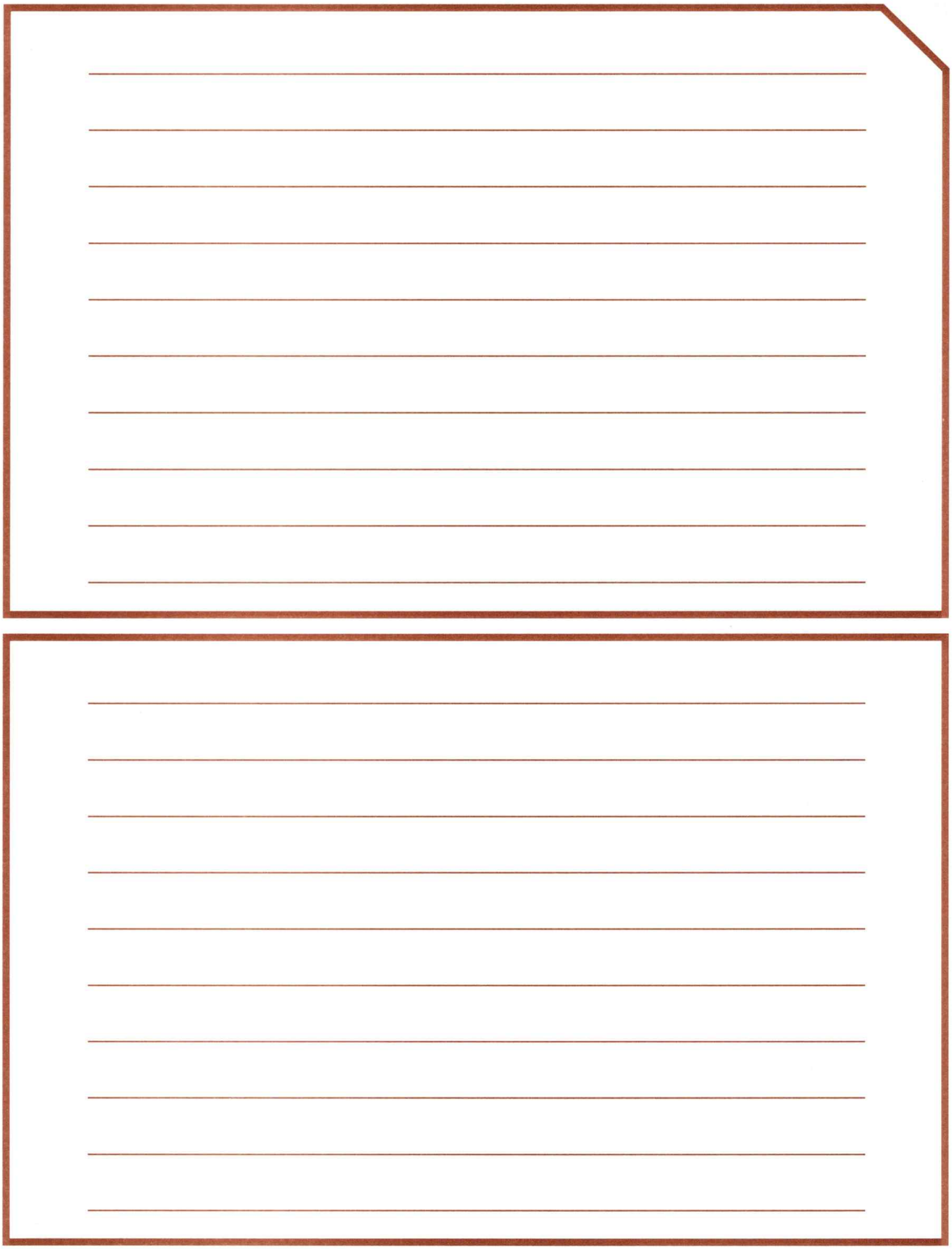
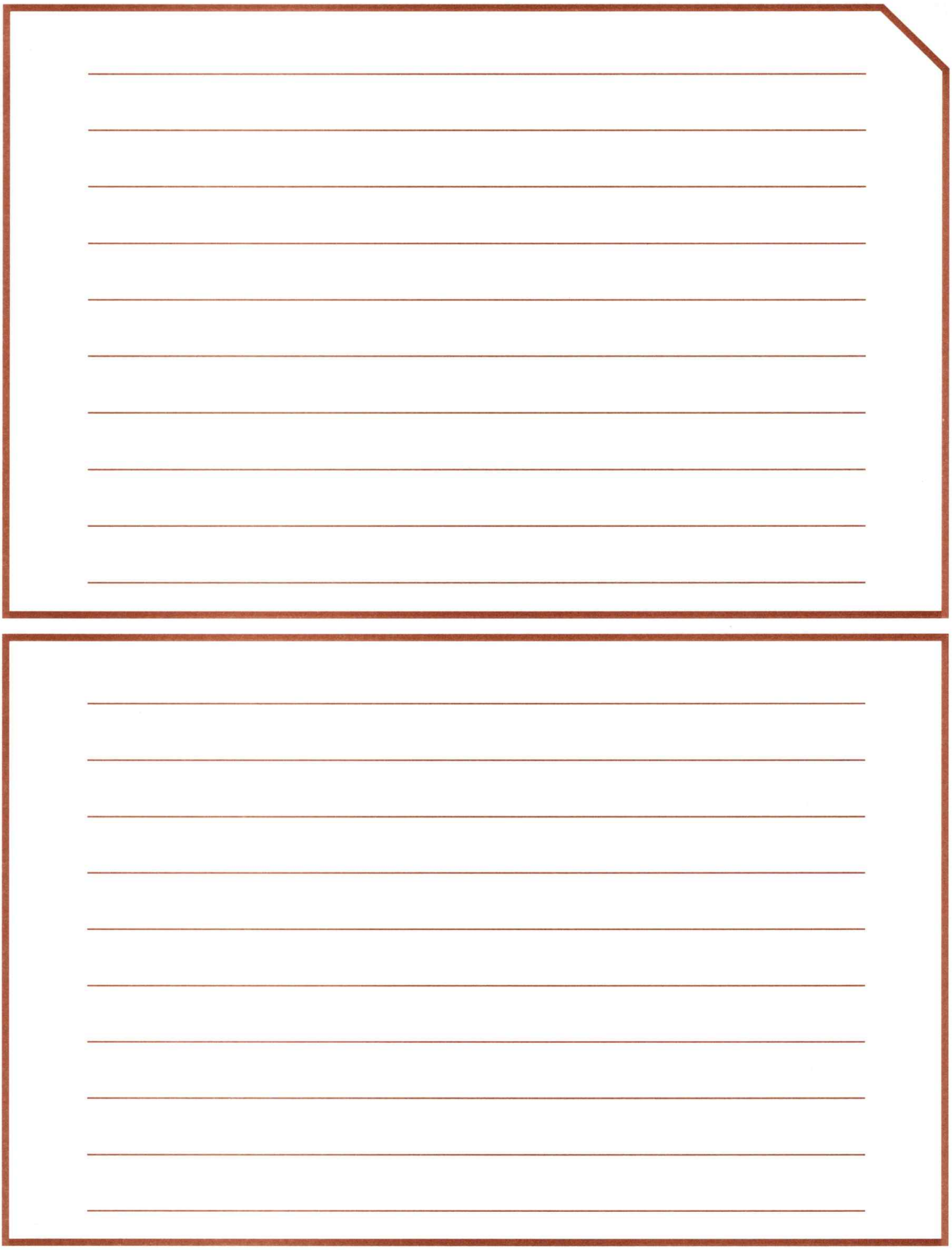


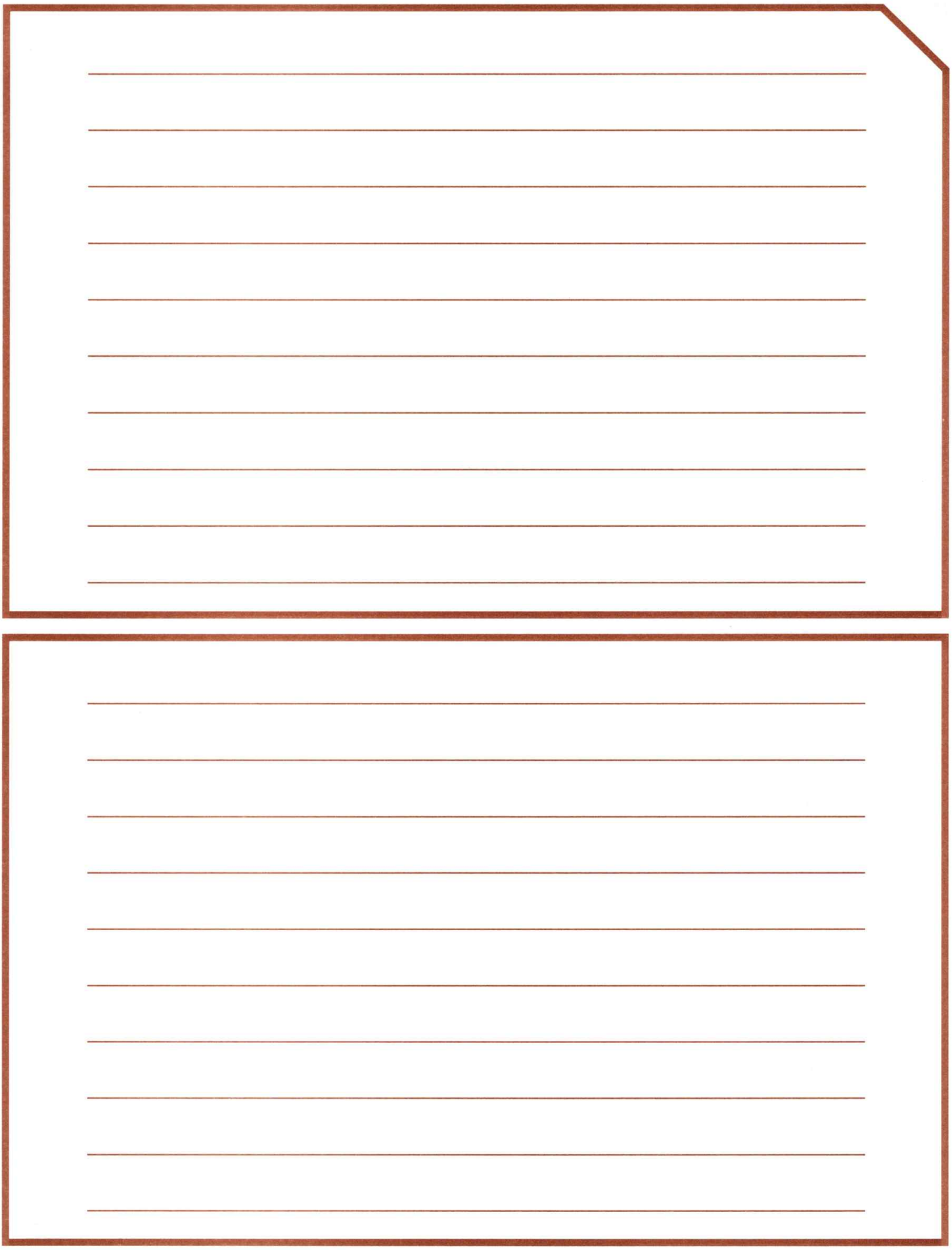
Interval scheduling Problem

Input: Set of requests $\{1..n\}$

i^{th} request starts at $s(i)$ and ends at $f(i)$

Objective: To find the largest compatible subset of these requests





Solution:

Initially R is the complete set of requests
& A is empty
While R is <u>not</u> empty
Choose a request $i \in R$ that has the smallest finish time
Add request i to A
Delete all requests from R that are not compatible w/ i
endwhile

Return A

Proof of Correctness

1. Show that A is a compatible set
2. Show that A is an optimal set

Say A is of size k
Say there is an opt. solution O

Requests in A: $i_1 \ldots\ldots i_k$
" " O: $j_1 \ldots\ldots j_m$

We will first prove that for all indices $r \leq k$, we have $f(i_r) \leq f(j_r)$

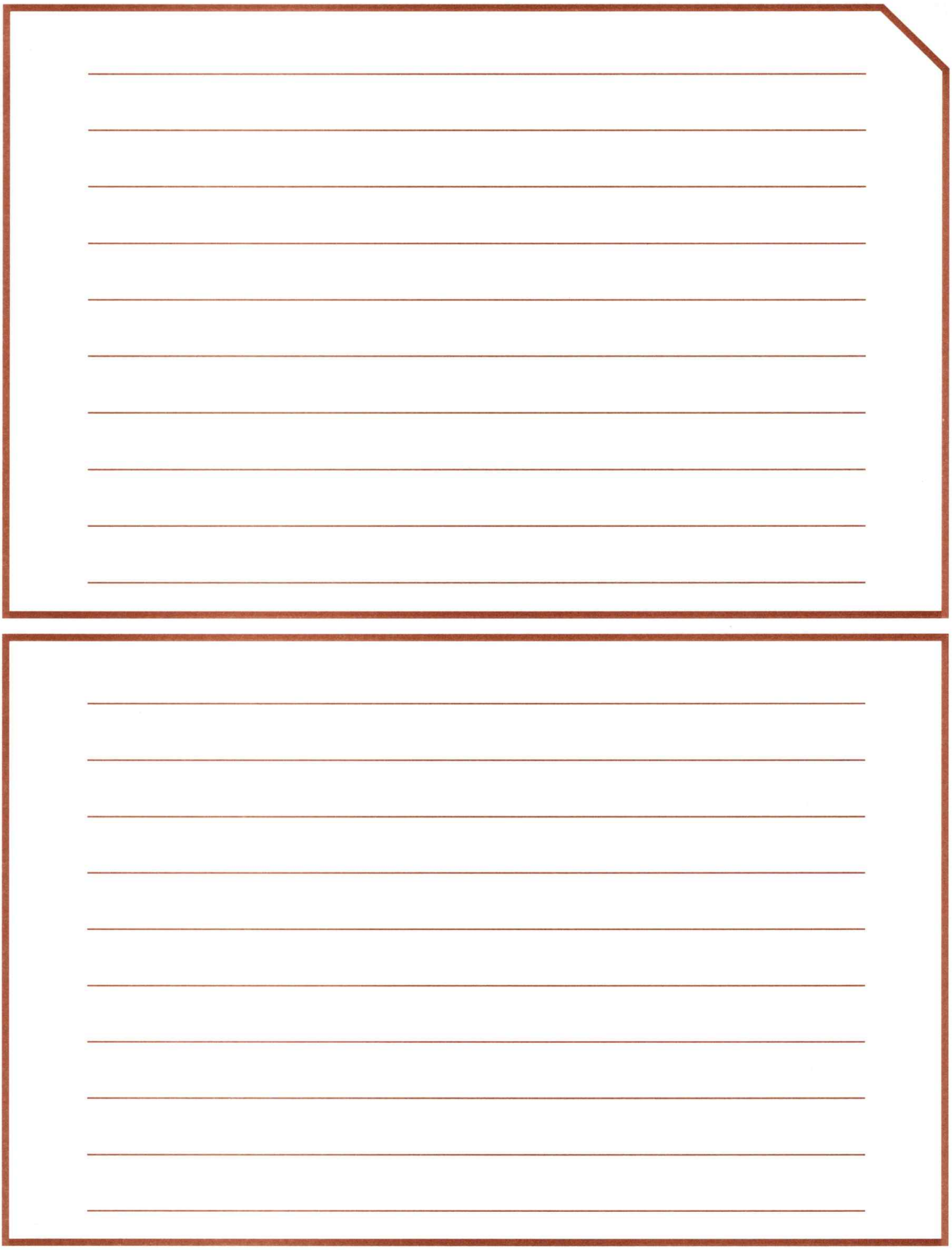
We can then easily prove that $|A| = |O|$

