Atomic Operations
1. Register Initialization
2. Arithmetic (+ - * /)
3. Comparison/branch
4. Memory read/write
5. Random
x mod y O(1): x – x / y

Tower of Hanoi
f(n, from, to, int):
  f(n-1, from, int, to)
  move n from → to
  f(n-1, int, to, from)
$f(n) = 2f(n-1)+1 = 2^n-1$

K-selection (2/3n)
1) Take element v ∈ S uniformly at random.
2) Divide S into S1 and S2 where:
S1 = the set of elements in S <= v;
S2 = the set of elements in S > v.
3) If |S1| ⩾ k, return S′ = S1 and k′ = k;
else return S′ = S2 and k′ = k − |S1|
→ succeed if rank(v) in [n/3, 2n/3] (1/3 prob)
Eval: 1 trial expected 3 times to succeed → O(3n) = O(n)
O(n)+O(2/3n)+O(n(2/3)^2)+…=n/(1-2/3)=O(n)

k-selection (1/2n)
1. run k-selection (2/3n) twice (|S′| → run again)
→ size of |S′′| <= n (2/3) (2/3) = 4n/9
2. repeat until pivots p1 in [n/4, n/2]; p2 in [n/2, 3n/4]
→ no matter where k is guarantee |S′| <= n/2
k<=r (p1),k′=k; r(p1)<k<r(p2),k′=k-r(p1);k>=r(p2),k′=k-r(p2)
- each pivot ¼ chance succeed (still O(n))(4 repeats each)

SPEX 1.3: **given function finding median in O(n), design deterministic k-selection in O(n)**
use function to determine pivot → f(n/2)+O(n) = O(n)

SPEX 1.4: **report number who ranks range [k1,k2].**
Use k-select find k1 and k2, then report all int in the range between k1 and k2

SPEX 1.5: Given set S of n distinct int, W = sum of element e in S, find e* in S s.t. following hold in O(n):
- $\sum_{e<e^*} e < W/2$
- $\sum_{e>e^*} e \le W/2.$

Randomly select pivot v in S → S1(<v) and S2(>v)
If sum of S1 < W/2 and sum of S2 <= W/2 → return v
If sum of S1 > W/2, S′ = S1; If sum of S2>W/2, S′ = S2
So modify k-sel to find v=e*

Geometric Series $\sum_{i=0}^{\infty} c^i n = \frac{n}{1-c} = O(n).$

Let $f(n)$ be a function that returns a positive value for every integer $n > 0$. We know:

$$\begin{aligned} f(1) &= O(1) \\ f(n) &\le \alpha \cdot f(\lceil n/\beta \rceil) + O(n^{\gamma}) \qquad (\text{for } n \ge 2) \end{aligned}$$

where $\alpha \ge 1$, $\beta > 1$, and $\gamma \ge 0$ are constants. Then:
- If $\log_\beta \alpha < \gamma$, then $f(n) = O(n^\gamma)$.
- If $\log_\beta \alpha = \gamma$, then $f(n) = O(n^\gamma \log n)$.
- If $\log_\beta \alpha > \gamma$, then $f(n) = O(n^{\log_\beta \alpha})$.

Count Inversion
#pairs (i,j) s.t. A[i] > A[j] and i < j
1. Divide + Binary Search
1) divide into two halves (A1, A2). 2) sort each half(n lg n)
3) count internal inversion in each 2f(n/2)
4) for i in A1, BS in A2 → find the pos of i in A2, count #crossing inversion (n lg n)
f(n) <= 2 f(n/2) + O(n lg n) ➔ O(n lg^2 n)
2. Merge Sort
Do merge sort, count crossing inversion at merge stage
Merge A[a], B[b] → for each a, #inv = b at that time
(i.e. #elements in B that is < A[a]) (begin=0)
f(n) <= 2 f(n/2) + O(n) ➔ O(n lg n)

SPEX 2.1: output all inv in array A in $O(n \lg^2 n + k)$, k = #inv
Method 1, print crossing inv after BS position

