Homework 5 Solutions
Fundamental Algorithms, Spring 2025, Professor Yap, Section Leader Dr. Bingwei Zhang

Due: Mon Apr 7, in GradeScope by 11:30pm.
HOMEWORK with SOLUTION

INSTRUCTIONS:

- We have a "NO LATE HOMEWORK" policy.
  Special permission must be obtained *in advance* if you have a valid reason.
- Any submitted solution must be fully your own (you must not look at a fellow student's solution, *even if you have discussed with him or her*). Likewise, you must not show your writeup to anyone. We take the academic integrity policies of NYU and our department seriously. When in doubt, ask.
- The official deadline is 11:30pm, but can resubmit as many times as you like before that time.

(Q1) (4x5 Points)
Exercise VI.1.2, p. 6.
Exact cost of binary counter from n to m.

THE QUESTION Recall that our cost model for incrementing a binary counter.
(a) What is the *exact number* of work units to count from 0 to 99?
(b) What is the *exact expression in n* for the work units to count from 0 to n?
(c) What is the *exact expression in m and n* for the work units to count from m to n ($m \leqslant n$)?
(d) Use your formula in (c) to determine the exact cost to increment from 100 to 200.

SOLUTION: (a) Exact answer is 194. You could do this the painful way by counting from 1 to 99. Note that $99 = (1100011)_2$ in binary since $99 = 64 + 32 + 2 + 1 = 2^6 + 2^5 + 2^1 + 2^0$. Thus $\Phi(D_{99}) = 4$. Then using the formula in part(b), we obtain a cost of $2 \cdot 99 - \Phi(D_{99}) = 198 - 4 = 194$ work units.
(b) It costs $2n - \Phi(D_n)$ work units to count from 0 to n.
(c) It costs $2(m - n) - \Phi(D_n) + \Phi(D_m)$. Justification: CHARGE = COST + $\Delta\Phi$. Therefore COST = CHARGE - $\Delta\Phi$. Charge here is $2(n - m)$ and $\Delta\Phi = \Phi(D_n) - \Phi(D_m)$.
(d) Answer: the exact cost is 200.

Justification: The exact cost is $TotalCharge - \Delta\Phi$. Clearly, $TotalCharge = 200$. It is not hard to see that $\Phi_{200} = \Phi_{100}$ because the number of 1's in the binary notation of 100 and 200 are identical. Thus $\Delta\Phi = 0$.

(Q2) (6x5 Points)
Exercise VI.2.3, p. 16.
Insertion and deletion on splay trees.

THE QUESTION Let T be a binary tree with nodes $u_1, \ldots, u_5$ where u_1 is root, u_{i+1} is the child of u_i. However, u_{i+1} is a left-child if i is odd, and a right-child if i is even.
(a) Attach the five keys $1, 2, 3, 4, 5$ to the nodes in T so that the result is a splay tree. Draw the resulting tree, denoted T_a.
(b) Splay 3 in T_a and show the resulting tree, denoted T_b.
(c) Insert 6 in T_a and show the resulting T_c.
(d) Insert 3.5 in T_a and show the resulting T_d.
(e) Delete 3 in T_a and show the resulting T_e.
(f) Delete 2 in T_a and show the resulting T_f.

SOLUTION FIGURE:

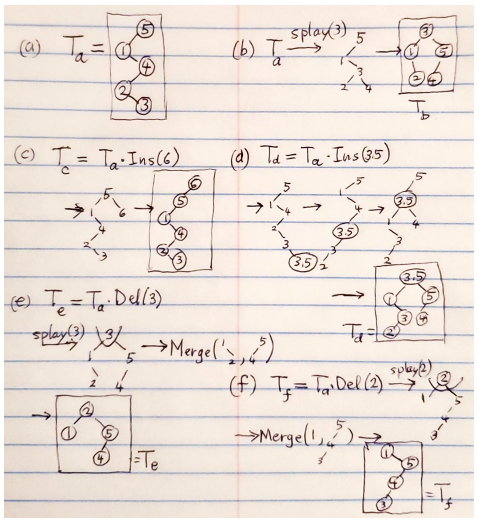

Figure 1: Splay trees $T_a, T_b, T_c, T_d, T_e, T_f$ CHECK: did I forget to splay T_c?

