CMPS 6610 Problem Set 04

Answers

Name:_ Aaron Dumont_____

Place all written answers from `problemset-04.md` here for easier grading.

- **1d.**

File	Fixed Cost	Huffman Cost	Ratio (Huffman/Fixed)
F1.txt	1340	826	0.6164
alice29.txt	1039367	676374	0.6508
asyoulik.txt	876253	606448	0.6921
fields.c	78050	56206	0.7201
grammar.lsp	26047	17356	0.6663

- I do see a consistent trend. In all 5 files, the Huffman coding cost is significantly less than the fixed-length coding cost. The ratio varies between 0.62 and 0.72 (being well below 1). The mean ratio is ~0.67 or 2/3. This trend exists because the character frequencies in the 5 files are not uniform. The files contain both natural language text and source code wherein many characters (like 'e', 't' or ' ') appear frequently and other characters (such as 'z', 'q' or '}') appear rarely. The Huffman algorithm, as opposed to fixed-length encoding, exploits the aforementioned characteristic by assigning very short bit codes to the frequent characters and longer bit codes to the rare characters. Consequently, this results in a lower weighted average cost compared to the fixed-length code (which uses the same number of bits for all characters regardless of their frequency).

- **1d.** If Huffman coding is used on a document with alphabet $\sum$ in which every character had the same frequency, the Huffman algorithm would build a perfectly balanced binary tree (or as balanced as possible). Every leaf node (character) would be at the same depth or at depths that differ by at most 1 (eg. if number of characters is not a perfect power of 2). The length of every character's code would be $\lceil \log_2 k \rceil$, where k is the number of unique characters in the alphabet. This code length would be ***identical to the number of bits required for a fixed-length encoding. Hence, the expected cost of the Huffman encoding would be exactly the same as the cost of a fixed-length encoding. This would be consistent across documents that have a uniform character frequency distribution.*** The total cost will always be $(TotalCharacters) \times \lceil \log_2 k \rceil$.

- **2a.** Given an array A of size n elements, we will treat this as an almost-complete binary tree and enforce the heap property from the bottom up. In a binary heap stored as an array, a parent (index i) has children at $2i+1$ and $2i+2$ and the heap property requires every parent $\leq$ its children. We can start at the end by identifying the last non-leaf-node in the tree.

Our array size is n elements with indices 0 to n-1. The last parent (non-leaf node) will be the **largest index** i whose left child $(2 * i + 1)$ is still inside the array. Hence, $2*i+1 \leq n-1$. Solve for i. $i \leq (n-2)/2$ or $i = [n/2]-1$

Hence, the last non-leaf node in the tree should be at index $i = [n/2] - 1$. All nodes after this are valid mini-heaps. We can then iterate backward looping from $i = [n/2] - 1$ down to the root (index 0). We can implement the operation **sift-down** or **heapify** operation. This operation assumes the subtrees rooted at the left and right children of i are already valid mini-heaps. It compares the node i with its children. If $A[i]$ is larger than its smallest child, it is swapped with the smallest child. This process is recursively executed (the node "sifts down") until the node i is smaller than both of its children or it becomes a leaf.

Work analysis:

The work of sift-down is proportional to the height of the node, not the height of the whole tree and the vast majority of nodes are near the bottom of the tree. Only one node (the root) is at height $logn$ and this is the only node that the full $O(logn)$ work is done on. The total work $W(n)$ is the sum of work done at each height k:

The total work for a single sift-down call starting at a node with subtree height k is: ** Total Work = (work per step) $\times$ (Max number of steps) = $O(1) \times k$ and is therefore $O(k)$.

$$W(n) = \sum_{k=0}^{\log n} (\text{number of nodes at height } k) \times O(k)$$

$$W(n) \approx \sum_{k=0}^{\log n} \left(\frac{n}{2^{k+1}}\right) \cdot O(k)$$

This sum $O\left(n \sum_{k=0}^{\log n} \frac{k}{2^k}\right)$ converges to a constant multiple of n. Therefore, the total work is $O(n)$.

