Chapter 3
Dynamic Programming

Divesh Aggarwal
School of Computing
Department of Computer Science

Backtracking Algorithm for Fibonacci Sequence

NUS National University of Singapore

Suppose we want to compute F_n given by the following recurrence

$$F_n = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ F_{n-1} + F_{n-2} & \text{otherwise} \end{cases}$$

```
RecFibo(n):
    if n = 0
        return 0
    else if n = 1
        return 1
    else
        return RecFibo(n − 1) + RecFibo(n − 2)
```

Backtracking can be slow

NUS National University of Singapore

$$T(0) = 1, \quad T(1) = 1, \quad T(n) = T(n-1) + T(n-2) + 1$$

$$\begin{aligned} T(n) &> 2\,T(n-2) \\ &> 2^2\,T(n-4) \\ &\;. \\ &\;. \\ &\;. \\ &> 2^{n/2}\,T(0) = 2^{n/2} \end{aligned}$$

Can we speed up this algorithm?

Backtracking algorithm repeats computation

NUS
National University of Singapore

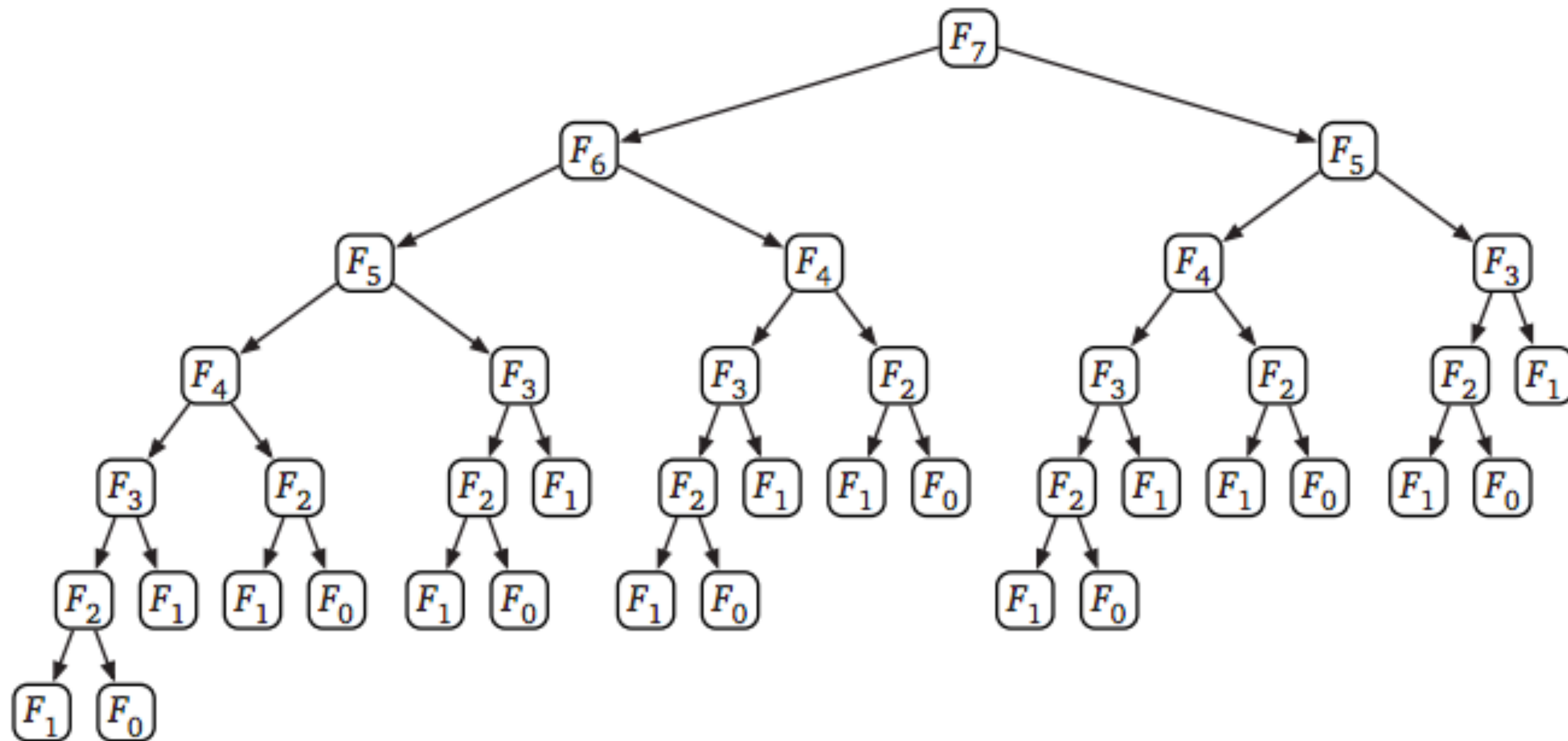


Figure 3.1. The recursion tree for computing F_7; arrows represent recursive calls.

