More Dynamic Programming

Arnar Bjarni Arnarson

September 23, 2024

School of Computer Science

Reykjavík University

What is dynamic programming?

- A problem solving paradigm
- Similar in some respects to both divide and conquer and backtracking
- Divide and conquer recap:
    - Split the problem into *independent* subproblems
    - Solve each subproblem recursively
    - Combine the solutions to subproblems into a solution for the given problem
- Dynamic programming:
    - Split the problem into *overlapping* subproblems
    - Solve each subproblem recursively
    - Combine the solutions to subproblems into a solution for the given problem

Dynamic programming formulation

- Formulate the problem in terms of smaller versions of the problem (recursively)
- Turn this formulation into a recursive function
- Memoize the function (remember results that have been computed)

Dynamic programming formulation

```
map<problem, value> memory;

value dp(problem P) {
    if (is_base_case(P)) {
        return base_case_value(P);
    }

    if (memory.find(P) != memory.end()) {
        return memory[P];
    }

    value result = some value;
    for (problem Q in subproblems(P)) {
        result = combine(result, dp(Q));
    }

    memory[P] = result;
    return result;
}
```

The Fibonacci sequence

The first two numbers in the Fibonacci sequence are 1 and 1. All other numbers in the sequence are defined as the sum of the previous two numbers in the sequence.

- Task: Find the nth number in the Fibonacci sequence
- Let's solve this with dynamic programming
- Formulate the problem in terms of smaller versions of the problem (recursively)

$$\begin{aligned}
\text{fibonacci}(1) &= 1 \\
\text{fibonacci}(2) &= 1 \\
\text{fibonacci}(n) &= \text{fibonacci}(n-2) + \text{fibonacci}(n-1)
\end{aligned}$$

The Fibonacci sequence

2. Turn this formulation into a recursive function

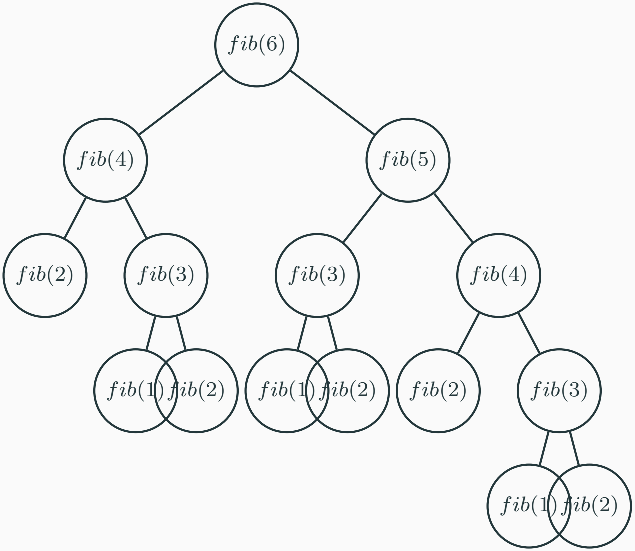
```
int fibonacci(int n) {
    if (n < 2) {
        return n;
    }

    int res = fibonacci(n - 2) + fibonacci(n - 1);

    return res;
}
```

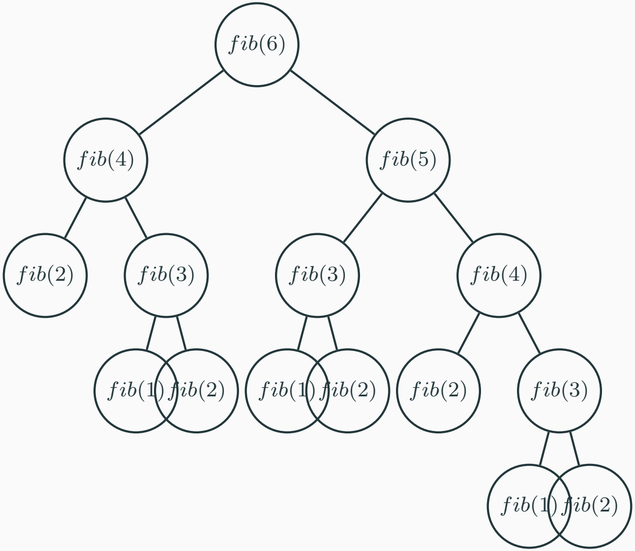
The Fibonacci sequence

- What is the time complexity of this?



