

The Secret Language of Numbers

- Note: HW was adjusted Saturday morning

e.g. Making change (\$)

93 cents = 3 quarters,
1 dimes,
1 nickels,
3 pennies

Subset sum

Input: $x_1, x_2, \dots, x_n \in \mathbb{N}$

$T \in \mathbb{N}$ "target"

Goal: Find a subset of x_i 's that sum to T

i.e. distinct $i_1, i_2, \dots, i_k \in [n]$ st.

$$x_{i_1} + x_{i_2} + \dots + x_{i_k} = T$$

Another "search problem"

Ideas for algorithms?

$$x_1 \geq x_2 \geq \dots \geq x_n$$

"greedy"

↑
start

$$T \geq x_1?$$

Yes

→ go to x_2 , $T - x_1$

No

→ go to x_2 , T

$$T = 5,$$

x 's

~~4~~, 3, 2

Brute Force (by enumeration)

① For all $I \subseteq [n]$ $\leftarrow$ how many? 2^n

① if $\sum_{i \in I} x_i = T$ then return TRUE

② Return FALSE

Runtime: $O(2^n n)$

Brute Force by recursion

(declare what function should do)

Step 1 recursive spec

step 2 code

↑ easy w/ good spec

← why?

recursion = induction

recursive = induction
spec = hypothesis

Recursive Spec

$$\text{subset-sum}(\{x_1, \dots, x_n\}, T) =$$

True, if there is a subset of $\{x_1, \dots, x_n\}$ that sums to T , otherwise False

Code:

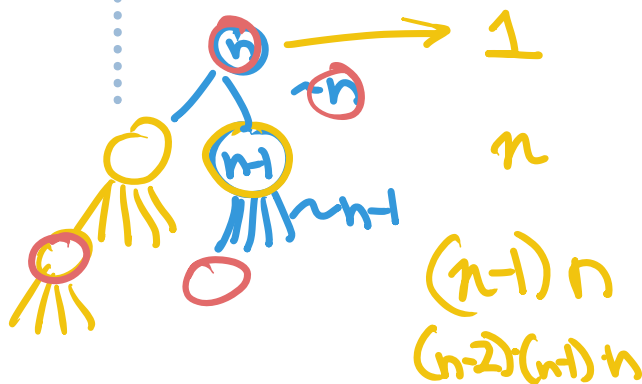
$$\text{subset-sum}(\{x_1, \dots, x_n\}, T) =$$
$$f(n) = n f(n-1) + O(1)$$
$$f(0) = 1$$

```

False if  $T < 0$ 
True if  $T = 0$ 
False if  $n = 0$ 
return
SS( $\{x_2, \dots, x_n\}, T - x_1$ )

```

~~False otherwise.~~


$$O(n!)$$

* Note: very simple code

Run-time:

$T(n)$ = run-time w/
n input #'s

then

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

subset-sum($\{x_1, \dots, x_n\}, T$) =

False if $T < 0$

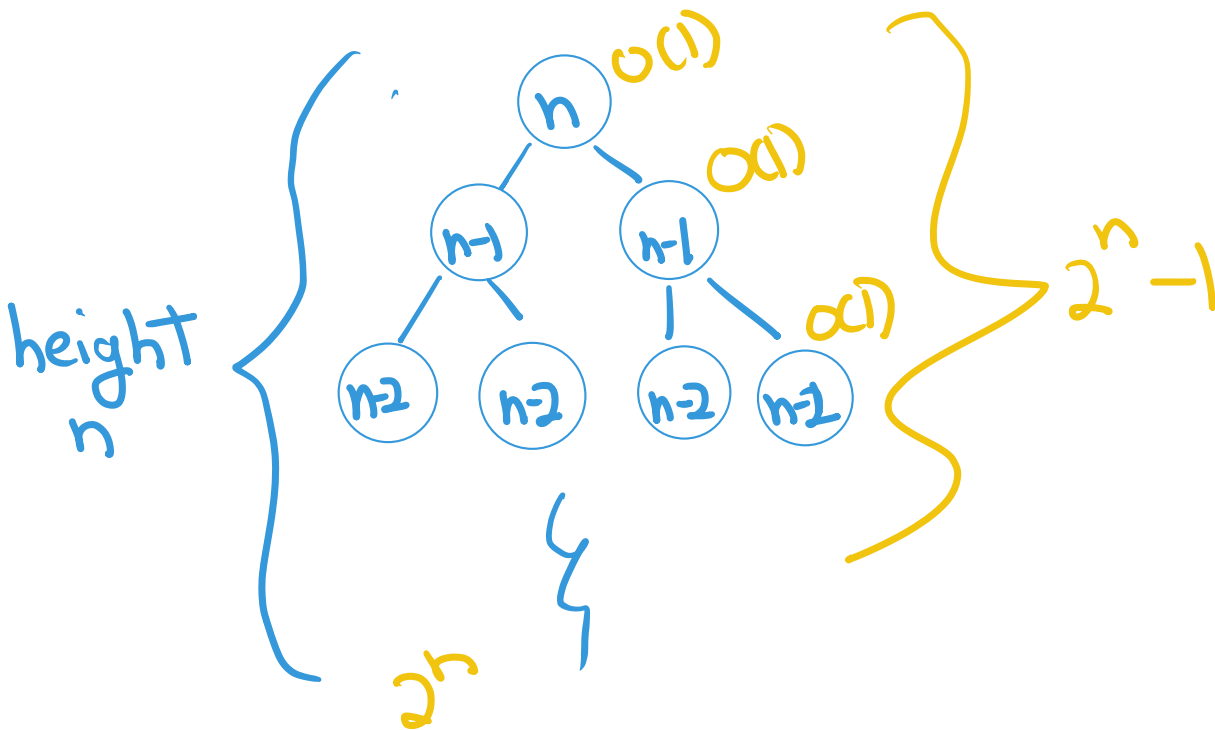
True if $T = 0$

False if $n = 0$

True if $SS(\{x_2, \dots, x_n\}, T - x_1) = \text{True}$

True if $SS(\{x_2, \dots, x_n\}, T) = \text{True}$

False otherwise



$2^{n+1} - 1$ nodes $\Rightarrow O(2^n)$ total work

$O(2^n)$ improves $O(n2^n)$

still not very good

Better?