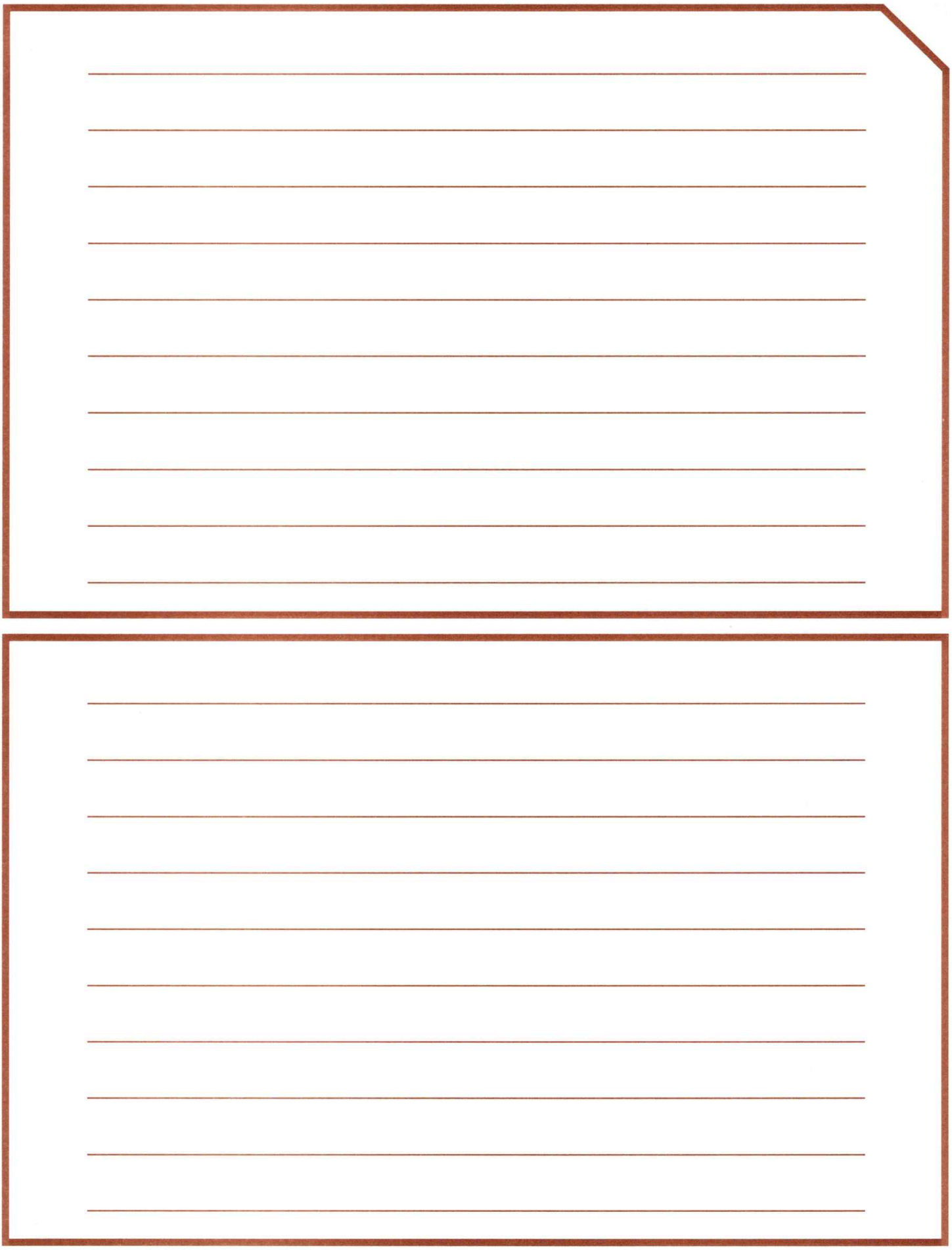
Implementation

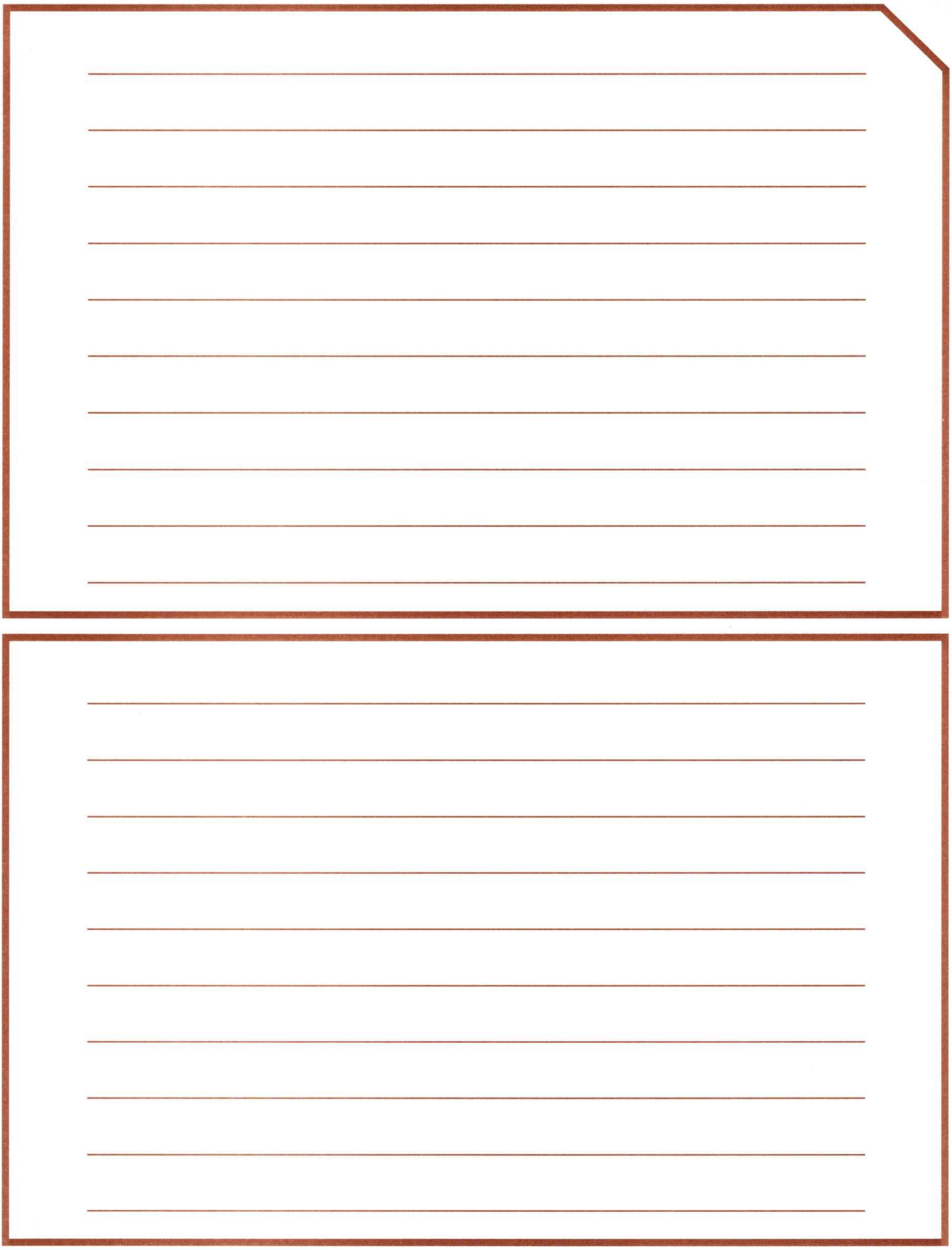
- Sort requests in order of finish time and label in this order:

$$f(i) \leq f(j) \quad \text{where } i < j$$

- Select requests in order of increasing $f(i)$, always selecting the first. Then iterate through the intervals in this order until reaching the first interval for which $s(j) \geq f(i)$







Fractional Knapsack

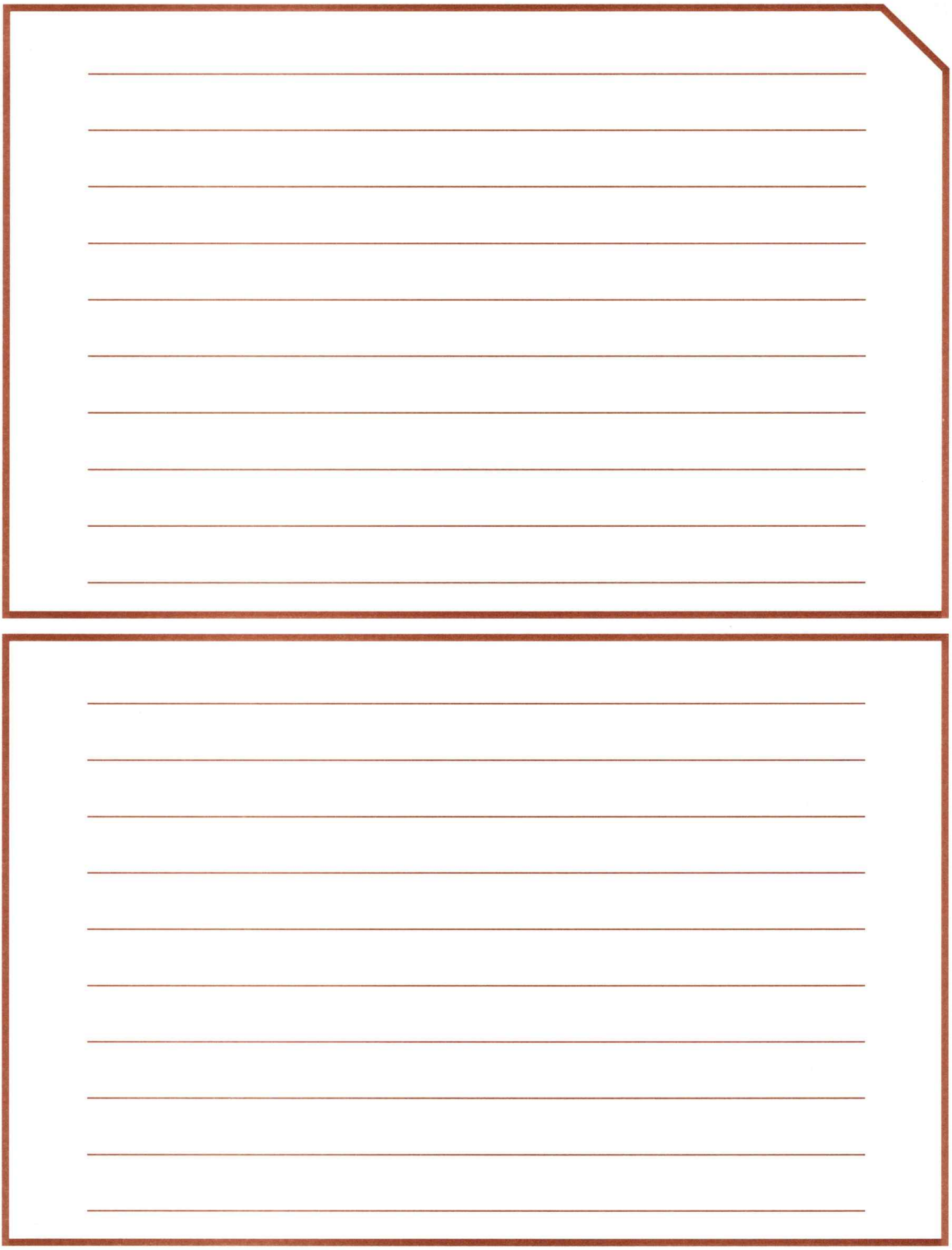
Knapsack has a weight capacity of $\underline{W}$.

We are given as input, a set of $\underline{n}$ objects with weight w_i and value v_i.

Objective: Fill up the knapsack to its weight capacity such that the value of items in knapsack is maximized.

