Greedy Algorithms and Dynamic Programming.

Sep 17, 2020

Till now, we have seen problems that have some immediate polynomial time (e.g., $O(n^3)$, $O(n^4)$) algorithms and we saw some tools like divide and conquer to make them more efficient.

For example:

- $O(n^3) \rightarrow O(n \log n)$
- $O(n^3) \rightarrow O(n)$

Now, we will see more examples where the obvious algorithm takes exponential time and the challenge is to first design some polynomial time algorithm.

Towards this, the two paradigms Greedy & DP are often helpful.

They will involve more clever and subtle ideas than what we have seen till now.

We will study Greedy & DP simultaneously.

Pick up a problem and see whether Greedy works or not, whether DP works or not.

Greedy Algorithm

- local optimality
- near-sighted
- don’t care about long-run

Proof of correctness is important because correctness is not obvious in most cases.

We will see some basic format of how to argue correctness.
Dynamic Programming

→ Recursion with a better memory management.

or → a systematic approach to search through all possible solutions.

Problem 1: You are allowed to choose five apples from a basket. You want to maximize the total weight of your apples. Greedy approach works here.

→ Choose heaviest, 2nd heaviest, ..., 5th heaviest.

Problem 2: Total weight of the apples ≤ 2 kg. Basket has apples of weight

→ 450 gms, 400 gms.

Greedy: $4 \times 450 = 1800$ gms.
Alternate: $5 \times 400 = 2000$ gms.

Problem 3: Subsequence Problem.

$S_1 = abcabagbcac$ sequence

$S_2 = cabc$ (subsequence)

Given two sequences $S_1$ & $S_2$, whether $S_2$ is a subsequence of $S_1$.

$|S_1| = n \quad |S_2| = p$

Approach 1: search through all subsequences of length $p$ and check if one of them is equal to $S_2$. 

no. of such subsequences = \( \binom{n}{p} \)

Approach 2: match letter by letter

\[ S_1 = \text{bacbcabacbaa} \]
\[ S_2 = \text{bcba} \]

Match the current letter in \( S_2 \) with its first occurrence you see in \( S_1 \), after the previous matching.

Argument for correctness

\[ S_1 = \text{bacbcabacbaa} \]
\[ S_2 = \text{bcba} \]

We want to argue that the greedy approach works. That is, if \( S_2 \) is a subsequence of \( S_1 \), then we should be able to see it by matching each subsequent letter of \( S_2 \) with its first occurrence in \( S_1 \), after the previous matching.

To show that this is indeed true, start with an arbitrary subsequence of \( S_1 \) that matches with \( S_2 \). Now, move the matching of first letter of \( S_2 \) to its first occurrence in \( S_1 \). You still have the valid subsequence. Repeat the argument for every letter of \( S_2 \) one by one.
Dynamic Programming

Simple Example:

\[ F_n = F_{n-1} + F_{n-2} \quad F_0 = F_1 = 1 \]

Fibonacci \((n)\):

\[
\begin{align*}
\text{if } n = 0 & \quad \text{return } 1 \\
\text{if } n = 1 & \quad \text{return } 1 \\
\text{else } & \quad \text{return } \text{Fibonacci} \,(n-1) + \text{Fibonacci} \,(n-2)
\end{align*}
\]

\[ n \rightarrow n-1 \rightarrow n-2 \rightarrow n-3 \rightarrow \ldots \rightarrow 1 \rightarrow \sqrt{2} \]

At least \(2^{n/2}\) recursive calls

Bad Implementation.

Better Implementation

Array \(F\) of length \(n+1\).

\[
\begin{align*}
F[0] &= 1 \\
F[1] &= 1 \\
\text{for } i = 2 \text{ to } n \\
F[i] &= F[i-1] + F[i-2] \\
\text{end for}
\end{align*}
\]

Memoization

already stored in array
In dynamic programming, you reduce your problem to one or many subproblems. And if the same subproblem instance is used multiple times then you solve it only once and afterwards, keep using its stored solution. If the total number of distinct subproblem instances needed during the course of your algorithm is polynomially bounded then your algorithm is efficient.

For example, in the Fibonacci problem we had n distinct instances of the subproblem Fibonacci(n), Fibonacci(n-1), ..., Fibonacci(0).

