Brute force solutions

Competitive Programming: Core Skills

Artur Riazanov

SPbSU

Outline

Introduction

- Sometimes the first solution you come up with is the correct one.

Introduction

- Sometimes the first solution you come up with is the correct one.
- But sometimes it's not.

Introduction

- Sometimes the first solution you come up with is the correct one.
- But sometimes it's not.
- In this lesson we are going to develop a method for designing solutions which are always correct.

Introduction

- Sometimes the first solution you come up with is the correct one.
- But sometimes it's not.
- In this lesson we are going to develop a method for designing solutions which are always correct.
- The catch is they are going to be slow.

Digits ordering

Largest number

Input: list of digits.

Output: the largest number that can be made of the digits.

Digits ordering

Largest number

Input: list of digits.
Output: the largest number that can be made of the digits.

Sample input: 3,7,5
Sample output: 735.

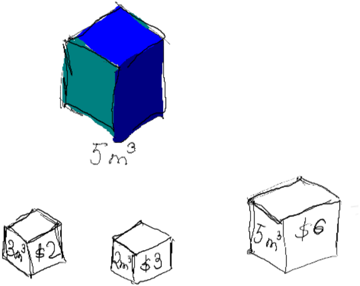
Digits ordering

Largest number

Input: list of digits.
Output: the largest number that can be made of the digits.

Sample input: 3,7,5
Sample output: 735.
The solution is to order the digits from the biggest one to the smallest one.

Robber's problem (aka knapsack problem)



Robber's problem

Input: a list of items with weights (kg) and costs ($) as well as the capacity of a bag (kg).
Output: the maximum total cost of items that fit in the bag.

Tempting Approach

- It's natural to process items in order of decreasing \$ per kg.

Tempting Approach

- It's natural to process items in order of decreasing $ per kg.
- Let's calculate utility $\frac{\text{cost}}{\text{weight}}$ for each item.

Tempting Approach

- It's natural to process items in order of decreasing \$ per kg.
- Let's calculate utility $\frac{\text{cost}}{\text{weight}}$ for each item.
- The better the utility the better the item.

Tempting Approach

- It's natural to process items in order of decreasing \$ per kg.
- Let's calculate utility $\frac{\text{cost}}{\text{weight}}$ for each item.
- The better the utility the better the item.
- Therefore we should try to put items with maximum utility first.

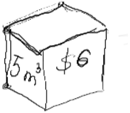
Tempting Approach

- It's natural to process items in order of decreasing \$ per kg.
- Let's calculate utility $\frac{\text{cost}}{\text{weight}}$ for each item.
- The better the utility the better the item.
- Therefore we should try to put items with maximum utility first.
- Nice and easy. But, unfortunately, wrong.

