Advanced Algorithms

Lecture 6: Dynamic programming

Announcements

- HW 1 due **<u>tomorrow</u>**
- *Video for lecture 3 <u>now up</u>!*

Last few lectures

- Divide and conquer
- Recurrences
- Many problems — merge-sort, integer/matrix multiplication, “fast selection”, ...

Dynamic programming

Toy problem: subset sum

Problem: given *n* **non-negative** integers A[0], A[1], …, A[n-1] and a "target" *S*, find if there is a <u>subset</u> of the A[i] that add up to *S*

(try every subset $\rightarrow$ $2^n \cdot \text{poly}(n)$).

- Brute force?
- Divide and conquer?

$$\overbrace{A_0 \cdots}^{t} \;\Big|\; \overbrace{\quad A_{n-1}}^{S-t}$$

$$\underbrace{\qquad}_{n/2} \qquad \underbrace{\qquad}_{n/2}$$

try all possible values for t.

Simple example of "sequential decision making"…

$$T(n) = 2(S+1)\,T\left(\frac{n}{2}\right) + c \rightsquigarrow \text{poly}(n) \cdot (S+1)^{\log n}.$$

Divide and conquer

Gives running time: $(S+1)^{\log n} \cdot \mathrm{poly}(n)$.

[Can be much better than 2^n]

Recursive procedure

$A = \{ \underline{A[0]}, \overbrace{A[1], \ldots, A[n-1]} \}$; target $\underline{\underline{S}}$.

use $A[0]$ — don't use $A[0]$

$\{A[1], \ldots, A[n-1]\}$; target $= S - A[0]$ $\quad$ $\{A[1], \ldots, A[n-1]\}$, target $= S$

$\{\underline{\underline{A[2]}}, \ldots\}$ .. $A[n-1]\}$

how long does the procedure take? $\quad$ 2^n time $\rightarrow$ exactly the same as brute-force.

$\{A[1], A[3], A[5]\}$?

Recursive procedure

What are sub-problems?

What is running time?

$\{A[r], A[r+1], \ldots, A[n-1]\}$; target : $S - \ldots$,

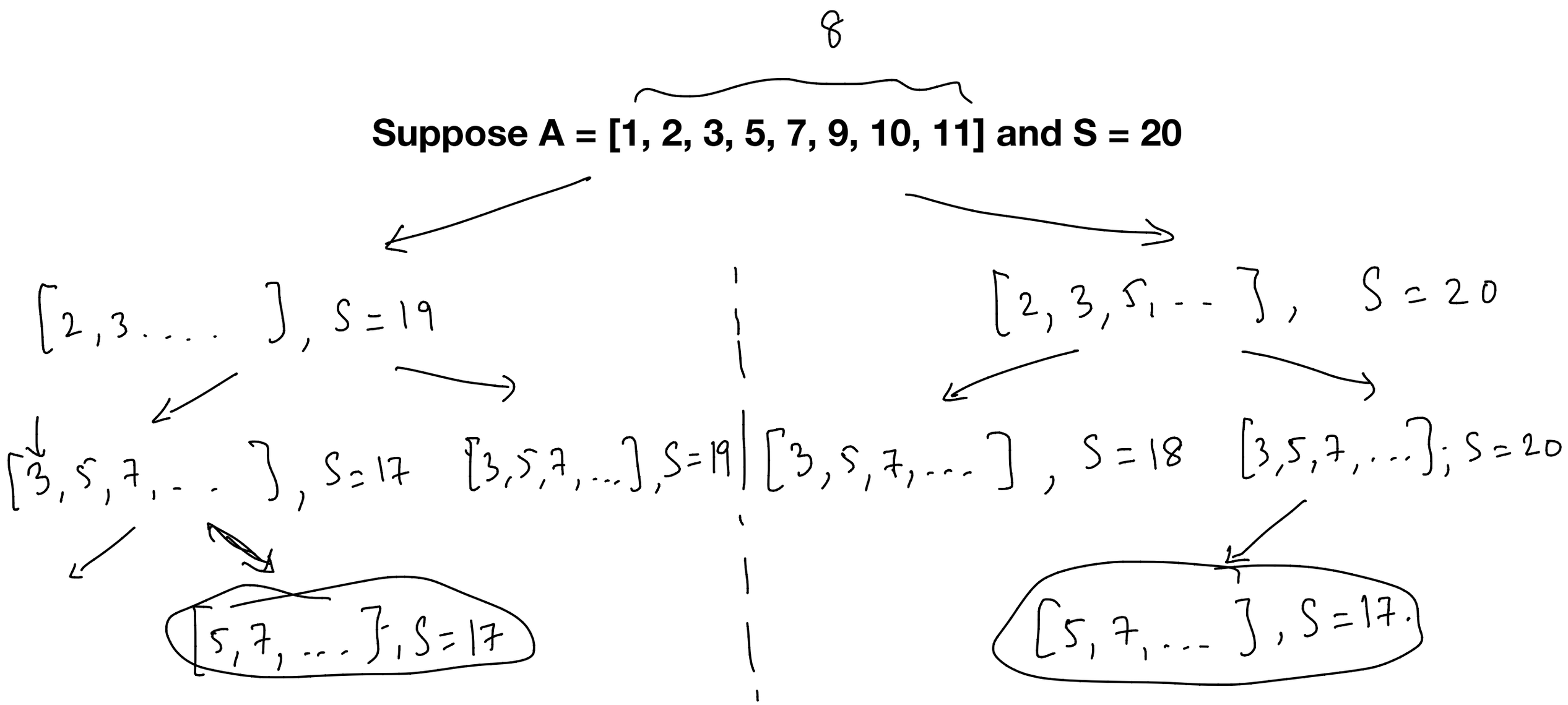
for some r.