Backtracking algorithm repeats computation

NUS National University of Singapore

The obvious reason for the recursive algorithm's lack of speed is that it computes the same Fibonacci numbers over and over and over. A single call to RecFibo(n) results in one recursive call to RecFibo($n-1$), two recursive calls to RecFibo($n-2$), three recursive calls to RecFibo($n-3$), five recursive calls to RecFibo($n-4$), and in general F_{k-1} recursive calls to RecFibo($n-k$) for any integer $0 \leq k < n$. Each call is recomputing some Fibonacci number from scratch.

Memo(r)ization: Remember everything

NUS National University of Singapore

This optimization technique, now known as *memoization* (yes, without an R), is usually credited to Donald Michie in 1967, but essentially the same technique was proposed in 1959 by Arthur Samuel.[5]

```
MemFibo(n):
  if n = 0
      return 0
  else if n = 1
      return 1
  else
      if F[n] is undefined
          F[n] ← MemFibo(n − 1) + MemFibo(n − 2)
      return F[n]
```

Why memoization without an "r"?

Memoization trims the recursion tree

NUS
National University of Singapore

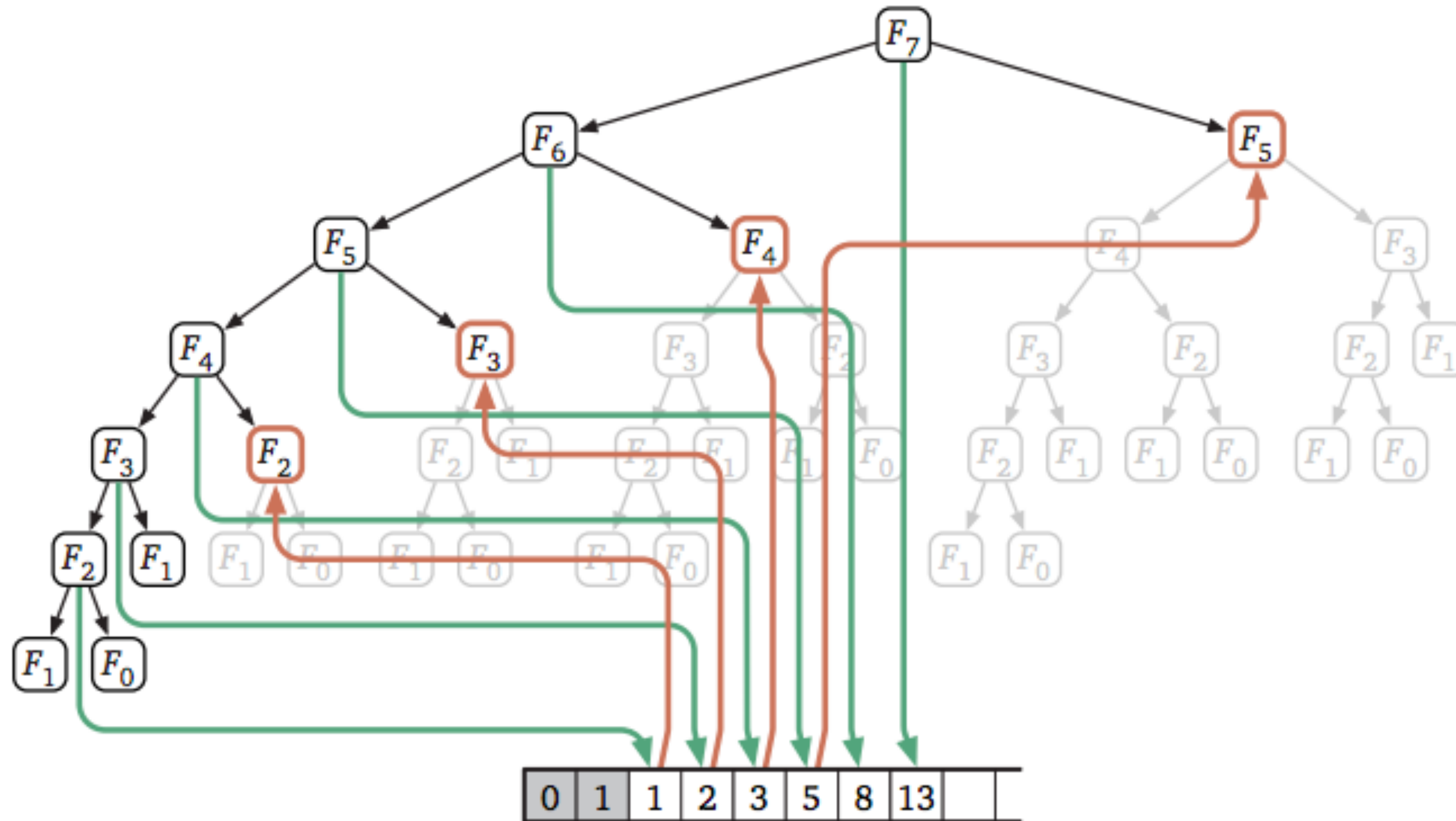


Figure 3.2. The recursion tree for F_7 trimmed by memoization. Downward green arrows indicate writing into the memoization array; upward red arrows indicate reading from the memoization array.

Dynamic Programming: Fill Deliberately

Once we see how the array $F[\,]$ is filled, we can replace the memoized recurrence with a simple for-loop that *intentionally* fills the array in that order, instead of relying on a more complicated recursive algorithm to do it for us accidentally.

```
ITERFIBO(n):
  F[0] ← 0
  F[1] ← 1
  for i ← 2 to n
      F[i] ← F[i − 1] + F[i − 2]
  return F[n]
```

Now the time analysis is immediate: ITERFIBO clearly uses $O(n)$ ***additions*** and stores $O(n)$ *integers*.

This is our first explicit ***dynamic programming*** algorithm.

Example: Text Segmentation

