Penn Engineering

ONLINE LEARNING

Video 2.1
Sampath Kannan

Optimization

- **Objective Function:** A function that assigns a value to each feasible solution

Optimization

- **Objective Function:** A function that assigns a value to each feasible solution
- **Optimization Problem:** Find the solution with the maximum(or minimum) objective function value

Optimization: Examples

Optimization problems appear everywhere!

Optimization: Examples

Optimization problems appear everywhere!

- **Shortest** path from location A to location B
- **Maximum** value of goods you can buy on a budget
- **Smallest** number of changes you need to make to transform one string into another
- Locating k hospitals in acommunity to **minimize** the maximum time anyone has to travel
- Compute the value of a function in the **fewest** steps

Optimization: Solutions

- Brute Force approach: look at objective function value of each possible solution and take the best
- There can be exponentially many solutions. Brute-force approach can take too long
- **Dynamic Programming** - efficient way to find the optimal solution for some problems
  - When can we use dynamic programming?
  - How can we apply it?

Penn Engineering

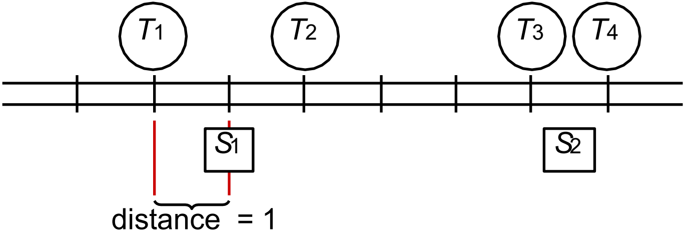
ONLINE LEARNING

Video 2.2

Sampath Kannan

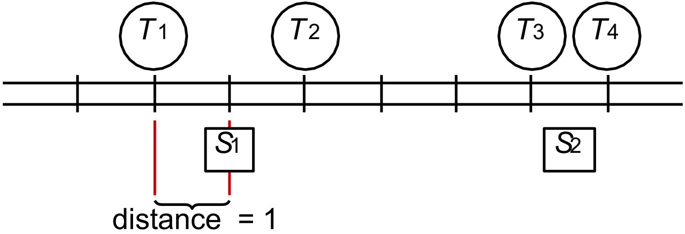
Station Placement

We want to place *k* stations along a train line so that the maximum distance between a town and its nearest station is minimized.



This diagram shows a cost 1 solution for *k* = 2 where the towns are located at positions 1, 3, 6 and 7.

Notation



- Notation: The towns area sequence of integers $t_1, t_2, \cdots, t_n$ and the stations area sequence of rational numbers $s_1, s_2, \cdots, s_k$

Top Level Decisions

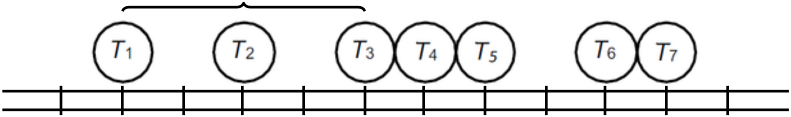
- Once we have the stations, we know which towns will use which.

Top Level Decisions

- Once we have the stations, we know which towns will use which.
- We can make a **top-level decision** about how many towns will use the left most station. Don’t know the answer so we have to try all possibilities!
- This is the idea of dynamic programming, we explore all choices and take the best one

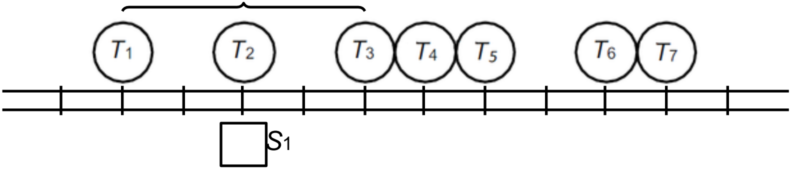
Top Level Decisions

If the first *i* towns use the left most station.. .



Top Level Decisions

If the first *i* towns use the left most station.. .

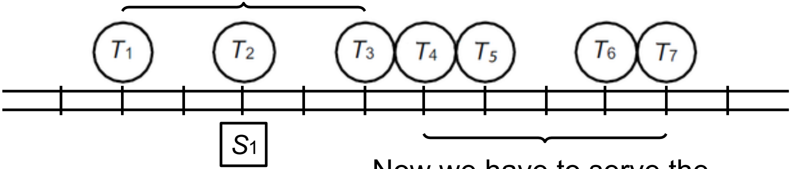


station should be at

$(t_i + t_1) / 2$

Top Level Decisions

If the first *i* towns use the left most station.. .



station should be at ($t_i + t_1$) / 2

Now we have to serve the remaining $n - i$ towns with $k - 1$ stations

Recurrence

- Locate([*i* , *j*], *l*) finds the best location for *A* stations to serve towns t_i through t_j and returns the maximum distance from any town to its nearest station.

Recurrence

- Locate([*i* , *j*], *A*) finds the best location for *A* stations to serve towns t_i through t_j and returns the maximum distance from any town to its nearest station.
- Recursive idea:

$$\text{Locate}([i, j], l) = \min_{x \in [i, j-1]} \max {}^{\cdot} \frac{t_x - t_i}{2}, \text{Locate}([x + 1, j], l - 1)$$

Penn Engineering

ONLINE LEARNING

Video 2.3
Sampath Kannan

Computing Locate

- Locate (*t*[*i* , *j*], *k*) finds the best location for *k* stations to serve towns t_i through t_j and returns the maximum distance from any town to its nearest station.
- Recursive idea: For *k* > 1

Locate(*t*[*i*, *j*], *k*) =

$$\min_{x \in \{i \ldots j-1\}} \max\left(\frac{t_x - t_i}{2}, \text{Locate}(t[x+1, j], k-1) \right)$$

- What does this mean? Why is it correct?

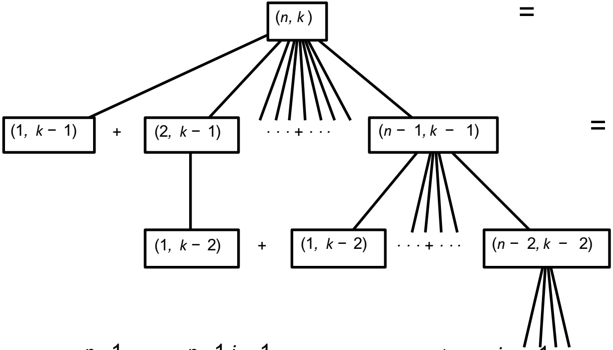
Computing Locate

- We want to compute Locate([1, n], k), the placement of k stations for n towns
- $T(n, k)$ = the time it takes to solve this problem

$$T(n, k) = \sum_{j=1}^{n-1} T(j, k-1) + n$$

$$T(1, k) = 1$$

$$T(n, 1) = 1$$



$$= n$$

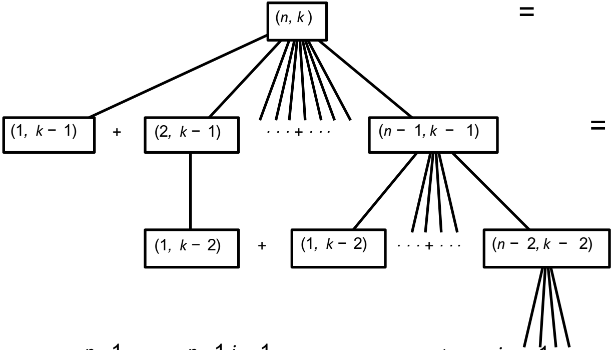
$$= \sum_{i_1=1}^{n-1} i_1$$

$$= \sum_{i_1=1}^{n-1} \sum_{i_2=1}^{i_1-1} i_2$$

$$= n + \sum_{i_1=1}^{n-1} i_1 + \sum_{i_1=1}^{n-1} \sum_{i_2=1}^{i_1-1} i_2 + \cdots + \sum_{i_1=1}^{n-1} \cdots \sum_{i_k-1=1}^{i_{k-2}-1} i_{k-1}$$

$$= n + \sum_{i=1}^{k-1} \sum_{m=1}^{n-i} m\binom{n-1-m}{i-1}$$

- This grows exponentially fast!



$$= n$$

$$= \sum_{i_1=1}^{n-1} i_1$$

$$= \sum_{i_1=1}^{n-1} \sum_{i_2=1}^{i_1-1} i_2$$

$$= n + \sum_{i_1=1}^{n-1} i_1 + \sum_{i_1=1}^{n-1} \sum_{i_2=1}^{i_1-1} i_2 + \cdots + \sum_{i_1=1}^{n-1} \cdots \sum_{i_k - 1=1}^{i_{k-2}-1} i_{k-1}$$

$$= n + \sum_{i=1}^{k-1} \sum_{m=1}^{n-i} m\binom{n-1-m}{i-1}$$

- This grows exponentially fast!
- Subproblem (1, *k* − 2) gets caled by (2, *k* − 1), (3, *k* − 1) *. ..*
- If we don’t recompute we can save time

Types of Subproblems

- What are the subproblems we see?