Interval Scheduling. [Kleinberg Tardos] Chapter 4

Resource / server doing computation:

\[
\begin{array}{cccc}
\square & \square & \square & \square \\
\end{array}
\]

n requests:
\[(s_1, t_1), (s_2, t_2), \ldots, (s_n, t_n)\]

Maximize the number requests you can cater to constraints:
If you accept a request then you have to allocate the resource for the whole duration of desired interval (no partial allocation)

The resource can be allocated to only one person at a time.
Greedy Algorithms

→ locally optimal, near-sighted, immediate benefit

→ sometimes intuitively clear, sometimes not.

→ correctness of the algorithm is most often not obvious, and thus requires a concrete argument.

Classic Example: minimum spanning tree

Interval Scheduling: [Kleinberg Tardos Chapter 4]

Given a set of intervals (on real number line) find the largest subset of disjoint intervals.

Example: → (1, 5), (0, 3), (2, 4), (1, 6), (3, 4), (5, 7)

Application: resource allocation requests for fixed intervals of time

• resource can be allocated to only one request at a time
• no partial allocation
• cater to maximum number of requests.

Searching through all possible solutions \( \cong 2^n \) solutions
Greedy Strategy 1: keep choosing the smallest interval (and removing those which intersect with chosen ones)

---

Greedy Strategy 2: earliest starting time

---

Greedy Strategy 3: keep choosing the interval with smallest no. of overlaps.

---

Greedy Strategy 4: keep choosing the interval with earliest ending time
Intuitively, you are maximizing the remaining time, thus maximizing the possible number of requests catered afterwards.

Claim: Greedy strategy 4 always gives an optimal solution.

General Framework of argument:

1. Show that there exists an optimal solution that agrees with the first greedy step.
2. The rest of the argument will work inductively.

Consider an optimal solution \( I_1, I_2, I_3, \ldots, I_k \).

Earliest ending interval \( I_0 \):

\[
\begin{align*}
&I_0 \quad I_1 \quad I_2 \quad \ldots \quad I_k \\
&\uparrow \\
\end{align*}
\]

Swap \( I_0 \) with \( I_1 \).

Consider a new solution \( I_0, I_2, I_3, \ldots, I_k \).

Claim It is a valid solution.

\[ \text{endtime}(I_0) < \text{endtime}(I_1) \leq \text{starttime}(I_2) \]

\[ \Rightarrow I_0 \text{ does not overlap with } I_2, I_3, \ldots, I_k \]

\( I_0, I_2, I_3, \ldots, I_k \) is an optimal solution agreeing with the first greedy step.
Inductive proof based on the number of intervals.

**Inductive Hypothesis:** Greedy algorithm works for any instance with up to \( n-1 \) intervals.

**Inductive Step:** It works for all instances with \( n \) intervals.

**Base Case:** \( n = 1 \). Obvious.

**Input:** \( I = \{I_1, I_2, I_3, \ldots, I_n\} \)

for convenience

assume \( \text{end time}\ (I_1) \leq \text{end time}\ (I_2) \leq \ldots \)

Algorithm chooses \( I_1 \)
then removes all intervals overlapping with \( I_1 \)

\( \rightarrow I' = I - \{\text{intervals overlapping with } I_1\} \)

Recursively applying the same algorithm on \( I' \).

By the inductive hypothesis, we know that
the algorithm gives optimal solution for \( I' \)

That is, \[ \text{Algo} (I') = \text{OPT} (I') \]

**Output:** \( I_1 + \text{Algo}(I') = I_1 + \text{OPT}(I') \)

**Claim:** \( I_1 + \text{any optimal solution for } I' \)

is an optimal solution for \( I \).
Claim: \( I_1 + \text{OPT}(X') \) is an optimal solution for \( X \).

Proof: Recall that we showed that there exist an optimal solution for \( X \) that contains \( I_1 \).

Let it be \( I_1, J_1, J_2, \ldots, J_k \).

Clearly \( J_1, J_2, \ldots, J_k \) are disjoint from \( I_1 \).

Hence, \( \{ J_1, J_2, \ldots, J_k \} \) is a valid solution for \( X' \).

\[ \Rightarrow \text{OPT}(X') \geq \{ J_1, J_2, \ldots, J_k \} \]

\[ \Rightarrow I_1 + \text{OPT}(X') \geq \{ I_1, J_1, J_2, \ldots, J_k \} \]

Optimal for \( X \).

\[ = \text{OPT}(X) \]

Pseudocode

Input: \((s_1, f_1), (s_2, f_2), \ldots, (s_n, f_n)\)

- Sort according to \( f_j \).