NUS National University of Singapore

For our next dynamic programming algorithm, let's consider the text segmentation problem from the previous chapter. We are given a string A[1, …, n] , and we want to know whether A can be partitioned into a sequence of words.

We solved this problem by defining a function $\mathit{Splittable}(i)$ that returns TRUE if and only if the suffix $A[i\,..\,n]$ can be partitioned into a sequence of words. We need to compute $\mathit{Splittable}(1)$. This function satisfies the recurrence

$$\mathit{Splittable}(i) = \begin{cases} \text{TRUE} & \text{if } i > n \\ \displaystyle\bigvee_{j=i}^{n} \big(\mathit{IsWord}(i,j) \wedge \mathit{Splittable}(j+1)\big) & \text{otherwise} \end{cases}$$

where $\mathit{IsWord}(i,j)$ is shorthand for $\mathit{IsWord}(A[i\,..\,j])$. This recurrence translates directly into a recursive backtracking algorithm that calls the IsWord subroutine $O(2^n)$ times in the worst case.

Backtracking repeats computation

NUS National University of Singapore

But for any fixed string $A[1..n]$, there are only n different ways to call the recursive function *Splittable*(i)—one for each value of i between 1 and $n+1$—and only $O(n^2)$ different ways to call IsWord(i, j)—one for each pair (i, j) such that $1 \leq i \leq j \leq n$. Why are we spending exponential time computing only a polynomial amount of stuff?

Dynamic Programming solution

NUS National University of Singapore

```
FASTSPLITTABLE(A[1..n]):
    SplitTable[n+1] ← TRUE
    for i ← n down to 1
        SplitTable[i] ← FALSE
        for j ← i to n
            if ISWORD(i, j) and SplitTable[j+1]
                SplitTable[i] ← TRUE
    return SplitTable[1]
```

Figure 3.3. Interpunctio verborum velox

$T(n) = O(n^2)$ calls to ISWORD

NUS National University of Singapore

Dynamic programming is *not* about filling in tables.
It's about smart recursion!

Dynamic programming algorithms are best developed in two distinct stages.