Types of Subproblems

- What are the subproblems we see?
- The new list of towns is always a suffice of the original list ($n - 1$ suffixes)
- The number of stations is always between 1 and k

Types of Subproblems

- What are the subproblems we see?
- The new list of towns is always a suffice of the original list ($n - 1$ suffixes)
- The number of stations is always between 1 and k
- The total number of subproblems is at most k ($n - 1$)
- Dynamic programming works when there aren't too many subproblems

Dynamic programming Locate

- Instead of thinking about a recursion tree, think about an array of subproblems *C*
- *C* [*i* , *j*] = the minimum cost of placing *j* station to serve towns *i* through *n*

Dynamic programming Locate

- Instead of thinking about a recursion tree, think about an array of subproblems *C*
- *C* [*i* , *j*] = the minimum cost of placing *j* station to serve towns *i* through *n*

```
Initialize C[i,j] = null
for all i let C[n,i] = 1
Locate(t[start,end],k):
  if t[start,k] = null
    c = Inf
    for x in start...end-1 do
      c = min(c, max((t[x]-t[end])/2,
                           Locate(t[x-1,end],k-1)))
    C[start,k] = c
    return c
  else
    return C[start,k]
```

Running Time

- Now each subproblem is only computed once.
- Each subproblem takes at most n operations to solve (remember there are at most$(n-1)k$ subproblems)

Running Time

- Now each subproblem is only computed once.
- Each subproblem takes at most n operations to solve (remember there are at most $(n - 1)\ k$ subproblems)
- So the new running time is $O(n^2k)$

Running Time

- Now each subproblem is only computed once.
- Each subproblem takes at most n operations to solve (remember there are at most $(n - 1)\ k$ subproblems)
- So the new running time is $O(n^2k)$
- We can also avoid the recursion entirely:

```
Locate(t[start,end],k):
  for all i let C[i,1] = (t[end]-t[i])/2
  for all i let C[end,i] = 0
  for j from 2 to k
    for i from end-1 to 1
      c = Inf
      for x from i to end-1
        c = min(c, max((t[x]-t[i])/2,C[x+1.j-1))
      C[i,j] = c
    return C[1,k]
```

Penn Engineering

ONLINE LEARNING

Video 2.4
Sampath Kannan

Dynamic Programming

What are the general properties of problems where dynamic programming is applicable?

Dynamic Programming

What are the general properties of problems where dynamic programming is applicable?

- **Optimal Substructure:** In order to solve the whole problem optimally, we need to solve certain subproblems optimally

Dynamic Programming

What are the general properties of problems where dynamic programming is applicable?

- **Optimal Substructure:** In order to solve the whole problem optimally, we need to solve certain subproblems optimally
- **Not-too-many subproblems:** The same few subproblems keep recurring, so we do not need to solve too many subproblems.

Dynamic Programming - Optimal Substructure

”Optimal Substructure”- when is it present?

- Another Example: Shortest path between two cities A and B.

Dynamic Programming - Optimal Substructure

”Optimal Substructure”- when is it present?

- Another Example: Shortest path between two cities A and B.
  - Suppose we have a set of cities, connected by roads
  - Want to find the shortest path connecting A and B,

$$A \rightarrow X_1 \rightarrow X_2 \rightarrow \ldots \rightarrow X_n \rightarrow B \quad = A \sim B$$

Dynamic Programming - Optimal Substructure

”Optimal Substructure”- when is it present?

- Another Example: Shortest path between two cities A and B.
  - Suppose we have a set of cities, connected by roads
  - Want to find the shortest path connecting A and B,

$$A \rightarrow X_1 \rightarrow X_2 \rightarrow \ldots \rightarrow X_n \rightarrow B \quad = \quad A \sim B$$

  - Suppose we knew one of the cities on that path was city C. What does this tell us about the shortest path from A to B?

$$A \sim C \sim B$$

Dynamic Programming - Optimal Substructure

”Optimal Substructure”- when is it present?

- Another Example: Shortest path between two cities A and B.
  - Suppose we have a set of cities, connected by roads
  - Want to find the shortest path connecting A and B,

$$A \rightarrow X_1 \rightarrow X_2 \rightarrow \ldots \rightarrow X_n \rightarrow B \quad = \quad A \sim B$$

  - Suppose we knew one of the cities on that path was city C. What does this tell us about the shortest path from A to B?

$$A \sim C \sim B$$

  - Shortest path from A to B:
    - Take the shortest path from A to C
    - Then take the shortest path from C to B.

Dynamic Programming - Optimal Substructure

”Optimal Substructure”- when is it present?

- Another Example: Shortest path between two cities A and B.
  - Suppose we have a set of cities, connected by roads
  - Want to find the shortest path connecting A and B,

$$A \rightarrow X_1 \rightarrow X_2 \rightarrow \ldots \rightarrow X_n \rightarrow B = A \sim B$$

  - Suppose we knew one of the cities on that path was city C. What does this tell us about the shortest path from A to B?

$$A \sim C \sim B$$

  - Shortest path from A to B:
    - Take the shortest path from A to C
    - Then take the shortest path from C to B.
  - This problem has optimal substructure!
    - Why? Optimal solutions is composed of the optimal solution to subproblems
    - Problem: It might be difficult to find this city C

Penn Engineering

ONLINE LEARNING

Video 2.5

Sampath Kannan

Dynamic Programming: LCS

Longest Common Subsequence(LCS)

- Given: two strings $s = s_1 s_2 \ldots s_m$ and $t = t_1\ t_2 \ldots t_n$
- Strings are over some alphabet (may be english, may be something else)

Dynamic Programming: LCS

Longest Common
Subsequence(LCS)

T	U	C	S	O	N

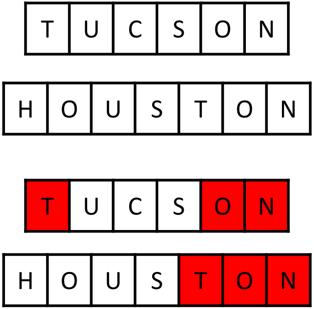
H	O	U	S	T	O	N

- Example: *s* = *TUCSON*, *t* = *HOUSTON*

Dynamic Programming: LCS

Longest Common Subsequence(LCS)

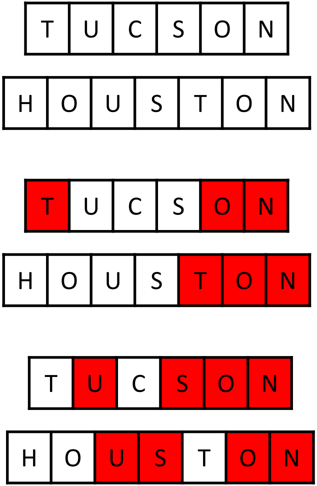
- Example: *s* = *TUCSON*, *t* = *HOUSTON*
- One common subsequence: TON (length 3)



Dynamic Programming: LCS

Longest Common Subsequence(LCS)

- Example: *s* = *TUCSON*, *t* = *HOUSTON*
- One common subsequence: TON (length 3)
- Longer one: USON (length 4, best)



T U C S O N

H O U S T O N

T U C S O N

H O U S T O N

T U C S O N

H O U S T O N

LCS: Top-Level Question

Can LCS be solved using dynamic programming?

- Given two subsequences, what might be a question for a top level decision?
- Note: the question must be such that its answers cover all possible solutions
- Idea 1: Does the longest common subsequence include the last letter of both strings?

LCS: Top-Level Question

Can LCS be solved using dynamic programming?

- Given two subsequences, what might be a question for a top level decision?
- Note: the question must be such that its answers cover all possible solutions
- Idea 1: Does the longest common subsequence include the last letter of both strings?
- If $s_m = t_n$ then they do: include it in the LCS, drop the last letters and recurse

T	U	C	S	O	N

H	O	U	S	T	O	N

LCS: Top-Level Question

Can LCS be solved using dynamic programming?

- Given two subsequences, what might be a question for a top level decision?
- Note: the question must be such that its answers cover all possible solutions
- Idea 1: Does the longest common subsequence include the last letter of both strings?
- If $s_m = t_n$ then they do: include it in the LCS, drop the last letters and recurse
- If not, don’t include. Drop the last letters and recurse.

T	U	C	S	O	N

H	O	U	S	T	O	N

LCS: Top-Level Question

Can LCS be solved using dynamic programming?

