Greedy Algorithms and Dynamic Programming.

Sep 17, 2020

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

For example $O(n^2) \longrightarrow O(n \log n)$

$O(n^3) \longrightarrow O(n)$

⋮

Now, we will see more examples where the obvious algorithm takes exponential time and the challange 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 then 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 $\boxed{\text{five}}$ apples from a basket. You want to maximize the total weight of your apples.

Greedy approach works here.

→ Choose heaviest, 2^{nd} heaviest, ..., 5^{th} heaviest

Problem 2 Total weight of the apples $\leq$ 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 = a c b g a b a g b c a sequence

S_2 = c a b g c (subsequence)

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

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

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

$S_1 =$ b a c b c b a b c a c b a a

$S_2 =$ b c b c a

Match the current letter in S_2 with its first occurence you see in S_1 after the previous matching.

<u>Argument for correctness</u>

S_1 b a c b c b a b c a c b a a

S_2 b c b c a

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 occurence in S_1 after the previous matching.

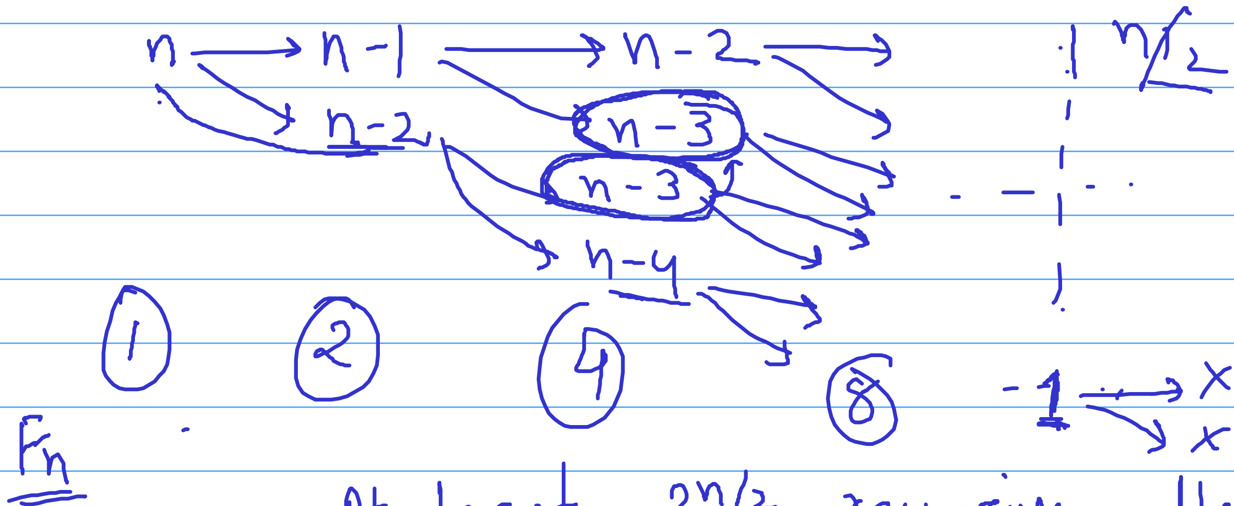
To show that this is indeed true, start with an arbitrary subsequence of S_1 that matches with S_2. Now, move the matching of first letter of S_2 to its first occurence in S_1. You still have the valid subsequence. Repeat the argument for every letter of S_2 one by one.

Dynamic Programming

Simple Example:

$$F_n = F_{n-1} + F_{n-2} \qquad F_0 = F_1 = 1$$

```
Fibonacci(n):
        if n=0   return 1
        if n=1   return 1
    else return  Fibonacci(n-1) + Fibonacci(n-2)
```



At least $2^{n/2}$ recursive calls

Bad Implementation.

Better Implementation

Array F of length $n+1$.

$F[0] = 1$

$F[1] = 1$

Memoization.

Bottom-up

```
for i=2 to n
    F[i] = F[i-1] + F[i-2].
end for.
```

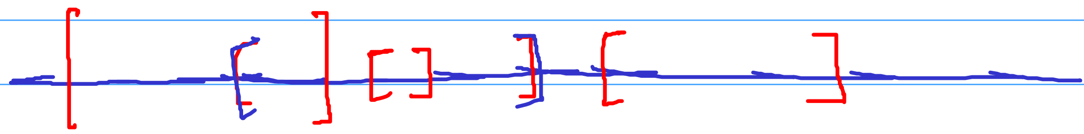
$\underbrace{F[i-1]}\ \underbrace{F[i-2]}$ already stored in array.

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(1)

Interval Scheduling. [Kleinberg Tardos Chapter 4]

Resource / server doing computation.



n requests.
(s_1, t_1) (s_2, t_2), ..., (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

Aug 30

→ locally optimal, near-sighted, immediate benefit

→ sometimes intuitively clear, sometimes not.

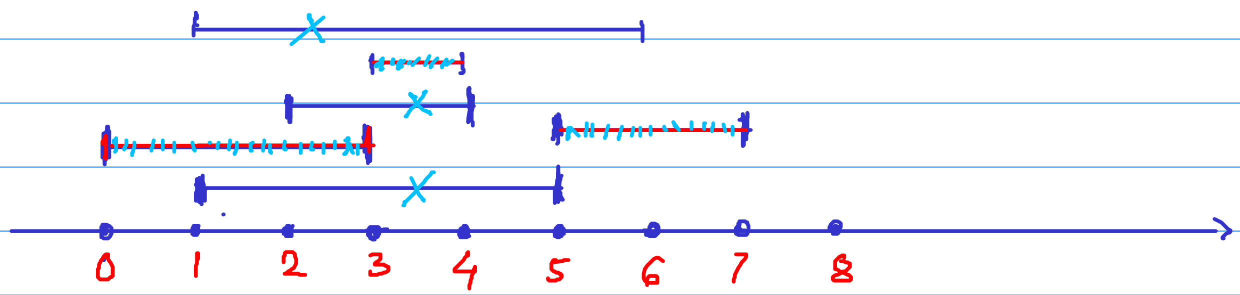
→ Correctness of the algorithm is most often not obvious, and thus requires a concrete argument

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)

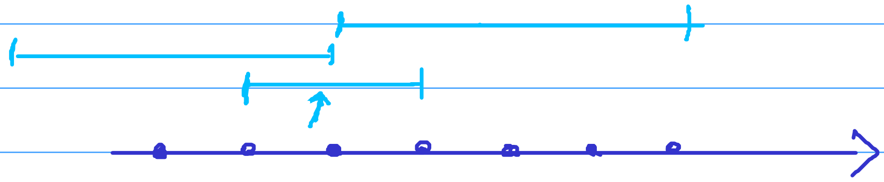


Application: resource allocation requests for fixed intervals of time

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

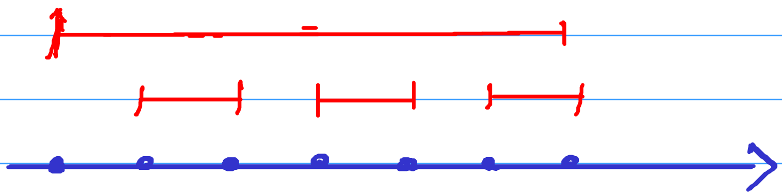
Searching through all possible solutions $\approx 2^n$ solutions

Greedy Strategy 1: keep choosing the smallest interval (and removing those which intersect with chosen ones)



Greedy 1
Optimal 2.

