CS 590: Algorithms

Design and Analysis Technique

Dynamic Programming and Greedy Algorithms

Dynamic Programming

8. Dynamic Programming

Dynamic Programming (Overview):

- It's a **technique** like divide-and-conquer, **not** a specific algorithm.
- Not really refers to computer programming. Developed back in the day when "programming" meant "tabular method".
- We use it for **optimization** problems.
  - Find a solution with the optimal value.
  - Minimization or Maximization (we will do both).
- **Four-step method:**
  1. **Characterize** the structure of an optimal solution.
  2. **Recursively** define the value of an optimal solution.
  3. **Compute** the value of an optimal solution, typically a bottom-up fashion.
  4. **Construct** an optimal solution from the computed information.

8. Dynamic Programming

Dynamic Programming Examples:

- Rod-cutting
- Longest Common Sequence
- Sequential Alignment – Smith-Waterman Algorithm
- Optimal BSTs

8.1. Rod Cutting

- How do we cut steel rods into pieces in order to maximize the revenue we can get?
- Each cut is free.
- The length of a rod is always an integral number of inches.

  - **Input:** A length n and table prices p_i for $i = 1, 2, \cdots, n$.
  - **Output:** The **maximum revenue** we can obtain for rods whose length sum up to n, computed as the sum of the prices for the individual rods.

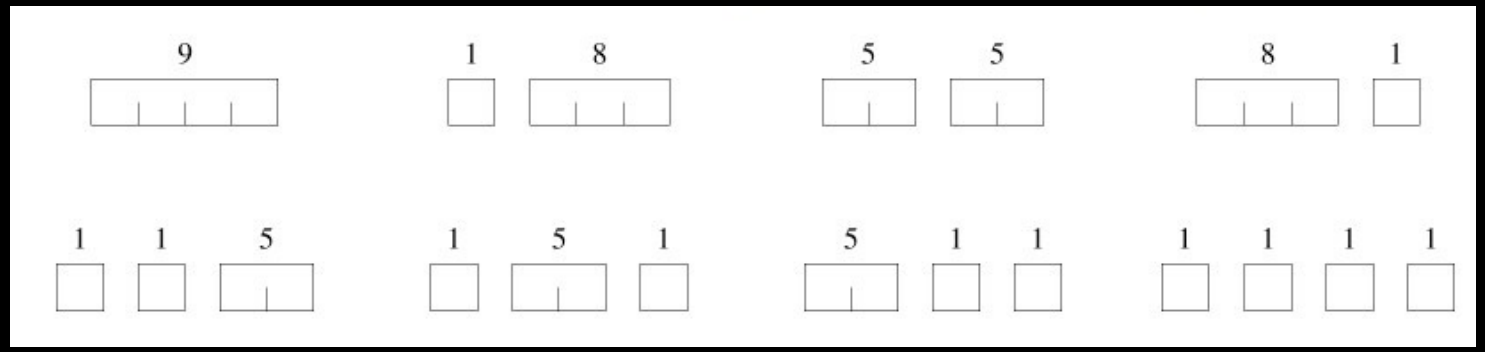
  ⇒ If p_n is large enough, then an optimal solution might not require any cuts (we leave them n inches long).

8.1. Rod Cutting

- Consider the example table below:

Length i	1	2	3	4	5	6	7	8
Price p_i	1	5	8	9	10	17	17	20

- We can cut the rod 2^{n-1} different ways.
- We can choose to cut or not to cut after each of the first $n-1$ inches (binary numbers).
- For example, there are 8 possible ways to cut a rod of length 4.



8.1. Rod Cutting

- Let r_i be the **maximum revenue** for a rod of length i.
- Express a solution as a sum of the individual rod length.
- We can determine the optimal revenues r_i for the example by inspection:

i	r_i	optimal solution
1	1	1 (no cuts)
2	5	2 (no cuts)
3	8	3 (no cuts)
4	10	2+2
5	13	2+3
6	17	6 (no cuts)
7	18	1+6 or 2+2+3
8	22	2+6

8.1. Rod Cutting

Determine the optimal revenue r_n by taking the maximum of

- p_n: the price we get by not making a cut.
- $r_1 + r_{n-1}$: the maximum revenue from a rod of 1 inch and a rod of $n-1$ inches.
- $r_2 + r_{n-2}$: the maximum revenue from a rod of 2 inches and a rod of $n-2$ inches.
- Until we reach $r_{n-1} + r_1$.
- This gives us:

$$r_n = \max(p_n, r_1 + r_{n-1}, r_2 + r_{n-2}, \dots, r_{n-1} + r_1)$$

Optimal substructure: To solve original problem of size n, solve on smaller sizes. We have two subproblems after a cut.

Optimal solution to original problem incorporates optimal solution to subproblems → **Solve subproblems independently**.

8.1. Rod Cutting

Simpler Way to decompose the problem:

- Every optimal solution has a **leftmost cut**.
- There is a first cut that gives a first piece of length i cut off the left end.
- The remaining piece of length $n - i$ is on the right.

Simplified Solution:

- We only need to divide **the remainder**, not the first piece.
- That leaves only **one subproblem**, rather than two subproblems to solve.
- If the solution has **no cuts**, then it has a first piece size of $i = n$ with revenue p_n and a remainder size of 0 with $r_0 = 0$.
- This gives us a simpler version of the equation for r_n:

$$r_n = \max_{1 \le i \le n} (p_i + r_{n-i})$$

8.1. Rod Cutting

length i	1	2	3	4	5	6	7	8
price p_i	1	5	8	9	10	17	17	20

i	r_i	optimal solution
1	1	1 (no cuts)
2	5	2 (no cuts)
3	8	3 (no cuts)
4	10	2+2
5	13	2+3
6	17	6 (no cuts)
7	18	1+6 or 2+2+3
8	22	2+6

8.1. Rod Cutting

Recursive top-down solution:

- Direct implementation of simpler equation for r_n.
- A call to **CUT-ROD(p, n)** returns the optimal revenue r_n.

8.1. Rod Cutting

Recursive top-down solution:

- Direct implementation of simpler equation for r_n.
- A call to **CUT-ROD(p, n)** returns the optimal revenue r_n.

Algorithm (CUT-ROD(p, n))

```
1       if (n = 0) then
2          return 0
3        q = −∞
4        for (1 ≤ i ≤ n) do
5          q = max(q,p[i]+CUT-ROD(p,n-i)
6        return q
```

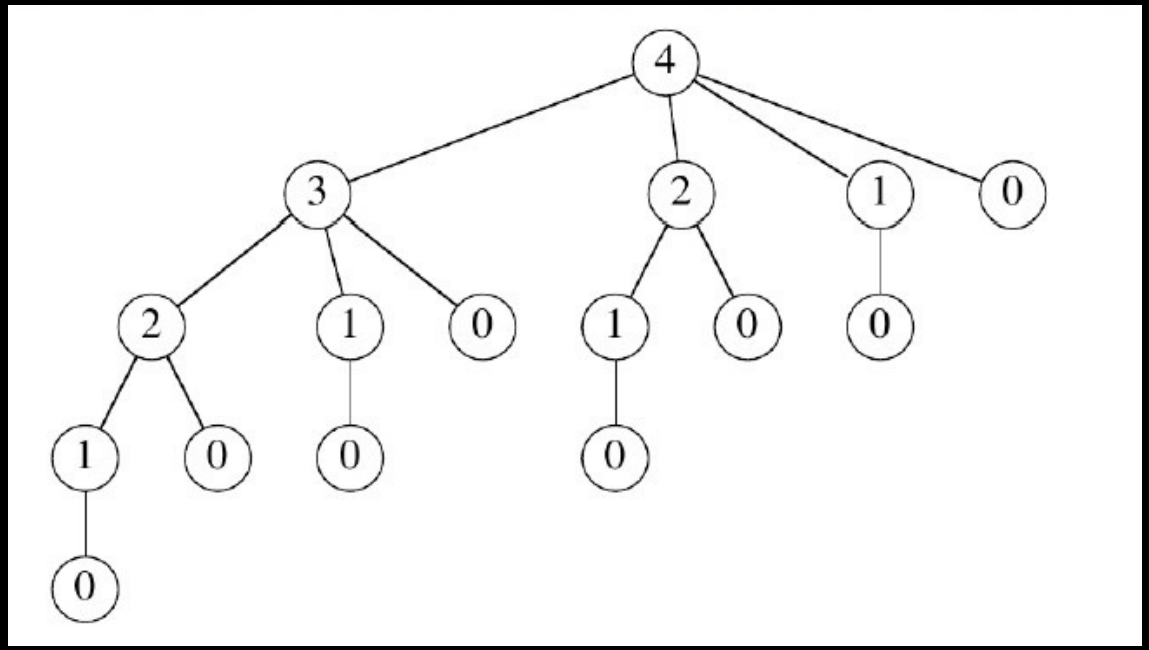
Problem: Our algorithm works, but it is **terribly inefficient**. The running time **doubles** each time n increases by 1.

- Could take more than an hour for n = 40.

8.1. Rod Cutting

Inefficient Example:

- **CUT-ROD** calls itself repeatedly, even on subproblems it has **already** solved.
- Tree for $n = 4$. Inside each node is the value of n for the call.



8.1. Rod Cutting

- Let T(n) equals the number of calls to CUT-ROD with **second parameter equal to n**.
- The recurrence looks as follows:

$$T(n) = \begin{cases} 1 & \text{if } n = 0 \\ 1 + \sum_{j=0}^{n-1} T(j) & \text{if } n > 1 \end{cases}$$

- The summation counts all calls when the **second** parameter is $j = n - 1$.
- Solution of the recurrence is $T(n) = 2^n$.

8.1. Rod Cutting

Dynamic-programming solution:

- Do not solve the **same subproblem** repeatedly.
- Arrange to solve each subproblem **just once**.
- **Save** the solution to a subproblem in a table, and **refer back** to the table whenever you revisit the subproblem.
- The goal is: **store, do not recompute**.
  - Time-memory trade-off
- This method can turn an exponential-time solution into a **polynomial-time** solution.