- Given two subsequences, what might be a question for a top level decision?
- Note: the question must be such that its answers cover all possible solutions
- Idea 1: Does the longest common subsequence include the last letter of both strings?
  - If $s_m = t_n$ then they do: include it in the LCS, drop the last letters and recurse
  - If not, don’t include. Drop the last letters and recurse.
  - Does this cover all possible solutions?

T	U	C	S	O	N

H	O	U	S	T	O	N

LCS: Top-Level Question

Can LCS be solved using dynamic programming?

- Given two subsequences, what might be a question for a top level decision?
- Note: the question must be such that its answers cover all possible solutions
- Idea 1: Does the longest common subsequence include the last letter of both strings?
  - If $s_m = t_n$ then they do: include it in the LCS, drop the last letters and recurse
  - If not, don’t include. Drop the last letters and recurse.
  - Does this cover all possible solutions?

T	U	C	S	O	N

H	O	U	S	T	O	N

Other ideas?

LCS: Top-Level Question

T	U	C	S	O	N

H	O	U	S	T	O	N

Can we come up with a better top-level decision?

LCS: Top-Level Question

Can we come up with a better top- level decision?

T	U	C	S	O	N

H	O	U	S	T	O	N

- First, a lemma:
- *Lemma:* If the last symbols of two strings match, there is a longest common subsequence that uses these symbols

LCS: Top-Level Question

Can we come up with a better top-level decision?

T	U	C	S	O	N

H	O	U	S	T	O	N

- First, a lemma:
- *Lemma:* If the last symbols of two strings match, there is a longest common subsequence that uses these symbols
  - Simple proof by contradiction!
  - Suppose both strings have *x* as their last symbol, but we find a LCS that does not end in *x* .
  - We could add *x* to the end of our current LCS, creating a new, better LCS.
  - If some other occurrence of *x* in either string is used in LCS, it doesn’t hurt to replace it with the last occurrence of *x* .

LCS: Top-Level Question

T	U	C	S	O	N

H	O	U	S	T	O	N

Can we come up with a better top-level decision?

- Good top-level question: If the last symbols don’t match, which one do we want to drop from consideration?
  - Note that one of them must be dropped!
  - We are not ignoring any possible solutions
  - How is this question better than our first approach?

LCS: Dynamic Programming

- Let LCS(i,j) representt he length of the longest common subsequence using the first i symbols of s and the first j symbols of t.
- Note that we could store the actual sequences themselves by carefully keeping track of indices

LCS: Dynamic Programming

- Let LCS(i,j) representt he length of the longest common subsequence using the first i symbols of s and the first j symbols of t.
- Note that we could store the actual sequences themselves by carefully keeping track of indices

```
if s_i = t_j:
   LCS(i,j) = 1 + LCS(i-1,j-1)
else:
   LCS(i,j) = max{ LCS(i-1,j), LCS(i,j-1)}
```

LCS: Dynamic Programming

- Let LCS(i,j) representt he length of the longest common subsequence using the first i symbols of s and the first j symbols of t.
- Note that we could store the actual sequences themselves by carefully keeping track of indices

```
if s_i = t_j:
   LCS(i,j) = 1 + LCS(i-1,j-1)
else:
   LCS(i,j) = max{ LCS(i-1,j), LCS(i,j-1)}
```

- We can write a simple recursive algorithm to implement this

LCS: Dynamic Programming

- Let LCS(i,j) representt he length of the longest common subsequence using the first i symbols of s and the first j symbols of t.
- Note that we could store the actual sequences themselves by carefully keeping track of indices

```
if s_i = t_j:
   LCS(i,j) = 1 + LCS(i-1,j-1)
else:
   LCS(i,j) = max{ LCS(i-1,j), LCS(i,j-1)}
```

- We can write a simple recursive algorithm to implement this
- How slow would it be? Would it be resolving any of the same problems?

LCS: Dynamic Programming

LCS: Dynamic
Programming

LCS: Dynamic Programming

LCS: Dynamic Programming

- Instead,we’ll use a bottom-up solution:
  - Solve the smalest subproblems first, build up to bigger solutions
  - Solve every subproblem exactly once

LCS: Dynamic Programming

LCS: Dynamic Programming

- Instead,we’ll use a bottom-up solution:
  - Solve the smalest subproblems first, build up to bigger solutions
  - Solve every subproblem exactly once
- What do *LCS*(*i* , 0) and *LCS*(0*, j*) represent?

LCS: Dynamic Programming

LCS: Dynamic Programming

- Instead,we'll use a bottom-up solution:
  - Solve the smalest subproblems first, build up to bigger solutions
  - Solve every subproblem exactly once
- What do *LCS*(*i* , 0) and *LCS*(0*, j*) represent?
  - One of our strings is empty, so LCS must be 0

LCS: Dynamic Programming

LCS: Dynamic Programming

- What do *LCS*(*i* , 0) and *LCS*(0*, j*) represent?
    - One of our strings is empty, so LCS must be 0
- Which subproblem is our solution to the whole problem?

LCS: Dynamic Programming

LCS: Dynamic Programming

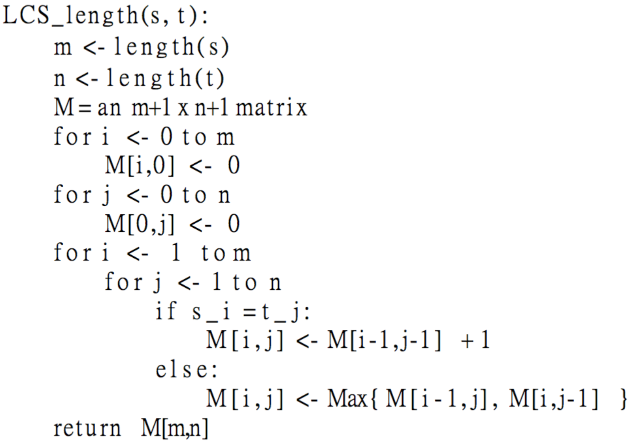
- What do *LCS*(*i* , 0) and *LCS*(0*, j*) represent?
  - One of our strings is empty, so LCS must be 0
- Which subproblem is our solution to the whole problem?
  - *LCS*(*m* , *n*)

LCS: Dynamic Programming

LCS: Dynamic Programming

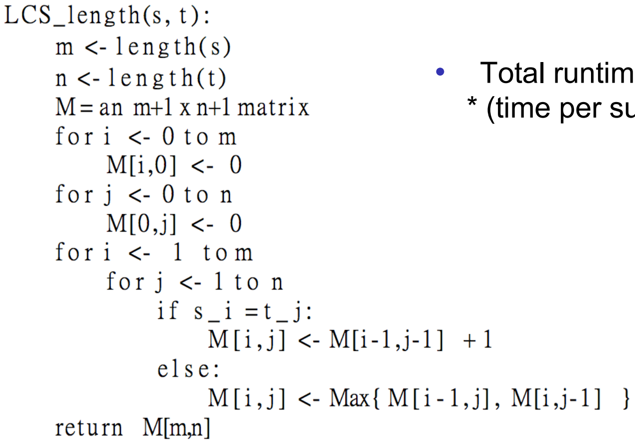
- Over all algorithm:
  - Create an (m + 1)x (n + 1) grid, where slot (i , j) represents the solution to $LCS(i,j)$.
  - Fill in the trivial subproblem solutions (LCS (i , 0) and LCS (0,j))
    Fill the grid row by row (or column by column) using the previous recurrence.

LCS: Algorithm And Runtime



```
LCS_length(s, t):
    m <- length(s)
    n <- length(t)
    M = an m+1 x n+1 matrix
    for i <- 0 to m
        M[i,0] <- 0
    for j <- 0 to n
        M[0,j] <- 0
    for i <- 1 to m
        for j <- 1 to n
            if s_i = t_j:
                M[i,j] <- M[i-1,j-1] + 1
            else:
                M[i,j] <- Max{ M[i-1,j], M[i,j-1] }
    return M[m,n]
```

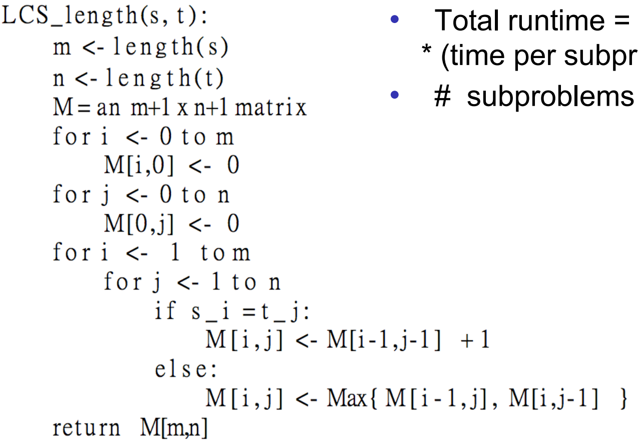
LCS: Algorithm And Runtime



```
LCS_length(s, t):
    m <- length(s)
    n <- length(t)
    M = an m+1 x n+1 matrix
    for i <- 0 to m
        M[i,0] <- 0
    for j <- 0 to n
        M[0,j] <- 0
    for i <- 1 to m
        for j <- 1 to n
            if s_i = t_j:
                M[i,j] <- M[i-1,j-1] + 1
            else:
                M[i,j] <- Max{ M[i-1,j], M[i,j-1] }
    return M[m,n]
```