1. **Formulate the problem recursively.**
2. **Build solutions to your recurrence from the bottom up.**

Greed is almost never good

NUS National University of Singapore

- If we're incredibly lucky, we can bypass all the recurrences and tables and so forth, and solve the problem using a greedy algorithm.
- Greedy algorithm looks at only one branch of the recursion tree
- For example, a greedy algorithm for the text segmentation problem might find the shortest (or, if you prefer, longest) prefix of the input string that is a word, accept that prefix as the first word in the segmentation, and then recursively segment the remaining suffix of the input string
- Similarly, a greedy algorithm for the longest increasing subsequence problem might look for the smallest element of the input array, accept that element as the start of the target subsequence, and then recursively look for the longest increasing subsequence to the right of that element.

If these sound like stupid hacks to you, pat yourself on the back; these aren't even close to correct solutions.

Greed is not good

NUS National University of Singapore

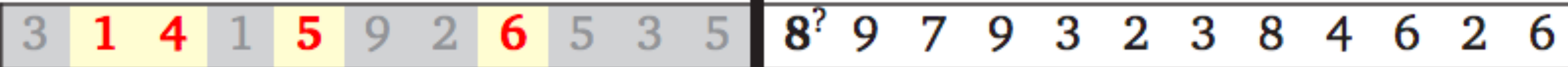
Greedy algorithms never work!
Use dynamic programming instead!

What, never?
No, never!
What, *never*?
Well. . . hardly ever.[10]

Whenever you write—or even *think*—the word "greeDY",
your subconscious is telling you to use DYnamic programming.

Longest increasing subsequence

Another problem we considered in the previous chapter was computing the length of the longest increasing subsequence of a given array $A[1..n]$ of numbers. We developed two different recursive backtracking algorithms for this problem. Both algorithms run in $O(2^n)$ time in the worst case; both algorithms can be sped up significantly via dynamic programming.



NUS National University of Singapore

We saw the following..

So we can reformulate our recursive *problem* as follows:

> Given two indices i and j, where $i < j$, find the longest increasing subsequence of $A[j..n]$ in which every element is larger than $A[i]$.

Let $LISbigger(i, j)$ denote the *length* of the longest increasing subsequence of $A[j..n]$ in which every element is larger than $A[i]$. Our recursive strategy gives us the following recurrence:

$$LISbigger(i,j) = \begin{cases} 0 & \text{if } j > n \\ LISbigger(i, j+1) & \text{if } A[i] \geq A[j] \\ \max \begin{Bmatrix} LISbigger(i, j+1) \\ 1 + LISbigger(j, j+1) \end{Bmatrix} & \text{otherwise} \end{cases}$$

The backtracking algorithm

```
LISBIGGER(i, j):
  if j > n
      return 0
  else if A[i] ≥ A[j]
      return LISBIGGER(i, j + 1)
  else
      skip ← LISBIGGER(i, j + 1)
      take ← LISBIGGER(j, j + 1) + 1
      return max{skip, take}
```

Finally, we connected our recursive strategy to the original problem: Finding the longest increasing subsequence of an array with no other constraints.

```
LIS(A[1 .. n]):
  A[0] ← −∞
  return LISBIGGER(0, 1)
```

Running time: $T(n) = 2T(n-1) + 1$, which as usual implies that $T(n) = O(2^n)$.

Dynamic Programming

Each recursive subproblem is identified by two indices i and j, so there are only $O(n^2)$ distinct recursive subproblems to consider. We can memoize the results of these subproblems into a two-dimensional array ***LISbigger***$[\mathbf{0..n}, \mathbf{1..n}]$.[12] Moreover, each subproblem can be solved in $O(1)$ time, not counting recursive calls, so we should expect the final dynamic programming algorithm to run in $\boldsymbol{O(n^2)}$ ***time***.

