CST Part IA: Algorithm, SV 5
Joe Yan
2017-2-24

1 Dynamic Programming

```
public static int n = 4;
public static Integer[] b = { 3, 2, 1, 1 };
public static Integer[] w = { 12, 14, 25, 32 };
public static int W = 50;
public static ArrayList<Integer[]> choice = new ArrayList<>();
public static Hashtable<List<Integer>, Integer> memory = new Hashtable<>();
public void constructPossibleChoice(int wi, int pointer, Integer[] ch) {
        if (wi < w[0]) {
                choice.add(ch);
        } else if (pointer >= 0) {
                Integer[] newch = new Integer[n];
                for (int i = 0; i < n; ++i) {
                        newch[i] = ch[i];
                }
                constructPossibleChoice(wi, pointer - 1, newch);
                if (wi - w[pointer] >= 0) {
                        newch[pointer] += 1;
                        constructPossibleChoice(wi - w[pointer], pointer, newch)
                            ;
                }
        }
}
public int findMinNoOfShelves(Integer[] bk) {
        Integer min = memory.get(Arrays.asList(bk));
        if (min != null) {
                return min;
        }
        boolean allZero = true;
        for (Integer i : bk) {
                if (i > 0)
                        allZero = false;
        }
        if (allZero) {
                return 0;
        }
        min = Integer.MAX_VALUE;
        Integer[] minKey = new Integer[n];
        boolean cont = true;
        for (Integer[] is : choice) {
                cont = true;
                Integer[] newbk = new Integer[n];
                for (int i = 0; i < n; ++i) {
                        if (bk[i] > 0 && is[i] > 0) {
                                // Check there are such books to remove.
                                cont = false;
                        }
                        newbk[i] = bk[i] - is[i];
                        if (newbk[i] < 0) {
                                newbk[i] = 0;
                        }
                }
                if (cont) {
                        continue;
                }
                int tempNewMin = findMinNoOfShelves(newbk);
                if (min > 1 + tempNewMin) {
                        for (int i = 0; i < n; ++i) {
                                minKey[i] = bk[i];
                        }
                        min = 1 + tempNewMin;
                }
        }
        for (Integer i : minKey) {
                System.out.print(i + " ");
        }
        System.out.print("with min: " + min + "\n");
        memory.put(Arrays.asList(minKey), min);
        return min;

\\sample of ArrayList<Integer[]> choice = new ArrayList<>()
0 0 0 1 with min: 1
0 0 1 0 with min: 1
0 0 1 1 with min: 2
0 1 0 0 with min: 1
0 1 0 1 with min: 1
0 1 1 0 with min: 1
0 1 1 1 with min: 2
0 2 0 0 with min: 1
0 2 0 1 with min: 2
0 2 1 0 with min: 2
0 2 1 1 with min: 2
1 0 0 0 with min: 1
1 0 0 1 with min: 1
1 0 1 0 with min: 1
1 0 1 1 with min: 2
2 0 0 0 with min: 1
2 0 0 1 with min: 2
2 0 1 0 with min: 1
2 0 1 1 with min: 2
3 0 0 0 with min: 1
3 0 0 1 with min: 2
3 0 1 0 with min: 2
3 0 1 1 with min: 2
1 1 0 1 with min: 2
1 1 0 0 with min: 1
1 2 0 1 with min: 2
2 1 0 0 with min: 1
3 1 0 1 with min: 2
1 2 0 0 with min: 1
2 2 0 0 with min: 2
3 1 0 0 with min: 1
3 2 0 1 with min: 2
1 1 1 0 with min: 2
1 2 1 0 with min: 2
2 1 1 0 with min: 2
2 2 1 0 with min: 2
3 1 1 0 with min: 2
3 2 1 1 with min: 3
}
```

The optimal solution is constructed by series of remove of "an almost full shelf of books".
"An almost full shelf of books" means a collection of books being able to fit in a shelf but adding any single book to the collection will make it unable to fit in a shelf.
The algorithm first constructs all possible almost full shelves of books in (ArrayList<Integer[]>choice).
The algorithm then uses the following structure of the optimal solution:

$$findMin(b[]) = \begin{cases} findMin(b[]) = Min(1 + findMin(remove\ choice[i]\ from\ b[])\ \forall i.\ \exists\ choice[i]\ to\ remove) & : \exists\ b[i]! = 0 \\ 0 & : \forall\ b[i] = 0 \end{cases}$$

The algorithm uses a hash table to memorize optimal solutions of sub-questions because I was unable to find a well-structured data structure to store the previous result. However hash table take $O(1)$ average cost for the insertion and search so I think this will not effect the efficiency

of the algorithm more than a constant factor.
Overall, the time complexity is $O(e^x)$