- Total runtime = (# subproblems) * (time per subproblem)

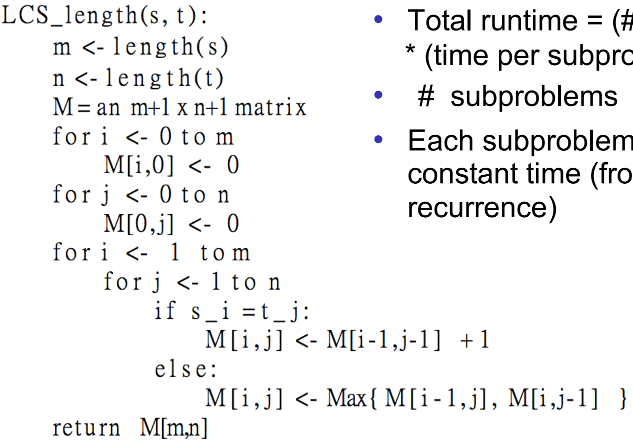
LCS: Algorithm And Runtime



```
LCS_length(s, t):
    m <- length(s)
    n <- length(t)
    M = an m+1 x n+1 matrix
    for i <- 0 to m
        M[i,0] <- 0
    for j <- 0 to n
        M[0,j] <- 0
    for i <- 1 to m
        for j <- 1 to n
            if s_i = t_j:
                M[i,j] <- M[i-1,j-1] + 1
            else:
                M[i,j] <- Max{ M[i-1,j], M[i,j-1] }
    return M[m,n]
```

- Total runtime = (# subproblems) * (time per subproblem)
- # subproblems = *mn*

LCS: Algorithm And Runtime



```
LCS_length(s, t):
    m <- length(s)
    n <- length(t)
    M = an m+1 x n+1 matrix
    for i <- 0 to m
        M[i,0] <- 0
    for j <- 0 to n
        M[0,j] <- 0
    for i <- 1 to m
        for j <- 1 to n
            if s_i =t_j:
                M[i,j] <- M[i-1,j-1] +1
            else:
                M[i,j] <- Max{ M[i-1,j], M[i,j-1] }
    return M[m,n]
```

- Total runtime = (# subproblems) * (time per subproblem)
- # subproblems = *mn*
- Each subproblem is calculated in constant time (from the recurrence)

LCS: Algorithm And Runtime

```
LCS_length(s, t):
    m <- length(s)
    n <- length(t)
    M = an m+1 x n+1 matrix
    for i <- 0 to m
        M[i,0] <- 0
    for j <- 0 to n
        M[0,j] <- 0
    for i <- 1 to m
        for j <- 1 to n
            if s_i = t_j:
                M[i,j] <- M[i-1,j-1] + 1
            else:
                M[i,j] <- Max{ M[i-1,j], M[i,j-1] }
    return M[m,n]
```

- Total runtime = (# subproblems) * (time per subproblem)
- # subproblems = *mn*
- Each subproblem is calculated in constant time (from the recurrence)

- **Total runtime of LCS:** *O*(*mn*)

LCS: Original Example

T	U	C	S	O	N

H	O	U	S	T	O	N

LCStable

		T	U	C	S	O	N
		0	0	0	0	0	0
H	0						
O	0						
U	0						
S	0						
T	0						
O	0						
N	0						

LCS: Original Example

T	U	C	S	O	N

H	O	U	S	T	O	N

LCStable

		T	U	C	S	O	N
		0	0	0	0	0	0
H	0	0					
O	0	0					
U	0	0					
S	0	0					
T	0	1					
O	0	1					
N	0	1					

LCS: Original Example

T	U	C	S	O	N

H	O	U	S	T	O	N

LCStable

		T	U	C	S	O	N
		0	0	0	0	0	0
H	0	0	0				
O	0	0	0				
U	0	0	1				
S	0	0	1				
T	0	1	1				
O	0	1	1				
N	0	1	1				

LCS: Original Example

T	U	C	S	O	N

H	O	U	S	T	O	N

LCStable

		T	U	C	S	O	N
		0	0	0	0	0	0
H	0	0	0	0			
O	0	0	0	0			
U	0	0	1	1			
S	0	0	1	1			
T	0	1	1	1			
O	0	1	1	1			
N	0	1	1	1			

LCS: Original Example

T	U	C	S	O	N

H	O	U	S	T	O	N

LCStable

		T	U	C	S	O	N
		0	0	0	0	0	0
H	0	0	0	0	0		
O	0	0	0	0	0		
U	0	0	1	1	1		
S	0	0	1	1	2		
T	0	1	1	1	2		
O	0	1	1	1	2		
N	0	1	1	1	2		

LCS: Original Example

T	U	C	S	O	N

H	O	U	S	T	O	N

LCStable

		T	U	C	S	O	N
		0	0	0	0	0	0
H	0	0	0	0	0	0	
O	0	0	0	0	0	1	
U	0	0	1	1	1	1	
S	0	0	1	1	2	2	
T	0	1	1	1	2	2	
O	0	1	1	1	2	3	
N	0	1	1	1	2	3	

LCS: Original Example

T	U	C	S	O	N

H	O	U	S	T	O	N

LCStable

		T	U	C	S	O	N
		0	0	0	0	0	0
H	0	0	0	0	0	0	0
O	0	0	0	0	0	1	1
U	0	0	1	1	1	1	1
S	0	0	1	1	2	2	2
T	0	1	1	1	2	2	2
O	0	1	1	1	2	3	3
N	0	1	1	1	2	3	4

Penn Engineering

ONLINE LEARNING

Video 2.6

Sampath Kannan

Optimal Static Binary Search Trees

- Input: n keys (in sorted order), $k_1 < k_2 < \ldots < k_n$, along with probability of each key being accessed, $p_1, p_2, \ldots, p_n$

Optimal Static Binary Search Trees

- Input: n keys (in sorted order), $k_1 < k_2 < \ldots < k_n$, along with probability of each key being accessed, $p_1, p_2, \ldots, p_n$
- Goal: Build a BST that minimizes average access time
  - Minimize the value $\sum_{i=0}^{n-1}(p_i)(depth(k_i) + 1)$

P (access) access time

Optimal Static Binary Search Trees

- Input: n keys (in sorted order), $k_1 < k_2 < \ldots < k_n$, along with probability of each key being accessed, $p_1, p_2, \ldots, p_n$
- Goal: Build a BST that minimizes average access time
  - Minimize the value $\sum_{i=0}^{n-1} (p_i)(depth(k_i) + 1)$

P (access)

access time

key	a	b	c	d	e	f	g	h	i	j
P(access)	0.15	0.05	0.08	0.1	0.2	0.05	0.1	0.07	0.15	0.05

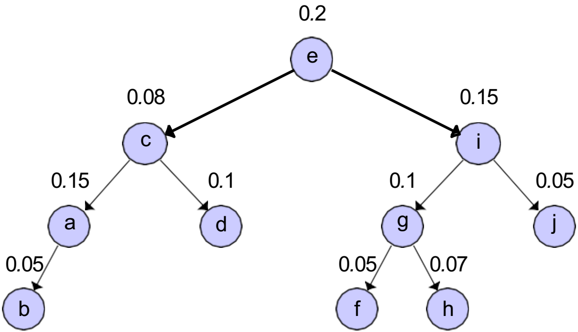
Optimal Static Binary Search Trees

- Input: n keys (in sorted order), $k_1 < k_2 < \ldots < k_n$, along with probability of each key being accessed, $p_1, p_2, \ldots, p_n$
- Goal: Build a BST that minimizes average access time
    - Minimize the value $\sum_{i=0}^{n-1} (p_i)(depth(k_i) + 1)$

P (access)

access time

key	a	b	c	d	e	f	g	h	i	j
P(access)	0.15	0.05	0.08	0.1	0.2	0.05	0.1	0.07	0.15	0.05



Optimal Static BSTs

- Question for top-level decision

Optimal Static BSTs

- Question for top-level decision
- Which element to make the root?

