Homework 4 Solutions
Fundamental Algorithms, Spring 2025, Professor Yap, Section Leader Dr. Bingwei Zhang

Due: Monday Mar 10, 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) (2x(8+8+8) Points)

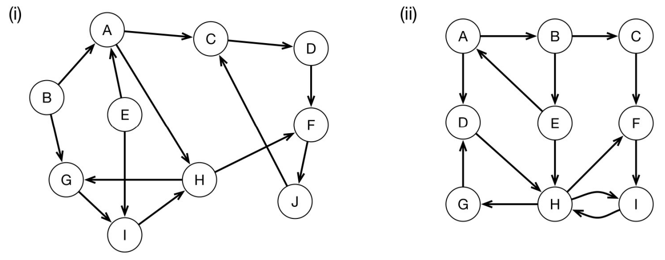


Figure 1: Graphs (i) G_{AJ} and (ii) G_{AI}

Consider the "DFS Pair"

$$\text{(TimeStamp DFS, DFS Driver)} \tag{1}$$

(cf. eq.(18) in ¶IV.38, p.36). Note that the **TimeStamp DFS** is described in ¶IV.42, p.40. Please simulate the execution of the DFS pair (1) on the input graph G (you must simulate the algorithm twice, for $G = G_{AJ}$ and for $G = G_{AI}$ in Figure 1).

The DFS Pair defines a DFS forest for the input G and computes a time span (`firstTime`$[u]$, `lastTime`$[u]$) for each vertex u of G. Please use the **canonical simulation**: this means that Whenever there is a choice of vertices, choose the one that is alphabetically first.

(a) Draw the DFS forest of each graph.

(b) Label each vertex u with its TimeStamp values (`firstTime`$[u]$, `lastTime`$[u]$).

(c) Using the time-stamp values, classify the edges of G.

SOLUTION in Figure 2

(Q2) (10+10+10+10 Points)
Exercise IV.6.5, p. 52.
Simulation of the SCC Algorithm on graph $G9$. Do parts (a) and (b) only (ignore part(c)).

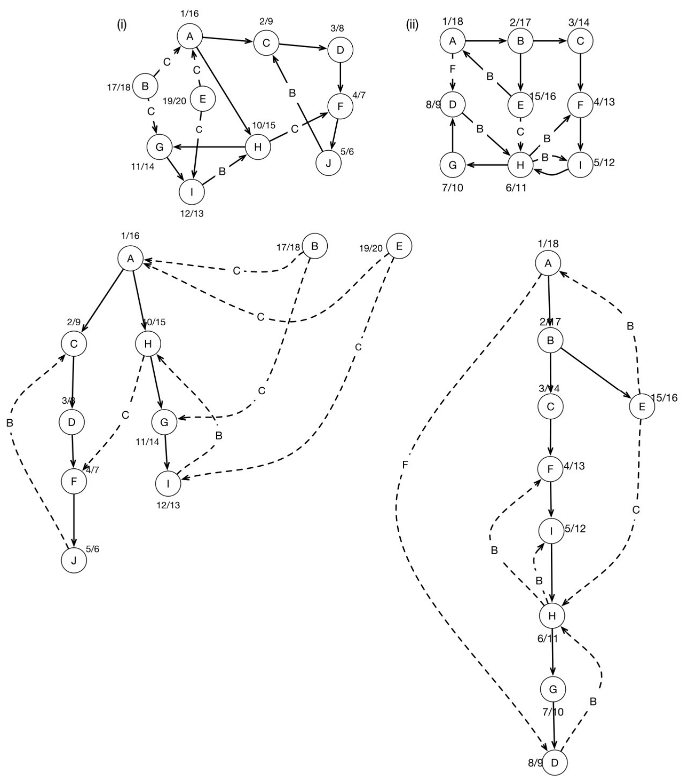


Figure 2: TimeStamp DFS Simulation on Graphs G_{AJ} and G_{AI}

THE QUESTION Hand simulation of the SCC algorithm on the graph $G9$ in Figure 3.

(a) Draw the reduced graph $G9^c$ of $G9$. Recall for any digraph G, its reduced graph is denoted G^c (see ¶14, p. 14).