Recursion + Caching

$$\text{subset-sum}(\{x_1, \dots, x_n\}, T) = \begin{cases} \text{False} & \text{if } T < 0 \\ \text{True} & \text{if } T = 0 \\ \text{False} & \text{if } n = 0 \\ \text{True} & \text{if } \text{S.S.}(\{x_2, \dots, x_n\}, T - x_1) = \text{True} \\ \text{True} & \text{if } \text{S.S.}(\{x_2, \dots, x_n\}, T) = \text{True} \\ \text{False} & \text{otherwise} \end{cases}$$

save answers $\Rightarrow$ never solve same
to ~~any~~ subproblem subproblem twice

Reorganize to simplify

Fix problem $(x_1, \dots, x_n)$ and T

① Define (for $i, S \in \mathbb{N}$)

$\text{subset-sum}(i, S) = \text{TRUE}$ if some subset
of $\{x_i, x_{i+1}, \dots, x_n\}$ adds to S ,
otherwise FALSE

② Code

(same algo as before)

$\text{subset-sum}(i, S) =$

- TRUE if $S = 0$
- FALSE if $S < 0$
- FALSE if $i > n$
- $\text{subset-sum}(i+1, S) \vee \text{subset-sum}(i+1, S - x_i)$
- otherwise

$n \times 2^n$

③ Allocate $n \times T$ -size 2-dimensional array
" $A[1, \dots, n, 1 \dots T]$ " ("A" for "Answers")
and save/check for answers in A

cached-subset-sum(i, S)

/ We assume there is an $n \times T$ array denoted $A[1..n, 1..T]$ (“A” for “Answer”) allocated in global memory, with all entries initially set to null. */*

1. If $S = 0$ then return true. ✓
2. If $S < 0$ then return false. ✓
3. If $i > n$ then return false. ✓
4. If $A[i, S]$ is null: ← ✓
 - A. $A[i, S] \leftarrow \text{subset-sum}(i + 1, S) \vee \text{subset-sum}(i + 1, S - x_1)$.
5. Return $A[i, S]$. ← ✓

Run time

(# subproblems) $\times$ (time per subproblem)

$$\boxed{nT} \times \boxed{O(1)} = \boxed{O(nT)}$$

(compare to $O(2^n)$ prev.)

Recursion + Caching = "Dynamic Programming"

Steps (required for class)

1. Recursive Spec

2. Pseudocode

3. say "caching" or "dynamic programming"

4. run time analysis

(often $(\# \text{ subproblems}) \times (\text{time per subproblem})$)

5. space analysis

(often $\# \text{ subproblems}$)

6. Initial call / how to solve orig. problem

(for us, "call subset-sum(1, T)")

7. Proof / justification of correctness

(when needed)



Flexible, easy recipe
for faster algorithms





Usually:

clean & correct #1

$\Rightarrow$ #2 easy

clean & correct #1, #2

$\Rightarrow$ #3~#7 easy

"Dynamic Programming"

1. Recursive Spec

2. Pseudocode

3. "caching"

4. run time

5. space

6. Initial call

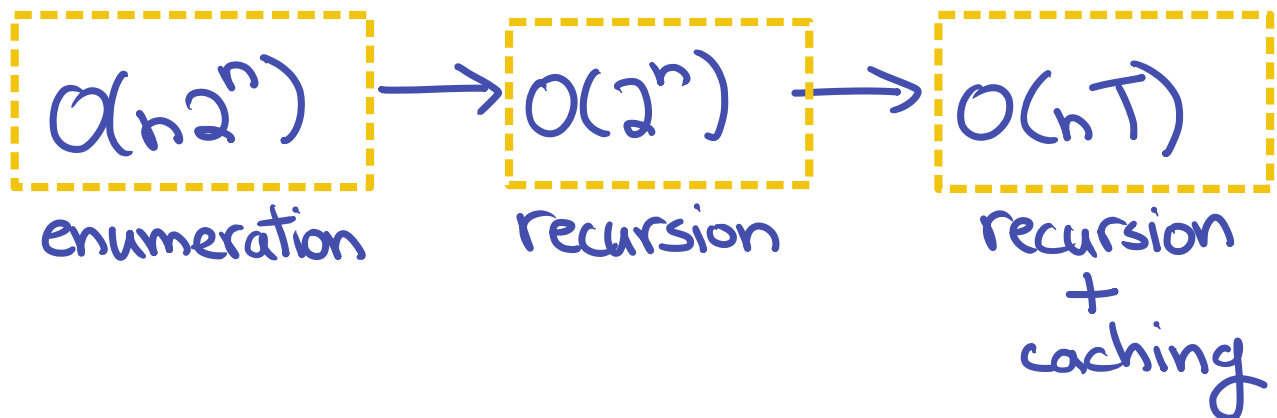
7. Proof

Zen:

#2 hard/awkward $\Rightarrow$ revisit, clean up #1

#3, ~, #7 hard/awkward $\Rightarrow$ revisit, clean up #1, #2

Like many things in this class,
sol'n clearer in hindsight



Is $O(nT)$ polynomial?