Optimal Static BSTs

- Question for top-level decision
- Which element to make the root?
- If we make k_i the root:

Optimal Static BSTs

- Question for top-level decision
- Which element to make the root?
- If we make k_i the root:
  - Search Tree Property:
    - The left subtree must consist of $k_1, \ldots, k_{i-1}$
    - The right subtree must consist of $k_{i+1}, \ldots, k_n$

Optimal Static BSTs

- Question for top-level decision
- Which element to make the root?
- If we make k_i the root:
  - Search Tree Property:
    - The left subtree must consist of $k_1, \ldots, k_{i-1}$
    - The right subtree must consist of $k_{i+1}, \ldots, k_n$
  - Optimal Sub structure Property:
    - The left and right subtrees should also be optimal BSTs for their elements

Optimal Static BSTs

- What subproblems might we have to solve?

Optimal Static BSTs

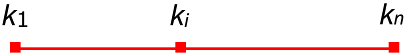
- What subproblems might we have to solve?

k_1 k_n

Goal: create optimal BST for keys $k_1 \ldots k_n$

Optimal Static BSTs

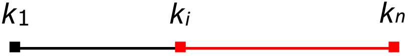
- What subproblems might we have to solve?



Choose k_i as root

Optimal Static BSTs

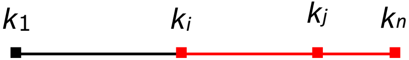
- What subproblems might we have to solve?



Compute right subtree (Using keys $k_{i+1} \ldots k_n$)

Optimal Static BSTs

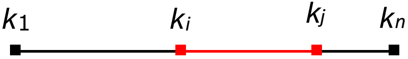
- What subproblems might we have to solve?



Choose k_j as root (of the right subtree)

Optimal Static BSTs

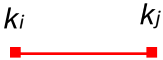
- What subproblems might we have to solve?



Compute its left subtree
(Using keys $k_{i+1} \ldots k_{j-1}$)

Optimal Static BSTs

- What subproblems might we have to solve?



Optimal Static BSTs

- What subproblems might we have to solve?
  - Potentially one for each contiguous set of keys [$k_i \ldots k_j$]

Optimal Static BSTs

- What subproblems might we have to solve?
  - Potentially one for each contiguous set of keys [$k_i \ldots k_j$]
- How many are there?

Optimal Static BSTs

- What subproblems might we have to solve?
  - Potentially one for each contiguous set of keys $[k_i \ldots k_j]$
- How many are there?
  - Each contiguous set is defined by choosing two keys from our set (smallest and largest key in the interval)
  - $\binom{n}{2} = \frac{n(n-1)}{2} \quad O(n^2)$

Optimal Static BSTs: Recurrence

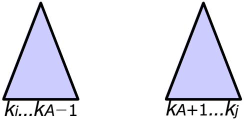
- Can we define a recurrence for this problem?

Optimal Static BSTs: Recurrence

- Can we define a recurrence for this problem?
- $T(i, j)$ = The average access time of an optimal BST for the keys $k_i, k_{i+1}..., k_j$

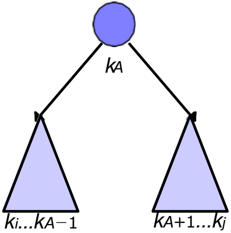
Optimal Static BSTs: Recurrence

- Can we define a recurrence for this problem?
- $T(i, j)$ = The average access time of an optimal BST for the keys $k_i, k_{i+1}\ldots, k_j$



Optimal Static BSTs: Recurrence

- Can we define a recurrence for this problem?
- $T(i, j)$ = The average access time of an optimal BST for the keys $k_i, k_{i+1} \ldots, k_j$



Search times for every element in both subtrees increases by 1

P(access element in left subtree)

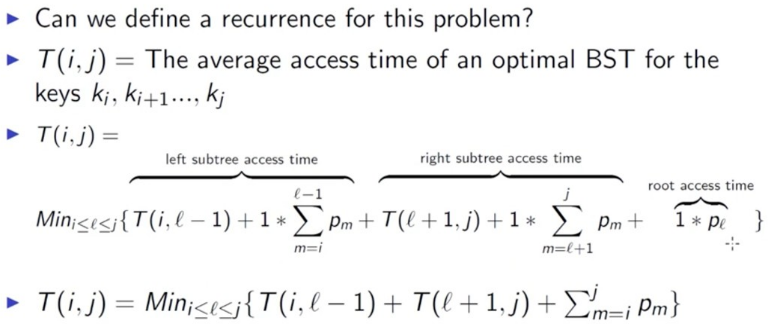
$= \Sigma_{m=i}^{A-1} p_m$

Optimal Static BSTs: Recurrence

- Can we define a recurrence for this problem?
- $T(i, j)$ = The average access time of an optimal BST for the keys $k_i, k_{i+1} \ldots, k_j$
- $T(i, j) =$

$$Min_{i \leq \ell \leq j} \{ \overbrace{T(i, \ell - 1) + 1 * \sum_{m=i}^{\ell-1} p_m}^{\text{left subtree access time}} + \overbrace{T(\ell + 1, j) + 1 * \sum_{m=\ell+1}^{j} p_m}^{\text{right subtree access time}} + \overbrace{1 * p_\ell}^{\text{root access time}} \}$$

Optimal Static BSTs: Recurrence

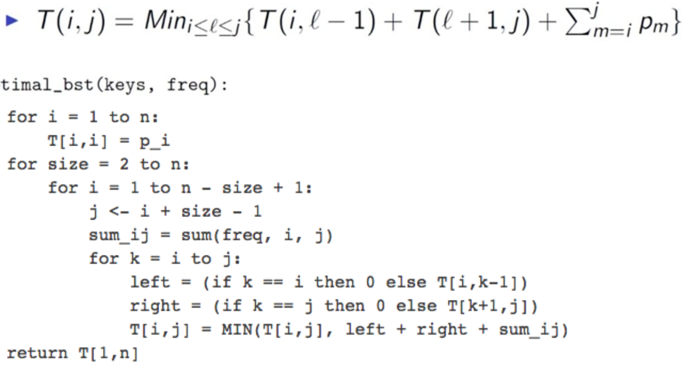


- Can we define a recurrence for this problem?
- $T(i,j)$ = The average access time of an optimal BST for the keys $k_i, k_{i+1}..., k_j$
- $T(i,j) =$

$$Min_{i\leq\ell\leq j}\{\overbrace{T(i,\ell-1)+1*\sum_{m=i}^{\ell-1}p_m}^{\text{left subtree access time}}+\overbrace{T(\ell+1,j)+1*\sum_{m=\ell+1}^{j}p_m}^{\text{right subtree access time}}+\overbrace{1*p_\ell}^{\text{root access time}}\}$$

- $T(i,j) = Min_{i\leq\ell\leq j}\{T(i,\ell-1)+T(\ell+1,j)+\sum_{m=i}^{j}p_m\}$

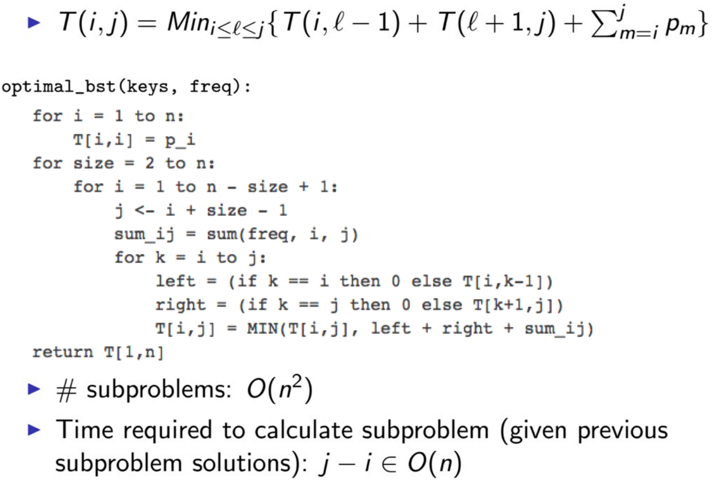
Optimal Static BSTs: Recurrence



- $T(i,j) = Min_{i \leq \ell \leq j}\{T(i, \ell - 1) + T(\ell + 1, j) + \sum_{m=i}^{j} p_m\}$

```
optimal_bst(keys, freq):
  for i = 1 to n:
      T[i,i] = p_i
  for size = 2 to n:
      for i = 1 to n - size + 1:
          j <- i + size - 1
          sum_ij = sum(freq, i, j)
          for k = i to j:
              left = (if k == i then 0 else T[i,k-1])
              right = (if k == j then 0 else T[k+1,j])
              T[i,j] = MIN(T[i,j], left + right + sum_ij)
  return T[1,n]
```