The Fibonacci sequence

- What is the time complexity of this? Exponential, almost $O(2^n)$



The Fibonacci sequence

3. Memoize the function (remember results that have been computed)

```
map<int, int> mem;

int fibonacci(int n) {
    if (n <= 2) {
        return 1;
    }

    if (mem.find(n) != mem.end()) {
        return mem[n];
    }

    int res = fibonacci(n - 2) + fibonacci(n - 1);

    mem[n] = res;
    return res;
}
```

The Fibonacci sequence

```
int mem[1000];
for (int i = 0; i < 1000; i++)
    mem[i] = -1;

int fibonacci(int n) {
    if (n <= 2) {
        return 1;
    }

    if (mem[n] != -1) {
        return mem[n];
    }

    int res = fibonacci(n - 2) + fibonacci(n - 1);

    mem[n] = res;
    return res;
}
```

The Fibonacci sequence

- What is the time complexity now?
- We have n possible inputs to the function: 1, 2, $\dots$, n.
- Each input will either:
    - be computed, and the result saved
    - be returned from memory
- Each input will be computed at most once
- Time complexity is $O(n \times f)$, where f is the time complexity of computing an input if we assume that the recursive calls are returned directly from memory ($O(1)$)
- Since we're only doing constant amount of work to compute the answer to an input, $f = O(1)$
- Total time complexity is $O(n)$

Maximum sum

- Given an array $\mathrm{arr}[0]$, $\mathrm{arr}[1]$, $\ldots$, $\mathrm{arr}[n-1]$ of integers, find the interval with the highest sum

-15	8	-2	1	0	6	-3

Maximum sum

- Given an array $\text{arr}[0]$, $\text{arr}[1]$, $\ldots$, $\text{arr}[n-1]$ of integers, find the interval with the highest sum

-15	8	-2	1	0	6	-3

- The maximum sum of an interval in this array is 13

Maximum sum

- Given an array $\text{arr}[0]$, $\text{arr}[1]$, $\ldots$, $\text{arr}[n-1]$ of integers, find the interval with the highest sum

-15	8	-2	1	0	6	-3

- The maximum sum of an interval in this array is 13
- But how do we solve this in general?
    - Easy to loop through all $\approx n^2$ intervals, and calculate their sums, but that is $O(n^3)$
    - We could use our static range sum trick to get this down to $O(n^2)$
    - Can we do better with dynamic programming?

Maximum sum

- First step is to formulate this recursively
- Let $\mathrm{max\_sum}(i)$ be the maximum sum interval in the range $0, \ldots, i$
- Base case: $\mathrm{max\_sum}(0) = \max(0, arr[0])$
- What about $\mathrm{max\_sum}(i)$?
- What does $\mathrm{max\_sum}(i-1)$ return?
- Is it possible to combine solutions to subproblems with smaller i into a solution for i?
- At least it's not obvious...

Maximum sum

- Let's try changing perspective
- Let $\mathrm{max\_sum}(i)$ be the maximum sum interval in the range $0, \ldots, i$, *that ends at* i
- Base case: $\mathrm{max\_sum}(0) = arr[0]$
- $\mathrm{max\_sum}(i) = \max(arr[i], arr[i] + \mathrm{max\_sum}(i-1))$
- Then the answer is just $\max_{0 \le i < n} \{ \mathrm{max\_sum}(i) \}$

Maximum sum

- Next step is to turn this into a function

```
int arr[1000];

int max_sum(int i) {
    if (i == 0) {
        return arr[i];
    }

    int res = max(arr[i], arr[i] + max_sum(i - 1));

    return res;
}
```

Maximum sum

- Final step is to memoize the function

```
int arr[1000];
int mem[1000];
bool comp[1000];
memset(comp, 0, sizeof(comp));

int max_sum(int i) {
    if (i == 0) {
        return arr[i];
    }
    if (comp[i]) {
        return mem[i];
    }

    int res = max(arr[i], arr[i] + max_sum(i - 1));

    mem[i] = res;
    comp[i] = true;
    return res;
}
```

Maximum sum

- Then the answer is just the maximum over all interval ends

```
int maximum = 0;
for (int i = 0; i < n; i++) {
    maximum = max(maximum, max_sum(i));
}

printf("%d\n", maximum);
```

Maximum sum

- If you want to find the maximum sum interval in multiple arrays, remember to clear the memory in between

Maximum sum

- What about time complexity?
- There are n possible inputs to the function
- Each input is processed in $O(1)$ time, assuming recursive calls are $O(1)$
- Time complexity is $O(n)$

Coin change

- Given an array of coin denominations d_0, d_1, $\ldots$, d_{n-1}, and some amount x: What is minimum number of coins needed to represent the value x?
- Remember the greedy algorithm for Coin change?
- It didn't always give the optimal solution, and sometimes it didn't even give a solution at all...
- What about dynamic programming?

