AlgoLab

Dynamic Programming and Brute Force Tricks

Dynamic Programming

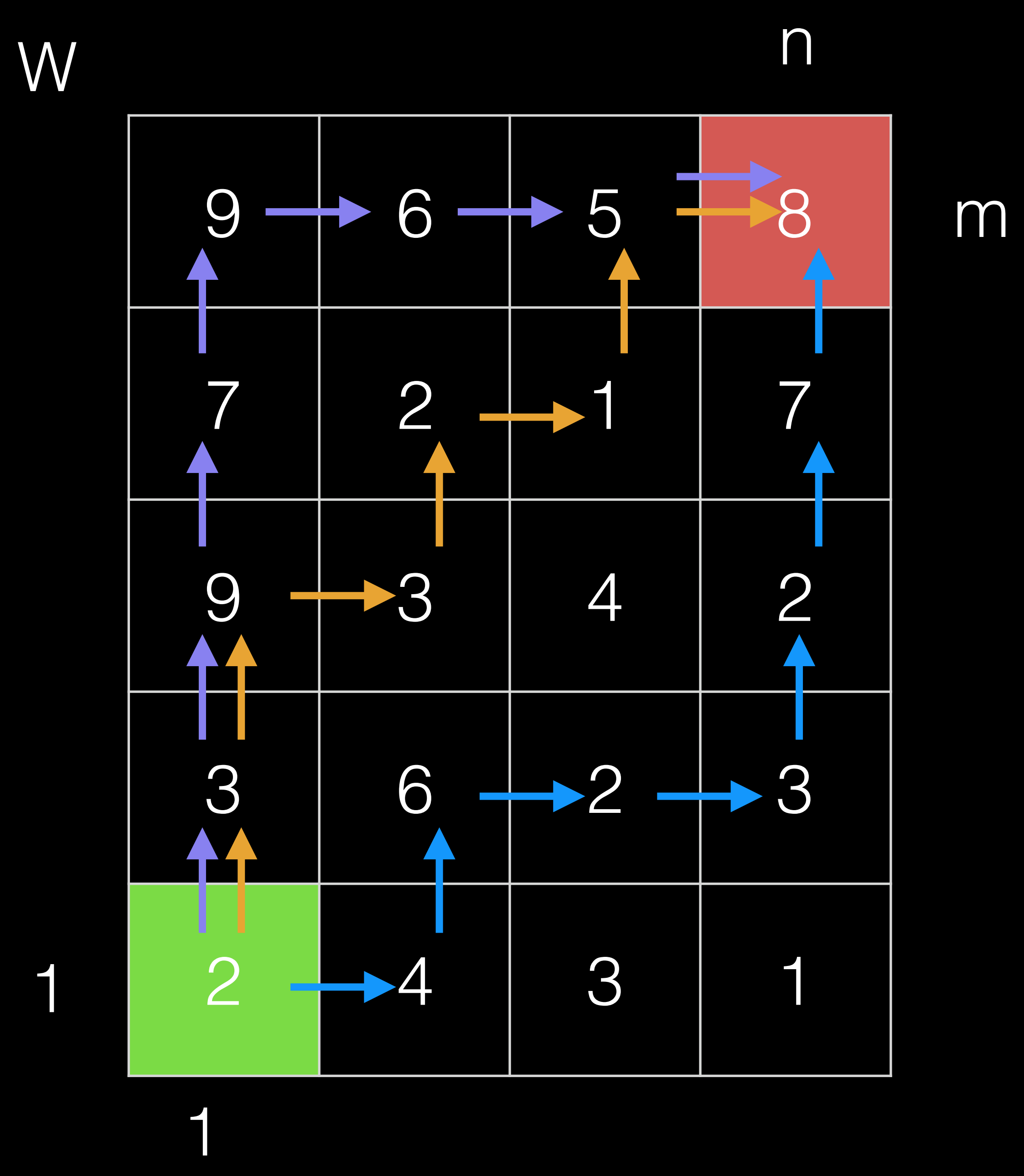
Problem

Find weight of "heaviest" monotone (1,1)-(m,n) path in W

2+3+9+3+2+1+5+8=33

2+4+6+1+3+2+7+8=34

2+3+9+7+9+6+5+8=49



Construct solution recursively

At (1,1) two options:

9	6	5	8
7	2	1	7
9	3	4	2
3	6	2	3
2	4	3	1

Construct solution recursively

At (1,1) two options: go up to (2,1)

9	6	5	8
7	2	1	7
9	3	4	2
3	6	2	3
2	4	3	1

Construct solution recursively

At (1,1) two options: go up to (2,1) go right to (1,2)

9	6	5	8
7	2	1	7
9	3	4	2
3	6	2	3
2	4	3	1

Construct solution recursively

At (1,1) two options: go up to (2,1) go right to (1,2)

Given the weight of the heaviest (2,1)-(m,n) path (1,2)-(m,n) path

9	6	5	8
7	2	1	7
9	3	4	2
3	6	2	3
2	4	3	1

Construct solution recursively

At (1,1) two options: go up to (2,1) go right to (1,2)

Given the weight of the heaviest (2,1)-(m,n) path (1,2)-(m,n) path

The weight of the heaviest (1,1)-(m,n) path is 2 + the maximum of the two

We divided into smaller subproblems

Subproblem is characterised by (i,j)

$f(i,j) :=$ “weight of heaviest (i,j)-(m,n) path”

$f(i,j) = W[i][j] + \max\{f(i+1,j), f(i,j+1)\}$

$f(m,n) = W[m][n]$

9	6	5	8
7	2	1	7
9	3	4	2
3	6	2	3
2	4	3	1

Translate to code

9	6	5	8
7	2	1	7
9	3	4	2
3	6	2	3
2	4	3	1

```
int f(int i, int j){
   int result;
   if(i == m and j == n) result = W[i][j];
   else if(i == m) result = W[i][j] + f(i,j+1);
   else if(j == n) result = W[i][j] + f(i+1,j);
   else result = W[i][j] + max(f(i+1,j),f(i,j+1));
   return result;
}
```

Runtime: try all paths

Can we do better?

A path is a sequence of m ups and n rights

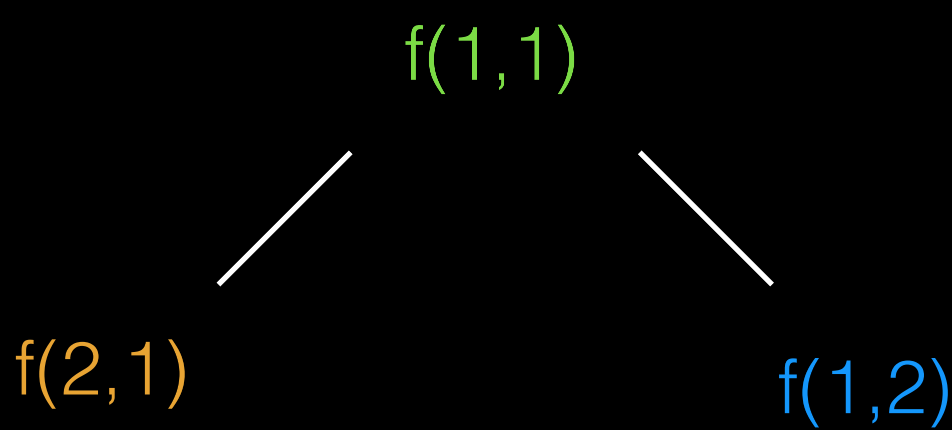
Choose m positions for ups among m+n possible positions