Greedy Strategy 2 : earliest starting time

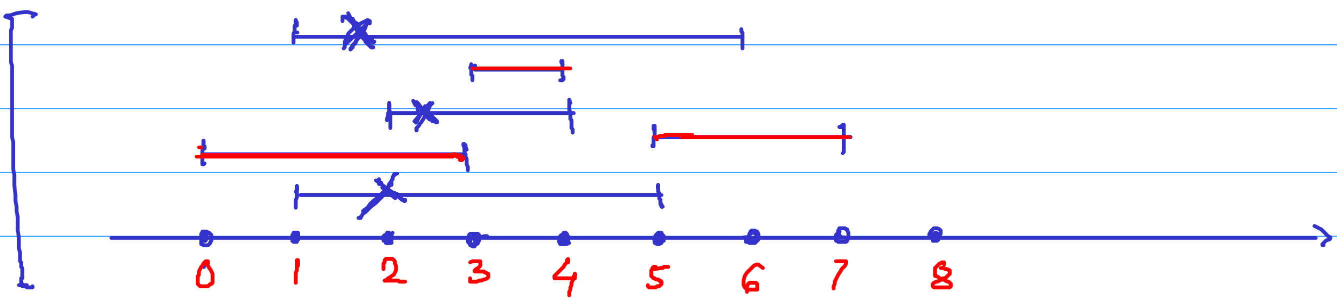


Greedy 1
Optimal 3

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

HW

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.

<u>Claim:</u> Greedy strategy 4 always gives an optimal solution

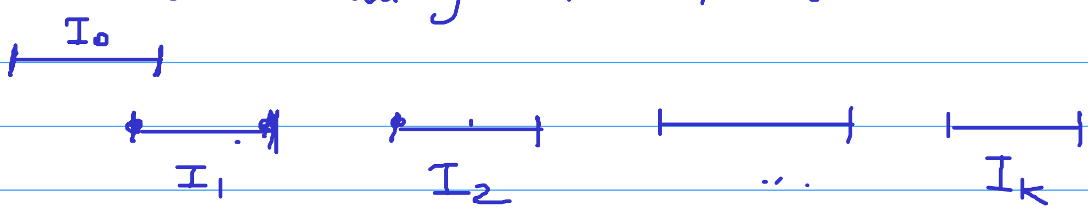
General Framework of argument:

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



swap I_0 with I_1.

Consider a new solution $I_0, I_2, I_3, \ldots, I_k$

<u>Claim</u> It is valid solution.

$$\text{endtime}(I_0) \leq \text{endtime}(I_1) \leq \text{starttime}(I_2)$$

$\Rightarrow$ I_0 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 first greedy step.

Inductive proof based on the number of intervals.

Inductive Hypothesis: Greedy algo 4 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: $\mathcal{I} = \{I_1, I_2, I_3, \cdots, I_n\}$

for convenience assume end time $(I_1) \le$ end time $(I_2) \le \cdots$

Algorithm chooses I_1. then removes all intervals overlapping with I_1

$\rightarrow \mathcal{I}' = \mathcal{I} - \{\text{intervals overlapping with } I_1\}$

Recursively applying the same algorithm on $\mathcal{I}'$

By the inductive hypothesis, we know that the algorithm gives optimal solution for $\mathcal{I}'$

That is, $\boxed{\text{Algo}(\mathcal{I}') = \text{OPT}(\mathcal{I}')}$

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

Claim: $(I_1 + \text{any optimal solution for } \mathcal{I}')$ is an optimal solution for $\mathcal{I}$.

Claim $I_1 + OPT(\mathcal{I}')$ is an optimal solution for $\mathcal{I}$.

Aug 31

Proof: Recall that we showed that there exist an optimal solution for $\mathcal{I}$ that contains I_1

Let it be $I_1, J_1, J_2, \dots J_\ell$

Clearly $J_1, J_2, \dots, J_\ell$ are disjoint from I_1.

Hence, $\{J_1, J_2, \dots, J_\ell\}$ is a valid solution for $\mathcal{I}'$.

$$\rightarrow |OPT(\mathcal{I}')| \geq |\{J_1, J_2, \dots, J_\ell\}|$$

$$\Rightarrow \underbrace{|I_1 + OPT(\mathcal{I}')|}_{\text{optimal for } \mathcal{I}.} \geq |\{I_1, J_1, J_2, \dots, J_\ell\}|$$

$$= |OPT(\mathcal{I})|$$

Pseudocode

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

- Sort according to f_j.