Coin change

- First step: formulate the problem recursively

- Let $\text{opt}(i, x)$ denote the minimum number of coins needed to represent the value x if we're only allowed to use coin denominations d_0, ..., d_i

- Base case: $\text{opt}(i, x) = \infty$ if $x < 0$
- Base case: $\text{opt}(i, 0) = 0$
- Base case: $\text{opt}(-1, x) = \infty$

- $\text{opt}(i, x) = \min \begin{cases} 1 + \text{opt}(i, x - d_i) \\ \text{opt}(i - 1, x) \end{cases}$

Coin change

```
int INF = 100000;
int d[10];

int opt(int i, int x) {
    if (x < 0) return INF;
    if (x == 0) return 0;
    if (i == -1) return INF;

    int res = INF;
    res = min(res, 1 + opt(i, x - d[i]));
    res = min(res, opt(i - 1, x));

    return res;
}
```

Coin change

```
int INF = 100000;
int d[10];
int mem[10][10000];
memset(mem, -1, sizeof(mem));

int opt(int i, int x) {
    if (x < 0) return INF;
    if (x == 0) return 0;
    if (i == -1) return INF;

    if (mem[i][x] != -1) return mem[i][x];

    int res = INF;
    res = min(res, 1 + opt(i, x - d[i]));
    res = min(res, opt(i - 1, x));

    mem[i][x] = res;
    return res;
}
```

Coin change

- Time complexity?
- Number of possible inputs are $n \times x$
- Each input will be processed in $O(1)$ time, assuming recursive calls are constant
- Total time complexity is $O(n \times x)$

Coin change

- How do we know which coins the optimal solution used?
- We can store backpointers, or some extra information, to trace backwards through the states
- See example...

Longest increasing subsequence

- Given an array $a[0]$, $a[1]$, $\ldots$, $a[n-1]$ of integers, what is the length of the longest increasing subsequence?

- First, what is a subsequence?
- If we delete zero or more elements from a, then we have a subsequence of a

- Example: $a = [5, 1, 8, 1, 9, 2]$

- $[5, 8, 9]$ is a subsequence
- $[1, 1]$ is a subsequence
- $[5, 1, 8, 1, 9, 2]$ is a subsequence
- $[]$ is a subsequence
- $[8, 5]$ is **not** a subsequence
- $[10]$ is **not** a subsequence

Longest increasing subsequence

- Given an array $a[0]$, $a[1]$, $\ldots$, $a[n-1]$ of integers, what is the length of the longest increasing subsequence?
- An increasing subsequence of a is a subsequence of a such that the elements are in (strictly) increasing order
- $[5, 8, 9]$ and $[1, 8, 9]$ are the longest increasing subsequences of $a = [5, 1, 8, 1, 9, 2]$
- How do we compute the length of the longest increasing subsequence?
- There are 2^n subsequences, so we can go through all of them
- That would result in an $O(n2^n)$ algorithm, which can only handle $n \leq 23$
- What about dynamic programming?

Longest increasing subsequence

- Let $\operatorname{lis}(i)$ denote the length of the longest increasing subsequence of the array $a[0]$, $\ldots$, $a[i]$
- Base case: $\operatorname{lis}(0) = 1$
- What about $\operatorname{lis}(i)$?
- We have the same issue as in the maximum sum problem, so let's try changing perspective

Longest increasing subsequence

- Let $\mathrm{lis}(i)$ denote the length of the longest increasing subsequence of the array $a[0]$, $\ldots$, $a[i]$, *that ends at* i
- Base case: we don't need one
- $\mathrm{lis}(i) = \max(1, \max_{j<i \text{ s.t. } a[j]<a[i]}\{1 + \mathrm{lis}(j)\})$

Longest increasing subsequence

```
int a[1000];
int mem[1000];
memset(mem, -1, sizeof(mem));

int lis(int i) {
    if (mem[i] != -1) {
        return mem[i];
    }

    int res = 1;
    for (int j = 0; j < i; j++) {
        if (a[j] < a[i]) {
            res = max(res, 1 + lis(j));
        }
    }

    mem[i] = res;
    return res;
}
```

Longest increasing subsequence

- And then the longest increasing subsequence can be found by checking all endpoints:

```
int mx = 0;
for (int i = 0; i < n; i++) {
    mx = max(mx, lis(i));
}

printf("%d\n", mx);
```

Longest increasing subsequence

- Time complexity?