Two basic approaches:

- Top-down with **memoization**.
- Bottom-up.

8.1. Rod Cutting

Top-down with memoization:

- **Memoizing** is remembering what we have computed **previously**.
- Solve the problem **recursively**, but **store** each result in a table.
- In order to find the solution of a subproblem, we fist look into this table.
  - If the answer is there, we will just **use** it.
  - Otherwise, we **compute** the solution to the subproblem and then **store** the solution in the table for future use.

8.1. Rod Cutting

Top-down with memoization:

- **Memoizing** is remembering what we have computed **previously**.
- Solve the problem **recursively**, but **store** each result in a table.
- In order to find the solution of a subproblem, we fist look into this table.
  - If the answer is there, we will just **use** it.
  - Otherwise, we **compute** the solution to the subproblem and then **store** the solution in the table for future use.

Algorithm (MEMOIZED-CUT-ROD(p, n))

```
1       let r[0, …, n] be a new array
2       for (0 ≤ i ≤ n) do
3             r[i] = −∞
4       return MEMOIZED-CUT-ROD-AUX(p, n, r)
```

8.1. Rod Cutting

Top-down with memoization:

- The memoized version of the recursive solution stores the solution to the subproblem of length i in the array entry $r[i]$.

Algorithm (MEMOIZED-CUT-ROD-AUX(p, n, r))

```
1       if (r[n] ≥ 0) then
2           return r[ n ]
3       if ( n = 0 ) then
4           q = 0
5       else
6           q = −∞
7           for (1 ≤ i ≤ n) do
8               q = max(q,p[i]+MEMOIZED-CUT-ROD-AUX(p,n−i,r)
9       r[i] = q
10      return q
```

8.1. Rod Cutting

Bottom-up:

- **Sort** the subproblems by size.
- Solve the **smaller** ones first.
- When solving a subproblem, we then already solved the smaller subproblems needed.

Algorithm (BOTTOM-UP-CUT-ROD(p, n))

```
1       let r[0, …, n] be a new array
2       r[0] = 0
3       for (1 ≤ j ≤ n) do
4           q = −∞
5           for (1 ≤ i ≤ j) do
6               q = max(q, p[i]+r[j−i])
7         r[j] = q
8       return r[n]
```

8.1. Rod Cutting

Running time:

The top-down (memoized) and the bottom-up algorithms run both in $\Theta(n^2)$ time.

Bottom-up:

- We have **doubly nested loops** in BOTTOM-UP-ROD-CUT.
- The number of iterations of the inner loop forms **an arithmetic series**.

Top-down:

- MEMOIZED-CUT-ROD solves each of the subproblems just **once**, and it solves subproblems for size $0, 1, \ldots, n$.
- To solve a subproblem of size n, the for-loop **iterates n times** $\Rightarrow$ over all recursive calls, the total number of iterations forms **an arithmetic series**.

8.1. Rod Cutting

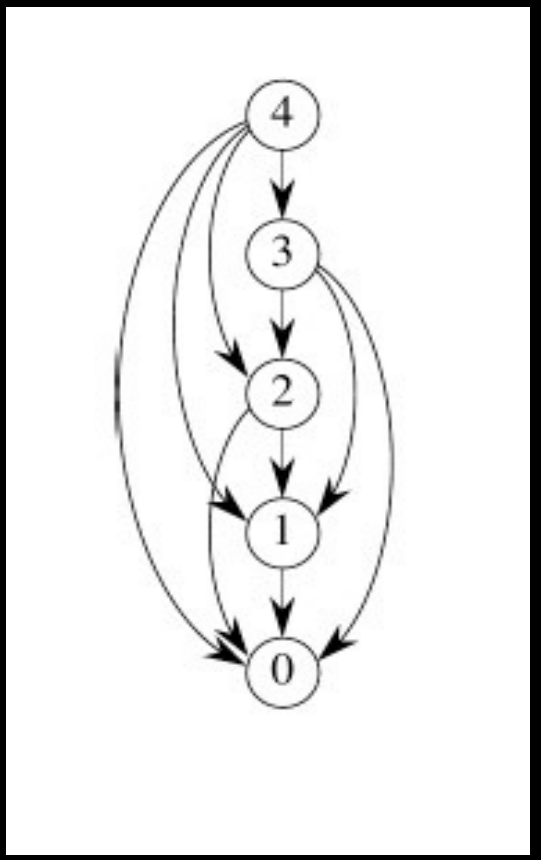
Subproblem graphs:

Question: How do we understand the subproblems that are involved and how they depend on each other?

We use a directed graph:

- **One vertex** for each **distinct subproblem**.
- Has a **direct edge (x, y)** if computing an optimal solution to the subproblem x directly requires knowledge of an optimal solution to the subproblem y.

Example: For rod-cutting problem with $n = 4$.



8.1. Rod Cutting

Subproblem graphs:

- Think of the subproblem graph as a **collapsed version** of the tree of recursive calls.
- All nodes for the same subproblems are collapsed into **a single vertex**.
- All edges go from **parent to child**.
- Subproblem graph can help us to determine the running time.
- Running time is the **sum of times needed to solve each subproblem** (each subproblem solved just once).
  - Time to compute the solution to a subproblem is typically **linear** in the out-degree (number of outgoing edges) of its vertex.
  - Number of subproblems equals **the number of vertices**.
- When these conditions hold, the running time is **linear** in the number of vertices and edges.

8.1. Rod Cutting

Reconstructing a solution:

- Focus not only on the value of an optimal solution, rather than the **choices** that produce an optimal solution.
- We extend the bottom-up approach to also **record the optimal choices**.
- We **save** these choices (first cut) in a separable table **s**.
- A separate algorithm will print the optimal choices based on s.

8.1. Rod Cutting

Algorithm (EXT-BOTTOM-UP-CUT-ROD(p, n))

```
1     let r[0, …, n] and s[0, …, n] be new arrays
2     r[0] = 0
3     for (1 ≤ j ≤ n) do
4         q = −∞
5         for (1 ≤ i ≤ j) do
6           if (q < p[i]+r[j-i]) then
7               q =p[i] + r[j-i]
8               s[j] = i
9         r[j] = q
10    return r an s
```

8.1. Rod Cutting

Reconstructing a solution:

```
Algorithm (PRINT-CUT-ROD-SOLUTION(p, n))
1    (r,s)= EXT-BOTTOM-UP-CUT-ROD(p,n)
2     while (n ≥ 0) do
3          print s[n]
4          n = n – s[n]
```

- Call to PRINT-CUT-ROD-SOLUTION(p, 8)

i	0	1	2	3	4	5	6	7	8
$r[i]$	0	1	5	8	10	13	17	18	22
$s[i]$	0	1	2	3	2	2	6	1	2

- Call to PRINT-CUT-ROD-SOLUTION(p, 6):
  - Prints s=6, sets n=0, and terminates.

i	r_i	optimal solution
1	1	1 (no cuts)
2	5	2 (no cuts)
3	8	3 (no cuts)
4	10	2+2
5	13	2+3
6	17	6 (no cuts)
7	18	1+6 or 2+2+3
8	22	2+6

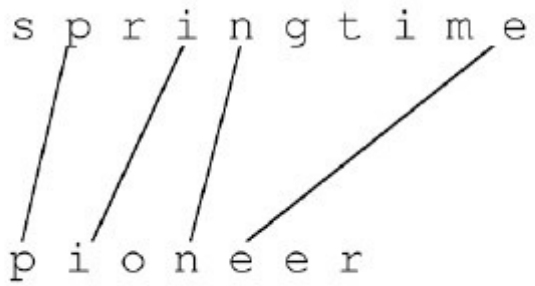
8.2. Longest Common Sequence (LCS)

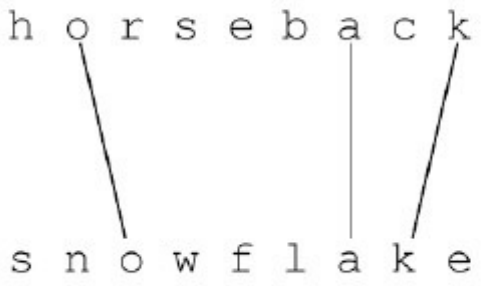
- **Problem:** We are given 2 sequences X=$[x_1, \dots, x_m]$ and Y=$[y_1, \dots, y_n]$. We want to find subsequence with the longest length common to both X and Y.
- **Subsequence:** A subsequence does not have to be consecutive, but it has to be **in order**.

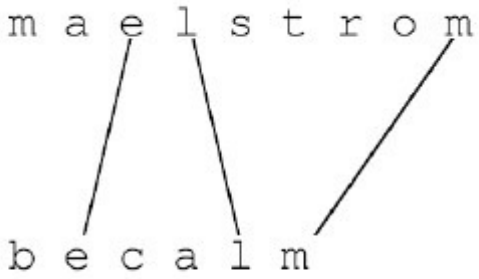
- Search a dictionary for all words that contain we are looking for as a subsequence.
- UNIX command **`grep '.*p.*i.*n,*e.*' dict`** does search the file **`dict`** for words with subsequence pine.
- Then we have to check if that word is actually the LCS.

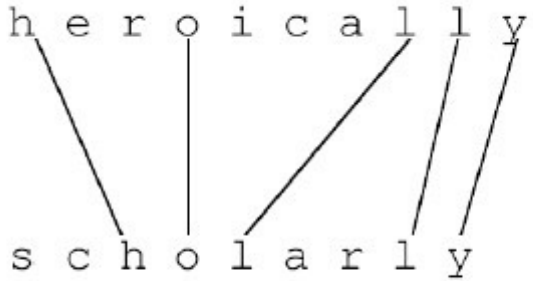
8.2. Longest Common Sequence (LCS)

Example: different types of trees.









$\Rightarrow$ subsequences: "pine", "oak", "elm", "holly"

8.2. Longest Common Sequence (LCS)

Brute-force algorithm:

- For every subsequence of X, we check whether it is a subsequence of Y.
- Running time: $\boldsymbol{\Theta(n2^m)}$.
  - $\boldsymbol{2^m}$ subsequences of X to check.
  - Each subsequence takes $\boldsymbol{\Theta(n)}$ time to check: We scan Y for first letter, from there search the second, and so on.

