INFORMATION & COMPUTER SCIENCES
UNIVERSITY *of* HAWAIʻI *at* MĀNOA

AlgoPARC

ICS 621: Analysis of Algorithms

Prof. Nodari Sitchinava

Dynamic Programming

Recursive Backtracking,
Pruned with a Lookup (Memo) Table

Recursive Backtracking,
Pruned with a Lookup (Memo) Table

(Faster if there are overlapping subproblems)

Example: Subset Sum

Problem (Subset Sum)**.** *Given a set* S *of* $1 \leq n \leq 20$ *integers and a positive integer* x*, is there a subset of* S *that sums to* x*?*

$$S = \{17, 5, 7, 15, 3, 8\} \qquad x = 16$$

Example: Subset Sum

Problem (Subset Sum). *Given a set S of $1 \leq n \leq 20$ integers and a positive integer x, is there a subset of S that sums to x?*

$$S = \{17, 5, 7, 15, 3, 8\} \qquad x = 16$$

Solution: Generate all possible subsets, add up the elements of each subset and check if the sum equals x

Example: Subset Sum

Problem (Subset Sum)**.** *Given a set S of $1 \leq n \leq 20$ integers and a positive integer x, is there a subset of S that sums to x?*

$$S = \{17, 5, 7, 15, 3, 8\} \qquad x = 16$$

$$0 \quad 1 \quad 0 \quad 0 \quad 1 \quad 1$$

Solution: Generate all possible subsets, add up the elements of each subset and check if the sum equals x

Example: Subset Sum

Problem (Subset Sum). *Given a set S of* $1 \leq n \leq 20$ *integers and a positive integer x, is there a subset of S that sums to x?*

$$S = \{17, 5, 7, 15, 3, 8\} \quad x = 16$$

0 1 0 0 1 1

Solution: Generate all possible subsets, add up the elements of each subset and check if the sum equals x

```
for (i = 0; i < 2^n; i++) {        // i-th subset out of 2^n
  sum = 0;
  for (int j = 0; j < n; j++)
    if (i & (1 << j))              // is j-th bit set in i?
      sum += S[j]                  // j is part of the subset
  if (sum == x) return true;       // or could return i
}
return false;
```

Example: Subset Sum

Problem (Subset Sum)**.** *Given a set S of* $1 \leq n \leq 20$ *integers and a positive integer x, is there a subset of S that sums to x?*

$$S = \{17, 5, 7, 15, 3, 8\} \quad x = 16$$

0 1 0 0 1 1 ← `i = 19`

Solution: Generate all possible subsets, add up the elements of each subset and check if the sum equals *x*

```
for (i = 0; i < 2^n; i++) {        // i-th subset out of 2^n
  sum = 0;
  for (int j = 0; j < n; j++)
    if (i & (1 << j))              // is j-th bit set in i?
      sum += S[j]                  // j is part of the subset
  if (sum == x) return true;       // or could return i
}
return false;
```

Subset Sum: Recursive Solution

Subset Sum: Recursive Solution

Top-down Solution: Prune search space as you generate partial solutions

$$S = \{17, 5, 7, 15, 3, 8\} \qquad x = 16$$

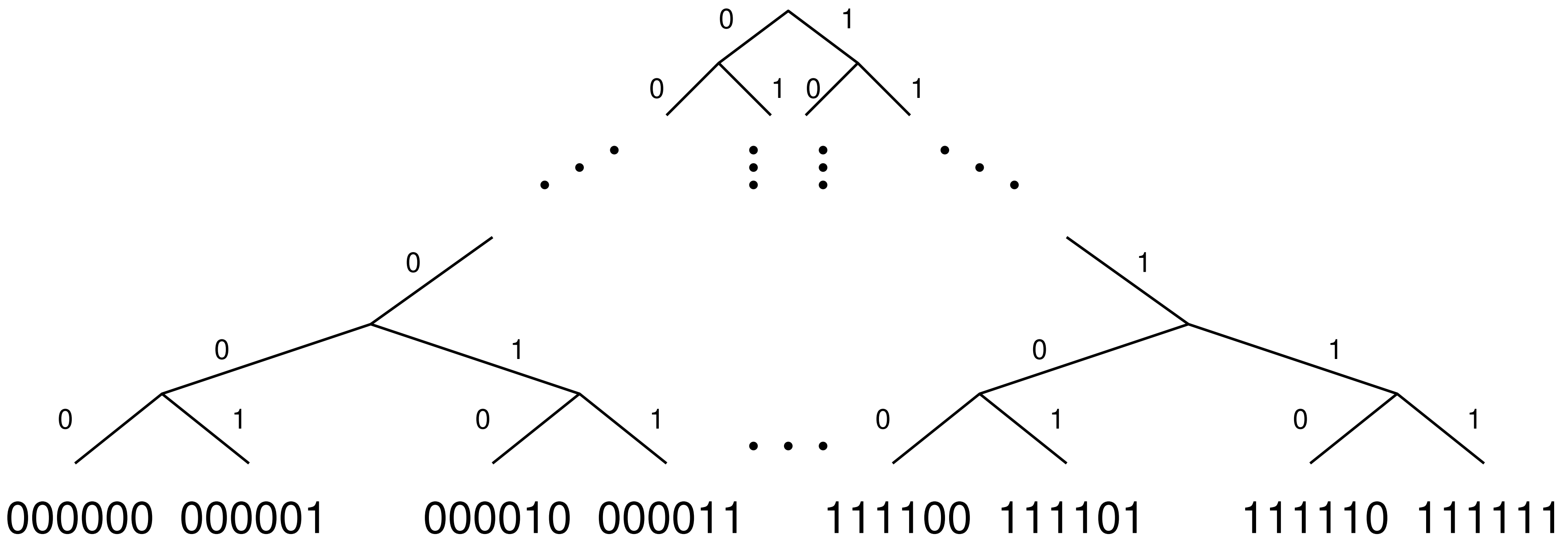
0 1 0 0 1 1

Subset Sum: Recursive Solution

Top-down Solution: Prune search space as you generate partial solutions

$$S = \{17, 5, 7, 15, 3, 8\} \qquad x = 16$$

0 1 0 0 1 1



Subset Sum: Recursive Solution

Subset Sum: Recursive Solution

```
int S[20] = {...}; int x = ...;      // initialize S & x

// returns true iff exists subset within S[i:n]
// that adds up to x
SubsetSum(S[i:n], x):



  else
    bool Si_notSelected = SubsetSum(S[i+1:n], x);
    bool Si_selected = SubsetSum(S[i+1:n], x-S[i]);
    return (Si_notSelected or Si_selected);
```

Subset Sum: Recursive Solution

```
int S[20] = {...}; int x = ...;     // initialize S & x

// returns true iff exists subset within S[i:n]
// that adds up to x
SubsetSum(S[i:n], x):
  if (x < 0 or i > n)  return false;
  else if (x == 0)     return true;
  else
    bool Si_notSelected = SubsetSum(S[i+1:n], x);
    bool Si_selected = SubsetSum(S[i+1:n], x-S[i]);
    return (Si_notSelected or Si_selected);
```

Subset Sum: Recursive Solution

```
int S[20] = {...}; int x = ...;     // initialize S & x

// returns true iff exists subset within S[i:n]
// that adds up to x
SubsetSum(S[i:n], x):
  if (x < 0 or i > n)  return false;
  else if (x == 0)     return true;
  else
    bool Si_notSelected = SubsetSum(S[i+1:n], x);
    bool Si_selected = SubsetSum(S[i+1:n], x-S[i]);
    return (Si_notSelected or Si_selected);

main() {
  return SubsetSum(S[1:n], x);
}
```