Ex. knapsack weight cap: $\underline{10}$

items	1	2	3	4	5
Values	10	20	15	2	8
weights	4	10	5	1	2



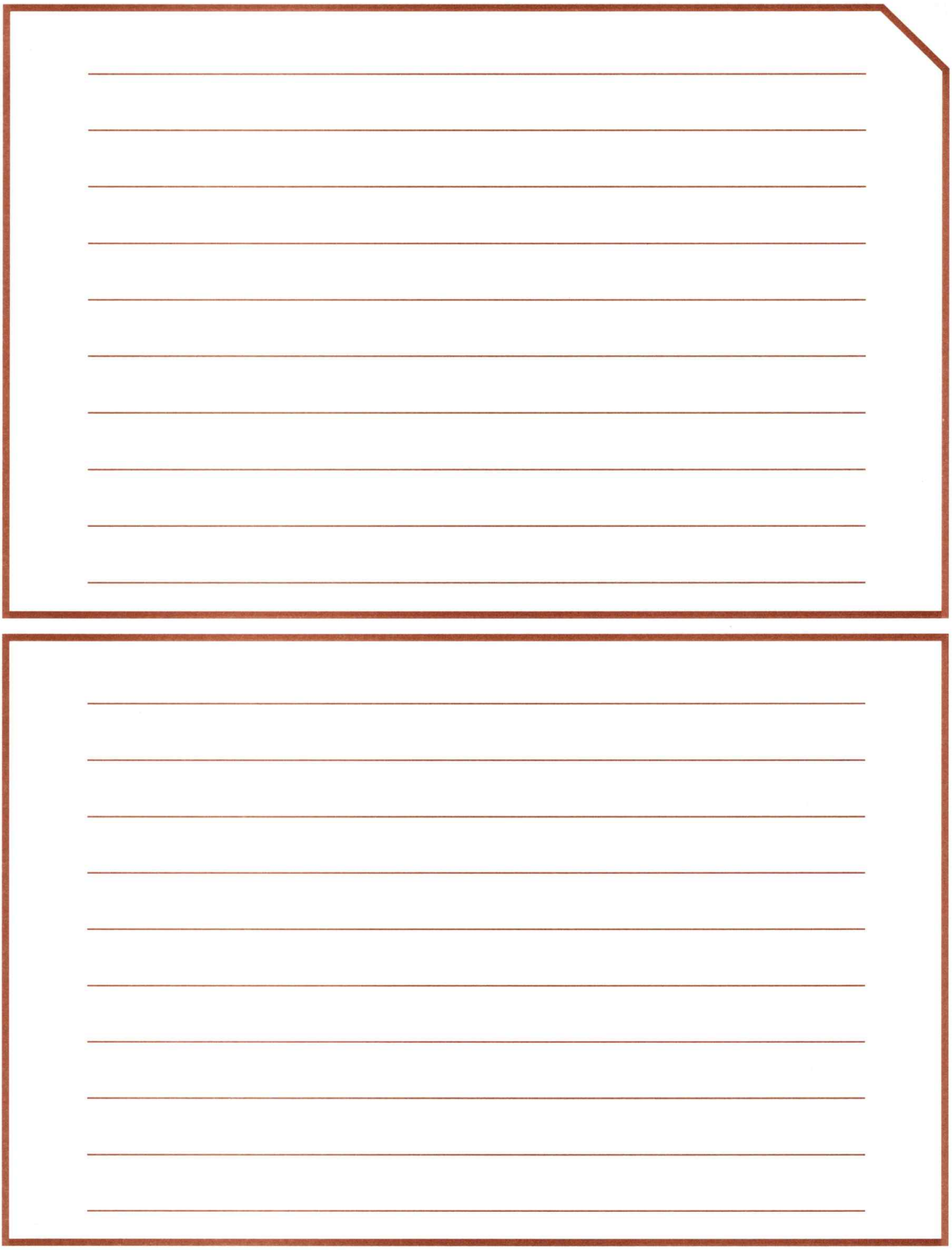
Scheduling to Minimize Lateness

Scheduling to Minimize Lateness

- Requests can be scheduled at any time
- Each request has a deadline
- Notation: $L_i = f(i) - d_i$

  L_i is called lateness for request i.

Goal: Minimize the Maximum Lateness $L = \max_i L_i$



Solution:

Schedule jobs in order of their deadline without any gaps between jobs.

Proof of Correctness

① There is an optimal solution with no gaps.

(2) Jobs with identical deadlines can be scheduled in any order without affecting Maximum Lateness.

(3) <u>Def.</u> Schedule A' has an inversion if a job i with deadline d_i is scheduled before job j with earlier deadline.

(4) All schedules with no inversions and no idle time have the same Maximum Lateness.

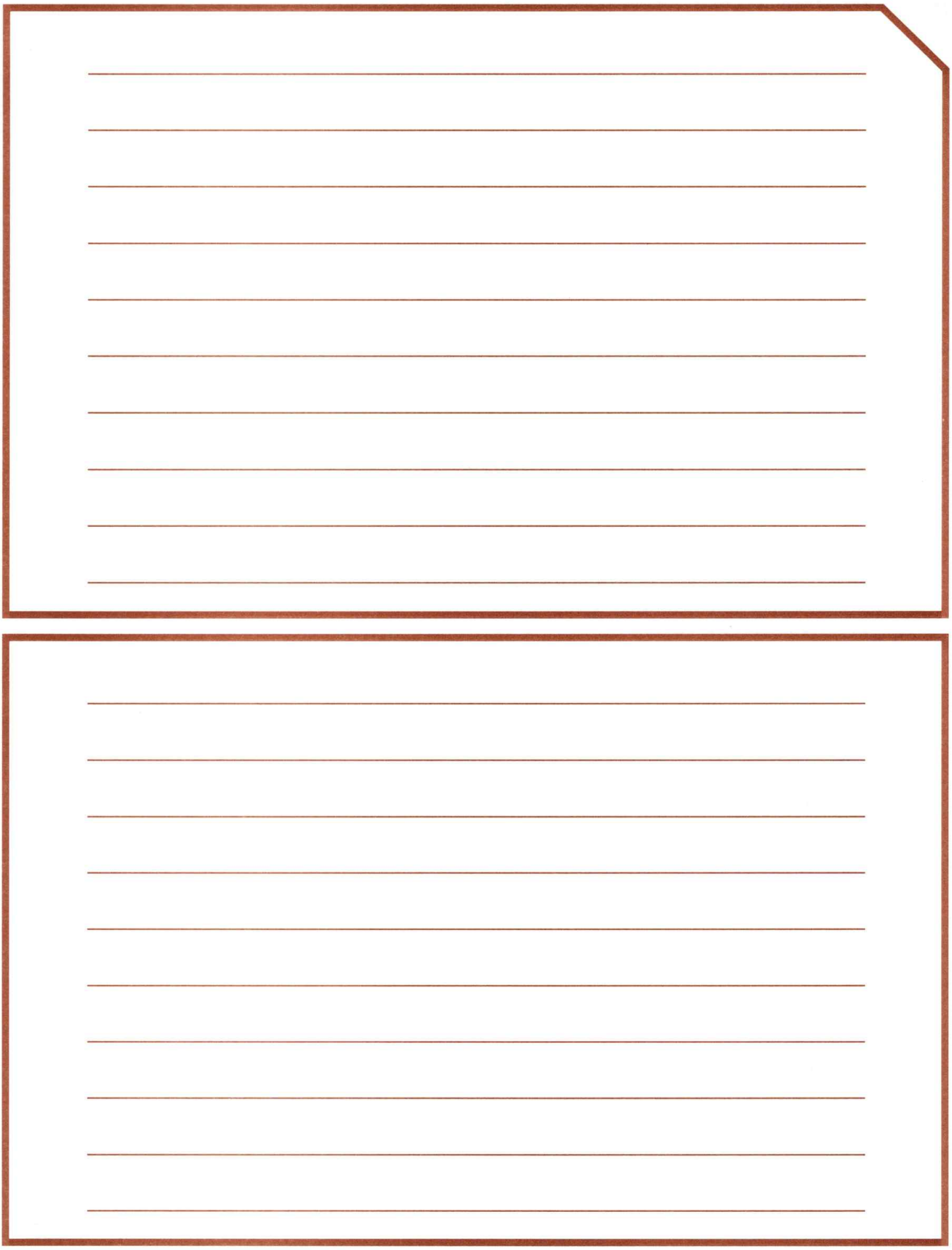
(5) There is an optimal schedule that has no inversions and no idle time.

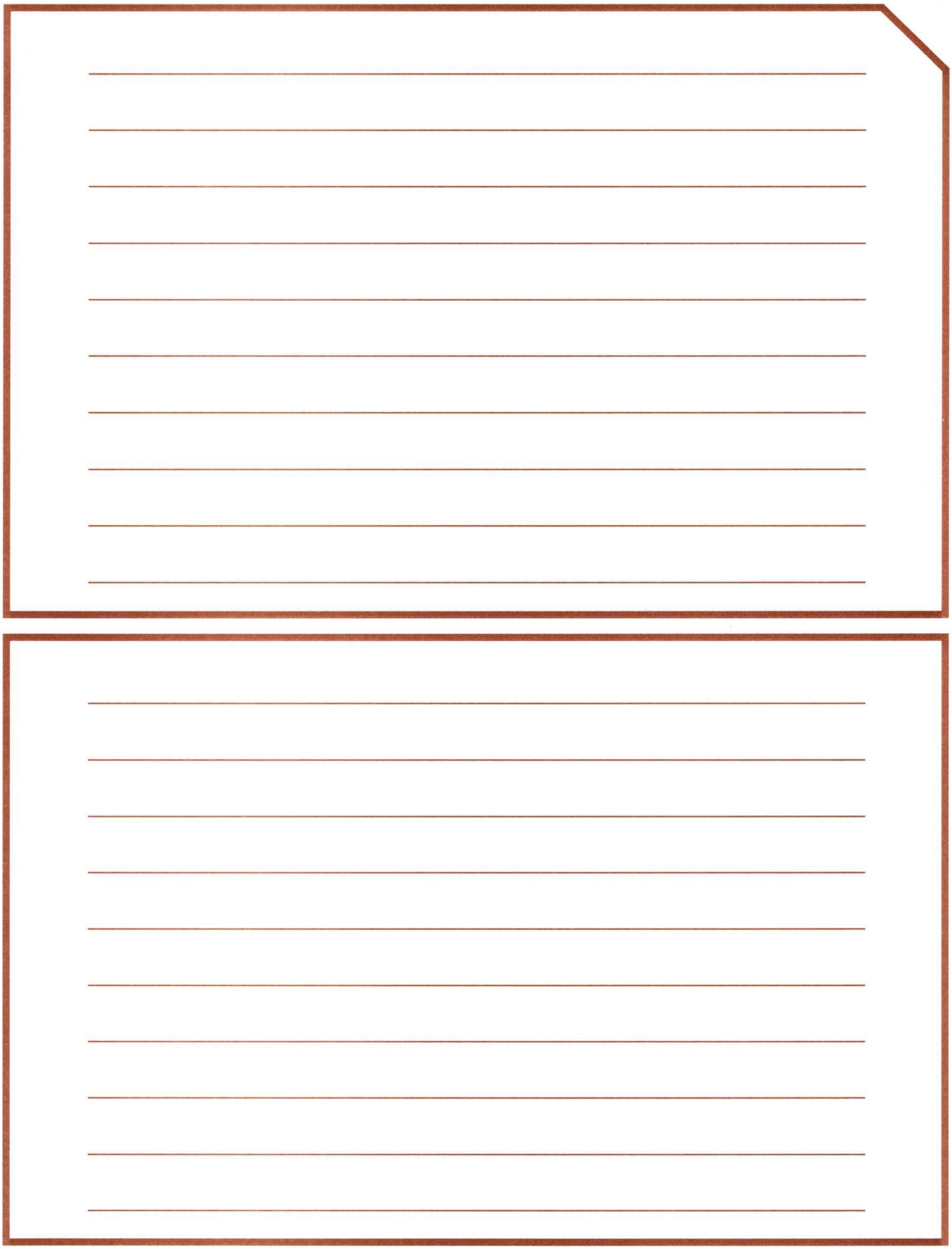
So, if there is an optimal solution that has inversions, we can eliminate the inversions one by one as shown above until there are no more inversions. This solution will also be optimal.

(6) Proved that there exists an optimal schedule with no inversions and no idle time.

Also proved that all schedules with no inversions and no idle time have the same Maximum Lateness.

Our greedy algorithm produces one such solution $\Rightarrow$ It will be optimal





Discussion 3

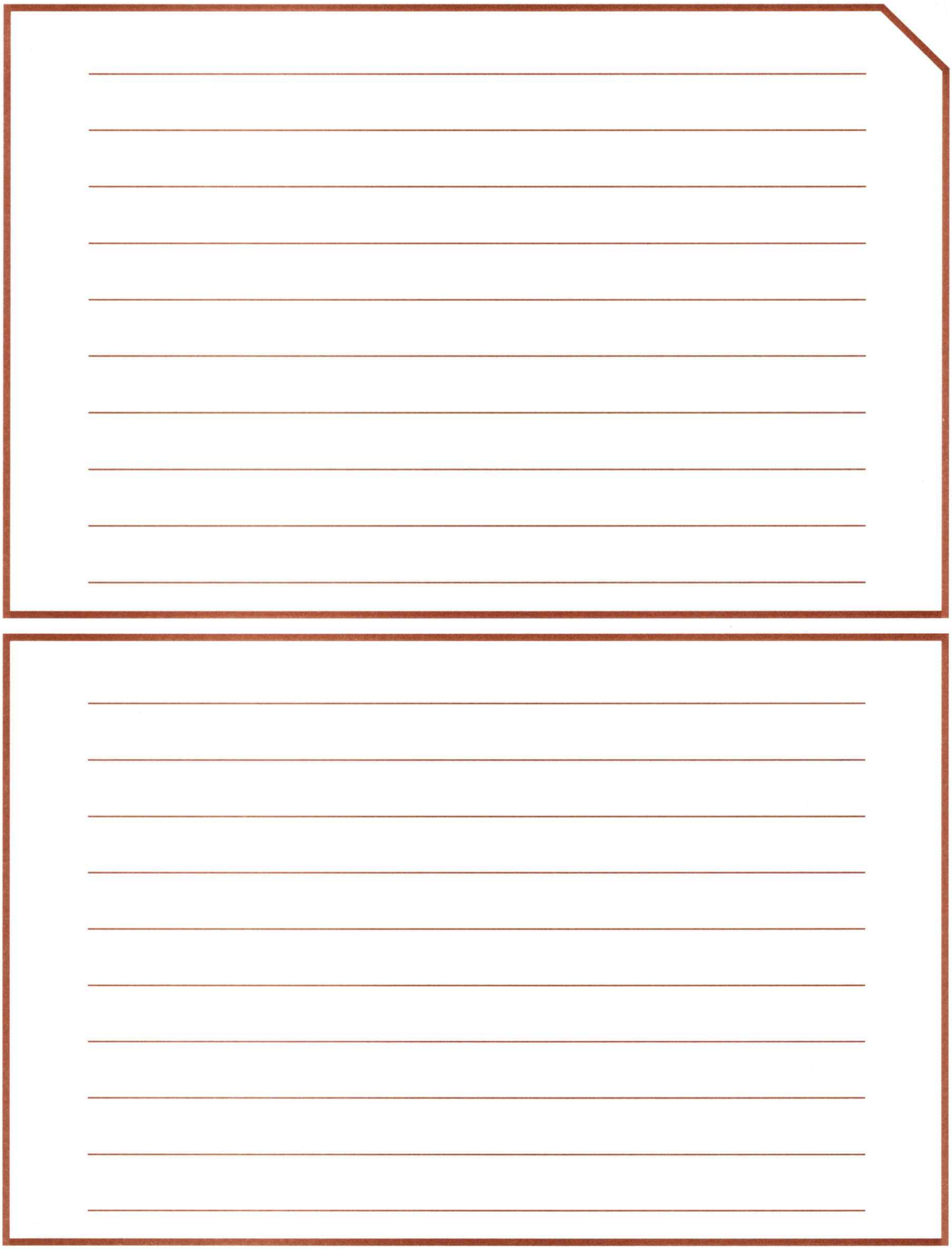
1. Let's consider a long, quiet country road with houses scattered very sparsely along it. We can picture the road as a long line segment, with an eastern endpoint and a western endpoint. Further, let's suppose that, despite the bucolic setting, the residents of all these houses are avid cell phone users. You want to place cell phone base stations at certain points along the road, so that every house is within four miles of one of the base stations.
Give an efficient algorithm that achieves this goal and uses as few base stations as possible. Prove that your algorithm correctly minimizes the number of base stations.