Dominance Counting
q dominates p if p.x ⩽ q.x and p.y ⩽ q.y

SPEX 2.2: Prove if dominance counting done at f(n) time, can count number of inversion in f(n) + O(n) time. Ans: Turn the array to 2D map, where x = A[i], y = -i, this done in O(n) time, then use dominance counting to count it in f(n) time.
1. Divide + Binary Search
1) divide into two halves S1 S2 based on x
  (find mid-cutting line w/ k-sel O(n))
2) solve each half (S1, S2) → 2 * f(n/2)
3) sort S2 points by y-coor → O(n lg n)
4) for each point in S1, binary search
$f(n) <= O(n) + 2f(n/2) + O(n \lg n) + O(n \lg n)$ ➔ $O(n \lg^2 n)$
2. Merge Sort
1) divide into two halves S1 S2 based on x (O(n))
2) solve each half (S1, S2) → return S1, S2 sorted by y
3) merge S1 and S2 points (similar to inv.) → return sorted
f(n) <= O(n) + 2f(n/2)+O(n) ➔ O(n lg n)

Matrix Multiplication

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} = \begin{bmatrix} p_5 + p_4 - p_2 + p_6 & p_1 + p_2 \\ p_3 + p_4 & p_1 + p_5 - p_3 - p_7 \end{bmatrix}$$

$$\begin{aligned} p_1 &= A_{11}(B_{12} - B_{22}) \\ p_2 &= (A_{11} + A_{12})B_{22} \\ p_3 &= (A_{21} + A_{22})B_{11} \\ p_4 &= A_{22}(B_{21} - B_{11}) \\ p_5 &= (A_{11} + A_{22})(B_{11} + B_{22}) \\ p_6 &= (A_{12} - A_{22})(B_{21} + B_{22}) \\ p_7 &= (A_{11} - A_{21})(B_{11} + B_{12}) \end{aligned}$$

f(n) = 7 f(n/2) + O(n^2) ➔ O(n^2.81)

SPEX 2.3-2.4:
Problem 3. Assuming $m \ge n$, give an algorithm to multiply an $m \times n$ matrix with an $n \times m$ matrix in $O(m^2 \cdot n^{0.81})$ time. (Hint: apply Strassen's algorithm to multiply $\lceil m/n \rceil^2$ pairs of order-n matrices.)

Problem 4. Assuming $m \ge n \ge t$, give an algorithm to multiply an $m \times n$ matrix with an $n \times t$ matrix in $O(m \cdot n \cdot t^{0.81})$ time. (Hint: apply Strassen's algorithm to multiply pairs of $t \times t$ matrices.)

sp2.3: 1.Divide the input matrices A and B into ⌈m/n⌉^2 pairs of order-n matrices. If m is not divisible by n, pad the matrices with zeros to make them divisible.
2. For each pair of matrices A_i and B_i, i ranges from 1 to ⌈m/n⌉^2: a. Apply Strassen's algorithm to multiply A_i and B_i, resulting in matrix C_i. b. Add matrix C_i to the corresponding block in the result matrix C. 4. Return C.
Eval: ⌈m/n⌉^2 pairs * O(n^2.81) = (m^2 n*0.81)
Sp2.4: ⌈m/t⌉⌈n/t⌉ * O(t^2.81)
Sp2.5: **merge k sorted arrays in O(nlogk)**. Ans: 1.build a min-heap with initial k 1st elements 2.remove min from heap and put in the final list. 3.add the 2nd one from the original array of the min (O(lgk)). (Each e is inserted and extracted once)

Activity Selection
Sort by ending time → greedily select next available

Next, we will prove that the algorithm returns an optimal solution. Let us start with a crucial claim.

Claim: Let $\mathcal{I} = [s, f]$ be the interval in S with the smallest finish time. There must be an optimal solution that contains $\mathcal{I}$.

Proof: Let T^* be an arbitrary optimal solution that does not contain $\mathcal{I}$. We will turn T^* into another optimal solution T containing $\mathcal{I}$.

Let $\mathcal{I}' = [s', f']$ be the interval in T^* with the **smallest** finish time. We construct T as follows: add all the intervals in T^* to T **except** $\mathcal{I}'$, and finally add $\mathcal{I}$ to T.

We will prove that all the intervals in T are disjoint. This indicates that T is also an optimal solution, and hence, will complete the proof.