Subset Sum: Recursive Solution

```
int S[20] = {...}; int x = ...;     // initialize S & x

// returns true iff exists subset within S[i:n]
// that adds up to x
SubsetSum(S[i:n], x):
  if (x < 0 or i > n)  return false;
  else if (x == 0)     return true;
  else
    return SubsetSum(S[i+1:n], x) or   // S[i] not selected
           SubsetSum(S[i+1:n], x-S[i]); // S[i] selected


main() {
  return SubsetSum(S[1:n], x);
}
```

Subset Sum: Recursive Solution

```
int S[20] = {...}; int x = ...;     // initialize S & x

// returns true iff exists subset within S[i:n]
// that adds up to x
SubsetSum(i, x):
  if (x < 0 or i > n)  return false;
  else if (x == 0)     return true;
  else
    return SubsetSum(i+1, x) or          // S[i] not selected
           SubsetSum(i+1, x-S[i]);        // S[i] selected


main() {
  return SubsetSum(1, x);
}
```

Subset Sum: Recursive Solution

```
int S[20] = {...}; int x = ...;     // initialize S & x

// returns true iff exists subset within S[i:n]
// that adds up to x
SubsetSum(i, x):
  if (x < 0 or i > n)  return false;
  else if (x == 0)     return true;
  else
    return SubsetSum(i+1, x) or          // S[i] not selected
           SubsetSum(i+1, x-S[i]);        // S[i] selected


main() {
  return SubsetSum(1, x);
}
```

SS[i][x]

SS[i+1][x] or
SS[i+1][x-S[i]]

Subset Sum: Dynamic Programming

```
int S[20] = {...}; int x = ...;     // initialize S & x

// returns true iff exists subset within S[i:n]
// that adds up to x
SubsetSum(i, x):
  if (x < 0 or i > n)  return SS[i][x] = false;
  else if (x == 0)     return SS[i][x] = true;
  else if (SS[i][x] != UNDEFINED) return SS[i][x];
  else
    return SS[i][x] = SubsetSum(i+1, x) or
           SubsetSum(i+1, x-S[i]);

main() {
  return SubsetSum(1, x);
}
```

Subset Sum: Dynamic Programming

```
int S[20] = {...}; int x = ...;     // initialize S & x
int SS[20][MAX_X]; memset(SS, UNDEFINED, sizeof(SS));
// returns true iff exists subset within S[i:n]
// that adds up to x
SubsetSum(i, x):
  if (x < 0 or i > n)  return SS[i][x] = false;
  else if (x == 0)     return SS[i][x] = true;
  else if (SS[i][x] != UNDEFINED) return SS[i][x];
  else
    return SS[i][x] = SubsetSum(i+1, x) or
           SubsetSum(i+1, x-S[i]);

main() {
  return SubsetSum(1, x);
}
```

0-1 Knapsack

Problem (0-1 Knapsack). *Given a set S of n items, each with its own value V_i and weight W_i for all $1 \leq i \leq n$ and a maximum knapsack capacity C, compute the maximum value of the items that you can carry. You cannot take fractions of items.*

0-1 Knapsack

Problem (0-1 Knapsack). *Given a set S of n items, each with its own value V_i and weight W_i for all $1 \leq i \leq n$ and a maximum knapsack capacity C, compute the maximum value of the items that you can carry. You cannot take fractions of items.*

- Optimization version of Subset Sum

0-1 Knapsack

Problem (0-1 Knapsack)**.** *Given a set S of n items, each with its own value V_i and weight W_i for all $1 \leq i \leq n$ and a maximum knapsack capacity C, compute the maximum value of the items that you can carry. You cannot take fractions of items.*

- Optimization version of Subset Sum

Example:

$\{(V_i, W_i)\} = \{(10, 17), (5, 7), (3, 8), (9, 15)\} \quad C = 16$

0-1 Knapsack: Recursive Solution

$$\{(V_i, W_i)\} = \{(10, 17), (5, 7), (3, 8), (9, 15)\} \qquad C = 16$$

0-1 Knapsack: Recursive Solution

$\{(V_i, W_i)\} = \{(10, 17), (5, 7), (3, 8), (9, 15)\} \quad C = 16$

$maxV(i, C)$ returns the maximum value among items $S[i : n]$ with remaining knapsack capacity of C

0-1 Knapsack: Recursive Solution

$\{(V_i, W_i)\} = \{(10, 17), (5, 7), (3, 8), (9, 15)\} \qquad C = 16$

$maxV(i, C)$ returns the maximum value among items $S[i : n]$ with remaining knapsack capacity of C

$$maxV(i, C) = \begin{cases} 0 & \text{if } i > n \\ 0 & \text{if } C \leq 0 \\ maxV(i+1, C) & \text{if } W_i > C \\ \max \left\{ \begin{array}{l} maxV(i+1, C) \\ V_i + maxV(i+1, C - W_i) \end{array} \right\} & \text{if } W_i \leq C \end{cases}$$

0-1 Knapsack: Recursive Solution

$$
maxV(i, C) = \begin{cases} 0 & \text{if } i > n \\ 0 & \text{if } C \leq 0 \\ maxV(i+1, C) & \text{if } W_i > C \\ \max \left\{ \begin{array}{l} maxV(i+1, C) \\ V_i + maxV(i+1, C - W_i) \end{array} \right\} & \text{if } W_i \leq C \end{cases}
$$

0-1 Knapsack: Recursive Solution

$$maxV(i, C) = \begin{cases} 0 & \text{if } i > n \\ 0 & \text{if } C \leq 0 \\ maxV(i+1, C) & \text{if } W_i > C \\ \max \begin{Bmatrix} maxV(i+1, C) \\ V_i + maxV(i+1, C - W_i) \end{Bmatrix} & \text{if } W_i \leq C \end{cases}$$

```
maxV(i,C) {
  if (i > n || C <= 0) return 0;

  if (W[i] > C)                        //can't take i-th item
    return maxV(i+1, C);
  return max(maxV(i+1, C),             //don't take i-th item
             V[i]+maxV(i+1, C-W[i]));     // take i-th item
}
```

0-1 Knapsack: Recursive Solution

$$maxV(i, C) = \begin{cases} 0 & \text{if } i > n \\ 0 & \text{if } C \leq 0 \\ maxV(i+1, C) & \text{if } W_i > C \\ \max \begin{Bmatrix} maxV(i+1, C) \\ V_i + maxV(i+1, C - W_i) \end{Bmatrix} & \text{if } W_i \leq C \end{cases}$$

```
maxV(i,C) {
  if (i > n || C <= 0) return 0;

  if (W[i] > C)                          //can't take i-th item
    return maxV(i+1, C);
  return max(maxV(i+1, C),               //don't take i-th item
             V[i]+maxV(i+1, C-W[i]));     // take i-th item
}
main() {
  return maxV(1, C);
}
```

0-1 Knapsack: DP solution

$$maxV(i, C) = \begin{cases} 0 & \text{if } i > n \\ 0 & \text{if } C \leq 0 \\ maxV(i+1, C) & \text{if } W_i > C \\ \max \begin{Bmatrix} maxV(i+1, C) \\ V_i + maxV(i+1, C - W_i) \end{Bmatrix} & \text{if } W_i \leq C \end{cases}$$