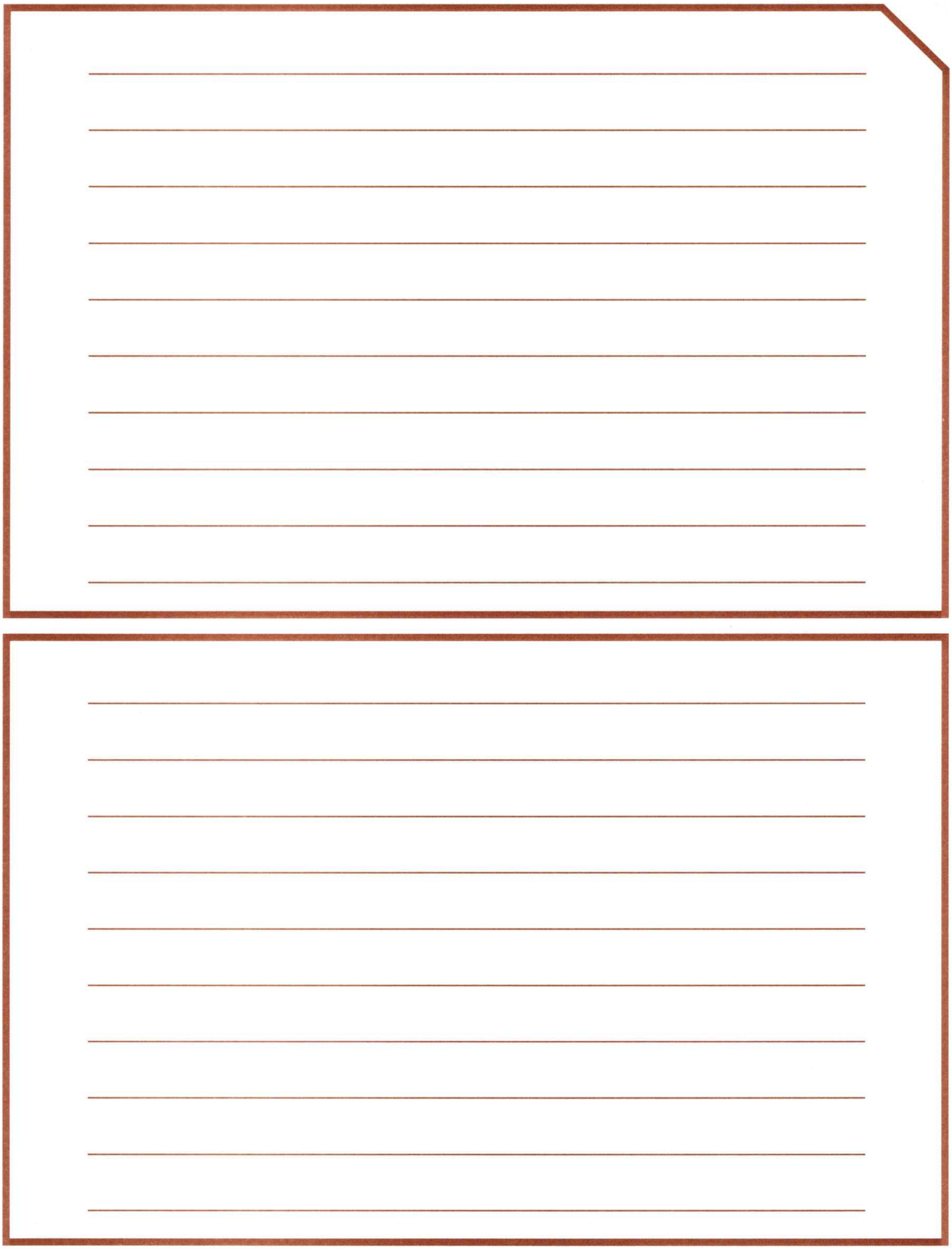
2. Your friend is working as a camp counselor, and he is in charge of organizing activities for a set of campers. One of his plans is the following mini-triathlon exercise: each contestant must swim 20 laps of a pool, then bike 10 miles, then run 3 miles. The plan is to send the contestants out in a staggered fashion, via the following rule: the contestants must use the pool one at a time. In other words, first one contestant swims the 20 laps, gets out, and starts biking. As soon as this first person is out of the pool, a second contestant begins swimming the 20 laps; as soon as he or she is out and starts biking, a third contestant begins swimming, and so on.
Each contestant has a projected *swimming time*, a projected *biking time*, and a projected *running time*. Your friend wants to decide on a *schedule* for the triathlon: an order in which to sequence the starts of the contestants. Let's say that the *completion time* of a schedule is the earliest time at which all contestants will be finished with all three legs of the triathlon, assuming the time projections are accurate.
What is the best order for sending people out, if one wants the whole competition to be over as soon as possible? More precisely, give an efficient algorithm that produces a schedule whose completion time is as small as possible. Prove that your algorithm achieves this.

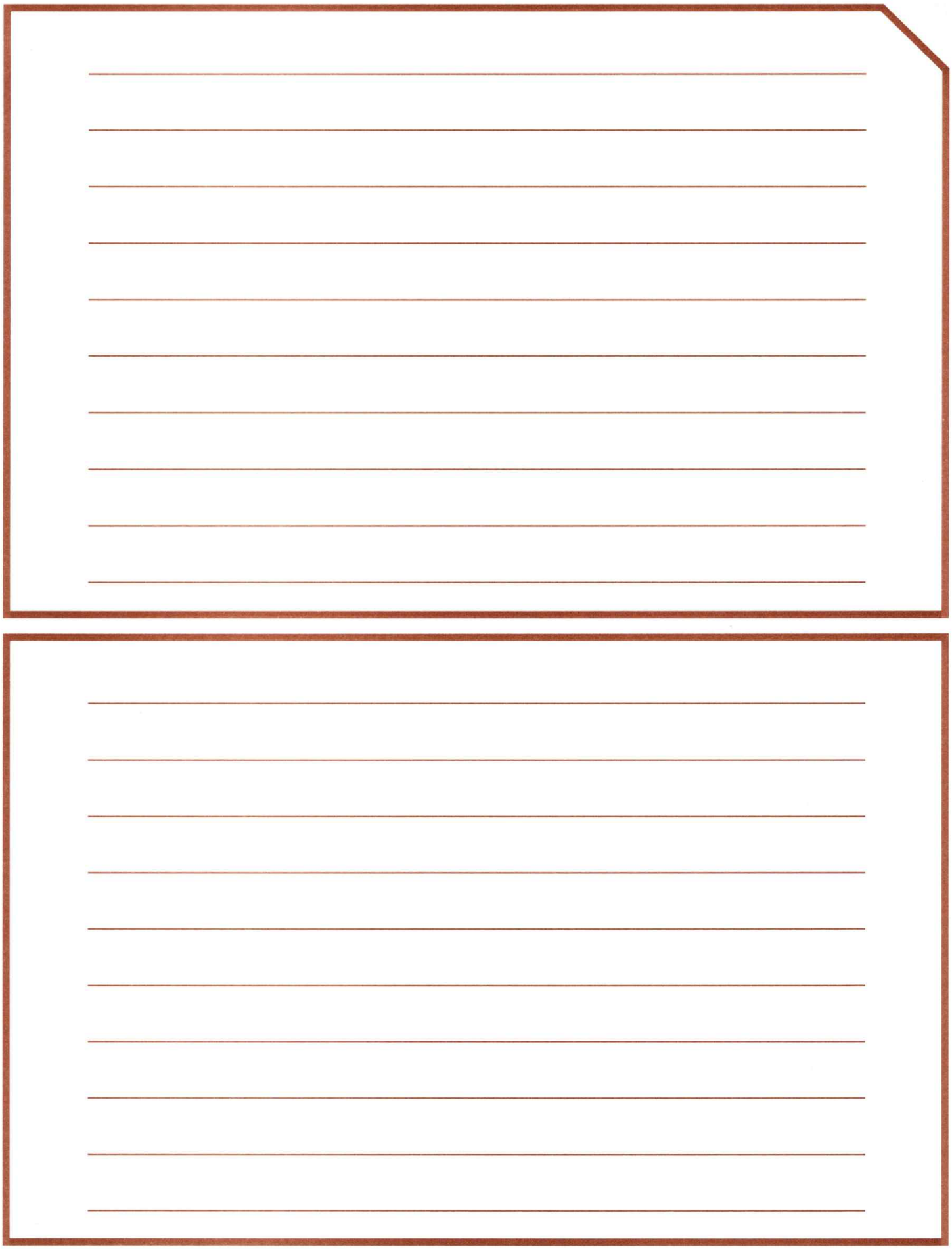
3. The values 1, 2, 3, . . . , 63 are all inserted (in any order) into an initially empty min-heap. What is the smallest number that could be a leaf node?

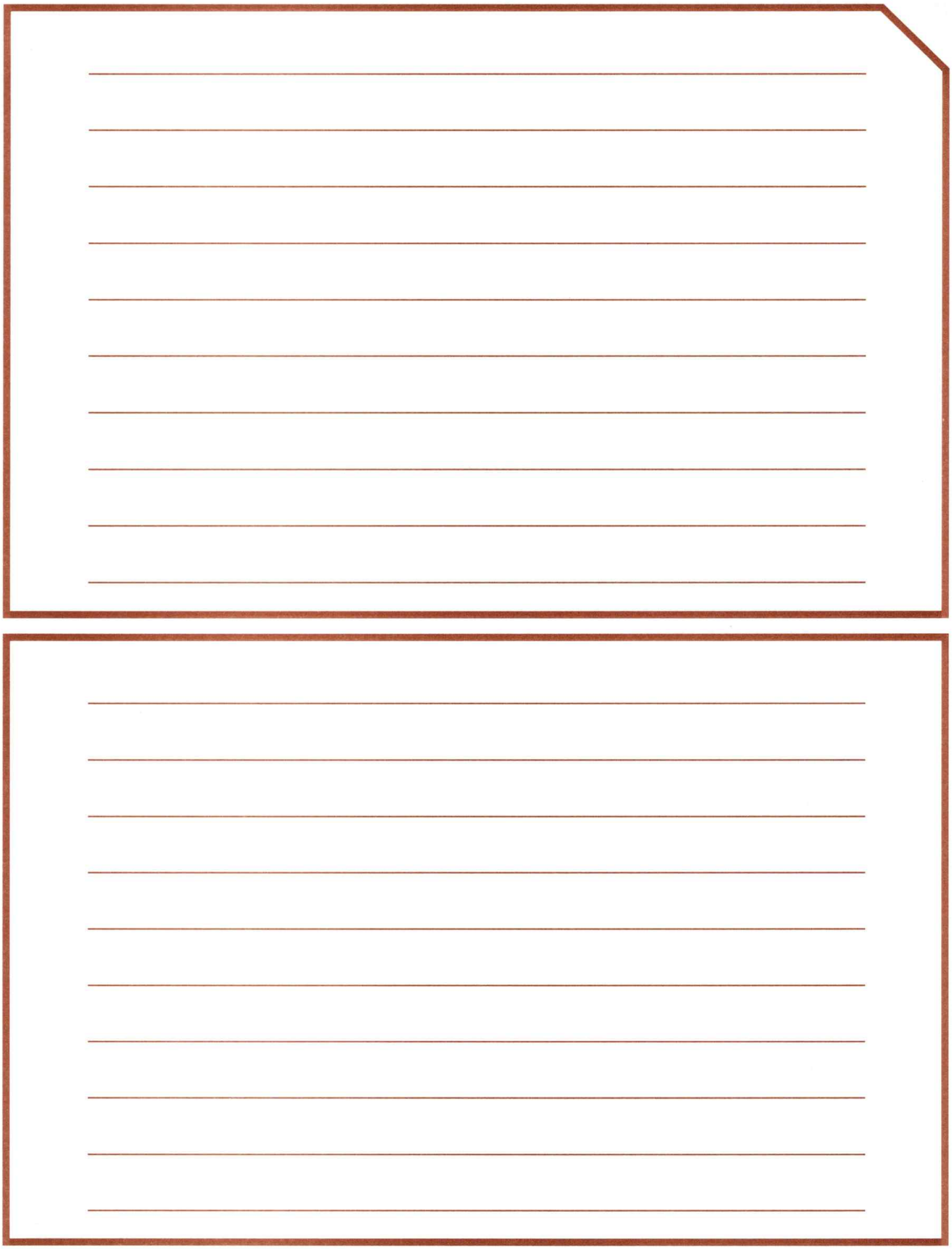
4. Given an unsorted array of size *n*. Devise a heap-based algorithm that finds the *k*-th largest element in the array. What is its runtime complexity?