The order in which the memoized recursive algorithm fills this array is not immediately clear; all we can tell from the recurrence is that each entry *LISbigger*$[i, j]$ is filled in *after* the entries *LISbigger*$[i, j+1]$ and *LISbigger*$[j, j+1]$

Fortunately, this partial information is enough to give us a *valid* evaluation

Dynamic Programming Solution

NUS
National University of Singapore

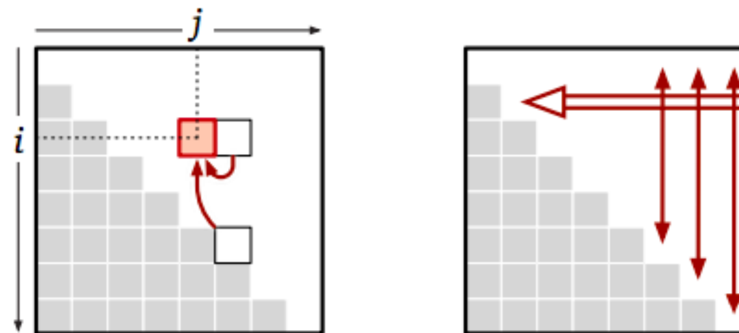


Figure 3.4. Subproblem dependencies for longest increasing subsequence, and a valid evaluation order

```
FastLIS(A[1..n]):
  A[0] ← −∞                               ⟨⟨Add a sentinel⟩⟩
  for i ← 0 to n                          ⟨⟨Base cases⟩⟩
      LISbigger[i, n+1] ← 0
  for j ← n down to 1
      for i ← 0 to j−1                    ⟨⟨...or whatever⟩⟩
          keep ← 1 + LISbigger[j, j+1]
          skip ← LISbigger[i, j+1]
          if A[i] ≥ A[j]
              LISbigger[i, j] ← skip
          else
              LISbigger[i, j] ← max{keep, skip}
  return LISbigger[0, 1]
```

Example: Edit Distance

The ***edit distance*** between two strings is the minimum number of letter insertions, letter deletions, and letter substitutions required to transform one string into the other. For example, the edit distance between FOOD and MONEY is at most four:

FOOD → MOOD → MON^D → MONED → MONEY

This distance function was independently proposed by Vladimir Levenshtein in 1965 (working on coding theory), Taras Vintsyuk in 1968 (working on speech recognition), and Stanislaw Ulam in 1972 (working with biological sequences). For this reason, edit distance is sometimes called ***Levenshtein distance*** or ***Ulam distance*** (but strangely, never "Vintsyuk distance").

Example: Edit Distance

Unfortunately, it's not so easy in general to tell when a sequence of edits is as short as possible. For example, the following alignment shows that the distance between the strings ALGORITHM and ALTRUISTIC is at most 6. Is that the best we can do?

```
A L G O R   I   T H M
A L   T R U I S T I C
```

Recursive substructure

```
A L G O R   I   T H M
A L   T R U I S T I C
```

- Suppose we have the gap representation for the shortest edit sequence for two strings.
- If we remove the last column, the remaining columns must represent the shortest edit sequence for the remaining prefixes.
- We can easily prove this observation by contradiction: If the prefixes had a shorter edit sequence, gluing the last column back on would gives us a shorter edit sequence for the original strings.
- So once we figure out what should happen in the last column, the Recursion can figure out the rest of the optimal gap representation.

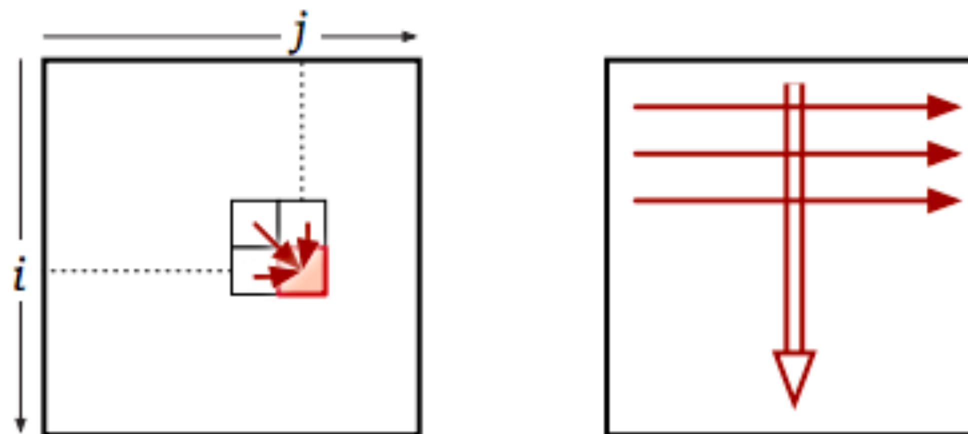
Recurrence

$$Edit(i,j) = \begin{cases} i & \text{if } j = 0 \\ j & \text{if } i = 0 \\ \min \left\{ \begin{array}{c} Edit(i, j-1) + 1 \\ Edit(i-1, j) + 1 \\ Edit(i-1, j-1) + [A[i] \neq B[j]] \end{array} \right\} & \text{otherwise} \end{cases}$$

Dynamic Programming

- **Dependencies:** Each entry $Edit[i, j]$ depends only on its three neighboring entries $Edit[i-1, j]$, $Edit[i, j-1]$, and $Edit[i-1, j-1]$.