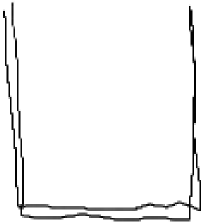
Example

3 kg
$2
$\frac{2}{3}$

2 kg
$3
$\frac{3}{2}$



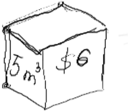
5 kg
$6
$\frac{6}{5}$

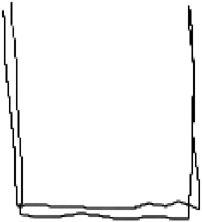


$V = 5$

Example

The Best





3 kg	2 kg	5 kg
\$2	\$3	\$6
$\frac{2}{3}$	$\frac{3}{2}$	$\frac{6}{5}$

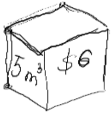
$V = 5$

Example

3 kg

\$2

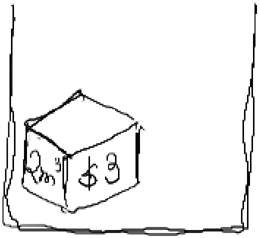
$\frac{2}{3}$



5 kg

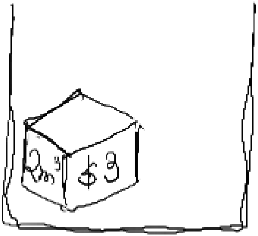
\$6

$\frac{6}{5}$



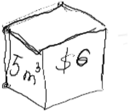
$V = 3;\ C = 3$

Example



The best

3 kg
\$2
$\frac{2}{3}$

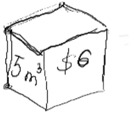


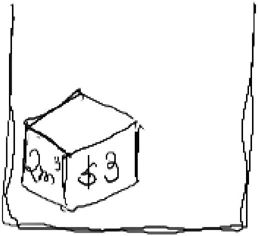
5 kg
\$6
$\frac{6}{5}$

$V = 3;\ C = 3$

Example

The best





3 kg
$2
$\frac{2}{3}$

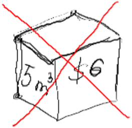
5 kg
$6
$\frac{6}{5}$

$V = 3; \, C = 3$

But the third item doesn't fit to the knapsack.

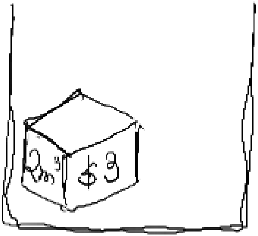
Example

The best



3 kg
$2
$\frac{2}{3}$

5 kg
$6
$\frac{6}{5}$



$V = 3; \; C = 3$

Example

3 kg

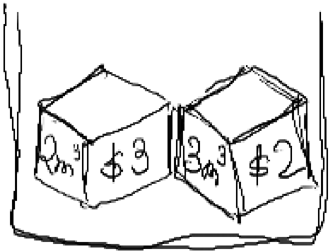
$2

$\frac{2}{3}$

5 kg

$6

$\frac{6}{5}$



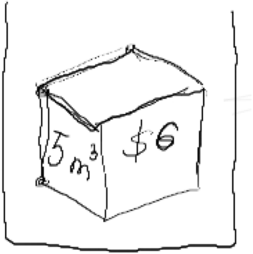
$V = 0;\ C = 5$

Example

- We got total cost 5.

Example

- We got total cost 5.
- But we could do better with the third item only:



How to prove

- Thus, the initial intuitive idea turned out to be wrong.

How to prove

- Thus, the initial intuitive idea turned out to be wrong.
- Then how to convince yourself that your approach is correct?

How to prove

- Thus, the initial intuitive idea turned out to be wrong.
- Then how to convince yourself that your approach is correct?
- The simplest thing to do is to check your algorithm with pen and paper against sample tests.

How to prove

- Thus, the initial intuitive idea turned out to be wrong.
- Then how to convince yourself that your approach is correct?
- The simplest thing to do is to check your algorithm with pen and paper against sample tests.
- But what to do if your solution got "wrong answer" verdict from the grader?

How to prove

- Thus, the initial intuitive idea turned out to be wrong.
- Then how to convince yourself that your approach is correct?
- The simplest thing to do is to check your algorithm with pen and paper against sample tests.
- But what to do if your solution got "wrong answer" verdict from the grader?
- It'd be good to have a solution which is always conceptually correct.

How to prove

- Thus, the initial intuitive idea turned out to be wrong.
- Then how to convince yourself that your approach is correct?
- The simplest thing to do is to check your algorithm with pen and paper against sample tests.
- But what to do if your solution got "wrong answer" verdict from the grader?
- It'd be good to have a solution which is always conceptually correct.
- And that's what we'll do!

Outline

Introduction

- In this video we will develop very powerful technique to solve problems, which works in almost all cases.

Introduction

- In this video we will develop very powerful technique to solve problems, which works in almost all cases.
- The approach yields slow solutions but it's conceptually correct by definition.

Introduction

- In this video we will develop very powerful technique to solve problems, which works in almost all cases.
- The approach yields slow solutions but it's conceptually correct by definition.
- Therefore it could be used to verify correctness of faster solutions for the same problem.

Search space

Almost every combinatorial problem falls in one of the following categories

1. Find an element of a set A satisfying some property (or a number of such elements).

Search space

Almost every combinatorial problem falls in one of the following categories

1. Find an element of a set A satisfying some property (or a number of such elements).
2. Find an element of a set A such that some objective function is minimized/maximized.