- **2b.**

  This algorithm is a single for loop that runs $n/2$ times eg.

  for i from $(n/2-1)$ down to 0

  sift-down(A, i)

  In a purely sequential algorithm, the span, $S(n) = O(n)$, the same as work.The span of a sequential for loop is the sum of the spans of each iteration. The span of one sift-down(i) is $O(k_i)$, where k_i is the height of the subtree at i. The total span is $\sum O(k_i)$. This sum is the exact same $O(n)$ for the work analysis.

However, if we perform the sift-down operation in parallel for each node at the same level this alters the span calculation. The algorithm would then proceed in sequential phases, starting from the bottom of the tree (the leaves) and moving up to the root.

The height of the tree, k, as mentioned is $\log n$. At each level, the nodes of the binary tree structure can be heapified in parallel ($O(1) + O(2) + \cdots + O(\log n)$). Therefore,

Total Span Calculation:

Since these $\log n$ phases must run sequentially, the total span is the sum of their individual spans:

$$\text{Total Span} = O(1) + O(2) + O(3) + \cdots + O(\log n)$$

$$\text{Total Span} = O\left(\sum_{k=1}^{\log n} k\right)$$

- This is the sum of an arithmetic series, which is $\frac{k(k+1)}{2}$. Substituting $k = \log n$:

$$\text{Total Span} = O\left(\frac{\log n(\log n + 1)}{2}\right) = O(\log^2 n)$$

- **3a.** Given that we wish to exchange N dollars for local currency where local currency is only in coins with denominations of powers of 2 [$D = 2^0, 2^1, \ldots, 2^k$], we can construct a greedy algorithm producing the fewest coins as follows:
- Start with an empty set of coins $\emptyset$
- Find the largest coin denomination 2^i in D that is less than or equal to the remaining amount N ($2^i \leq N$).
- Add this coin 2^i to your set.
- Subtract the value from the remaining amount: $N = N - 2^i$.
- Repeat the above steps until $N = 0$.

  This algorithm is equivalent to finding the binary representation of N. Each '1' in the binary string $N = b_k \ldots b_1 b_0$ corresponds to one coin value 2^i where $b_i = 1$.

  Suppose I have \$29 dollars to exchange.

  Possible coins in powers of 2 for the exchange include $\{16, 8, 4, 2, 1\}$

  Greedy Coins selected through our algorithm: $\{16, 8, 4, 1\}$

$$\begin{array}{lccccccc} \text{Binary '1's:} & 1 & 1 & 1 & 0 & 1 & & \\ & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & & \\ & 16 & +8 & +4 & +0 & +1 & = 29 \end{array}$$

- Hence, the set of coins we pick using the greedy algorithm is identical to the set of powers of 2 that have a $'1'$ in the binary representation.
- **3b.** To prove this algorithm is optimal, we must prove the greedy choice and optimal substructure properties.

  1. Proof of Greedy Choice Property

  Let 2^k be the largest coin denomination relevant to this problem such that $2^k \leq N$. There exists an optimal solution for N that includes at least one coin of value 2^k. The general intuition here is that if we don't take the biggest possible coin, we will be forced to make up its value using a combination of smaller coins; for coins of power-of-2, any combination of smaller coins is always a worse deal (ie. requires more coins) than just taking the one big coin first. We will show this with a proof by contradiction.

