Algorithm

Homework 3

駱皓正
r05227124
Dept. of Psy

1. (25) Suppose you're consulting for a bank that's concerned about fraud detection, and they come to you with the following problem. They have a collection of n bank cards that they've confiscated, suspecting them of being used in fraud. Each bank card is a small plastic object, containing a magnetic stripe with some encrypted data, and it corresponds to a unique account in the bank. Each account can have many bank cards corresponding to it, and we'll say that two bank cards are equivalent if they correspond to the same account. It's very difficult to read the account number off a bank card directly, but the bank has a high-tech "equivalence tester" that takes two bank cards and, after performing some computations, determines whether they are equivalent. Their question is the following: among the collection of n cards, is there a set of more than $n/2$ of them that are all equivalent to one another? Assume that the only feasible operations you can do with the cards are to pick two of them and plug them in to the equivalence tester. Show how to decide the answer to their question with only $O(n \log n)$ invocations of the equivalence tester.

Answer:

Here I employ the D&C skill to solve the problem. I donate a card which is equivalent to other $n/2$ or more cards as M.

EqTest(*C, p, q*)

//C[p, …, q] and n = |C|

1 **if (**n = 0**) then return** 0 //C[]
2 **if (**n = 1**) then return** p //C[p]
3 **else**
4 r = $\lfloor (p+q)/2 \rfloor$
5 m_1 = **EqTest(*C, p, r*)**
6 m_2 = **EqTest(*C, r+1, q*)**
7 M = **Combine-EqTest(*C, m_1, m_2, p, r, q*)**
8 **if (**M ≠ 0**) then return** M **else return** 0

Correctness:

DIVIDE: by line 4-6

Like merge-sort, first, algorithm splits *n* cards into 2 halves, and recursively call itself on each half. When splitting into the single card, the single card would be returned as M. At the other higher levels, recursive calls would return their Ms if any or 0 if not found.

COMBINE: by line 7

Here I donate B as combined 2 split subsets of cards (i.e. RHS & LHS). Knowing that if there is a M in B, then M must be M of one of RHS & LHS of cards or both. Here exist 3 conditions.

I. If RHS & LHS both return 0, then B cannot have M. Hence, the call returns 0.

II. If RHS returns M and meanwhile LHS returns 0, then we can compare M with cards within B and count the number of cards which are equal to M. If it is a M then returns M, otherwise returns 0. (same way for reverse condition)

III. If RHS & LHS both return M_r & M_l. Do the same way in II for M_r & M_l. If one of them is a M in B then returns it, otherwise returns 0.

Running Time Analysis:

We can write down the recurrence relation by rationales above. For costs of recursion, two recursive calls are with half size of original size. For costs of combination, in worst case III, 2*n* comparisons are needed.

$$T(n) = 2T\left(\frac{n}{2}\right) + 2n$$

By master theorem, this algorithm is bound by O(*n* log *n*).

Reference:

class "D&C" power point; data structure; no coworkers

2. (25) There are n items in a store. For i =1, 2, …. , n, item i has weight $w_i > 0$ and worth $v_i > 0$. A thief can carry a maximum weight of W pounds in a knapsack. Each item can be broken into smaller pieces, so the thief may decide to carry only a fraction x_i of item i, where $0 \le x_i \le 1$. Item i contributes $x_i w_i$ to the total weight in the knapsack, and $x_i v_i$ to the value of the load. Design an algorithm which can maximize $\sum_{(i=1, n)} x_i v_i$ subject to $\sum_{(i=1, n)} x_i w_i \le W$ in O(n *log* n) time.

Answer:

This problem applies divide-&-conquer and greedy-algorithm skills. To maximize total worth and meanwhile be bound by W restriction. First, I could calculate worth per pound p_i for i item. Next, the merge sort is employed to sorting the p_i in a descending order. In the end, by the greedy rule which indicates "carry worthiest piece as possible," the algorithm put a worthiest piece in a knapsack for each iteration till a knapsack is full. Here is the algorithm and assume the pounds of each item and W are integer.

MaxWorth(*I*, *W*)

1 merge-sort(v_1/w_1, …, v_n/w_n) //merge-sort by D&C bound by O($n$lg$n$)
2 P: {p_1, ..., p_n} = sorted items in descending order of their worth per weight p_i
3 M = 0 //maximum total worth
4 **for** j **from** 1 **to** W **do**
5 i = the item which p_i is highest in P and $w_i \neq 0$
6 M = M + p_i
7 $w_i = w_i - 1$
8 **return** M

M is maximum total worth and if W is sufficiently smaller than $\sum_{(i=1, n)} w_i$ then it will be filled up when termination. Time complexity analysis: merge-sort takes O($n$lg$n$) and filling the knapsack takes constant time due to W. Hence, the time complexity is bound by O($n$lg$n$).

Reference:

class "D&C and Greedy algorithm" power point; no coworkers

3. (25) We are planning a hiking trip with total distance n kilometers. On the route, there are hotels located *i* kilometers from the starting point for every integer *i*. The hotel located *i* kilometers from the starting point has cost c_i. Assume that we want to walk at most 20 kilometers each day, design an efficient algorithm to minimize the total hotel cost. You may assume c_n=0. (Partial points for larger runtime.)

Answer:

Dynamic programming is employed here and induction occurs on each hotel/kilometer *i*. Let Cost(*i*) record the total minimum cost we will pay for hotels when we arrive *i* hotel, which the cost of *i*-th hotel is counted. Here I donate O as starting point. Assume $c_O = 0$ and $c_n = 0$. By the restriction of 20 kilometers per day, here is the recurrence relation:

$$\text{Cost}(i) = C_i + \text{Min}\{\text{Cost}(i-20), \ldots, \text{Cost}(i-1)\}$$

The algorithm is designed by bottom-up iteration method. One dimensional array M[*i*] is to store Cost(*i*).