\( f = -\infty \) (finish time of latest interval selected so far)

for \( (i = 1 \text{ to } n) \)

\( \rightarrow \) if \( (s_i > f) \) then

select \((s_i, f_i)\)

\( f \leftarrow f_i \)
HW Assignments

deadlines \( d_1, d_2, \ldots, d_n \)
time required \( l_1, l_2, \ldots, l_n \)

Lateness of \( i \)-th assignment = \((t_i - d_i)\) if \( t_i > d_i \)

Minimize maximum lateness over all assignments.

\( \rightarrow \) smallest length first?
\( \rightarrow \) earliest deadline first?
\( \rightarrow \) minimum \( d_i - l_i \) first?

Example:

<table>
<thead>
<tr>
<th></th>
<th>( A_1 )</th>
<th>( A_2 )</th>
<th>( A_3 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>length</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>deadlines</td>
<td>2</td>
<td>4</td>
<td>6</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>( A_1 )</th>
<th>( A_2 )</th>
<th>( A_3 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lateness</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>Lateness</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Lateness</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Orders:
- \( A_3 A_2 A_1 \)
- \( A_2 A_1 A_3 \)
- \( A_1 A_2 A_3 \)

Red shows maximum lateness. Best order is \( A_1 A_2 A_3 \).

\( \Rightarrow \) Interval scheduling: cater to all requests with using minimum number of servers.
\( \{ \text{one server can cater to one request at a time} \} \)

Equivalently, finding the minimum number of platforms required for a set of trains stopping at a station.
Interval Scheduling: another variant

- select a set of disjoint intervals to maximize the total length of selected intervals.

For example, there might be a profit proportional to the duration of use.

Greedy Strategy 1: keep picking the longest length intervals

Greedy Strategy 2: earliest starting time
Dynamic Programming:

Recursion with better memory management.

General Idea: try to categorize the possible solutions into different types. For each type, find the optimal solution via recursion. Then compare the various types of optimal solutions with each other.

Interval scheduling with maximum total length:

\[ I = \{ I_1, I_2, I_3, \ldots, I_n \} \]

Two kinds of solutions:

- Contain \( I_1 \)
- Don't contain \( I_1 \)

Can we find optimal from both the kinds recursively?

\[ I' = I - \text{overlap}(I_1) \]

\[ \text{OPT}(I') + I_1 \]

\[ I'' = I - I_1 \]

\[ \text{OPT}(I'') \]
\[ \text{OPT}(X) = \max \left\{ I_1 + \text{OPT}(X'), \text{OPT}(X'') \right\} \]

Example

\[ I = \{(0,5),(1,8),(5,7),(4,9),(3,4)\} \]

\[ I_1 = (0,5) \]

\[ I' = \{(5,7)\} \]

\[ I'' = \{(1,8),(5,7),(4,9),(3,4)\} \]

\[ I \rightarrow \{(1,8),(5,7),(4,9),(3,4)\} \]

\[ \rightarrow \{(5,7),(4,9),(3,4)\} \]

\[ \rightarrow \{(3,4)\} \rightarrow \{(4,9),(3,4)\} \]

\[ \rightarrow \{(3,4)\} \]

Nothing clever here. We are simply trying to go over all possible solutions recursively.

No. of recursive calls seems to be growing exponentially because each call makes two new recursive calls.

Efficient implementation possible if no. of total distinct recursive calls is small.
HW work out the recursion tree and figure out whether the number of recursive calls is growing exponentially or polynomially

\[ \mathcal{I} = \left\{ \begin{array}{l} (1,3) \quad (11,13) \quad (21,23) \quad (31,33) \\ I_1 \quad I_2 \quad I_3 \quad I_4 \\ (2,4) \quad (12,14) \quad (22,24) \quad (32,34) \\ I_5 \quad I_6 \quad I_7 \quad I_8 \end{array} \right\} \]

If we have 2n intervals, we will have at least \(2^n\) distinct recursive calls or subproblems.
\[ \mathcal{I} = \{ (1,3), (2,4), (11,13), (12,14), (21,23), (22,24), (31,33), (32,34) \} \]

Only 2n distinct recursive calls or subproblems.

It seems if the intervals are arranged in a particular order, the number of distinct subproblems will be small.

Possible orders:
1. starting time
2. ending time
sort the intervals in increasing order of starting time

\[ I_1, I_2, I_3, \ldots, I_n \]

Claim: The input set of intervals for any recursive call will look like

\[ \{ I_j, I_{j+1}, I_{j+2}, \ldots, I_n \} \]

(as opposed to an arbitrary subset of intervals)

This is because when we remove intervals overlapping with \( I_k \), the remaining set is simply all intervals with starting time > end-time(\( I_k \)).