Search space

Almost every combinatorial problem falls in one of the following categories

1. Find an element of a set A satisfying some property (or a number of such elements).
2. Find an element of a set A such that some objective function is minimized/maximized.

We will call the set A search space.

Examples

Superstring

Given m strings $s_1, \ldots, s_m$ consisting of letters "a" and "b" only and an integer n. Find a string s of length n containing each s_i (for all $1 \leq i \leq m$) as a substring.

Examples

Superstring

Given m strings $s_1, \ldots, s_m$ consisting of letters "a" and "b" only and an integer n. Find a string s of length n containing each s_i (for all $1 \leq i \leq m$) as a substring.

Sample

Input: $m = 2$; $n = 3$; $s_1 =$ ab, $s_2 =$ ba

Output: aba (aba, aba) (another valid output is bab)

Search space

- One way to solve a problem is to simply go through all possible candidate solutions. For the superstring problem, the search space consists of all strings of length n over the alphabet $\{a, b\}$. For each such string, we check whether it is indeed a superstring of $s_1, \ldots, s_m$

Search space

- One way to solve a problem is to simply go through all possible candidate solutions. For the superstring problem, the search space consists of all strings of length n over the alphabet $\{a, b\}$. For each such string, we check whether it is indeed a superstring of $s_1, \ldots, s_m$
- Let's consider another testcase: $n = 4$, $s_1 = \text{bab}$, $s_2 = \text{abb}$.

Search space

- One way to solve a problem is to simply go through all possible candidate solutions. For the superstring problem, the search space consists of all strings of length n over the alphabet $\{a, b\}$. For each such string, we check whether it is indeed a superstring of $s_1, \ldots, s_m$
- Let's consider another testcase: $n = 4$, $s_1 = \texttt{bab}$, $s_2 = \texttt{abb}$.
- There are only $2^4 = 16$ strings of four letters "a" and "b".

Search space

Candidate	bab	abb	Candidate	bab	abb
aaaa	×	×	baaa	×	×
aaab	×	×	baab	×	×
aaba	×	×	baba	baba	×
aabb	×	aabb	**babb**	babb	babb
abaa	×	×	bbaa	×	×
abab	abab	×	bbab	bbab	×
abba	×	×	abba	×	×
abbb	×	abbb	bbbb	×	×

Examples

Maximum subarray problem

Given an array $a_1, \ldots, a_n$. What is the largest possible sum $a_l + a_{l+1} + \ldots + a_{r-1} + a_r$ for $1 \le l \le r \le n$?
Note that a_i could be negative.

Sample

...

Input: $a = (4, 1, -2, 3, -10, 5)$

Output: The best subarray is $(\underbrace{4, 1, -2, 3}, -10, 5)$ and the sum is $4 + 1 + (-2) + 3 = 6$.

Search space

- Search space for the maximum subarray problem is the set of all subarrays of the array a.

Search space

- Search space for the maximum subarray problem is the set of all subarrays of the array a.
- Subarray is determined by its first and last elements: l and r.

Search space

- Search space for the maximum subarray problem is the set of all subarrays of the array a.
- Subarray is determined by its first and last elements: l and r.
- For this problem understanding what the search space is instantly provides us with the solution.

Search space

- Search space for the maximum subarray problem is the set of all subarrays of the array a.
- Subarray is determined by its first and last elements: l and r.
- For this problem understanding what the search space is instantly provides us with the solution.
- Enumerate all pairs (l, r) such that $l \leq r$, for each pair compute the sum $a_l + \cdots + a_r$, and take the maximum.

Examples

Robber's problem

you have a knapsack of volume W and n items of volumes $w_1, \ldots, w_n$ and costs $c_1, \ldots, c_n$. What is the largest total cost of the set of items whose total weight does not exceed W?

Examples

Robber's problem

you have a knapsack of volume W and n items of volumes $w_1, \ldots, w_n$ and costs $c_1, \ldots, c_n$. What is the largest total cost of the set of items whose total weight does not exceed W?

Input: $V = 5$; $n = 3$

$v_1 = 3 \quad v_2 = 2 \quad v_3 = 5$

$c_1 = 2 \quad c_2 = 3 \quad c_3 = 6$

Output: The best solution is to put the last item to the knapsack and get the total cost 6.

