Dynamic programming

Bit operations provide an efficient and convenient way to implement dynamic programming algorithms whose states contain subsets of elements, because such states can be stored as integers. Next we discuss examples of combining bit operations and dynamic programming.

Optimal selection

As a first example, consider the following problem: We are given the prices of k products over n days, and we want to buy each product exactly once. However, we are allowed to buy at most one product in a day. What is the minimum total price? For example, consider the following scenario ($k = 3$ and $n = 8$):

	0	1	2	3	4	5	6	7
product 0	6	9	5	2	8	9	1	6
product 1	8	2	6	2	7	5	7	2
product 2	5	3	9	7	3	5	1	4

In this scenario, the minimum total price is 5:

	0	1	2	3	4	5	6	7
product 0	6	9	5	**2**	8	9	1	6
product 1	8	**2**	6	2	7	5	7	2
product 2	5	3	9	7	3	5	**1**	4

Let $\texttt{price}[x][d]$ denote the price of product x on day d. For example, in the above scenario $\texttt{price}[2][3] = 7$. Then, let $\texttt{total}(S,d)$ denote the minimum total price for buying a subset S of products by day d. Using this function, the solution to the problem is $\texttt{total}(\{0 \ldots k-1\}, n-1)$.

First, $\texttt{total}(\emptyset, d) = 0$, because it does not cost anything to buy an empty set, and $\texttt{total}(\{x\}, 0) = \texttt{price}[x][0]$, because there is one way to buy one product on the first day. Then, the following recurrence can be used:

$$\texttt{total}(S,d) = \min(\texttt{total}(S,d-1), \min_{x \in S}(\texttt{total}(S \setminus x, d-1) + \texttt{price}[x][d]))$$

This means that we either do not buy any product on day d or buy a product x that belongs to S. In the latter case, we remove x from S and add the price of x to the total price.

The next step is to calculate the values of the function using dynamic programming. To store the function values, we declare an array

```
int total[1<<K][N];
```

where K and N are suitably large constants. The first dimension of the array corresponds to a bit representation of a subset.

First, the cases where $d = 0$ can be processed as follows:

```
for (int x = 0; x < k; x++) {
    total[1<<x][0] = price[x][0];
}
```

Then, the recurrence translates into the following code:

```
for (int d = 1; d < n; d++) {
    for (int s = 0; s < (1<<k); s++) {
        total[s][d] = total[s][d-1];
        for (int x = 0; x < k; x++) {
            if (s&(1<<x)) {
                total[s][d] = min(total[s][d],
                                  total[s^(1<<x)][d-1]+price[x][d]);
            }
        }
    }
}
```

The time complexity of the algorithm is $O(n2^k k)$.

From permutations to subsets

Using dynamic programming, it is often possible to change an iteration over permutations into an iteration over subsets[1]. The benefit of this is that $n!$, the number of permutations, is much larger than 2^n, the number of subsets. For example, if $n = 20$, then $n! \approx 2.4 \cdot 10^{18}$ and $2^n \approx 10^6$. Thus, for certain values of n, we can efficiently go through the subsets but not through the permutations.

As an example, consider the following problem: There is an elevator with maximum weight x, and n people with known weights who want to get from the ground floor to the top floor. What is the minimum number of rides needed if the people enter the elevator in an optimal order?

For example, suppose that $x = 10$, $n = 5$ and the weights are as follows:

person	weight
0	2
1	3
2	3
3	5
4	6

In this case, the minimum number of rides is 2. One optimal order is $\{0,2,3,1,4\}$, which partitions the people into two rides: first $\{0,2,3\}$ (total weight 10), and then $\{1,4\}$ (total weight 9).

[1] This technique was introduced in 1962 by M. Held and R. M. Karp [34].

The problem can be easily solved in $O(n!n)$ time by testing all possible permutations of n people. However, we can use dynamic programming to get a more efficient $O(2^n n)$ time algorithm. The idea is to calculate for each subset of people two values: the minimum number of rides needed and the minimum weight of people who ride in the last group.

Let $\texttt{weight}[p]$ denote the weight of person p. We define two functions: $\texttt{rides}(S)$ is the minimum number of rides for a subset S, and $\texttt{last}(S)$ is the minimum weight of the last ride. For example, in the above scenario

$$\texttt{rides}(\{1,3,4\}) = 2 \quad \text{and} \quad \texttt{last}(\{1,3,4\}) = 5,$$

because the optimal rides are $\{1,4\}$ and $\{3\}$, and the second ride has weight 5. Of course, our final goal is to calculate the value of $\texttt{rides}(\{0 \ldots n-1\})$.