Min-Hotel-Cost(C_O, C_1, …, C_n)

```
1   M[0] = 0
2   for i from 1 to 20 do
3       M[i] = C_i
4   for i from 21 to n do
5       M[i] = C_i + Min{M[i-20], …, M[i-1]}
6   return M[n]
```

M[n] is the minimum total hotel cost when we arrived *n*-th hotel. My algorithm is correct, since if we stop by the *i*-th hotel today, we must start from at most *i*-20-th hotel. My algorithm chooses the minimum total cost among start points, then after adding the current cost, the result must be minimum. The running time analysis includes extract-min among 20 numbers which is a constant time and fill up a one-dimensional array which is bound by O(*n*). Hence, the time complexity is O(*n*).

Reference:

class "DP" power point; no coworkers

4. (25) Suppose you're consulting for a company that manufactures PC equipment and ships it to distributors all over the country. For each of the next n weeks, they have a projected supply s_i of equipment (measured in pounds), which has to be shipped by an air freight carrier. Each week's supply can be carried by one of two air freight companies, A or B.

   - ✧ Company A charges a fixed rate r per pound (so it costs $r \cdot s_i$ to ship a week's supply s_i).
   - ✧ Company B makes contracts for a fixed amount c per week, independent of the weight. However, contracts with company B must be made in blocks of four consecutive weeks at a time.

   A *schedule*, for the PC company, is a choice of air freight company (A or B) for each of the n weeks, with the restriction that company B, whenever it is chosen, must be chosen for blocks of four contiguous weeks at a time. The *cost* of the schedule is the total amount paid to companies A and B, according to the description above.

   Give a polynomial-time algorithm that takes a sequence of supply values s_1, s_2, …, s_n and returns a *schedule* of minimum cost. (Partial points for larger runtime.)

   Example. Suppose $r = 1$, $c = 10$, and the sequence of values is

   11, 9, 9, 12, 12, 12, 12, 9, 9, 11.

   Then the optimal schedule would be to choose company A for the first three weeks, then company B for a block of four consecutive weeks, and then company A for the final three weeks.

Answer:

First and foremost, here designs a recursive DP relation. My attempt is to make induction on supply index i. Let Com-Cost(i) be the minimum total cost for the sum of costs of s_1 to s_i. For each condition:

$$\text{Com-Cost}(0) = 0$$

for $i = 0$

Due to restriction of company B, we must apply company A when $i < 4$.

$$\text{Com-Cost}(i) = \text{Com-Cost}(i-1) + r \times s_i$$

for $i < 4$

For the rest, we can choose either A for i or B for i-3 to i to gain total min cost.

$$\text{Com-Cost}(i) = \text{Min}\{\text{Com-Cost}(i-1) + r \times s_i \ , \ \text{Com-Cost}(i-4) + 4c\}$$

for $i \geq 4$

According to the question, apart from giving the minimum total cost of the schedule, we should provide the optimal *schedule* when termination. In the following algorithm, I, by bottom-up iteration method, let the array M[i] store the Com-Cost(i) and let the two-dimensional array OS[i,] store the optimal *schedule* of first i weeks. Here I assume the number of supplies/weeks n is larger than 4.

AB-Schedule(*S, r, c*)

```
1   M[0] = 0
2   for j from 1 to n do
3       OS[0, j] = 0
4   for i from 1 to 3 do
5       M[i] = M[i − 1] + r×s_i
6       for j from 1 to n do
7           OS[i, j] = OS[i-1, j]        //copy from the former row
8       OS[i, i] = "A"
9   for i from 4 to n do
10      if (M[i − 1] + r×s_i < M[i − 4] + 4c) then
11          M[i] = M[i − 1] + r×s_i
12          for j from 1 to n do
13              OS[i, j] = OS[i-1, j]    //copy from the former row
14          OS[i, i] = "A"
15      else
16          M[i] = M[i − 4] + 4c
17          for j from 1 to n do
18              OS[i, j] = OS[i-4, j]    //copy from the former 4 row
19          for j from i-3 to i do
20              OS[i, j] = "B"
21  return OS[n, 1…n] and M[n]
```

My algorithm builds the optimal schedule OS[n, 1…n] when termination by bottom-up iteration and its min-cost M[n] of it. The running time analysis is bound by $O(n^2)$ to fill up the OS[,] matrix, which is a polynomial running time.

To elaborate the OS[,] and its correction, the following matrix performs the **Example** in the question.

OS		*j*										
	index	0	1	2	3	4	5	6	7	8	9	10
	0	0	0	0	0	0	0	0	0	0	0	0
	1	0	A	0	0	0	0	0	0	0	0	0
	2	0	A	A	0	0	0	0	0	0	0	0
	3	0	A	A	A	0	0	0	0	0	0	0
	4	0	B	B	B	B	0	0	0	0	0	0
i	5	0	A	B	B	B	B	0	0	0	0	0
	6	0	A	A	B	B	B	B	0	0	0	0
	7	0	A	A	A	B	B	B	B	0	0	0
	8	0	A	A	A	B	B	B	B	A	0	0
	9	0	A	A	A	B	B	B	B	A	A	0
	10	0	A	A	A	B	B	B	B	A	A	A

Reference:

class "DP" power point;

no coworkers