- There are n possible inputs
- Each input is computed in $O(n)$ time, assuming recursive calls are $O(1)$
- Total time complexity is $O(n^2)$

- This will be fast enough for $n \leq 10\ 000$, much better than the brute force method!
- (It can be done faster ($\mathcal{O}(n \log(n))$) with dynamic programming optimizations, but we're not covering that right now)

Longest common subsequence

- Given two strings (or arrays of integers) $a[0], \ldots, a[n-1]$ and $b[0], \ldots, b[m-1]$, find the length of the longest subsequence that they have in common.

- $a = \texttt{"b}\underline{\texttt{an}}\texttt{an}\underline{\texttt{inn}}\texttt{"}$
- $b = \texttt{"k}\underline{\texttt{anin}}\texttt{a}\underline{\texttt{n}}\texttt{"}$
- The longest common subsequence of a and b, `"aninn"`, has length 5

Longest common subsequence

- Let $\operatorname{lcs}(i, j)$ be the length of the longest common subsequence of the strings $a[0]$, ..., $a[i]$ and $b[0]$, ..., $b[j]$

- Base case: $\operatorname{lcs}(-1, j) = 0$
- Base case: $\operatorname{lcs}(i, -1) = 0$

- $\operatorname{lcs}(i, j) = \max \begin{cases} \operatorname{lcs}(i, j-1) \\ \operatorname{lcs}(i-1, j) \\ 1 + \operatorname{lcs}(i-1, j-1) & \text{if } a[i] = b[j] \end{cases}$

Longest common subsequence

```
string a = "bananinn",
       b = "kaninan";
int mem[1000][1000];
memset(mem, -1, sizeof(mem));

int lcs(int i, int j) {
    if (i == -1 || j == -1) {
        return 0;
    }
    if (mem[i][j] != -1) {
        return mem[i][j];
    }

    int res = 0;
    res = max(res, lcs(i, j - 1));
    res = max(res, lcs(i - 1, j));

    if (a[i] == b[j]) {
        res = max(res, 1 + lcs(i - 1, j - 1));
    }

    mem[i][j] = res;
```

Longest common subsequence

- Time complexity?

- There are $n \times m$ possible inputs
- Each input is processed in $O(1)$, assuming recursive calls are $O(1)$
- Total time complexity is $O(n \times m)$

DP over bitmasks

- Remember the bitmask representation of subsets?
- Each subset of n elements is mapped to an integer in the range 0, $\ldots$, $2^n - 1$
- This makes it easy to do dynamic programming over subsets

Traveling salesman problem

- We have a graph of n vertices, and a cost $c_{i,j}$ between each pair of vertices i, j. We want to find a cycle through all vertices in the graph so that the sum of the edge costs in the cycle is minimal.
- This problem is NP-Hard, so there is no known deterministic polynomial time algorithm that solves it
- Simple to do in $O(n!)$ by going through all permutations of the vertices, but that's too slow if $n > 11$
- Can we go higher if we use dynamic programming?

Traveling salesman problem

- Without loss of generality, assume we start and end the cycle at vertex 0
- Let $\mathrm{tsp}(i, S)$ represent the cheapest way to go through all vertices in the graph and back to vertex 0, if we're currently at vertex i and we've already visited the vertices in the set S
- Base case: $\mathrm{tsp}(i, \text{all vertices}) = c_{i,0}$
- Otherwise $\mathrm{tsp}(i, S) = \min_{j \notin S} \{ c_{i,j} + \mathrm{tsp}(j, S \cup \{j\}) \}$