SOLUTION: (a) $u_1.key = 5, u_2.key = 1, u_3.key = 4, u_4.key = 2, u_5.key = 3$. See Figure 1.

(Q3) (9x4 Points)
Exercise III.5.2, p. 44.
Insertion and deletion on an external BST. The insertion/deletion algorithms for external BSTs in ¶III.40-42 (p. 41 ff). They are relatively straight forward. The insertion algorithm is used in our next question.

THE QUESTION (a) Show the result of inserting the following letters, `h, e, l, o, w, r, d` (in this order) into an external BST that initially contains the letter `*`. Assume the letters have the usual alphabetic sorting order but `*` is smaller than any other letter. Show the external BST after each insertion. No rebalancing is required.
(b) Starting with the final tree in part(a), now delete `w` and then delete `h`.

See Solution Figure.

(Q4) (20+12+4 Points
Exercise VI.2.23, p. 31-32.
Read up on External Splay trees (§VI.2.2, p. 26ff.). You will need to simulate the splay protocol (¶VI.17, p. 29). The key idea of the splay protocol is that *after each lookup, you must do a splay*! We state the question here because part(c) is not in your Lecture Notes.

THE QUESTION Consider the string

$$X = (\texttt{abcba})^m$$

where `a,b,c` are distinct ASCII characters and $m \geqslant 2$.
(a) How many bits have you transmitted after processing the first 5 characters (`abcba`) in string X? Show the external splay tree at this stage.
(b) What is the total number of bits transmitted for X? Recall that our splay protocol requires that, after inserting a new character `x`, you must first splay the parent of the `*` leaf, then splay the parent of the `x` leaf.
(c) How many bits have you saved by using our splay protocol on X?

SOLUTION FIGURE:

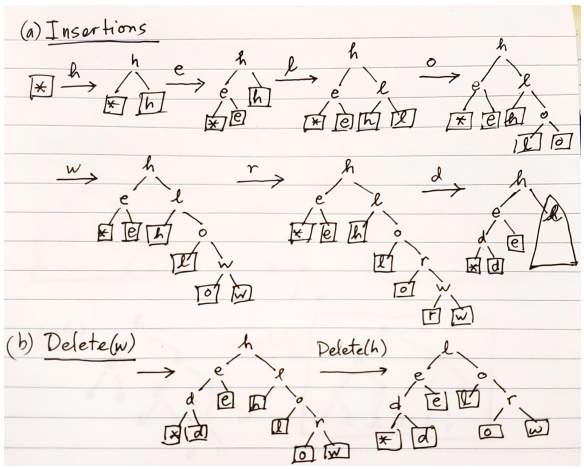

Figure 2: SOLUTION (a): Insert `h, e, l, o, w, r, d`. (b): Delete `w, h`.

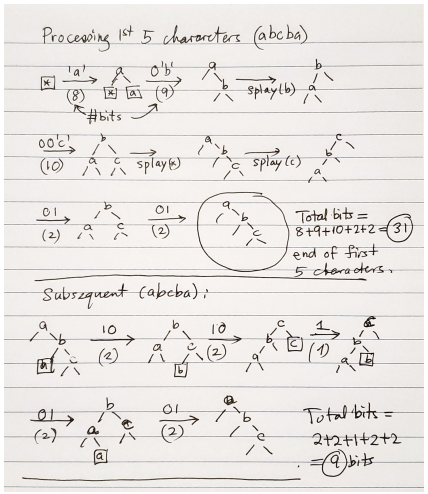

Figure 3: Splay compression of $X = (\texttt{abcba})^m$

SOLUTION: (a) The first 5 characters requires 31 bits as shown in Figure 3.
(b) Each subsequent 5 characters requires 9 bits. The splay tree at the end of processing each substring `abcba` ends up in exactly the same state as at the end of part (a). Hence the total number of bits is $31 + (m-1)9 = 22 + 9m$ bits.
(c) The ASCII code for `abcba` is 40 bits, so we use $40m$ bits to encode X. The savings is therefore $40m - (22 + 9m) = 31m - 22$ bits. There is a savings even for $m = 1$.