Search space

- What is the search space for the robber's problem?

Search space

- What is the search space for the robber's problem?
- It's all sets of items.

Search space

- What is the search space for the robber's problem?
- It's all sets of items.
- For the given example, possible sets are the following: $\varnothing, \{1\}, \{2\}, \{3\}, \{1,2\}, \{1,3\}, \{2,3\}, \{1,2,3\}$.

Search space

- What is the search space for the robber's problem?
- It's all sets of items.
- For the given example, possible sets are the following: $\varnothing, \{1\}, \{2\}, \{3\}, \{1,2\}, \{1,3\}, \{2,3\}, \{1,2,3\}$.
- Not all of these sets fit into the backpack, but it's easy to check: compute the total weight of the set and check whether it exceeds the capacity of the backpack.

Search space

- What is the search space for the robber's problem?
- It's all sets of items.
- For the given example, possible sets are the following: $\varnothing, \{1\}, \{2\}, \{3\}, \{1,2\}, \{1,3\}, \{2,3\}, \{1,2,3\}$.
- Not all of these sets fit into the backpack, but it's easy to check: compute the total weight of the set and check whether it exceeds the capacity of the backpack.

Search space: summary

Problem	Search space
Superstring	strings consisting of letters "a" and "b"

Search space: summary

Problem	Search space
Superstring	strings consisting of letters "a" and "b"
Robber's problem	all possible sets of items

Search space: summary

Problem	Search space
Superstring	strings consisting of letters "a" and "b"
Robber's problem	all possible sets of items
Maximum subarray	pairs (l, r) such that $l \leq r$

Exploring the search space

- For the maximum subarray problem the search space gives us the solution instantly.

Exploring the search space

- For the maximum subarray problem the search space gives us the solution instantly.
- We can try all possible pairs with two nested `for` cycles.

Exploring the search space

- For the maximum subarray problem the search space gives us the solution instantly.
- We can try all possible pairs with two nested `for` cycles.
- For the substring problem we want to try all possible strings of n symbols.

Exploring the search space

- For the maximum subarray problem the search space gives us the solution instantly.
- We can try all possible pairs with two nested `for` cycles.
- For the substring problem we want to try all possible strings of n symbols.
- It'd be good to have n nested `for` cycles iterating through letters “a” and “b”.

n nested for cycles

Your pseudo-Python code will look like this for the superstring problem:

```
for x[0] in ['a','b']:
    for x[1] in ['a','b']:
        # ...
        for x[n-1] in ['a','b']:
            # check if x contains
            # all strings s[1], ... s[m]
```

Outline

1. Intuitive solutions
2. Search space
3. Backtracking

Introduction

- In this video we will finally understand how to write basic solution for combinatorial problems with backtracking.

Introduction

- In this video we will finally understand how to write basic solution for combinatorial problems with backtracking.
- Backtracking is roughly the way how to write n nested `for` cycles.

Enumerating all strings x over $\{a, b\}$ of length n:

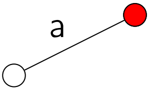
```
for x[0] in ['a','b']:
    for x[1] in ['a','b']:
        # ...
        for x[n-1] in ['a','b']:
            # do something with x
```

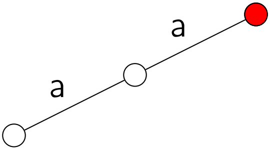
Recursion

Enumerating all strings x over $\{a, b\}$ of length n:

```
for x[0] in ['a','b']:
    for x[1] in ['a','b']:
        # ...
        for x[n-1] in ['a','b']:
            # do something with x
```

The simplest possible way to simulate this “code” with an actual code is via recursion.


a


a
a

Recursion

The key idea is to look at n nested `for` cycles like this:

```
for x[0] in ['a','b']:
    # remaining n-1 for cycles
```

Recursion

The key idea is to look at n nested `for` cycles like this:

```
for x[0] in ['a','b']:
    # remaining n-1 for cycles
```

So we can implement the function recursively.