M[i][C]

```
maxV(i,C) {
  if (i > n || C <= 0) return 0;

  if (W[i] > C)                          //can't take i-th item
    return maxV(i+1, C);
  return max(maxV(i+1, C),               //don't take i-th item
             V[i]+maxV(i+1, C-W[i]));     // take i-th item
}
main() {
  return maxV(1, C);
}
```

M[i+1][C]

M[i+1][C]

M[i+1][C-W[i]]

0-1 Knapsack: DP solution

$$maxV(i, C) = \begin{cases} 0 & \text{if } i > n \\ 0 & \text{if } C \leq 0 \\ maxV(i+1, C) & \text{if } W_i > C \\ \max \left\{ \begin{array}{l} maxV(i+1, C) \\ V_i + maxV(i+1, C - W_i) \end{array} \right\} & \text{if } W_i \leq C \end{cases}$$

```
maxV(i,C) {
  if (i > n || C <= 0) return 0;

  if (W[i] > C)                          //can't take i-th item
    return maxV(i+1, C);
  return max(maxV(i+1, C),               //don't take i-th item
             V[i]+maxV(i+1, C-W[i]));     // take i-th item
}
main() {
  return maxV(1, C);
}
```

0-1 Knapsack: DP solution

$$maxV(i, C) = \begin{cases} 0 & \text{if } i > n \\ 0 & \text{if } C \leq 0 \\ maxV(i+1, C) & \text{if } W_i > C \\ \max \begin{Bmatrix} maxV(i+1, C) \\ V_i + maxV(i+1, C - W_i) \end{Bmatrix} & \text{if } W_i \leq C \end{cases}$$

```
maxV(i,C) {
  if (i > n || C <= 0) return 0;
  if (M[i][C] != UNDEFINED) return M[i, C];
  if (W[i] > C)                              //can't take i-th item
    return M[i][C]=maxV(i+1, C);
  return M[i][C]=max(maxV(i+1, C),           //don't take i-th item
                V[i]+maxV(i+1, C-W[i]));         // take i-th item
}
main() {
  return maxV(1, C);
}
```

0-1 Knapsack: DP solution

$$maxV(i, C) = \begin{cases} 0 & \text{if } i > n \\ 0 & \text{if } C \leq 0 \\ maxV(i+1, C) & \text{if } W_i > C \\ \max \left\{ \begin{array}{l} maxV(i+1, C) \\ V_i + maxV(i+1, C - W_i) \end{array} \right\} & \text{if } W_i \leq C \end{cases}$$

```
int M[maxN+1][maxC];
maxV(i,C) {
  if (i > n || C <= 0) return 0;
  if (M[i][C] != UNDEFINED) return M[i, C];
  if (W[i] > C)                          //can't take i-th item
    return M[i][C]=maxV(i+1, C);
  return M[i][C]=max(maxV(i+1, C),       //don't take i-th item
              V[i]+maxV(i+1, C-W[i]));       // take i-th item
}
main() { memset(M, UNDEFINED, sizeof(M));
  return maxV(1, C);
}
```

Longest Increasing Subsequence (LIS)

Problem. *Given a sequence $A[1...n]$, determine the length of its longest subsequence (not necessarily contiguous) that is in **increasing** order.*

Longest Increasing Subsequence (LIS)

Problem. *Given a sequence $A[1...n]$, determine the length of its longest subsequence (not necessarily contiguous) that is in* ***increasing*** *order.*

Example: $A = \{-7, 10, 9, 2, 3, 8, 8, 1\}$.
Solution: LIS = 4: $\{-7, 2, 3, 8\}$

Longest Increasing Subsequence (LIS)

Problem. *Given a sequence $A[1...n]$, determine the length of its longest subsequence (not necessarily contiguous) that is in* ***increasing*** *order.*

Example: $A = \{-7, 10, 9, 2, 3, 8, 8, 1\}$.
Solution: LIS = 4: $\{-7, 2, 3, 8\}$

Hint 1: LIS(i) returns the size of the longest increasing subsequence that ***terminates*** on $A[i]$.

Longest Increasing Subsequence (LIS)

Problem. *Given a sequence $A[1...n]$, determine the length of its longest subsequence (not necessarily contiguous) that is in **increasing** order.*

Example: $A = \{-7, 10, 9, 2, 3, 8, 8, 1\}$.
Solution: LIS = 4: $\{-7, 2, 3, 8\}$

Hint 1: LIS(i) returns the size of the longest increasing subsequence that ***terminates*** on $A[i]$.

Hint 2: When deciding whether to add $A[i]$, find the longest subsequence in $A[1 : i - 1]$ that allows adding $A[i]$.

LIS: Recursive Solution

LIS: Recursive Solution

$A = \{-7, 10, 9, 2, 3, 8, 8, 1\}$

$$LIS(i) = \begin{cases} 1 & \text{if } i = 1 \\ \max \left\{ \begin{array}{ll} 1 & \\ LIS(1) + 1 & \text{if } A[i] > A[1] \\ LIS(2) + 1 & \text{if } A[i] > A[2] \\ LIS(3) + 1 & \text{if } A[i] > A[3] \\ \ldots & \ldots \\ LIS(i-1) + 1 & \text{if } A[i] > A[i-1] \end{array} \right\} & \text{if } i > 1 \end{cases}$$

LIS: Recursive Solution

$A = \{-7, 10, 9, 2, 3, 8, 8, 1\}$

$$LIS(i) = \begin{cases} 1 & \text{if } i = 1 \\ \max \left\{ \begin{array}{ll} 1 & \\ LIS(1) + 1 & \text{if } A[i] > A[1] \\ LIS(2) + 1 & \text{if } A[i] > A[2] \\ LIS(3) + 1 & \text{if } A[i] > A[3] \\ \ldots & \ldots \\ LIS(i-1) + 1 & \text{if } A[i] > A[i-1] \end{array} \right\} & \text{if } i > 1 \end{cases}$$

```
LIS(i)

  if (i == 1) return 1;
  best = 1;  // LIS = { A[i] }
  for (int j = 1; j < i; j++)
    if (A[i] > A[j]) {
      current = LIS(j) + 1;
      if (best < current)
        best = current;
    }
  return best;
```

LIS: Recursive Solution

$A = \{-7, 10, 9, 2, 3, 8, 8, 1\}$

$$LIS(i) = \begin{cases} 1 & \text{if } i = 1 \\ \max \left\{ \begin{array}{ll} 1 & \\ LIS(1) + 1 & \text{if } A[i] > A[1] \\ LIS(2) + 1 & \text{if } A[i] > A[2] \\ LIS(3) + 1 & \text{if } A[i] > A[3] \\ \ldots & \ldots \\ LIS(i-1) + 1 & \text{if } A[i] > A[i-1] \end{array} \right\} & \text{if } i > 1 \end{cases}$$

```
LIS(i)

  if (i == 1) return 1;
  best = 1;  // LIS = { A[i] }
  for (int j = 1; j < i; j++)
    if (A[i] > A[j]) {
      current = LIS(j) + 1;
      if (best < current)
        best = current;
    }
  return best;
```

```
main()
  best = 0;
  for (i = 1; i <= n; i++ ){
    current = LIS(i);
    if (best < current)
      best = current;
  }
  return best;
```