(Q5) (6+12+8 Points)
Exercise VII.2.2, p.19.
Computing $L(X, Y)$ and $LCS(X, Y)$ for DNA strings. Please do only (a-c), skip part(d).

THE QUESTION

(a) Let $X =$ `GACT'CGAA` and $Y =$ `TTCA'CGCA`. Please compute $L(X, Y)$ by filling in a matrix $M[0..8, 0..8]$. What is the value of $L(X, Y)$?

(b) In general, when $|X| = m, |Y| = n$, you can compute $LCS(X, Y)$ by constructing a directed acyclic graph (DAG) denoted $G_1(X, Y)$. The vertex set $V_1(X, Y)$ of $G_1(X, Y)$ is defined as the subset of $[0..m] \times [0..n]$ which contains a single source node (m, n) (called the **root**) and all those vertices (i, j) where $M[i, j] \geqslant 1$ and which are *reachable* from (m, n). (Cf. ¶VII.13, page 14)

(c) What is $LCS(X, Y)$ in part(a)?
Please draw $G_1(X, Y)$ in part(a). Describe how to you can extract $LCS(X, Y)$ from $G_1(X, Y)$ in a hand simulation. Actually, *you will discover the need for a slightly augmented version of* $G_1(X, Y)$, *which we may denote as* $G_1^+(X, Y)$.

(d) Give an algorithm $MultisetLCS(G, X, Y)$ which returns $LCS^+(X, Y)$ when called with $G = G_1(X, Y)$. Recall that $LCS^+(X, Y)$ is the multiset version of $LCS(X, Y)$ (Cf. ¶VII.7, page 10)
HINT: $MultisetLCS(G, X, Y)$ basically calls a **modified DFS** shell on the root $(m, n) \in V_1(X, Y)$. Let $mDFS(u)$ denote the modified DFS call on some vertex $u \in V_1(X, Y)$. The modification amounts to NOT coloring the vertices as `seen`/`unseen`/`done`, so that $mDFS(u)$ can visit a vertex multiple times! For each vertex $u \in V_1(X, Y)$, store a string $u.str$ that is equal to $str(p)$ for some path from the root to u.

(a) See SOLUTION in Figure 4

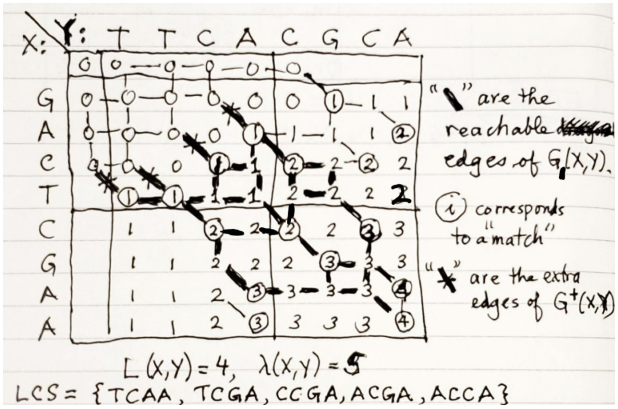

Figure 4: $(X, Y) =$ (`GACT'CGAA`, `TTCA'CGCA`) and the digraph $G_1(X, Y)$.

