Subset sum and Knapsack problem

More Dynamic Programming Problems

Martin Chikov

November 4, 2018

Swiss Olympiad in Informatics

Subset Sum Problem

Subset Sum

Problem: Given is a list $a_0, a_1, \ldots, a_{n-1}$ of non-negative integers and an integer S.

Find out if it is possible to choose some of these numbers so that their sum is equal to S

Examples of Subset Sum

List A = [5; 10; 20; 50; 100; 200; 500] and S = 390:

Examples of Subset Sum

List A = $[5; 10; 20; 50; 100; 200; 500]$ and $S = 390$:

Answer: No.

Examples of Subset Sum

List A = [5; 10; 20; 50; 100; 200; 500] and S = 390:
Answer: No.

List A = [5; 10; 20; 50; 100; 200; 500] and S = 875:

Examples of Subset Sum

List A = [5; 10; 20; 50; 100; 200; 500] and S = 390:
Answer: No.

List A = [5; 10; 20; 50; 100; 200; 500] and S = 875:
Answer: Yes.

Brute-force solution

Idea: Check all sums we can achieve

Let say we have a set with i elements. In order to find if we can make sum k using some of these elements, there are 2 cases we need to check:

Brute-force solution

Idea: Check all sums we can achieve

Let say we have a set with i elements. In order to find if we can make sum k using some of these elements, there are 2 cases we need to check:

1. Can we make the sum k using the first $i-1$ elements? (is k one of the subset sums of the first $i-1$ elements)

Brute-force solution

Idea: Check all sums we can achieve

Let say we have a set with i elements. In order to find if we can make sum k using some of these elements, there are 2 cases we need to check:

1. Can we make the sum k using the first $i-1$ elements? (is k one of the subset sums of the first $i-1$ elements)
2. Can we make the sum $k-a_i$ using the first $i-1$ elements? (can we make k by adding the value of a_i to each of the subset sums of the first $i-1$ elements)

Brute-force solution - example

Example: $A = [3; 7; 10]$

The sets of the subset sums are:

Using the first 3 elements:

$\{()$, (3), (7), (3+7=10), (10), (3+10=13), (7+10=17), (3+7+10=20)$\}$

Using the first 2 elements:

$\{()$, (3), (7), (3+7=10)$\}$

Using the first 1 element:

$\{()$, (3)$\}$

Using the first 0 elements:

$\{()\}$

Implementation: Recursion

Brute-force solution - Recursion

```
vector<int> a;
bool Subsetsum (int i, int k) {
    if (k == 0)
        return true;
    if (i == 0 && k != 0)
        return false;
    if (a[i-1] > k)
        return Subsetsum(i-1, k);
    return Subsetsum(i-1, k) ||
           Subsetsum(i-1, k-a[i-1]);
}
bool answer = Subsetsum(a.size(),sum);
```

Brute-force solution - Run time analysis

The total amount of subsets we have to check is equal to 2^n.

For smaller values of n this can work relatively fast.
However for bigger values ($n = 100$):

$$2^{100} > \text{Edge of the observable universe}$$

How can we do better?

DP's four steps

1. Define sub problems.
2. Find a general recurrence formula to solve a sub problem using the solution to other sub problems.
3. Find base case(s).
4. Which is the relevant sub problem?

DP's four steps - Step 1

1.Define sub problems:

$S(i, k)$ - using the first i elements can we make sum k?

DP's four steps - Step 2

2.General formula:

Let's say we know all solutions $S(i, k)$ for the first i-1 elements.

We consider a new element: a_i

For any sum k we have 2 cases:

DP's four steps - Step 2

2.General formula:

Let's say we know all solutions $S(i,k)$ for the first i-1 elements.

We consider a new element: a_i

For any sum k we have 2 cases:

- We don't add the element to the sum
  $S(i,k) = S(i-1,k)$

DP's four steps - Step 2

2.General formula:

Let's say we know all solutions $S(i, k)$ for the first i-1 elements.

We consider a new element: a_i

For any sum k we have 2 cases:

- We don't add the element to the sum
  $S(i, k) = S(i-1, k)$
- We add the element to the sum
  $S(i, k) = S(i-1, k-a[i])$

DP's four steps - Step 2

2.General formula:

Let's say we know all solutions $S(i,k)$ for the first i-1 elements.

We consider a new element: a_i

For any sum k we have 2 cases:

- We don't add the element to the sum
  $S(i,k) = S(i-1,k)$
- We add the element to the sum
  $S(i,k) = S(i-1,k-a[i])$

  Therefore our general formula is:

  $S(i,k) = S(i-1,k) \parallel S(i-1,k-a[i])$

DP's four steps - Step 3

3.Find base case(s).

$S(0,0) = true$ - using 0 elements we can make $sum = 0$

$S(0,x) = false \; (for \; x > 0)$ - using 0 elements we can't make any $sum > 0$

DP's four steps - Step 4

4.Relevant sub problem:

$S(n, sum)$ - using n elements can we make sum?

Subset Sum - Dynamic Programming

```
void Subsetsum_dp() {
    vector<vector<bool> > dp(n+1, vector<int>(s+1, 0));
    dp[0][0]=1;
    for(int i=1; i<=n; i++) {
        for(int k=0; k<=s; k++) {
            if(k-v[i-1]<0)
               dp[i][k] = dp[i-1][k];
            else
               dp[i][k]= dp[i-1][k] || dp[i-1][k-v[i-1]];
        }
    }
    if(dp[n][s]==1) cout << "Yes\n";
    else cout << "No\n";
}
```

Dynamic programming solution - Run time analysis

For each element we have to calculate whether we can achieve sum k or not. Therefore we have:

Running time of $O(nS)$

(where n is the number of elements and S is the sum we want to check)

Knapsack Problem

Knapsack problem

Problem: Given a set of list $a_0, a_1, \ldots, a_{n-1}$ of items each with a weight $w_0, w_1, \ldots, w_{n-1}$ and a value $v_0, v_1, \ldots, v_{n-1}$.