LIS: Recursive Solution

$A = \{-7, 10, 9, 2, 3, 8, 8, 1\}$

$$LIS(i) = \begin{cases} 1 & \text{if } i = 1 \\ \max \left\{ \begin{array}{ll} 1 & \\ LIS(1) + 1 & \text{if } A[i] > A[1] \\ LIS(2) + 1 & \text{if } A[i] > A[2] \\ LIS(3) + 1 & \text{if } A[i] > A[3] \\ \ldots & \ldots \\ LIS(i-1) + 1 & \text{if } A[i] > A[i-1] \end{array} \right\} & \text{if } i > 1 \end{cases}$$

```
LIS(i)
  if (L[i]!=UNDEFINED) return L[i];
  if (i == 1) return L[i] = 1;
  best = 1;  // LIS = { A[i] }
  for (int j = 1; j < i; j++)
    if (A[i] > A[j]) {
      current = LIS(j) + 1;
      if (best < current)
        best = current;
    }
  return L[i] = best;
```

```
main()
  best = 0;
  for (i = 1; i <= n; i++ ){
    current = LIS(i);
    if (best < current)
      best = current;
  }
  return best;
```

Bottom-up DP

Bottom-up DP

Iterative DP (no recursion)

Example: Subset Sum

Problem (Subset Sum)**.** *Given a set* S *of* $1 \leq n \leq 20$ *integers and a positive integer* x*, is there a subset of* S *that sums to* x*?*

$$S = \{17, 5, 7, 15, 3, 8\} \qquad x = 16$$

Example: Subset Sum

Problem (Subset Sum)**.** *Given a set S of $1 \leq n \leq 20$ integers and a positive integer x, is there a subset of S that sums to x?*

$$S = \{17, 5, 7, 15, 3, 8\} \qquad x = 16$$

$$0 \quad 1 \quad 0 \quad 0 \quad 1 \quad 1$$

Example: Subset Sum

Problem (Subset Sum). *Given a set* S *of* $1 \leq n \leq 20$ *integers and a positive integer* x*, is there a subset of* S *that sums to* x*?*

$$S = \{17, 5, 7, 15, 3, 8\} \qquad x = 16$$

$$0 \quad 1 \quad 0 \quad 0 \quad 1 \quad 1$$

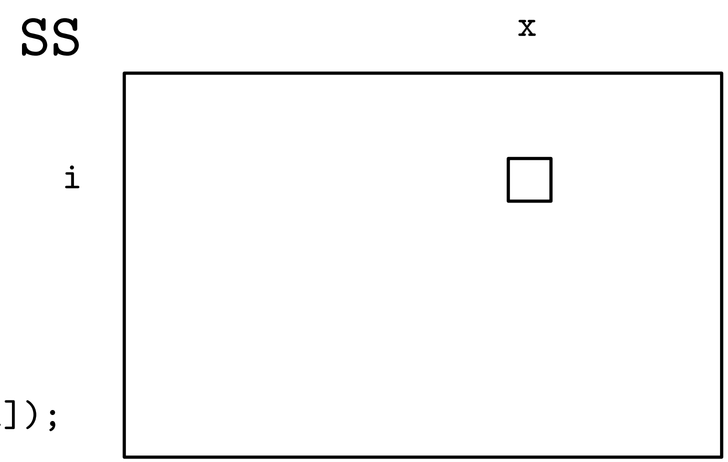
```
int S[20] = {...}; int x = ...;     // initialize S & x
int SS[20][MAX_X]; memset(SS, UNDEFINED, sizeof(SS));
// returns true iff exists subset within S[i:n]
// that adds up to x
SubsetSum(i, x):
  if (x < 0 or i > n)  return S[i][x] = false;
  else if (x == 0)     return S[i][x] = true;
  else if (SS[i][x] != UNDEFINED) return SS[i][x];
  else
    return SS[i][x] = SubsetSum(i+1, x) or SubsetSum(i+1, x-S[i]);

main() {
  return SubsetSum(0, x);
}
```

Example: Subset Sum

```
int S[20] = {...}; int x = ...;     // initialize S & x
int SS[20][MAX_X]; memset(SS, UNDEFINED, sizeof(SS));
// returns true iff exists subset within S[i:n-1]
// that adds up to x
SubsetSum(i, x):
  if (x < 0 or i > n)  return S[i][x] = false;
  else if (x == 0)     return S[i][x] = true;
  else if (SS[i][x] != UNDEFINED) return SS[i][x];
  else
    return SS[i][x] = SubsetSum(i+1, x) or SubsetSum(i+1, x-S[i]);
main() {
  return SubsetSum(0, x);
}
```

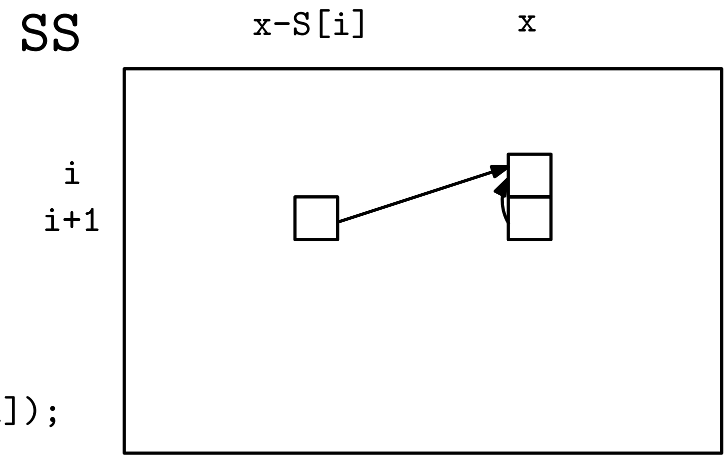
Example: Subset Sum

```
int S[20] = {...}; int x = ...;     // initialize S & x
int SS[20][MAX_X]; memset(SS, UNDEFINED, sizeof(SS));
// returns true iff exists subset within S[i:n-1]
// that adds up to x
SubsetSum(i, x):
  if (x < 0 or i > n)  return S[i][x] = false;
  else if (x == 0)     return S[i][x] = true;
  else if (SS[i][x] != UNDEFINED) return SS[i][x];
  else
    return SS[i][x] = SubsetSum(i+1, x) or SubsetSum(i+1, x-S[i]);
main() {
  return SubsetSum(0, x);
}
```



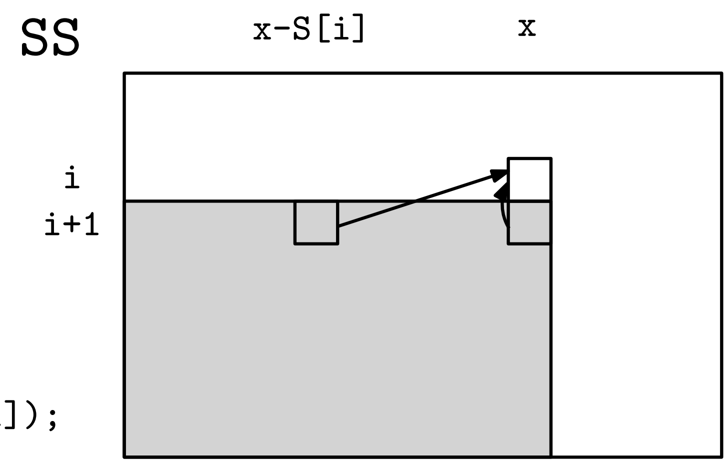
Example: Subset Sum

```
int S[20] = {...}; int x = ...;     // initialize S & x
int SS[20][MAX_X]; memset(SS, UNDEFINED, sizeof(SS));
// returns true iff exists subset within S[i:n-1]
// that adds up to x
SubsetSum(i, x):
  if (x < 0 or i > n)  return S[i][x] = false;
  else if (x == 0)     return S[i][x] = true;
  else if (SS[i][x] != UNDEFINED) return SS[i][x];
  else
    return SS[i][x] = SubsetSum(i+1, x) or SubsetSum(i+1, x-S[i]);
main() {
  return SubsetSum(0, x);
}
```