Traveling salesman problem

```
const int N = 20;
const int INF = 100000000;
int c[N][N];
int mem[N][1<<N];
memset(mem, -1, sizeof(mem));
int tsp(int i, int S) {
    if (S == ((1 << N) - 1)) {
        return c[i][0];
    }
    if (mem[i][S] != -1) {
        return mem[i][S];
    }
    int res = INF;
    for (int j = 0; j < N; j++) {
        if (S & (1 << j))
            continue;
        res = min(res, c[i][j] + tsp(j, S | (1 << j)));
    }

    mem[i][S] = res;
    return res;
}
```

Traveling salesman problem

- Then the optimal solution can be found as follows:

```
printf("%d\n", tsp(0, 1<<0));
```

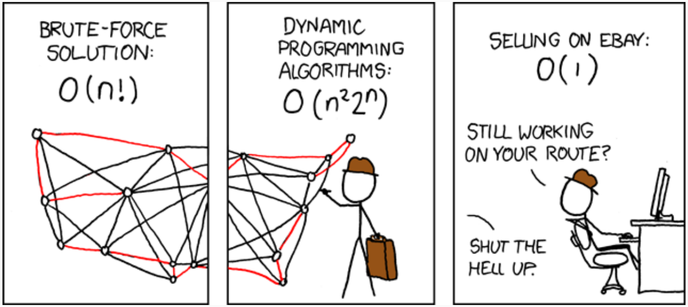
Traveling salesman problem

- Time complexity?

- There are $n \times 2^n$ possible inputs
- Each input is computed in $O(n)$ assuming recursive calls are $O(1)$
- Total time complexity is $O(n^2 2^n)$

- Now n can go up to about 20

Traveling salesman problem



Top-down vs. bottom-up

- What we have been doing so far is usually called top-down dynamic programming
- I.e. you start with the main problem (the top) and split it into smaller problems recursively (down)
- In some cases it can be better to do things bottom-up, which is pretty much just the reverse order
- Consider for example the fibonacci numbers. Then we'd start at the base cases and count up

Top-down vs. bottom-up

- Bottom-up is generally faster, but it has the issue that we have to make sure we iterate through the sub problems in the right order, since recursion doesn't automatically handle that for us
- Then why would we use bottom-up?
- Sometimes knowing in what order we go through the states can be useful, let's take an example

Decelerating jump

- Suppose we have n squares in a row, each containing a value a_i.
- We start at the first cell and want to jump through the cells.
- If we land on cell i we get a_i points and want to maximize our points.
- We can jump to any cell in front of us, but our jump can never go further than our last one.
- $n \leq 3000$, $-10^9 \leq a_i \leq 10^9$

Solution function

- Let us then find a recursive function describing the answer.
- Let $f(i, j)$ be the maximum score one can get starting at i and using jumps of at most length j. Then

$$f(i,j) = \begin{cases} a_n & \text{if } i = n \\ \max\limits_{1 \le k \le \min(j, n-i)} f(i+k, k) + a_i & \text{otherwise} \end{cases}$$

- Then we have n^2 states and it takes linear time to calculate each one. Thus a top-down solution would run in $\mathcal{O}(n^3)$, which is too slow.

Optimization

- What about bottom up?
- Each $f(i, j)$ only depends on values of f with greater i and lesser j, so we can calculate them in increasing order of j and decreasing order of i.

First implementation

```
#define INF (1LL << 60)
int main() {
    ll n; cin >> n;
    ll d[n][n], a[n];
    for(int i = 0; i < n; ++i) cin >> a[i];
    for(int i = 0; I < n; ++i) for(int j = 0; j < n; ++j) d[i][j] = -INF;
    d[n - 1][1] = a[n - 1];
    for(int i = n - 2; i >= 0; --i) d[i][1] = d[i + 1][1] + a[i];
    for(int i = 0; i < n; ++i) d[n - 1][i] = a[n - 1];
    for(int j = 2; j < n; ++j) for(int i = n - 2; i >= 0; --i)
        for(int k = 1; k < min(j + 1, n - i); k++)
            d[i][j] = max(d[i][j], d[i + k][k] + a[i]);
    cout << d[0][n - 1] << '\n';
}
```

Optimized?

- This is still $\mathcal{O}(n^3)$, so still not good enough.
- We note that when we calculate $f(i,j)$ we use the maximum value along the diagonal $f(i+1,1), f(i+2,2), \ldots, f(i+k,k)$.
- So we just add a prefix array for the diagonals to calculate those values in $\mathcal{O}(1)$.
- This will make the solution $\mathcal{O}(n^2)$, which is good enough, something that couldn't be done with top-down.