YES
60%

NO
10%

30%

Is $O(nT)$ polynomial? No

$$\text{size of } T = \boxed{\log_2 T} \text{ (bits)}$$

$$T \approx \underline{2^{\text{size}(T)}}$$

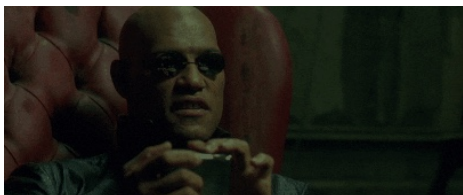
$\Rightarrow O(nT)$ is exponential in size of T

e.g. $T \approx 2^n \Rightarrow \text{size}(T) = O(n)$
 $O(nT) = (n2^n)$

Better?

How to show that a problem
should be hard?

Use it to solve another problem
that we are already convinced is hard



Theorem

poly time algo for subset sum



poly-time algo for SAT

(that would explain a lot!) $(x_1 \vee \bar{x}_2 \vee x_3) \wedge (\bigwedge_1 \sim v \cdot v)$

Approach: $\swarrow$ each clause has 3 variables

Input: 3-SAT formula $\mathcal{S}(x_1, \dots, x_n)$

n variables, m clauses

Output: Subset sum problem w/

$O(mn)$ #'s each w/ $O(mn)$ digits

s.t. subset sum
feasible



$\mathcal{S}(x_1, \dots, x_n)$
satisfiable

Key idea (trick): numbers are bit-strings too!

use many digits

each digit encodes 1 piece of logic

① Reinterpret 3SAT as "subset" problem

$X = \text{symbols } \{x_1, \bar{x}_1, x_2, \bar{x}_2, \dots, x_n, \bar{x}_n\}$

clause $(x_1 \vee \bar{x}_2 \vee x_3)$ = set $\{x_1, \bar{x}_2, x_3\} \subset X$

goal: Find a set $S \subset X$ s.t.

① for each clause C_i , $S \cap C_i \neq \emptyset$

② for each variable x_j , S contains
exactly 1 of $\{x_j, \bar{x}_j\}$

② start making #'s! " x_j " $\rightarrow$ # " A_j "

- we will use $m+n$ digits base 4

$A_j = \underbrace{0 \ 1 \ 1 \ 0 \ 1 \ 0 \ 0 \ 0 \ 1 \ 0 \ 0}_{m \text{ slots}} \underbrace{0 \ 1 \ 0 \ 0 \ 1 \ 0 \ 0}_{n \text{ slots}} \in \{0,1\}^{m+n}$

(models x_j)

m slots (1 per clause) C_i

n slots (1 per variable) x_j

- for each C_i , set C_i th digit to 1 if $x_j \in C_i$ (not $\bar{x}_j$), else 0
- set " x_j th digit" to 1, others to 0

Similar for $\bar{x}_j$ " $\bar{A}_j$ "

$\bar{A}_j = \underbrace{1 \ 0 \ 1 \ 0 \ 1 \ 0 \ 0 \ 1 \ 0 \ 0}_{m \text{ slots for each clause}} \in \{0,1\}^{mn}$

(models $\bar{x}_j$)

n slots for each variable x_j

- for each C_i , set C_i th digit to 1 if $\bar{x}_j \in C_i$ (not x_j), else 0
- set " x_j th digit" to 1, others to 0

Now we have

Literals $X = \{x_1, \bar{x}_1, \dots, x_n, \bar{x}_n\}$ $\longleftrightarrow$ Long #'s $\{A_1, \bar{A}_1, \dots, A_n, \bar{A}_n\}$

Set $S \subset X \longleftrightarrow \overset{\text{(sum)}}{D} = \sum_{x_j \in S} A_j + \sum_{\bar{x}_j \in S} \bar{A}_j$

- ① $\forall$ constraint $C_i, C_i \cap S \neq \emptyset \longleftrightarrow$ ① $\forall$ constraint $C_i, C_i \text{th digit of } D \geq 1$
- ② $\forall$ variables $x_j, \underline{|\{x_j, \bar{x}_j\} \cap S| = 0} \longleftrightarrow$ ② $\forall$ variable $x_j, x_j \text{th digit of } D = 1$

need to fix:

① $\forall$ constraint $C_i, C_i \text{th digit of } D \geq 1$

want equals

① $\forall$ constraint C_i ,
 C_i th digit of $D \geq 1$

want equals

Easy fix "Free padding"

For constraint C_i , create two copies of

$B_i = 00 \sim 01000 \sim$
 $\uparrow$
 C_i th digit

lets you increase C_i th digit from
 or

Finally, set

$T = \underbrace{333 \sim 3}_{m \text{ clauses}} \underbrace{111 \sim 1}_{n \text{ variables}}$

Literals

$$X = \{x_1, \bar{x}_1, \dots, x_n, \bar{x}_n\}$$

Long #'s

$$\{A_1, \bar{A}_1, \dots, A_n, \bar{A}_n\}$$

+ "padding" $\{B_1, B_1, B_2, B_2, \dots, B_m, B_m\}$

$$T = 3 \sim 3 \sim 1 \sim 1$$

$$\exists \text{ satisfying } S \iff \exists \text{ subset summing to } T$$

Note: easy to do construction in poly time

Theorem

poly time algo for subset sum



poly-time algo for SAT

(that would explain a lot!)

