张文超 202223338 8

Problem 1

Think of the houses as points $H = \{H_1, \cdots, H_n\}$ on a line in that order from left to right, and the wells as same other k points $P_1, \cdots P_k$ on the same line. If each point H_i is within 4 miles of a certain point P_j, the scheme is available.

We can use the greedy strategy: put P_1 exactly four miles to the right of H_1, remove all the houses covered by P_1, that is, within 4 miles of P_1, and then recursively solve the sub-problem containing the rest of the houses. Call the rest of the houses as H. Let $P = (P_1, \cdots, P_k)$ be the solution returned by the algorithm.

① There exists an optimal solution O which puts a wells at P_1. Let $O = \{P_1, \cdots P_m\}$ be any optimal solution. Clearly P_1 cannot be to the right of P_1, or ~~she~~ else H_1 is not covered. Thus, the set of houses P_1 covers (all houses within 8 miles to the right of H_1) contains

#

the set of houses P_1 covers. Consequently, $O = \{P_1, P_2, \cdots, P_m\}$ is a feasible solution which is optimal since it has as many ~~a~~ wells as O.

② Let $O = \{P_1, P_2, \cdots, P_m\}$ is an optimal solution which contains P_1, the $D = \{P_2, \cdots, P_m\}$ is optimal for the sub-problem H containing houses that P_1 does not cover.

If there is a better solution to the sub-problem H, then that solution along with P_1 would be feasible (all of H are covered) and would be better than O, ~~vit~~ violating the optimality of O.

The cost of the solution ~~k~~ is $1 + (k-1) = 1 + OPT(H) = 1 + cost(o) = cost(D) = OPT(H)$. The first equality follows from the induction hypothesis that the algorithm for H. The second equality follows from ②. The third follows from the defination of O in the ②. If the H_i are already sorted, the algorithm runs in time $O(n)$. If not, $O(n \lg n)$ is needed.

Problem 2

The problem is a variant of the weighted shortest job first (WSJF) algorithm, which is used to ~~pro~~ ~~prep~~ prioritize jobs based on their cost of delay and duration. The cost of delay measures the economic impact of not completing a job ~~in~~ on time, while the duration mearuses how long it takes to complete a job.

We could define:

Cost of delay = weight × finishing time

Druation = ~~leo~~ length

~~So to~~ Score = (weight * ~~king~~ finishing time) / length

The intuition behind this formula is that jobs with higher ~~sco~~ scores have higher ~~a~~ urgency and value per unit of time, so they should be done first.

~~To find the optimal schedule~~, to prove the algorithm is correct, we can use a contradiction argument. Suppose there exists another ~~st~~ schedule that has a lower weighted sum than the one ~~p~~ produced by WSJF. This means that there must be two adjacent jobs in ~~the~~ this schedule that are swapped compared to WSJF. Let i and j be these two jobs, such that i comes before j in WSJF but after j in the other schedule.

Let S_i and S_j be their scores according to WSJF. Since i comes before j in WSJF.

$$C_i = t_i \qquad C_j = t_i + t_j$$

Let D_i and D_j be their completion times according to the other schedule. Since j comes before i in the other schedule,

$$\cancel{D_i = t} \quad D_j = t_j \qquad D_i = t_j + t_i$$

$$\text{Difference} = (W_i * D_i + w_j * D_j) - (W_i * C_i + W_j * C_j)$$
$$= W_i * t_j - w_j * t_i$$

Difference > 0 since $S_i > S_j$ implies $w_i / t_i > w_j / t_j$ implies $w_i * t_j > w_j * t_i$

This shows that swapping i and j increases the weighted sum instead of decreasing it, ~~cond~~ contradicting our assumption. ~~that~~

Therefore, WSJF produce an optimal schedule for minimizing the weighted sum of completion times.

Problem 3

If $x_1 \leq x_2 \leq \cdots \leq x_n$ and $y_1 \leq y_2 \leq \cdots \leq y_n$

then $x_n y_1 + x_{n-1} y_2 + \cdots + x_1 y_n \leq x_{\sigma(1)} y_1 + x_{\sigma(2)} y_2 + \cdots + x_{\sigma(n)} y_n$

$$\leq x_1 y_1 + x_2 y_2 + \cdots + x_n y_n$$

$$\log\left(\prod_{i=1}^{n} a_i^{b_i}\right) = \sum_{i=1}^{n} b_i \log(a_i)$$

Therefore, by the rearrangement ~~ineq~~ inequality, we can maximize payoff by sorting both sets A and B in nondecreasing order.

To prove that this algorithm maximize the payoff, ~~you~~ we can use induction on n. The base case is trivial: when $n=1$, there is only ~~two~~ one way to order both sets. For the introduction step, assume that the algorithm works for any pair of sets with $n-1$ elements. Now consider any ~~pai~~ pair of sets with n elements: $A = \{a_1, a_2 \cdots a_n\}$ and $B = \{b_1, b_2, \cdots b_n\}$.