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

```
for (i = 1 to n)
  → if (s_i ≥ f) then
        select (s_i, f_i)
        f ← 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 $t_i > d_i$
t_i ← actual submission

Minimize maximum lateness over all assignments.

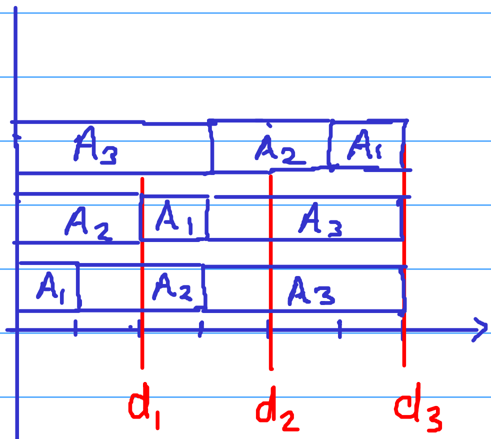
→ smallest length first?
→ earliest deadline first?
→ minimum $d_i - l_i$ first?

Example:

	A_1	A_2	A_3
length	1	2	3
deadlines	2	4	6

	A_1	A_2	A_3		orders
Lateness	4	1	0	←	$A_3\ A_2\ A_1$
Latness	1	0	0	←	$A_2\ A_1\ A_3$
Latness	0	0	0	←	$A_1\ A_2\ A_3$

Red shows maximum lateness.
Best order is $A_1 A_2 A_3$.

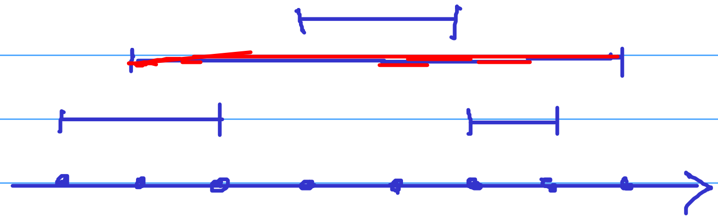


② Interval scheduling: cater to all requests with using minimum number of servers.
(one server can cater to one request at a time)

Equivalently, finding the minimum number of platforms required for a set of trains stopping at a station.

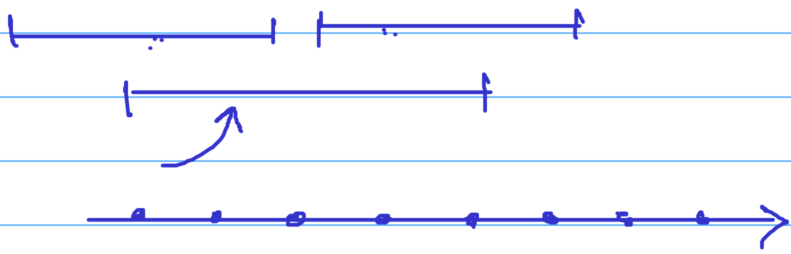
Interval Scheduling : another variant

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

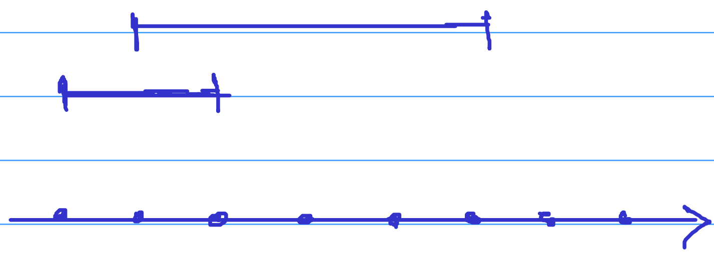


For example, there might be a profit proportional to the duration of use.

Greedy strategy 1 keep picking the longest length intervals



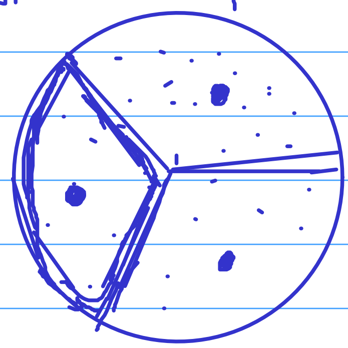
Greedy Strategy 2 : earliest starting time



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.

$\mathcal{I} = \{ I_1, I_2, I_3, \ldots, I_n \}$

Two kinds of solutions

→ contain I_1

→ don't contain I_1

Can we find optimal from both the kinds recursively?

$\mathcal{I}' = \mathcal{I} - \underline{\underline{\text{overlap}(I_1)}}$

$OPT(\mathcal{I}') + I_1$

$\mathcal{I}'' = \mathcal{I} - I_1$ $\quad OPT(\mathcal{I}'')$

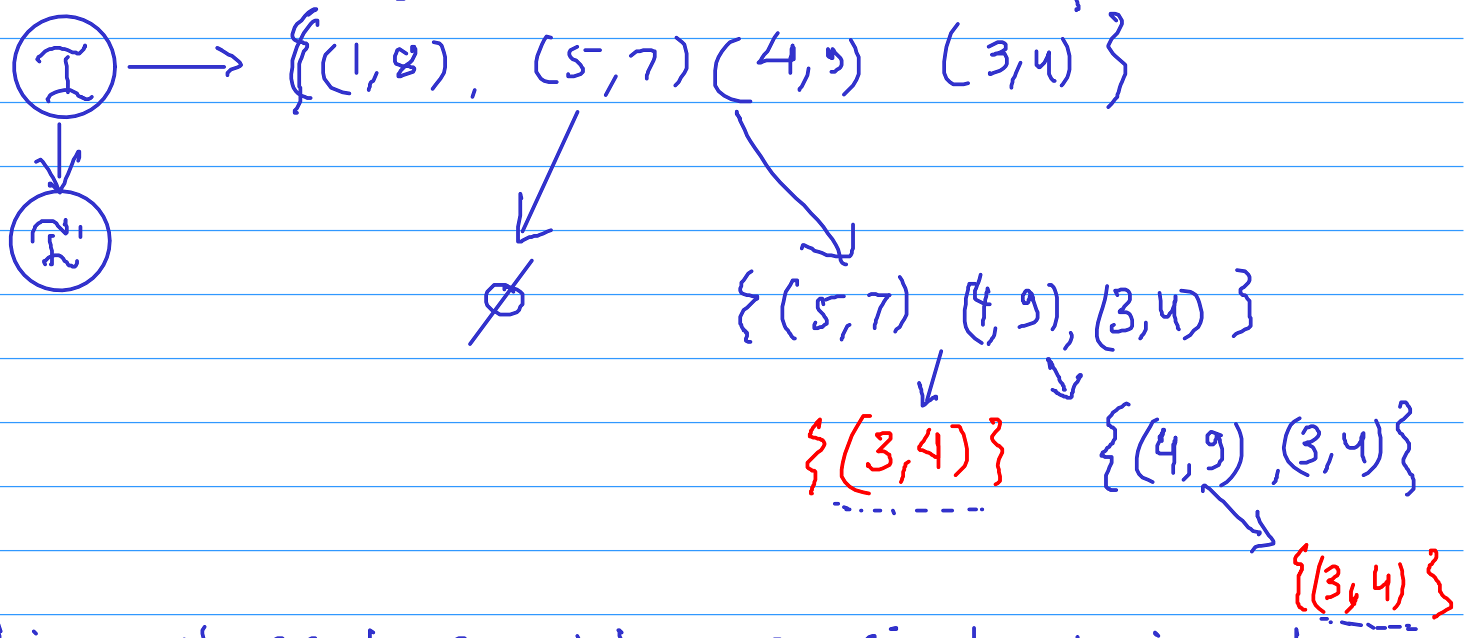
$$\mathrm{OPT}(\mathcal{I}) = \max\left\{ I_1 + \mathrm{OPT}(\mathcal{I}'),\ \mathrm{OPT}(\mathcal{I}'') \right\}$$

<u>Example</u> $\mathcal{I} = \{(0,5)\ (1,8),\ (5,7)\ (4,9),\ (3,4)\}$

$I_1 = (0,5)$

$\mathcal{I}' = \{(5,7)\}$

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



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 <u>distinct</u> recursive calls is <u>small</u>.

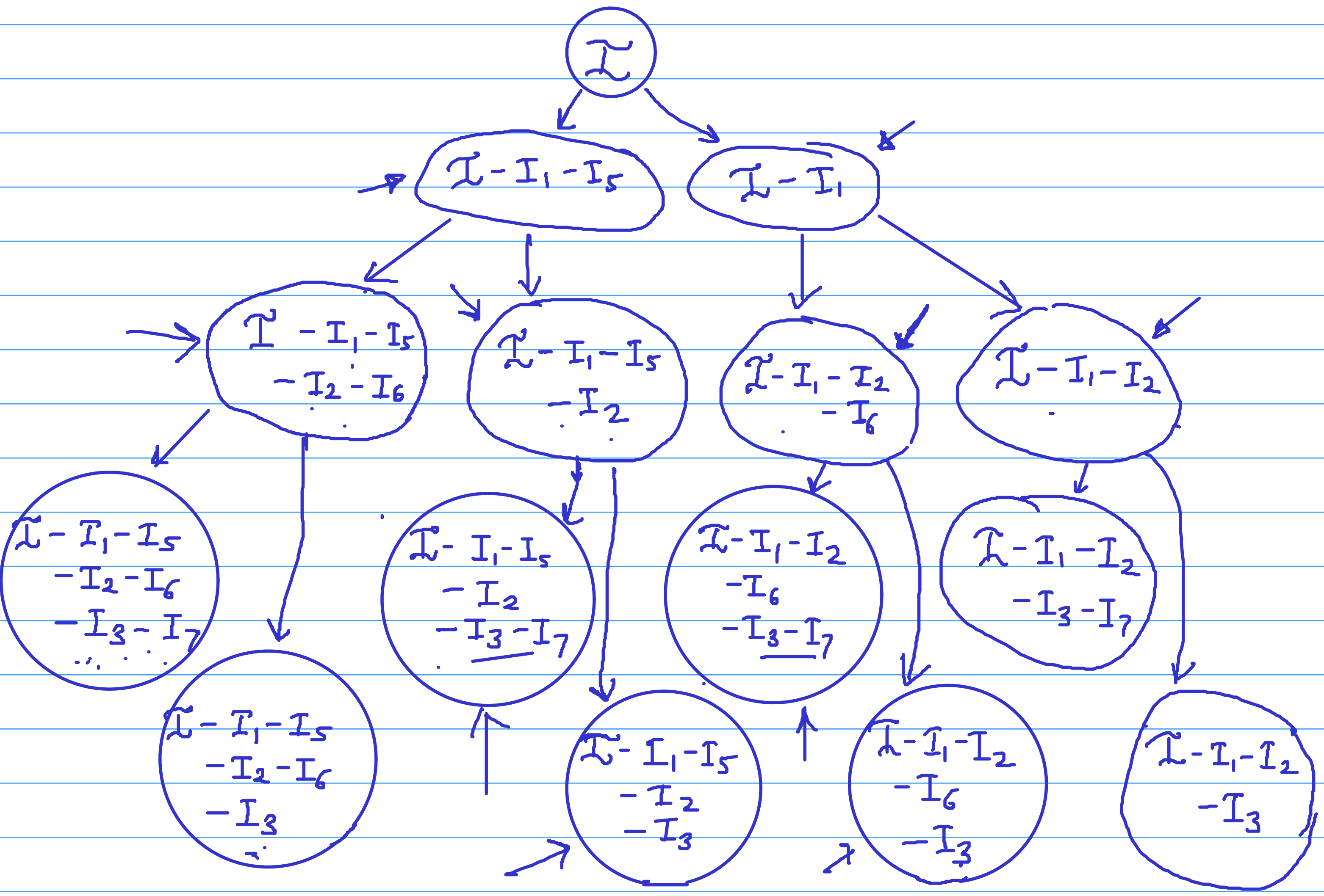
$\mathcal{I} = \{(1,3), (11,13), (21,23), (31,33), (2,4), (12,14), (22,24), (32,34)\}$

$I_1 \quad I_2 \quad I_3 \quad I_4 \quad I_5 \quad I_6 \quad I_7 \quad I_8$

Sep 2

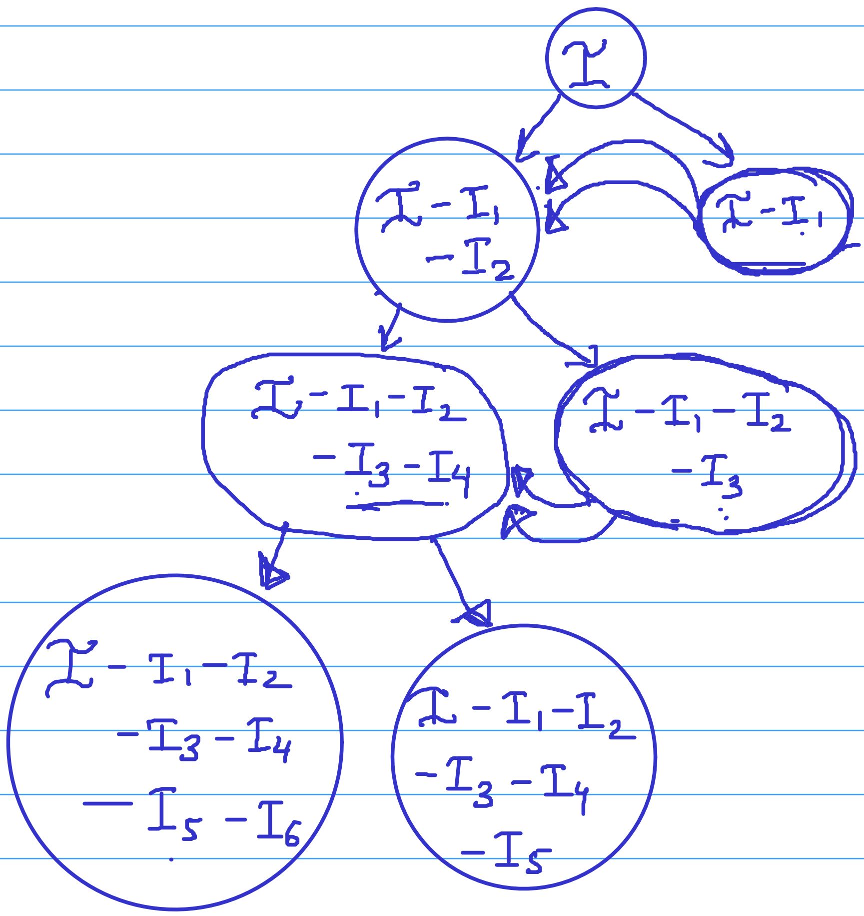
<u>HW</u> work out the recursion tree and figure out whether the number of distinct recursive calls is growing exponentially or polynomially

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



If we have $2n$ intervals, we will have at least 2^n distinct recursive calls or subproblems

$$\mathcal{I} = \{ \underset{I_1}{(1,3)}, \underset{I_2}{(2,4)}, \underset{I_3}{(11,13)}, \underset{I_4}{(12,14)}, \underset{I_5}{(21,23)}, \underset{I_6}{(22,24)}, \underset{I_7}{(31,33)}, \underset{I_8}{(32,34)} \}$$



Only 2n distinct recursive calls or subproblems.

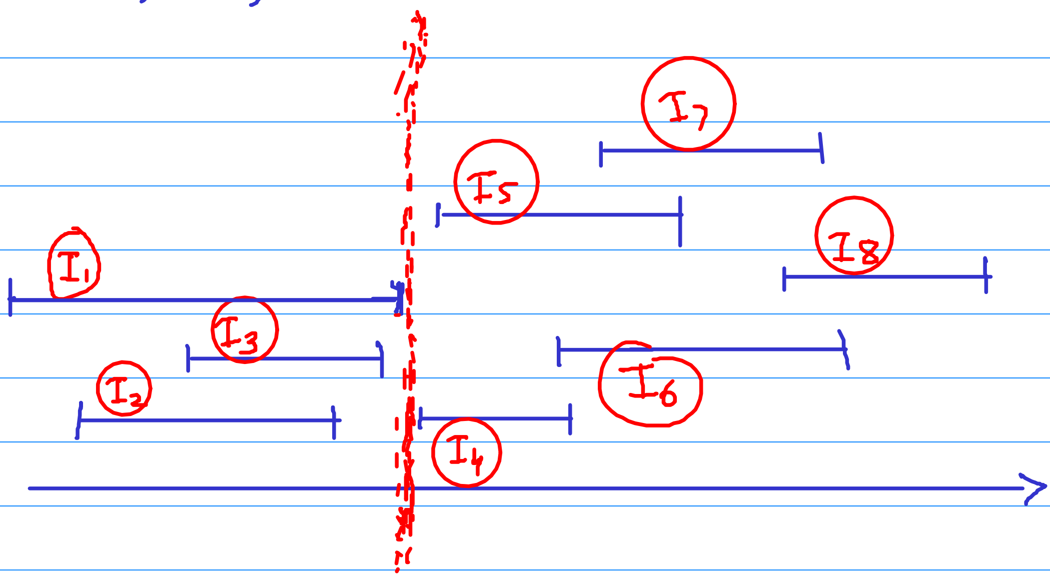
It seems if the intervals are arranged in a particular order, the number of distinct subproblems will be small.

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

$\rightarrow \{I_j, I_{j+1}, I_{j+2}, \ldots, I_n\}$

(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

Opt ← array with all zeros.
Opt[n] ← length(I_n)

Output ALG(1);

ALG(j): // ALG(j) computes optimal solution for → $\{I_j, I_{j+1}, \dots I_n\}$

if Opt[j] > 0 return Opt[j]
// means already solved and stored.

else

$$Opt[j] \leftarrow \max \begin{cases} ALG(j+1) \\ length(I_j) + ALG(p(j)) \end{cases}$$

return Opt[j]

p(j): least index k such that start(I_k) > end(I_j)

Iterative Implementation:

Opt ← array with all zeros.
Opt[n] ← length(I_n)

for (j = n-1 to 1)

$$Opt[j] \leftarrow \max \begin{cases} Opt[j+1] \\ length(I_j) + Opt[p(j)] \end{cases}$$

HW Add code to compute the optimal set of intervals.

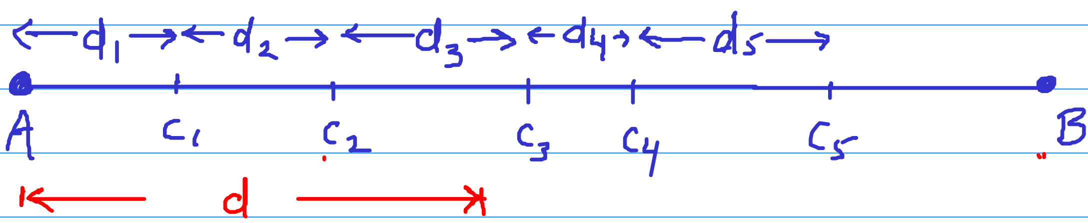
Conclusion: Order of processing the input is important.

Intervals: Order of starting time / ending time

Sequences: left to right

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

HW 1



Maximum travel in a day $-$ $\underline{d}$.

Night stay prices $-$ $P_1, P_2, \ldots$