- **Evaluation order:** If we fill this array in standard row-major order—row by row from top down, each row from left to right—then whenever we reach an entry in the array, all the entries it depends on are already available. (This isn't the *only* evaluation order we could use, but it works, so let's go with it.)



The Algorithm

```
EditDistance(A[1..m], B[1..n]):
    for j ← 0 to n
        Edit[0, j] ← j

    for i ← 1 to m
        Edit[i, 0] ← i
        for j ← 1 to n
            ins ← Edit[i, j − 1] + 1
            del ← Edit[i − 1, j] + 1
            if A[i] = B[j]
                rep ← Edit[i − 1, j − 1]
            else
                rep ← Edit[i − 1, j − 1] + 1
            Edit[i, j] ← min {ins, del, rep}

    return Edit[m, n]
```

Space/Time Complexity: O(mn)

Subset Sum

NUS National University of Singapore

$SS(i,t) = \textsc{True}$ if and only if some subset of $X[i\,..\,n]$ sums to t.

We need to compute $SS(1,T)$. This function satisfies the following recurrence:

$$SS(i,t) = \begin{cases} \textsc{True} & \text{if } t = 0 \\ \textsc{False} & \text{if } t < 0 \text{ or } i > n \\ SS(i+1,t) \vee SS(i+1,t-X[i]) & \text{otherwise} \end{cases}$$

For Dynamic programming, we want to avoid recursing on $t < 0$

$$SS(i,t) = \begin{cases} \textsc{True} & \text{if } t = 0 \\ \textsc{False} & \text{if } i > n \\ SS(i+1,t) & \text{if } t < X[i] \\ SS(i+1,t) \vee SS(i+1,t-X[i]) & \text{otherwise} \end{cases}$$

Subset Sum

- **Data structure:** We can memoize our recurrence into a two-dimensional array $S[1..n+1, 0..T]$, where $S[i,t]$ stores the value of $SS(i,t)$.
- **Evaluation order:** Each entry $S[i,t]$ depends on at most two other entries, both of the form $SS[i+1,\cdot]$. So we can fill the array by considering rows from bottom to top in the outer loop, and considering the elements in each row in arbitrary order in the inner loop.
- **Space and time:** The memoization structure uses $O(nT)$ space. If $S[i+1,t]$ and $S[i+1,t-X[i]]$ are already known, we can compute $S[i,t]$ in constant time, so the algorithm runs in $\boldsymbol{O(nT)}$ ***time***.

The Algorithm

```
FastSubsetSum(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]
```

Optimal Binary Search Trees

NUS National University of Singapore

Fix the frequency array f, and let $OptCost(i,k)$ denote the total search time in the optimal search tree for the subarray $A[i..k]$. We derived the following recurrence for the function $OptCost$:

$$OptCost(i,k) = \begin{cases} 0 & \text{if } i > k \\ \sum_{j=i}^{k} f[j] + \min_{i \le r \le k} \left\{ \begin{array}{l} OptCost(i, r-1) \\ \quad + OptCost(r+1, k) \end{array} \right\} & \text{otherwise} \end{cases}$$

For any pair of indices $i \le k$, let $F(i,k)$ denote the total frequency count for all the keys in the interval $A[i..k]$:

$$F(i,k) := \sum_{j=i}^{k} f[j]$$

The function F(i,k)

This function satisfies the following simple recurrence:

$$F(i,k) = \begin{cases} f[i] & \text{if } i = k \\ F(i,k-1) + f[k] & \text{otherwise} \end{cases}$$

We can compute all possible values of $F(i,k)$ in $O(n^2)$ time using—you guessed it!—dynamic programming! The usual mechanical steps give us the following dynamic programming algorithm:

```
InitF(f[1..n]):
  for i ← 1 to n
      F[i, i−1] ← 0
      for k ← i to n
          F[i,k] ← F[i,k−1] + f[k]
```