Approach:

Input: 3-SAT formula $\mathcal{S}(x_1, \dots, x_n)$

n variables, m clauses

Output: Subset sum problem w/

$3n+3m$ #'s each w/ $O(m+n)$ digits

s.t. subset sum
feasible



$\mathcal{S}(x_1, \dots, x_n)$
satisfiable

Key idea (trick): numbers are bit-strings too!

use many digits

each digit encodes 1 piece of logic

Worth pointing out:

good approximation algorithms for
subset sum (to be cont'd...)

hardness of subset sum is
useful for crypto (lattice based crypto...)

Approximating Subset Sum (for Fun)

if solving subset sum exactly is hard,
what if we allow 1% error?

Exercise 3.6. (Approximating subset sum.) Let $\epsilon \in (0,1)$ be fixed. Here we treat ϵ as a fixed constant (like $\epsilon = .1$, for 10% error); in particular, running times of the form $O(n^{O(1/\epsilon)})$ count as a polynomial.

A $(1 \pm \epsilon)$ -approximation algorithm for subset sum is one that (correctly) either:

1. Returns a subset whose sum lies in the range $[(1 - \epsilon)T, (1 + \epsilon)T]$.
2. Declares that there is no subset that sums to (exactly) T .

Note that such an algorithm does not solve the (exact) subset sum problem.

1. Suppose every input number x_i was “small”, in the sense that $x_i \leq \epsilon T$. Give a polynomial time $(1 \pm \epsilon)$ -approximation algorithm for this setting.
2. Suppose every input number x_i was “big”, in the sense that $x_i > \epsilon T$. Give a polynomial time $(1 \pm \epsilon)$ -approximation algorithm for this setting.
3. Now give a polynomial time $(1 \pm \epsilon)$ -approximation algorithm for subset sum in the general setting (with both big and small inputs).

Exercise 3.6. (Approximating subset sum.) Let $\epsilon \in (0,1)$ be fixed. Here we treat ϵ as a fixed constant (like $\epsilon = .1$, for 10% error); in particular, running times of the form $O(n^{O(1/\epsilon)})$ count as a polynomial.

A $(1 \pm \epsilon)$ -approximation algorithm for subset sum is one that (correctly) either:

1. Returns a subset whose sum lies in the range $[(1 - \epsilon)T, (1 + \epsilon)T]$.
2. Declares that there is no subset that sums to (exactly) T .

Note that such an algorithm does not solve the (exact) subset sum problem.

1. Suppose every input number x_i was “small”, in the sense that $x_i \leq \epsilon T$.
Give a polynomial time $(1 \pm \epsilon)$ -approximation algorithm for this setting.

Exercise 3.6. (Approximating subset sum.) Let $\epsilon \in (0,1)$ be fixed. Here we treat ϵ as a fixed constant (like $\epsilon = .1$, for 10% error); in particular, running times of the form $O(n^{O(1/\epsilon)})$ count as a polynomial.

A $(1 \pm \epsilon)$ -approximation algorithm for subset sum is one that (correctly) either:

1. Returns a subset whose sum lies in the range $[(1 - \epsilon)T, (1 + \epsilon)T]$.
2. Declares that there is no subset that sums to (exactly) T .

Note that such an algorithm does not solve the (exact) subset sum problem.

1. Suppose every input number x_i was “small”, in the sense that $x_i \leq \epsilon T$. Give a polynomial time $(1 \pm \epsilon)$ -approximation algorithm for this setting.
2. Suppose every input number x_i was “big”, in the sense that $x_i > \epsilon T$. Give a polynomial time $(1 \pm \epsilon)$ -approximation algorithm for this setting.

Exercise 3.6. (Approximating subset sum.) Let $\epsilon \in (0,1)$ be fixed. Here we treat ϵ as a fixed constant (like $\epsilon = .1$, for 10% error); in particular, running times of the form $O(n^{O(1/\epsilon)})$ count as a polynomial.

A $(1 \pm \epsilon)$ -approximation algorithm for subset sum is one that (correctly) either:

1. Returns a subset whose sum lies in the range $[(1 - \epsilon)T, (1 + \epsilon)T]$.
2. Declares that there is no subset that sums to (exactly) T .

Note that such an algorithm does not solve the (exact) subset sum problem.

1. Suppose every input number x_i was “small”, in the sense that $x_i \leq \epsilon T$. Give a polynomial time $(1 \pm \epsilon)$ -approximation algorithm for this setting.
2. Suppose every input number x_i was “big”, in the sense that $x_i > \epsilon T$. Give a polynomial time $(1 \pm \epsilon)$ -approximation algorithm for this setting.
3. Now give a polynomial time $(1 \pm \epsilon)$ -approximation algorithm for subset sum in the general setting (with both big and small inputs).

Chapter 3

Subset-Sum, and the secret language of numbers

Consider the following problem, called **subset sum**. You are given as input n natural numbers $x_1, \dots, x_n \in \mathbb{N}$, and a target value $T \in \mathbb{N}$. The goal is to identify a subset of the numbers that sum to T :

$$x_{i_1} + x_{i_2} + \dots + x_{i_k} = T,$$

for distinct indices $i_1, i_2, \dots, i_k \in [n]$. Subset sum is not an uncommon problem. We encounter it when we pay for groceries in cash and make exact change in quarters, dimes, nickels and pennies. It is also another example of a search problem. Given a candidate solution consisting of some numbers, we can easily verify that they are distinct numbers from the input, and that they add up to T .