Use induction to prove why this is optimal

SPEX 3.2: disprove greedily choose **shortest length**
[1,10] [8,12] [11,30]

SPEX 3.3: disprove fractional knapsack(sum(x1/wi*vi), for I in [1,n]: xi=min{W,wi}, W=W-wi) : (2,10),(3,30),W=4

Minimum Spanning Tree
Tree: connected undirected simple graph w/o cycles
G := (V, E); w : E → N^+; given any edge e in E, w(e)=weight
Spanning tree: tree that contains all vertices from V and uses only edges from G
Prim's Algorithm
S = set of vertices in current tree; V\S = set of remaining vertices; cross edge = edge connecting S and V\S
- Repeatedly take lightest **cross** edge (until |V|-1 edges)
Properties of tree
1) n-1 edges, connected, no cycles ⇔ a tree on n vertices
2) n edges, connected ➔ cycle
3) tree add an edge → cycle → remove one edge from cycle → tree

SPEX 4.1: prove for any two distinct nodes u, v, exist exactly one simple path(no vertex appear twice) from u to v.
if <1 → disjoint (defy connected prop)
if >1 → (a) exist node x s.t. u → x or x → v contain cycle
(b) there are two edges connecting u → v (anw exist cycle)
Proof – Induction base case
Claim: exist one MST that includes edges by Prim's
Let {a, b} = lightest edge, must returned by Prim's
Case 1: {a, b} in MST → done
Case 2: add {a, b} to MST → cycle formed → remove any one in the cycle → become tree again! (use 3rd prop of tree)
Tree formed here must smaller weight
Proof – Inductive
Suppose E MST that includes first I edges picked by Prim's
→ E MST that includes first i+1 edges picked by Prim's

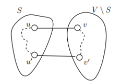
Inductive Case: Assuming that the claim holds for $i \le k-1$ ($k \ge 2$), next we prove that it also holds for $i = k$. Let $\{u, v\}$ be the k-th edge added by our algorithm, and S be the set of vertices already in the algorithm's tree before the addition of $\{u, v\}$. Without loss of generality, suppose that $u \in S$, and $v \notin S$.

By the inductive assumption, we know that there is some MST T that includes all the first $k-1$ edges. If T also includes edge $\{u, v\}$, then the claim already holds.

Next, we consider the case where T does not have $\{u, v\}$.

SPEX 4.4 (The cut property): if e is an S-cross edge that has min weight among S-cross edges, e must belong to MST.
Ans: If T contains e, the proof is done.
If T does not contain e, we add e to T and create a cycle.
We walk cross then stop after e′ that takes back to S. e′ is S-cross edge that has heavier or equal weight as e. Remove e′

We add $\{u, v\}$ to T, which creates a cycle. Let us walk on this cycle starting from v, cross $\{u, v\}$ into S, keep walking within S until traveling out of S for the first time. Let the edge that brought us out of S be $\{u', v'\}$.



Note that both $\{u, v\}$ and $\{u', v'\}$ are extension edges right before the moment our algorithm picks the k-th edge. Since $\{u, v\}$ has the smallest weight among all the extension edges, we know that the weight of $\{u, v\}$ is smaller than or equal to that of $\{u', v'\}$.

Now, remove edge $\{u', v'\}$ from T, which gives another tree T'. The cost of T' cannot be more than that of T. This means that T' must also be an MST.

We thus have proved that the claim holds for $i = k$ as well. □

O((|V|+|E|) log |V|): use min heap (deletemin O(log|V|)

Huffman Code
Encoding: maps each letter in alphabet to a binary string
Prefix-free rule: no letter's codeword is prefix of other letter's codeword
Prefix code: encoding scheme satisfying the prefix-free rule
Goal: produce prefix code w/ shortest avg length
Code tree on alphabet
Binary tree st: (a) every **leaf** in T ⇔ every **letter** in alphabet
(b) each internal node, left edge label 0, right edge label 1
For all codeword, length = level in code tree
Avg len = avg height of T = sum(a in alp) freq(a)*level(a)
Swapping: the two lowest frequency leaves (same parent) with the leaves x,y (same parent) which are the children of an arbitary internal node with the largest level in an opt T and get a T′. T′ is optimal as well