  Proof

  1. Let S be any optimal solution for N. We know that $N < 2^{k+1}$. If we go back to our example above, we wanted to exchange $N = 29$ and possible coins were $16, 8, 4, 2, 1$. Coin $2^k = 16$ and coin $2^{k+1} = 32$ which exceeds N. Therefore S can contain one coin of value 2^k at most (as two would be $2 \cdot 2^k = 2^{k+1}$, which is greater than N).
  2. We will assume that S does not contain a 2^k coin. All coins in S must therefore have a value of 2^{k-1} or less.
  3. As S must sum to N and $N \geq 2^k$, all of the coins in S must sum to at least 2^k.
  4. We know that $2^k = 2 \cdot 2^{k-1}$. hence, any set of coins with values $\leq 2^{k-1}$ that sums to 2^k must contain at least two coins. For example two 2^{k-1} coins or four 2^{k-2} coins, etc.
  5. We can take a subset of coins from S that sums to 2^k such as two 2^{k-1} coins and replace them with a single 2^k coin. This new solution we will call S'. S' now sums to N but has one fewer coin than S. This ***contradicts*** our assumption that S was the optimal solution (with the smallest number of coins). Therefore, our assumption above must be false. Any optimal solution

S must contain the greedy choice (one 2^k coin) and we have proven the greedy choice property.

2. Proof of Optimal Substructure Property

An optimal solution for N contains an optimal solution for the subproblem $N' = N - 2^k$, where 2^k is the greedy choice. The general intuition for this proof is that we cannot build an optimal solution for a big problem out of a sub-optimal solution for a small problem. We will do this using a "swap" proof/proof by contradiction.

Proof

1. Let S be an optimal solution for N. Based upon the proof above for Greedy Choice Property, we know S must contain one 2^k coin. Let $S' = S - 2^k$. The set S' is a solution for the subproblem $N' = N - 2^k$, and the total number of coins is $|S| = 1 + |S'|$.
2. We will now prove by contradiction that S' must be optimal for N'. Assume S' is ***not*** an optimal solution for N'. This means there must be some other solution, S'', for N' that uses fewer coins, so $|S''| < |S'|$. If S'' exists, we could construct a new solution for the original problem N by taking $S_{new} = S'' \cup \{2^k\}$.
3. The size of this new solution would be $|S_{new}| = |S''| + 1$. As stated above, we assumed $|S''| < |S'|$, so it follows that $|S''| + 1 < |S'| + 1$. which means $|S_{new}| < |S|$. This, however, contradicts our initial assumption that S was the optimal solution for N. Therefore, S' must be an optimal solution for the subproblem N'.

Since both the proof of greedy choice property and optimal substructure property hold, the greedy algorithm we developed is optimal. - **3c.** The input, or the amount of money to exchange, is N. The available denominations of coins are $[D = 2^0, 2^1, \ldots, 2^k]$, where k is the largest integer such that $2^k \leq N$. Taking $\log_2$ of both sides means $k = [\log_2 N]$. The number of denominations we must consider, $m = k + 1$, is therefore $O(\log(N))$.

For any denomination $c = 2^j$, if we use it once, the remaining amount $A' = A - 2^j$. Since we only consider c if $A < 2^{j+1}$ (as \$2ˆ{j+1} would have been taken in the previous step), the new amount $A' = A - 2^j < 2^{j+1} = 2^j$. The new amount is now strictly less than c, so the coin 2^j can never be used more than once. Therefore, this is

equivalent to finding the binary representation of N. Some pseudocode could be:

1. $k = floor(\log_2(N))$
2. Amount = N
3. Solution = []
4. for j from k down to 0:
5. $c = 2^j$
6. if Amount $\geq c$
7. Solution.add(c)
8. Amount = Amount -c
9. return Solution

Work Analysis Setup is performed in lines 1-3. Calculating the integer log of N takes $O(\log(N))$ time while lines 2 and 3 are run in constant time. Therefore $W_{setup} = O(\log(N))$

In the loop in lines 4-8, the loop iterates from $j = k$ down to $j = 0$. The total number of iterations is $k + 1$. Since $k = [\log_2 N]$, the number of iterations is $O(\log(N))$. The work in each iteration is constant time $O(1)$ such as dividing by 2, comparing Amount to c, appending to a list and subtraction.