Optimal Static BSTs: Recurrence

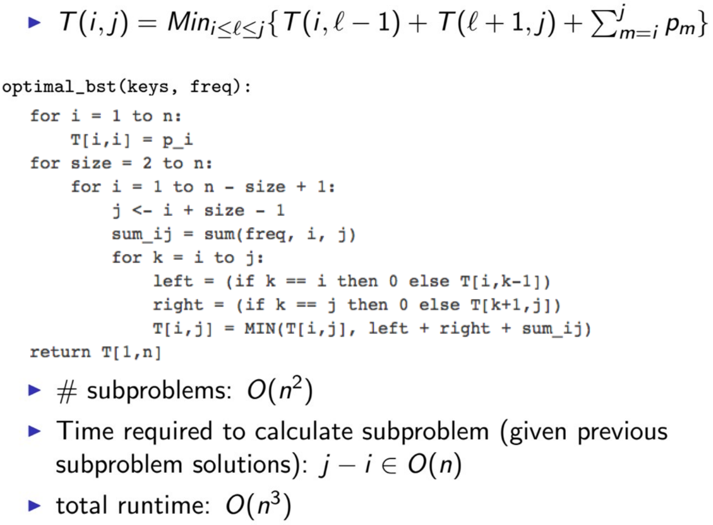


- $T(i,j) = Min_{i \leq \ell \leq j}\{T(i, \ell - 1) + T(\ell + 1, j) + \sum_{m=i}^{j} p_m\}$

```
optimal_bst(keys, freq):
   for i = 1 to n:
       T[i,i] = p_i
   for size = 2 to n:
       for i = 1 to n - size + 1:
           j <- i + size - 1
           sum_ij = sum(freq, i, j)
           for k = i to j:
               left = (if k == i then 0 else T[i,k-1])
               right = (if k == j then 0 else T[k+1,j])
               T[i,j] = MIN(T[i,j], left + right + sum_ij)
   return T[1,n]
```

- # subproblems: $O(n^2)$
- Time required to calculate subproblem (given previous subproblem solutions): $j - i \in O(n)$

Optimal Static BSTs: Recurrence



- $T(i,j) = Min_{i \leq \ell \leq j}\{T(i,\ell-1) + T(\ell+1,j) + \sum_{m=i}^{j} p_m\}$

```
optimal_bst(keys, freq):
   for i = 1 to n:
       T[i,i] = p_i
   for size = 2 to n:
       for i = 1 to n - size + 1:
           j <- i + size - 1
           sum_ij = sum(freq, i, j)
           for k = i to j:
               left = (if k == i then 0 else T[i,k-1])
               right = (if k == j then 0 else T[k+1,j])
               T[i,j] = MIN(T[i,j], left + right + sum_ij)
   return T[1,n]
```

- # subproblems: $O(n^2)$
- Time required to calculate subproblem (given previous subproblem solutions): $j - i \in O(n)$
- total runtime: $O(n^3)$

Dynamic Programming - Design

- How do we design dynamic programming algorithms?
- Understand if your problem has the two properties
- If so, ask what the top-level decision question is

Dynamic Programming - Design

- How do we design dynamic programming algorithms?
- Understand if your problem has the two properties
- If so, ask what the top-level decision question is
  - There could be many possible questions
  - Picking the right one is an art

Dynamic Programming - Design

- How do we design dynamic programming algorithms?
- Understand if your problem has the two properties
- If so, ask what the top-level decision question is
    - There could be many possible questions
    - Picking the right one is an art
- Next, design a recursive algorithm that solves the whole problem by considering each answer to the top-level question, and recursively solving the resulting subproblems

Dynamic Programming - Design

- Helpful to write the solution as a recurrence

Dynamic Programming - Design

- Helpful to write the solution as a recurrence
  - Express the cost as a recurrence relation that considers each possible answer to the top-level question
  - Generaly parallels the algorithm

Dynamic Programming - Design

- Helpful to write the solution as a recurrence
  - Express the cost as a recurrence relation that considers each possible answer to the top-level question
  - Generaly parallels the algorithm

- Another design choice: whether to implement the algorithm ”top-down” or ”bottom-up”

Dynamic Programming - Design

- Helpful to write the solution as a recurrence
  - Express the cost as a recurrence relation that considers each possible answer to the top-level question
  - Generaly parallels the algorithm

- Another design choice: whether to implement the algorithm ”top-down” or ”bottom-up”
  - top-down: Start at the top-level problem, recursively solve the subproblems.

Dynamic Programming - Design

- Helpful to write the solution as a recurrence
  - Express the cost as a recurrence relation that considers each possible answer to the top-level question
  - Generaly parallels the algorithm
- Another design choice: whether to implement the algorithm ”top-down” or ”bottom-up”
  - top-down: Start at the top-level problem, recursively solve the subproblems.
  - bottom-up: Iteratively compute solutions to subproblems, starting at the smallest subproblems first.

Dynamic Programming - Design

- Helpful to write the solution as a recurrence
  - Express the cost as a recurrence relation that considers each possible answer to the top-level question
  - Generaly parallels the algorithm
- Another design choice: whether to implement the algorithm ”top-down” or ”bottom-up”
  - top-down: Start at the top-level problem, recursively solve the subproblems.
  - bottom-up: Iteratively compute solutions to subproblems, starting at the smallest subproblems first.
    - Generally a little bit faster
    - Requires some work to make sure subproblems are being solved in the correct order

Dynamic Programming - Design

- Helpful to write the solution as a recurrence
  - Express the cost as a recurrence relation that considers each possible answer to the top-level question
  - Generaly parallels the algorithm
- Another design choice: whether to implement the algorithm ”top-down” or ”bottom-up”
  - top-down: Start at the top-level problem, recursively solve the subproblems.
  - bottom-up: Iteratively compute solutions to subproblems, starting at the smallest subproblems first.
    - Generally a little bit faster
    - Requires some work to make sure subproblems are being solved in the correct order
- Basic algorithm **only computes the cost of the solution.** Keeping track of the actual solution requires some extra book keeping

Penn
Engineering

ONLINE LEARNING

Video 2.7
Sampath Kannan

Greedy Algorithms

- For some problems, we can immediately decide what the best answer to the top-level decision question is.
- If so, we can make that choice and solve the resulting smaller problems.

Greedy Algorithms

- For some problems, we can immediately decide what the best answer to the top-level decision question is.
- If so, we can make that choice and solve the resulting smaller problems.
- Algorithms that work like this are called **greedy algorithms:**

  We greedily choose the best option at the moment because it will also lead to the best solution overall.

- In general, greedy algorithms are easy to design, but hard to prove correct.

Greedy: Example

- **Input:** n jobs, $j_1, j_2, ... j_n$, and at ime T
- Job j_i takes t_i units of time to complete
- Only one job can be performed at a time

Greedy: Example

- **Input:** n jobs, $j_1, j_2, \ldots j_n$, and a time T
- Job j_i takes t_i units of time to complete
- Only one job can be performed at a time
- **Goal:** Order the jobs in away that maximizes total # of jobs completed

Greedy: Example

- Top-level decision: Which job should be completed first?

Greedy: Example

- Top-level decision: Which job should be completed first?
- The greedy choice would be to complete the job that takes the smallest amount of time first

Greedy: Example

- Top-leve Idecision: Which job should be completed first?
- The greedy choice would be to complete the job that takes the smallest amount of time first
  - Ø Why? Maximizes amount of time remaining after completion of 1 job

Greedy: Example

- Solution:
  - Sort the jobs in increasing order of t_i (time for each job)
  - Complete as many jobs as possible in this order, until T units of time have passed

Greedy: Example

- Solution:
  - Sort the jobs in increasing order of t_i (time for each job)
  - Complete as many jobs as possible in this order, until T units of time have passed
- Correctness of the greedy solution may seem in clear in this case.
- In more complicated problems, this is less obvious!

Penn
Engineering
ONLINE LEARNING

Video 2.8
Sampath Kannan

Greedy Algorithms

Another (more complicated) scheduling problem:

- **Input:** nevents, $e_1, e_2, \ldots, e_n$
- Each event e_i starts at time s_i and finishes at time f_i
- Only one event can be running at any given time
- **Goal:** Find the maximum number of events we can schedule

Scheduling: Some Ideas

Top level decision: Which event to schedule first? What are some greedy choices that we could make?

Scheduling: Some Ideas

Top level decision: Which event to schedule first? What are some greedy choices that we could make?

- Idea 1: Schedule the event with the earliest start time. Remove any events that conflict with it.

Scheduling: Some Ideas

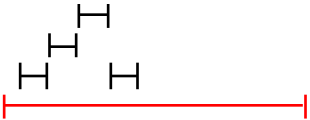
Top level decision: Which event to schedule first? What are some greedy choices that we could make?

- Idea 1: Schedule the event with the earliest start time. Remove any events that conflict with it.
- Will this work?

