CS314 - Subset Sum

Note Title 3/5/2010

Announcements

- Reminder - check on scholarships for sophomores + juniors (in math/cs dept office)
- HW - due Tuesday by 4pm
- No class next week
- Check website over break
- Test is Friday after break

Subset Sum

Given a set X of positive integers $X[1..n]$ & a number T, decide if any subset of X sums to T.

Ex: $X = \{8, 6, 7, 5, 3, 10, 9\}$

$T = 15$

Answer? True

How did we set up the recursion?

$X[1]$ is either included in subset or not.

$\underbrace{X[2..n]}$ subset summing to either $\underline{T}$ or $\underline{T - X[1]}$

$O(2^n)$

$T(n) = 2T(n-1) + O(1)$

Define boolean function:

$S(i,t)$ = some subset of $X[i..n]$ sums to t

Goal: Compute $S(1,T)$

Recurrence:

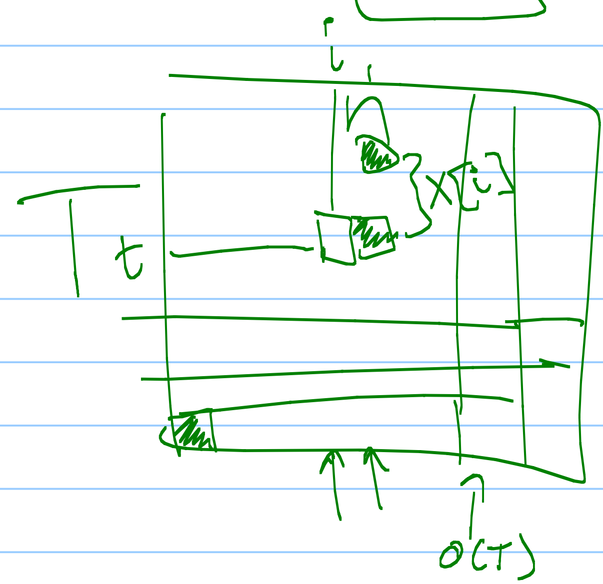
$$S(i,t) = \begin{cases} \text{TRUE} & \text{if } t = 0, \\ \text{FALSE} & \text{if } t < 0 \text{ or } i > n, \\ S(i+1,t) \vee S(i+1,t-X[i]) & \text{otherwise.} \end{cases}$$

$\boxed{S(i,t)}$

How many possible values for i?

at most $n+1$

What about t? $\boxed{T+1}$ (?)



Memoize :

If $S(i+1, t)$ & $S(i+1, t - X[i])$ are known, then filling in $\underline{S(i,t)}$ takes $O(1)$ time.

<u>Runtime</u>: $O(nT)$ since $O(1)$ time per entry of table.

<u>Space</u>: $O(nT)$

we could just keep 2 columns

↳ $O(T)$ space

Pseudocode:

```
SubsetSum(X[1..n], T):
    S[n+1, 0] ← True
    for t ← 1 to T
        S[n+1, t] ← False
    for i ← n downto 1
        S[i, 0] = True
        for t ← 1 to X[i] − 1
            S[i, t] ← S[i+1, t]        ⟨⟨Avoid the case t < 0⟩⟩
        for t ← X[i] to T
            S[i, t] ← S[i+1, t] ∨ S[i+1, t − X[i]]
    return S[1, T]
```

Wait a minute - subset sum is NP-Hard!

So did we 'just' solve P = NP ??

(No, we didn't)

Well, how does this compare to our first <u>recursive</u> algorithm?

$O(2^n)$ versus $\overbrace{O(nT)}^{\text{is really exponential also}}$

input size is $O(\cancel{n} + \log T)$

$n \log T$

Question: What is a hard problem?

So far, we've seen poly time algorithms
& exponential time algs.

- something we can't solve
- something difficult to implement
- something that can't be reduced
to smaller/simpler problems
- slower problems are harder

The Halting Problem

Q: Can we write a program which accepts as input another program & input, then decides if the program will run forever or halt on that input.

↑ NO

(So if it contains infinite loop, will run forever, for example, & our program will say that.)

This problem is undecidable.

Note: Our program can't just run the input program.

Why?

You're stuck in same infinite loop!

Thm: The halting problem is undecidable. (that is, no program to solve it can exist!)

pf: by contradiction

Assume we have $H(P, I)$ which outputs "halts" or "loops forever".

P = program, I = input

So we could run $H(P, P)$.

We'll use H to define a new function:

Define $K(P)$

- Run $H(P,P)$.
- If $H(P,P)$ outputs "halts", then $K(P)$ will run forever.
- If $H(P,P)$ outputs "runs forever", then $K(P)$ will halt.

Question: What happens when I run $K(K)$?

$K(K)$ runs $H(K,K)$

Contradiction! So H can't exist