(b) Run the Strong Component algorithm on $G9$. There is a unique output, based on our convention for choosing which node to visit next. Please show intermediate results:
(i) The reverse graph,
(ii) Output of Reckless Ranking algorithm,
(iii) The DFS forest (you may ignore forward and back edges, but don't omit the forest edges).

(c) Do shell programming to check in an input digraph is semi-connected.

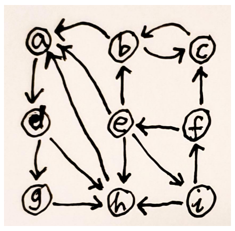


Figure 3: Digraph $G9$

(a) SOLUTION in Figure 5

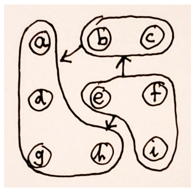
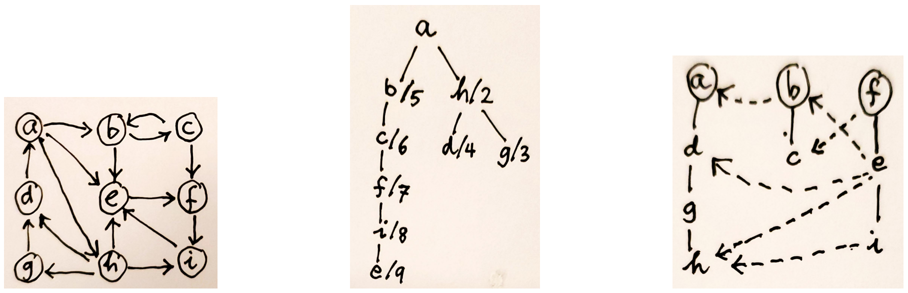


Figure 4: The reduced graph $G9^c$

Figures for Part(b) in Figure 5



(i) $G9^{rev}$ (ii) Ranking of nodes in DFS tree (iii) DFS forest

Figure 5:

SOLUTION: Running the Reckless Ranking Algorithm on the reverse graph $G9^{rev}$ in Figure 5(i) to get the DFS tree of Figure 5(ii) together a ranking of the nodes. The ranking gives us the order

$$(a, h, g; d, b, c; f, i, e).$$

Then call the "Strong Component Driver" on $G9$ using this ranking, to get the DFS forest Figure 5(iii) Each tree in this forest corresponds to a strong component. Forest edges indicate the edges of the reduced graph $G9^c$. This should agree with your picture of part(a).

(Q3) (12+12 Points)
(V. Shoup) Let G be a digraph where every node u has a *color*$[u]$, which is either black or white. Let k be a positive integer. A path in G is called **k-alternating** if it changes color at least k times. Note that the path need not be simple (i.e., it may contain repeated nodes). For example, in Figure 6, the path $A{-}C{-}B{-}D{-}E$ is a 3-alternating path: $C{-}B$ is a white to black transition, $B{-}D$ is black to white, and $D{-}E$ is white to black.

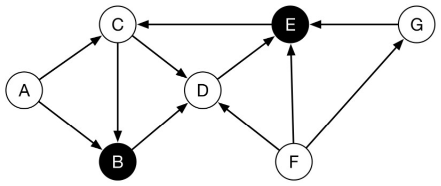


Figure 6: Graph for alternating path

YOU MUST USE SHELL PROGRAMMING IN THE FOLLOWING 2 ALGORITHMS.

(a) Design an algorithm that determines if an input DAG has a k-alternating path. Your algorithm takes as input a digraph $G = (V, E; \mathit{color})$ (in adjacency list representation) and a positive integer k. It outputs true or false. Your algorithm should run in time $O(|V|+|E|)$. HINT: start by running a topological sorting algorithm, and then compute for each vertex u, the value $m[u]$, which is the largest k such that there is a k-alternating path starting at u.