SOLUTION: (b) The graph $G_1(X,Y)$ for part(a) is shown in Figure 4. Note that $G_1(X,Y)$ has 3 types of edges:
(i) **Diagonal**: (i,j)–$(i-1,j-1)$ where $X[i] = Y[j]$. Call $X[i]$ the **matched letter** at this diagonal edge.
(ii) **Horizontal**: (i,j)–$(i-1,j)$ where $M[i,j] = M[i-1,j]$.
(iii) **Vertical**: (i,j)–$(i,j-1)$ where $M[i,j] = M[i,j-1]$.
For any unique path $p = (u_1, u_2, \ldots, u_\ell)$ in $G_1(X,Y)$, we can define the string $str(p)$ as follows: if there k diagonal edges along this path, say u_{i_j}–u_{i_j+1} $(j = 1, \ldots, k)$ is a diagonal edge for

$$1 \leqslant i_1 < i_2 < \cdots < i_k < \ell$$

then we define

$$str(p) = a_1 a_2 \cdots a_k$$

where a_j is the matched letter of the diagonal u_{i_j}–u_{i_j+1}.

In case u_ℓ is a sink of $G_1(X,Y)$, then there is also a diagonal edge u_ℓ–u' where $u' \notin V_1(X,Y)$. By adding u' to $V_1(X,Y)$, and adding the edges $u_\ell - u'$ to $G_1(X,Y)$, we obtain the needed **augmented graph** $G_1^+(X,Y)$. If the matched letter of u–u' is b, we can append b to $str(p)$ and denote this as $str^+(p)$ (call it the **augmented string** of p. Note that the augmented string is only defined when the last vertex of p is a sink of $G_1(X,Y)$. Then the reverse string $(str^+(p))^R$ belongs to $LCS(X,Y)$. Indeed $LCS(X,Y)$ is precisely the set $\left\{(str^+(p))^R : p \text{is a maximal length path of } G_1(X,Y)\right\}$.
Comments: It is possible that $val(p) = val(p')$ even though $p \neq p'$. Therefore, you will get a multiset unless you remove redundancies.

SOLUTION: (c): Using $G_1(X,Y)$, we can determine that

$$LCS(X,Y) = \{TCAA, TCGA, CCGA, ACGA, ACCA\}.$$

It turns out that $LCS^+(X,Y) = LCS(X,Y)$ in this case.

(d) Following the hint, we define $MultisetLCS(G)$ as follows:

```
MultisetLCS(G):
    LL ← ∅ where LL is a multiset of strings.
    (m,n).str ← ϵ is the empty string and (m,n) is the root.
    mDFS((m,n))
    Return LL    ◁ This is LCS+(X,Y)
```

It remains to program the macros of our $mDFS$ shell, specifically, PREVISIT and VISIT:

```
PREVISIT(v,u):
    v.str ← u.str
    If (u–v is a diagonal)
        v.str ← v.str; a where a = X[i] = Y[j] if u = (i,j).
```

```
VISIT(v):,
    If (v is a sink)    ◁ output a string
        Append the v.str+ to the multiset LL;
            here v.str+ denotes the augmented string
            corresponding to v.str (cf. str+(p) above)
```

(Q6) (12+4 Points)
Exercise VII.3.4, p.39.

Computing $A(X,Y)$ where $(X,Y) = (\texttt{google}, \texttt{yahoo})$. You must also construct the optimal alignment (X_*, Y_*) such that $A(X,Y) = \Delta(X_*, Y_*)$.

THE QUESTION Compute the alignment distance $A(X,Y)$ between $X = \texttt{google}$ and $Y = \texttt{yahoo}$ using the alignment cost (41) and (42) (in Example 7, ¶VII.32, p.37) For this purpose, assume `y` is a consonant. Also, express $\Delta(X,Y)$ as a direct alignment cost.

SOLUTION:

	ϵ	y	a	h	o	o
ϵ	0	3	6	9	12	15
g	3	1	4	7	10	13
o	6	4	2	5	7	10
o	9	7	5	4	5	7
g	12	10	8	6	6	7
l	15	13	11	9	8	8
e	18	16	14	12	10	9

Thus $\Delta(\texttt{google}, \texttt{yahoo}) = 9$. We reconstruct from this table the optimal alignment: $9 = \Delta_*(\texttt{google}, \texttt{ya*hoo})$.

