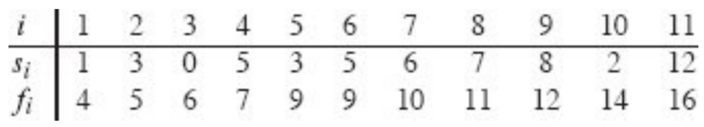
# Assignment 5

Qiming Lyu

2025-11-10

1. *Proof.* We follow the textbook's activities $a_i$ given by Figure 1.



| $i$ | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| $s_i$ | 1 | 3 | 0 | 5 | 3 | 5 | 6 | 7 | 8 | 2 | 12 |
| $f_i$ | 4 | 5 | 6 | 7 | 9 | 9 | 10 | 11 | 12 | 14 | 16 |

Figure 1: A set $\{a_1, a_2, \dots, a_{11}\}$ of activities.
Activity $a_i$ has start time $s_i$ and finish time $f_i$.

Greedy-by-earliest-start picks $a_3$ first since $s_3 = 0$. After taking $a_3$, the remaining compatible activities must start at at least 6, so the best you can do is, for example, $\{a_3, a_7, a_{11}\}$, which has size 3.

But there exists a larger compatible set, for example, $\{a_1, a_4, a_8, a_{11}\}$, which has size 4.

Therefore, choosing the activity that starts first is **NOT** a valid greedy strategy. □

2. - **Algorithm**:
   (a) Sort the elements of $A$ in ascending order, such that
   $$a_1 \le a_2 \le \cdots \le a_n.$$
   (b) Sort the elements of $B$ in ascending order, such that
   $$b_1 \le b_2 \le \cdots \le b_n.$$
   (c) The bijective function $f$ is defined as
   $$f(a_i) = b_i, \quad \forall\, i = 1, 2, \dots, n.$$
   - **Proof of correctness**:

   *Proof.* Take any bijection $g$. If there exists $i < j$ with $a_i < a_j$ but $g(a_i) = b_j$ and $g(a_j) = b_i$ where $b_i \le b_j$, then swapping the images of $a_i$ and $a_j$ changes the score by
   $$(a_i b_i + a_j b_j) - (a_i b_j + a_j b_i) = (a_j - a_i)(b_j - b_i) \ge 0. \tag{1}$$
   So swap increases the score and makes no difference when one of the pairs is equal.

By repeatedly fixing such inversions, we eventually reach the bijection defined by the algorithm, which must have the maximum score.

BTW, note that (1) holds for any real numbers, which includes negative numbers as well. This means the algorithm works regardless of whether the numbers in $A$ and $B$ are positive or negative. □

- **Runtime**: Sorting $A$ and $B$ each takes $O(n \log n)$ time, and multiplying and adding the pairs takes $O(n)$ time. Therefore, the total runtime is

$$O(n \log n).$$

3. - **Algorithm**:
    - Sort the elements of $S$ in ascending order, such that

$$x_1 \leq x_2 \leq \cdots \leq x_n.$$

```
– Initialize an empty set $U$ of unit intervals and set $i \leftarrow 1$.
– While $i \leq n$:
    ∗ Let $a \leftarrow x_i$. Add the unit interval $[a, a+1]$ to $U$.
    ∗ Advance $i$ to the first index such that $x_i > a + 1$.
```

- **Proof of correctness**:

*Proof.* Let $U = \{[a_1, a_1 + 1], [a_2, a_2 + 1], \ldots, [a_k, a_k + 1]\}$ be the set of unit intervals returned by the algorithm, where $a_1 < a_2 < \cdots < a_k$.

We prove by induction on the number of points. Base case, $S = \emptyset$, the algorithm returns $U = \emptyset$, which is optimal.

We first show the structure of an optimal solution. Let $x$ be the leftmost uncovered point when the algorithm chooses its next interval. Any feasible solution must contain some interval $[b, b+1]$ such that $b \leq x \leq b+1$. The optimal solution always takes $b = x$ so that it can cover as many points as possible to the right of $x$ and does not waste any coverage to the left of $x$. Hence there exists an optimal solution whose first interval is $[x, x+1]$.

Apply this with $x = x_1$, the leftmost point in $S$, we see that there exists an optimal solution whose first interval is $[a_1, a_1 + 1]$.

Then we examine the optimal substructure. Remove all the points in $[a_1, a_1 + 1]$, and call the remaining set $S' = \{x | x \in S \wedge x > a_1 + 1\}$. We see that covering $S'$ with the minimum number of unit intervals is exactly the same as covering $S$ after fixing the first interval to be $[a_1, a_1 + 1]$. By the inductive hypothesis, our algorithm covers $S'$ optimally. Therefore, our algorithm covers $S$ optimally.

□

4.

```
function MATRIX-MULTIPLY(A, B)
    // A is a p x q matrix
    // B is a q x r matrix
    // Returns the product matrix C = A * B of size p x r
    require A.cols = B.rows
    p ← A.rows
    q ← A.cols
    r ← B.cols
    C ← A zero matrix of size p x r

    for i ← 1 to p do
        for j ← 1 to r do
            for k ← 1 to q do
                C[i, j] ← C[i, j] + A[i, k] * B[k, j]
            done
        done
    done

    return C
end

function MULTIPLY-CHAIN(i, j, s, A)
    // A is array of matrices A[1..n]
    // s is the split table computed by MATRIX-CHAIN-ORDER
    // i and j are the indices of the subchain A[i..j] to be multiplied
    // Returns the product of matrices A[i..j]
    if i = j then
        return A[i]

    k ← s[i, j]
    L ← MULTIPLY-CHAIN(i, k, s, A)
    R ← MULTIPLY-CHAIN(k + 1, j, s, A)
    return MATRIX-MULTIPLY(L, R)
end
```

Listing 1: MultiplyChain(i, j, s, A) Implementation