SOLUTION: Following the hint, our goal is to compute the array $m[u \in V]$. We have the following recursive formula for $m[u]$:

$$m[u] \quad = \quad \begin{cases} 0 & \text{if } u \text{ is a sink,} \\ \max\{m[v] + d(u,v) : u\text{–}v \in E\} & \text{else.} \end{cases}$$

where

$$d(u,v) = \begin{cases} 1 & \text{if } \mathit{color}[u] \neq \mathit{color}[v], \\ 0 & \text{else.} \end{cases}$$

This suggests that we compute $m[u]$ in a reverse topological order:

```
Let $(v_1, \ldots, v_n)$ be the topological sort of $G$.
Initialize the array $m[1..n]$ to 0.
for $i = n$ down to 1 (in reverse order!)
    for each $j$ adjacent to $i$,
        $m[i] \leftarrow \max\{m[i], m[j] + d(i,j)\}$
    if $m[i] \geqslant k$, return true.
Return false.
```

BUT this is NOT an acceptable solution because I asked you to use shell programming!

Actually, to compute $m[u]$ in reverse topological order is great for us! That is because, recall that in the topological sort of G, we assign the rank to $v \in V$ in *reverse order*, i.e., we find rank n first, then rank $n-1$, and finally rank 1. Recall that the rank of v rank is assigned only during its POSTVISIT. So we only need program some macros: besides the global array $m[V]$, we also assume a global boolean variable `ReturnValue` which is the output value.

```
$\boxed{\text{DRIVER\_INIT}(G^{rev})}$:
        Initialize each $m[v] = 0$ for $v \in V$.
        ReturnValue $\leftarrow$ false
```

```
$\boxed{\text{POSTVISIT}(u)}$:
        for each $v$ adjacent to $u$,
            $m[u] \leftarrow \max\{m[u], m[v] + d(u,v)\}$.
        if $m[u] \geqslant k$, ReturnValue $\leftarrow$ true
```

Comments: Note that in the non-shell program, we can return false as soon as we discover some $m[u] \geqslant k$. But we cannot "return" in shell programming!

(b) Now generalize your algorithm to solve this problem on an arbitrary graph. HINT: start by running an Strong Component algorithm, and use the algorithm from part (a) as a subroutine.

SOLUTION: Let $G = (V, E)$ be digraph, and $G^c = (V^c, E^c)$ its reduced graph. So each component $C \subseteq V$ of G is a vertex in V^c. Note that if C contains both black and white vertices, then G contains a k-alternating path. Otherwise, we can color each component of G with the color of any (and all) vertices in the components. It follows that G^c is DAG whose vertices are colored either white or black. Moreover, G has a k-alternating path iff G^c has a k-alternative path. The pseudo-code

```
Input: digraph $G = (V, E; color)$ and integer $k \geqslant 1$.
Returns true iff $G$ has a $k$-alternating path.
    1. Run the Strong Component on $G$.
        The output is a DFS forest represented by the parent array $p[1..n]$.
    2. Check if any DFS tree has an edge whose endpoints have different color.
        for each $i = 1, \ldots, n$,
            if $color[i] \neq color[p[i]]$, return true.
    3. Compute the adjacency list representation of $G^c$ with the unique
        color for each component.
    4. Run the algorithm of part(a) on $G^c$ and return its ReturnValue.
```

(Q4) (12 Points)
Exercise V.1.6, p. 14.
Improving on the Karp-Held trick for bruteforce search.

THE QUESTION Improve the bin packing upper bound in Lemma 2 (Karp-Held) in p. 6, from $O((n/e)^{n+(1/2)})$ to to $O((n/e)^{n-(1/2)})$.
HINT: Repeat the trick which saved us a factor of n in the first place. Fix two weights w_1, w_2. Consider two cases: either w_1, w_2 belong to the same bin or they do not.

SOLUTION: If π is an n-permutation, and $w = (w_1, \ldots, w_n)$, then

$$\pi(w) := (w_{\pi(1)}, \ldots, w_{\pi(n)}).$$

As usual, S_n denote the set of all n-permutations. Also let $w' = (w_3, \ldots, w_n)$ be the result of omitting w_1 and w_2. Fix any optimal solution $(B_1, \ldots, B_{Opt(w)})$. There are two cases: the weights w_1 and w_2 either both belong to the same bin in this solution, or they do not.
(CASE 1: Same Bin) In this case, we generate all $(n-2)!$ permutations of $w_3, \ldots, w_n$, and for each of these permutations, we append w_1, w_2 and we solve linear bin packing problem.
(CASE 2: Different Bins) In this case, we take each of the $(n-2)!$ permutations, and prepend w_1 at the front, and append w_2 at the back of the permutation. Then we solve linear bin-packing on this instance.

