# Why is the knapsack problem pseudo-polynomial?

I know that `Knapsack` is NP-complete while it can be solved by DP. They say that the DP solution is `pseudo-polynomial` , since it is exponential in the "length of input" (i.e. the numbers of bits required to encode the input). Unfortunately I did not get it. Can anybody explain that `pseudo-polynomial` thing to me slowly ?

language-agnostic complexity-theory dynamic-programming knapsack-problem

edited Sep 19 '12 at 12:21
Bill the Lizard ♦
**161k** 107 390 671

asked Dec 27 '10 at 12:19
Michael
**6,933** 17 58 131

**1** possible duplicate of How to understand the knapsack problem is NP-complete? – marcog Dec 27 '10 at 12:41

Thanks, I should have checked SO before. – Michael Dec 27 '10 at 12:44

See also en.wikipedia.org/wiki/Pseudo-polynomial_time – Dan Burton Mar 7 '11 at 17:39

add a comment

## 4 Answers

The running time is O(NW) for an unbounded knapsack problem with N items and knapsack of size W. W is not polynomial in the length of the input though, which is what makes it *pseudo*-polynomial.

Consider W = 1,000,000,000,000. It only takes 40 bits to represent this number, so input size = 40, but the computational runtime uses the factor 1,000,000,000,000 which is O($2^{40}$).

So the runtime is more accurately said to be O(N.$2^{\text{bits in W}}$), which is exponential.

Also see:

- How to understand the knapsack problem is NP-complete?
- The NP-Completeness of Knapsack
- Complexity of dynamic programming algorithm for the 0-1 knapsack problem
- Pseudo-polynomial time

edited May 6 '13 at 0:07
Manav Kataria
**1,334** 8 20

answered Dec 27 '10 at 12:35
marcog
**42.8k** 24 132 163

**1** I am a bit loss. Why is the input size 30 when it takes 40 bits to represent W? – Hery Jan 31 '13 at 2:17

**3** Why do they claim O(NW) in the first place then? – mcb Feb 5 '13 at 0:38

Does an *exact* solution (i.e. with `=` and not one with `<=` ) also run in pseudo-polynomial time. The wikipedia link shows the solution to a `<=` instance. – Jus12 Aug 28 '13 at 7:41

O(N*W) but why W=2^bits in W for the knapsack? – vibneiro Nov 21 '13 at 19:50

add a comment

The Knapsack algorithm's run-time is bound not only on the size of the input (n - the number of items) but also on the magnitude of the input (W - the knapsack capacity) O(nW) which is exponential in how it is represented in computer in binary (2^n) .The computational complexity (i.e how processing is done inside a computer through bits) is only concerned with the **size of the inputs,** not their **magnitudes/values**.

Disregard the value/weight list for a moment. Let's say we have an instance with knapsack capacity 2. W would take two bits in the input data. Now we shall increase the knapsack capacity to 4, keeping the rest of the input. Our input has only grown by one bit, but the computational complexity has increased twofold. If we increase the capacity to 1024, we would have just 10 bits of the input for W instead of 2, but the complexity has increased by a factor of 512. Time complexity grows exponentially in the size of W in binary (or decimal) representation.

Another simple example that helped me understand the pseudo-polynomial concept is the naive primality testing algorithm. For a given number n we are checking if it's divided evenly by each integer number in range 2..√n, so the algorithm takes √(n−1) steps. But here, n is the magnitude of the input, not it's size.

```
                     Now The regular O(n) case
```

By contrast, searching an array for a given element runs in polynomial time: O(n). It takes at most n steps and here n is the size of the input (the length of the array).

[ see here ]

Calculating bits required to store decimal number

edited Oct 16 '13 at 6:41 answered Oct 16 '13 at 5:50
shaunak1111
**192** 3 8

add a comment

---

In most of our problems, we're dealing with large lists of numbers which fit comfortably inside standard int/float data types. Because of the way most processors are built to handle 4-8 byte numbers at a time at no additional cost (relative to numbers than fit in, say, 1 byte), we rarely encounter a change in running time from scaling our numbers up or down within ranges we encounter in real problems - so the dominant factor remains just the sheer quantity of data points, the n or m factors that we're used to.

(You can imagine that the Big-O notation is hiding a constant factor that divides-out 32 or 64 bits-per-datum, leaving only the number-of-data-points whenever each of our numbers fit in that many bits or less)

But try reworking with other algorithms to act on data sets involving big ints - numbers that require more than 8 bytes to represent - and see what that does to the runtime. The magnitude of the numbers involved always makes a difference, even in the other algorithms like binary sort, once you expand beyond the buffer of safety conventional processors give us "for-free" by handling 4-8 byte batches.

The trick with the Knapsack algorithm that we discussed is that it's unusually sensitive (relative to other algorithms ) to the magnitude of a particular parameter, W. Add one bit to W and you double the running time of the algorithm. We haven't seen that kind of dramatic response to changes in value in other algorithms before this one, which is why it might seem like we're treating Knapsack differently - but that's a genuine analysis of how it responds in a non-polynomial fashion to changes in input size.

edited Oct 16 '13 at 16:00 answered Oct 16 '13 at 6:11
shaunak1111
**192** 3 8

add a comment

---

> So the runtime is more accurately said to be $O(N.2^{\text{bits in W}})$, which is exponential.

So that means by similar way we can also say-

> The runtime is more accurately be $O(N^{\text{bits in N}}.W)$, which is exponential.

Is that right?

answered Feb 25 '14 at 7:54
azizul fahim
**92** 9

add a comment

---

**Not the answer you're looking for? Browse other questions tagged language-agnostic complexity-theory dynamic-programming knapsack-problem or ask your own question.**