→ Some suffix of the original array.

→ target is some integer between 0 & S.

Looking closer – example

Suppose A = [1, 2, 3, 5, 7, 9, 10, 11] and S = 20



target sum is always an integer between 0 . . . . S.

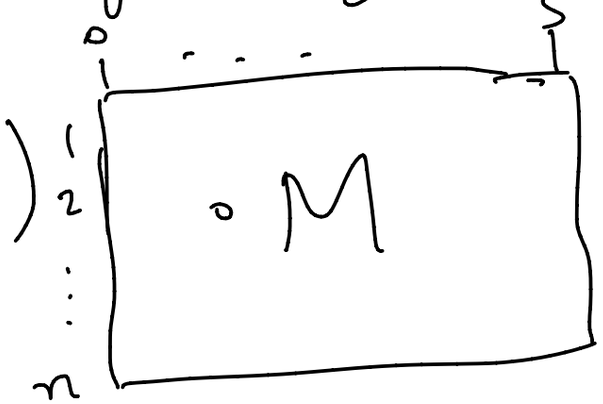
$\Rightarrow$ at every level, there are only (S+1) distinct sub-problems.

Avoiding duplicate solves

- Can we "store answers"?
- How many sub-problems are there in total?

each sub-problem is of the form $[A[r], \ldots, A[n-1]]$

Sub-problem $\equiv$ (r, s')



target $= s'$,

$0 \le s' \le S$.

A total of only $n(S+1)$ sub-problems.

Modified procedure

SubsetSum(start-index $= r$, target sum $= S$):

- Compute / look-up subsetSum($r+1$, $S - A[r]$) → need to check if $S \geq A[r]$.
- Compute / look-up subsetSum($r+1$, S)
- return YES if either answer was YES

  else return NO.

Compute / Look-up : check if answer exists in the array M, If yes, return $M[r, S]$, else run the recursive-procedure.

Running time

→ # of sub-problems in total is $n(S+1)$.

→ each sub-problem is only being solved <u>once</u>.

→ ~~over~~ all the work is done in the process of solving some sub-problem.

Overall run time $= O(nS)$

$A[0], \ldots, A[n-1]$; target $= S$.

$S = \underline{315{,}791{,}074{,}32}$

$\boxed{3157910}$

Is this polynomial?

↳ in input size?

$\{A[0], \ldots, A[n-1]\}, S$

input size:

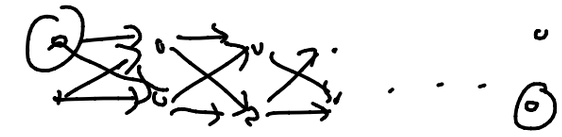
$\log(A[0]) + \log(A[1]) + \ldots + \log A[n-1] + \log S.$