No. of distinct subproblems \( \leq n \).
Recursive Implementation

\( \text{Opt} \leftarrow \text{array with all zeros.} \)
\( \text{Opt}[n] \leftarrow \text{length}(I_n) \)

Output \( \text{ALG}(i) \);

\text{ALG}(j): \quad \text{// ALG}(j) \text{ computes optimal solution for } \{I_j, I_{j+1}, \ldots, I_n\}

if \text{Opt}[j] > 0 \quad \text{return } \text{Opt}[j]
\quad \text{// means already solved and stored.}

else

\( \text{Opt}[j] \leftarrow \max \left\{ \text{ALG}(j+1), \text{length}(I_j) + \text{ALG}(p(j)) \right\} \)

return \text{Opt}[j]

Iterative Implementation:

\( \text{Opt} \leftarrow \text{array with all zeros.} \)
\( \text{Opt}[n] \leftarrow \text{length}(I_n) \)

for \((j = n-1 \text{ to } 1)\)

\( \text{Opt}[j] \leftarrow \max \left\{ \text{Opt}[j+1], \text{length}(I_j) + \text{Opt}[p(j)] \right\} \)
Add code to compute the optimal set of intervals.

Conclusion: Order of processing the input is important.

Intervals: Order of starting time/ending time

Sequences: left to right

Try to ensure that no. of distinct subproblems is small.

Maximum travel in a day - d.

Night stay prices - P_1, P_2, ...

Minimize total stay cost during the journey.

\( O(n) \)

No. of distinct subproblems = \( n \).

\( \text{HW 1} \)

<table>
<thead>
<tr>
<th>A</th>
<th>c_1</th>
<th>c_2</th>
<th>c_3</th>
<th>c_4</th>
<th>c_5</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>d</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

No. of distinct subproblems = \( n \).

\( \text{HW 2} \)

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

\( A = b a c \ b c \ b a c \ a b c \ a c b a \)

\( B = ' b c b c a ' \)

Match B inside A with minimum cost.

\( O(\text{mn}) \)
Subset sum problem

Given set of integers \( A = \{ a_1, a_2, a_3, \ldots, a_n \} \)

is there a subset with sum zero?

Example:
\[
2, -5, 1, 7, -4, -6 \quad \{2, 7, -5, -4\}
\]

Solutions containing an
not containing an

is there a subset of \( \{2, -5, 1, 7, -4\} \) with sum zero?

is there a subset of \( \{2, -5, 1, 7, -4\} \) with sum six?

Subproblem \( A_j = \{a_1, \ldots, a_j\} \), number \( N \)
is there a subset of \( A_j \) with sum \( N \).

No. of distinct subproblems = \( n \times \sum_{i} |a_i| \)

ALG \( (j, N) \):
\[
\rightarrow \text{ALG} (j-1, N) \quad \text{OR} \quad \text{ALG} (j-1, N-a_j)
\]
Pseudo polynomial time.

A polynomial time algorithm is supposed to take
time \( \text{poly}(n, \text{no. of bits in } a_1, a_2, \ldots) \).
**Subset Sum Pseudocode**

\( S \leftarrow \text{Boolean two-dim array of size } N \times (\sum |a_i| + 1) \)

\( S[i, k] \) will denote whether there is a subset of first \( j \) numbers with sum = \( k \).

Say, range of \( j \) is \( 1 \leq j \leq n \).

And range of \( k \) is \( \text{neg} \leq k \leq \text{pos} \).

\[
\text{Sum of all negative numbers} \quad \text{Sum of all positive numbers}
\]

**Initialization:**

\[
S[1, k] \leftarrow \begin{cases} 
\text{True} & \text{for } k = a_i \\
\text{False} & \text{for all other values of } k.
\end{cases}
\]

for ( \( j = 2 \) to \( n \) )

for ( \( \text{neg} \leq k \leq \text{pos} \) )

\[
S[j, k] \leftarrow S[j-1, k] \text{ OR } S[j-1, k-a_j]
\]

To construct the desired set

Assume \( S[n, 0] \) is true.

\[
\text{sum} \leftarrow 0
\]

for ( \( j = n \) to 1 )

if \( S[j-1, \text{sum}] = \text{true} \), don't take \( a_j \)

else if \( S[j-1, \text{sum} - a_j] = \text{true} \), take \( a_j \) and

\[
\text{sum} \leftarrow \text{sum} - a_j
\]
Knapsack problem

\[ n \text{ objects } \text{weights } w_1, w_2, \ldots, w_n \]

\[ \text{Values } v_1, v_2, \ldots, v_n \]

Select a subset \( S \) s.t.

\[ \sum_{i \in S} w_i \leq W \]

and the total value \( \sum_{i \in S} v_i \) is maximized.