3.1 Brute force algorithms

Let us design and analyze some algorithms for subset sum. We will focus on the decision version for simplicity but it will be easy to see how to modify our algorithms to return a feasible solution, if one exists.

Perhaps the simplest approach is brute force: enumerate all subsets of the numbers, and return true if any the subsets add up to T .

brute-force($\{x_1, \dots, x_n\}, T$)

1. For all subsets $I \subset [n]$
 - A. If $\sum_{i \in I} x_i = T$ then return true.
2. Return false.

This algorithm is (obviously) correct because it exhausts all possibilities. But it is also highly inefficient. The algorithm runs in roughly $O(n2^n)$ time, broken down as looping over 2^n subsets of $[n]$, and for each summing up to n numbers. As we have discussed before, $O(n2^n)$ will never terminate even for modest values of n .

3.1.1 Recursion

Towards a more efficient algorithm, let us rewrite the brute force algorithm recursively. When designing a recursive algorithm, it is good practice to specify our function semantically. Such a spec implicitly establishes the induction hypothesis that ultimately justifies the recursion. In this case, such a description is fairly straightforward.

subset-sum($\{x_1, \dots, x_n\}, T$) = true if there is a subset of $\{x_1, \dots, x_n\}$ that sums to T , and otherwise false.

Let us now implement what we have just defined.

subset-sum($\{x_1, \dots, x_n \in \mathbb{N}\}, T$)

1. If $T = 0$ then return true. *// The empty set sums to 0*
2. If $n = 0$ then return false. *// The empty set sums to 0*
3. If either one of
 - A. subset-sum($x_2, \dots, x_n, T$) or *// excludes x_1 from the subset sum*
 - B. subset-sum($x_2, \dots, x_n, T - x_1$) *// includes x_1 in the subset sum*returns true, then return true.
4. Otherwise return false.

It is easy to see the algorithm is correct by induction on n ¹. The algorithm still enumerates all possible combinations, although this time it proceeds through the combinations one variable at a time. Consider the first time we call the function on the input $x_1, \dots, x_n$ and T . The algorithm guesses whether or not to include x_1 , as follows. It first checks to see if there is a feasible solution that doesn't use x_1 , by recursing on all the remaining numbers with the same target value T . In the second recursive call, it checks to see if there is a feasible solution that uses x_1 , by recursing on the remaining numbers with a new target value of $T - x_1$. By induction on n these recursive calls will correctly decide if there is a subset sum for their respective subproblems. If either recursive call returns true, then

¹More precisely, assuming the specification holds inductively for subproblems, it is easy to see that the pseudocode implements the specification.

there is a subset sum. If neither recursive call succeeds, then since any solution must either include or exclude x_1 , we conclude that the subset sum problem is infeasible.

As per the running time, let $T(n)$ be the running time on a problem instance of n numbers. We have

$$\begin{aligned}T(0) &= O(1) \\T(n) &= 2T(n-1) + O(1)\end{aligned}$$

A quick recursion tree calculation (which reveals a binary tree of height n) shows that we have

$$T(n) = O(2^n).$$

This an improvement on the $O(n2^n)$ running time we started with, by a factor of n . While a factor of n improvement is usually cause for celebration, here it is insignificant relative to $O(2^n)$. We are still stuck at exponential time algorithm. Can we do better?

3.1.2 Caching

We can try to apply one of our favorite tricks: caching. In preparation for caching, let us index the arguments to our recursive function more compactly as follows. Fix an instance of subset sum, defined by $x_1, \dots, x_n \in \mathbb{N}$ and $T \in \mathbb{N}$. For $i, S \in \mathbb{N}$, we define

subset-sum(i, S) = true if there is a subset of $\{x_i, \dots, x_n\}$ that sums to S , and false otherwise.

The following pseudocode implements subset-sum based on the new definition above, and is just a more compact version of our previous pseudocode.

subset-sum(i, S)

1. if $S = 0$ then return true
2. if $S < 0$ then return false
3. if $i > n$ then return false
4. return subset-sum($i+1, S$) $\vee$ subset-sum($i+1, S - x_1$).

To solve our initial problem, we call subset-sum($1, T$).

Now, let us cache the answers to all our recursive calls so we never solve the same problem twice. This means that whenever we solve a subproblem for the first time, we also save the answer in some table or array. In every recursive call,

we first check this table to see if we have already solved the subproblem. If so, then we can return the saved answer. If not, then we execute the pseudocode above, and also store the answer before returning it.

Note that we don't have to cache the problems where $S \leq 0$ or $i > n$ because these answers are fixed. We only save problems of the form $\text{subset-sum}(i, S)$ where $i \in [n]$ and $S \in [T]$. The values can be stored in an $[n] \times [T]$ two-dimensional array (or in a hash table). The total size to store all the answers is $O(nT)$. The pseudocode below gives an example of such an implementation.

cached-subset-sum(i, S)

/ We assume there is an $n \times T$ array denoted $A[1..n, 1..T]$ ("A" for "Answer") allocated in global memory, with all entries initially set to null. */*

1. If $S = 0$ then return true.
2. If $S < 0$ then return false.
3. If $i > n$ then return false.
4. If $A[i, S]$ is null:
 - A. $A[i, S] \leftarrow \text{subset-sum}(i + 1, S) \vee \text{subset-sum}(i + 1, S - x_1)$.
5. Return $A[i, S]$.