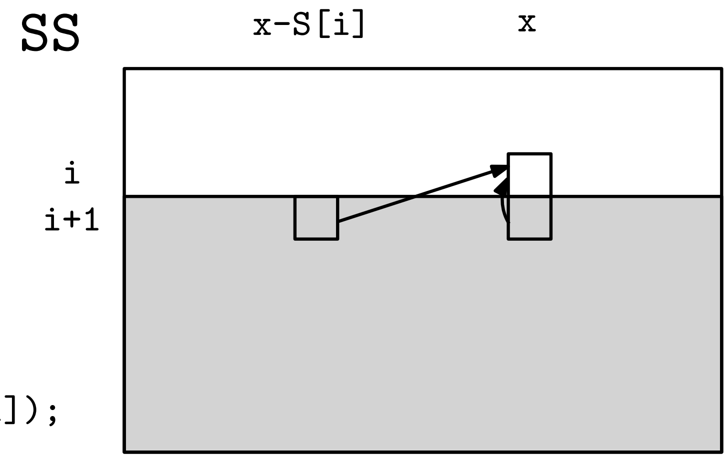
Example: Subset Sum

```
int S[20] = {...}; int x = ...;     // initialize S & x
int SS[20][MAX_X]; memset(SS, UNDEFINED, sizeof(SS));
// returns true iff exists subset within S[i:n-1]
// that adds up to x
SubsetSum(i, x):
  if (x < 0 or i > n)  return S[i][x] = false;
  else if (x == 0)     return S[i][x] = true;
  else if (SS[i][x] != UNDEFINED) return SS[i][x];
  else
    return SS[i][x] = SubsetSum(i+1, x) or SubsetSum(i+1, x-S[i]);
main() {
  return SubsetSum(0, x);
}
```



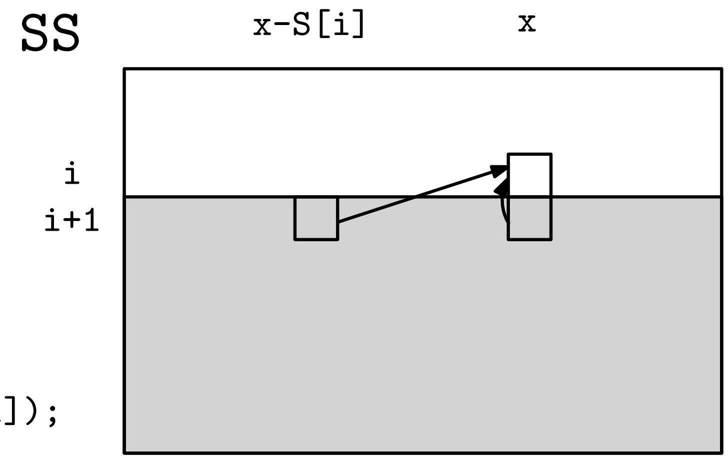
Example: Subset Sum

```
int S[20] = {...}; int x = ...;     // initialize S & x
int SS[20][MAX_X]; memset(SS, UNDEFINED, sizeof(SS));
// returns true iff exists subset within S[i:n-1]
// that adds up to x
SubsetSum(i, x):
  if (x < 0 or i > n)  return S[i][x] = false;
  else if (x == 0)     return S[i][x] = true;
  else if (SS[i][x] != UNDEFINED) return SS[i][x];
  else
    return SS[i][x] = SubsetSum(i+1, x) or SubsetSum(i+1, x-S[i]);
main() {
  return SubsetSum(0, x);
}
```



Example: Subset Sum

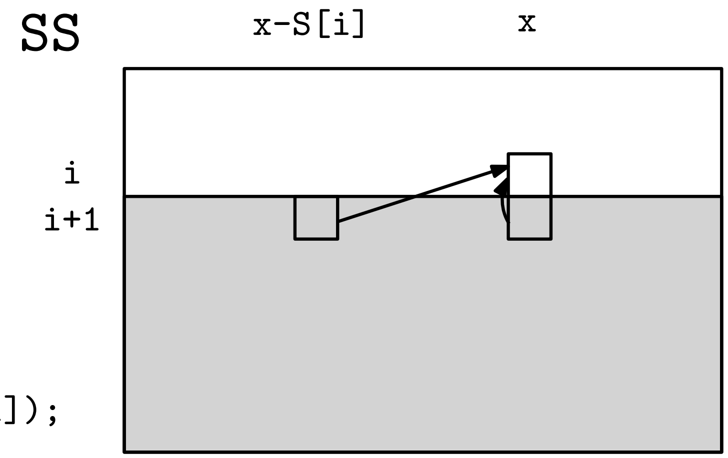
```
int S[20] = {...}; int x = ...;     // initialize S & x
int SS[20][MAX_X]; memset(SS, UNDEFINED, sizeof(SS));
// returns true iff exists subset within S[i:n-1]
// that adds up to x
SubsetSum(i, x):
  if (x < 0 or i > n)  return S[i][x] = false;
  else if (x == 0)     return S[i][x] = true;
  else if (SS[i][x] != UNDEFINED) return SS[i][x];
  else
    return SS[i][x] = SubsetSum(i+1, x) or SubsetSum(i+1, x-S[i]);
main() {
  return SubsetSum(0, x);
}
```




```
int S[20] = {...}; int x = ...;     // initialize S & x
int SS[20+1][MAX_X]; memset(SS, false, sizeof(SS));
// returns true iff exists subset within S[i:n]
// that adds up to x
main():
  for (i = 19; i >=0; i--)
     SS[i,0] = true;
  for (i = 19; i >=0; i--)
     for (j = 1; j < MAX_X; j++)
       SS[i,j] = SS[i+1][j] or SS[i+1][j-S[i]]
  return SS[0][x];
```

Example: Subset Sum

```
int S[20] = {...}; int x = ...;     // initialize S & x
int SS[20][MAX_X]; memset(SS, UNDEFINED, sizeof(SS));
// returns true iff exists subset within S[i:n-1]
// that adds up to x
SubsetSum(i, x):
  if (x < 0 or i > n)  return S[i][x] = false;
  else if (x == 0)     return S[i][x] = true;
  else if (SS[i][x] != UNDEFINED) return SS[i][x];
  else
    return SS[i][x] = SubsetSum(i+1, x) or SubsetSum(i+1, x-S[i]);
main() {
  return SubsetSum(0, x);
}
```




```
int S[20] = {...}; int x = ...;     // initialize S & x
int SS[20+1][MAX_X]; memset(SS, false, sizeof(SS));
// returns true iff exists subset within S[i:n]
// that adds up to x
main():
  for (i = 19; i >=0; i--)
     SS[i,0] = true;
  for (i = 19; i >=0; i--)
     for (j = 1; j < MAX_X; j++)
       SS[i,j] = SS[i+1][j] or SS[i+1][j-S[i]]
  return SS[0][x];
```