Resursion

We will write the function `nestedFors` with additional parameter `firstFor` and it'll behave like

firstFor	Behaviour
0	`threeFors`
1	`twoFors`
2	`oneFor`
3	`print(x)`

```
def nestedFors(n, firstFor, x):
    if firstFor < n:
        for x[firstFor] in ['a','b']:
            nestedFors(n, firstFor + 1, x)
    else:
        print(x)
```

Resursion

We will write the function `nestedFors` with additional parameter `firstFor` and it'll behave like

firstFor	Behaviour
0	`threeFors`
1	`twoFors`
2	`oneFor`
3	`print(x)`

```
def nestedFors(n, firstFor, x):
    if firstFor < n:
        for x[firstFor] in ['a','b']:
            nestedFors(n, firstFor + 1, x)
    else:
        print(x)
```

Analyse the code

When we do have another `for` cycles to run:

```
if firstFor < n:
    for x[firstFor] in ['a','b']:
        nestedFors(n, firstFor + 1, x)
```

we call our function recursively, starting with the next `for`.

Analyse the code

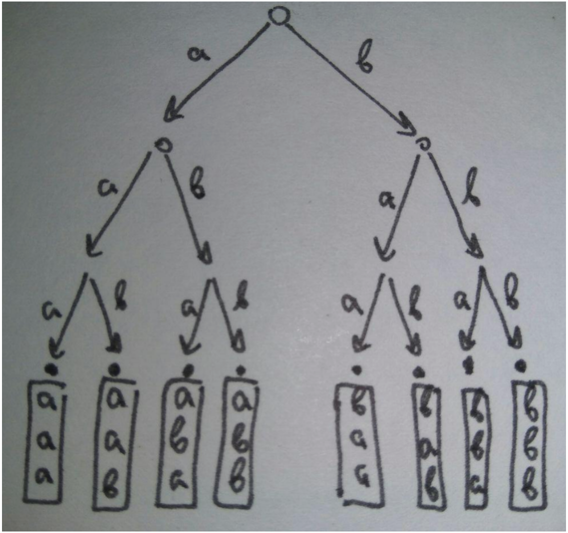
When we do have another `for` cycles to run:

```
if firstFor < n:
    for x[firstFor] in ['a','b']:
        nestedFors(n, firstFor + 1, x)
```

we call our function recursively, starting with the next `for`. Or we are in the deepest level of the recursion and we just do the job with x:

```
print(x)
```

Visualisation



Robber's problem

- Search space for the robber's problem is the set of all sets of items.

Robber's problem

- Search space for the robber's problem is the set of all sets of items.
- How to enumerate all sets of n items?

Robber's problem

- Search space for the robber's problem is the set of all sets of items.
- How to enumerate all sets of n items?
- Basically, it is the same as enumerating all strings over $\{0, 1\}$ of length n!

Set to string

Set	Items		
	1	2	3
$\varnothing$	0	0	0
$\{1\}$	1	0	0
$\{2\}$	0	1	0
$\{1,2\}$	1	1	0
$\{3\}$	0	0	1
$\{1,3\}$	1	0	1
$\{2,3\}$	0	1	1
$\{1,2,3\}$	1	1	1

Robber's problem

Recall our example: $n = 3$; $V = 5$ and

$v_1 = 3 \quad v_2 = 2 \quad v_3 = 5$

$c_1 = 2 \quad c_2 = 3 \quad c_3 = 6$

Items 1	2	3	Set	Weight	Cost
0	0	0	$\varnothing$	0	0
1	0	0	$\{1\}$	3	2
0	1	0	$\{2\}$	2	3
1	1	0	$\{1,2\}$	$2+3=5$	$3+2=5$
0	0	1	$\{3\}$	5	6
1	0	1	$\{1,3\}$	$3+5=8$	$2+6=8$
0	1	1	$\{2,3\}$	$2+5=7$	$3+6=9$
1	1	1	$\{1,2,3\}$	$2+3+5=10$	$3+2+6=11$

Robber's problem

Therefore we reduced robber's problem to the same n nested for cycles!

Conclusion

Designing a brute force solution:

1. Identify the search space
2. Design a way of enumerating all its elements
3. Turn it into a solution

The resulting solution is usually slow, but: it is clearly correct and can be used for debugging