Balanced margins

\[ W = 68 \]

average slack \( 14/3 \approx 4.6 \)

Suppose there are ten tourist guides, of which, six can speak French and six can speak German (two can speak both French and German). Everyone comes with their own charges. We want to select five.

Suppose there are ten tourist guides, of which, six can speak French and six can speak German (two can speak both French and German). Everyone comes with their own charges. We want to select five.

We are given a sequence of words of lengths

\[ w_1, w_2, w_3 \ldots w_n \]

Each line can have at most \( W \) characters.

If a line has from the \( i \)-th to \( j \)-th word then its slack is defined to be

\[ s = W - [ (w_i + 1) + (w_{i+1} + 1) + \cdots + (w_{j-1} + 1) + w_j ] \]
\[ l_1, l_2, \ldots, l_k \]

\[ \text{Variance} = \text{avg} (l_1^2, l_2^2, \ldots, l_k^2) - \left( \text{avg} (l_1, l_2, \ldots, l_k) \right)^2 \]

Arrange the words in lines to minimize the sum of squares of the slacks of all lines.

**Puzzle** find integers \( a, b, c \) such that

\[ a + b + c = 14 \]

and \( a^2 + b^2 + c^2 \) is minimized.

**Answer:**

What do you observe?

**Balanced Margins**

**Idea 1:** fit as many words as you can

\[ \rightarrow \text{can be very unbalanced} \]

**Greedy Idea:**

compute the average slack per line. Go line by line and try to keep the slack for each line as close as possible to the average slack.

\[ \frac{15}{3}, \frac{5}{5}, \frac{2}{4} \]

\[ 13, 5, 4, 5, 51, 45 \]

\[ \frac{11}{3}, 3.66 \]
Dynamic Programming.

- Try to categorize the set of all possible solutions.

Each solution is a partition of words into \( k \) lines:

\[
\eta = \eta_1 + \eta_2 + \eta_3 + \cdots + \eta_k
\]

Categories:
\[
\eta_k = 1 \quad \text{last line has 1 word.}
\]
\[
\eta_k = 2 \quad \text{last line has 2 words.}
\]
\[
\eta_k = \eta - 1 \quad \text{last line has \( \eta - 1 \) words.}
\]

Assuming last line has words \( w_{p+1}, \ldots, w_n \) (i.e. \( n-p \) words), can you compute the optimal solution via a subproblem?

\[
\text{OPT}(w_1, w_2, \ldots, w_p) + \text{slack}(w_{p+1}, w_{p+2}, \ldots, w_n)^2
\]

\[
\text{OPT}(n) = \min \left\{ \begin{array}{l}
\text{OPT}(n-1) + [W - w_n]^2 \\
\text{OPT}(n-2) + [W - w_n - w_{n-1} - 1]^2 \\
\text{OPT}(n-3) + [W - w_n - w_{n-1} - w_{n-2} - 2]^2 \\
\end{array} \right.
\]

Running time \( O(n^2) \)
Pseudocode for computing the optimal value and the optimal solution.

\[ S \leftarrow \text{array of length } n+1 \]
\[ N \leftarrow \text{array of length } n+1 \]

// \( S[j] \) denotes the minimum sum of squares of slacks for first \( j \) words, for \( 1 \leq j \leq n \)

// \( N[j] \) denotes the index of the first word in the last line in the optimal arrangement of first \( j \) words.

\[ S[0] \leftarrow 0 \; ; \; S[j] \leftarrow \infty \; \text{for } j > 0 \; ; \; N[0] \leftarrow 0 \]

\begin{verbatim}
for (j = 1 to n)
    for (r = j to 1)
        Slack \leftarrow W - (w_r + w_{r+1} + \ldots + w_j + j-r)
        if (Slack \geq 0 \text{ and } S[j] > S[r-1] + (Slack)^2)
            S[j] \leftarrow S[r-1] + (Slack)^2
            N[j] \leftarrow r
\end{verbatim}

Optimal Arrangement:
Run \text{Arrange}(n). 
\text{Arrange}(j) \{ 
    Arrange(N(j)-1)
    print words from \( N(j) \) to \( j \) in a new line.
\}
Optimal Binary Search Tree

Suppose we know how frequent are the search queries for each of the elements.