Can also fill it out in ascending order of `i`, by looking at subproblems `S[0:i]`, instead of `S[i:n-1]`

0-1 Knapsack

Problem (0-1 Knapsack). *Given a set S of n items, each with its own value V_i and weight W_i for all $1 \leq i \leq n$ and a maximum knapsack capacity C, compute the maximum value of the items that you can carry. You cannot take fractions of items.*

0-1 Knapsack

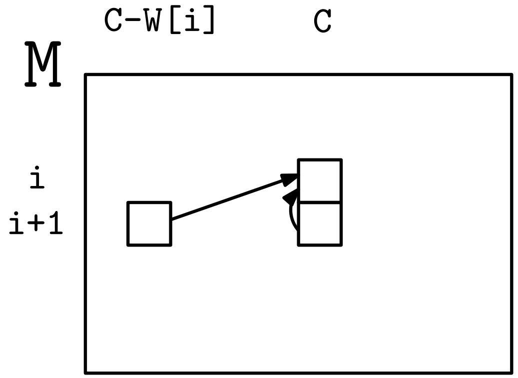
Problem (0-1 Knapsack)**.** *Given a set S of n items, each with its own value V_i and weight W_i for all $1 \leq i \leq n$ and a maximum knapsack capacity C, compute the maximum value of the items that you can carry. You cannot take fractions of items.*

```
int M[maxN+1][maxC];
maxV(i,C) {
  if (i > n || C <= 0) return 0;
  if (M[i][C] != UNDEFINED) return M[i, C];
  if (W[i] > C)
    return  M[i][C]=maxV(i+1, C);
  return M[i][C]=max(maxV(i+1, C),
                V[i]+maxV(i+1, C-W[i]));
}
main() { memset(M, UNDEFINED, sizeof(M));
  return maxV(1, C);
}
```

0-1 Knapsack

Problem (0-1 Knapsack). *Given a set S of n items, each with its own value V_i and weight W_i for all $1 \leq i \leq n$ and a maximum knapsack capacity C, compute the maximum value of the items that you can carry. You cannot take fractions of items.*

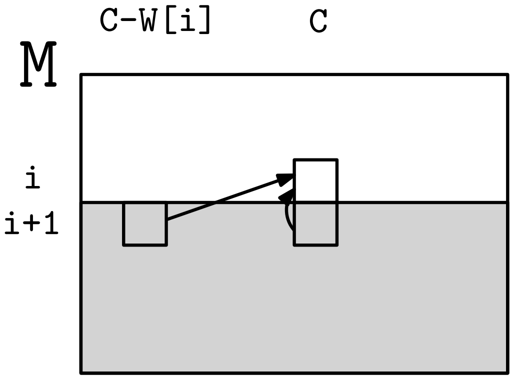
```
int M[maxN+1][maxC];
maxV(i,C) {
  if (i > n || C <= 0) return 0;
  if (M[i][C] != UNDEFINED) return M[i, C];
  if (W[i] > C)
    return M[i][C]=maxV(i+1, C);
  return M[i][C]=max(maxV(i+1, C),
             V[i]+maxV(i+1, C-W[i]));
}
main() { memset(M, UNDEFINED, sizeof(M));
  return maxV(1, C);
}
```



0-1 Knapsack

Problem (0-1 Knapsack). *Given a set S of n items, each with its own value V_i and weight W_i for all $1 \leq i \leq n$ and a maximum knapsack capacity C, compute the maximum value of the items that you can carry. You cannot take fractions of items.*

```
int M[maxN+1][maxC];
maxV(i,C) {
  if (i > n || C <= 0) return 0;
  if (M[i][C] != UNDEFINED) return M[i, C];
  if (W[i] > C)
    return M[i][C]=maxV(i+1, C);
  return M[i][C]=max(maxV(i+1, C),
               V[i]+maxV(i+1, C-W[i]));
}
main() { memset(M, UNDEFINED, sizeof(M));
  return maxV(1, C);
}
```



Why Bottom-up DP?

Why Bottom-up DP?

Cons:

- Extra work if recursion is already defined
- Less intuitive
- Might use extra memory (all memo entries must be filled)

Why Bottom-up DP?

Cons:

- Extra work if recursion is already defined
- Less intuitive
- Might use extra memory (all memo entries must be filled)

Pros:

- A bit faster (no recursive overhead)
- Easier to analyze running time

Why Bottom-up DP?

Cons:

- Extra work if recursion is already defined
- Less intuitive
- Might use extra memory (all memo entries must be filled)

Pros:

- A bit faster (no recursive overhead)
- Easier to analyze running time
- Sometimes top-down recursion is harder (or impossible) to define

Example

Problem. *In a football game, after every scoring play, the cheerleaders do as many jumps as the total number of points on the scoreboard. For example, if the team first scored a touchdown (7 pts), then a field goal (3 pts), then a safety (2 pts), the cheerleaders did* $7 + 10 + 12 = 29$ *total jumps.*
Given the number n *of total jumps, compute the largest possible number of points scored in the game?*

Example

Problem. *In a football game, after every scoring play, the cheerleaders do as many jumps as the total number of points on the scoreboard. For example, if the team first scored a touchdown (7 pts), then a field goal (3 pts), then a safety (2 pts), the cheerleaders did* $7 + 10 + 12 = 29$ *total jumps.*
Given the number n of total jumps, compute the largest possible number of points scored in the game?

You may assume the possible points are given as a set of S of m positive integers, with the largest value at most 20. For example, in regular football rules, $m = 5$ and $S = \{2, 3, 6, 7, 8\}$.

Recursive Solution

Recursive Solution

Let $P(n)$ define the maximum number of points for n total jumps. What is the recursive definition of $P(n)$?

Hint: Try all possible values for the last score and take the maximum

Recursive Solution

Let $P(n)$ define the maximum number of points for n total jumps. What is the recursive definition of $P(n)$?

Hint: Try all possible values for the last score and take the maximum

$$P(n) = \max \left\{ \begin{array}{l} P(n - S[0]) + S[0] \\ P(n - S[1]) + S[1] \\ P(n - S[2]) + S[2] \\ \ldots \\ P(n - S[m-1]) + S[m-1] \end{array} \right\} \quad ?$$

Recursive Solution

Let $P(n)$ define the maximum number of points for n total jumps. What is the recursive definition of $P(n)$?

Hint: Try all possible values for the last score and take the maximum

$$P(n) = \max \left\{ \begin{array}{l} P(n - S[0]) + S[0] \\ P(n - S[1]) + S[1] \\ P(n - S[2]) + S[2] \\ \ldots \\ P(n - S[m-1]) + S[m-1] \end{array} \right\} \quad ?$$

In the previous example:

- Points scored: 7, 3, 2
- Total jumps = 7 + 10 + 12 = 29
- $P(29) = P(29 - 2) + 2$?

Recursive Solution

Let $P(n)$ define the maximum number of points for n total jumps. What is the recursive definition of $P(n)$?

Hint: Try all possible values for the last score and take the maximum

$$P(n) = \max \left\{ \begin{array}{l} P(n - S[0]) + S[0] \\ P(n - S[1]) + S[1] \\ P(n - S[2]) + S[2] \\ \ldots \\ P(n - S[m-1]) + S[m-1] \end{array} \right\} \quad ?$$

In the previous example:

- Points scored: 7, 3, 2
- Total jumps = (7 + 10) + 12 = 29
- $P(29) = P(29 - 2) + 2$?