The total work is therefore the sum of the two above. $W_{total} = O(\log(N)) + O(\log(N)) = O(\log(N))$. **The total work is $O(\log(N))$ as the algorithm is dominated by a single loop that runs $O(\log(N))$ times, with each iteration performing a constant $O(1)$ amount of work.

Span Analysis As we have discussed previously, span is the longest chain of sequential dependencies. This is a sequential algorithm. If we look at the iterations, the calculation in iteration $j-1$ requires the result from iteration j and hence, iteration $j-1$ cannot start until iteration j is finished. There is no opportunity for parallelism as presented in the algorithm so $S_{total} = O(\log(N)) + O(\log(N)) = O(\log(N))$.

- **4a.** The greedy algorithm we devised for Geometrica does not work in Fortuito. We will present a counterexample below to corroborate this.

We must find the minimum number of coins to make change for amount N using an arbitrary set of k denominations $D = D_0, D_1, ..., D_{k-1}$.

Counter Example Suppose we have $N = 6$. The Denominations include: $D : 1, 3, 4$

The Greedy solution we developed for Problem 3 proceeds as follows: - Take

coin 4, remainder = 2 - Take coin 1, remainder = 1 - Take coin 1, remainder = 0 - This takes a total of **3 coins**

The optimal solution proceeds as follows: - Take coin 3, remainder = 3 - Take coin 3, remainder = 0 - This takes a total of **2 coins**

Therefore, this counterexample proves that the greedy algorithm does not always produce the fewest number of coins in its solution. This problem lacks the greedy choice property.

- **4b.** The optimal substructure property states that an optimal solution can be constructed from optimal solutions of smaller subproblems.

Stating this property in the context of this problem: Let $Optimal(N)$ be the minimum number of coins required to make change for an amount N. An optimal solution for $Optimal(N)$ must use some first coin, D_i. The remaining $Optimal(N) - 1$ coins in that solution must themselves form an optimal solution for the subproblem of making change for the remaining amount, defined as $N - D_i$.

Proof We will perform a proof by contradiction. - 1. We will assume we have an optimal solution for N, which we will denote as S_N. This solution contains $Optimal(N)$ coins. S_N consists of a first coin, D_i and a set of remaining coins, S', that sum to $N - D_i$. It thus follows that the size of the solution is $1 + |S'|$. - 2. We will now assume, in contradiction to the above, that S' is ***not*** an optimal solution for the subproblem of $N - D_i$. Given this, there exists some of other set of coins, S'', that also sums to $N - D_i$ but has fewer coins - that is, $|S''| < |S'|$. - 3. If the assertions in 2. were true, we could create a *new* solution for N by combining the first coin D_i with the set S''. The total number of coins would be $1 + |S''|$. Given that $|S''| < |S'|$ as stated above, then adding 1 to each side of the inequality, $1 + |S''| < 1 + |S'|$. This means that our new solution, S'' has fewer coins than our original "optimal" solution S_N. This is a contradiction.

> Therefore, our assumption in 2. above must be false. The set of remaining coins S' must be an optimal solution for the subproblem $N - D_i$ and this problem has an optimal substructure property.

- **4c.** Using the optimal substructure property, we have created a dynamic programming algorithm using top-down memoization to avoid recomputing solutions to subproblems. We can create an algorithm comprised of three components:

1. memo function - this function adds a memory or checks the memory table before the recursive function runs to ensure that the same subproblem is only solved once. This function checks the memo table for a given input. If it finds a result it returns the result, and if it finds nothing it runs the computation function, stores the new result and then returns that result to avoid re-computing the same problem.