After adding caching to the pseudocode, each problem takes only constant time (excluding recursive calls). Thus the total running time becomes

$$(\# \text{ subproblems}) \cdot \left(\begin{array}{c} \text{time per subproblem} \\ \text{excl. recursive calls} \end{array} \right) = nT \cdot O(1) = O(nT).$$

This running time looks much better than the previous, exponential running time of $O(2^n)$. (More on this in a moment.)

Our algorithm above only decides if a subset sum solution exists. Of course we may also be interested in the numbers that make up the sum. One approach to extracting the numbers is to use the decision algorithm as a black box; see exercise 3.2. (SAT was similarly *self-reducible*). A more efficient approach is as follows. In the algorithm above, each subproblem $\text{subset-sum}(i, S)$ that returns true came to that conclusion based on some immediate choice - namely, whether or not to include x_i , which leads to one of two subproblems that also returns true. Let us also record, in an additional table, the choice of subproblem that leads to true (when that is the case). Then, after solving the decision problem and building up this table, we have a trail of choices through a sequence of subproblems (starting from $\text{subset-sum}(1, T)$) that gives the actual set of numbers that add up to T .

3.1.3 Caching: a meta-algorithm

The difference in running time comes from the caching. It is a general technique that can be extended to all sorts of recursions, and is more commonly known as **dynamic programming**. The recipe is simple. Recursion, plus a little bit of memory to avoid redundancy, makes for more efficient algorithms.

The steps taken to convert the recursion into a more efficient dynamic programming algorithm were fairly mechanical and problem independent. When applying it to a problem, rather than going through these routine details again and again, the following points usually suffice to understand what is going on. (In this class, they are all required.) The following steps are also useful in helping you structure your thoughts and apply dynamic programming effectively, and with some practice, effortlessly.

1. A concise specification of the recursive function you are trying to implement, such as our one sentence description of `subset-sum( $i, S$ )`. It should be clear what is the input, and what is the desired output.
2. Some kind of pseudocode or formula that implements the specification, such as the four line code for `subset-sum( $i, S$ )` above.
3. Some explicit mention of the use of caching or dynamic programming to avoid recomputing subproblems.
4. An analysis of the total running time, which often breaks down as the number of subproblems times the amount of time per subproblem, excluding recursive calls.
5. An analysis of the space usage, which is often just the number of subproblems.
6. A brief description of the recursive call that produces that answer we are actually looking for. Above, this is `subset-sum( $1, T$ )`.
7. When needed, some justification for the correctness of the algorithm.

The critical thinking largely occurs in the first step, where we specify the function we want to implement. With a clear specification, the implementation in the second step can be straightforward. This is because recursive algorithms leverage induction, and a clearer specification gives a more useful induction hypothesis. Conversely, if it is difficult to implement the code in step 2, this may indicate that the specification is not quite right.

Once steps 1 and 2 are done correctly and clearly, then the remaining steps can be straightforward. In particular, the correctness of the algorithm is often

self-evident and hardly requires a proof. If one of the later steps feels awkward or difficult, then this might be best addressed by clarifying the specification or the pseudocode.

Our first example of dynamic programming, subset-sum, was relatively simple. But in upcoming discussions we will see how versatile this tool is, and the large design space it encumbers.

3.1.4 A polynomial running time?

Let us now return to subset-sum. Recall that our first goal for any problem is try to establish a running time that is a polynomial function of the input size. Do we now have a polynomial time algorithm for subset sum? $O(nT)$ might look like a polynomial, but in fact it is not a polynomial in the input size.

The catch harkens back to our very first discussion about bits: the *bit complexity* of T is only $\log(T)$. That is, we only need $\log(T)$ bits to express T . Thus $O(nT) = O\left(n2^{\log(T)}\right)$ is actually *exponential* in the input size of T . For example, if $T = 2^n$, then our $O(nT)$ running time is still exponential in n , while the input size of T is only $O(n)$. So for all the hoopla about dynamic programming, and the practical improvements for small T , we have not actually succeeded in obtaining a polynomial time algorithm.

3.2 Packing in the complexity

Subset sum doesn't look complicated, but it has resisted a number of attempts to obtain a (truly) efficient algorithm. To alleviate our frustration, let us instead see if we can find good evidence that subset sum is actually *hard*; that is, unlikely to have a polynomial time algorithm.

How can I convince you that subset sum is hard? After all, it is a very simple looking problem. Quite probably all instances you've encountered in real life (namely, making change for cash) were very easy. And so what if we're stumped. We have overcome problems before. Surely, all we need is a good attitude, a hot cup of coffee, and copious amounts of scratch paper!

Recall the SAT problem, where we are given a 3-SAT formula and we want to find a satisfying assignment. Now *there's* a problem. Since we can reduce any general Boolean formula to a 3-SAT formula of roughly the same size, solving 3-SAT is tantamount to solving *anything* that can be expressed in Boolean logic. There is no shame in admitting that 3-SAT is too hard to expect a polynomial time algorithm for. So what if I told you that

any algorithm for subset sum can be efficiently repurposed to solve 3-SAT.

That is, a polynomial time algorithm for subset-sum *would imply a polynomial time algorithm for 3-SAT*. That would be a satisfying explanation for our inability to find an efficient algorithm for subset sum: the problem is actually much more important than we realized.

The main theorem of this section asserts precisely this, as follows.

Theorem 3.1. *If there is a polynomial time algorithm for subset sum, then there is a polynomial time algorithm for SAT.*

We have already seen that SAT and 3-SAT are polynomial time equivalent. To prove the theorem, we will describe a *polynomial time reduction* from 3-SAT to subset sum.