Minimize total stay cost during the journey.

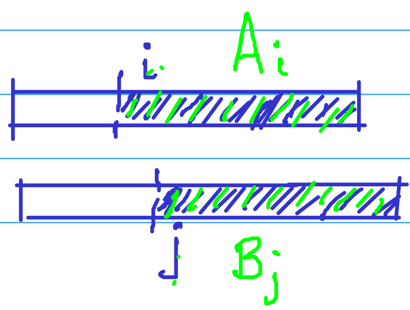
$\underline{\underline{O(n)}}$

no. of distinct subproblems $= n$.

HW 2

$$\begin{array}{l} \ 2\ 5\ 3\ 1\ 2\ 5\ 3\ 1\ 3\ 1\ 2\ 4\ 1 \\ \rightarrow A = b\,a\,c\,b\,c\,b\,a\,b\,c\,a\,c\,b\,a \\ \rightarrow B = b\ c\ b\ c\ a \end{array}$$

Match B inside A with minimum cost



no. of distinct subproblems $O(mn)$

Subset Sum problem

Given set of integers $A = \{a_1, a_2, a_3, \dots, a_n\}$

is there a subset with sum zero?

Example: $2, -5, 1, 7, -4, -6$ $\qquad \{2, 7, -5, -4\}$

Solutions $\rightarrow$ containing a_n

$\searrow$ not containing a_n

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 $A_j = \{a_1, \dots, a_j\}$, 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):

$\rightarrow$ ALG$(j-1, N)$ OR ALG$(j-1, N-a_j)$

Pseudo polynomial time.

A polynomial time algorithm is supposed to take time poly(n, no. of bits in $a_1, a_2 \dots$).

Subset Sum Pseudocode

$S \leftarrow$ Boolean two-dim array of size $n \times (\sum |a_i| + 1)$

$S[j, k]$ will denotes whether there is a subset of first j numbers with sum $= k$.

say, range of j $\quad 1 \le j \le n$.

and range of k $\quad \underbrace{neg}_{\text{sum of all negative numbers}} \le k \le \underbrace{pos}_{\text{sum of all positive numbers}}$

Initialization:

$$S[1, k] \leftarrow \begin{cases} \text{True} & \text{for } k = a_1 \\ \text{False} & \text{for all other valuse of } k. \end{cases}$$

for ($j = 2$ to n)

    for ($neg \le k \le pos$)

        $S[j, k] \leftarrow S[j-1, k]$ OR $S[j-1, k-a_j]$

To construct the desired set

Assume $S[n, 0]$ is true.

sum $\leftarrow 0$

for ($j = n$ to 1)

    if $S[j-1, sum] =$ true, don't take a_j

    elseif $S[j-1, sum - a_j] =$ true, take a_j and sum $\leftarrow$ sum $- a_j$

HW Knapsack problem

n objects weights $w_1, w_2, \ldots, w_n$
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.

Sep 6

Balanced margins

average slack $14/3 = 4.66$

$W = 68$ slack

Suppose there ten tourist guides, of which, six can speak French and 0
six can speak German (two can speak both French and German). 8
Everyone comes with their own charges. We want to select five. 6

Suppose there ten tourist guides, of which, six can speak French 4
and six can speak German (two can speak both French and German). 4
Everyone comes with their own charges. We want to select five. 6

We are given a sequence of words of lengths $w_1, w_2, w_3 \cdots w_n$.

21
21
20
21

Each line can have at most W characters.

$w_i \sqcup w_{i+1} \sqcup \ldots \sqcup w_j$

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

$$s = W - \underbrace{\left[(w_i + 1) + (w_{i+1} + 1) + \cdots + (w_{j-1} + 1) + w_j\right]}$$

$l_1, l_2, \cdots l_k$

$$\text{Variance} = \text{avg}(l_1^2, l_2^2, \cdots l_k^2) - (\underbrace{\text{avg}(l_1, l_2, \cdots, l_k)}_{\uparrow \text{ fixed.}})^2$$

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

<u>Puzzle</u> 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

<u>Idea 1:</u> fit as many words as you can

→ can be very unbalanced

<u>Greedy Idea</u>:

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.

doesn't give an optimal solution.

5	5	2	vis
	1	4	
	5	5	51, 45.

is

$11/3 = 3.66$

Dynamic Programming.

- Try to categorize the set of all possible solutions

Each solution is a partition of words into k lines.

$$n = n_1 + n_2 + n_3 + \dots + n_k$$

Categories: $n_k = 1$ last line has 1 word.

$n_k = 2$ last line has 2 words.

$\vdots$

$n_k = n-1$ last line has $n-1$ words.

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

$$OPT(w_1, w_2, \dots, w_p) + slack(w_{p+1}, w_{p+2} \dots w_n)^2$$

$$OPT(n) = \min \begin{cases} OPT(n-1) + [W - w_n]^2 \\ OPT(n-2) + [W - w_n - w_{n-1} - 1]^2 \\ OPT(n-3) + [W - w_n - w_{n-1} - w_{n-2} - 2]^2 \\ \vdots \end{cases}$$

Running time $O(n^2)$

Pseudocode for computing the optimal value and the optimal solution.

$S \leftarrow$ array of length $n+1$

// $S[j]$ denotes the minimum sum of squares of slacks for first j words, for $1 \le j \le n$

$N \leftarrow$ array of length $n+1$

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

$S[0] \leftarrow 0$; $S[j] \leftarrow \infty$ for $j>0$; $N[0] \leftarrow 0$