2. recursive function - this core recursive function contains the logic for solving the change-making problem. It defines the problem's optimal substructure. It is called with a state (i, n) where i is the number of coin types we are allowed to use and n, the amount of change we are trying to make. The function works by checking cases:

- Base case 1: (case (_,0)) - the problem is solved. The cost to make 0 is 0 coins. It returns 0.
- Base case 2: (case (0, _)) - We have 0 coin types left but n is still greater than 0 so it is impossible to make change. It will return infinity.
- Recursive Case: (case _) - All other situations. We have i coins and need to make amount n. The function should solve this by finding the *minimum* of 2 smaller, recursive subproblems. It gets the current coin's value, coin_w.
    - Choice 1 - Skip. This skips the ith coin and solves the problem for the same amount n, but only using the first $i - 1$ coins.
    - Choice 2- Take. This asks what is the minimum number of coins if the ith coin is used. It first checks in $n \geq coin_w$. If not, it can't take the coin and this choice is inifinity. If it can take the coin, it calculates 1+ (the solution to a new subproblem). »- The 1 counts the coin we just took. »- The new subproblem is $(i, n - coin_w)$. We must solve for the *remaining amount* $(n - coin_w)$. We must pass i (not $i - 1$) as this is an unbounded problem where we are allowed to re-use the *ith* coin.
    - **Final Result**: The function returns the min of Choice 1 (Skip) or Choice 2 (Take).

3. wrapper function - to initialize the memo table and start the process. It gets the total number of denominations, k. It creates a new, empty memo table and then calls the recursive function on the full-sized problem "solve for k coins and amount N". It then returns the final answer.

- A simplified pseudocode is as follows:

// D is the array of k coin denominations {D[0], D[1], ... D[k-1]} // k is the number of denominations // N is the target amount

// 1. Create the memoization table, size (k+1) x (N+1) memo_table = new 2D array, initialized to -1

// 2. Define the main recursive function function solve_change(i, n):

```
// --- Base Cases ---

// Base Case 1: Success!
```

```
if n == 0:
    return 0

// Base Case 2: Failure (went past 0 or ran out of coins)
if n < 0 or i == 0:
    return infinity // "infinity" means "impossible"

// --- Memoization Check ---

// If we've already solved this, return the stored answer
if memo_table[i][n] != -1:
    return memo_table[i][n]

// --- Recursive Step ---

// Get the current coin's value (using i-1 for 0-indexing)
coin_w = D[i-1]

// Choice 1: "Skip" this coin
// Solve using i-1 coins for the same amount n
choice_skip = solve_change(i-1, n)

// Choice 2: "Take" this coin
choice_take = infinity // Assume we can't take it
if n >= coin_w:
    // We can take it. The cost is 1 + the solution for the remaining amount.
    // We pass 'i' (not i-1) because we can re-use this coin.
    choice_take = 1 + solve_change(i, n - coin_w)

// --- Store and Return ---

// The best solution is the minimum of the two choices
result = min(choice_skip, choice_take)

// Store this new result in our "memory"
memo_table[i][n] = result

return result
```

// 3. Start the algorithm by solving the original, full problem final_answer = solve_change(k, N)

Work and Span Analysis - **Work**: The work will be calculated by the number of subproblems multiplied by the work per subproblem. »- Number of subproblems: The state is (i, n), where i ranges from $0tok$. This gives $O(k) \times O(N) = O(kN)$ distinct subproblems (nodes in the DAG). »- Work per subproblem: The work inside the recursive function (assuming memoized calls

are $O(1)$) involves a few comparison, an array lookup, an addition and a min operation which can all be done in constant time ($O(1)$).

- **Span**: The span is the longest path in the recursion DAG. »- A subproblem (i, n) depends on $(i-1, n)$(the "Skip" call) and $(i, n - coin_w)$(the "Take" call). The longest path must trace from the start (k, N) to a base case like $(0, 0)$. This requires k "Skip" steps to get i to 0 and (in the worst case of a \$1 coin) N take steps to get n to 0. These dependencies form a chain and therefore the longest path is O$(k + N)$.

  Total work: $O(kN) \times O(1) = O(kN)$.

- **5a.**

- Yes, the optimal substructure property appears to hold for weighted task selection. We will use a proof by contradiction.