Optimal substructure:

- Notation:
  - X_i = prefix$[x_1, \ldots, x_i]$
  - Y_i = prefix$[y_1, \ldots, y_i]$

8.2. Longest Common Sequence (LCS)

Theorem:

Let $Z = [z_1, \dots, z_k]$ be any LCS of X and Y.

1. If $\boldsymbol{x_m = y_n}$, then $z_k = x_m = y_n$ and Z_{k-1} is an LCS of X_{m-1} and Y_{n-1}.
2. If $\boldsymbol{x_m \neq y_n}$, then $z_k \neq x_m \Rightarrow Z$ is an LCS of X_{m-1} and Y.
3. If $\boldsymbol{x_m \neq y_n}$, then $z_k \neq y_n \Rightarrow$ Z is an LCS of X and Y_{n-1}.

8.2. Longest Common Sequence (LCS)

Proof of Theorem 1:

- First show that $\boldsymbol{z_k = x_m = y_n}$.
- But we now assume that it is not (contradiction).
- Then make a subsequence $\boldsymbol{Z' = [z_1, \dots, z_k, x_m]}$.
- It is a common sequence of X and Y and has length $\boldsymbol{k + 1} \Rightarrow \boldsymbol{Z'}$ is a longer common sequence than Z .
- $\Rightarrow$ contradicts Z being an LCS.
- Now show $\boldsymbol{Z_{k-1}}$ **is an LCS of** $\boldsymbol{X_{m-1}}$ **and** $\boldsymbol{Y_{n-1}}$:
- It is a common sequence. Now, suppose there exists a common sequence W of X_{m-1} and Y_{n-1} that is longer than $Z_{k-1} \Rightarrow$ the length of $W \geq k$.
- Make subsequence W' by appending x_m to W.
- W' is a common sequence of X and Y, has length $\geq k + 1$
- $\Rightarrow$ Contradicts Z being an LCS.

8.2. Longest Common Sequence (LCS)

Proof of Theorem 1:

- X = [m,a,e,l,s,t,r,o,m]
- Y = [b,e,c,a,l,m]
- Z = [e,l]
- Z’ = [e,l,m]

- Z_{k-1} = [e]
- W = [e,l]
- W’ = [e,l,m]

8.2. Longest Common Sequence (LCS)

Proof of Theorem 2:

- If $z_k \neq x_m$,
- Then Z is a common sequence of X_{m-1} and Y.
- Suppose there exists a subsequence W of X_{m-1} and Y with length $> k$.
- Then W is a common subsequence of X and Y.
- $\Rightarrow$ contradicts Z being an LCS.

Proof of Theorem 3:

- Symmetric to 2.

8.2. Longest Common Sequence (LCS)

Proof of Theorem 2:

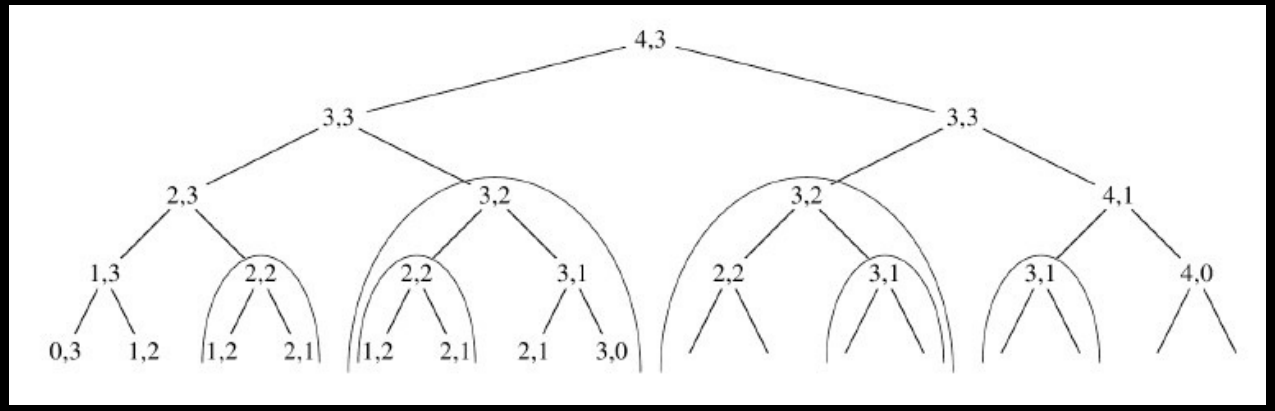
- X = [p,i,o,n,e,e,r], X_{m-1} = [p,i,o,n,e,e]
- Y = [s,p,r,i,n,g,t,i,m,e]
- z_k= e not r
- Z = [p,i,n,e]

8.2. Longest Common Sequence (LCS)

Recursive formulation:

- We define $\boldsymbol{c[i,j]}$ = **length** of LCS of X_i and Y_j.
- We want $\boldsymbol{c[m,n]}$.

$$\boldsymbol{c[i.j] = \begin{cases} 0 & \text{if } i = 0, j = 0 \\ c[i-1, j-1] + 1 & \text{if } i, j > 0 \text{ and } x_i = y_k \\ \max(c[i-1, j], c[i, j-1] & \text{if } i, j > 0 \text{ and } x_i \neq y_j \end{cases}}$$



8.2. Longest Common Sequence (LCS)

Algorithm (LCS-LENGTH(X, Y, m, n))

```
(1)  let b[1...m, 1...n] and c[0...m, 0...n] be new tables
(2)  for (1 ≤ i ≤ m) do
(3)     c[i, 0] = 0
(4)  for (1 ≤ j ≤ n) do
(5)     c[0, j] = 0
(6)  for (1 ≤ i ≤ m) do
(7)     for (1 ≤ j ≤ n) do
(8)        if (x_i == y_j) then
(9)           c[i, j] = c[i − 1, j − 1] + 1
(10)          b[i, j] =↖
(11)       else
(12)          if (c[i − 1, j] ≥ c[i, j − 1]) then
(13)             c[i, j] = c[i − 1, j]
(14)             b[i, j] =↑
(15)          else
(16)             c[i, j] = c[i, j − 1]
(17)             b[i, j] =←
(18)       fi
(19) return c and b
```

8.2. Longest Common Sequence (LCS)

- Initial call is `PRINT-LCS(b,X,m,n)`.
- `b[i,j]` points to the table entry whose subproblem we used in solving the LCS of X_i and Y_j.
- When `b[i,j]=`↖, we have extended LCS by one character. So, the LCS=entries with ↖ in them.

```
Algorithm (PRINT-LCS(b, X, i, j))
1  if (i=0,j=0) then
2     return
3  if (b[i,j]=↖) then
4     PRINT-LCS(b,X,i-1,j-1)
5     print xi
6  else
7     if (b[i,j]=↑) then
8        PRINT-LCS(b,X,i-1,j)
9     else
10       PRINT-LCS(b,X,i,j-1)
```

8.2. Longest Common Sequence (LCS)

Example: LCS for spanking and amputation?

		A	M	P	U	T	A	T	I	O	N
	0	0	0	0	0	0	0	0	0	0	0
S	0										
A	0										
P	0										
A	0										
N	0										
K	0										
I	0										
N	0										
G	0										

Algorithm (LCS-LENGTH(X, Y, m, n))

```
(1)  let b[1...m, 1...n] and c[0...m, 0...n] be new tables
(2)  for (1 ≤ i ≤ m) do
(3)      c[i, 0] = 0
(4)  for (1 ≤ j ≤ n) do
(5)      c[0, j] = 0
```

8.2. Longest Common Sequence (LCS)

Example: Fill table c[0…m, 0…n]

		A	M	P	U	T	A	T	I	O	N
	0←	0←	0←	0←	0←	0←	0←	0←	0←	0←	0←
S	0←	0↑	0↑	0↑	0↑	0↑	0↑	0↑	0↑	0↑	0↑
P	0←	0←	0←	1↖	1←	1←	1←	1←	1←	1←	1←
A	0←	1↑	1←	1↑	1↑	1↑	2↖	2↑	2↑	2↑	2↑
N	0←	1↑	1↑	1↑	1↑	1↑	2↑	2↑	2↑	2↑	3↖
K	0←	1↑	1↑	1↑	1↑	1↑	2↑	2↑	2↑	2↑	3↑
I	0←	1↑	1↑	1↑	1↑	1↑	2↑	2↑	3↖	3←	3↑
N	0←	1↑	1↑	1↑	1↑	1↑	2↑	2←	3↑	3↑	4↖
G	0←	1↑	1↑	1↑	1↑	1↑	2↑	2↑	3↑	3←	4↑

```
for (1 ≤ i ≤ m) do
  for (1 ≤ j ≤ n) do
    if (x_i == y_j) then
      c[i,j] = c[i − 1,j − 1] + 1
      b[i,j] =↖
    else
      if (c[i − 1,j] ≥ c[i,j − 1]) then
        c[i,j] = c[i − 1,j]
        b[i,j] =↑
      else
        c[i,j] = c[i,j − 1]
        b[i,j] =←
    fi
```

8.2. Longest Common Sequence (LCS)

		A	M	P	U	T	A	T	I	O	N
	0←	0←	0←	0←	0←	0←	0←	0←	0←	0←	0←
S	0←	0↑	0↑	0↑	0↑	0↑	0↑	0↑	0↑	0↑	0↑
P	0↑	0←	0←	1↖	1←	1←	1←	1←	1←	1←	1←
A	0↑	1↖	1←	1↑	1↑	1↑	2↖	2←	2↑	2↑	2↑
N	0←	1↑	1↑	1↑	1↑	1↑	2↑	2↑	2↑	2↑	3↖
K	0←	1↑	1↑	1↑	1↑	1↑	2↑	2↑	2↑	2↑	3↑
I	0←	1↑	1↑	1↑	1↑	1↑	2↑	2↑	3↖	3←	3↑
N	0←	1↑	1↑	1↑	1↑	1↑	2↑	2←	3↑	3↑	4↖
G	0←	1↑	1↑	1↑	1↑	1↑	2↑	2↑	3↑	3←	4↑

```
Algorithm (PRINT-LCS(b, X, i, j))
1   if (i=0,j=0) then
2       return
3   if (b[i,j]=↖) then
4       PRINT-LCS(b,X,i-1,j-1)
5       print x_i
6   else
7     if (b[i,j]=↑) then
8         PRINT-LCS(b,X,i-1,j)
9     else
10        PRINT-LCS(b,X,i,j-1)
```

8.3. Sequence Alignment

- Dynamic programming algorithm that performs a **local sequence alignment**.
- Determines similar regions between two nucleotide or protein sequences.
- For us sequences are strings with 4 different characters.
- Sequence $X = x_1, \ldots, x_n$ (along table rows) and $Y = y_1, \ldots, y_m$ (along table columns) are given.