```
for (j = 1 to n)
    for (r = j to 1)
        slack ← W − (w_r + w_{r+1} + ... + w_j + j−r)
        if (slack ≥ 0 and S[j] > S[r−1] + (slack)^2)
            S[j] ← S[r−1] + (slack)^2
            N[j] ← r
```

Optimal Arrangement:
Run Arrange(n).

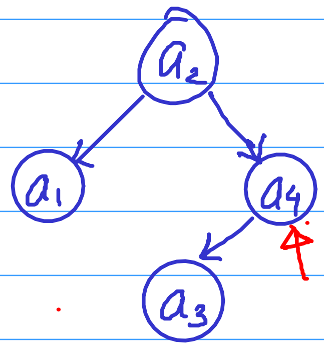
```
Arrange(j) {
    Arrange(N(j) − 1)
    print words from N(j) to j
        in a new line.
}
```

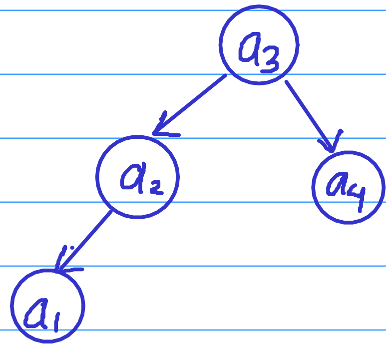
Optimal Binary Search Tree

Sep 7

$a_1 < a_2 < a_3 < a_4$

7×2
5×1
4×3
6×2
$\overline{\ \ 43\ \ }$



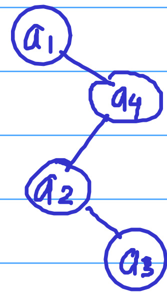


7×3
5×2
4×1
6×2

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

Say

		frequencies
a_1	$\rightarrow$	7
a_2	$\rightarrow$	5
a_3	$\rightarrow$	4
a_4	$\rightarrow$	6



7×1
$+ 6 \times 2$
$+ 5 \times 3$
$+ 4 \times 4$
$\overline{\ \ 50\ \ }$

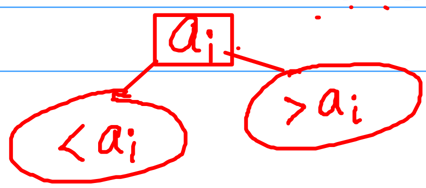
	Total search cost
First binary tree	43
Second binary tree	47

Compute the binary search tree with **minimum** total search cost $\sum_i f_i h_i$

where $f_i \rightarrow$ frequency(a_i) $h_i \rightarrow$ depth(a_i)

Greedy Idea 1:

maximum frequency element $\rightarrow$ root

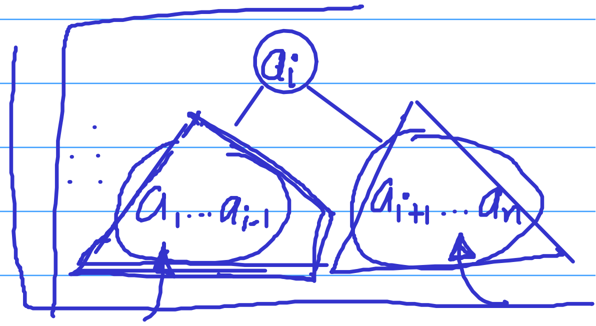


Dynamic Programming

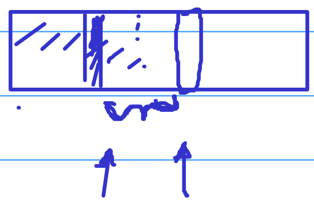
$a_1 < a_2 < a_3 \cdots < a_n$

Possible solutions : every valid binary search tree.

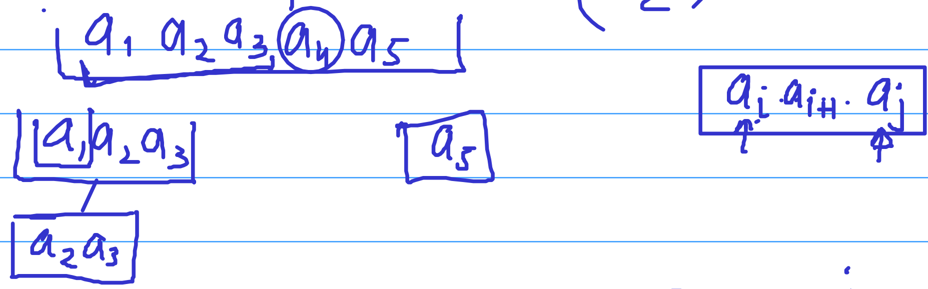
Categories : Root $\leftarrow a_1$
Root $\leftarrow a_2$
$\vdots$
Root $\leftarrow a_n$



$$\text{Tree}(a_1, \cdots a_n) = \min_{1 \le i \le n} \left\{ \begin{array}{l} \vdots \\ \text{Tree}(a_1, \cdots a_{i-1}) + \text{Tree}(a_{i+1}, \cdots, a_n) \\ + \sum_{j=1}^{n} f_j \cdot 1 \end{array} \right.$$

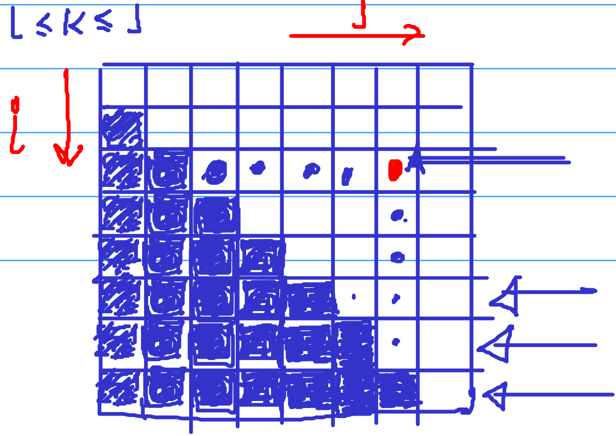


How many distinct subproblems $= \binom{n}{2}$



$$\text{Tree}(i, j) = \min_{\substack{k \\ i \le k \le j}} \{ \text{Tree}(i, k-1) + \text{Tree}(k+1, j) \} + \sum_{k=i}^{j} f_k$$

$j > i$



OPT $\leftarrow$ 2 dim array. $\qquad\qquad$ $\underline{\underline{O(n^3)}}$

OPT $[i, i] \leftarrow f_i$ for each i

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

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

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

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          UGC_TG ACU
→ GAATGCA           _GAATG _CA
```