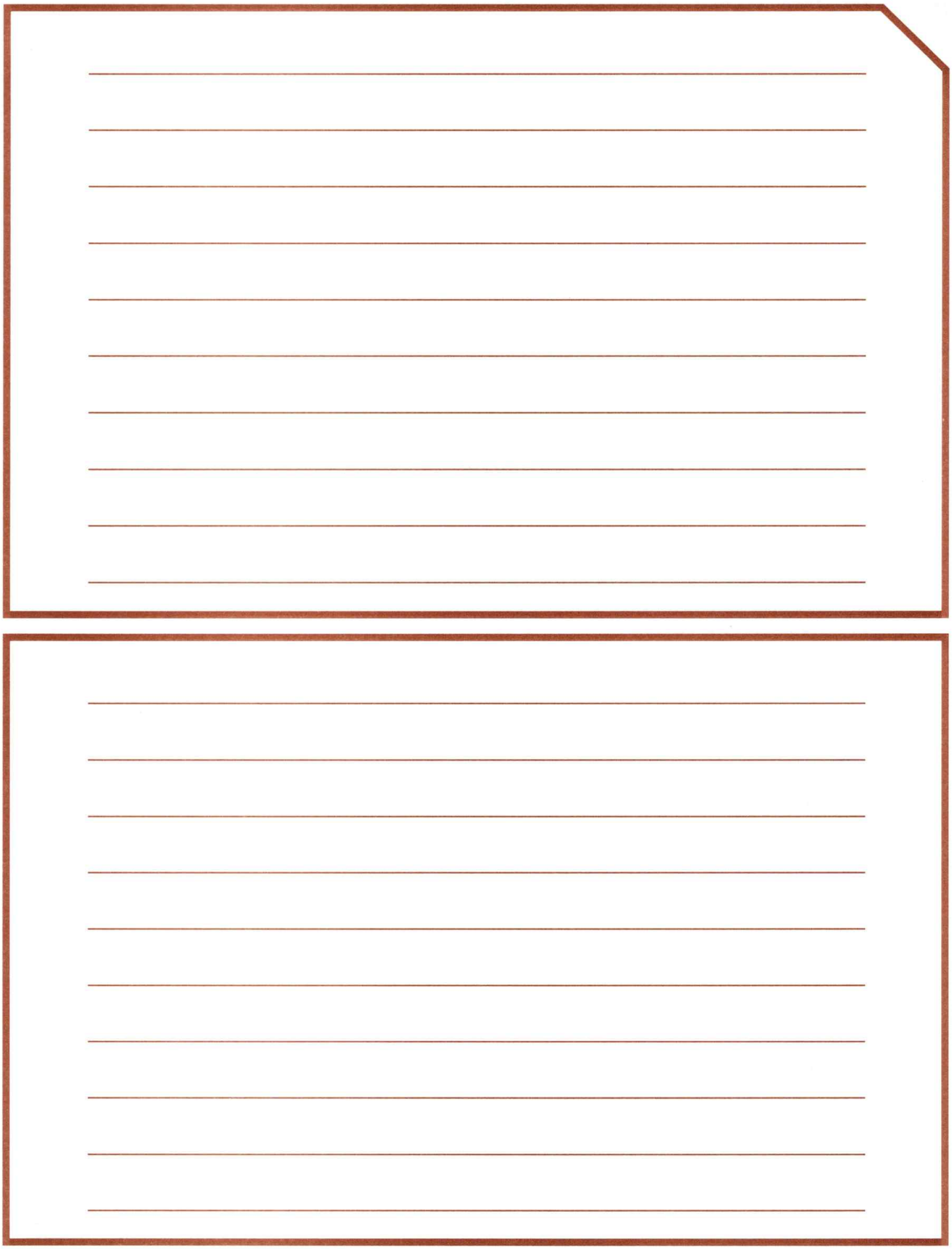
5. Suppose you have two min-heaps, A and B, with a total of *n* elements between them. You want to discover if A and B have a key in common. Give a solution to this problem that takes time O(n log n) and explain why it is correct. Give a brief explanation for why your algorithm has the required running time. For this problem, do not use the fact that heaps are implemented as arrays; treat them as abstract data types.











Priority Queues

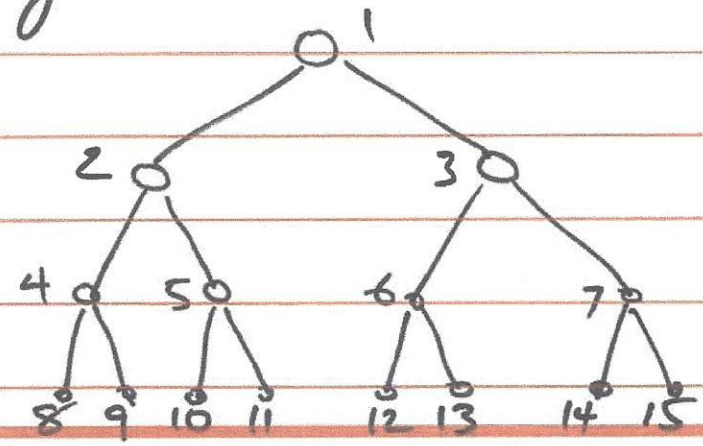
A priority queue has to perform these two operations fast!

1- Insert an element into the set

2- Find the smallest element in the set

Background

Def. A binary tree of depth $\underline{k}$ which has exactly $2^k - 1$ nodes is called a full binary tree.



Def. A binary tree with $\underline{n}$ nodes and of depth $\underline{k}$ is complete iff its nodes correspond to the nodes which are numbered 1 to $\underline{n}$ in the full binary tree of depth $\underline{k}$.

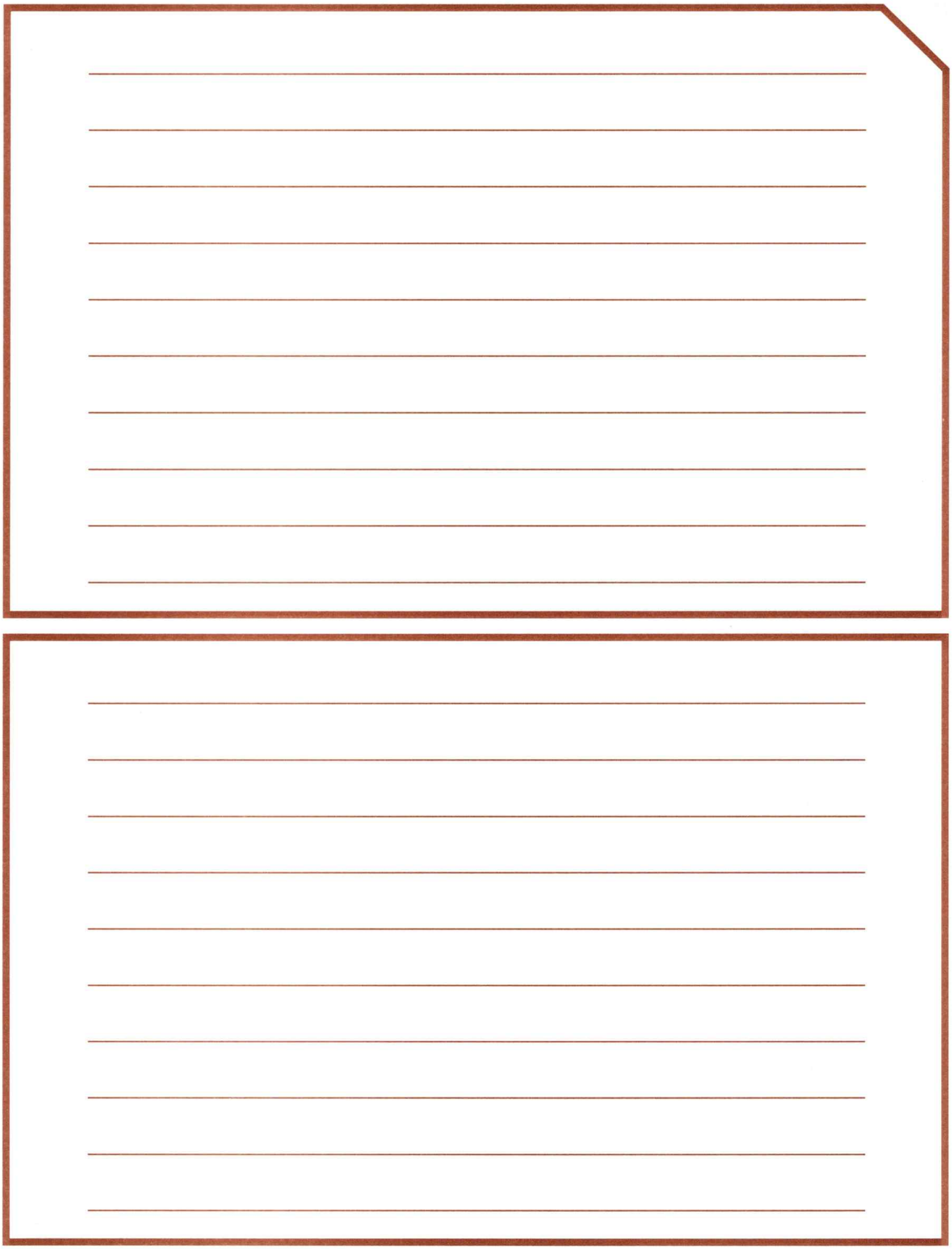
Traversing a complete binary tree stored as an array

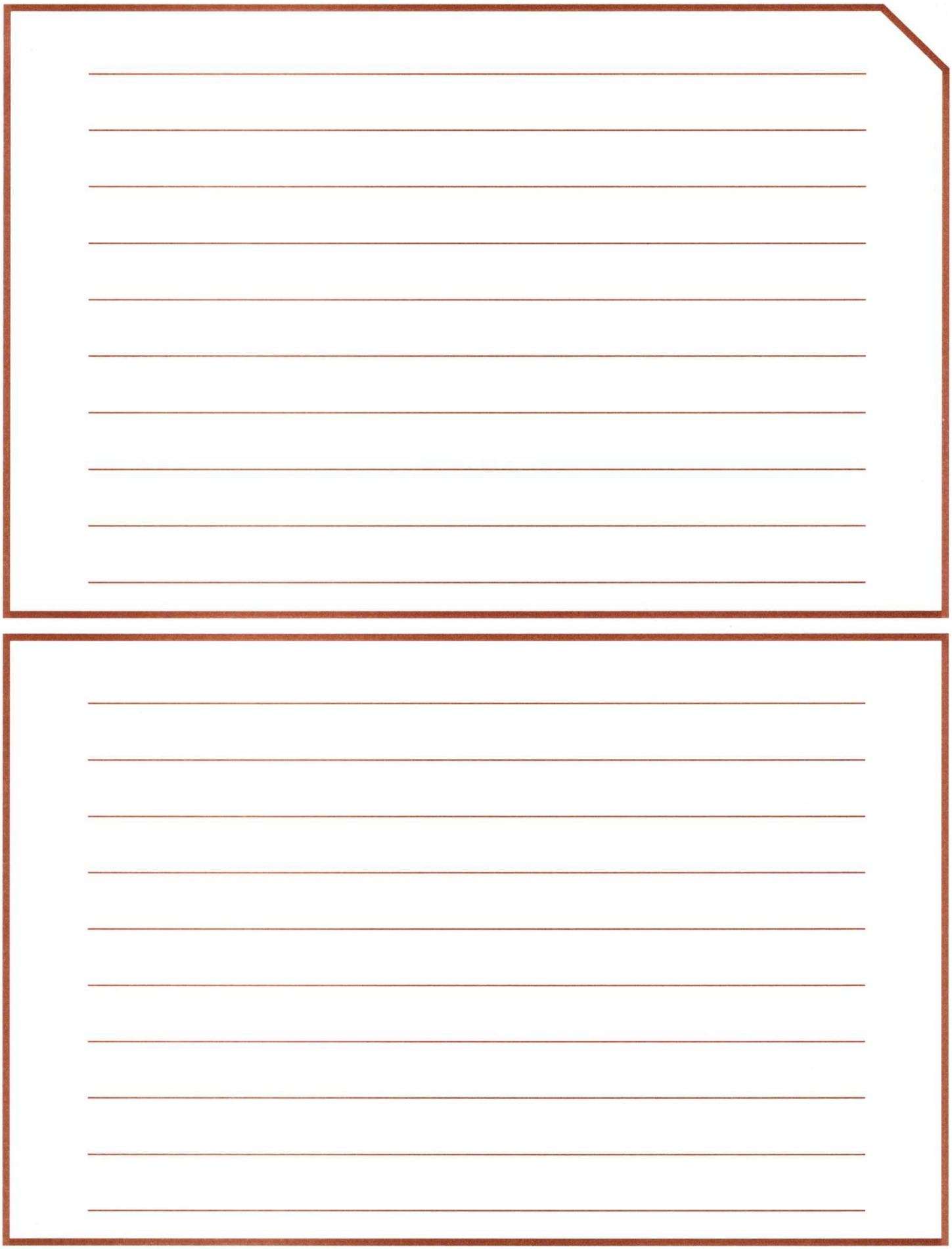
Parent(i) is at $\lfloor i/2 \rfloor$ if $i \neq 1$
if $i=1$, i is the root

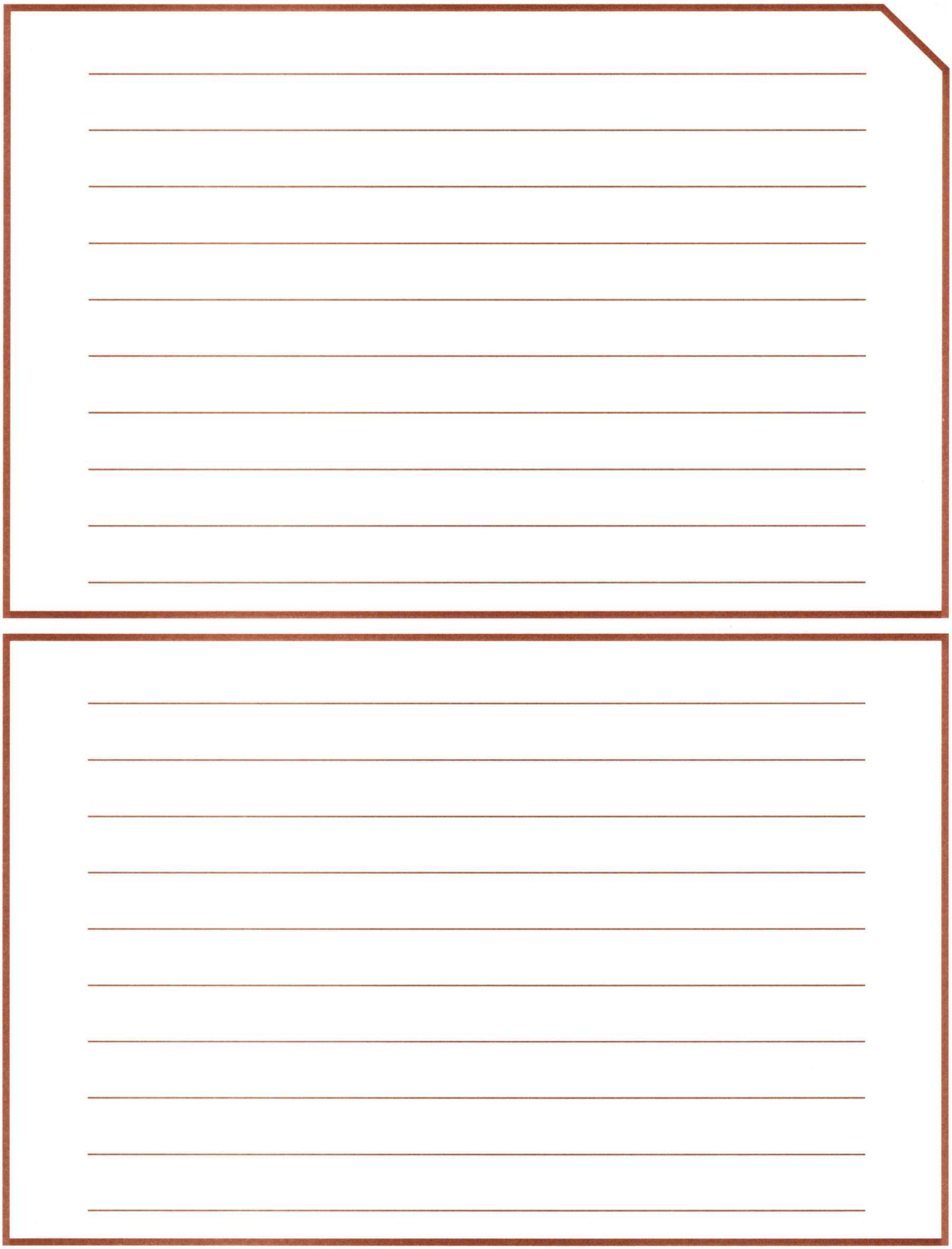
Lchild(i) is at $2i$ if $2i \leq n$
otherwise it has no left child