8.3. Sequence Alignment

- Sequence alignment is recursively defined as follows:

$M(i, 0) = 0$, for all $0 \leq i \leq n$

$M(0, j) = 0$, for all $0 \leq j \leq m$

$$M(i,j) = \max \begin{cases} M(i-1, j-1) + 2 & if\ x_i = y_j \\ M(i-1, j-1) - 1 & if\ x_i \neq y_j \\ M(i-1, j) - 1 & if\ x_i \neq y_j, \text{insert } \text{“}-\text{”} \text{ into } Y \\ M(i, j-1) - 1 & if\ x_i \neq y_j, \text{insert } \text{“}-\text{”} \text{ into } X \end{cases}$$

8.3. Sequence Alignment

- The matchings X' and Y' might contain "-" filter characters.
- X' (or Y') might not contain all characters out of X (or Y).

	Example 1	Example 2	Example 3
X	abababda	cacacccbab	cdbaabbdca
X'	a–bababda	ca–cacccbab	c–dba––abbdca
Y'	acbabab–a	cadaadcc–––	cadcacca–bd––
Y	acbababa	bccadaadcc	cadcaccabd
M(n,m)	12	4	5

	Example 4	Example 5
X	caacbdacca	dcacccbbba
X'	caacb–dacc–a	dcacccb–bba
Y'	c––cbcd–ccba	dca–––badba
Y	bccbcdccba	aadcabadba
M(n,m)	9	7

8.3. Sequence Alignment

Algorithm (SMITH-WATERMAN(X, Y))

```
1  New Tables: H[0…n,0…m], P[0…n,0…m]
2  for (0≤i≤n) do
3     H[i][0]=P[i][0]=0
4  for (0≤j≤m) do
5     H[0][j]=P[0][j]=0
6  for (1≤i≤n) do
7     for (1≤j≤m) do
8        if (Xi=Yj) then
9           p1 = H[i-1][j-1]+2
10       else
11          p1 = H[i-1][j-1]-1
12       p2 = H[i-1][j]-1
13       p3 = H[i][j-1]-1
14       H[i][j] = max{p1,p2,p3}
```

SMITH-WATERMAN(X, Y) continue…

```
14       H[i][j] = max{p1,p2,p3}
15       if (H[i][j] = p1) then
16          P[i][j] = ↖
17       else
18          if (H[i][j]= p2) then
19             P[i][j] = ↑
20          else
21             P[i][j] = ←
22 return (H, P)
```

8.3. Sequence Alignment

```
Algorithm (PRINT-SEQ-ALIGN-X(X, P, n, m))
1  if (P[n][m]=↖) then
2      PRINT-SEQ-ALIGN-X(X,p,n-1,m-1)
3      print Xn
4  else
5      if (P[n][m]=←) then
6          PRINT-SEQ-ALIGN-X(X,P,n,m-1)
7          print Xn
8      else
9          PRINT-SEQ-ALIGN-X(X,P,n-1,m)
10         print -
```

- The algorithm PRINT-SEQ-ALIGN-Y(Y, P, n, m) is symmetric to PRINT-SEQ-ALIGN-X.
- PRINT-SEQ-ALIGN-Y outputs the matching sequence Y' for Y instead of the matching sequence X' for X.

8.3. Sequence Alignment

	-	a	b	a	b	a	b	d	a
-	0	0	0	0	0	0	0	0	0
a	0								
c	0								
b	0								
a	0								
b	0								
a	0								
b	0								
a	0								

```
Algorithm (SMITH-WATERMAN(X, Y))
2   for (0≤i≤n) do
3       H[i][0]=P[i][0]=0
4   for (0≤j≤m) do
5       H[0][j]=P[0][j]=0
```

8.3. Sequence Alignment

	-	a	b	a	b	a	b	d	a
-	0	0	0	0	0	0	0	0	0
a	0	2	1	2	1	2	1	0	2
c	0	1	1	1	1	1	1	0	1
b	0	0	3	2	3	2	3	2	1
a	0	2	2	5	4	5	4	3	4
b	0	1	4	4	7	6	7	6	5
a	0	2	3	6	6	9	8	7	8
b	0	1	4	5	8	8	11	10	9
a	0	2	3	6	7	10	10	10	12

Algorithm (SMITH-WATERMAN(X, Y))

```
6   for (1≤i≤n) do
7       for (1≤j≤m) do
8           if (Xi=Yj) then
9               p1 = H[i-1][j-1]+2
10          else
11              p1 = H[i-1][j-1]-1
12          p2 = H[i-1][j]-1
13          p3 = H[i][j-1]-1
14          H[i][j] = max{p1,p2,p3}
```

8.3. Sequence Alignment

	-	a	b	a	b	a	b	d	a
-	0	0	0	0	0	0	0	0	0
a	0	2↖	1↑	2↖	1↑	2↖	1↑	0↑	2↖
c	0	1←	1↖	1←	1↖	1←	1↖	0↖	1←
b	0	0←	3↖	2↑	3↖	2↑	3↖	2↑	1↑
a	0	2↖	2←	5↖	4↑	5↖	4↑	3↑	4↖
b	0	1←	4↖	4←	7↖	6↑	7↖	6↑	5↑
a	0	2↖	3←	6↖	6←	9↖	8↑	7↑	8↖
b	0	1←	4↖	5←	8↖	8←	11↖	10↑	9↑
a	0	2↖	3←	6↖	7←	10↖	10←	10↖	12↖

SMITH-WATERMAN(X, Y) continue…

```
14          H[i][j] = max{p1,p2,p3}
15          if (H[i][j] = p1) then
16            P[i][j] = ↖
17          else
18            if (H[i][j]= p2) then
19              P[i][j] = ↑
20            else
21              P[i][j] = ←
22    return (H, P)
```

8.3. Sequence Alignment

	-	a	b	a	b	a	b	d	a
-	0	0	0	0	0	0	0	0	0
a	0	2↖	1↑	2↖	1↑	2↖	1↑	0↑	2↖
c	0	1←	1↖	1←	1↖	1←	1↖	0↖	1←
b	0	0←	3↖	2↑	3↖	2↑	3↖	2↑	1↑
a	0	2↖	2←	5↖	4↑	5↖	4↑	3↑	4↖
b	0	1←	4↖	4←	7↖	6↑	7↖	6↑	5↑
a	0	2↖	3←	6↖	6←	9↖	8↑	7↑	8↖
b	0	1←	4↖	5←	8↖	8←	11↖	10↑	9↑
a	0	2↖	3←	6↖	7←	10↖	10←	10↖	12↖

```
Algorithm (PRINT-SEQ-ALIGN-X(X, P, n, m))
1   if (P[n][m]=↖) then
2       PRINT-SEQ-ALIGN-X(X,p,n-1,m-1)
3       print Xn
4   else
5       if (P[n][m]=←) then
6         PRINT-SEQ-ALIGN-X(X,P,n,m-1)
7         print Xn
8       else
9         PRINT-SEQ-ALIGN-X(X,P,n-1,m)
10        print -
```

PRINT-SEQ-ALIGN-X(X,P,n,m)prints

- X′ = acbabab-a

PRINT-SEQ-ALIGN-Y(Y,P,n,m)prints

- We print - if P[i][j]=← and print Y_n if P[i][j]=↑.
- Y′ = a-bababda

8.3. Optimal Binary Search Trees

- We are given a sequence $K = [k_1, \dots, k_n]$ of n distinct keys, sorted $(k_1 < \cdots < k_n)$.
- We want to build a BST from the keys.
- For each key k_i, we have probability p_i that a search is for k_i.
- We want a BST with minimum expected search cost.
- The actual cost is = number of items examined.