Let a_m and b_m be the maximum elements in each set. Without loss of generality ~~, we~~ assume that they are at position m. Let $A' = A - \{a_m\}$ and $B' = B - \{b_m\}$. By ~~By introduction~~ induction hypothesis, the optimal ordering for these smaller ~~stes~~ sets is to sort them in non-decreasing order.

Now, consider any ordering for the original sets that does not put a_m and b_m at position n. This means that there exists some $i < n$ such that either $a_i > a_m$ or $b_i > b_m$ or both. By swapping these elements with a_m and b_m, we can ~~x~~ increase the payoff by

$$a_i^{b_i} a_m^{b_m} < a_i^{b_m} a_m^{b_i}$$

This follows from taking logarithms on both sides and applying the rearrangement inequality to $\{\log(a_i), \log(a_m)\}$ and $\{b_i, b_m\}$.

Therefore, any ordering that does not put a_m and b_m at position n is sub-optimal. Hence, the optimal ordering is to ~~d~~ sort both sets in non-decreasing order.

The running time of this algorithm depends on how you sort each set. If you use an efficient sorting algorithm such as quick sort, $O(n \log n)$ is needed.

Problem 4:

(a) Suppose that $n > 1$ and $m \geq 2$.

For each row i, there are at most n choices for the pixel to remove. However, if we fix a choice for row i, then for row $i+1$, there are at most 3 choices for the pixel to remove: either the same column as row i, or one column to the left or right of it. Therefore, the number of possible seams is bounded below by $n \times 3^{m-1}$, which grows exponentially in m.

(b) Dynamic programming

We define a two-dimensional array M of size $m \times n$, where each entry $M[i,j]$ stores the minimum cumulative ~~distr~~ disruption measure of a seam ending at pixel $A[i,j]$. We can complete this array using the following recurrence:

$$M[i,j] = d[i,j] + \min(M[i-1,j-1], M[i-1,j], M[i-1,j+1])$$

for all valid indices (i,j), with base cases:

$$M[0,j] = d[0,j]$$

for all valid indices (j). The minimum value in the last row at M gives us the minimum disruption measure of any seam.

To find the best seam itself, we can backtrack from this minimum value and trace the path of pixels that led to it. We can store the path in an array S of size m, where each entry $S[i]$ stores the ~~colo~~ column index of the pixel removed from row i.

```
S = new int[m]
for i = 0 to m-1:
    S[i] = -1
minVal = infinity
minIndex = -1
for j=0 to n-1:
    if M[m-1][j] < minVal:
        minVal = M[m-1][j]
        minIndex = j
S[m-1] = minIndex
for i = m-2 downto 0:
    prevMinVal = infinity
    prevMinIndex = -1
    for k = max(0, S[i+1]-1)
        to min(n-1, S[i+1]+1):
        if M[i][k] < prevMinVal:
            preMinVal = M[i][k]
            preMinIndex = k
    S[i] = prevMinIndex
return S
```

The running time of this algorithm is $O(mn)$, where m is the number of rows and n is the number of columns. This is because we need to fill each entry of M once, and then backtrack from one entry per row.

Problem 5

Dynamic programming

This idea is to use a two-dimensional array dp of size $(n+1) \times (M+1)$, where each entry $dp[i][j]$ stroes the number of ways to order dishes from the first i dishes such that their total price adds up to j. We can complete this array using the following recurrence: $dp[i][j] = dp[i-1][j] + dp[i-1][j-a_i]$ for all valid indices (i,j), with base cases:

$$dp[0][0]=1 \quad dp[0][j]=0 \text{ for } j>0$$
$$dp[i][0]=1 \text{ for } i>0$$

The intuition behind this recurrence is that for each dish i, we have two choices: either we order it or we skip it. If we order it, then we need to find ways to order dishes from the first $i-1$ dishes such that their total price adds up to $j-a_i$. If we skip it, then we need to find ways to order dishes from the first $i-1$ dishes such that their total price adds up to j. The number of Ways for each choice is stroed in $dp[i-1][j-a_i]$ and $dp[i-1][j]$ respectively, and we add them up to get $dp[i][j]$.

The final answer will be ~~stror~~ stored in $dp[n][M]$, which gives us the number of ways to order dishes from all n dishes such that their total price adds up to M.

$O(nM)$, n is the number of dishes and M is Jack's money. Because we need to fill each entry of dp once.

Problem 6

Dynamic program

$dp[i][j]$ ~~stored stored~~ stores a boolean value indicating ~~where~~ whether $r[0 \ldots i+j-1]$ can be formed by interleaving $s[0 \ldots i-1]$ and $t[0, j-1]$. size: $(n+1) \times (m+1)$

$dp[i][j] = (dp[i-1][j]$ and $r[i+j-1] = s[i-1])$
or $(dp[i][j-1]$ and $r[i+j-1] = t[j-1])$

for all valid indices (i, j), with base cases:

$dp[0][0] = true$

$dp[0][j] = dp[0][j-1]$ and $r[j-1] = t[j-1]$ for $j > 0$

$dp[i][0] = dp[i-1][0]$ and $r[i-1] = s[i-1]$ for $i > 0$

The final answer ~~is~~ will be stored in $dp[n][m]$.

$O(nm)$, n is the length of s and m is the length of t.