Yes! Because overlapping subproblems

$f(1,1)$

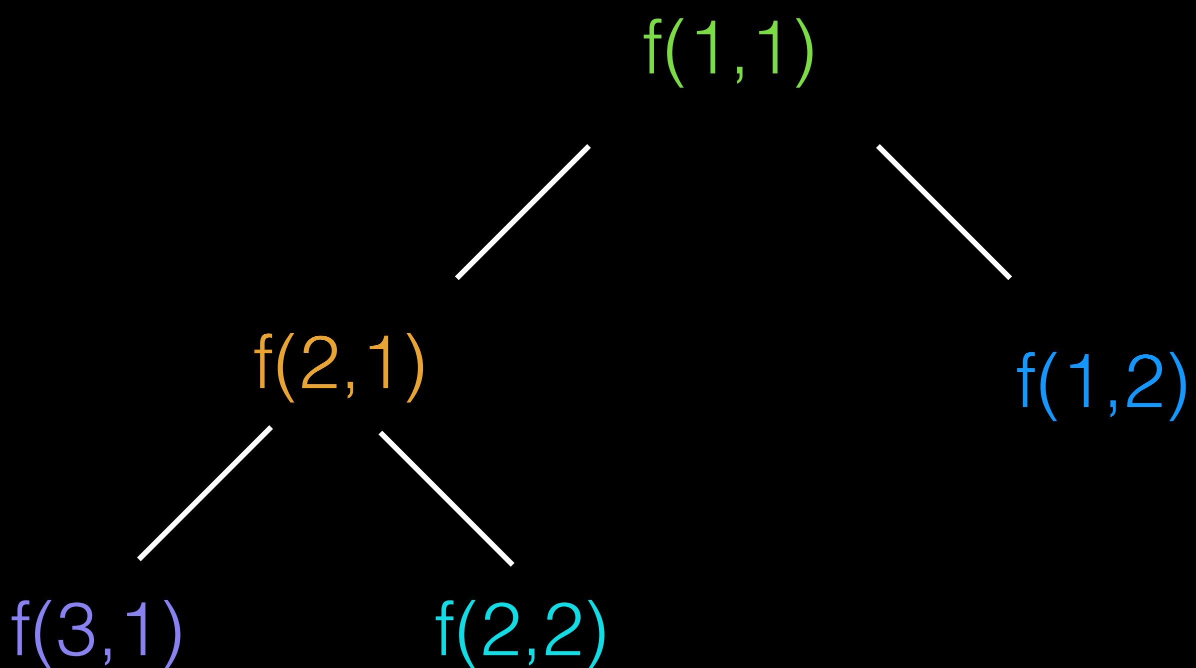
9	6	5	8
7	2	1	7
9	3	4	2
3	6	2	3
2	4	3	1

Yes! Because overlapping subproblems



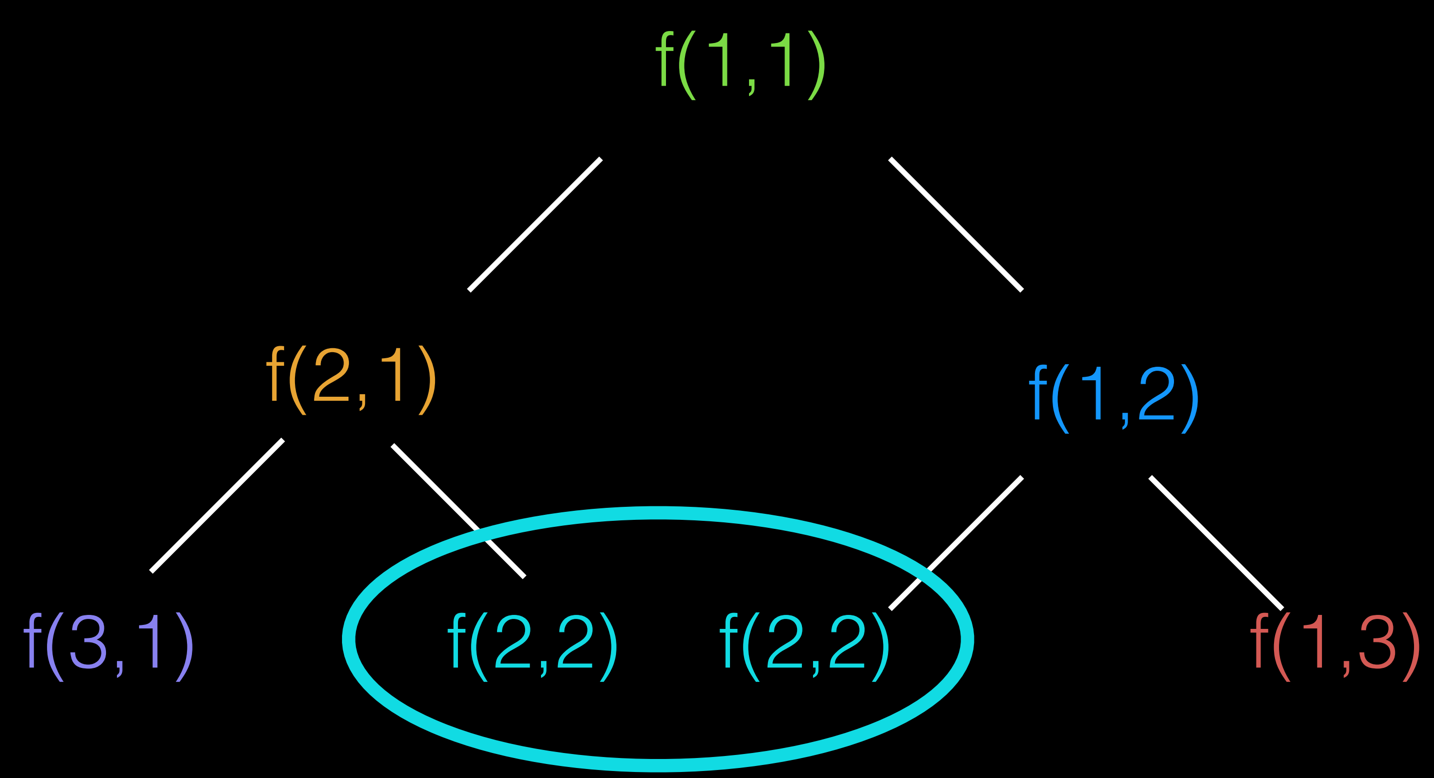
9	6	5	8
7	2	1	7
9	3	4	2
3	6	2	3
2	4	3	1

Yes! Because overlapping subproblems



9	6	5	8
7	2	1	7
9	3	4	2
3	6	2	3
2	4	3	1

Yes! Because overlapping subproblems



9	6	5	8
7	2	1	7
9	3	4	2
3	6	2	3
2	4	3	1

Idea of Dynamic Programming:

Solve subproblems only once, by storing solutions

Recursion + Memo

```
vector<vector<int> > memo(m,vector<int>(n,-1));
int f(int i, int j){
  int result;
  if(memo[i][j] == -1){
    if(i == m and j == n) result = W[i][j];
    else if(i == m) result = W[i][j] + f(i,j+1);
    else if(j == n) result = W[i][j] + f(i+1,j);
    else result = W[i][j] + max(f(i+1,j),f(i,j+1));
    memo[i][j] = result;
  }else{
    result = memo[i][j];
  }
  return result;
}
```

Recursion + Memo

```
vector<vector<int> > memo(m,vector<int>(n,-1));
int f(int i, int j){
   int result;
   if(memo[i][j] == -1){
      if(i == m and j == n) result = W[i][j];
      else if(i == m) result = W[i][j] + f(i,j+1);