Choose which items to take such that the total weight is less than or equal to C and the total value of the items is as large as possible.

Examples of Knapsack Problem

- Weight $W = [5; 4; 6; 3]$
  Value $V = [10; 40; 30; 50]$
  Knapsack capacity $= 10$

Examples of Knapsack Problem

- Weight W $=$ [5; 4; 6; 3]
  Value V $=$ [10; 40; 30; 50]
  Knapsack capacity $=$ 10

  Answer $=$ 90 (we take a_1 and a_3 with total weight $=$ 7)

Examples of Knapsack Problem

- Weight W = [5; 4; 6; 3]
  Value V = [10; 40; 30; 50]
  Knapsack capacity = 10

  Answer = 90 (we take a_1 and a_3 with total weight = 7)

- Weight W = [3; 5; 5; 6]
  Value V = [10; 60; 60; 100]
  Knapsack capacity = 10

Examples of Knapsack Problem

- Weight W = [5; 4; 6; 3]
  Value V = [10; 40; 30; 50]
  Knapsack capacity = 10
  Answer = 90 (we take a_1 and a_3 with total weight = 7)

- Weight W = [3; 5; 5; 6]
  Value V = [10; 60; 60; 100]
  Knapsack capacity = 10
  Answer = 120 (we take a_1 and a_2 with total weight = 10)

Brute-force solution

Brute-force Idea: Try all of the possibilities.
Running time: $O(2^n)$ - too slow.

Let's try thinking of a DP solution!

DP's four steps - Step 1

1.Define sub problems:

$S(i, k)$ - what is the maximum total value we can achieve using the first i items, with the chosen items total weight less or equal to k?

DP's four steps - Step 2

2.General formula:

Let's assume we know all solutions $S(i,k)$ for the first $i-1$ items.

We consider a new item with value v_i and weight w_i

For any total weight k we have 2 cases:

DP's four steps - Step 2

2.General formula:

Let's assume we know all solutions $S(i, k)$ for the first $i - 1$ items.
We consider a new item with value v_i and weight w_i
For any total weight k we have 2 cases:

- We don't take the item or if $w_i > k$
  $S(i, k) = S(i - 1, k)$

2.General formula:

Let's assume we know all solutions $S(i, k)$ for the first $i - 1$ items.

We consider a new item with value v_i and weight w_i

For any total weight k we have 2 cases:

- We don't take the item or if $w_i > k$
  $S(i, k) = S(i - 1, k)$
- We take the item and $w_i \leq k$
  $S(i, k) = S((i - 1, k - w_i) + v_i)$

DP's four steps - Step 2

2.General formula:

Let's assume we know all solutions $S(i, k)$ for the first $i - 1$ items.

We consider a new item with value v_i and weight w_i

For any total weight k we have 2 cases:

- We don't take the item or if $w_i > k$
  $S(i, k) = S(i - 1, k)$
- We take the item and $w_i \leq k$
  $S(i, k) = S((i - 1, k - w_i) + v_i)$

  Therefore our general formula is:

  $S(i, k) = max(S(i - 1, k), S((i - 1, k - w_i) + v_i)$

DP's four steps - Step 3

3.Find base case(s).

$S(0, k) = 0$ - using 0 items the best value we can have is 0

$S(i, 0) = 0$ - using i items the best value with weight 0 is 0

4.Relevant sub problem:
$S(n, C)$ - using n items what is the maximum total value with weight less or equal to C?

Knapsack - Dynamic Programming

```
int Knapsack_dp() {
    vector<vector<int> > dp(n+1, vector<int>(c+1, 0));
    for (int i=1; i<=n; i++) {
        for (int k=1; k<=c; k++) {
            if (w[i-1] <= k)
                dp[i][k] = max(v[i-1] + dp[i-1][k-w[i-1]],
                               dp[i-1][k]);
            else
                dp[i][k] = dp[i-1][k];
        }
    }
    return dp[n][c];
}
```

Dynamic programming solution - Run time analysis

For each element we have to calculate the best value with weight at most k. Therefore we have:

Running time of $O(nC)$

(where n is the number of elements and C is the knapsack capacity)

Final remarks

- Dynamic programming is tricky at first, because of the big overall picture

Final remarks

- Dynamic programming is tricky at first, because of the big overall picture
- Look at the small problems and base cases - solve them

Final remarks

- Dynamic programming is tricky at first, because of the big overall picture
- Look at the small problems and base cases - solve them
- Use the solution of these small problems to solve it for a slightly bigger one

Final remarks

- Dynamic programming is tricky at first, because of the big overall picture
- Look at the small problems and base cases - solve them
- Use the solution of these small problems to solve it for a slightly bigger one
- Using these small blocks construct your solution

Final remarks

- Dynamic programming is tricky at first, because of the big overall picture
- Look at the small problems and base cases - solve them
- Use the solution of these small problems to solve it for a slightly bigger one
- Using these small blocks construct your solution

Most importantly - practice! The more experience you have the easier and faster you'll see the concepts!