Proof: Huffman return optimal
Base case: n=2,one letter 0, one letter1 ➔optimal
Assume true for n=k-1,where k>=3, show it holds for n=k.
Let r1, r2 be 2 lowest frequency letters. There is an opt T with r1 and r2 have the same parent. R1 and r2 also have same parent in Thuff. Construct an alphabet′ by remove r1 and r2 and add a letter r* with freq(r1)+freq(r2). T′ be the tree by remove r1 and r2 from T. avg h of T = avg h of T′ +freq(r1)+freq(r2). Avg h of Thuff= avg h of Thuff′+freq(r1)+freq(r2). T′huff is opt, so T′huff<=T′

Proof: optimal prefix code tree must be full (int node has 2 children)
If some internal node had only one child then we could simply get rid of this node and replace it with its unique child. This would decrease the total cost of the encoding.

SPEX 5.2: alphabet w/ n letters, n is power of 2. **Prove prefix code constructed has avg len of at most⌈log(2,n)⌉**
Inductive: Assume it works for n= k, prove it works for n=2k. When n=2k, we can divide it into 2 k size subset S1 and S2, their avg length = ⌈log(2,k)⌉.Merge S1,S2, new avg length<=⌈log(2,k)⌉+1. ⌈log(2,k)⌉+1<=⌈log(2,2k)⌉<= ⌈log(2,n)⌉

SPEX 5.3: worst case O(n lg n) **implementation of Huffman**
use min heap to maintain S: delete/extract/insert: O(lg n)
repeat n-1 times: extract 2 min node, insert 1 node
→ 3 (n-1) * O(lg n) = O(n lg n)

SPEX 5.4: freq(i) is strictly higher than freq(i+1), **prove codeword of i cannot > that of i+1**. Ans: If the level of i and i+1 is the same, the proof is done. Else, let l1, l2 be the level of i and (i+1) respectively. T′ is the code tree where I and (i+1) is swapped. In T, avg height = summation of freq(σ)*level(σ) (without i and i+1) + l1*freq(i) + l2*freq(i+1). In T′, summation of freq(σ)*level(σ) (without i and i+1) + l2*freq(i) + l1*freq(i+1). If l1<l2, as freq(i)>freq(i+1), l1*freq(i) + l2*freq(i+1)< l2*freq(i) + l1*freq(i+1). Avg of T < that of T′. If l1>l2, T is not optimal➔contradiction. Codeword I is no longer than that of (i+1).

SPEX 5.5: n letters same freq, n is power of 2. Use Huffman, length of shortest codeword? Ans: balanced binary tree➔log(2,n) levels: x x x x x x x x → 2x 2x 2x 2x → 4x 4x → 8x full binary tree obtained → #bits = lg 2n

Dynamic Programming
Store result of repeatedly computed f(n) in array
opt(n) = max (i=1 to n) (P[i] + opt(n-i))
Memo method: improve recursion by using an array to remember the ans of subproblems (O(n^2)): Resolve subproblem f (1): O(1) time …… Resolve subproblem f (n): O(n) time, given f (1), ..., f (n − 1).
Cutting method: define bestSub(n)=k, i.e. 1st segment's length=k, the rest segments obtained from bestSub(n-k)

Given

$$opt(n) = \max_{i=1}^{n} (P[i] + opt(n-i))$$

define $bestSub(n) = k$ if maximization is obtained at $k = i$ (i.e., first segment having length k).

Example

length i	1	2	3	4
price $P[i]$	1	5	8	9

$bestSub(4) = 2$, $bestSub(3) = 3$, $bestSub(2) = 2$, $bestSub(1) = 1$

After we have computed $bestSub(i)$ for every $i \in [1, n]$, the best method for cutting up a rod of length n can be obtained in $O(n)$ time. (Think: why?)

For each $i \in [1, n]$, computing $bestSub(i)$ is no more expensive than computing $opt(i)$. This is left as a regular exercise.

We conclude that the rod cutting problem can be solved in $O(n^2)$ time.