      else if(j == n) result = W[i][j] + f(i+1,j);
      else result = W[i][j] + max(f(i+1,j),f(i,j+1));
      memo[i][j] = result;
   }else{
      result = memo[i][j];
   }
   return result;
}
```

Runtime:

Recursion + Memo

```
vector<vector<int> > memo(m,vector<int>(n,-1));
int f(int i, int j){
  int result;
  if(memo[i][j] == -1){
    if(i == m and j == n) result = W[i][j];
    else if(i == m) result = W[i][j] + f(i,j+1);
    else if(j == n) result = W[i][j] + f(i+1,j);
    else result = W[i][j] + max(f(i+1,j),f(i,j+1));
    memo[i][j] = result;
  }else{
    result = memo[i][j];
  }
  return result;
}
```

Runtime: Write in cell at most once: #writes $\leq$ #cells

Recursion + Memo

```
vector<vector<int> > memo(m,vector<int>(n,-1));
int f(int i, int j){
  int result;
  if(memo[i][j] == -1){
    if(i == m and j == n) result = W[i][j];
    else if(i == m) result = W[i][j] + f(i,j+1);
    else if(j == n) result = W[i][j] + f(i+1,j);
    else result = W[i][j] + max(f(i+1,j),f(i,j+1));
    memo[i][j] = result;
  }else{
    result = memo[i][j];
  }
  return result;
}
```

