Algorithms

Jane Doe

Table of Contents

1. Greedy Algorithms

1.1. Interval Scheduling

Lecture 1 — Jan 1, 2025

1.1.1. Computational Problem: Interval Scheduling

Suppose we have a list of n tasks that we want to schedule on a single processor. Each activity is specified by its start and end times, and only one activity can be scheduled on the resource at a time. Note an activity cannot be paused, i.e., it uses the resource continuously between its start and end times.

Our goal is to schedule the maximum possible number of tasks.

1.1.2. Remark

We can formalize the previous problem as follows:

Call our set of tasks $S = \{1, 2, ..., n\}$. The i-th task is given by a tuple $(s(i), f(i))$ with $s(i) \leq f(i)$, for $i = 1, ..., n$, where $s(i)$ represents its start time and $f(i)$ represents its finish time.

Define a *feasible schedule* as a subset in which no two tasks overlap. Thus, our goal is to find a feasible schedule of maximum size.

1.1.3. Example

Suppose we have 5 tasks:

$$\{(3, 6), (1, 4), (4, 10), (6, 8), (0, 2)\}$$

A feasible schedule cannot have two activities that overlap, so we cannot accept both (1, 4) and (3, 6). However, (3, 6) and (6, 8) are acceptable, because second only begins when first ends. Note that the optimal solution in this example has size 3.

1.1.4. Remark

We have a few possible greedy strategies:

1. First In First Out (FIFO): pick the task that starts first, remove overlapping activities and repeat
2. Shortest first: pick the activity with the shortest duration, remove overlapping activities and repeat
3. Min overlap first: Count the number of other jobs that overlap, then choose one with smallest overlap count

But if we analyze these, we can find simple counterexamples for the first one (an activity that's really long is first) and the third one (a shorter interval in the middle that overlaps with 2 which could fit together). We could also construct a counterexample example for the second option.

1.1.5. Algorithm: Earliest Finish Time (EFT)

We claim that the optimal greedy strategy is to sort the process jobs in order by earliest finish time.

We sort jobs by order:

$$f(j_1) \leq f(j_2) \leq \cdots \leq f(j_n)$$

Pseudocode

```
1 A = {1}; j = 1;
2 for i = 2 to n do
3 |  if s(i) >= f(j) then
4 |  |  A = A + {i}; j = i;
5 return A
```

Python Implementation

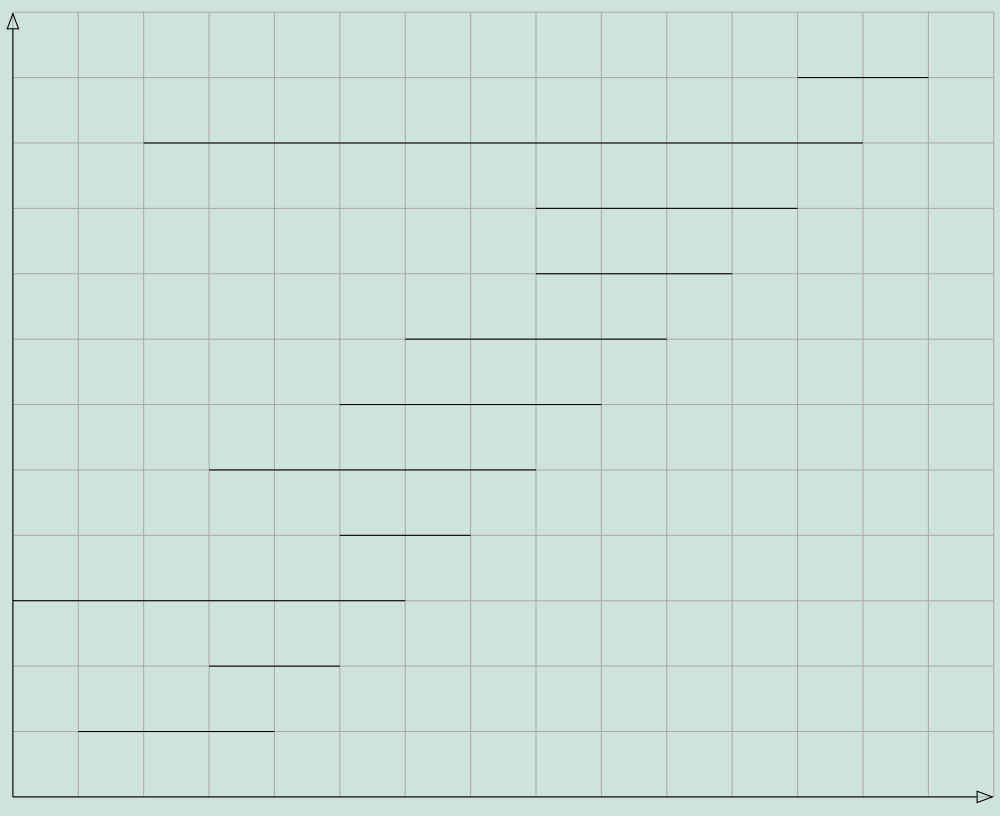
```py
S = [(3, 6), (1, 4), (4, 10), (6, 8), (0, 2)]

def eft(S: list[tuple[int]]) -> list[int]:
    A, j = [], 1
    for i in range(2, n):
        if S[i][0] >= S[j][1]:
            A.append(i)
            j = i

    return A
```

1.1.6. Example

Activity	Start	Finish
1	1	4
2	3	5
3	0	6
4	5	7
5	3	8
6	5	9
7	6	10
8	8	11
9	8	12
10	2	13
11	12	14



The greedy algorithm first chooses `1`, then skips `2` and `3`. Next it chooses `4`, and skips `5`, `6`, and `7`. Finally, it should choose `8` and `11`.

1.1.7. *Proposition*

> The Earliest Finish Time Algorithm correctly solves the Interval Scheduling Problem.

Proof: Suppose $a_1, ..., a_k$ are the indices of jobs in the greedy schedule, and $b_1, ..., b_m$ are jobs in an optimal schedule OPT. We want to show $k = m$.

Our intuition should be that the greedy algorithm makes the resource free again as soon as possible. In particular, $f(a_1) \leq f(b_1)$, that is, the greedy algorithm stays ahead. We formalize this as follows.

1.1.8. *Lemma:* Greedy Structure Lemma

> For every $i \leq k$, we have $f(a_i) \leq f(b_i)$.

Proof: We proceed by induction. Note the $i = 1$ case is trivial. By the induction hypothesis, we have $f(a_{i-1}) \leq f(b_{i-1})$. Therefore $f(a_{i-1}) \leq s(b_i)$, since clearly $f(b_{i-1}) \leq s(b_i)$. But notice $s(b_i) \leq f(b_i)$, which implies $f(a_i) \leq f(b_i)$ as desired.

□

Notice by the lemma, $f(a_k) \leq f(b_k)$, but $f(b_k) \leq f(b_m)$, so $f(a_k) \leq f(b_m)$, implying $k \leq m$. But $m \leq k$ since OPT is optimal, implying $m = k$ as desired.

□

1.1.9. Runtime Analysis

Since the Earliest Finish Time algorithm involves sorts its values, it is at least $O(n \log n)$. Then notice we iterate through the sorted values, an $O(n)$ operation, so the algorithm is $O(n \log n)$ overall.