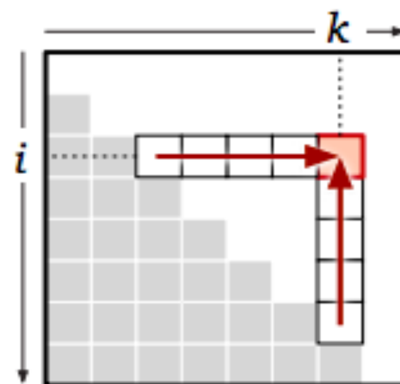
Back to Optimal Binary Search Trees

NUS National University of Singapore

We will use this short algorithm as an initialization subroutine. This initialization allows us to simplify the original *OptCost* recurrence as follows:

$$OptCost(i,k) = \begin{cases} 0 & \text{if } i > k \\ F[i,k] + \min\limits_{i \le r \le k} \left\{ \begin{array}{l} OptCost(i, r-1) \\ \quad + OptCost(r+1, k) \end{array} \right\} & \text{otherwise} \end{cases}$$

- **Dependencies:** Each entry $OptCost[i,k]$ depends on the entries $OptCost[i, j-1]$ and $OptCost[j+1,k]$, for all j such that $i \le j \le k$. In other words, each table entry depends on all entries either directly to the left or directly below.



Computing Optimal Cost

NUS National University of Singapore

The following subroutine fills the entry $OptCost[i, k]$, assuming all the entries it depends on have already been computed.

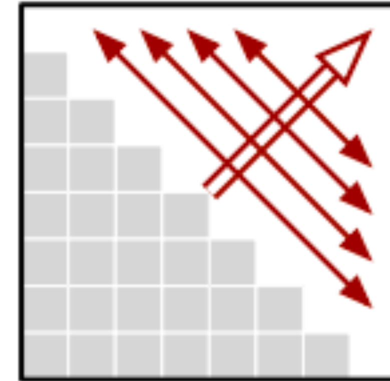
```
ComputeOptCost(i, k):
  OptCost[i, k] ← ∞
  for r ← i to k
      tmp ← OptCost[i, r − 1] + OptCost[r + 1, k]
      if OptCost[i, k] > tmp
          OptCost[i, k] ← tmp
  OptCost[i, k] ← OptCost[i, k] + F[i, k]
```

- **Evaluation order:** There are at least three different orders that can be used to fill the array. The first one that occurs to most students is to scan through the table one diagonal at a time, starting with the trivial base cases $OptCost[i, i-1]$ and working toward the final answer $OptCost[1, n]$, like so:

Dynamic Programming Algorithms(s)

NUS National University of Singapore

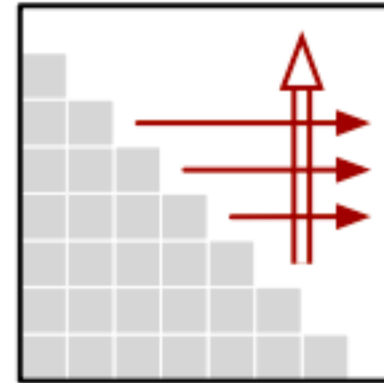
```
OptimalBST(f[1..n]):
  InitF(f[1..n])
  for i ← 1 to n+1
      OptCost[i, i−1] ← 0
  for d ← 0 to n−1
      for i ← 1 to n−d          ⟨⟨...or whatever⟩⟩
          ComputeOptCost(i, i+d)
  return OptCost[1, n]
```



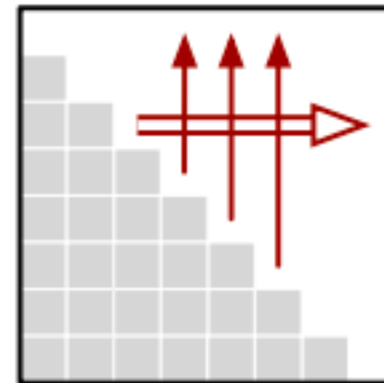
We could also traverse the array row by row from the bottom up, traversing each row from left to right, or column by column from left to right, traversing each columns from the bottom up.