Dependency
Consider function $f(i, j)$ defined for any $i \in [0, n]$ and $j \in [0, m]$:

$$f(i,j) = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ f(i-1, j-1) + 1 & \text{if } i, j > 0 \text{ and } x[i] = y[j] \\ \max\{f(i, j-1), f(i-1, j)\} & \text{if } i, j > 0 \text{ and } x[i] \ne y[j] \end{cases}$$

Subproblem f(i,j) may depend on one subproblem f(i-1, j-1) or two subproblems f(i, j-1) and f(i-1,j). Depends on if x[i] = y[j]. For any f (i,j), we compute it in O(1), given the outputs of the subproblems it depends on. We compute f (n,m) in **O(nm)** time
Dependency graph: A → B if do A before B
SPEX 6.1: fibonacci f(x) = f(x-1) + f(x-2), O(n) algo
d[0] = 0, d[1] = 1
for i in (2 to n): d[i] = d[i-2] + d[i-1]

SPEX 6.2:
Problem 2. Let A be an array of n integers. Consider the following recursive function which is defined for any i, j satisfying $1 \le i \le j \le n$:

$$f(i,j) = \begin{cases} 0 & \text{if } i = j \\ A[i] \cdot A[j] + \min_{k=i+1}^{j-1} f(i,k) & \text{if } i \ne j \end{cases}$$

Design an algorithm to calculate $f(1, n)$ in $O(n^3)$ time.

$$f(a,b) = \begin{cases} 0 & \text{if } a \ge b \\ (\sum_{c=a}^{b} A[c]) + \min_{c=a+1}^{b-1}\{f(a,c) + f(c,b)\} & \text{otherwise} \end{cases}$$

Design an algo to calc f(1, n) in $O(n^3)$

We will launch n rounds. In the i-th round (i ∈ [1, n]), we calculate all the f(a, b) satisfying 1 ⩽ a ⩽ b ⩽ n and b = a + i − 1. The strategy ensures that when f(a, b) is computed, f(a, c) and f(c, b) are ready for all c ∈ [a, b]. Hence, the computation of f(a, b) takes O(n) time. The total running time is $O(n^3)$ because there are $O(n^2)$ values to compute.

SPEX 6.5: rod-cutting modification: each cut costs c, dp algo to solve in $O(n^2)$
opt(n) = max(max (i=1 to n-1) (P[i] + opt(n-i)) + c), P[n])

Longest common subsequence (LCS)
Theorem: Let z be any LCS of x and y, and k the length of z.
If x[n] = y[m] then z[k] = x[n] = y[m] and z[1 : k − 1] is an LCS of x[1 : n − 1] and y[1 : m − 1].
If x[n] != y[n], then at least one holds: z is an LCS of x[1 : n − 1] and y ,or z is an LCS of x and y[1 : m − 1].
SPEX 7.3(find longest common substrings): create 2D array dp, where dp[i][j] represents the longest length of the substrings x[1:i] and y[1:j].use maxLen to store the current max length and use endIndexX to store the ending indice of the lc substring in x.
if x[i] == y[j]: dp[i][j] = dp[i-1][j-1] + 1 ; else: dp[i][j] = 0, if dp[i][j]>maxlen, update maxlen and endIndexX
lc substring can be obtained by extracting the substring from x starting at index (endIndexX - maxLen) and ending at endIndexX. The time complexity of this algorithm is O(mn)

SPEX7 Q4-5: find longest path in 2d array, where each cell assigned w/ a distinct number, only can go from point A to B if (1) they are neighbour (share an edge) (2) value at A < value at B solution: 1) build a graph where vertex A → B if can go from A to B. 2) perform longest path in DAG (note increasing path → acyclic) #nodes = n^2, #edge = at most 4n^2 = O(n^2) → total O(n^2)

Big Omega Proof
Lower bound on function
$f(n)$ is $\Omega(g(n))$ if there are positive constants C and k such that:
$f(n) \ge C * g(n)$ whenever $n > k$
$f(n)$ is $\Omega(g(n))$ is equivalent to $\exists C \exists k \forall n\, (n > k \to f(n) \ge C * g(n))$
To prove big-Omega, find *witnesses*, specific values for C and k, and prove $n > k$ implies $f(n) \ge C * g(n)$.