Fast implementation

```
#define INF (1LL << 60)
int main() {
    ll n; cin >> n;
    ll d[n][n], e[n], a[n];
    for(int i = 0; i < n; ++i) cin >> a[i];
    for(int i = 0; I < n; ++i) for(int j = 0; j < n; ++j) d[i][j] = -INF;
    d[n - 1][1] = e[n - 1] = a[n - 1];
    for(int i = n - 2; i >= 0; --i) e[i] = d[i][1] = d[i + 1][1] + a[i];
    for(int i = 0; i < n; ++i) d[n - 1][i] = a[n - 1];
    for(int j = 2; j < n; ++j) {
        e[n - j] = max(e[n - j], d[n - 1][j] = a[n - 1]);
        for(int i = n - 2; i >= 0; --i) {
            d[i][j] = e[i + 1] + a[i];
            if(i >= j - 1) e[i - j + 1] = max(e[i - j + 1], d[i][j]);
        }
    }
    cout << d[0][n - 1] << '\n';
}
```

Subset sum problem

- Another common dynamic programming task is known as the subset sum problem.
- Given n positive integers $a_1, \ldots, a_n$ find if there is a subset with sum c. Variants also include finding the sum closest to c, greatest sum not exceeding c and so on.
- The naïve solution here would involve checking every subset, which if done efficiently (for example with gray codes) takes $O(2^n)$, which is quite slow.

Subset sum problem

- Let $f(i, s)$ be a boolean function answering whether there exists a subset of $a_1, \ldots, a_i$ with sum s.
- Then

$$f(i,s) = \begin{cases} \texttt{true} \text{ if } i = s = 0 \\ \texttt{false} \text{ if } i = 0, s \neq 0 \\ \texttt{false} \text{ if } s < 0 \\ f(i-1, j) \;\; \texttt{or} \;\; f(i-1, j - a_i) \text{ otherwise} \end{cases}$$

- The different variants can then be read from the values of f. Each state takes $O(1)$ to compute, and there are $n(a_1 + \cdots + a_n)$ states. Denoting the sum by Σ this makes our time complexity $O(n\Sigma)$, which isn't great, but is often better than $O(2^n)$.

Subset sum problem

```
const int N = 20;
const int SIGMA = 10000;
int a[N];
int dp[N][SIGMA];
// use int so -1 is unmemoized
// 0 and 1 are the bools as usual
bool subsetsum(int i, int s) {
    if(i < 0) return s == 0;
    if(s < 0) return false;
    if(dp[i][s] != -1) return dp[i][s];
    return dp[i][s] = subsetsum(i - 1, s) || subsetsum(i - 1, s - a[i]);
}
```

Subset sum problem - variant

- Say we want to find the most even way to split the numbers into two groups, that is to say in a way that minimizes the difference of the sums of the two groups.
- Furthermore we want to actually output these numbers rather than just the difference in sum.
- We can use the subset sum solution to do this, simply adding a table that keeps track of what choices we made at what point.

Subset sum problem

```
#include <bits/stdc++.h>
using namespace std;
vector<int> a;
vector<vector<int>> dp, mv;
bool subsetsum(int i, int s) {
        if(i < 0) return s == 0;
        if(s < 0) return false;
    if(dp[i][s] != -1) return dp[i][s];
    dp[i][s] = 0;
    if(subsetsum(i - 1, s)) {
        dp[i][s] = 1;
        mv[i][s] = 1;
    } else if(subsetsum(i - 1, s - a[i])) {
        dp[i][s] = 1;
        mv[i][s] = 0;
    }
        return dp[i][s];
}
```

Subset sum problem

```
int main() {
        int n, sm = 0; cin >> n;
        a = vector<int>(n);
        for(int i = 0; i < n; ++i)
                cin >> a[i], sm += a[i];
        dp = mv = vector<vector<int>>(n, vector<int>(sm, -1));
        int bst = sm / 2;
        while(!subsetsum(n - 1, bst)) bst--;
        vector<bool> group1(n, false);
        for(int i = n - 1; i >= 0; --i) {
                if(!mv[i][bst]) {
                        group1[i] = true;
                        bst -= a[i];
                }
        }
        cout << "Difference: " << abs(bst - (sm - bst)) << '\n';
        cout << "Group 1: ";
        for(int i = 0; i < n; ++i) if(group1[i]) cout << a[i] << ' ';
        cout << "\nGroup 2: ";
        for(int i = 0; i < n; ++i) if(!group1[i]) cout << a[i] << ' ';
        cout << '\n';
}
```

Multidimensional DP comment

- The order in which you put the dimensions in a multidimensional dp can affect the running time.
- This is due to cache locality, so if you are fetching sequentially from one dimension and not the other, this can make one order faster.
- Usually it doesn't matter, but in a few cases it might.