δ 0 cost α_{CA} δ α_{UA}

$\boxed{3\delta + \alpha_{CA} + \alpha_{UA}}$

Given

Gap Penalty δ

Mismatch cost (for each pair) α_{xy}

```
UGC_TG_ACU
_GAATGCA__
```

$5\delta + \alpha_{CA}$

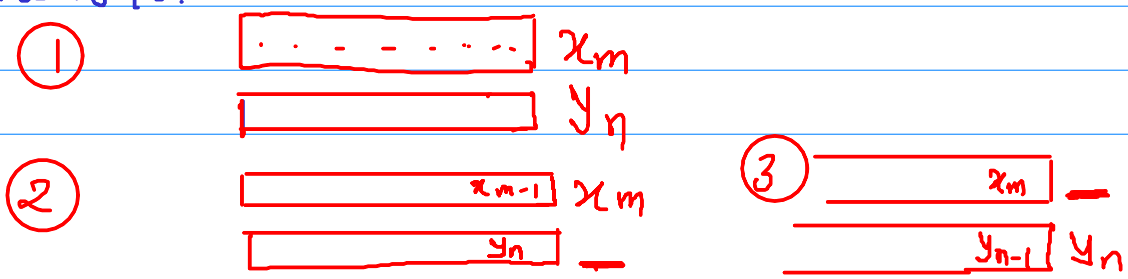
Total cost = sum of the gap and mismatch costs.

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

Input: $x_1 x_2 \cdots x_m$ and $y_1 y_2 \cdots y_n$

δ, $\{\alpha_{x_i y_j}\}$

Categories of solutions.



$x_1 \cdots x_{m-1}$
$y_1 \cdots y_{n-1}$

$$OPT(m,n) = \min \begin{cases} \alpha_{x_m y_n} + OPT(m-1, n-1) \\ \delta + OPT(m-1, n) \\ \delta + OPT(m, n-1) \end{cases}$$

$OPT(i, j)$ no. of distinct subproblems $m \cdot n$

Implementation

$A \longleftarrow$ 2D array $m \times n$

// $A[i, j]$ denotes the minimum cost of alignment for $x_1 x_2 \cdots x_i$ and $y_1 y_2 \cdots y_j$

$A[i, 0] \leftarrow \delta i$ for each i
$A[0, j] \leftarrow \delta j$ for each j

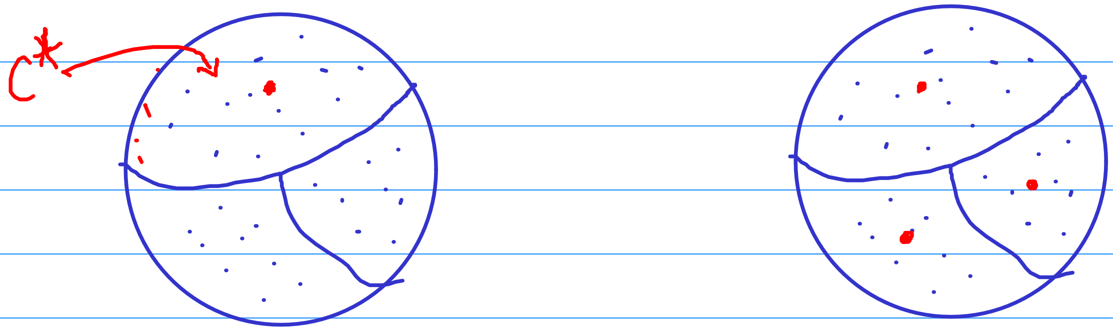
for (i = 1 to n)
  for (j = 1 to n)
    $A[i, j] = \min \{ \alpha_{x_i y_j} + A[i-1, j-1],$
    $\delta + A[i-1, j],$
    $\delta + A[i, j-1]$
    $\}$

HW Space $O(mn)$
Can you get space $O(m+n)$ and time $O(mn)$ } Divide and Conquer

Sep 20

Summarizing Greedy and Dynamic Programming

- Dividing the set of possible solutions into multiple categories.



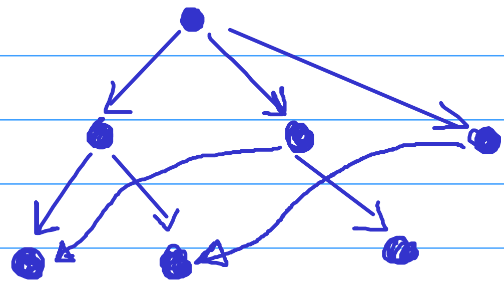
Greedy: there must be an optimal solution in a certain category C^* (greedy choice)

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

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

Greedy

DP



No. of distinct subproblems should be small.

Data Compression: Coding

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

a ⟶ 00001
b ⟶ 00010
⋮

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	•	z	— — • •
t	—	q	— — • —
a	• —		

⋮

Problem • — • — → aa, eta, aet, etet

Solution: Gap after every character

Prefix Code

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

Ex
$$\left.\begin{array}{l} x \to 00 \\ y \to 001 \end{array}\right\}$$ Not a prefix code

Ex
$$\left.\begin{array}{l} a \to 0 \\ b \to 10 \\ c \to 11 \end{array}\right\}$$ Prefix Code.

abaca $\to$ $\underset{a}{0}\,\underset{b}{10}\,\underset{a}{0}\,\underset{c}{11}\,\underset{a}{0}$

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

Freq		Code 1	Code 2
0.4	A	00	0
0.4	T	01	10
→ 0.1	P	10	110 ←
→ 0.1	G	11	111 ←
		2	$0.4\times1 + 0.4\times2 + 0.1\times3 + 0.1\times3 = 1.8$

Avg Bit length

100 char — 200 bits — 180 bits.