Taking the best among these $2 \times (n-2)!$ solutions, we obtain the optimum. Thus the complexity of this solution is $O(n(n-2)!) = O((n-1)!)$ time.
Comments: Food for thought: can we try to improve another factor of n? We would have to fix w_1, w_2, w_3. In case w_1, w_2, w_3 land in different bins in an optimal solution, it is not so clear we can find this solution by trying the $(n-3)!$ permutations of $w_4, \ldots, w_n$.

(Q5) (2x(8+8+8) Points)
Exercise V.3.1, p. 24.
Showing "strong" counter examples to some greedy criteria. The text explains what we mean by "strong" counter examples. [1]

There are two parts: parts(i) and (ii). In part(ii), you only have to find strong counter examples to 3 of the 4 criteria. **You do not have to prove the optimality of the remaining criteria.**

THE QUESTION The text gave four different greedy criteria (a)-(d) for the activities selection problem.
(i) Show that (b), (c), (d) are suboptimal using "strong" counter examples (we prefer visualized intervals). Extra credit if (b), (c), (d) are shown with the *same* set of activities. How small can this set of activities be?
(i) Each of the criteria (a')-(d') have an inverted version in which we sort in decreasing order. Again, one of these inverted criteria is optimal, and the other three suboptimal. Prove the optimality of one, and provide counter examples for the other three.

[1]Please do not report the obvious typo of "(i)" being used instead of "(ii)".

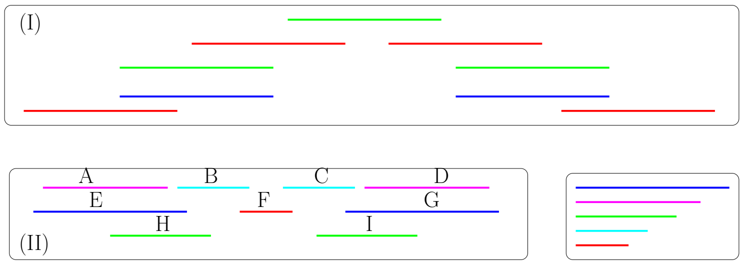


Figure 7: Counter Examples for criteria (b), (c), (d)

SOLUTION: (i) We proved in the text that (a) [increasing stop times] is optimal. Our introductory example shows that (b) [increasing start times] and (c) [increasing duration] are suboptimal.
(a-iv) Increasing conflict degree. A weak counter example is shown in Figure 7(I): using our criterion, we would pick the middle interval, and then we can pick at most two more (e.g., the green intervals). But the optimal solution is the four red intervals. Note that this is a weak counter example. But if you duplicate two of the blue intervals, you can get a strong counter example.

THE EXTRA CREDIT: Thanks to Jinil Jang for this example. Consider the 9 intervals (A,B,C,D,E,F,G,H,I) in Figure 7(II): the optimal solution has 4 intervals (A,B,C,D). But using any of the other three criteria, you only get three intervals. For (b), sorting by increasing start times, you will pick E. This allows you to choose at most two more intervals. For (c), sorting by increasing duration times, you will choose F. That implies you can choose at most two more intervals on either side of F. For (d), the degree of conflict criterion, you will again choose interval F.
(ii) The optimal criteria is (a'), sorting the start times in decreasing order. To prove its optimality, we follow the proof for the optimality of (a) in the text. Suppose the greedy solution using decreasing start times finds k compatible activities, and these are their start times:

$$s_1 \geqslant s_2 \geqslant \cdots \geqslant s_k$$

Suppose the optimal solution found ℓ compatible activities with start times

$$s'_1 \geqslant s'_2 \geqslant \cdots \geqslant s'_\ell.$$

It is easy to prove by induction that $s_i \geqslant s'_i$ for each $i = 1, \ldots, \ell$. Now suppose the greedy method is suboptimal. Then $k < \ell$. Then look at s'_{k+1}. We have $s_k \geqslant s'_k \geqslant s'_{k+1}$. Clearly, the activity represented by s'_{k+1} is compatible with the greedy solution. That means the greedy algorithm would have chosen this interval as well, contradiction.