Given a 3-SAT formula $f(x_1, \dots, x_n)$, we will construct a subset sum problem that is feasible iff f is satisfiable. This might seem ridiculous at first. Subset sum uses only basic arithmetic, and arithmetic seems crude compared to Boolean logic. But it is indeed possible. The (simple yet profound) trick is to realize that *integers are bit strings too*: two numbers are equal iff their bit strings are equal. At a high level, we will create a subset sum problem where each number has a large number of digits. Each digit encode a different logical constraint. A feasible subset sum implies that every digit matches, which means that all logical constraints are met.

Consider an instance of 3-SAT given by a formula $f(x_1, \dots, x_n)$ with n variables and m disjunctive clauses, where each clause has three distinct variables, possibly negated. We let $C_1, \dots, C_m$ denote the m clauses, and treat them as subsets of the set of symbols $X = \{x_1, \bar{x}_1, \dots, x_n, \bar{x}_n\}$. For example, a clause $(x_1 \vee \bar{x}_2 \vee x_3)$ corresponds to the set $\{x_1, \bar{x}_2, x_3\}$. In this way, we can recast 3-SAT problem as a “subset” problem as follows:

Find a set $S \subset X$ that satisfies the following constraints.

1. *For each constraint $C_i \subset X$, $X \cap S \neq \emptyset$.*
2. *For each Boolean variable x_j , S contains exactly one of x_j or $\bar{x}_j$.*

Above, the set $S \subset X$ indicate whether we assign true or false to each variable x_j . By item 2, for each j , S contains either x_j or $\bar{x}_j$. We interpret $x_j \in S$ as assigning $x_j = \text{true}$ and $\bar{x}_j \in S$ as assigning $x_j = \text{false}$. With this setup, item 1 is really saying that (the assignment corresponding to) S has to satisfy every clause.

As mentioned above, we will use numbers with a large number of digits and use each digit to encode some logic. In particular, we will create one digit for each constraint on S listed above. We will have 1 digit for each constraint C_i (corresponding to the constraints in item 1) and 1 digit for each variable x_j

(corresponding to the constraints in item 2). Thus $m + n$ digits in all. We will also work with numbers in base 4 (without loss of generality). One can alternatively think of each $(m + n)$ -digit, base-4 number as an $(m + n)$ -length string composed of one of 4 letters; i.e., in $\{0,1,2,3\}^{m+n}$.

We will create $2n + 2m$ numbers – one number for each x_j and each $\bar{x}_j$ in X , as well as $2m$ additional variables (two for each clause). For clarity, we will denote the numbers we create by capital letters $A, B, C, \dots$. For each x_j , we will create a number A_j as follows. Recall that we have reserved a digit for each constraint and a digit for each variable. We describe A_j in terms of the value of each digit (out of 4).

1. For each constraint C_i , if $x_j \in C_i$, we set the C_i th digit (i.e., the digit corresponding to C_i) of A_j to 1. Otherwise we set that digit to 0.
2. We set the digit corresponding to x_j to 1.
3. We set the remaining digits corresponding to other variables x_k ($k \neq j$) to 0.

For each negated symbol $\bar{x}_j \in X$ we create a number $\bar{A}_j$ similarly and with the obvious adjustments:

1. For each constraint C_i , if $x_j \in C_i$, we set the C_i th digit (i.e., the digit corresponding to C_i) of $\bar{A}_j$ to 1. Otherwise we set that digit to 0.
2. We set the digit corresponding to x_j to 1.
3. We set the remaining digits corresponding to other variables x_k ($k \neq j$) to 0.

This creates the first $2n$ numbers and we have promised to create $2m$ more. To motivate these remaining variables let us review what we have already created. We claim that

A set $S \subset X$ is a satisfying assignment iff, letting

$$D = \sum_{x_j \in S} A_j + \sum_{\bar{x}_j \in S} \bar{A}_j$$

denote the sum of numbers corresponding to symbols in S , we have the following:

1. *For every variable x_j , the x_j th digit of D is exactly 1.*
2. *For every constraint C_i , the C_i th digit of D is at least 1.*

This is not quite a subset sum problem, but I think it is fair to say we are getting closer. We have two sets of constraints on D . The first family, corresponding to the variables x_j , is naturally suitable for subset sum, as it specifies *exactly* what the numbers have to add up to for each of each digits. However, the second set of constraints, for the digits corresponding to D , are based on inequalities rather than equalities. The issue is that, in 3-SAT, we could satisfy a clause C_i by taking one, two, or all three of the participating variables. Thus, for the corresponding digit, the values $\{1, 2, 3\}$ should all be allowed. How do we express the fact that we allow for a *range* of digits, in the native language of subset sum?

The following adjustment effectively implements *padding*. For each constraint C_i , we create two copies of the same number, called B_i . B_i has value 1 in the C_i th digit and value 0 in all the other digits. The role of B_i is simply to allow us to artificially increase the C_i th digit of D . In particular, with two copies of B_i , we can add up to two units to the C_i th digit of D . Values of 1 or 2 can easily be converted to 3. Crucially, a value of 0 cannot be increased to 3.

We now select our target value T . For each clause C_i , we set the C_i th digit of T to 3. For each variable x_j , we set the x_j th digit of T to 1. Note that the size of our subset sum problem is polynomial in the size of the 3SAT formula.

In conclusion, our construction satisfies the following claim, which establishes theorem 3.1.

The Boolean formula f is satisfiable iff there is a subset of the A_j 's, $\bar{A}_j$'s, and (2 copies each of the) B_i 's (as defined above) that adds up to T .