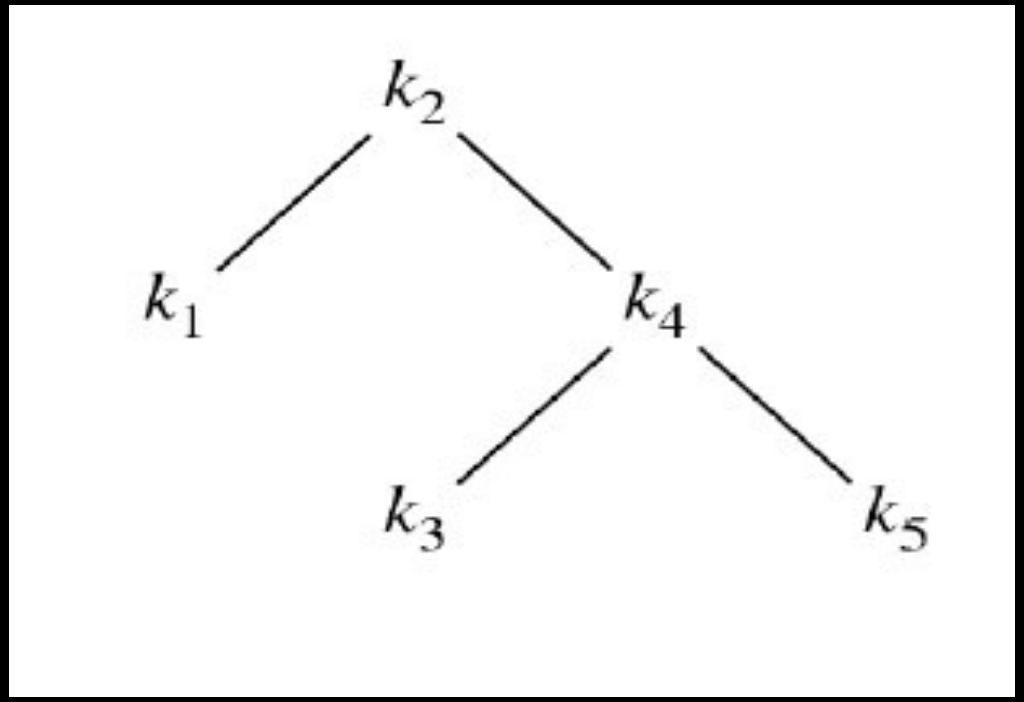
$$E[search\ cost\ in\ T] = \sum_{i=1}^{n} (d_T(k_i) + 1) \cdot p_i = 1 + \sum_{i=1}^{n} d_T(k_i) \cdot p_i$$

Depth

8.3. Optimal Binary Search Trees

i	1	2	3	4	5
p_i	0.25	0.2	0.05	0.2	0.3

$$E[search\ cost\ in\ T] = \sum_{i=1}^{n} d_T(k_i) \cdot p_i$$

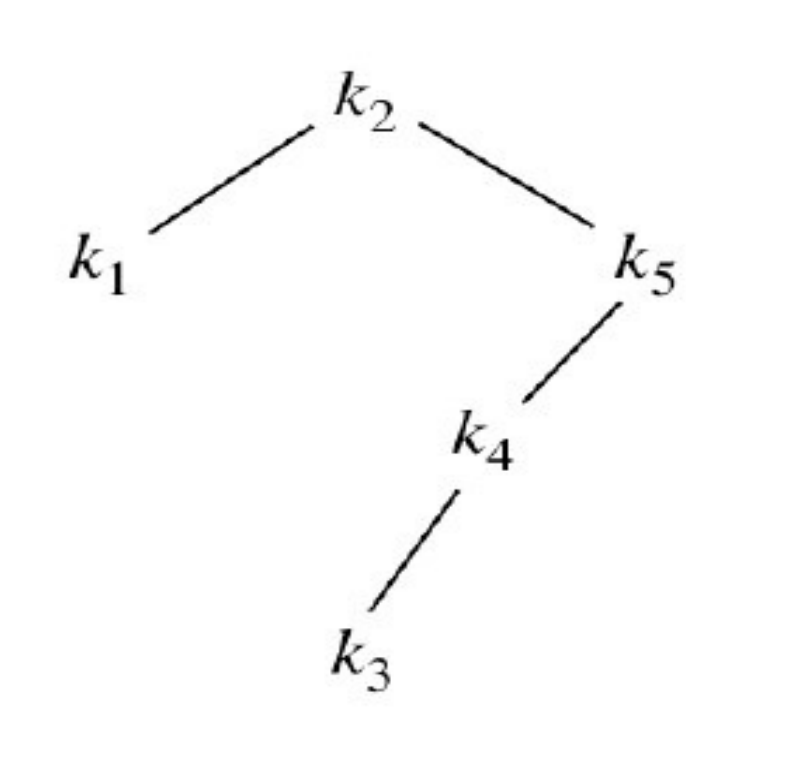


i	$d_T(k_i)$	$d_T(k_i) \cdot p_i$
1	1	0.25
2	0	0
3	2	0.1
4	1	0.2
5	2	0.6
	$\sum d_T(k_i) \cdot p_i$	1.15
	E=	2.15

8.3. Optimal Binary Search Trees

i	1	2	3	4	5
p_i	0.25	0.2	0.05	0.2	0.3

$$E[search\ cost\ in\ T] = \sum_{i=1}^{n} d_T(k_i) \cdot p_i$$



i	$d_T(k_i)$	$d_T(k_i) \cdot p_i$
1	1	0.25
2	0	0
3	3	0.15
4	2	0.4
5	1	0.3
	$\sum d_T(k_i) \cdot p_i$	1.10
	E=	2.10

8.3. Optimal Binary Search Trees

Observation:

- The optimal BST might not have the smallest height.
- The optimal BST might not have the key with the highest probability at the root.

Build by exhaustive checking?

1. We construct each n-node BST.
2. We then put in the keys for each of the BST's.
3. We then compute the expected search cost.

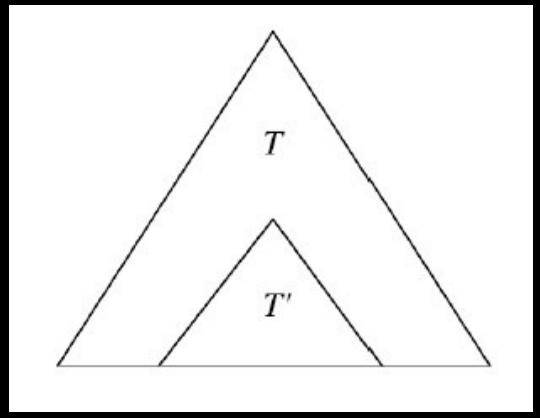
Problem:

- There are $\Omega\left(\frac{4^n}{n^{\frac{3}{2}}}\right)$ different BST's with n nodes.

8.3. Optimal Binary Search Trees

Optimal substructure:

- We consider any subtree of a BST. It contains keys in a contiguous range $k_i, \ldots, k_j$, for some $1 \leq i \leq j \leq n$.

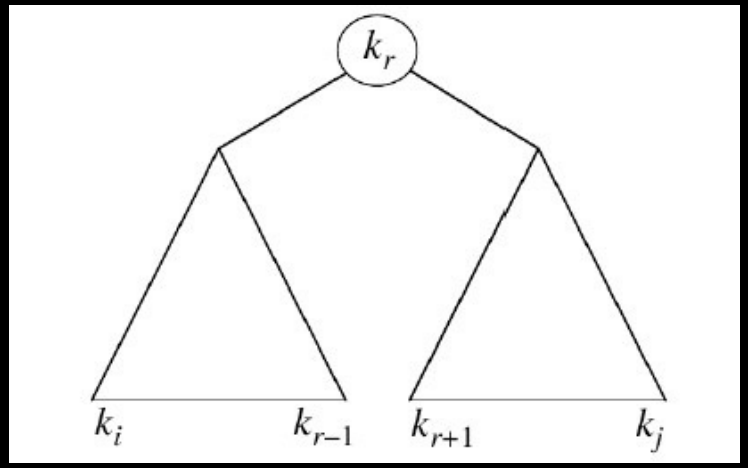


- If the BST T is an optimal BST and T contains the subtree T' with keys $k_i, \ldots, k_j$, then T' must be an optimal BST for keys $k_i, \ldots, k_j$.

8.3. Optimal Binary Search Trees

Solution:

- We use the optimal substructure to construct an optimal solution to the problem from optimal solutions to subproblem.



- If the BST T is an optimal BST and T contains the subtree T' with keys $k_i, \ldots, k_j$, then T' must be an optimal BST for keys $k_i, \ldots, k_j$.

8.3. Optimal Binary Search Trees

Solution (continued)

- We are guaranteed to find an optimal BST for $k_i, \ldots, k_j$ if the following two conditions are satisfied.
  - We examine all candidate roots k_r for $i \leq r \leq j$.
  - We determine all optimal BST's containing $k_i, \ldots, k_{r-1}$ (left subtree) and containing $k_{r+1}, \ldots, k_j$ (right subtree).

8.3. Optimal Binary Search Trees

Recursive Solution:

- Subproblem domain:
  - Find an optimal BST for $k_i, \ldots, k_j$, where $i \geq 1, j \leq n$, and $j \geq i - 1$.
  - The tree is empty when $j = i - 1$.
- We define $e[i, j]$ = expected search cost of the optimal BST for $k_i, \ldots, k_j$.
- If $j = i - 1$, then $e[i, j] = 0$.
- If $j \geq i$,
  - We select a root k_r, for some $i \leq r \leq j$.
  - We make an optimal BST with $k_i, \ldots, k_{r-1}$ as left subtree.
  - We make an optimal BST with $k_{r+1}, \ldots, k_j$ as right subtree.
  - Note: if $r = i$, the left subtree is $k_i, \ldots, k_{i-1}$. Similar for $r = j$. The right subtree is $k_{j+1}, \ldots, k_j$.

8.3. Optimal Binary Search Trees

Recursive Solution (continue):

- When a subtree becomes a subtree of a node:
  - The depth of every node is in the subtree increases by 1.
  - The expected search cost increases by

$$w(i,j) = \sum_{l=i}^{j} p_l .$$

- If k_r is the root of an optimal BST for $k_i, \ldots, k_j$:

$$e[i,j] = p_r + e[i,r-1] + w(i,r-1) + e[r+1,j] + w(r+1,j).$$

Root Left subtree Right subtree

8.3. Optimal Binary Search Trees

Recursive Solution (continue):

- But we have

$$w(i,j) = w(i,r-1) + p_r + w(r+1,i). \qquad \text{Eq. 8-1}$$

- Therefore, we get

$$e[i,j] = e[i,r-1] + e[r+1,j] + w(i,j).$$

- This assumes that we already known which of the keys we use as k_r.
- We need to try all possible candidates, and pick the best one:

$$e[i,j] = \begin{cases} 0 & \text{if } j = i-1 \\ \min\limits_{i \le r \le j} \{e[i,r-1] + e[r+1,j] + w(i,j)\} & \text{if } i \le j \end{cases} \qquad \text{Eq. 8-2}$$

8.3. Optimal Binary Search Trees

Computing an optimal solution

- We will store the values in a table as we did before, $e[1 \ldots n+1, 0 \ldots n]$ can store $e[n+1, n]$ and $e[1,0]$.
  - We will use only entries $e[i, j]$, where $j \geq i - 1$.
  - We will also compute $R[i, j]$ =root of the subtree with keys $k_i, \ldots, k_j$, for $1 \leq i \leq j \leq n$.
- We need $w[1 \ldots n+1, 0 \ldots n]$ table to avoid $w(i, j)$ re-computation: there will be additional $\Theta(j - i)$ each time.
  - $w[i, i-1] = 0$ for $1 \leq i \leq n$.
  - $w[i, j] = w[i, j-1] + p_j$ for $1 \leq i \leq j \leq n$.
  - Computes all $\Theta(n^2)$ values in $O(1)$ time each.