Big Oh Proof
Upper bound on function
- $f(n)$ is $O(g(n))$ if there are positive constants C and k such that: $f(n) \le C * g(n)$ whenever $n > k$
- $f(n)$ is $O(g(n))$ is equivalent to: $\exists C \exists k \forall n\, (n > k \to f(n) \le C * g(n))$
- To prove big-Oh, find *witnesses*, specific values for C and k, and prove $n > k$ implies $f(n) \le C * g(n)$.

EX1.5 D&C k-selection using randomization.
If n = 1, then we simply return the only element in S. For n > 1, we proceed as follows: Randomly pick an int v in S → get rank r of v in S. If r = k, return v. If r > k, produce an array S1′with elements that are < v. Recurse by finding the k-th smallest in S1′. Otherwise, produce an array S2′containing the elements> v. Recurse by finding the (r − k)-th smallest in S′.
Prove O(n) expected time.
Let f(n) be the expected time of the above algorithm on an input of size n. Clearly, f(0) = O(1) and f(1) = O(1).
Consider n > 1. The rank r of v is uniformly distributed in [1, n], namely, for each i ∈ [1, n], Pr[r = i] = 1/n. When r = i, it determines a "left subset" containing the i − 1 integers of S smaller than v, and a "right subset" of size n − i. In the worst case, we recurse into the larger of the two subsets, namely, we would need to solve the problem on an array of size max(i − 1, n − i). This gives rise to the following recurrence (for some constant α > 0):

$$\begin{aligned} f(n) &\le \alpha \cdot n + \frac{1}{n}\sum_{i=1}^{n} f(\max\{i-1, n-i\}) \\ &\le \alpha \cdot n + \frac{2}{n}\sum_{i=\lceil n/2 \rceil}^{n} f(i-1) \end{aligned}$$

We will prove that the recurrence leads to f(n) ⩽ cn for some constant c > 0. First, this is obviously true for n ⩽ 24 when c is at least a certain constant, say β (when n = O(1), the algorithm definitely finishes in constant time). Suppose that f(n) ⩽ cn for n ⩽ k − 1 where k ⩾ 24. Set t = ⌈k/2⌉.

$$\begin{aligned} f(k) &\le \alpha \cdot k + \frac{2}{k}\sum_{i=t}^{k} c(i-1) = \alpha \cdot k + \frac{2c}{k}\sum_{i=t-1}^{k-1} i \\ &= \alpha \cdot k + \frac{2c}{k}\frac{(k+t-2)(k-t+1)}{2} < \alpha \cdot k + \frac{c(k^2+3t-t^2)}{k} \\ &< (\alpha + c)k + 3c - c\frac{t^2}{k} \le (\alpha+c)k + 3c - c\frac{(k/2)^2}{k} \\ &= (\alpha+c)k + 3c - ck/4 \end{aligned}$$

We need the above to be at most ck, namely:

$$\begin{aligned} (\alpha+c)k + 3c - ck/4 &\le ck \\ \Leftrightarrow \alpha k + 3c &\le ck/4 \\ \Leftarrow \begin{cases} ck/4 \ge 2\alpha k \\ ck/4 \ge 6c. \end{cases} \\ \Leftarrow \begin{cases} c \ge 8\alpha \\ k \ge 24. \end{cases} \end{aligned}$$

Hence, setting c = max{β, 8α} completes the proof.

EX 2.4: find maximum subarray (A[x] in [-inf, +inf])
(a) O(n) find subarray end with n: scan from n downto 1, maintain sum of A[i..n], use max of them as ans
(b) O(n lg n) find max subarray:
1) break into halves → 2*f(n/2) find max subar in each half
2) consider max suffix subarray of left + max prefix subarray of right (O(n))
Return max of f(left), f(right), left suffix + right prefix
f(n) <= 2 * f(n/2) + O(n) = O(n lg n)

EX 2.5: matrix mul w/ n != power of 2, ensure $O(n^{2.81})$
If n is not a power of 2, let m be the smallest power of 2 that is > n. If A, B are the n × n input matrices, obtain an m × m matrix A′ by padding m − n dummy rows and columns to A containing only 0 values, and similarly, an m × m matrix B′ from B. Calculate A′B′ in $O(m^{2.81}) = O((2n)^{2.81}) = O(n^{2.81})$ time. AB can be obtained by discarding the last m − n rows and columns from the matrix A′B′ .