Scheduling: Some Ideas

Top level decision: Which event to schedule first? What are some greedy choices that we could make?

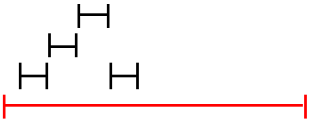
- Idea 1: Schedule the event with the earliest start time. Remove any events that conflict with it.
- Will this work?



Scheduling: Some Ideas

Top level decision: Which event to schedule first? What are some greedy choices that we could make?

- Idea 1: Schedule the event with the earliest start time. Remove any events that conflict with it.
- Will this work?

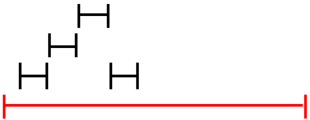


- Idea 2: Schedule the shortest event. Removing any events that conflict with it.

Scheduling: Some Ideas

Top level decision: Which event to schedule first? What are some greedy choices that we could make?

- Idea 1: Schedule the event with the earliest start time. Remove any events that conflict with it.
- Will this work?

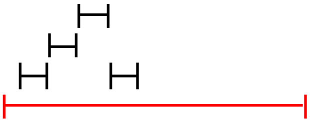


- Idea 2: Schedule the shortest event. Removing any events that conflict with it.
- Will this work?

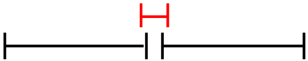
Scheduling: Some Ideas

Top level decision: Which event to schedule first? What are some greedy choices that we could make?

- Idea 1: Schedule the event with the earliest start time. Remove any events that conflict with it.
- Will this work?



- Idea 2: Schedule the shortest event. Removing any events that conflict with it.
- Will this work?



Scheduling: Correct Greedy Algorithm

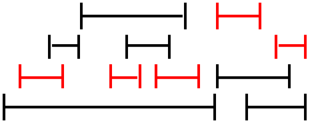
- Idea: Sort events by finish time!
- Let us renumber the events in order of increasing finish time,
  $e_1, e_2, ..., e_n$
- Pick e_1; eliminate events that overlap with e_1; Recurse.

Scheduling: Correct Greedy Algorithm

- Idea: Sort events by finish time!
- Let us renumber the events in order of increasing finish time,
  $e_1, e_2, \ldots, e_n$
- Pick e_1; eliminate events that overlap with e_1; Recurse.
- Why is this agood idea? Could an optimal algorithm schedule more events?

Scheduling: Correct GreedyAlgorithm

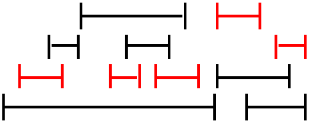
- Idea: Sort events by finish time!
- Let us renumber the events in order of increasing finish time,
  $e_1, e_2, ..., e_n$
- Pick e_1; eliminate events that overlap with e_1; Recurse.
- Why is this agood idea? Could an optimal algorithm schedule more events?



Scheduling: Correct Greedy Algorithm

- Idea: Sort events by finish time!
- Let us renumber the events in order of increasing finish time,

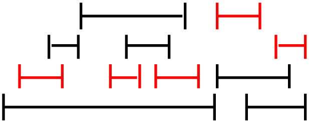
  $e_1, e_2, ..., e_n$
- Pick e_1; eliminate events that overlap with e_1; Recurse.
- Why is this agood idea? Could an optimal algorithm schedule more events?



- Intuition: Our greedy first choice leaves the maximum time for remaining events.

Scheduling: Correct Greedy Algorithm

- Idea: Sort events by finish time!
- Let us renumber the events in order of increasing finish time,
  $e_1, e_2, \ldots, e_n$
- Pick e_1; eliminate events that overlap with e_1; Recurse.
- Why is this agood idea? Could an optimal algorithm schedule more events?



- Intuition: Our greedy first choice leaves the maximum time for remaining events.
- Optimum couldn't do better.

Greedy Algorithm: Correctness

We must prove that:

Greedy Algorithm: Correctness

We must prove that:

- The first choice that the greedy algorithm makes can be continued on to an optimal solution (greedy choice)

Greedy Algorithm: Correctness

We must prove that:

- The first choice that the greedy algorithm makes can be continued on to an optimal solution (greedy choice)
- After the greedy choice is made, solving the rest of the problem optimally will solve the entire problem optimally (optimal substructure)

Greedy Algorithm: Correctness

Correctness of the greedy choice:

Greedy Algorithm: Correctness

Correctness of the greedy choice:

- Let $O = e_{o_1}\, e_{o_2} \ldots e_{o_k}$ be an optimal sequence of events in increasing order of finish time.

Greedy Algorithm: Correctness

Correctness of the greedy choice:

- Let $O = e_{o_1} e_{o_2} \ldots e_{o_k}$ be an optimal sequence of events in increasing order of finish time.
- **Exchange property:** If e_{o_1}ƒ $= e_1$, we can replace e_{o_1} by e_1 and still get an optimal solution

Greedy Algorithm: Correctness

Correctness of the greedy choice:

- Let $O = e_{o_1} e_{o_2} ... e_{o_k}$ be an optimal sequenceof events in increasing order of finish time.
- **Exchange property:** If e_{o_1}ƒ = e_1, we canreplace e_{o_1} by e_1 and still get an optimal solution
- Note that e_1 does not overlap $[s_{o_2} ... f_{o_k}]$

Greedy Algorithm: Correctness

Correctness of the greedy choice:

- Let $O = e_{o_1} e_{o_2} ... e_{o_k}$ be an optimal sequenceof events in increasing order of finish time.
- **Exchange property:** If e_{o_1}ƒ $= e_1$, we canreplace e_{o_1} by e_1 and still get an optimal solution
- Note that e_1 does not overlap $[s_{o_2} ... f_{o_k}]$

Optimal substructure:

GreedyAlgorithm: Correctness

Correctness of the greedy choice:

- Let $O = e_{o_1}\, e_{o_2} \ldots e_{o_k}$ be an optimal sequenceof events in increasing order of finish time.
- **Exchange property:** If e_{o_1}ƒ = *e*1, we canreplace e_{o_1} by *e*1 and still get an optimal solution
- Note that *e*1 does not overlap $[s_{o_2} \ldots f_{o_k}]$

Optimal substructure:

- Consider an optimal solution$O = e_{o_1} e_{o_2} \ldots e_{o_k}$
- The solution $e_{o_2} \ldots e_{o_k}$ must be the optimal sequence of events for the time interval $[s_{o_2} \ldots f_{o_k}]$
- Otherwise, take the better sequence and replace$e_{o_2} \ldots e_{o_k}$ with it!

Penn Engineering

ONLINE LEARNING

Video 2.9

Sampath Kannan

File Compression

- Suppose we are sending English text over the internet, we need a way to encode the symbols as a sequence of ones and zeros.
- We'd like to minimize the total amount of information sent (the length of the sequence).
- Idea: use shorter encoding for more frequent symbols (e.g., 'e', 'r', 't').

File Compression

- Suppose we are sending English text over the internet, we need a way to encode the symbols as a sequence of ones and zeros.
- We'd like to minimize the total amount of information sent (the length of the sequence).
- Idea: use shorter encoding for more frequent symbols (e.g., 'e', 'r', 't').
- Bad Example:
    - $a \mapsto 010, f \mapsto 0101, g \mapsto 1110, s \mapsto 110, \ldots$

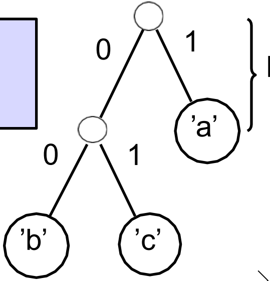
File Compression

- Suppose we are sending English text over the internet, we need a way to encode the symbols as a sequence of ones and zeros.
- We'd like to minimize the total amount of information sent (the length of the sequence).
- Idea: use shorter encoding for more frequent symbols (e.g., 'e', 'r', 't').
- Bad Example:
    - *a*›→010*, f* ›→0101*, g*›→1110*, s* ›→110*, . . .*
- Problem: 0101110 can be $\boxed{010}\boxed{1110}$ = 'ag', or $\boxed{0101}\boxed{110}$ = 'fs'
- Solution: No code can be a prefix of another

Prefix Codes

Let the alphabet be$\Sigma = \{a, b, c\}$ and for each symbol $x \in \Sigma$ we let $f(x)$ be the frequency of x and $d(x)$ be the length of the encoding of x.