(Q7) (12+8 Points)
Exercise VII.4.3 (page 47).
Optimal triangulation of a hexagon. Note that to obtain the optimal triangulation, you need to fill in another matrix for the splitters.

THE QUESTION Consider the optimal triangulation of the abstract hexagon using the weight function $W(i,j,k) = a_i^2 + a_j^2 + a_k^2$ where $(a_1, \ldots, a_6) = (4, 1, 3, 2, 2, 1)$.
(a) What is the optimal cost?
(b) What is the optimal triangulation?
Please show your working by filling in the following matrix. We fill the diagonal $C[i,i]$ entry with the value a_i^2 to help your computation. For part(b), You will need to fill in a corresponding splitter matrix.

	1	2	3	4	5	6
1	(16)	0				
2		(1)	0			
3			(9)	0		
4				(4)	0	
5					(4)	0
6						(1)

SOLUTION:
(a) The optimal cost is 47, as shown in the matrix below. The diagonal entries (circled grayed entries) contain the square of the parameter values: $a_1^2, \ldots, a_6^2$. This is helpful for filling in the matrix. We also circle those entries that are used in constructing the optimal final solution.

	1	2	3	4	5	6
1	(16)	(0)	26	35	44	(47)
2		(1)	0	(14)	(23)	(29)
3			(8)	0	17	23
4				(4)	(0)	9
5					(4)	(0)
6						(1)

(b) You need to fill in a similar matrix to keep track of the splitters. You see that at the optimal triangulation is $T = \{2\text{–}6, 2\text{–}5, 2\text{–}4\}$.

(Q8) (5+5+7+10+15 Points)

We generalize the activities selection problem of §V.3. Fix an integer $k \geqslant 1$. Let A be a non-empty set of intervals. We view an interval I as the time of a activity, and let $s(I), f(I)$ denote the start and finish time of the activity. Morever the interval has[1] the form $I = (s(I), f(I)]$. The **cover number** of A is $cover\#(A) := \max\{cover\#(A, x) : x \in \mathbb{R}\}$ where $cover\#(A, x)$ is the number of intervals in A that contains x. E.g., The cover number of $A = \{(0, 3], (2, 5], (3, 4]\}$ is 2. We say A is **k-feasible** if $cover\#(A) \leqslant k$. In the activities selection problem (§V.3), we called A is compatible if $cover\#(A) \leqslant 1$.

The **(Interval) k-Cover Problem** is this: *given A, compute a subset $C \subseteq A$ of maximal size which is k-feasible.*

Let us introduce two binary relations on intervals:
(1) Write $I \leqslant_f J$ if $f(I) \leqslant f(J)$.
(2) Write $I \prec J$ if $s(J) < f(I) < f(J)$. Note that (1) is a partial order, but (2) is not.

(a) If $cover\#(A) \leqslant 2$ and $I \in A$, then there is at most one $J \in A$ such that $I \prec J$.

SOLUTION: Assume I, J, K are 3 distinct intervals in A. If $I \prec J$ and $I \prec K$ then $s(J) < f(I) < f(J)$ and $s(K) < f(I) < f(K)$. If $x = \max\{s(J), s(K)\}$ then $x \in I \cap J \cap K$, contradicting $cover\#(A) \leqslant 2$.

(b) If $cover\#(A) \leqslant 2$, and I, J, K are 3 distinct intervals in A. If $I \prec K$ and $J \prec K$, then $I \cap J = \emptyset$.

SOLUTION: If $x \in I \cap J$, then $cover\#(A, x) \geqslant 3$, contradiction.

(c) $cover\#(A) \geqslant 3$ if and only if there are 3 distinct intervals $I, J, K \in A$ such that $I \cap J \neq \emptyset$ and $I \prec K$ and $J \prec K$.