Runtime:

Write in cell at most once: #writes $\leq$ #cells = $m \cdot n$

If recursive call, we write: #calls $\leq$ $2 \cdot$#writes

$O(m \cdot n)$

Construct table explicitly

Take recurrence relation to fill in the table T

$f(i,j)$:= “weight of heaviest (i,j)-(m,n) path” =: $T[i][j]$

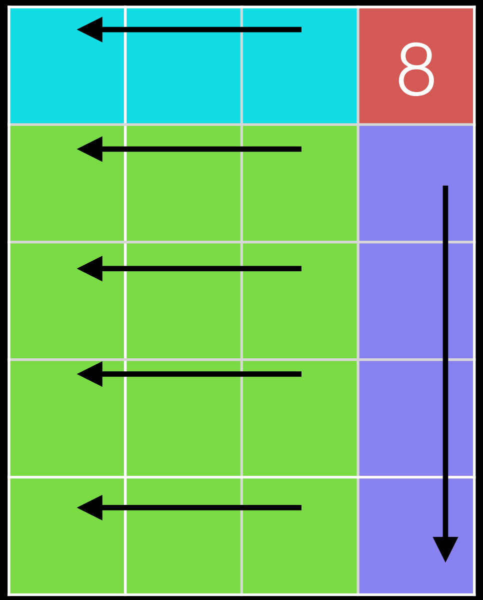
$f(m,n) = W[m][n] = T[m][n]$

$f(i,j) = W[i][j] + \max\{f(i+1,j), f(i,j+1)\}$ $\quad$ $T[i][j] = W[i][j] + \max\{T[i+1][j], T[i][j+1]\}$

When you fill the table, the solutions to the subproblems must be known!

Order in which you fill table is reversed

Translate to code

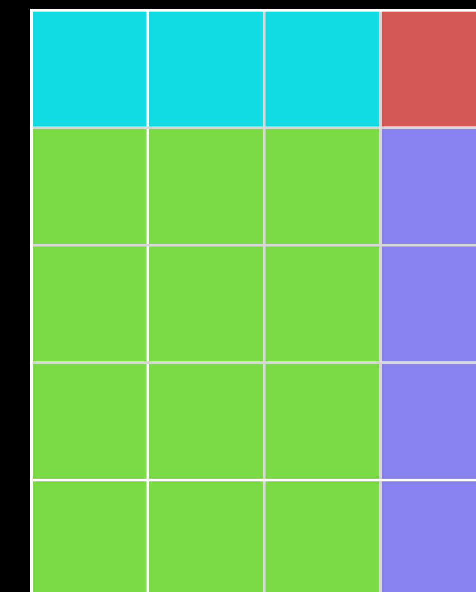



```
vector<vector<int> > T(m,vector<int>(n));
T[m][n] = W[m][n];
for(int i = m-1; i > 0; --i)
  T[i][n] = W[i][n] + T[i+1][n];
for(int j = n-1; j > 0; --j)
  T[m][j] = W[m][j] + T[m][j+1];
for(int i = m-2; i > 0; --i){
  for(int j = n-2; j > 0; --j){
    T[i][j] = M[i][j] + max(T[i][j+1],T[i+1][j]);
  }
}
```

Runtime: nested loops from 1 to m and 1 to n $O(m \cdot n)$

What if we also want to know a heaviest path?

Do not store the partial path!   Reconstruct whether you went up or right



```
vector<pair<int,int> > path;
int i,j; i = j = 1;
path.push_back(make_pair(i,j));

while(!(i == m && j == n)){
  if(i == m) path.push_back(make_pair(i,++j));
  else if(j == n) path.push_back(make_pair(++i,j));
  else{
    if(T[i][j+1] < T[i+1][j]) path.push_back(make_pair(i,++j));
    else path.push_back(make_pair(++i,j));
  }
}
```

Runtime: length of path $O(m+n)$

You can similarly compute all heaviest paths, or count them…   Exercise

Wrap up

Idea of DP: solve subproblems only once!

Store solutions of subproblems

Start by defining recurrence relation

Implement it. It will be correct but slow

Are there overlapping subproblems?