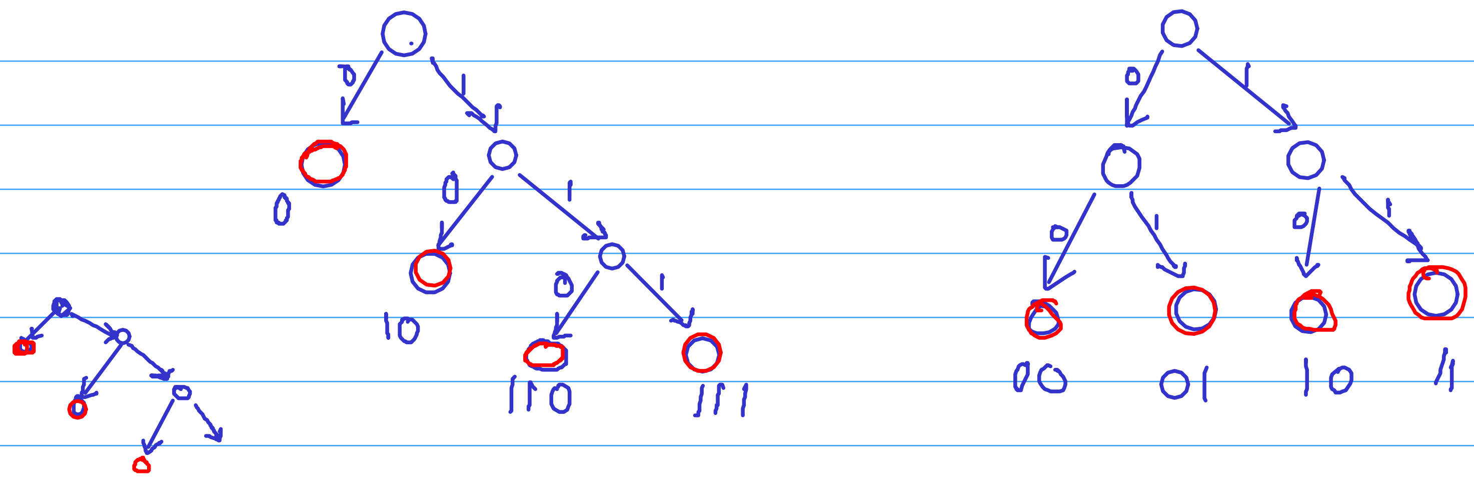
Prob Given frequencies $f_1, f_2, \cdots, f_n$ for n characters find a prefix code that minimizes $\underbrace{\sum_i f_i l_i}_{\text{avg encoding length}}$.

$(\sum f_i = 1)$

l_i: length of the encoding 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.

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

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

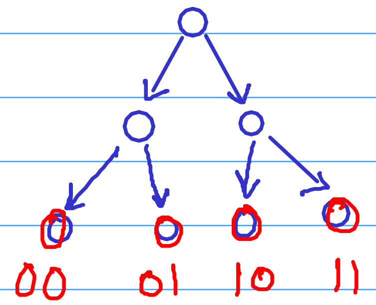
0.32 0.25 0.18 0.25

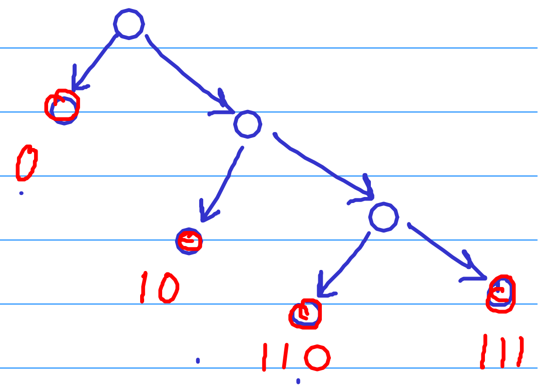
Sep 21

Observations:

low frequency → higher length
high frequency → lower length

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





(0.25, 0.25, 0.25, 0.25)		
(0.28, 0.24, 0.24, 0.24)	2	$0.28 \times 1 + 0.24 \times 2 + 0.24 \times 3 + 0.24 \times 3 = 2.2$
(0.37, 0.21, 0.21, 0.21)	2	= 2.05
(0.35, 0.35, 0.15, 0.15)	2	$\boxed{= 1.95}$

Assign '0' if frequency higher than certain threshold.

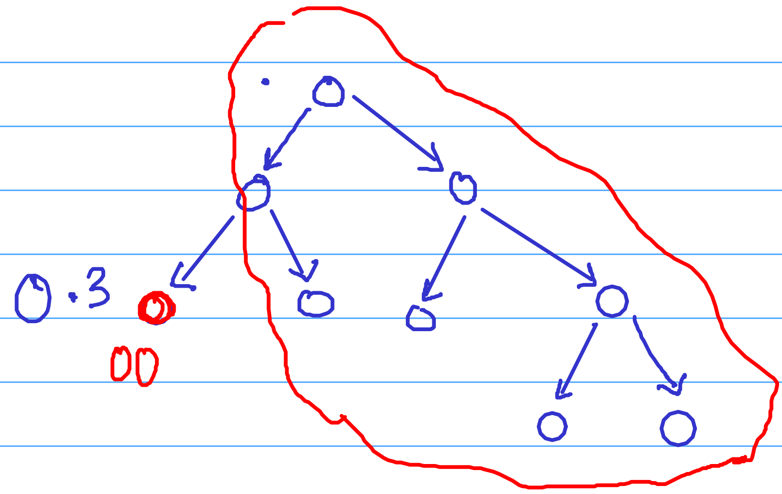
Doesn't work.
Cannot decide l_i just by looking at f_i.

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

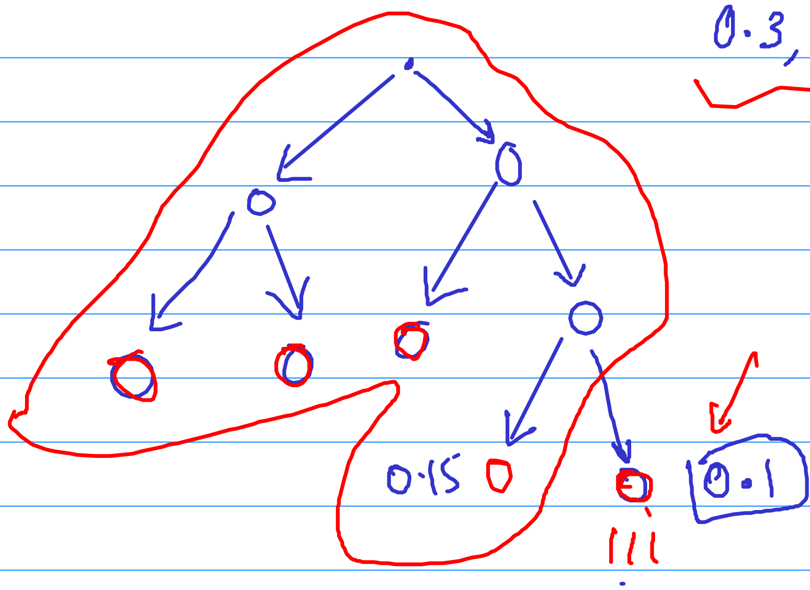
Can we reduce the rest of the encodings to a subproblem?