3.3 Perfect is the enemy of good

Let us pause and reflect on our developments. We started with a simple and common problem: subset sum. We have some brute force algorithms, which when combined with caching (dynamic programming), leads to a reasonable running time for small target values. But this running time was not strongly polynomial; for large numbers, the algorithm can be very slow. The basic reason our algorithm did not qualify as polynomial time is that a very large integer x requires only $\lceil \log x \rceil$ many bits to represent it, and ultimately we seek running times that are polynomial in the input *size*.

Our reduction from 3SAT to subset sum suggests that subset sum is much harder to solve than one might have expected. Moreover, we see explicitly in the reduction how the many bits of large numbers can be used to encode the rich complexity of a 3SAT problem.

Let us also point out that the difficulty of subset sum lies in our insistence on an *exact* answer to the problem. By contrast, suppose we had an algorithm that satisfies the following approximation guarantee:

If the subset sum is feasible, then return a subset $x_{i_1}, \dots, x_{i_k}$ whose sum lies in the range $[.99T, 1.01T]$. If the subset sum problem is infeasible, then either return a subset sum in the range $[.99T, 1.01T]$, or output that it is infeasible.

For many real applications, where the sums represent a measurable quantity, being off by a little is not the end of the world. On the other hand, since an approximate sum in the above sense does not imply an exact solution, it does not risk deciding a SAT problem in disguise. This gives hope for a polynomial time approximation algorithm. Indeed, there *are* polynomial time approximation algorithms that satisfy the above guarantee, even if we replace $[.99T, 1.01T]$ with $[(1 - \epsilon)T, (1 + \epsilon)T]$ for any fixed error parameter $\epsilon \in (0, 1)$. (See exercise 3.6.) We may discuss approximation algorithms later, but for the moment, it is good to be aware of their existence and motivation.

The other side of the coin - where the hardness of subset sum is *useful* - is cryptography. Loosely speaking, we can interpret the solution to a subset sum problem as a password - it is hard to guess (as far as we know), but easy to verify. Many other problems have this characteristic (as we will see) but subset sum is particularly simple which has some computational advantages. The Merkle-Hellman system, one of the first public-key cryptographic systems, is based on subset sum, and in particular relies on the assumption that subset sum is hard. Lattice based cryptography, a more modern cryptographic technique, is (at a high level) based on the difficulty of multi-dimensional extensions of subset sum. See (e.g.) the notes [Pei13] for more details.

3.4 Exercises

Exercise 3.1. Suppose instead that we had reduced from 1-in-3-SAT to subset sum. How might this have simplified our construction in section 3.2?

Exercise 3.2. At the end of section 3.1.2, we discussed how to use an additional table to also extract the subset sum solution. Here we develop an alternative approach that was briefly alluded to in section 3.1.2. It is not as efficient as building a secondary table, but the technique is more general and of independent interest.

Suppose one had a black box algorithm for deciding subset sum problems. That is, given an instance of subset sum described by n integers $x_1, \dots, x_n \in \mathbb{N}$ and an additional target value T , the algorithm returns true if there exists a feasible solution, and otherwise false. Show that one can use this decision algorithm as a subroutine to obtain an efficient constructive algorithm for subset sum.

Exercise 3.3. Consider the following special case of subset-sum called the *partition problem*. The input consists of n numbers $x_1, \dots, x_n \in \mathbb{N}$. The goal is to partition the n numbers into two parts so that the sums of each part are equal. Either design an efficient algorithm for the partition problem, or show that the partition problem is at least as hard as 3-SAT.

Exercise 3.4. Consider the following variation of subset sum which allows for an additive error of 1. The input consists of n numbers $x_1, \dots, x_n \in \mathbb{N}$ and a target $T \in \mathbb{N}$, like subset sum. The goal is to decide if there is a subset of $x_1, \dots, x_n$ that sums to either $T - 1$, T , or $T + 1$. Either (a) design an efficient algorithm for this variant of subset sum, or (b) show that an efficient algorithm for this problem implies an efficient algorithm for SAT.

Exercise 3.5. Consider the following variation of subset-sum called *Kobe subset sum*. The input is similar to normal subset sum, with n numbers $x_1, \dots, x_n \in \mathbb{N}$ and $T \in \mathbb{N}$, except we are promised that each number x_i lies in the range $\{8, \dots, 24\}$. Either design an efficient algorithm for Kobe subset sum, or show that Kobe subset sum is at least as hard as 3-SAT.

Exercise 3.6. (Approximating subset sum.) Let $\epsilon \in (0, 1)$ be fixed. Here we treat ϵ as a fixed constant (like $\epsilon = .1$, for 10% error); in particular, running times of the form $O(n^{O(1/\epsilon)})$ count as a polynomial.

A $(1 \pm \epsilon)$ -approximation algorithm for subset sum is one that (correctly) either:

1. Returns a subset whose sum lies in the range $[(1 - \epsilon)T, (1 + \epsilon)T]$.
2. Declares that there is no subset that sums to (exactly) T .

Note that such an algorithm does not solve the (exact) subset sum problem.

1. Suppose every input number x_i was “small”, in the sense that $x_i \leq \epsilon T$. Give a polynomial time $(1 \pm \epsilon)$ -approximation algorithm for this setting.
2. Suppose every input number x_i was “big”, in the sense that $x_i > \epsilon T$. Give a polynomial time $(1 \pm \epsilon)$ -approximation algorithm for this setting.
3. Now give a polynomial time $(1 \pm \epsilon)$ -approximation algorithm for subset sum in the general setting (with both big and small inputs).