EX 3.4: activity selection disproof greedy
Let S be the input set of intervals. Initialize an empty T, and then repeat the following steps until S is empty:
- (Step 1) Add to T the interval I ∈ S that **overlaps with the fewest other intervals** in S.
- (Step 2) Remove from S the interval I as well as all the intervals that overlap with I.

Finally, return T as the answer.
Prove: the above algorithm does not guarantee an optimal solution.
S = {[1, 10], [2, 22], [3, 23], [20, 30], [25, 45], [40, 50], [47, 62], [48, 63], [60, 70]}

Past paper: Determine the rank of a1=A1[n/2] and a2=A2[m/2], where A1, A2 are sorted, suppose a1<a2. a1 is greater than (n/2) − 1 elements in A1 and at most (n/2) − 1 elements in A2; hence, its rank in S is at most 1 + (n/2) − 1 + (n/2) − 1 = n − 1. b1 is greater than the first n/2 elements in A1 and the first (n/2) − 1 elements in A2; hence, its rank in S is at least n. **Find k-smallest number of 2 sorted array in O(logn + logm) time.** If n = 1, then we compare A[1] with B[k]. If A[1] < B[k], return min{B[k − 1],A[1]}; else, return B[k]. Otherwise, consider n ⩾ 2 and m ⩾ 2. Let a = A[n/2] and b = B[m/2]. Assume a < b. Rank(a) in A∪B is at most (n+m)/2, and rank(b) is at least (n+m)/2. If k<=(n+m)/2, none of the elements in B[m/2+1 : m] can be the final answer. We recurse on A, B[1 : m/2], and k. If k > (n + m)/2, none of the elements in A[1 : n/2] can be the final answer. We recurse on A[1+n/2 : n], B, and k−n/2. spend O(1) constant time before entering recursion. Each time we recurse, either A shrinks in half or B does.

Past paer(Dynamic programming for coins)
1. When d1 = 4 and d2 = 3. The algo is not optimal for n = 6.
2. Take an arbitrary optimal solution that uses x′1, x′2, and x′3 coins of d1, d2, and d3, respectively. Hence:
5x1 +2x2 +x3 = 5x′1 +2x′2 +x′3 (1) We will show
4x1+x2 ⩾ 4x′1+x′2. (2)
Plugging (2) into (1) yields: x1 + x2 + x3 ⩽ x′1 + x′2 + x′3 , which indicates that {x1 , x2 , x3 } is optimal.
To prove (2), first observe that x1 ⩾ x′1 (because otherwise 5x′1 ⩾ 5(x1 + 1) > n). We distinguish two cases:
Case 1: x1 = x′1. We must have x2 ⩾ x′2 because otherwise 2x′2 +x′3 ⩾ 2(x2 +1)+x1 > n. It follows that (2) holds.
Case 2: x1 > x′1. It suffices to prove x′2 ⩽ 4 because this will yield 4(x1 − x′1) + x2 ⩾ 4 ⩾ x′2, which then gives (2).
To prove x′2 ⩽ 4, observe that if x′2 ⩾ 5, we can replace 5 coins of 2 dollars with 2 coins of 5 dollars, contradicting the optimality of {x′1, x′2, x′3}.

Past paper (find k largest numbers in S with size n in O(nlogk)) If k = 1, simply return the maximum element in S in O(n) time. Otherwise, spend O(n) time finding the median e of S (i.e., the element with rank n/2 in S). Divide S into S1 ={e′ ∈S|e′ ⩽e}andS2 ={e′ ∈S|e′ >e},which can also be done in O(n) time. Recursively find the (k/2)-split set T1 of S1 and the (k/2)-split set T2 of S2. Return T1∪T2.
f(n,1)=O(n) f(n,k)=O(n)+2f(n/2,k/2)

longest strictly increase subsequence: for i from 1 to n: use dp[i] to remember the current length (should include nums[i] itself). Iterate all nums[j] where j is from 1 to i-1, if nums[j]<nums[i]: use maxlen to record the longest length before nums[i], and dp[i]=maxlen+1 ---O(n^2)

envelope 套娃(O(nlogn))：[w,h] first sort the array with w in ascending order, when there are same w, sort by h with descending order. Then use lsisub to find the best subseq