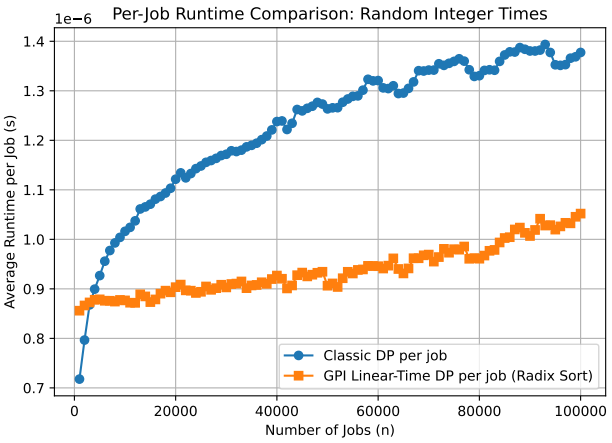
We design four controlled experiments to evaluate the performance of GPI under different job generation patterns. Each experiment captures different structural characteristics of real-world workloads, stressing various aspects of job scheduling.

Both the classical and our linear-time algorithms are implemented in Python 3 and benchmarked on the same hardware. Final answers were cross-checked for sanity. Runtimes are averaged over 10 trials per input size to mitigate noise. Garbage collection is triggered only before a trial, not during. As Spreadsort was unavailable to us in Python, we substitute a similar Recursive Bucket Sort. Our Recursive Bucket Sort partitions the data based on value ranges (like Bucket Sort), recursively sorts large buckets, and terminates with Timsort [6] for small buckets. Despite its simplicity, it achieves empirical linear-time performance under smooth distributions, validating its suitability for preprocessing in weighted job scheduling.

In each trial, we generate n jobs (an instance), and for each experiment, we vary the number of jobs n from 1,000 to 100,000 and plot two runtime metrics to visualize scaling trends:

- **Average runtime (seconds)**: Measures wall-clock time to solve a full instance.
- **Average runtime per job (seconds/job)**: Total time divided by n, useful for visualizing marginal cost and highlighting logarithmic overhead within the classical DP.





4.2. Experiment Details

Experiment 1: Random Integer Times. Job start and end times are drawn independently and uniformly from a large integer domain, i.e., $[0, 10^6]$. This setup represents no special structure — jobs vary arbitrarily in duration, overlap randomly, and predecessors are neither clustered nor sparse. This test acts as a baseline reference for expected behavior under generic conditions. This reference resembles mixed workloads in general-purpose scheduling applications. Our GPI uses Radix Sort to sort the job times as they are bounded integers with few digits.

Experiment 2: Normally Distributed Start Times (Smooth Clustered Inputs). Job start times are drawn from a truncated normal distribution $N\left(\frac{K}{2}, \left(\frac{K}{10}\right)^2\right)$ within $[0, K]$, where $K = 10^9$ and durations are sampled uniformly from a small float range (i.e., $[1.0, 1000.0]$). This models job bursts in real-world systems with smooth, dense clusters. Our GPI uses Recursive Bucket Sort, which efficiently handles such smooth (yet non-uniform) input distributions.

Experiment 3: Zipf Durations with Early Start Bursts. Job start times are drawn from a truncated exponential distribution (clustered near 0), modeling early bursts commonly seen in real-world workloads. Durations follow a scaled Zipf distribution with exponent $s = 2$ (i.e., $d = 100 \cdot Z$ for $Z \sim \text{Zipf}(2)$), capped at 10^6 to prevent extreme outliers. This yields mostly short jobs with a few long ones, causing some jobs to have nearby predecessors while others reference distant ones, which disrupts cache locality during predecessor lookups. Our GPI uses Recursive Bucket Sort to efficiently handle this skewed real-valued input distribution. This experiment highlights how our algorithm performs under strongly non-uniform and cache-unfriendly predecessor distributions.

Experiment 4: Bucket-Sort-Friendly Uniform Start Times. Job start times are drawn uniformly from $[0, K]$ with $K \gg n$, and durations are sampled uniformly from a small float range (i.e., $[1.0, 1000.0]$), so the end times are approximately uniform as well. This produces a well-spread timeline ideal for Bucket Sort, which our GPI uses. The uniform distribution ensures

[1] https://github.com/amitjoshi2724/gpi-job-scheduling

minimal collisions per bucket. Collisions are sorted using Timsort (a hybrid merge/insertion sort that exploits ordered subarrays [6]), highlighting Bucket Sort's strengths when input times are approximately uniformly-distributed across a large domain.

4.3. Results

Our results align with theoretical expectations. Classical DP always shows mild upward curvature in the Overall Runtime charts, consistent with its $O(n\log(n))$ time complexity. In the Per-Job Runtime charts, Classical DP displays a logarithmic trend, reflecting the increasing marginal cost of processing each additional job.

GPI in Experiments 1 and 4 achieved true linear sorting, producing a linear slope in the Overall Runtime charts. The corresponding Per-Job Runtime curves remain mostly flat, indicating that the marginal runtime per job is constant, which supportis our linear-time claim.

GPI in Experiments 2 and 3 used Recursive Bucket Sort on non-uniform inputs so the sorting was not strictly linear. We still observe significant speedups as n increases, as well as relatively flat Per-Job Runtime curves. This result exhibits strong practical performance even under input distributions that deviate from ideal conditions.

5. Conclusion

We show that when job times permit linear-time sorting or close to it, such as when they are bounded integers or drawn from smooth distributions, GPI can solve the weighted job scheduling problem can be solved in $O(n)$ time instead of the classical $O(n\log(n))$. GPI sorts twice, and then replaces repeated binary searches with a single linear-time pass to find all predecessors and uses array-indexed predecessor lookups during DP. Our solution gives a theoretical and practical speedup over the classical DP as well as being conceptually simple, cache-friendly, and even parallelizable.

References

[1] Jon Kleinberg and Éva Tardos. *Algorithm Design*. Pearson, 1st edition, 2005. https://www.pearson.com/en-us/subject-catalog/p/algorithm-design/P200000000305/9780321295354

[2] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press, 3rd edition, 2009. DOI: 10.7551/mitpress/9435.001.0001

[3] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *Data Structures and Algorithms*. Addison-Wesley, 1983. ISBN: 978-0-201-00023-8.

[4] Steven J. Ross, "The Spreadsort High-performance General-case Sorting Algorithm," in *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA)*, Volume 3, CSREA Press, 2002, pp. 1100–1106.

[5] Arthur B. Kahn, "Topological sorting of large networks," *Communications of the ACM*, vol. 5, no. 11, pp. 558–562, 1962.

[6] Tim Peters. *Timsort: A hybrid stable sorting algorithm*, 2002. https://github.com/python/cpython/blob/main/Objects/listsort.txt.