SOLUTION: ($\Rightarrow$): assume $cover\#(A) \geqslant 3$. So there are $I, J, K \in A$ such that $I \cap J \cap K \neq \emptyset$. Let $x \in I \cap J \cap K$. Wlog, let $I \leqslant_f J \leqslant_f K$. This means that $s(K) \leqslant x$ Since $x < f(I)$ this means $s(K) < f(I)$, which proves $I \prec K$. Similarly, $s(K) < f(J)$ which proves $J \prec K$.
($\Leftarrow$): Suppose A contains I, J, K as described. Since $I \cap J \neq \emptyset$, we conclude that $I \cap J = (s^*, f^*]$ where

$$\max\{s(I), s(J)\} = s^* < f^* = \min\{f(I), f(J)\}. \tag{1}$$

The $I \prec K$ implies $s(K) < f(I) < f(K)$. Similarly $s(K) < f(J) < f(K)$. Thus $s(K) < f^* < f(K)$. It follows that $(s^*, f^*] \cap K$ is non-empty. I.e., $cover\#(A) \geqslant 3$.

(d) **Algorithm Overview**: we first sort the input intervals

$$I_1 \leqslant_f I_2 \leqslant_f \cdots \leqslant_f I_n.$$

[1] This trivial change from the original $(s(I), f(I)]$ allows us to talk of $cover\#(A, f(I))$ more meaningfully. FOR SIMPLICITY, we assume that for $f(I)$ is unique among the intervals in A. Why?

For each $i = 1, \ldots, n$, we either put I_i into $Bin(1)$ or into $Bin(2)$, or discard I_i. We maintain the invariant that $Bin(1)$ and $Bin(2)$ are each compatible sets. We finally output $Bin(1) \cup Bin(2)$. Let $idx \in \{1, 2\}$ be the "index" of the bin for inserting $I+i$: $Bin(idx).append(I_i)$. We call $Bin(idx)$ the **current bin** and $Bin(3 - idx)$ te **other bin**. Let $fTime(1)$ denote $\max\{f(I) : I \in Bin(1)\}$; and $fTime(2)$ is similarly defined.

2Cover$(A) \to B$
INPUT: A is a set of n intervals
OUTPUT: $B \subseteq A$ is an optimal set cover number $\leqslant 2$
Sort the n intervals in A as
$I_1 \leqslant_f I_2 \leqslant_f \cdots \leqslant_f I_n$.
Let $Bin(1), Bin(2)$ be sets of intervals, initially empty.
Let $fTime(1) \leftarrow fTime(2) \leftarrow 0$
Let $curr \leftarrow 1$ ◁ $curr \in \{1, 2\}$
Also $other := 3 - curr$
For $i = 1, \ldots, n$,
(A) If $s(I_i) \geqslant fTime(curr)$,
$Bin(curr).append(I_i)$
$fTime(curr) \leftarrow f(I_i)$
(B) Else if $(s(I_i) \geqslant fTime(other)$,
$curr \leftrightarrow other$ ◁ *flips the value of curr*
$Bin(curr).append(I_i)$
$fTime(curr) \leftarrow f(I_i)$
(C) Else
Discard I_i ◁ *i.e., do nothing*
Return $B = Bin(1) \cup Bin(2)$.

Please hand simulate the algorithm on the following input (already sorted):

$$A = \{I_1 : (0, 3], I_2 : (1, 4], I_3 : (4, 5], I_4 : (2, 6], I_5 : (5, 7], I_6 : (4, 8], I_7 : (7, 9].\}$$

SOLUTION: $Bin(1) = (I_1 : (0, 3], I_6 : (4, 8])$.
$Bin(2) = (I_2 : (1, 4], I_3 : (4, 5], I_5 : (5, 7], I_7 : (7, 9])$.
Discarded: $I_4 : (2, 6]$.