Dynamic Programming Algorithms(s)

```
OptimalBST2(f[1..n]):
  InitF(f[1..n])
  for i ← n+1 downto 1
      OptCost[i, i−1] ← 0
      for j ← i to n
          ComputeOptCost(i, j)
  return OptCost[1, n]
```



```
OptimalBST3(f[1..n]):
  InitF(f[1..n])
  for j ← 0 to n+1
      OptCost[j+1, j] ← 0
      for i ← j downto 1
          ComputeOptCost(i, j)
  return OptCost[1, n]
```



Time and Space Complexity

- **Time and space:** The memoization structure uses $O(n^2)$ space. No matter which evaluation order we choose, we need $O(n)$ time to compute each entry $OptCost[i,k]$, so our overall algorithm runs in $\boldsymbol{O(n^3)}$ ***time***.

Maximum Independent Set in a Tree

For any node v in T, let $MIS(v)$ denote the size of the largest independent set in the subtree rooted at v. Any independent set in this subtree that excludes v itself is the union of independent sets in the subtrees rooted at the children of v. On the other hand, any independent set that *includes* v necessarily excludes all of v's children, and therefore includes independent sets in the subtrees rooted at v's grandchildren. Thus, the function MIS obeys the following recurrence, where the nonstandard notation $w \downarrow v$ means "w is a child of v":

Maximum Independent Set in a Tree

$$MIS(v) = \max\left\{\sum_{w \downarrow v} MIS(w),\ 1 + \sum_{w \downarrow v} \sum_{x \downarrow w} MIS(x)\right\}$$

We need to compute $MIS(r)$, where r is the root of T.

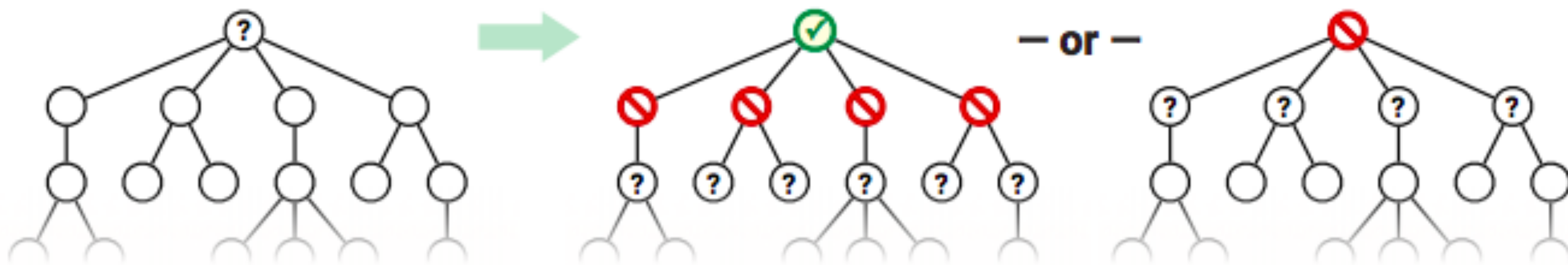


Figure 3.5. Computing the maximum independent set in a tree

How to Memoize?

NUS National University of Singapore

What data structure should we use to memoize this recurrence? The most natural choice is ***the tree T itself!*** Specifically, for each vertex v in T, we store the result of $MIS(v)$ in a new field $v.MIS$. (In principle, we *could* use an array instead, but then we'd need pointers back and forth between each node and its corresponding array entry, so why bother?)

What's a good order to consider the subproblems? The subproblem associated with any node v depends on the subproblems associated with the children and grandchildren of v. So we can visit the nodes in any order we like, provided that every vertex is visited before its parent; in particular, we can use a standard ***post-order*** traversal.

Dynamic Programming Algorithms

Here is the resulting dynamic programming algorithm. Yes, it's still recursive, because that's the most natural way to implement a post-order tree traversal.

```
TreeMIS(v):
  skipv ← 0
  for each child w of v
      skipv ← skipv + TreeMIS(w)
  keepv ← 1
  for each grandchild x of v
      keepv ← keepv + x.MIS
  v.MIS ← max{keepv, skipv}
  return v.MIS
```

What's the running time of the algorithm? The non-recursive time associated with each node v is proportional to the number of children and grandchildren of v; this is hard to analyze.

But we can turn the analysis around: Each vertex contributes a constant amount of time to its parent and its grandparent! Because each vertex has at most one parent and at most one grandparent, the algorithm runs in **O(n) time**.