2 String Searching

```
public static ArrayList<Integer> stringSearch(char[] string, char[]
    pattern) {
        ArrayList<Integer> result = new ArrayList<>();
        int lenStr = string.length;
        int lenPat = pattern.length;
        ArrayList<Integer> bt = new ArrayList<>();
        for (int i = 0; i <= lenStr - lenPat; ++i) {
                if (pattern[0] == string[i]) {
                        bt.add(i);
                }
        }
        if (lenPat == 1) {
                return bt;
        }
        for (Integer p : bt) {
                for (int i = 1; i < lenPat; ++i) {
                        if (string[p + i] != pattern[i]) {
                                break;
                        }
                        if (i == lenPat - 1) {
                                result.add(p);
                        }
                }
        }
        return result;
}

public static ArrayList<Integer> btGenerate(char[] string, char[]
    pattern) {
        int rep = 0;
        int lenStr = string.length;
        int lenPat = pattern.length;
        ArrayList<Integer> bt = new ArrayList<>();
        while (++rep < lenPat) {
                if (pattern[rep] != pattern[0]) {
                        break;
                }
        }
        for (int i = rep - 1; i <= lenStr - lenPat; i += rep) {
                if (pattern[0] == string[i]) {
                        for (int j = -1; j > -rep; --j) {
                                if (pattern[0] != string[i + j]) {
                                        break;
                                } else {
                                        --i;
                                }
                        }
                        bt.add(i);
                }
        }
        return bt;
}
```

Shift window First the algorithm will try to find all ‘o’ in the string and store their index. Then it checks whether the pattern fits the substring of string starting from these index with the same length as the pattern. If it fits then the algorithm find a pattern in such string.
Let n be the length of the string, n_p be the length of the pattern.
Assume there are k patterns in the string.
Searching through the whole string costs $O(n)$ to generate indexes.

Searching for windows starting from each stored index costs $O(kn_p)$.
Overall time complexity is $O(n + kn_p)$. If the k and n_p is relatively small comparing to n then the time complexity is dominated by the length of the string. However, when special input such as a string with 1000 'a's and a pattern with 500 'a's happens, the time complexity suddenly raises to $O(n^2)$ which is much worse than linear.

Repeat prefix When we want to search a pattern with a repeated start such as "eed". It is reasonable to add the length of such repeat to the counter each time when the algorithm do linear search through the whole string. Because if a "eed" pattern start from index i in the string then the index i and i+1 will both be 'e'. As the algorithm find 'e' by adding 2 to the counter each time, then it will check back to see whether it is a "second" 'e' and always puts the index of first 'e' into the arraylist bt.
Let such repeat prefix has length l.
Notice the algorithm is expected to do $\frac{n}{l} + k \times \frac{l(l-1)}{2} \times \frac{1}{l} = \frac{n}{l} + \frac{k(l-1)}{2}$ by probability calculation.
Overall time complexity now is $O(\frac{n}{l} + kn_p)$.
For kn_p part is not changed because the ratio of $\frac{l-1}{2}$ and n_p is a constant. i.e. There is just a constant factor difference for kn_p.

Hash function This can be further improved by a proper implementation of a hash function when the algorithm build the bt arraylist try to collect those lost information even if those characters are not the same as the first character of the pattern.
e.g. The algorithm can map "abcd" to $Hash(\text{``}abcd\text{''}) = 1\times27^3+2\times27^2+3\times27^1+4\times27^0$.
Let the next character in the string to be e. Then the algorithm shift the window by one character to the right and the calculation of $Hash(\text{``}bcde\text{''}) = (Hash(\text{``}abcd\text{''}) - 1 \times 27^3) * 27 + 5 \times 27^0$. This takes a constant time independent from the length of the pattern to calculate the hash valve of the next window in the string. And more important fact is the algorithm now is trying to collect all information to these hash values for each window. It has $O(n)$ time complexity to calculate such hash values for all windows with the same length as the pattern in the string.
Then the algorithm can calculate the $Hash(pattern)$ and compare it to the list of calculated hash values in the step before. If the hash function is injection which means it has no collision the algorithm just returns those matches. Otherwise the has function has collisions, then do one more step to check whether the pattern is exactly the same as the substring in the window. This step has $O(k)$ time complexity under the assumption of the application of a collision-free hash function.
Overall it takes $O(n + k) = O(n)$ time to execute the whole algorithm even in the worst case. But the implementation of hash function may be costly and so in very random input the average running time of this hash approach may be worse than the previous one.