(e) Prove that this algorithm is correct.
HINT: This may be tricky – extra credit if you give a rigorous proof. Note that that each discarding of an interval is "justifiable" by part(c). Also note that the idx only flips when we insert the I_i to a bin different that I_{i-1}.
See Figure 5

SOLUTION: We don't expect you to write out a totally rigorous proof. Partial credits are given generously. Here is one outline of a rigorous proof:

(P1) If the input A is 2-feasible, then our algorithm will output A. This proof is given below.

(P2) Let $Greedy(A)$ be the size of the 2-feasible set that is computed by our algorithm.
THEOREM: If $A \subseteq A^*$ then $Greedy(A) \leqslant Greedy(A^*)$.
This proof can be reduced to the special case where $|A^*| = |A| + 1$. To analyze this, you need to give detailed analysis in which you compare the "state" $Greedy(A^*)$ and $Greedy(A)$. We can reduce these states to 7 possibilities as illustrated in Figure 5. Here a "state" is a pair of intervals, $(fTime(other) < fTime(this))$ for $Greedy(A^*)$, and the corresponding interval for $Greedy(A)$.

(P3) We conclude that $Greedy(A)$ gives the optimal value. If not, let $B \subseteq A$ be a 2-feasible set with $|B| > Greedy(A)$:

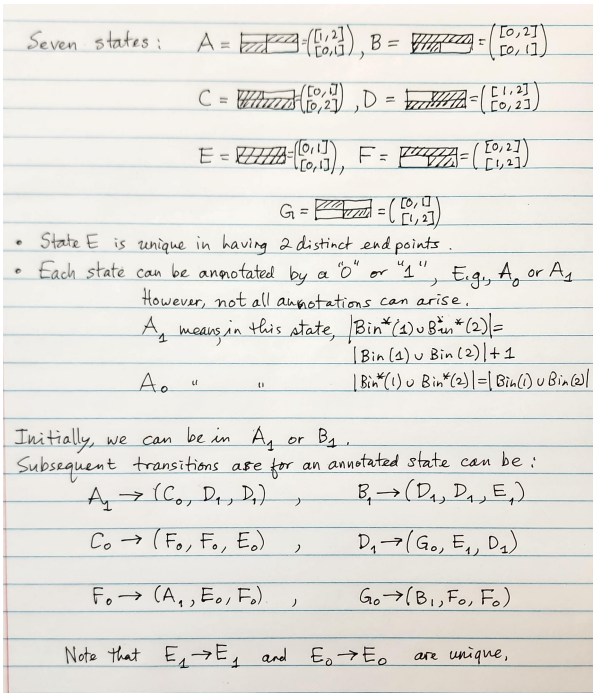

Figure 5: 7 States $A, B, \ldots, G$ and their transitions used in comparing $Greedy(A + I^*)$ and $Greedy(A)$.

(i) By (P2), we know that $Greedy(B) \leqslant Greedy(A)$.
(ii) By (P1), we know that $Greedy(B) = |B|$. This is a contradiction since we assumed $|B| > Greedy(A)$.

Proof of (P1): Our algorithm maintains this invariant: let the two set of intervals at stage i be $Bin(1)$ and $Bin(2)$, and $fTime(i)$ be finish time of $Bin(i)$ $(i = 1, 2)$. Let $other = 3 - curr$ where $curr \in \{1, 2\}$. We note that our algorithm maintains these two invariants.

(C1) $fTime(curr) > fTime(other)$ and
(C2) $cover\#(Bin(1 + 2), fTime(other)) = 2$

where $Bin(1+2) = Bin(1) \cup Bin(2)$. You now see why we changed $[s, f)$ to $(s, f]$: in the old notation, we cannot claim (C2). This invariant implies that when we discard an interval in case (C), we have discovered a situation indicated by part(c), which partly justifies the discarding. You can see this invariant holds right after we flip the index in case (B), and is maintained by cases (A) and (B).

Moreover, this implies that if $cover\#(A) \leqslant 2$, then we never discard any interval (the algorithm is therefore correct for such inputs).