The other three are suboptimal. Counter examples are very easy to come by, so we omit it.

(Q6) (2x(14+8) Points)
This is Exercise V.4.23 (p.40). We have slightly re-written the question here:

THE QUESTION Consider the strings:
(a) $s =$`hello␣world!`.
(b) $s =$`hi!␣my␣little␣world!`.

We want to the optimal encoding of these strings using "trits" instead of bits. A **trit** is a symbol in an alphabet Σ with just 3 symbols. We may assume $\Sigma = \{1, 0, \overline{1}\}$. You can think of $\overline{1}$ as -1, so the 3 values are $0, \pm 1$.

The optimal ternary Huffman code C for the string s can be constructed using the algorithm in the previous exercise (Exercise V.4.22). Draw the optimal ternary Huffman code C for strings s in parts (a) and (b) above. In each case, also determine the length of the ternary encoding, $|C(s)|$.

Solution in Figure 8

(Q7) (22 Points)

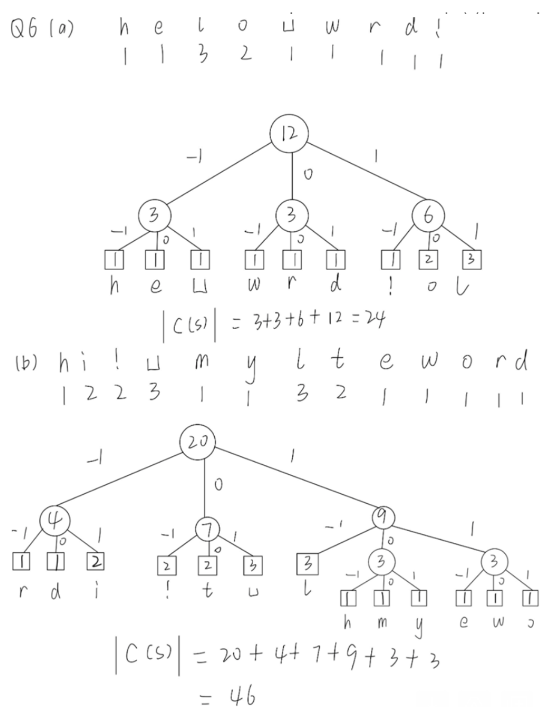


Figure 8: Solution from Huang Shuai

Exercise V.4.19, p. 39.
The Huffman tree algorithm takes $O(n \log n)$ because it uses a priority queue. In this question, we show that $O(n)$ time suffices when the input frequencies are already sorted.

HINT: first describe your idea and/or data structure. Then present the pseudo-code, briefly arguing why it is $O(n)$ time.

SOLUTION: We start out with a sorted list

$$F : f_1 \leqslant f_2 \leqslant \cdots \leqslant f_n$$

of frequencies. How can we avoid the $O(\log n)$ cost to extract the minimum element from the current list of Huffman trees? In particular, we would like to extract the minimum element in $O(1)$ time. The idea is to keep track of another sorted sequence

$$G : g_1 \leqslant g_2 \leqslant \cdots \leqslant g_m,$$

where $m \geqslant 0$ comprising all the frequencies of trees that we have merged. In the general step, suppose we have

$$F : f_i \leqslant f_{i+1} \leqslant \cdots \leqslant f_n$$

and

$$G : g_j \leqslant g_{j+1} \leqslant \cdots \leqslant g_m$$

for some $i \leqslant n+1$ and $j \leqslant m+1$. Note that $i = n+1$ means F is empty, and $j = m+1$ means G is empty. At most one of these two lists are non-empty. Thus, we maintain three variables: i, j, m.

HERE IS THE ALGORITHM:
Initially $m = 0$, $i = 1$ and $j = 1$. Call m the "step number". In the mth step, we want to compute the value g_{m+1}. E.g., when $m = 0$, we want to compute g_1. The mth step has 3 operations:

(I) Extract the minimum a of all the current Huffman frequencies from $F \cup G$: we just choose

$$\min\{f_i, g_j\},$$

and delete this minimum from the appropriate list. Note that if $i = n+1$, then we treat f_{n+1} as ∞; and if $j = m+1$, then we treat g_{m+1} as ∞. If $f_i = \min\{f_i, g_j\}$ is removed from F, we update $i \leftarrow i+1$; otherwise, $g_j = \min\{f_i, g_j\}$ is removed from G, and we update $j \leftarrow j+1$.