$a \mapsto 1$
$b \mapsto 00$
$c \mapsto 01$



Note: depth(x) = $d(x)$

Goal: minimize $\sum_{x \in Sigma} d(x)f(x)$

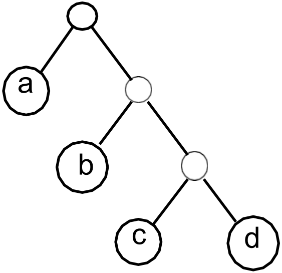
Attempt 1 (Greedy):

Most frequent symbol›→0, recurse on remaining symbols

Attempt 1 (Greedy):

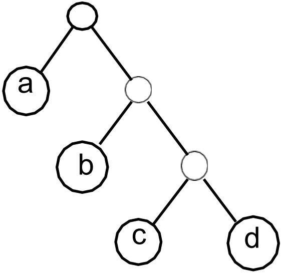
Most frequent symbol›→0, recurse on remaining symbols

$\Sigma = \{a, b, c, d\}$
$f(a) = 0.26, f(b) = 0.255$
$f(c) = 0.245, f(d) = 0.24$



Attempt 1 (Greedy):
Most frequent symbol›→0,
recurse on remaining symbols

$\Sigma = \{a, b, c, d\}$
$f(a) = 0.26, f(b) = 0.255$
$f(c) = 0.245, f(d) = 0.24$



The cost of this tree is ≈ 2.2 but if we gave each symbol a code of length 2 the cost would be 2

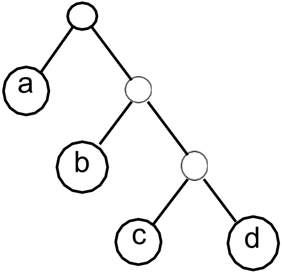
Attempt 1 (Greedy):

Most frequent symbol›→0, recurse on remaining symbols

$\Sigma = \{a, b, c, d\}$

$f(a) = 0.26, f(b) = 0.255$

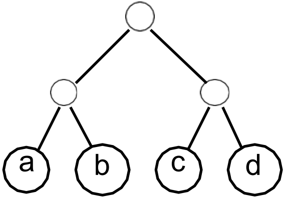
$f(c) = 0.245, f(d) = 0.24$



The cost of this tree is ≈ 2.2 but if we gave each symbol a code of length 2 the cost would be 2

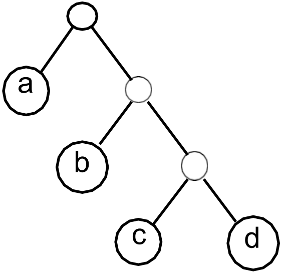
Attempt 2:

Ignore frequencies, give every symbol code of length $|\log(|\Sigma|)|$



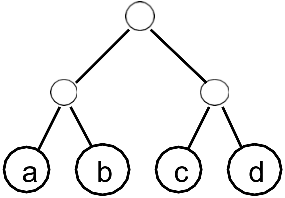
Attempt 1 (Greedy):
Most frequent symbol›→0, recurse on remaining symbols

$\Sigma = \{a, b, c, d\}$
$f(a) = 0.26, f(b) = 0.255$
$f(c) = 0.245, f(d) = 0.24$



The cost of this tree is $\approx$ 2.2 but if we gave each symbol a code of length 2 the cost would be 2

Attempt 2:
Ignore frequencies, give every symbol code of length $|\log(|\Sigma|)|$

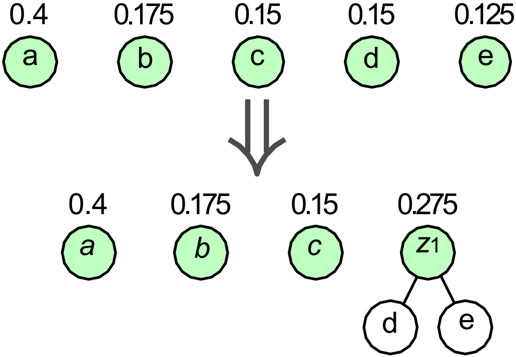


Consider: $f(a) = 1/2$, $f(b) = 1/4$, $f(c) = 1/8$, $f(d) = 1/8$.

Cost = 2 but attempt 1 is better (1.75)

Correct Algorithm

Idea: The two lowest frequency symbols *x* , *y* will be siblings so replace them with a new node *z* of frequency *f* (*x*) + *f* (*y*) and recursively solve the problem



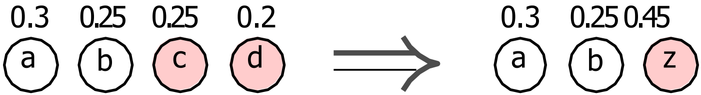
Penn
Engineering

ONLINE LEARNING

Video 2.10
Sampath Kannan

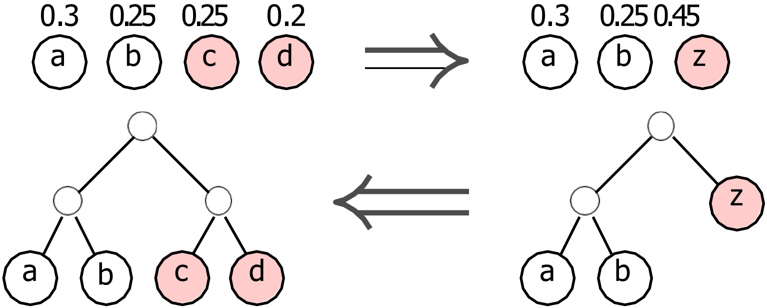
Huffman Coding

Replace two lowest frequency symbols x, y with a new symbol z of frequency $f(x) + f(y)$ and recurse on $n - 1$ symbols.



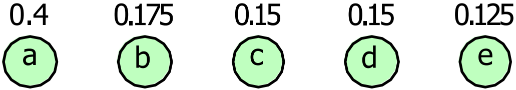
Huffman Coding

Replace two lowest frequency symbols x , y with a new symbol z of frequency $f(x) + f(y)$ and recurse on $n - 1$ symbols.

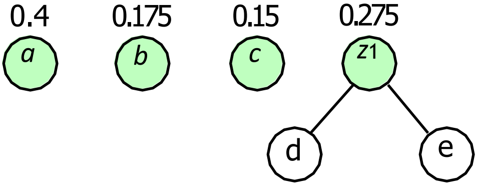


Take the resulting tree on $n - 1$ symbols and replace z with x and y

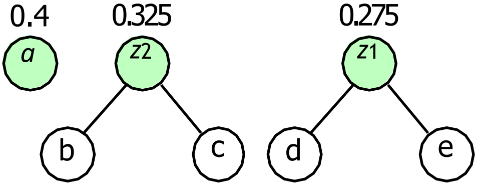
Example Huffman



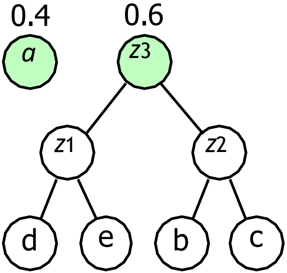
Example Huffman



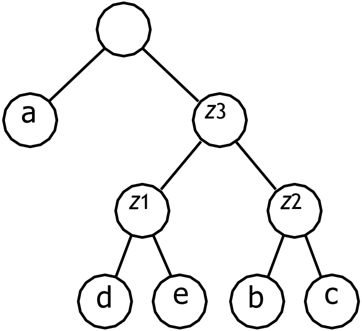
Example Huffman



Example Huffman



Example Huffman



Why does this work?

- Some two symbols are siblings at the largest depth

Why does this work?

- Some two symbols are siblings at the largest depth
- The algorithm we saw puts symbols with lowest frequencies there

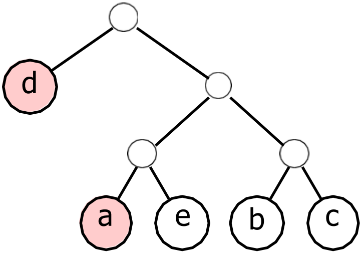
Why does this work?

- Some two symbols are siblings at the largest depth
- The algorithm we saw puts symbols with lowest frequencies there

- If an optimal tree existed with higher frequency symbols at the deepest level we could swap the lowest frequency symbol with one of the deepest ones and get a tree with a lower cost. This is contradiction since we started by saying the tree was optimal.

Example Swap

Recall the previous example where $f(a) = 0.4$ and $f(d) = 0.15$. Here is an encoding tree where one of the lowest frequency symbol (d) is swapped with a higher frequency one (a).

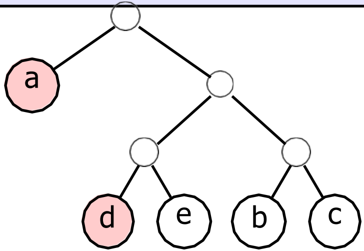


The string“. . . aadaadaadaa . . . ” gets encoded to:

100	100	0	100	100	0	100	100	0	100	100

Example Swap

See how the encoding length shrinks when we swap ’a’ and ’d’



“. .. aadaadaadaa .. .” is now encoded to:

0	0	100	0	0	100	0	0	100	0	0