Say frequencies

\[
\begin{align*}
a_1 & \rightarrow 7 \\
a_2 & \rightarrow 5 \\
a_3 & \rightarrow 4 \\
a_4 & \rightarrow 6 \\
\end{align*}
\]

Total search cost

First binary tree 43

Second binary tree 47

Compute the binary search tree with minimum total search cost \(\sum_i f_i h_i\)

where \(f_i\) → frequency \((a_i)\) \(h_i\) → depth \((a_i)\)

Greedy Idea 1:

maximum frequency element → root
Dynamic Programming \[ a_1 < a_2 < a_3 ... < a_n \]

Possible solutions: every valid binary search tree.

Categories:
- \( \text{Root} \leftarrow a_1 \)
- \( \text{Root} \leftarrow a_2 \)
- \( \text{Root} \leftarrow a_n \)

\[
\text{Tree}(a_1, \ldots, a_n) = \min_{1 \leq i \leq n} \left\{ \text{Tree}(a_1, \ldots, a_{i-1}) + \text{Tree}(a_{i+1}, \ldots, a_n) + \sum_{j=1}^{n} f_j \cdot 1 \right\}
\]

How many distinct subproblems = \( \binom{2n}{2} \)

\[
\text{Tree}(i, j) = \min_{k} \left\{ \text{Tree}(i, k-1) + \text{Tree}(k+1, j) \right\} + \sum_{k=i}^{j} f_k
\]

\( j > i \)
OPT ← 2 dim array.

$O(n^3)$

$OPT[i, i] ← f_i$ for each $i$

for ($i = n-1$ to 1)

for ($j = i+1$ to $n$)

$$OPT[i, j] = \sum_{k=1}^{j} f_k + \min_{k: i \leq k \leq j} \{OPT[i, k-1] + OPT[k+1, j]\}$$

Optimal cost for $(a_i, a_{i+1}, \ldots, a_j)$

Add/modify code to compute the optimal solution.
Sequence Alignment

Computational biology, spell checking

Similarity between two strings

Needleman and Wunsch defined a notion of similarity

```
Example
UGCTGACU
→ GAATGCA
```

Given

- Gap Penalty \( \delta \)
- Mismatch cost (for each pair) \( \alpha_{xy} \)

Total cost = Sum of the gap and mismatch costs.

Find an alignment of the two strings with minimum total cost.

Input: \( x_1, x_2, \ldots, x_m \) and \( y_1, y_2, \ldots, y_n \)

\( \delta, \{\alpha_{xy}\} \)

Categories of solutions:

1. \[ \ldots \quad \vdots \quad \vdots \quad \cdots \quad \vdots \quad \ldots \]
   \[ \vdots \quad \ddots \quad \ddots \quad \cdots \quad \ddots \quad \vdots \]

2. \[ \ldots \quad \vdots \quad \vdots \quad \cdots \quad \vdots \quad \ldots \]
   \[ \vdots \quad \ddots \quad \ddots \quad \cdots \quad \ddots \quad \vdots \]

3. \[ \ldots \quad \vdots \quad \vdots \quad \cdots \quad \vdots \quad \ldots \]
   \[ \vdots \quad \ddots \quad \ddots \quad \cdots \quad \ddots \quad \vdots \]
\[ \text{OPT}(m, n) = \begin{cases} \alpha x_m y_n + \text{OPT}(m-1, n-1) \\ \min \left\{ \delta + \text{OPT}(m-1, n) \right\} \\ \delta + \text{OPT}(m, n-1) \end{cases} \]

\[ \text{OPT}(i, j) \quad \text{no. of distinct subproblems: } m \times n \]

**Implementation**

1. \( A \leftarrow 2D \text{ array } m \times n \)
2. // \( A[i, j] \) denotes the minimum cost of alignment for \( x_1 x_2 \cdots x_i \) and \( y_1 y_2 \cdots y_j \)
   
   \( A[i, 0] \leftarrow \delta_i \quad \text{for each } i \)
   
   \( A[0, j] \leftarrow \delta_j \quad \text{for each } j \)

   for (i = 1 to n)
   
   for (j = 1 to n)

   \[ A[i, j] = \min \left\{ \alpha x_i y_j + A[i-1, j-1], \right\} \]
   
   \[ \delta + A[i-1, j], \]
   
   \[ \delta + A[i, j-1] \]

**HW**

Space \( O(mn) \)

Can you get space \( O(m+n) \) and time \( O(mn) \) \{ Divide and Conquer \}
Summarizing Greedy and Dynamic Programming

- Dividing the set of possible solutions into multiple categories.

Greedy: there must be an optimal solution in a certain category $C^*$ (greedy choice)

DP: will take best of the optimal solutions from each category.