Knapsack problem

- The subset sum problem time complexity is exponential, as the sums of the numbers is exponential in the size of the actual input.
- Among similar "hard" problems (in terms of time complexity) is the knapsack problem.
- We have n items, each with some value and some weight. We also have a knapsack with a maximum weight capacity and want to maximize the value with respect to this condition.

Knapsack problem

- We can once more use dynamic programming to solve this.
- We let $f(i, j)$ be the maximum value we can get from the first i items if our maximum weight is j.
- Let $v_1, \dots, v_n$ be the values and $w_1, \dots, w_n$ the weights. Then

$$f(i, j) = \begin{cases} -\infty \text{ if } j < 0 \\ 0 \text{ otherwise if } i = 0 \\ \max(f(i-1, j), f(i-1, j-w_i) + v_i) \text{ otherwise} \end{cases}$$

- The time complexity is then $O(nS)$ where S is the sum of the weights.
- We'll leave translating this into code as an exercise.

Egg dropping problem

- The previous problems are all very well known and classic in computer science. Let us also take a slightly less common example called the egg dropping problem.
- We have a building with k floors and we wish to figure out from which floor we have to drop an egg so it breaks. I.e. for some x dropping it from the x-th floor the egg will break, but dropping it from the $(x-1)$-st floor the egg won't break it.
- If we have n eggs, how few trials can we get away with in the worst case?

Egg dropping problem

- Say we drop the egg from a floor y.
- If the egg breaks, we only need to check floors $< y$ and have one less egg. This is essentially the same problem again but with $y - 1$ floors and $n - 1$ eggs.
- If the egg doesn't break we only need to check floors $> y$, so the problem is again the same with $k - y$ floors and n eggs.
- Since we are looking at the worst case, our result is the worse of these two.
- Since we can choose any y, we take the best result among all y.

Egg dropping problem

- Let $f(n, k)$ be the minimum number of trials for n eggs and k floors.
- We note that if we have one egg, we must always go through the floors in order since we can't afford to break an egg.
- All together this gives us

$$f(i, j) = \begin{cases} 1 \text{ if } k = 1 \\ 0 \text{ if } k = 0 \\ k \text{ if } i = 1 \\ \min_{1 \leq x \leq k} 1 + \max(f(i-1, x-1), f(i, j-x)) \end{cases}$$

- We see that this takes $O(k)$ per state and we have $O(nk)$ states, so the time complexity is $O(nk^2)$.

Egg dropping problem

- As a side note, this can also be solved in $O(nk)$ with a different dynamic programming approach.
- Try considering calculating the maximum number of floors you can check with n eggs and k trials using dynamic programming.
- Using some more clever ideas this can even be brought down to $O(n \log(k))$, but we won't need this here.

Complex states

- Most problems so far have had a state representable with a fixed number of integer values.
- This has allowed us to use static arrays for memoization.
- What if the state is not easily representable by integers or parameters are dynamic in count?
- For example: a larger range of integers, a string, a vector, or a custom type

Complex states

```
map<vector<int32_t>, int32_t> mem;
int32_t dp(const vector<int32_t>& state) {
    vector<int32_t> next_state(state);
    int32_t result{ 0 };
    for (auto& x : next_state) {
        if (x > 0) {
            int32_t old = x;
            x /= 2;
            result += dp(next_state);
            x = old;
        }
    }
    return result;
}
```

Complex states

```
map<tuple<type1_t, type2_t, ...>, result_t> mem;
// alternatively use unordered_map, need to ensure state is hashable
result_t dp(type1_t param1, type2_t param2, ...) {
    const auto state{ tie(param1, param2, ...) };
    if (is_base_case(state)) {
        return 0; // or 1 or whatever it should be
    }
    if (const auto it{ mem.find(state) }; it != mem.end()) {
        return it->second;
    }
    result_t result{};
    // compute answer recursively
    return mem[state] = result;
}
```