- 1. Assume the tasks $A = \{a_0, ..., a_{n-1}\}$ are sorted by their finish times. Let *opt* by an optimal solution (a set of compatible tasks with maximum total value) for the full set of tasks. This task a_i has a value, v_i. As a_i is in the solution, no other task in *opt* can conflict with it. Therefore, all other tasks in *opt* must finish before a_i starts. -2. We will now define a subproblem which is to find the optimal solution for all tasks in A that are compatible with a_i. Let opt' = *opt* - $\{a_i\}$ be the rest of our optimal solution. The value of the solution is $value(O) = value(O') + v_i$. -3. Now assume that opt' is not an optimal solution for this subproblem. This means that there must be some other set of compatible tasks S' whose total value is greater then opt' (ie. $value(S') > value(O')$). »»- If this were true, we could create a new solution $S = S' \cup \{a_i\}$. This new solution S would be valid (since all tasks in S' are compatible with a_i). »»- Its total value would be $value(S) = value(S') + v_i$. Since we assumed $value(S') > value(O')$, it follows that $value(S) > value(O')$. This means that $value(S) > value(O)$ which contradicts our initial assumption that *opt* was the optimal solution.

- Therefore, our assumption must be false. opt' must be an optimal solution for the subproblem of tasks compatible with a_i. This proves that an optimal solution to the problem contains optimal solutions to its subproblems.

- **5b.**

  The greedy choice property does not hold for this problem. Greedy criteria will fail to produce an optimal solution. Two counter examples to lend credence to this assertion include:

1. Greedy by Highest Value (v_i):

   A greedy algorithm might pick the task with the greatest value first.

- Tasks: »- a_1: (start=0, finish=10, value=100) »- a_2: (start=1, finish=3, value=40) »- a_3: (start=3, finish=5, value=40) »- a_4: (start=5, finish=7, value=40) »- Greedy Choice: The algorithm picks a_1 (value 100), as it's the highest. This task conflicts with a_2, a_3, and a_4. »- Greedy Solution: $\{a_1\}$, Total Value = 100. »- Optimal Solution: $\{a_2, a_3, a_4\}$, Total Value = 120.
- **Conclusion: Greedy by value fails**.

2. Counterexample: Greedy by Earliest Finish Time (f_i)

   Although this strategy may work for the unweighted problem, it fails here. - Tasks: - a_1: (start=0, finish=5, value=10) - a_2: (start=4, finish=10, value=100) - Greedy Choice: The algorithm picks a_1 first (finish time 5). This task conflicts with a_2. - Greedy Solution: $\{a_1\}$, Total Value = 10. - Optimal Solution: $\{a_2\}$, Total Value = 100.

- **Conclusion: Greedy by earliest finish time fails.**

  With these two counterexamples, we have shown that the greedy property choice does not hold for this problem.

- **5c.** In this problem, we have proven that we have an optimal subtructure property but not the greedy choice property so dynamic programming should be a great fit to solve this. We can use a top-down memoization approach.
- We will sort the tasks by finish time. Then for each task i we can decide whether to "take" it or "skip" it.
- We will use a wrapper function to setup and a recursive function to compute the maximum value obtainable from tasks a_0 through a_i.

1. Wrapper function - this performs the setup by sorting and pre-computing values.

   function find_max_value(Tasks):

   // 1. Sort tasks by finish time (f_i) A = sort_tasks_by_finish_time(Tasks) n = length(A)

   // 2. Pre-compute the last compatible task array `p` // p[i] = largest index j < i such that A[j].finish_time <= A[i].start_time // (This can be found in O(n log n) time using binary search) p = pre_compute_compatible_tasks(A)

   // 3. Create the 1D memo table memo_table = new 1D array of size n, initialized to -1

   // 4. Call the recursive function to solve for all tasks // (We pass n-1, the index of the last task) return solve_tasks_recursive(n-1, A, p, memo_table)