Recursive Solution

Let $P(n)$ define the maximum number of points for n total jumps. What is the recursive definition of $P(n)$?

Hint: Try all possible values for the last score and take the maximum

$$P(n) = \max \left\{ \begin{array}{l} P(n - S[0]) + S[0] \\ P(n - S[1]) + S[1] \\ P(n - S[2]) + S[2] \\ \ldots \\ P(n - S[m-1]) + S[m-1] \end{array} \right\} \quad ?$$

In the previous example:

- Points scored: 7, 3, 2
- Total jumps = (7 + 10) + 12 = 29
- $P(29) = P(29 - 2) + 2$? ← P(29) = P(17) + 2

Recursive Solution

Let $P(n)$ define the maximum number of points for n total jumps. What is the recursive definition of $P(n)$?

Hint: Try all possible values for the last score and take the maximum

$$P(n) = \max \left\{ \begin{array}{l} P(n - S[0]) + S[0] \\ P(n - S[1]) + S[1] \\ P(n - S[2]) + S[2] \\ \dots \\ P(n - S[m-1]) + S[m-1] \end{array} \right\} \quad ?$$

In the previous example:

If given the current score x:

$$P(29) = P(29 - x) + 2$$

- Points scored: 7, 3, 2
- Total jumps = (7 + 10) + 12 = 29
- $P(29) = P(29 - 2) + 2$? ← P(29) = P(17) + 2

Recursive Solution

Let $P(n)$ define the maximum number of points for n total jumps. What is the recursive definition of $P(n)$?

Hint: Try all possible values for the last score and take the maximum

$$P(n) = \max \left\{ \begin{array}{l} P(n - S[0]) + S[0] \\ P(n - S[1]) + S[1] \\ P(n - S[2]) + S[2] \\ \dots \\ P(n - S[m-1]) + S[m-1] \end{array} \right\} \quad ?$$

In the previous example:

- Points scored: 7, 3, 2
- Total jumps = (7 + 10) + 12 = 29
- $P(29) = P(29 - 2) + 2$? ← P(29) = P(17) + 2

If given the current score x:

$$P(29) = P(29 - x) + 2$$
$$P(n) = P(n - x) + S[i]$$

Recursive Solution

Let $P(n)$ define the maximum number of points for n total jumps. What is the recursive definition of $P(n)$?

Hint: Try all possible values for the last score and take the maximum

$$P(n, x) = \max \left\{ \begin{array}{l} P(n-x, x-S[0]) + S[0] \\ P(n-x, x-S[1]) + S[1] \\ P(n-x, x-S[2]) + S[2] \\ \ldots \\ P(n-x, x-S[m-1]) + S[m-1] \end{array} \right\}$$

In the previous example:

- Points scored: 7, 3, 2
- Total jumps = (7 + 10) + 12 = 29
- $P(29) = P(29 - 2) + 2$? ← P(29) = P(17) + 2

If given the current score x:

$$P(29) = P(29 - x) + 2$$
$$P(n) = P(n - x) + S[i]$$

Recursive Solution

Let $P(n)$ define the maximum number of points for n total jumps. What is the recursive definition of $P(n)$?

Hint: Try all possible values for the last score and take the maximum

$$x = P(n, x) = \max \left\{ \begin{array}{l} P(n-x, x-S[0]) + S[0] \\ P(n-x, x-S[1]) + S[1] \\ P(n-x, x-S[2]) + S[2] \\ \ldots \\ P(n-x, x-S[m-1]) + S[m-1] \end{array} \right\}$$

In the previous example:

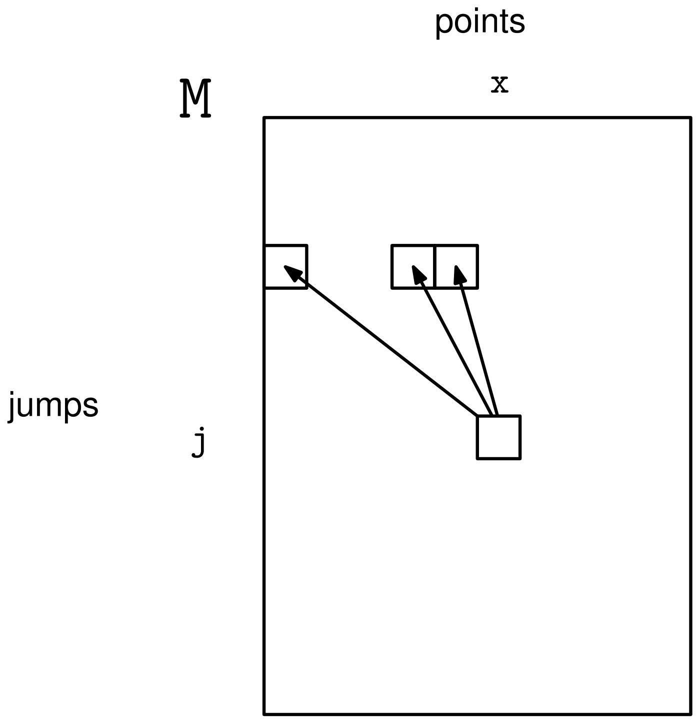
- Points scored: 7, 3, 2
- Total jumps = (7 + 10) + 12 = 29
- $P(29) = P(29-2) + 2$? ← P(29) = P(17) + 2

If given the current score x:

$$P(29) = P(29-x) + 2$$
$$P(n) = P(n-x) + S[i]$$

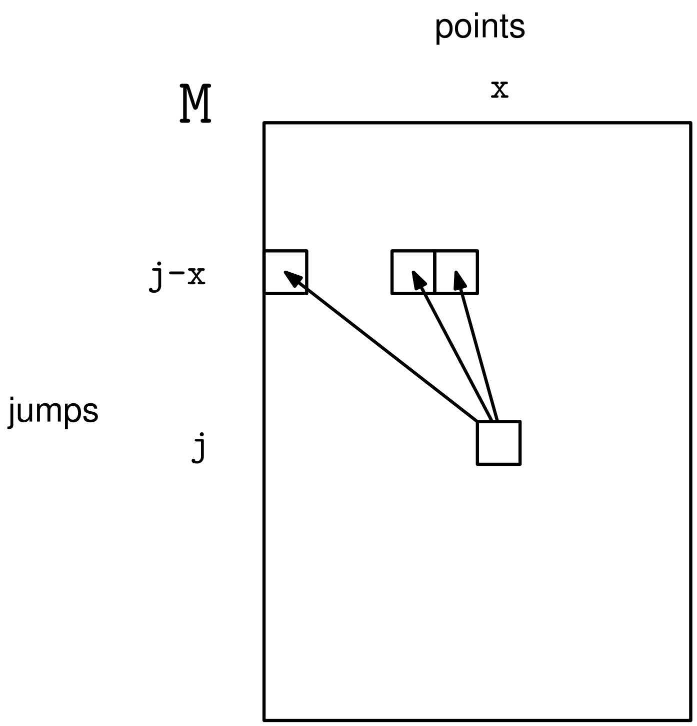
Bottom-up Solution

$$x = P(n, x) = \max \left\{ \begin{array}{l} P(n-x, x-S[0]) + S[0] \\ P(n-x, x-S[1]) + S[1] \\ P(n-x, x-S[2]) + S[2] \\ \ldots \\ P(n-x, x-S[m-1]) + S[m-1] \end{array} \right\}$$