run time $= O(n S)$

$n = 1000$

$S \times 1000$

100-digit #



Example 2: shortest path

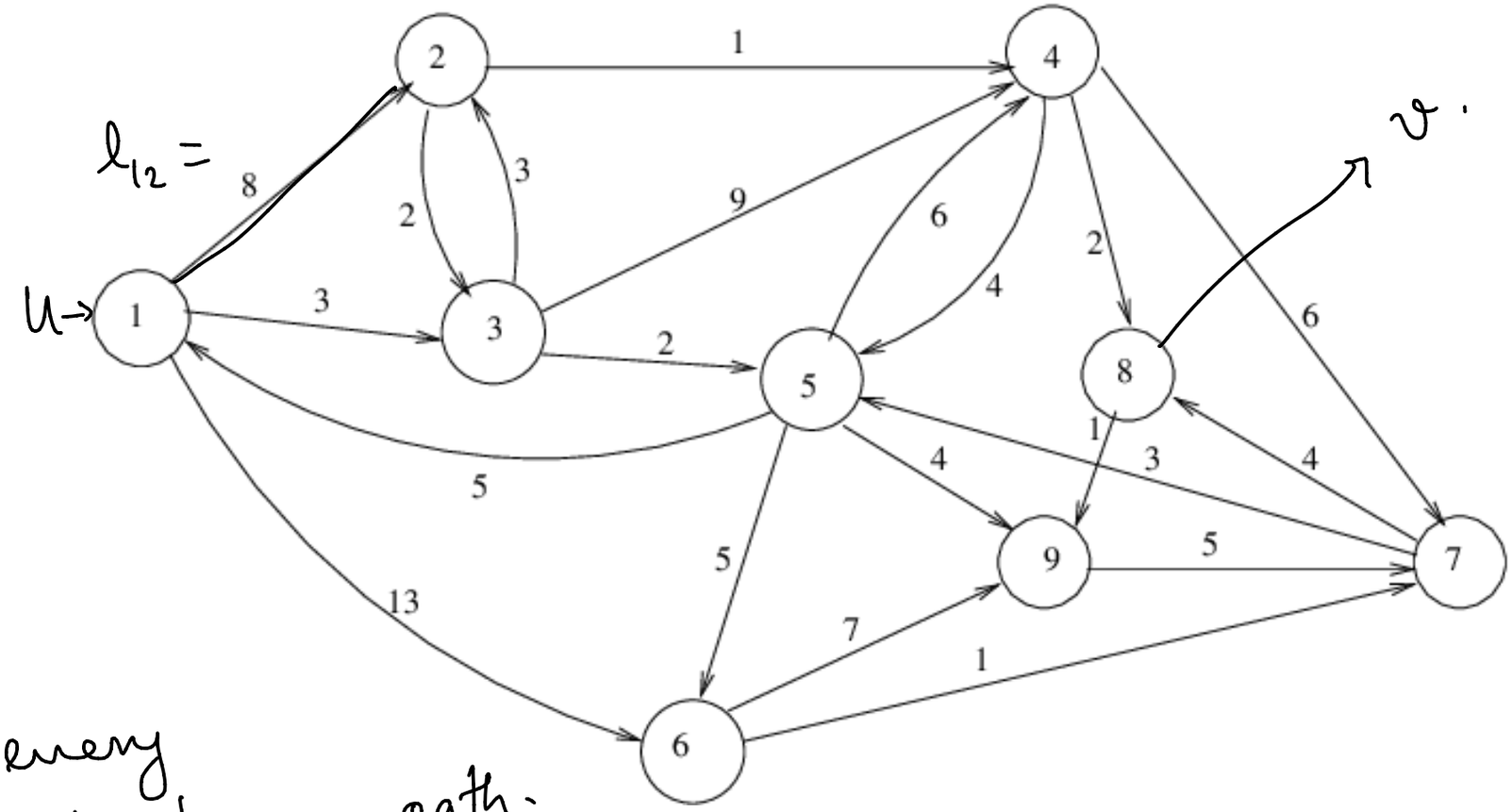
n : number of vertices

m : # edges.

Problem: given a directed graph G = (V, E), two nodes u, v, and a parameter *L*, find the shortest path of ~~length~~ *L* from u —> v

total length.

hops → # vertices on the path.

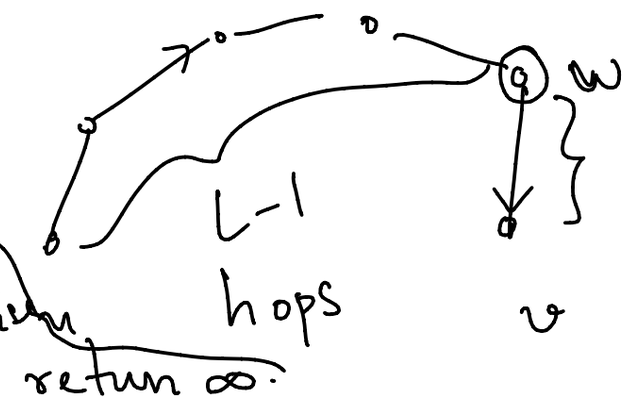


[edge lengths are arbitrary.]

Naive soln:

try every L-hop -path.

Recursive procedure?



ShortestPath(u, v, L):

base-case: if $L = 0$ and $v = u$, return 0; otherwise return ∞.

for each w s.t. (w, v) is an edge:

compute ShortestPath$(u, w, L-1) := x$

look-up.

let dist $= x + l_{wv}$

pick w for which "dist" is smallest & return that value.

Obsns:

→ This procedure is correct, as we "consider" every path

→ This procedure ~~has the~~ takes time $\propto$ # paths.

Sub-problems

Every sub-problem looks like $(u, v', L') \Rightarrow$

parameters v' & L' <u>determine</u> what the sub-problem is!

sub-problems $\leq$? $\boxed{(L+1) \cdot n}$

<u>Qn:</u> Can we store the answers?

Algorithm, run time

replace "compute" with "compute / look-up".

running time $\equiv$

Proof of correctness

Common features

- Sequential decision making
- Some resource “depleting” (steps remaining / #remaining)
- **<u>Key:</u>** past decisions lead to some “state”; we can then solve sub-problem starting at the state (ignoring past)

Examples next class

- Cake-eating problem
- Traveling salesman problem