Add memo (usually this does the trick)

Construct table

Practice finding recurrence relation!

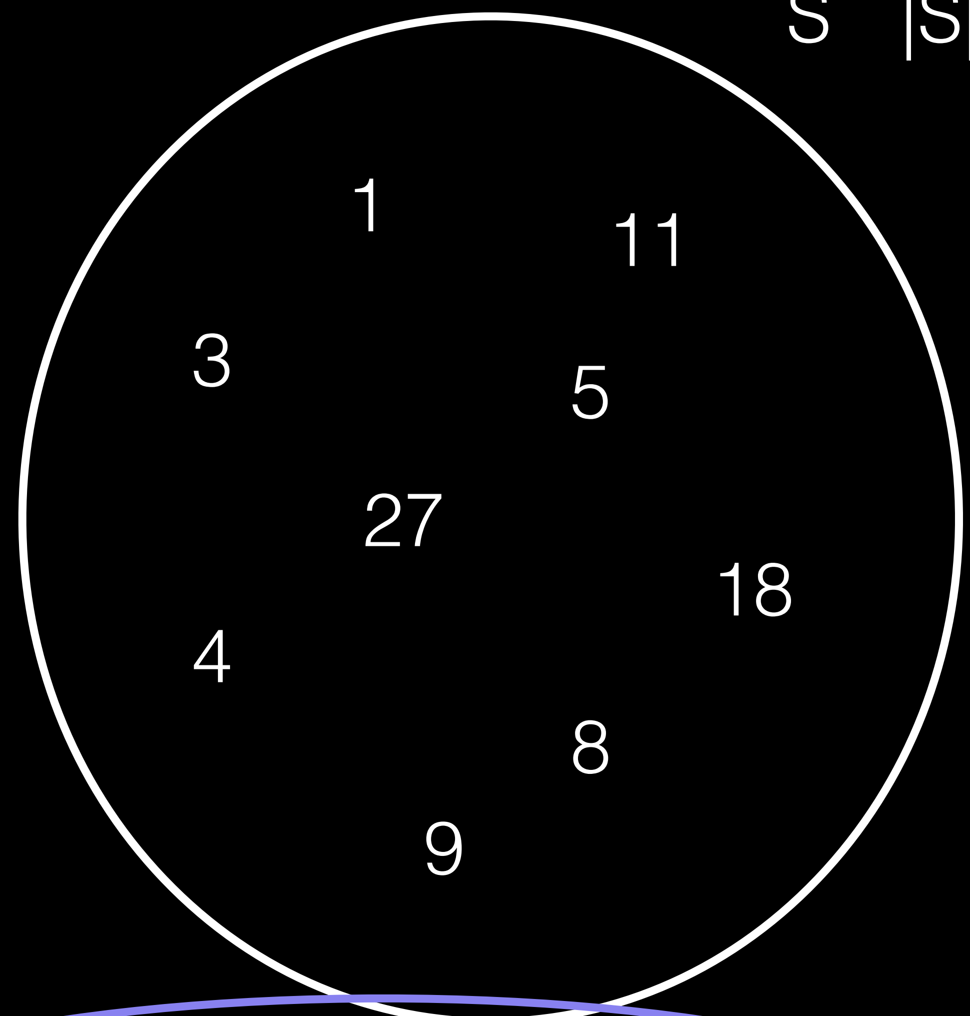
Knapsack, LCS, LIS, Coin Change, Edit Distance…

Brute Force

Problem

Is there a subset of S which sums to k?

S $|S| =: n$



k = 8? Yes! 1+3+4 or 8

k = 1000? No!

k = 37? Yes! 18+9+4+5+1

NP complete problem :(

n is small: brute force

We try this

k is small: dynamic programming

Exercise

Recursively construct all subsets (backtracking)

In the beginning two options: take first element into subset or not take

Given the solution for (S\S[1],k-S[1]) and (S\S[1],k)

The solution of (S,k) is just the "or" of the two

f(i,j) := "is there a subset of S\{S[1], …, S[i]} which sums to j"

f(i,j) = f(i+1,j-S[i]) or f(i+1,j)

f(i,0) = yes, for all i   f(i,j) = no, for all i and j < 0   f(n,j) = no, for all j > 0

```
bool f(i,j){
  if(j == 0) return true;
  if(i == n || j < 0) return false;
  return f(i+1,j-S[i]) || f(i+1,j);
}
```