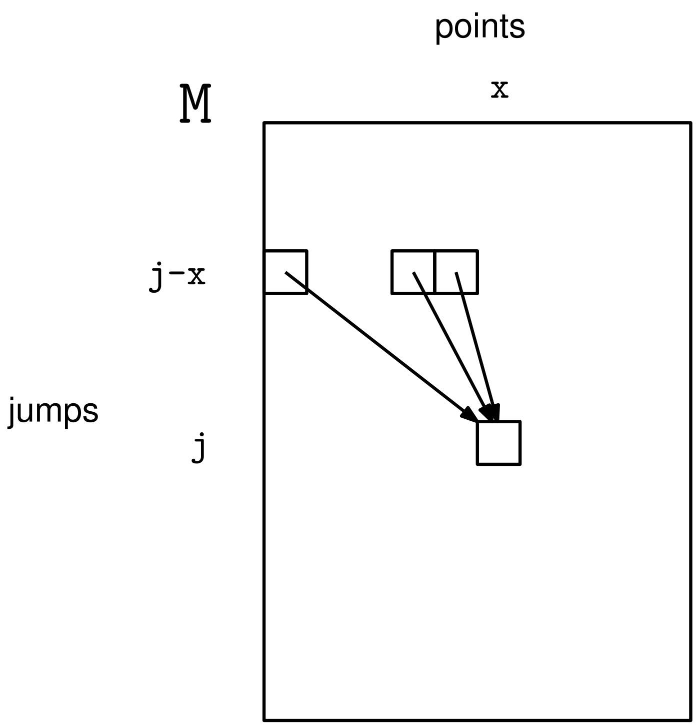
Bottom-up Solution

$$x = P(n, x) = \max \left\{ \begin{array}{l} P(n - x, x - S[0]) + S[0] \\ P(n - x, x - S[1]) + S[1] \\ P(n - x, x - S[2]) + S[2] \\ \ldots \\ P(n - x, x - S[m - 1]) + S[m - 1] \end{array} \right\}$$



Bottom-up Solution

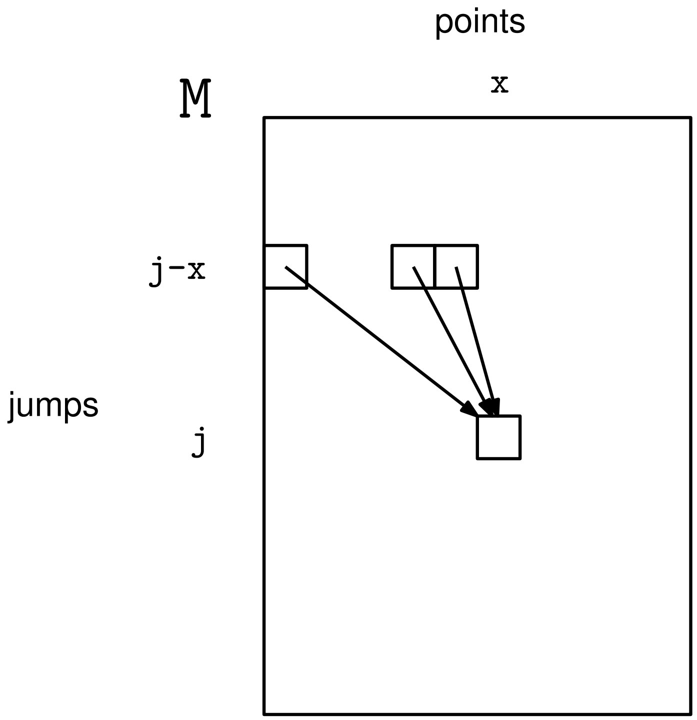
$$x = P(n, x) = \max \left\{ \begin{array}{l} P(n - x, x - S[0]) + S[0] \\ P(n - x, x - S[1]) + S[1] \\ P(n - x, x - S[2]) + S[2] \\ \ldots \\ P(n - x, x - S[m - 1]) + S[m - 1] \end{array} \right\}$$



Bottom-up Solution

$$x = P(n, x) = \max \begin{Bmatrix} P(n-x, x-S[0]) + S[0] \\ P(n-x, x-S[1]) + S[1] \\ P(n-x, x-S[2]) + S[2] \\ \ldots \\ P(n-x, x-S[m-1]) + S[m-1] \end{Bmatrix}$$

$M[j, x]$ = "Is it possible to have x points for j jumps?"

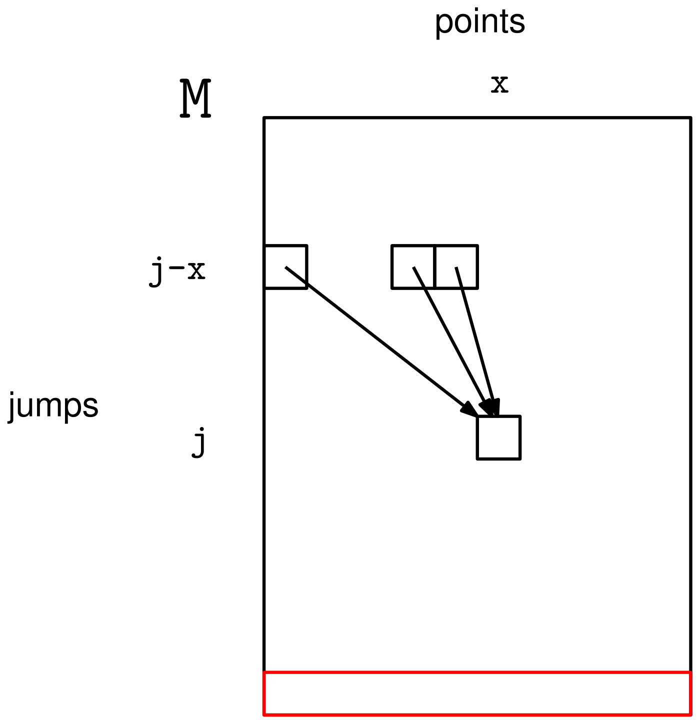


$$M[j, x] = \text{OR} \begin{Bmatrix} M[j-x, x-S[0]] \\ M[j-x, x-S[1]] \\ M[j-x, x-S[2]] \\ \ldots \\ M[j-x, x-S[m-1]] \end{Bmatrix}$$

Bottom-up Solution

$$x = P(n,x) = \max \left\{ \begin{array}{l} P(n-x, x-S[0]) + S[0] \\ P(n-x, x-S[1]) + S[1] \\ P(n-x, x-S[2]) + S[2] \\ \dots \\ P(n-x, x-S[m-1]) + S[m-1] \end{array} \right\}$$

$M[j,x]$ = "Is it possible to have x points for j jumps?"



$$M[j,x] = \text{OR} \left\{ \begin{array}{l} M[j-x, x-S[0]] \\ M[j-x, x-S[1]] \\ M[j-x, x-S[2]] \\ \dots \\ M[j-x, x-S[m-1]] \end{array} \right\}$$

Answer: the rightmost x in the n-th row, s.t. $M[n][x] = true$

Homework

- Programming assignment (DP)
- Understand Optimal BST problem & solution Ch 14.5