8.3. Optimal Binary Search Trees

Algorithm

- We have a first for-loop that initialize the entries in e and w for subtrees with 0 keys.
- We have a main for-loop that:
  - The iterations for l works on subtrees with l keys.
  - Idea: we compute in the order of the subtree sizes, smaller (1 key) to larger (n keys).

8.3. Optimal Binary Search Trees

Algorithm

```
Algorithm (OPTIMAL-BST(p, q, n))
1  e[1…n+1,0…n], w[1…n+1,0…n], R[1…n,1…n]
2  for (i ≤ i ≤ n+1) do
3     e[i,i-1]=0,w[i,i-1]=0
4  for (1≤ l ≤ n) do
5     for (1 ≤ i ≤ n-l+1)do
6        j = i+l-1
7        e[i,j] = ∞
8        w[i,j] = w[i,j-1]+p_j                    //Eq. 8-1
9        for (i≤ r ≤ j) do
10          t = e[i,r-1]+e[r+1,j]+w[i,j]    //Eq. 8-2
11          if (t < e[i,j]) then
12            R[i,j] = r
13 return e and root
```

8.3. Optimal Binary Search Trees

i	1	2	3	4	5
p_i	0.25	0.2	0.05	0.2	0.3

```
2  for (i≤i≤n+1) do
3      e[i,i-1]=0,w[i,i-1]=0
```

e	0	1	2	3	4	5
1	0					
2		0				
3			0			
4				0		
5					0	
6						0

w	0	1	2	3	4	5
1	0					
2		0				
3			0			
4				0		
5					0	
6						0

R	1	2	3	4	5
1	1				
2		2			
3			3		
4				4	
5					5

8.3. Optimal Binary Search Trees

i	1	2	3	4	5
p_i	0.25	0.2	0.05	0.2	0.3

```
4   for (1≤ l ≤ n) do
5    for (1≤i≤n-l+1)do
6      j = i+l-1
7      e[i,j] = ∞
8      w[i,j] = w[i,j-1]+pj
9      for (i≤ r ≤ j) do
10       t = e[i,r-1]+e[r+1,j]+w[i,j]
11       if (t < e[i,j]) then
12         R[i,j] = r
```

w	0	1	2	3	4	5
1	0	0.25				
2		0	0.2			
3			0	0.05		
4				0	0.2	
5					0	0.3
6						0

R	1	2	3	4	5
1	1				
2		2			
3			3		
4				4	
5					5

e	0	1	2	3	4	5
1	0	0.25				
2		0	0.2			
3			0	0.05		
4				0	0.2	
5					0	0.3
6						0

8.3. Optimal Binary Search Trees

i	1	2	3	4	5
p_i	0.25	0.2	0.05	0.2	0.3

```
4   for (1≤ l ≤ n) do
5    for (1≤i≤n-l+1)do
6      j = i+l-1
7      e[i,j] = ∞
8      w[i,j] = w[i,j-1]+pj
9      for (i≤ r ≤ j) do
10       t = e[i,r-1]+e[r+1,j]+w[i,j]
11       if (t < e[i,j]) then
12         R[i,j] = r
```

w	0	1	2	3	4	5
1	0	0.25	0.45			
2		0	0.2			
3			0	0.05		
4				0	0.2	
5					0	0.3
6						0

For i=1, j=2:

- w[1,2]=w[1,1]+p_2=0.25+0.2=0.45
- r=1: e[1,2]=e[1,0]+e[2,2]+w[1,2]=0+0.2+0.45
- r=2: e[1,2]=e[1,1]+e[3,2]+w[1,2]=0.25+0+0.45

e	0	1	2	3	4	5
1	0	0.25	0.65			
2		0	0.2			
3			0	0.05		
4				0	0.2	
5					0	0.3
6						0

R	1	2	3	4	5
1	1	1			
2		2			
3			3		
4				4	
5					5

8.3. Optimal Binary Search Trees

i	1	2	3	4	5
p_i	0.25	0.2	0.05	0.2	0.3

```
4   for (1≤ l ≤ n) do
5    for (1≤i≤n-l+1)do
6      j = i+l-1
7      e[i,j] = ∞
8      w[i,j] = w[i,j-1]+pj
9      for (i≤ r ≤ j) do
10       t = e[i,r-1]+e[r+1,j]+w[i,j]
11       if (t < e[i,j]) then
12         R[i,j] = r
```

w	0	1	2	3	4	5
1	0	0.25	0.45			
2		0	0.2	0.25		
3			0	0.05		
4				0	0.2	
5					0	0.3
6						0

For i=2, j=3:

- w[2,3]=w[2,2]+p_3=0.2+0.05=0.25
- r=2: e[2,3]=e[2,1]+e[3,3]+w[2,3]=0+0.05+0.25
- r=3: e[2,3]=e[2,2]+e[4,3]+w[2,3]=0.2+0+0.25

e	0	1	2	3	4	5
1	0	0.25	0.65			
2		0	0.2	0.3		
3			0	0.05		
4				0	0.2	
5					0	0.3
6						0

R	1	2	3	4	5
1	1	1			
2		2	2		
3			3		
4				4	
5					5

8.3. Optimal Binary Search Trees

i	1	2	3	4	5
p_i	0.25	0.2	0.05	0.2	0.3

```
4   for (1≤ l ≤ n) do
5    for (1≤i≤n-l+1)do
6      j = i+l-1
7      e[i,j] = ∞
8      w[i,j] = w[i,j-1]+pj
9      for (i≤ r ≤ j) do
10       t = e[i,r-1]+e[r+1,j]+w[i,j]
11       if (t < e[i,j]) then
12         R[i,j] = r
```

For i=1, j=3:

- w[1,3]=w[1,2]+p_3=0.42+0.05=0.5
- r=1: e[1,3]=e[1,0]+e[2,3]+w[1,3]=0+0.3+0.5
- r=2: e[1,3]=e[1,1]+e[3,3]+w[1,3]=0.25+0.05+0.5
- r=3: e[1,3]=e[1,2]+e[4,3]+w[1,3]=0.65+0+0.5

w	0	1	2	3	4	5
1	0	0.25	0.45	0.5		
2		0	0.2	0.25		
3			0	0.05		
4				0	0.2	
5					0	0.3
6						0

e	0	1	2	3	4	5
1	0	0.25	0.65	0.8		
2		0	0.2	0.3		
3			0	0.05		
4				0	0.2	
5					0	0.3
6						0

R	1	2	3	4	5
1	1	1	1		
2		2	2		
3			3		
4				4	
5					5

8.3. Optimal Binary Search Trees

i	1	2	3	4	5
p_i	0.25	0.2	0.05	0.2	0.3

w	0	1	2	3	4	5
1	0	0.25	0.45	0.5	0.7	1.0
2		0	0.2	0.25	0.45	0.75
3			0	0.05	0.25	0.55
4				0	0.2	0.5
5					0	0.3
6						0

e	0	1	2	3	4	5
1	0	0.25	0.65	0.8	1.25	2.10
2		0	0.2	0.3	0.75	1.35
3			0	0.05	0.3	0.85
4				0	0.2	0.7
5					0	0.3
6						0

R	1	2	3	4	5
1	1	1	1	2	2
2		2	2	2	4
3			3	4	5
4				4	5
5					5

8.3. Optimal Binary Search Trees

Time:

- We have 3 (deep) nested for loop, each loop index takes values $\leq n \Rightarrow O(n^3)$.
- We can also show $\Omega(n^3) \Rightarrow \Theta(n^3)$.

8.3. Optimal Binary Search Trees

Construct an optimal solution

```
Algorithm (CONSTRUCT-OPTIMAL-BST(R))
1  r = R[1,n]
2  print r "is the root"
3  CONSTRUCT-OPT-SUBTREE(1,r-1,r,"left",R)
4  CONSTRUCT-OPT-SUBTREE(r+1,n,r,"right",R)
```

```
Algorithm (CONSTRUCT-OPT-SUBTREE(i,j,r,dir,R))
1  if (i ≤ j) then
2     t = R[i,j]
3     print t "is " dir "child of" r
4     CONSTRUCT-OPT-SUBTREE(1,t-1,t,"left",R)
5     CONSTRUCT-OPT-SUBTREE(t+1,j,t,"right",R)
```

8.3. Optimal Binary Search Trees

Construct an optimal solution

```
Algorithm (CONSTRUCT-OPTIMAL-BST(R))
1  r = R[1,n]
2  print r "is the root"
3  CONSTRUCT-OPT-SUBTREE(1,r-1,r,"left",R)
4  CONSTRUCT-OPT-SUBTREE(r+1,n,r,"right",R)
```

```
Algorithm (CONSTRUCT-OPT-SUBTREE(i,j,r,dir,R))
1  if (i ≤ j) then
2     t = R[i,j]
3     print t "is " dir "child of" r
4     CONSTRUCT-OPT-SUBTREE(1,t-1,t,"left",R)
5     CONSTRUCT-OPT-SUBTREE(t+1,j,t,"right",R)
```

R	1	2	3	4	5
1	1	1	1	2	2
2		2	2	2	4
3			3	4	5
4				4	5
5					5

2 is the root
1 is left child of 2
5 is right child of 2
4 is left child of 5
3 is left child of 4

8.4. Elements of dynamic programming

Optimal substructure:

- We show that a solution to a problem consists of making a choice, which leaves us with one or more subproblems to solve.
- Suppose that you are given this last choice that leads to an optimal solution.
  - Given this choice, we determine which subproblems arise and how to characterize the resulting space of subproblems.
  - We show that the solutions of the subproblems used within the optimal solution must themselves be optimal (usually cut-and-paste).
    - Cut the non-optimal solution from the subproblem.
    - Paste in an optimal solution.
    - Get a better solution to the original problem.