Runtime: try all subsets $O(2^n)$ ok for n up to say 25

Sets and Bits

Represent sets by characteristic vector

$\Omega=\{e_1,\ldots,e_n\}$ $\quad S \subseteq \Omega$ $\quad s \in \{0,1\}^n$ $\quad s_i = 1$ iff $e_i \in S$

"Encode" characteristic vector as integer

s is integer $\quad$ i-th bit of s is 1 iff $e_i \in S$

Bit-wise operations on integers in C++

and

a = 1100
b = 1010
a & b = 1000

or

a = 1100
b = 1010
a | b = 1110

xor

a = 1100
b = 1010
a ^ b = 0110

not

a = 1100
~a = 0011

bit-shift $\quad a << b = a \cdot 2^b$ $\quad a >> b = a / 2^b$

set i-th bit: 1<<i

Sets and Bits

Represent sets by characteristic vector

$\Omega=\{e_1,\ldots,e_n\}$ $S \subseteq \Omega$ $s \in \{0,1\}^n$ $s_i = 1$ iff $e_i \in S$

"Encode" characteristic vector as integer

s is integer i-th bit of s is 1 iff $e_i \in S$

Set operations:

union: a | b intersection: a & b

subtraction: a & ~b negation: 1…1 ^ a

add i-th element: a |= 1 << i remove i-th element: a &= ~(1 << i)

check i-th element: (a & 1<< i) != 0

Solve subset sum: iterate over all subsets

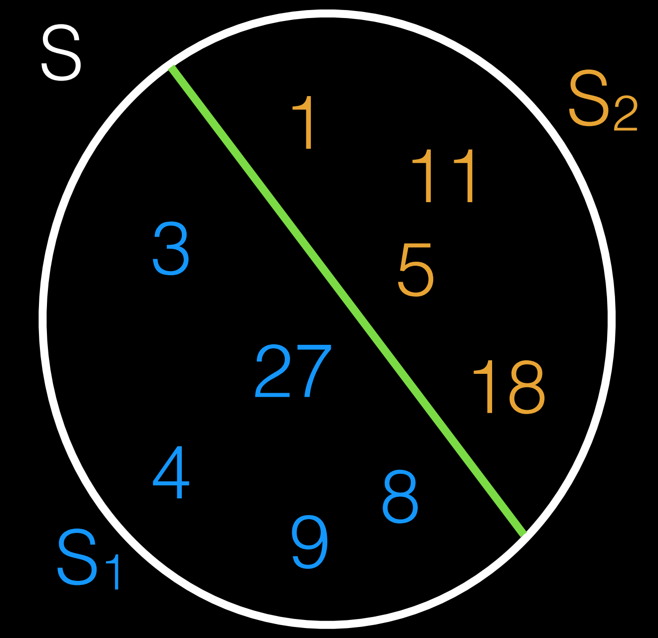
```
for(int s = 0; s < 1<<n; ++s){
   int sum = 0;
   for(int i = 0; i < n; ++i){
      if(s & 1<<i) sum += S[i];
   }
   if(sum == k) return true;
}
```

maybe stop if sum > k

Runtime: nested loops from 0 to $2^n - 1$ and 0 to $n-1$

$O(n2^n)$

Nice trick: Split and list



Split S into disjoint S_1 and S_2 of size n/2

Observation: There is a subset of S summing to k

iff

There are subsets of S_1 and S_2 summing to k_1 and k_2 with $k = k_1 + k_2$

List all subset sums of S_1 and S_2 in L_1 and L_2 — Runtime $O(2^{n/2+1})$

Sort L_2 — Runtime $O(n \cdot 2^{n/2})$

Go through L_1 — Runtime $O(2^{n/2} \cdot 2^{n/2}) = O(2^n)$

Check for each k_1 whether there is $k_2 := k - k_1$ in L_2 with binary search!

Runtime $O(n \cdot 2^{n/2})$

ok for n up to say 40 :)