2. Recursive Function: This the recursive, memoized function that computes the maximum value obtainable from tasks a_0 through a_i.

   function solve_tasks_recursive(i, A, p, memo_table):

   // — Base Case — // If we have no tasks (index < 0), the total value is 0. if i < 0: return 0

   // — Memoization Check — if memo_table[i] != -1: return memo_table[i]

   // — Recursive Step (modeled on Knapsack's recurrence) —

   // Choice 1: "Skip" task i // The value is the best we could do with the first i-1 tasks. choice_skip = solve_tasks_recursive(i-1, A, p, memo_table)

   // Choice 2: "Take" task i // The value is v[i] + the optimal solution for all tasks // compatible with i (which is found at index p[i]). choice_take = A[i].value + solve_tasks_recursive(p[i], A, p, memo_table)

   // — Store and Return —

   // The optimal solution is the MAX of these two choices. result = max(choice_skip, choice_take) memo_table[i] = result

   return result

Work and Span Analysis:

Work: The work of an algorithm is its total number of operations from start to finish and for this particular alogrithm outlined above it considers the Setup and Dynamic Programming (solving the problem).

1. Setup work: This is the preprocessing we must perform before the recursive dynamic programming solution can be formulated. It is comprised of 2 parts: »a. Sorting: We will sort by finish times and will use heap sort which we have proven before takes $O(nlogn)$ operations. »b. Pre-computing the p array: It will store the index j of the last tasks (of the sorted list) that finishes before task i starts. This essentially finds the last compatible task for each of the n tasks. This can be done using a binary search on the sorted list. We have previously shown that binary search is $O(nlogn)$.
2. Dynamic Programming Work: »a. Number of Subproblems: The state is represented by i, which ranges from 0 to $n-1$ yielding $O(n)$ distinct subproblems. »b. Work per Subproblem: The work inside each call is constant time: one max operation, one addition and two table lookups. This is $O(1)$.

- Overall work: $W_{total} = O(nlogn)$ (Sort) + $O(nlogn)$ (Pre-compute) + $O(n)$ (Dynamic Programming) = $O(nlogn)$.
- Hence $W_{total} = O(nlogn)$.

Span: Span is the longest chain of dependent operations and the total span for this problem is the span of its sequential parts added together: $TotalSpan = Span(Setup) + Span(DP)$

1. Setup Span: This phase has 2 steps that must be done in order: sorting then pre-computing. »a. Sorting: We will use HeapSort. HeapSort's span is dominated by the extraction of n items one by one where it creates a dependency chain of n steps where each step (ie. a $heapify$) has a span of $O(logn)$. The total span is therefore $O(nlogn)$. »b. Pre-compute: We use binary search to find $p[i]$ for each task i. Finding $p[i]$ is completely independent of finding $p[j]$ which means all n binary searches can be executed in parallel, with each search comprised of a span of $O(logn)$.

   Therefore the total setup span = $O(nlogn) + O(logn) = O(nlogn)$.

2. Dynamic Programming Span: »Longest Path in DAG: The solve_tasks_recursive(i) function makes a recursive call to solve_tasks_recursive(i-1) (the "Skip" choice).This creates a sequential dependency chain from $i = n - 1$ down to 0. To solve for $n - 1$, you must wait for $n - 2$, which must wait for $n - 3$, and so on.This path, $(n - 1) \to (n - 2) \to \cdots \to 0$, has a length of n. The longest path is therefore $O(n)$.

   Span per Node: The work inside each function call (assuming memoized lookups are $O(1)$) is just a few comparisons, lookups, an addition, and a max operation. This is all constant time, so the span is $O(1)$.

   Total DP Span = $O(n) \times O(1) = O(n)$.

Hence, $Span_{total} = O(nlogn) + O(n) = O(nlogn)$.

Note: If I used MergeSort which has a span of $O(log^2 n)$, the span of the dynamic programming part would dominate and the overall total span would be $O(n)$.