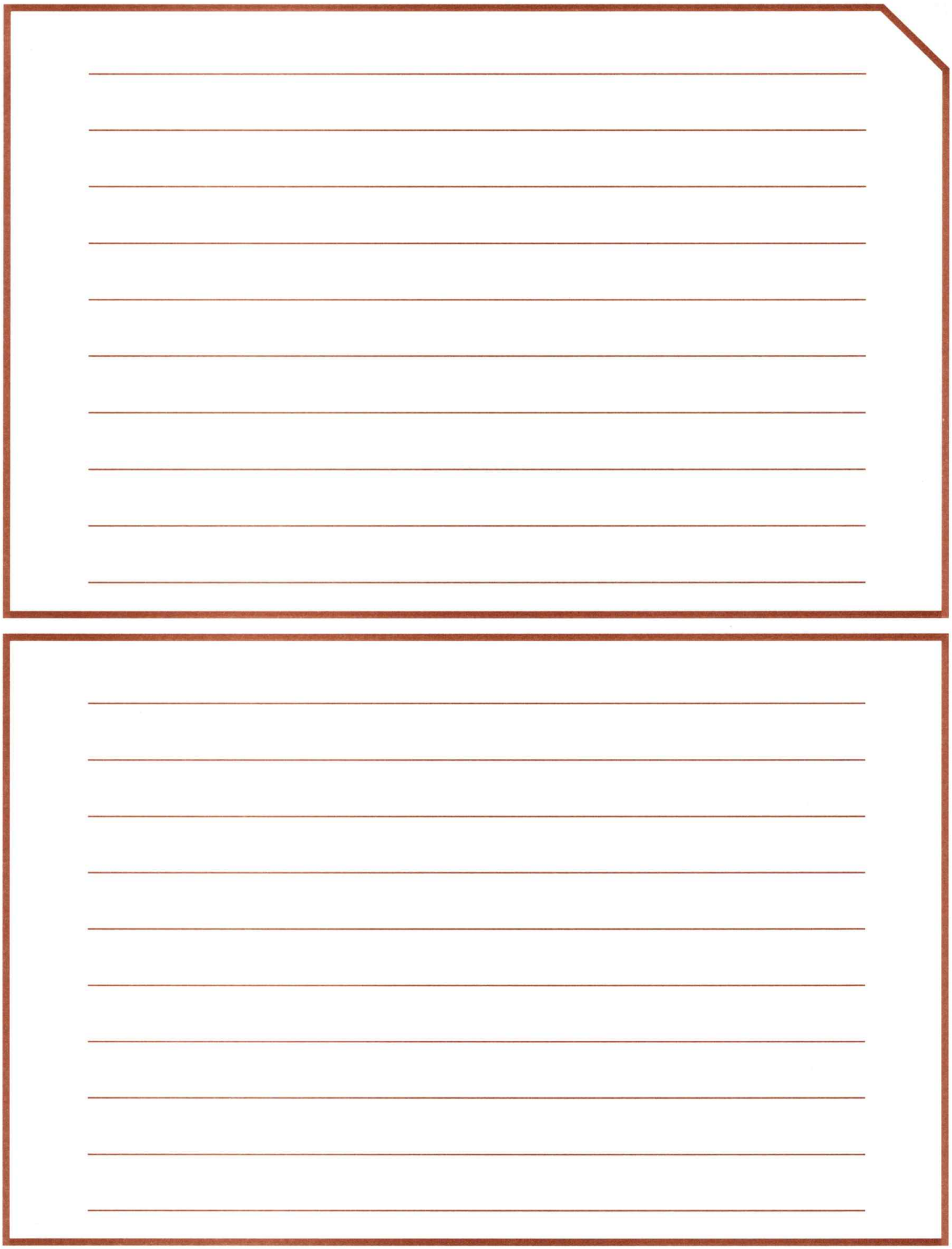
Rchild(i) is at $2i+1$ if $2i+1 \leq n$
otherwise it has no right child

Def. A binary heap is a complete binary tree with the property that the value (of the key) at each node is at least as large as the values at its children (Max heap)









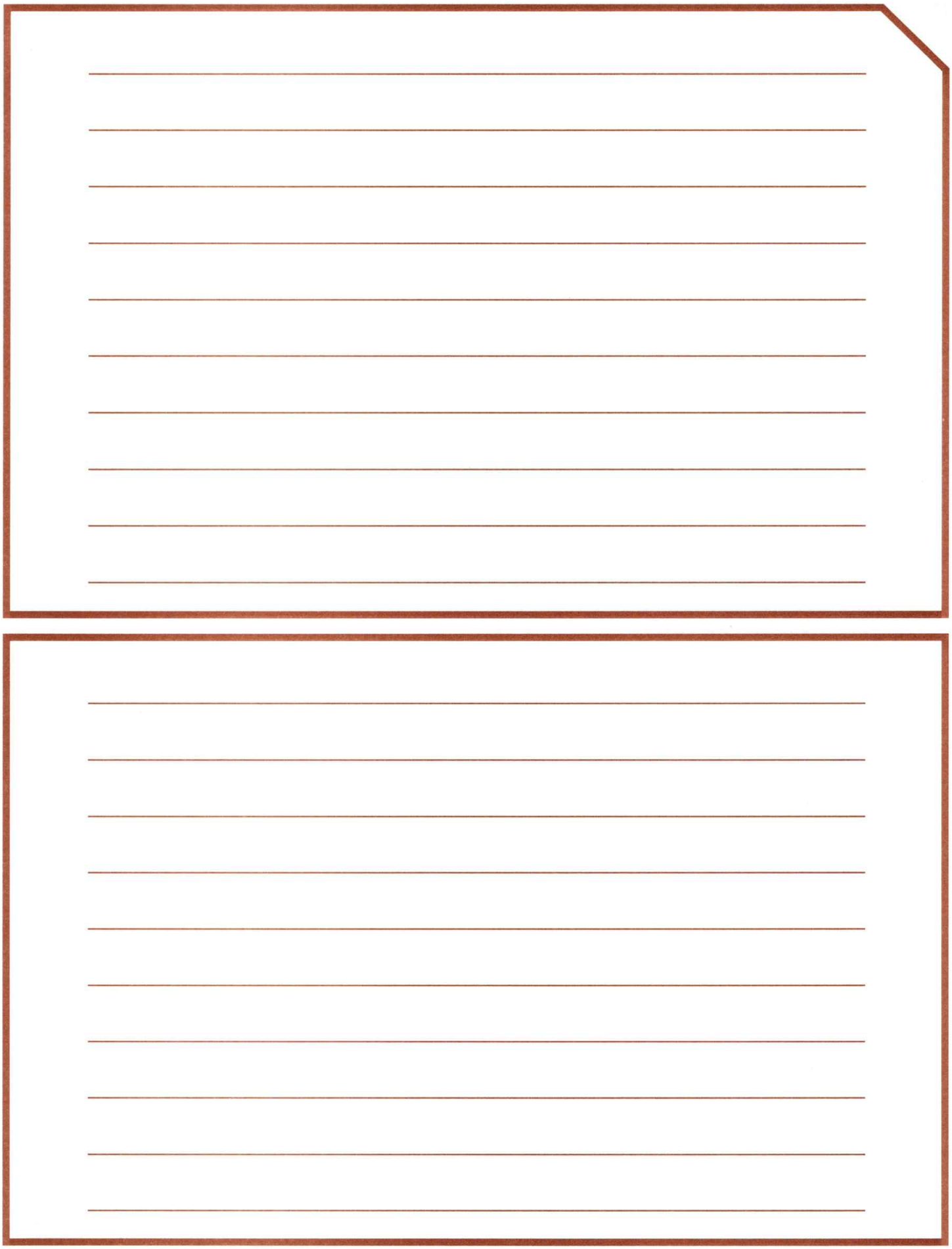
Problem Statement

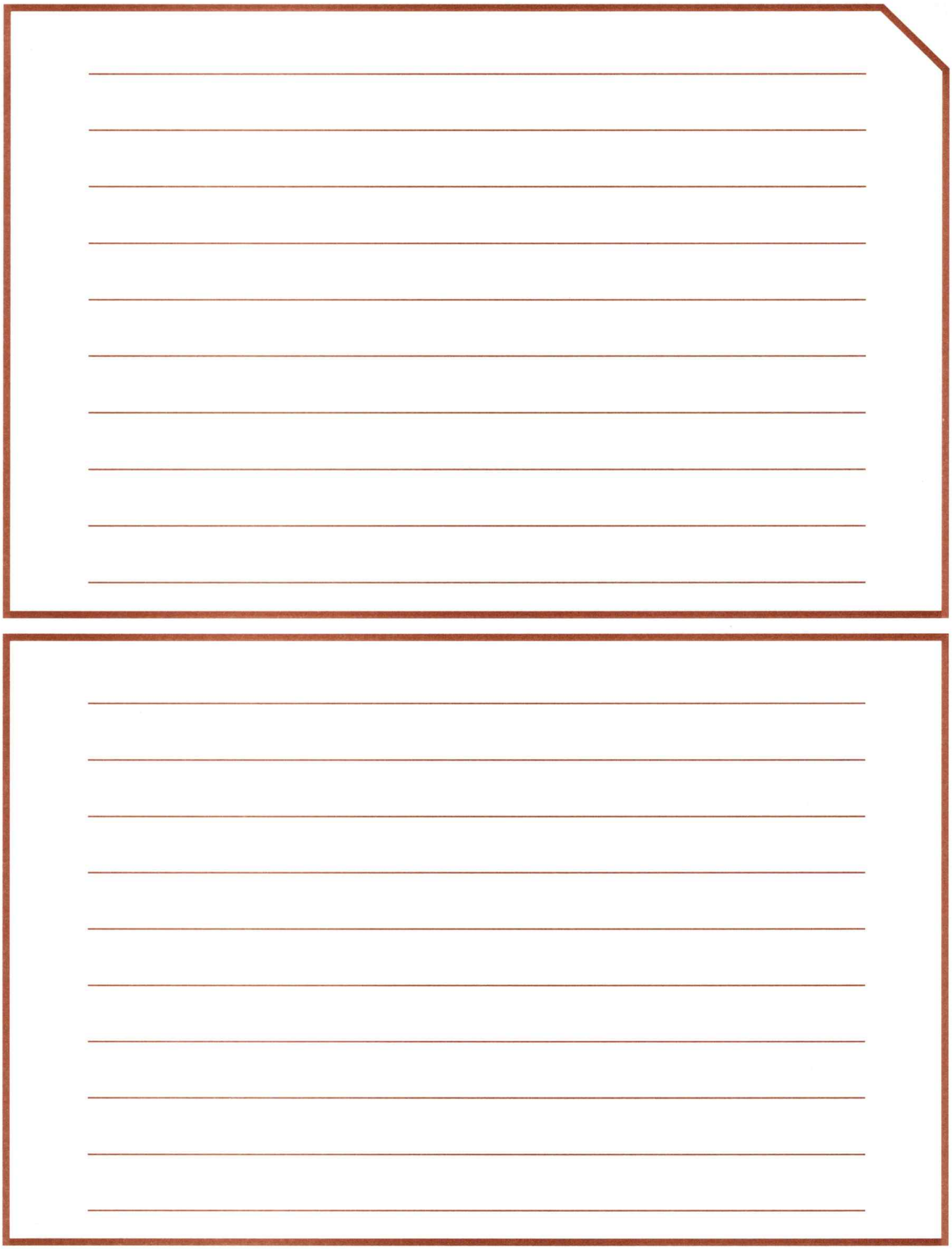
Input: An unsorted array of length $\underline{n}$

Output: Top $\underline{k}$ values in the array ($k < n$)

Constraints:

- You cannot use any additional memory
- Find an algorithm that runs in time $O(n \lg k)$





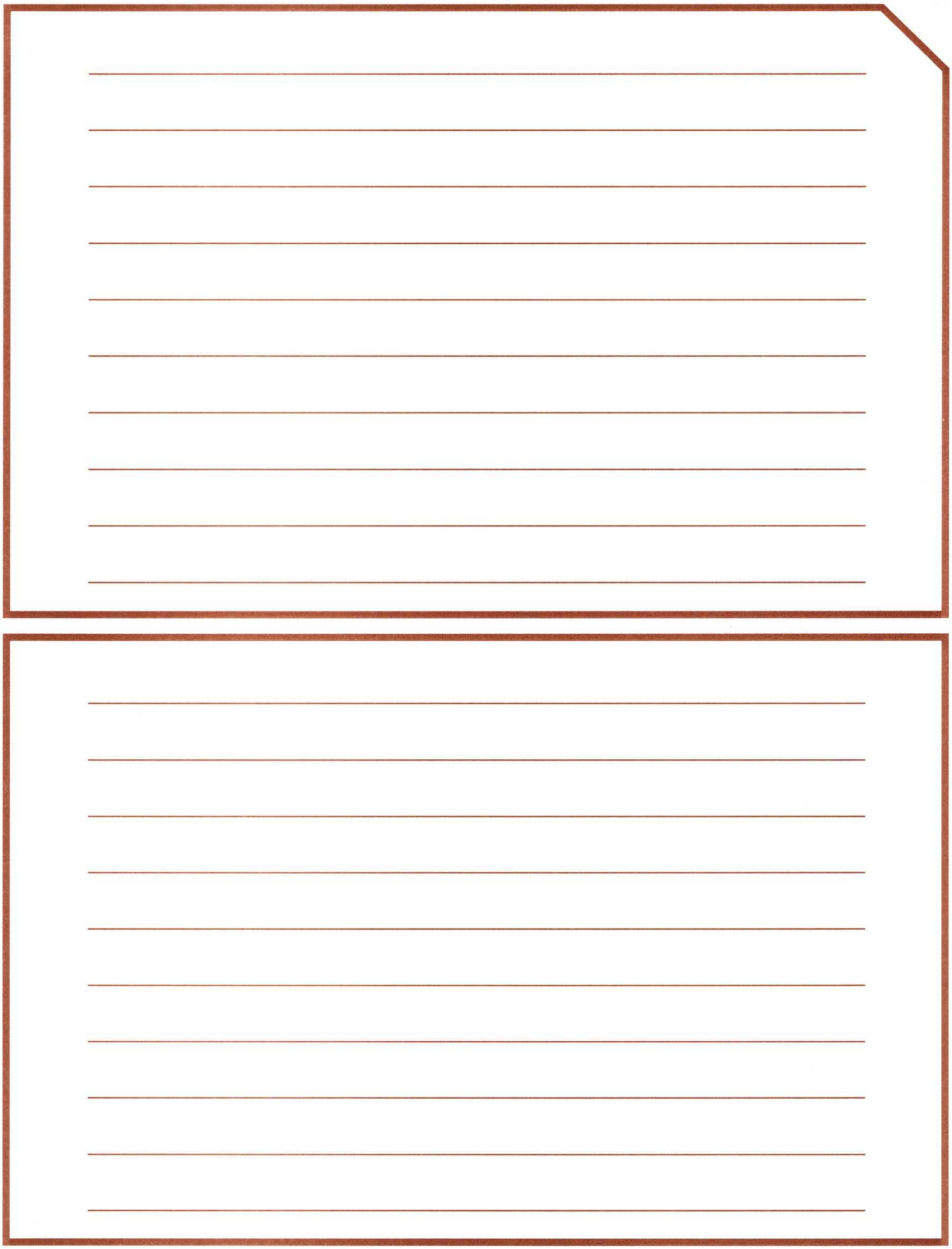
Def. A binomial tree B_k is an ordered tree defined recursively