(II) Extract another minimum b from $F \cup G$.

(III) Now we have extracted two successive minimas a and b from $F \cup G$. We merge them into a new Huffman tree whose frequency is $a+b$, and define $g_{m+1} \leftarrow a+b$. We then insert g_{m+1} into the end of the list of G. We also update the variable $m \leftarrow m+1$.

Thus is it clear that we can do these 3 operations in $O(1)$. The algorithm halts when F is empty (i.e., $i = n+1$) and G has one element (i.e., $j = m$). Moreover, $m = n-1$. So algorithm clearly has complexity $O(n)$.
But why is this algorithm correct? The correctness of this algorithm algorithm follows from the correctness of the usual Huffman tree algorithm. The only thing we have to prove is that we only need to prove that $g_{m+1} \geqslant g_m$ so that the list G remains sorted. Suppose $g_m = a+b$ and $g_{m+1} = a'+b'$. Then we see that $\max\{a,b\} \leqslant \min\{a',b'\}$. This proves that $g_m \leqslant g_{m+1}$.

(Q8) (2x16 Points)

Exercise V.6.8, p.68.
Hand simulation of Kruskal's and Prim's algorithm on graph G_7. Note that both algorithms follow the Generic Greedy MST Algorithm (p. 62).
(a) For Prim's algorithm, please use the tabular method described in detail in ¶V.37 (p. 60).
(b) Kruskal's algorithm is based on the Kruskal's criteria described in ¶V.39 (p. 64). To hand simulate it, you just sort the edges of the graph in non-decreasing order, say $e_1, e_2, \ldots, e_m$. For each e_i, you either accept or reject it.

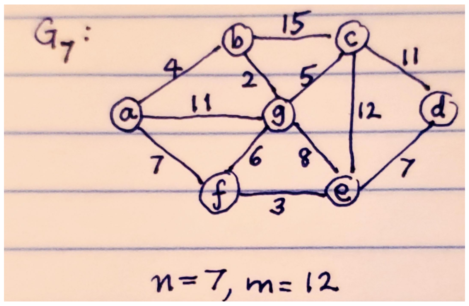


Figure 9: Graph G_7

THE QUESTION Consider the weighted bigraph G_7 in Figure 9.

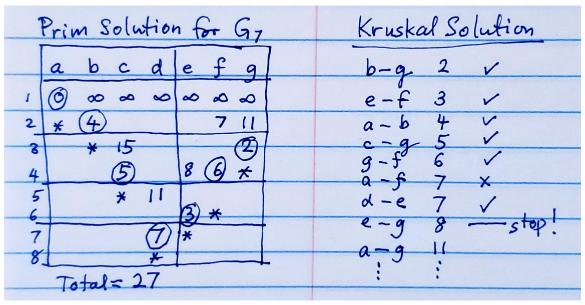
(a) Please compute the MST of G_7 using Kruskal's algorithm. What is the cost of the MST? Also list the edges of the MST.

(b) Please compute the MST of G_7 using Prim's algorithm. You must list the edges of the MST.

SOLUTION: See Figure 10 for the simulation of Prim's and Kruskal's algorithm.
(a) The cost of the MST is 27.
(b) The MST edges are

$$a\text{–}b, \quad b\text{–}g, \quad g\text{–}c, \quad , g\text{–}f, \quad e\text{–}f, \quad , d\text{–}e.$$

SOLUTION in Figure 10



Prim Solution for G_7

	a	b	c	d	e	f	g
1	⓪	∞	∞	∞	∞	∞	∞
2	*	④				7	11
3		*	15				②
4			⑤		8	⑥	*
5			*	11			
6					③	*	
7				⑦	*		
8				*			

Total= 27

Kruskal Solution

b–g	2	✓
e–f	3	✓
a–b	4	✓
c–g	5	✓
g–f	6	✓
a–f	7	×
d–e	7	✓
e–g	8	—— stop!
a–g	11	
⋮	⋮	

Figure 10: Simulation of Prim's and Kruskal's Algorithm on G_7