- To compute the optimal solution from a chosen category $\rightarrow$ smaller subproblem (Recursion)

No. of distinct subproblems should be small.
Data Compression: Coding

assign a fixed length 0-1 string to each character.

\[ a \rightarrow 00001 \]
\[ b \rightarrow 00010 \]
\[ \vdots \]

5 bit encoding can work for up to 32 characters.

Is a smaller length encoding possible?

With fixed length - not possible

Variable length encoding

- can be more efficient when no. of characters is not \(2^k\)
- can use smaller length codes for more frequent characters.

Example: Morse Code (dots and dashes and spaces)

\[ e \rightarrow \cdot \]
\[ t \rightarrow - \]
\[ a \rightarrow \_\_ \]
\[ \vdots \]
\[ z \rightarrow \ldots \]
\[ q \rightarrow \ldots \]

Problem \[ \ldots \] \[ \leftrightarrow \]
\[ a \_a \]
\[ e \_t \_a \]
\[ a \_e \_t \_e \_t \_e \]

Solution: Gap after every character
**Prefix Code**

**Def:** For any two different characters \( x \) and \( y \), \( C(x) \) should not be a prefix of \( C(y) \).

\[
\begin{align*}
x & \rightarrow 00 \quad \text{Not a prefix code} \\
y & \rightarrow 001 \\
\end{align*}
\]

**Ex**

\[
\begin{align*}
a & \rightarrow 0 \\
b & \rightarrow 10 \\
c & \rightarrow 11 \\
\end{align*}
\]

Prefix code.

\[
\text{abaca} \rightarrow 0100110
\]

**Claim:** For a prefix code, any 0-1 string is unambiguously decodable.

Just scan left to right, as soon as the current substring matches one of the codewords, output the corresponding character.

---

**Optimal Prefix Codes**

<table>
<thead>
<tr>
<th>Freq</th>
<th>Code 1</th>
<th>Code 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.4</td>
<td>A</td>
<td>00</td>
</tr>
<tr>
<td>0.4</td>
<td>T</td>
<td>01</td>
</tr>
<tr>
<td>0.1</td>
<td>P</td>
<td>10</td>
</tr>
<tr>
<td>0.1</td>
<td>G</td>
<td>11</td>
</tr>
</tbody>
</table>

\[
\frac{2}{100} + 0.4 \times \frac{2}{180} + 0.4 \times \frac{2}{180} + 0.1 \times \frac{3}{180} + 0.1 \times \frac{3}{180} = 1.8
\]

**Avg Bit length**
Prob  Given frequencies $f_1, f_2, \ldots, f_n$ for $n$ characters find a prefix code that minimizes $\sum_i f_i \ell_i$.

$(\sum f_i = 1)$

length of the encoding

avg encoding length for $i$-th character

Observation: Each prefix code corresponds to a binary tree.

codewords correspond to leaves of the tree.

Approach 1: assign 0 to highest frequency.

Approach 2: assign 0 to highest freq if it is above some threshold.

Approach 3: Do a balanced division of frequencies into two parts.

$0.32 \ 0.25 \ 0.18 \ 0.25$
Observations:

\[
\begin{align*}
\text{low frequency} & \rightarrow \text{higher length} \\
\text{high frequency} & \rightarrow \text{lower length}
\end{align*}
\]

Approach 1: for highest frequency character, assign ‘0’ i.e. length one codeword.

\[
(0.25, 0.25, 0.25, 0.25)
\]

\[
0.25 \times 1 + 0.25 \times 1 + 0.25 \times 1 + 0.25 \times 1 = 1.0
\]

\[
(0.28, 0.24, 0.24, 0.24)
\]

\[
2 \times 0.28 + 1 \times 0.24 + 1 \times 0.24 + 1 \times 0.24 = 2.2
\]

\[
(0.37, 0.21, 0.21, 0.21)
\]

\[
2 \times 0.37 + 1 \times 0.21 + 1 \times 0.21 + 1 \times 0.21 = 2.05
\]

\[
(0.35, 0.35, 0.15, 0.15)
\]

\[
2 \times 0.35 + 0.35 \times 0.15 + 1 \times 0.15 = 1.95
\]

Assign '0' if frequency higher than certain threshold.

Doesn't work.

Cannot decide 1, just by looking at \( f_1 \).
Suppose there is some way to decide the encoding length for highest frequency character. Say length 2. ‘00’

0.3 0.25, 0.2, 0.15, 0.1

can we reduce the rest of the encodings to a subproblem?