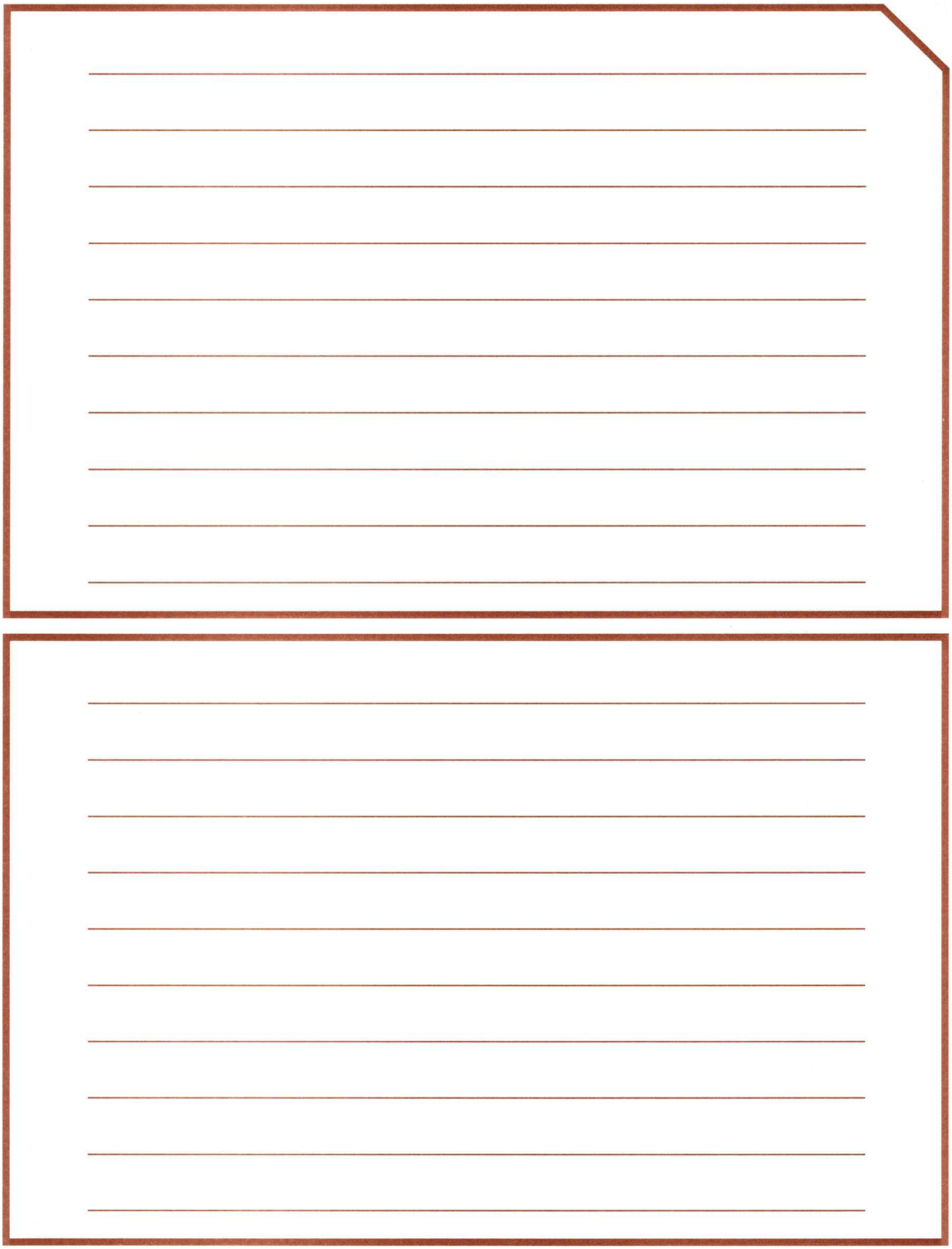
- Binomial tree B_0 consists of one node
- Binomial tree B_k consists of 2 binomial trees B_{k-1} that are linked together such that root of one is the leftmost child of the root of the other.

Def. A binomial Heap H is a set of binomial trees that satisfies the following properties:

1- Each binomial tree in H obeys the min-heap property

2- For any non-negative integer k, there is at most one binomial tree in H whose root has degree k.





Amortized Cost Analysis

Ex. 1

```
for i = 1 to n
    push or Pop
endfor
```

push takes $O(1)$
Pop takes $O(1)$

Our worst case run time complexity =

Ex. 2

```
for i = 1 to n
    push or pop or multipop
endfor
```

push takes $O(1)$ worst-case
pop takes $O(1)$ worst-case
multipop takes $O(n)$ worst-case

Our worst case run time complexity =

Aggregate Analysis

- We show that a sequence of $\underline{n}$ operations (for all $\underline{n}$) takes worst-case time $T(n)$ total.

- So, in the worst case, the amortized cost (average cost) per operation will be $T(n)/n$

observation: Multipop takes $O(n)$ time if there are $\underline{n}$ elements pushed on the stack

A sequence of $\underline{n}$ pushes takes $O(n)$
multipop takes $O(n)$ worst-case

$T(n) = O(n)$
when amortized over $\underline{n}$ operations,
average cost of an operation $= O(n)/n = O(1)$

Ex. 2

```
for i = 1 to n
    push or pop or multipop
end for
```

push takes $O(1)$ amortized
pop " $O(1)$ "
multipop " $O(1)$ "

our worst case runtime complexity =

Accounting Method

- We assign different charges (amortized costs) to different operations

- If the charge for an operation exceeds its actual cost, ~~we~~ the excess is stored as credit.

- The credit can later helps pay for operations whose actual cost is higher than their amortized cost.

- Total credit at any time = total amortized cost − total actual cost

- Total credit can never be negative.

Ex. 2 for $i = 1$ to n

push or pop* or multipop

endfor

try # 1

assign a ~~cost~~ charge of 1 to each operation

op.	charge	Actual Cost	total Credit
push	1	1	0
push	1	1	0
multipop	1	2	-1

try #2

assign charges as follows:

push 2

pop 0

multipop 0

op.	charge	Actual Cost	total Credit
push	2	1	1
push	2	1	2
multipop	0	2	0

Ex. 2

```
for i = to n
    push or pop or multipop
end for
```

	Amortized cost
push	$O(1)$
pop	$O(1)$
multipop	$O(1)$

worst-case run time complexity $= O(n)$

- Fibonacci heaps are loosely based on binomial heaps.

- A Fibonacci heap is a collection of min-heap trees similar to Binomial heaps, however, trees in a Fibonacci heap are not constrained to be binomial trees. Also, unlike binomial heaps, trees in Fibonacci heaps are not ordered.

- Link to Fibonacci heap animation:

www.cs.usfca.edu/ngalles/JavascriptVisual/FibonacciHeap.html

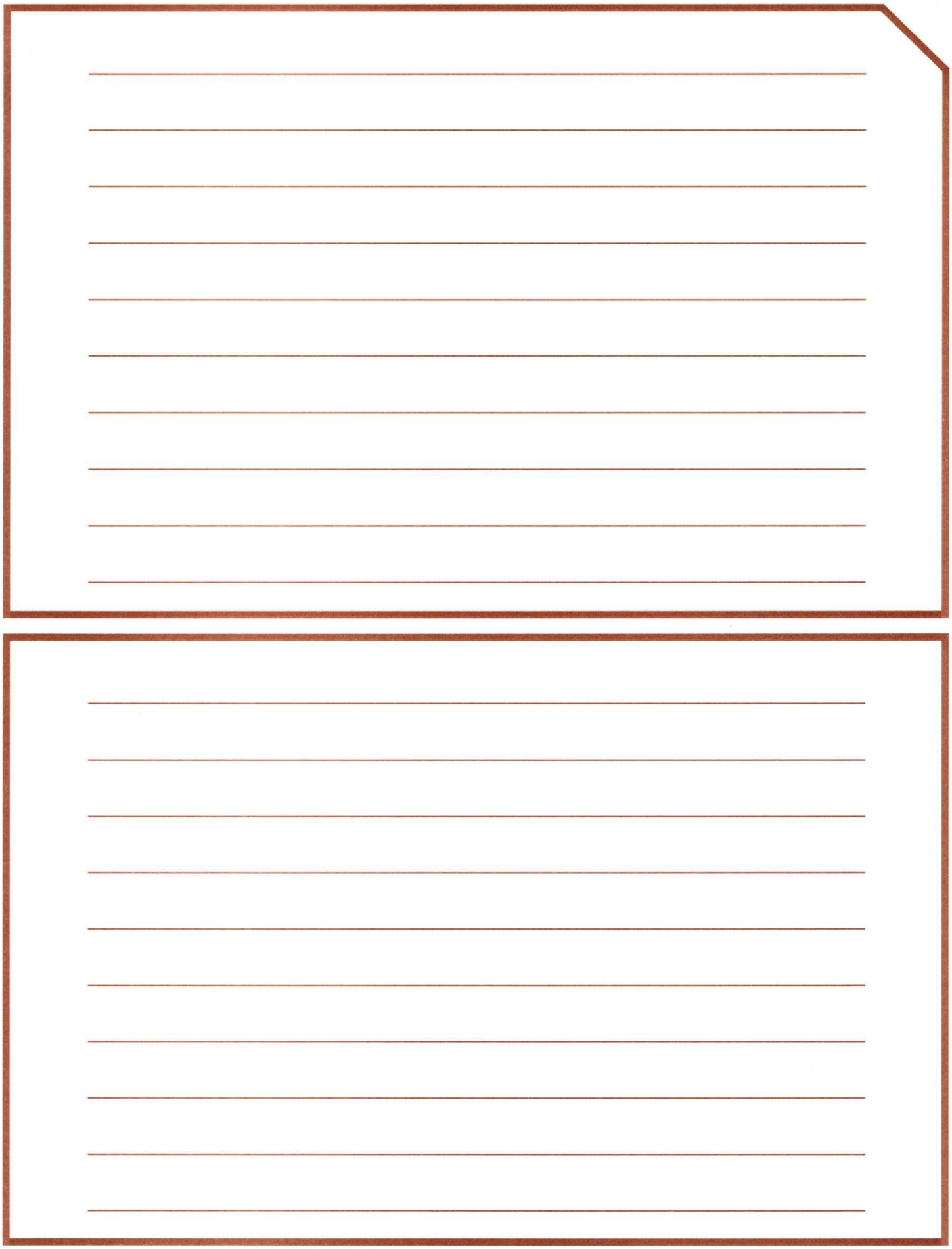
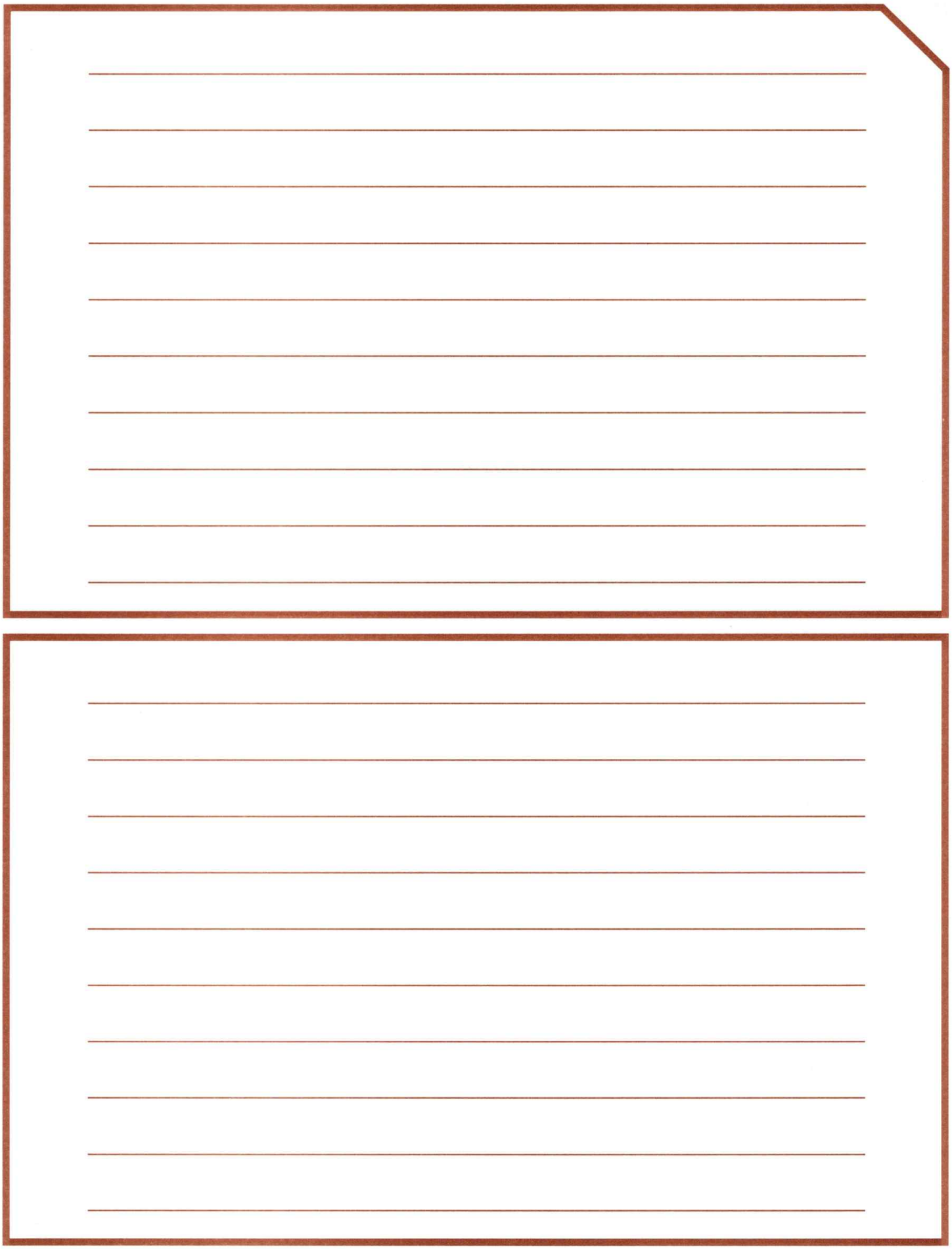
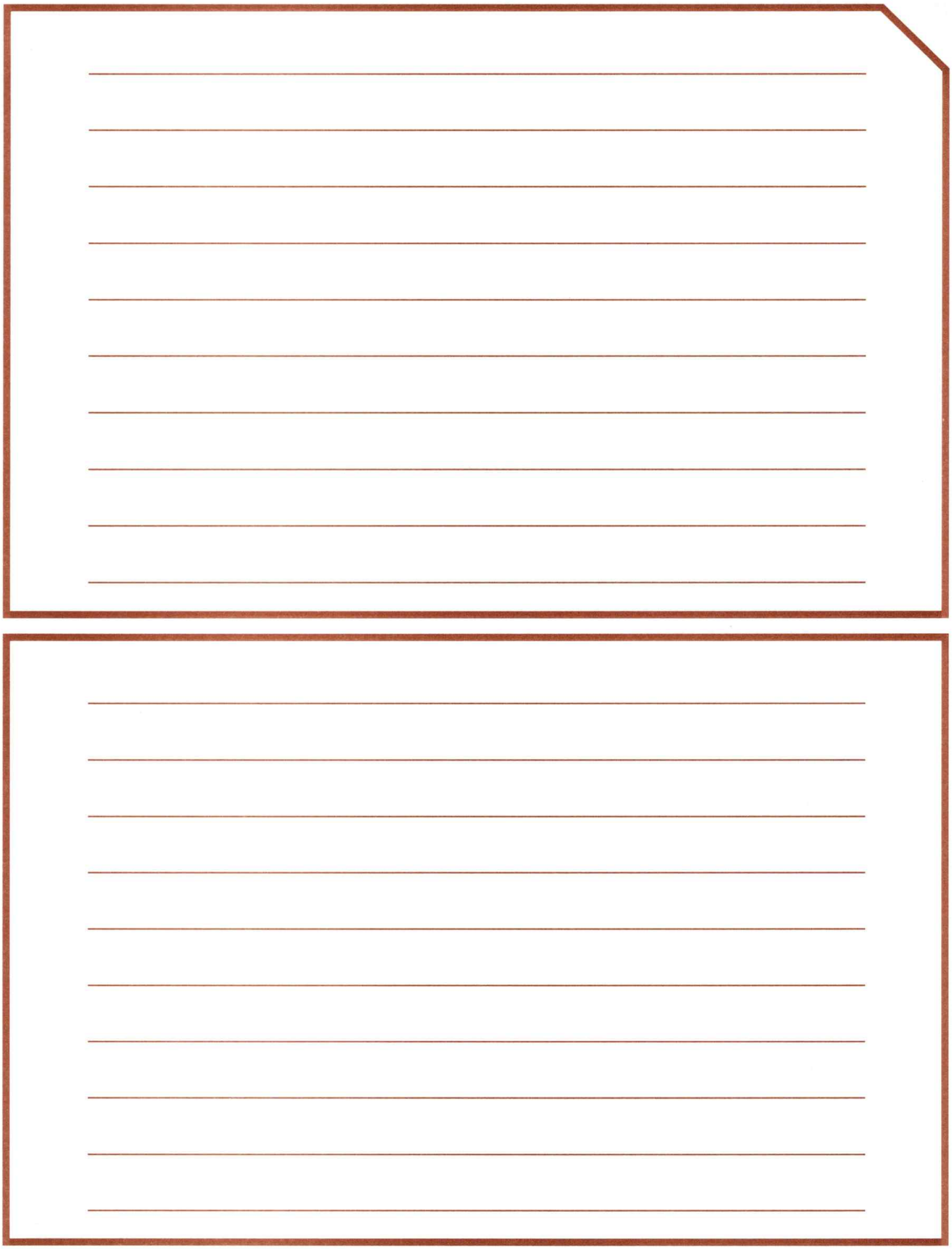
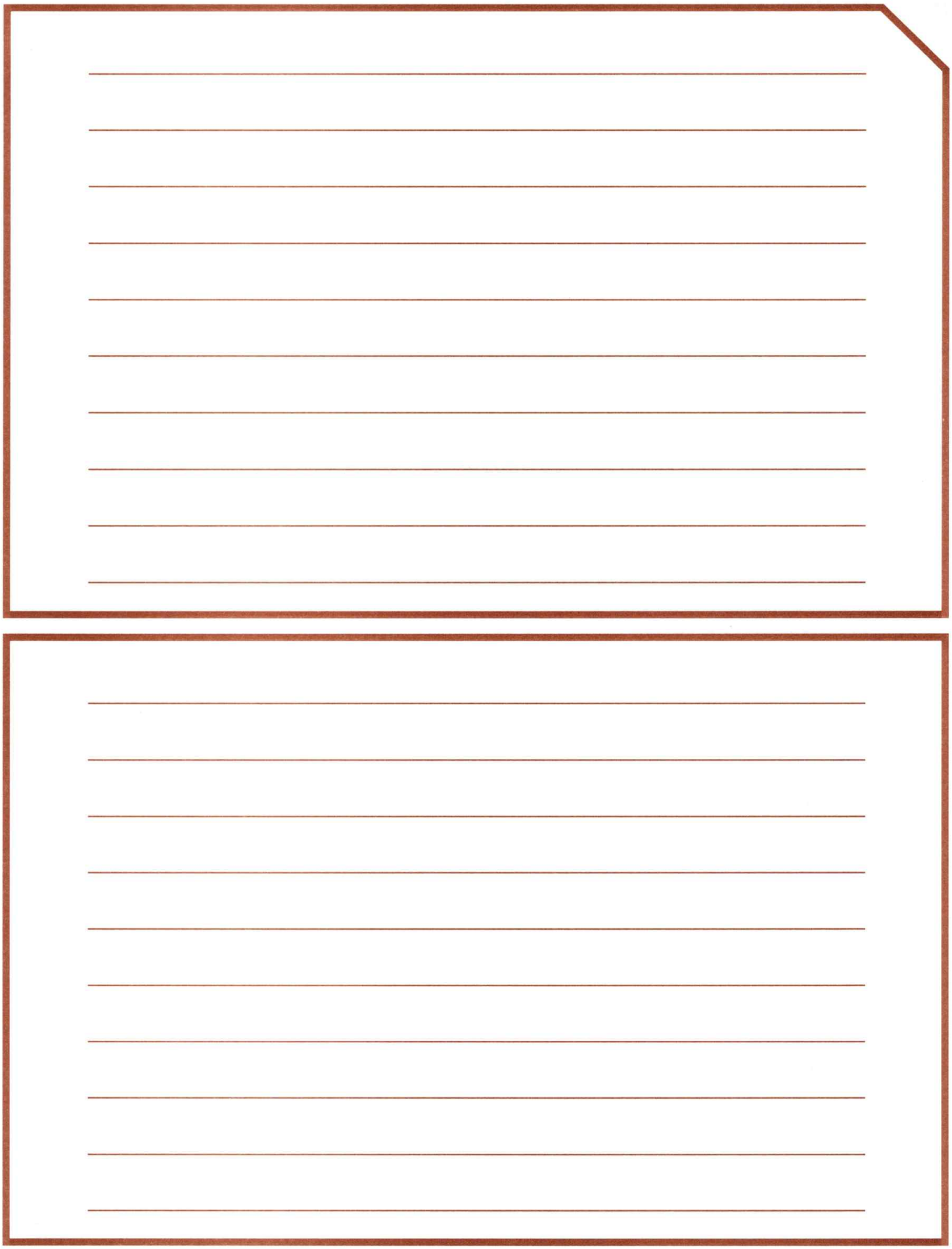
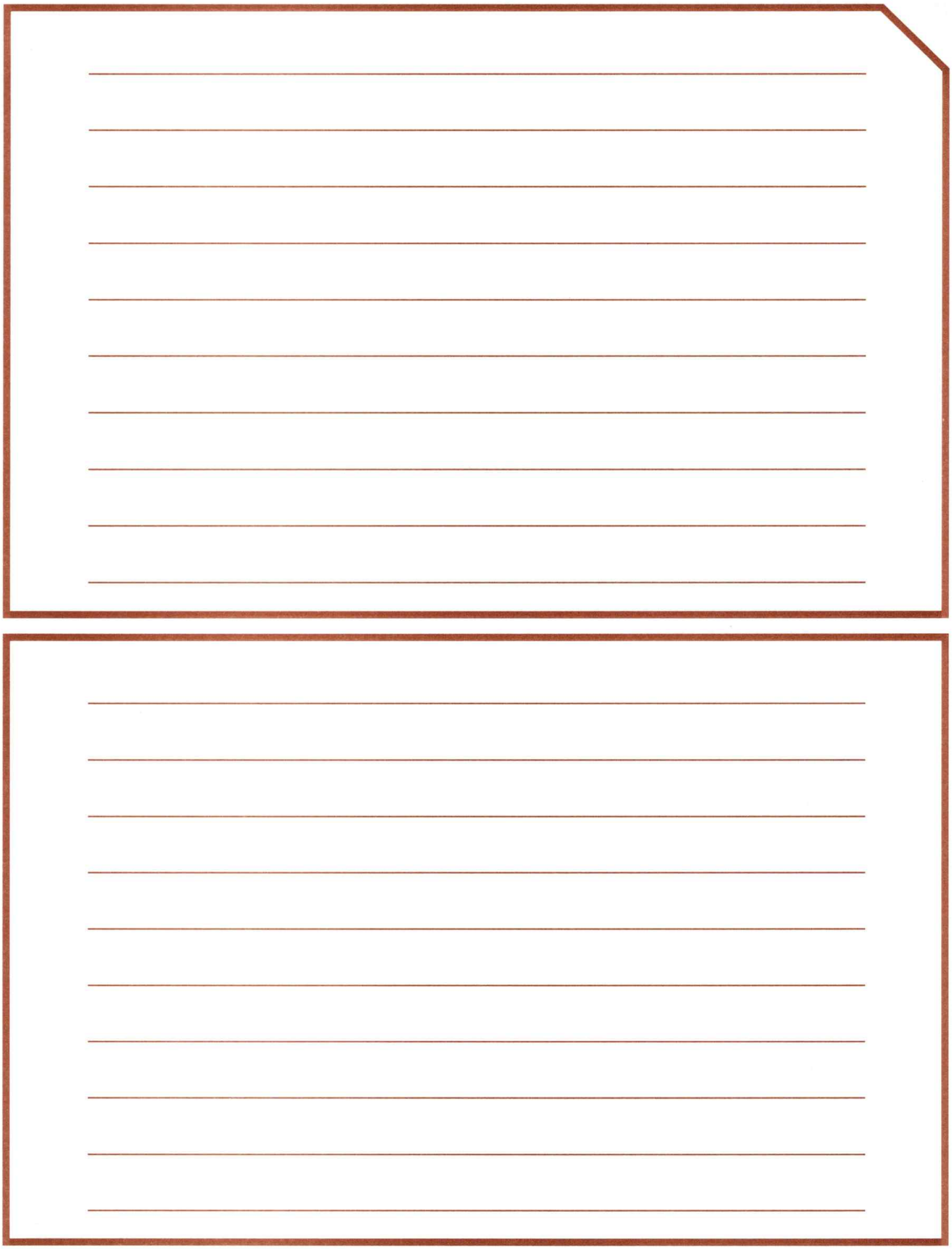
	Binary Heap	Binomial Heap	Fibonacci Heap
Find-Min	$O(1)$	$O(\lg n)$	$O(1)$
Insert	$O(\lg n)$	"	$O(1)$
Extract-Min	"	"	$O(\lg n)$
Delete	"	"	$O(\lg n)$
Decrease-key	"	"	$O(1)$
Merge	$O(n)$	"	$O(1)$
Construct	$O(n)$	$O(n)$	$O(n)$

Discussion 3

1. Let's consider a long, quiet country road with houses scattered very sparsely along it. We can picture the road as a long line segment, with an eastern endpoint and a western endpoint. Further, let's suppose that, despite the bucolic setting, the residents of all these houses are avid cell phone users. You want to place cell phone base stations at certain points along the road, so that every house is within four miles of one of the base stations.
Give an efficient algorithm that achieves this goal and uses as few base stations as possible. Prove that your algorithm correctly minimizes the number of base stations.

2. Your friend is working as a camp counselor, and he is in charge of organizing activities for a set of campers. One of his plans is the following mini-triathlon exercise: each contestant must swim 20 laps of a pool, then bike 10 miles, then run 3 miles. The plan is to send the contestants out in a staggered fashion, via the following rule: the contestants must use the pool one at a time. In other words, first one contestant swims the 20 laps, gets out, and starts biking. As soon as this first person is out of the pool, a second contestant begins swimming the 20 laps; as soon as he or she is out and starts biking, a third contestant begins swimming, and so on.
Each contestant has a projected *swimming time*, a projected *biking time*, and a projected *running time*. Your friend wants to decide on a *schedule* for the triathlon: an order in which to sequence the starts of the contestants. Let's say that the *completion time* of a schedule is the earliest time at which all contestants will be finished with all three legs of the triathlon, assuming the time projections are accurate.
What is the best order for sending people out, if one wants the whole competition to be over as soon as possible? More precisely, give an efficient algorithm that produces a schedule whose completion time is as small as possible. Prove that your algorithm achieves this.

3. The values 1, 2, 3, . . . , 63 are all inserted (in any order) into an initially empty min-heap. What is the smallest number that could be a leaf node?

4. Given an unsorted array of size *n*. Devise a heap-based algorithm that finds the *k*-th largest element in the array. What is its runtime complexity?

5. Suppose you have two min-heaps, A and B, with a total of *n* elements between them. You want to discover if A and B have a key in common. Give a solution to this problem that takes time O(n log n) and explain why it is correct. Give a brief explanation for why your algorithm has the required running time. For this problem, do not use the fact that heaps are implemented as arrays; treat them as abstract data types.