8.4. Elements of dynamic programming

Space of subproblems:

- We need to ensure that we consider a wide enough range of choices and subproblems.
- We try all the choices, solve all the subproblems resulting from each of the choices.
- We pick the choice whose solution, along with its subproblem solutions, is best.
- How do we characterize the space of subproblems?
  - We keep the space as simple as possible.
  - We expand it as necessary.

8.4. Elements of dynamic programming

Examples:

- Rod Cutting
  - The space of subproblems was rods of length $n - i$, for $1 \leq i \leq n$.
  - There is no need to try a more general space of subproblems.

8.4. Elements of dynamic programming

Examples:

- Optimal BSTs
  - Suppose we had tried to constrain the space of subproblems to subtrees with keys $k_1, \ldots, k_j$.
  - An optimal BST would have root k_r, for some $1 \leq r \leq j$.
  - We get the subproblems $k_1, \ldots, k_{r-1}$ and $k_{r+1}, \ldots, k_j$.
  - Then, if we cannot guarantee that $r = j$ and therefore an empty subproblem $k_{r+1}, \ldots, k_j$, this non-empty subproblem is then not of the form $k_1, \ldots, k_j$.
  - It is therefore necessary to allow the subproblems to vary at both ends, i.e., allow both i and j to vary.

8.4. Elements of dynamic programming

Optimal substructure varies across problem domains:

1. How many subproblems are used in an optimal solution?
2. How many choices in determining which subproblem(s) to use?

- Rod cutting:
  1) 1 subproblem (of size n-i)
  2) n choices
- LCS:
  1) 1 subproblem
  2) We have either
     - 1 choice, if $x_i = y_j$, LCS of X_{i-1} and Y_{j-1}
     - 2 choices, if $x_i \neq y_j$, LCS of X_{i-1} and Y, and LCS of X and Y_{j-1}

8.4. Elements of dynamic programming

Optimal substructure varies across problem domains:

- Optimal BST:
  1) 2 subproblem ($k_i, \dots, k_{r-1}$ and $k_{r+1}, \dots, k_j$)
  2) $\mathrm{j} - \mathrm{i} + 1$ choices for k_r in $k_i, \dots, k_j$. Once we determine optimal solutions to subproblems, we choose among the $\mathrm{j} - \mathrm{i} + 1$ candidates for k_r.

8.4. Elements of dynamic programming

Running Time:

Informally depends on (# of subproblems overall) x (# of choices)

- Rod Cutting:
  - $\Theta(n)$ subproblems, $\leq n$ choices for each $\Rightarrow O(n^2)$ running time.
- LCS:
  - $\Theta(nm)$ subproblems, ≤ 2 choices for each $\Rightarrow \Theta(nm)$ running time.
- Optimal BST:
  - $\Theta(n^2)$ subproblems, $O(n)$ choices for each $\Rightarrow O(n^3)$ running time.

8.4. Elements of dynamic programming

Running Time:

Informally depends on (# of subproblems overall) x (# of choices)

- Rod Cutting:
  - $\Theta(n)$ subproblems, $\leq n$ choices for each $\Rightarrow O(n^2)$ running time.
- LCS:
  - $\Theta(nm)$ subproblems, ≤ 2 choices for each $\Rightarrow \Theta(nm)$ running time.
- Optimal BST:
  - $\Theta(n^2)$ subproblems, $O(n)$ choices for each $\Rightarrow O(n^3)$ running time.

8.4. Elements of dynamic programming

Top-Down Approach:

- Memoization
  - Store, do not recompute.
  - Make a table indexed by subproblem.
  - When solving a subproblem,
    - We look-up in the table.
    - If the answer is there, we use it.
    - Otherwise, we compute the answer, then store it.

Bottom-up Approach:

- We determine in what order we would want to access the table, and fill it in that way.

Greedy Algorithms

9.1. Greedy Algorithm

Greedy Algorithm:

- Similar to dynamic programming.
- Used for optimization problems.

Idea:

- When we choose, we make the one that **looks the best** at the current moment.
- We make a **locally optimal choice** in the hope of getting a globally optimal solution.
  - Greedy algorithms do not always yield an optimal solution.
  - We look at general characteristics when greedy algorithms give an optimal solution.

9.2. Activity selection

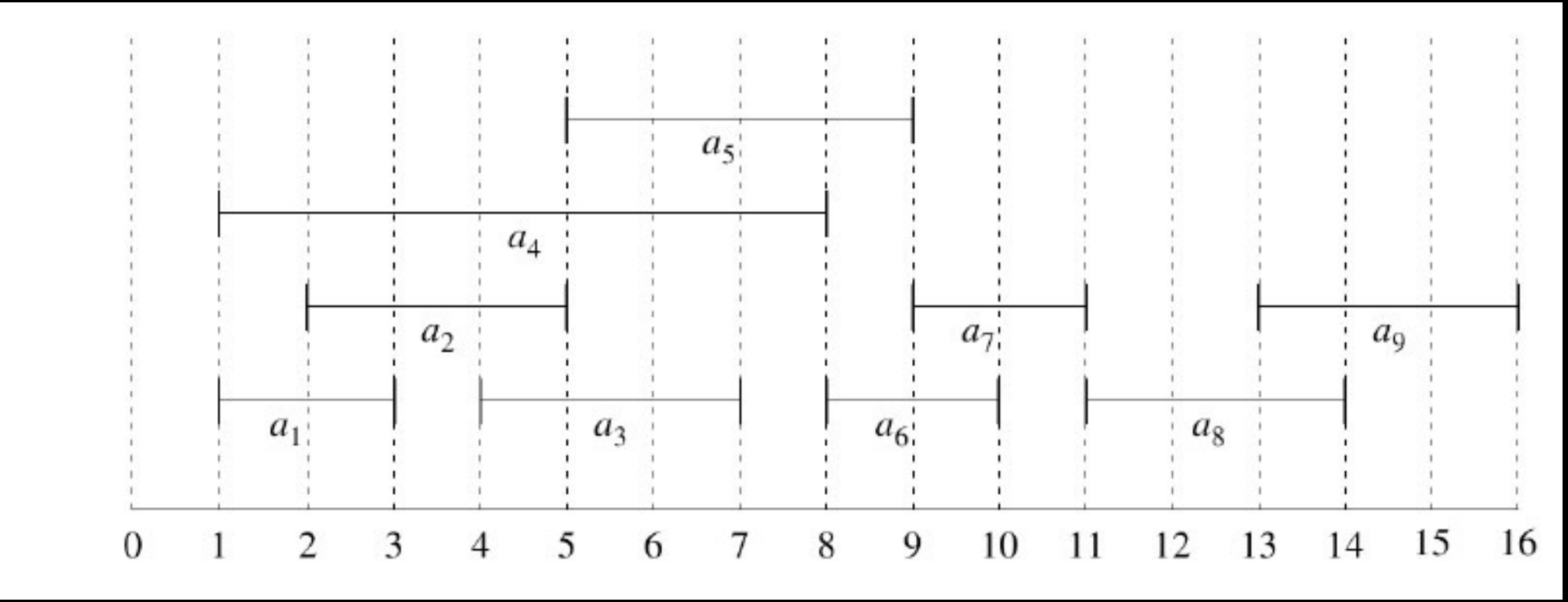
- Suppose there are $\boldsymbol{n}$ **activities** that require exclusive use of a common resource, e.g., scheduling the use of a classroom.
- Let the activity set is $\boldsymbol{S = \{a_1, \ldots, a_n\}}$.
- Each activity, a_i, needs resources during the period $\boldsymbol{[s_i, f_i)}$ (half-open interval), where s_i = starting time and f_i = finishing time.
- Assume that activities are sorted by the finishing time:

$$f_1 \leq f_2 \leq \cdots \leq f_{n-1} \leq f_n.$$

- Goal: To select the **largest** possible set of non-overlapping (**mutually compatible**) activities.
- Note: We can have many other activities:
  - Schedule room for the longest time.
  - Maximize income rental fees.

9.2. Activity selection

i	1	2	3	4	5	6	7	8	9
s_i	1	2	4	1	5	8	9	11	13
f_i	3	5	7	8	9	10	11	14	16



9.2. Activity selection

Optimal substructure of activity selection:

- Let $S_{ij} = \{a_k \in S: f_i \leq s_k < f_k \leq s_j\}$.
  - a_k starts after a_i finishes and finishes before a_j starts.
- Activities in S_{ij} are compatible with
  - all activities finish by f_i and
  - all activities start no earlier than s_j.
- Let A_{ij} be a maximum-size set of mutually compatible activities in S_{ij}.
- Let $a_k \in A_{ij}$ be some activity in A_{ij}. Then we have two subproblems:
  - Find mutually compatible activities in S_{ik}
  - Find mutually compatible activities in S_{kj}

9.2. Activity selection

- Let $A_{ik} = A_{ij} \cap S_{ik}$ = activities in A_{ij} that finish before a_k starts and $A_{kj} = A_{ij} \cap S_{kj}$ = activities in A_{ij} that finish after a_k starts.
- Then $A_{ij} = A_{ik} \cup \{a_k\} \cup A_{kj} \Rightarrow |A_{ij}| = |A_{ik}| + |A_{kj}| + 1$.
- The optimal solution A_{ij} must include optimal solutions for two subproblems for S_{ik} and S_{kj}.
  - Suppose we can find a set A'_{kj} of mutually compatible activities in S_{kj}, where $|A'_{kj}| > |A_{kj}|$.
  - Then the size of mutually compatible activities would be $|A_{ik}| + |A'_{kj}| + 1 > |A_{ik}| + |A_{kj}| + 1$.