Not able to frame it as a smaller instance of the same problem.

→ Suppose we can fix the length for the least frequent character.

0.3, 0.25, 0.2, 0.15, 0.1

same issue.
Shannon and Fano (1940's)

Balanced Partition

Divide the list into two parts such that the total frequency of each part is as close as possible to 0.5.

0.35 0.35 0.20 0.10

0.45

0.55

0.35 x 1
+ 0.35 x 2
+ 0.2 x 3
+ 0.1 x 3

= 1.95

By balanced partition approach we are getting average encoding length = 2. But there is a better solution with 1.95.

Try various possibilities for the partition?

Exponentially many possibilities.
**Obs 1:** If you fix a binary tree, then there is a natural way to characters to leaves.

\[0.35, 0.35, 0.2, 0.1\]

Lowest frequency character has the largest depth.

**Obs 2:** The leaf with the largest depth must have a sibling which is a leaf.

\[f_1 > f_2 > ... > f_{n-1} > f_n\]

**Claim:** The lowest and second lowest frequency characters can be mapped to two largest depth siblings.
Now, the rest of the tree can found recursively.

\[ f_1, f_2, f_3, f_4, f_5 \quad \text{(in decreasing order)} \]

\[
\begin{align*}
\text{Cost}_0: & \quad 2f_1 + 2f_2 + 2f_3 + 3f_4 + 3f_5 \\
\text{Cost}_1: & \quad 2f_1 + 2f_2 + 2f_3 + 2f_4 + 2f_5 \\
\text{Cost}_0 = \text{Cost}_1 + f_4 + f_5
\end{align*}
\]

For any binary tree with \( n \) leaves where \( f_{n-1} \) and \( f_n \) are siblings, there is a corresponding binary tree with \( n-1 \) leaves with labelings:

\[ f_1, f_2, f_3, \ldots, f_{n-2}, f_{n-1} + f_n \]

**Algorithm:**

- **Input:** \( f_1 \geq f_2 \geq \ldots \geq f_{n-1} \geq f_n \) (\( n \) char)

  Recursively compute optimal binary tree for

  \[ f_1, f_2, \ldots, f_{n-2}, f_{n-1} + f_n \] (\( n-1 \) char)

  For the leaf labeled \( f_{n-1} + f_n \), add two children with label \( f_{n-1} \), \( f_n \).
Example

0.3, 0.25, 0.18, 0.15, 0.12

0.3, 0.25, 0.18, 0.27

0.3, 0.27, 0.43

0.43, 0.57

\[ \sum f_i l_i \]
Proof of Correctness

1. There is an optimal solution where $f_{n-1}$ and $f_n$ are siblings.

2. There is a one-to-one correspondence between

   $A = \{ \text{full binary trees with } n \text{ leaves labeled } f_1, f_2, \ldots, f_n \}$

   where $f_n$ and $f_{n-1}$ are siblings

   $B = \{ \text{full binary trees with } n-1 \text{ leaves labeled } f_1, f_2, f_3, \ldots, f_{n-2}, f_{n-1} + f_n \}$

Say, $A = \{ T_1, T_2, T_3, \ldots, T_N \}$

$B = \{ R_1, R_2, R_3, \ldots, R_N \}$

Remove leaves labeled $f_{n-1}, f_n$.

Label their parent $f_{n-1} + f_n$.

For the leaf labeled $f_{n-1} + f_n$, add two children labeled $f_{n-1}, f_n$.

Moreover $\text{cost}(T_j) = \text{cost}(R_j) + f_{n-1} + f_n$

Thus, $R_j$ is optimal in $B$ $\iff$ $T_j$ is optimal in $A$. 
Huffman Codes:
Text over some large alphabet size \( \rightarrow \) space efficient bit representation

* Que: Can we apply this technique when the data is already in 0/1 bit representation?

Suppose there is data where 0 is much more frequent than 1.

```
0 1 0 0 0 0 1 1 0 1 0 0 0 0 1 1 0 0 1 1 0 0 1 0
```

\( a \rightarrow 000 \rightarrow 0 \)

\( b \rightarrow 001 \rightarrow 100 \)

\( e \rightarrow \ldots \)

\( h \rightarrow 111 \rightarrow 1101 \)

- There are many other data compression techniques
  - Adaptive
  - Algebraic

```
               0.35
                / \
        0.3      0.2
               |  \
          0.15
               \
```

```
               0.35
                / \
        0.3      0.3
               |  \
          0.35
               \
```

```
               0.2
                / \
        0.15 0.35
               |    |  \
          0.3 0.3
               \
```