We can calculate the values of the functions recursively and then apply dynamic programming. The idea is to go through all people who belong to S and optimally choose the last person p who enters the elevator. Each such choice yields a subproblem for a smaller subset of people. If $\texttt{last}(S \setminus p) + \texttt{weight}[p] \leq x$, we can add p to the last ride. Otherwise, we have to reserve a new ride that initially only contains p.

To implement dynamic programming, we declare an array

```
pair<int,int> best[1<<N];
```

that contains for each subset S a pair $(\texttt{rides}(S), \texttt{last}(S))$. We set the value for an empty group as follows:

```
best[0] = {1,0};
```

Then, we can fill the array as follows:

```
for (int s = 1; s < (1<<n); s++) {
    // initial value: n+1 rides are needed
    best[s] = {n+1,0};
    for (int p = 0; p < n; p++) {
        if (s&(1<<p)) {
            auto option = best[s^(1<<p)];
            if (option.second+weight[p] <= x) {
                // add p to an existing ride
                option.second += weight[p];
            } else {
                // reserve a new ride for p
                option.first++;
                option.second = weight[p];
            }
            best[s] = min(best[s], option);
        }
    }
}
```

Note that the above loop guarantees that for any two subsets S_1 and S_2 such that $S_1 \subset S_2$, we process S_1 before S_2. Thus, the dynamic programming values are calculated in the correct order.

Counting subsets

Our last problem in this chapter is as follows: Let $X = \{0 \ldots n-1\}$, and each subset $S \subset X$ is assigned an integer $\texttt{value}[S]$. Our task is to calculate for each S

$$\texttt{sum}(S) = \sum_{A \subset S} \texttt{value}[A],$$

i.e., the sum of values of subsets of S.

For example, suppose that $n = 3$ and the values are as follows:

- $\texttt{value}[\emptyset] = 3$
- $\texttt{value}[\{0\}] = 1$
- $\texttt{value}[\{1\}] = 4$
- $\texttt{value}[\{0,1\}] = 5$
- $\texttt{value}[\{2\}] = 5$
- $\texttt{value}[\{0,2\}] = 1$
- $\texttt{value}[\{1,2\}] = 3$
- $\texttt{value}[\{0,1,2\}] = 3$

In this case, for example,

$$\begin{aligned}\texttt{sum}(\{0,2\}) &= \texttt{value}[\emptyset] + \texttt{value}[\{0\}] + \texttt{value}[\{2\}] + \texttt{value}[\{0,2\}] \\ &= 3 + 1 + 5 + 1 = 10.\end{aligned}$$

Because there are a total of 2^n subsets, one possible solution is to go through all pairs of subsets in $O(2^{2n})$ time. However, using dynamic programming, we can solve the problem in $O(2^n n)$ time. The idea is to focus on sums where the elements that may be removed from S are restricted.

Let $\texttt{partial}(S,k)$ denote the sum of values of subsets of S with the restriction that only elements $0 \ldots k$ may be removed from S. For example,

$$\texttt{partial}(\{0,2\},1) = \texttt{value}[\{2\}] + \texttt{value}[\{0,2\}],$$

because we may only remove elements $0 \ldots 1$. We can calculate values of $\texttt{sum}$ using values of $\texttt{partial}$, because

$$\texttt{sum}(S) = \texttt{partial}(S, n-1).$$

The base cases for the function are

$$\texttt{partial}(S,-1) = \texttt{value}[S],$$

because in this case no elements can be removed from S. Then, in the general case we can use the following recurrence:

$$\texttt{partial}(S,k) = \begin{cases} \texttt{partial}(S,k-1) & k \notin S \\ \texttt{partial}(S,k-1) + \texttt{partial}(S \setminus \{k\},k-1) & k \in S \end{cases}$$

Here we focus on the element k. If $k \in S$, we have two options: we may either keep k in S or remove it from S.

There is a particularly clever way to implement the calculation of sums. We can declare an array

```
int sum[1<<N];
```

that will contain the sum of each subset. The array is initialized as follows:

```
for (int s = 0; s < (1<<n); s++) {
    sum[s] = value[s];
}
```

Then, we can fill the array as follows:

```
for (int k = 0; k < n; k++) {
    for (int s = 0; s < (1<<n); s++) {
        if (s&(1<<k)) sum[s] += sum[s^(1<<k)];
    }
}
```

This code calculates the values of $\texttt{partial}(S,k)$ for $k = 0 \ldots n-1$ to the array `sum`. Since $\texttt{partial}(S,k)$ is always based on $\texttt{partial}(S,k-1)$, we can reuse the array `sum`, which yields a very efficient implementation.