9.2. Activity selection

A recursive solution:

- Dynamic programming approach: It is possible since the optimal solution A_{ij} must include optimal solution to the subproblem for S_{ik} and S_{kj}.
  - Create a table $c[i, j]$ to store the size of optimal solution for S_{ij}.
  - Since $|A_{ij}| = |A_{ik}| + |A_{kj}| + 1$, we can fille the table using
$$c[i, j] = c[i, k] + c[k, j] + 1.$$
  - We do not know which activity a_k to choose.

$$c[i, j] = \begin{cases} 0 & \text{if } S_{ij} = \emptyset \\ \max\limits_{a_k \in S_{ij}} \{c[i, k] + c[k, j] + 1\} & \text{if } S_{ij} \neq \emptyset \end{cases}$$

9.2. Activity selection

Greedy approach:

- Choose an activity to add to the optimal solution before solving the subproblems.
- We can consider only the greedy choice:
  - The activity that leaves the resource available for as many other activities as possible.
- Question: Which activity leaves the resource available for the most other activities?
- Answer: The first activity to finish - we choose any if we have more than one.

9.2. Activity selection

Greedy approach:

- We choose activity a_1 since activities are sorted by finishing time.
- This leaves us with only one subproblem to solve:
  - Find a maximum size set of mutually compatible activities that start after a_1 finishes, because no a_i has $f_i < f_1 \Rightarrow$ no $f_i \leq s_1$.
- The one subproblem we need to solve is the set of activities that start after a_k finishes

$$S_k = \{a_i \in S : s_i \geq f_k\}$$

- If a_1 is our greedy choice, then S_1 remains as only subproblem to solve – S_1 is the set of activities but a subproblem at the same time.

9.2. Activity selection

Greedy approach:

- If a_1 is a part of optimal solution, the optimal solution to the problem has a_1 and optimal solution of S_1.
- How do we know if a_1 is always a part of some optimal solution?

9.2. Activity selection

Greedy approach:

- If S_k is not $\emptyset$ and a_m has the earliest finishing time in S_k, then a_m is included in some optimal solution.
  - Suppose A_k be an optimal solution to S_k, and let a_j have the earliest finishing time of any activity in A_k.
    - If $a_j = a_m$, then A_k is the solution.
    - Otherwise, we claim A'_k is the new solution that is $A'_k = A_k - \{a_j\} \cup \{a_m\}$.
      - If all activities in A'_k are disjoint, a_j is first activity in A'_k to finish and $f_m \leq f_j$.
    - Since $|A'_k| = |A_k|$, A'_k is an optimal solution to S_k that includes a_m!

9.2. Activity selection

Greedy approach:

- From the argument of the sub-solution, we find that we do not need the full power of dynamic programming.
- We do not need to work bottom-up fashion either.
- Instead, we can repeatedly choose the activity that finishes first and keep the activities that are compatible with that one.
- We can stop when no activities remain.
- In top-down fashion,
  - We can make choice, then solve a subproblem.
  - In other words, we do not have to solve subproblems before making a choice.

9.2. Activity selection

Recursive greedy algorithm:

- Let the array s and f have starting and finishing times.
- Assume that array f is already sorted in monotonically increasing order.
- In each recursion, we add a fictious activity a_0 with $f_0 = 0$, so that $S_0 = S$, the entire set of activities.

9.2. Activity selection

Recursive greedy algorithm:

Current Subproblem index

Original number of activities

```
Algorithm (REC-ACTIVITY-SELECTOR(s, f, k, n))
1  m = k + 1
2  while (m ≤ n and s[m] < f[k]) do
3     m = m + 1              //find the next compatible m
4  if (m ≤ n) then
5     return {a_m} ∪ REC-ACTIVITY-SELECTOR(s,f,m,n)
6  else
7     return ∅
```

Running Time:

- $\Theta(n)$: each activity is examined exactly once.

9.2. Activity selection

i	1	2	3	4	5	6	7	8	9
s_i	1	2	4	1	5	8	9	11	13
f_i	3	5	7	8	9	10	11	14	16

Algorithm (REC-ACTIVITY-SELECTOR(s, f, k, n))

```
1   m = k + 1
2   while (m ≤ n and s[m] < f[k]) do
3     m = m + 1
4   if (m ≤ n) then
5     return {a_m} ∪ REC-ACTIVITY-
    SELECTOR(s,f,m,n)
6   else
7     return ∅
```

RAC(a,s,f,0,n)

1. m=1; $\{a_1\}$URAS(s,f,1,n)
2. k=1,m=3,$\{a_1,a_3\}$ URAS(s,f,3,n)
3. k=3,m=6, $\{a_1,a_3,a_6\}$ URAS(s,f,6,n)
4. k=6,m=9, $\{a_1,a_3,a_6,a_9\}$ URAS(s,f,9,n)
5. k=9,m=10, ∅

9.2. Activity selection

Loop Invariants:

- Initialization: we have a, s, and f and f is in monotonically increasing order.
- Maintenance: The while loop checks activities, $a_{k+1}, a_{k+2}, \ldots, a_n$ until it finds the activity a_m that is compatible with a_k - that is $s_m \geq f_k$.
- Termination:
  - If a_m is found, it terminates. Then, the algorithm recursively solve S_m and returns the solution with a_m included.
  - If a_m is not found, then an empty set if returned.

9.2. Activity selection

Iterative greedy algorithm:

- We can also solve the problem iteratively.

```
Algorithm (GREEDY-ACTIVITY-SELECTOR(s, f))
1  n = s.length
2  A = {a_1}
3  k = 1
4  for (2 ≤ m ≤ n) do
5     if (s[m] ≥ f[k]) then
6        A = A ∪ a_m
7        k = m
8  return A
```

Running Time: $\Theta(n)$

9.2. Activity selection

Iterative greedy algorithm:

i	1	2	3	4	5	6	7	8	9
s_i	1	2	4	1	5	8	9	11	13
f_i	3	5	7	8	9	10	11	14	16

Algorithm (GREEDY-ACTIVITY-SELECTOR(s, f))

```
1  n = s.length
2  A = {a1}
3  k = 1
4  for (2 ≤ m ≤ n) do
5     if (s[m] ≥ f[k]) then
6        A = A ∪ am
7        k = m
8  return A
```

RAC(s, f)

0. A = $\{a_1\}$
1. k=1, m=3, A = $\{a_1, a_3\}$, k=3
2. k=3, m=4, A = $\{a_1, a_3\}$, k=3
3. k=3, m=5, A = $\{a_1, a_3\}$, k=3
4. k=3, m=6, A = $\{a_1, a_3, a_6\}$,k=6
9. k=6, m=9, A = $\{a_1, a_3, a_6, a_9\}$, k=9
10. k=9, m=10, A = $\{a_1, a_3, a_6, a_9\}$

9.2. Activity selection

Loop Invariants:

- Initialization: we have a_1 in our solution A and starts the for-loop.
- Maintenance: The solution set A appends a_m if s[m]≧f[k] and k becomes m.
- Termination:
  - When m=n+1 and terminates.

9.3. Greedy strategy

Greedy strategy:

- We choose the one that seems the bets at that moment.
  - We determined the optimal substructure.
  - We developed a recursive solution.
  - If we made a greedy choice, we saw that only one subproblem remains.
  - We proved that it is always safe to make the greedy choice.
  - We developed a recursive greedy algorithm and iterative greedy algorithm.

9.3. Greedy strategy

- The algorithms look like dynamic programming at first.
  - We started by defining subproblems S_{ij}.
- But the greedy choice allow us to restrict the subproblems to be of the form S_k.
  - We defined the subproblems using the S_k form.
  - Then we proved that greedy choice of a_m gives an optimal solution to S_k.

9.4. Greedy-Choice Property

Greedy-choice property:

- Can assemble a globally optimal solution by making locally optimal greedy choices.
  - We look at the optimal choice.
  - We modify the optimal solution to include the greedy choice if the greedy choice is not included.
  - We stop if the optimal solution includes the greedy choice.
- We can get efficiency gains from the greedy-choice property.
  - We preprocess the input to put it into greedy order.
  - We can use a priority queue if we have dynamic data.

9.5. Greedy Algorithm vs. Dynamic Programming

- Dynamic Programming:
  - Make a choice at each step.
  - Choice depends on knowing optimal solutions to subproblems. Solve subproblem first.
  - Solve bottom-up.
- Greedy Algorithm:
  - Make a choice at each step.
  - Make the choice before solving the subproblems.
  - Solve top-down.

9.5. Greedy Algorithm vs. Dynamic Programming

0-1 knapsack problem:

- n items.
- Item i is worth v_i dollar and weights w_i pounds.
- Find a most valuable subset of itmes with total weight $\leq$ W.
- Have to either take or not take an item (cannot take part of it).

Fractional knapsack problem:

- Like 0-1 knapsack but can take fractions of an item.

- Both knapsack problems have an optimal substructure.
- Fractional knapsack has the greedy choice property.
- 0-1 knapsack does not have the greedy choice property.

9.5. Greedy Algorithm vs. Dynamic Programming

Fractional knapsack problem:

- We rank items by value/weight: $\frac{v_i}{w_i}$.
- Take items in descreasing order of value/weight: $\frac{v_i}{w_i} \geq \frac{v_{i+1}}{w_{i+1}}$
- Take all of the items with greatest value/weight, and possibly a fraction of the next item.

9.5. Greedy Algorithm vs. Dynamic Programming

Algorithm (FRACTIONAL-KNAPSACK(v, w, W))

```
1  load = 0
2  i = 1
3  while (load < W and i ≤ n) do
4     if (w_i ≤ W - load) then
5        take all of item i
6     else
7        take (W−load)/w_i of item i
8     add what was taken to load
9     i = i + 1
```

9.5. Greedy Algorithm vs. Dynamic Programming

0-1 knapsack problem:

i	1	2	3
v_i	60	100	120
w_i	10	20	30
$\frac{v_i}{w_i}$	6	5	4

Suppose the maximum weight W = 50.

Dynamic Programming:

- We take item 2 and 3.
  - Total Value: 220
  - Total Weight: 50

Greedy Algorithm:

- We take item 1 and 2.
  - Total Value: 160
  - Total Weight: 30