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

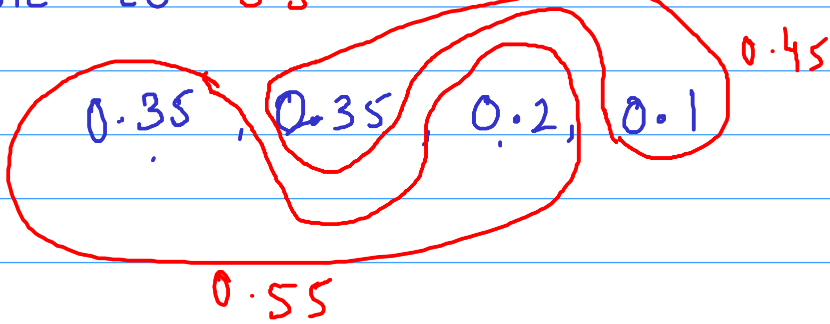


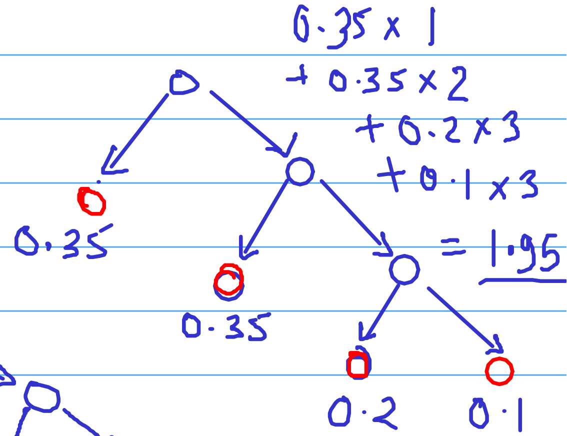
same issue.

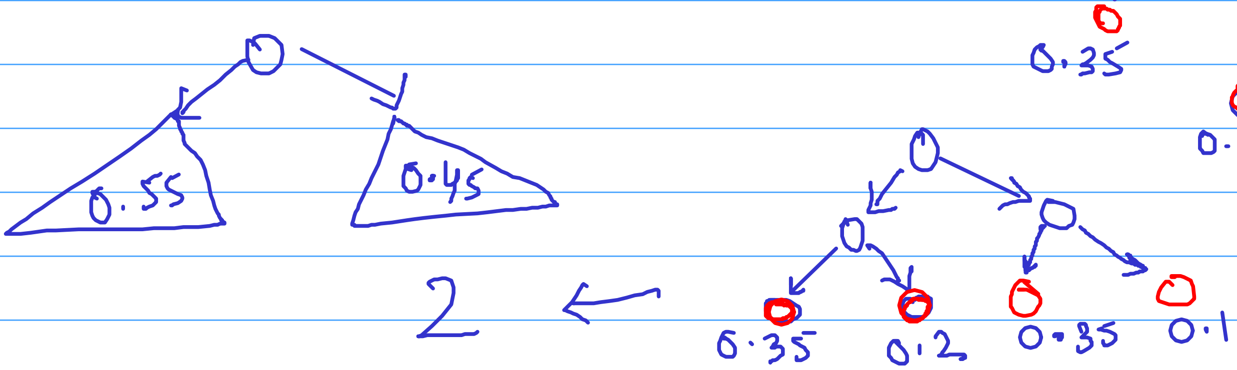
Shannon and Fano (1940's)

Balanced Partition

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

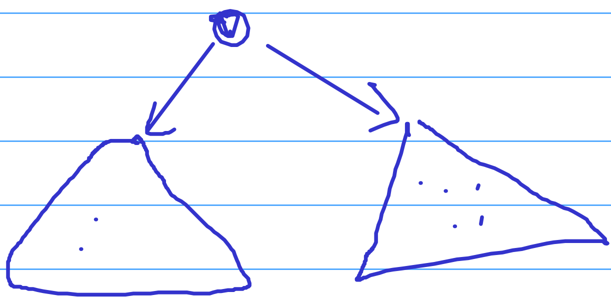






By balanced partition approach we are getting average encoding length = 2. But there is a better solution with 1.95.

→ Try various possibilities for the partition?

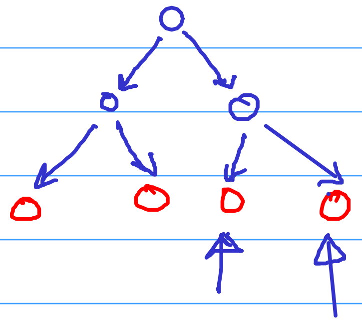


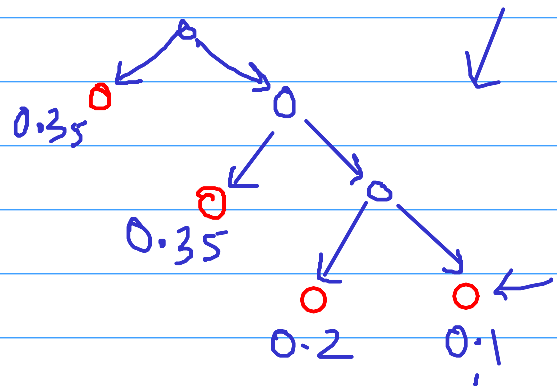
Exponentially many possibilities.

Huffman [1952]

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

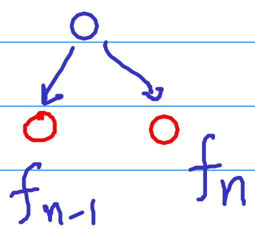




Lowest frequency character has the largest depth.

Obs2: The leaf with the largest depth must have a sibling which is a leaf.

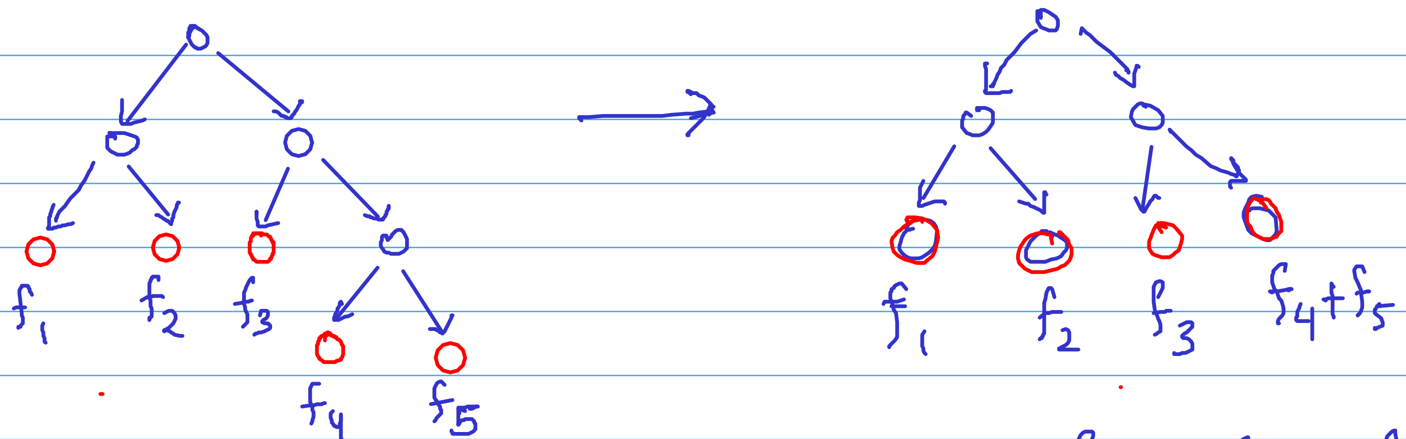
$$f_1 \geq f_2 \geq \cdots \geq \underbrace{f_{n-1} \geq f_n}$$



Claim: 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 (in decreasing order)

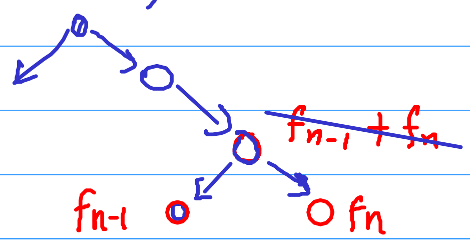


$\text{cost}_0: 2f_1+2f_2+2f_3+3f_4+3f_5$ $\quad$ $\text{cost}_1: 2f_1+2f_2+2f_3+2f_4+2f_5$

$$\boxed{\text{cost}_0 = \text{cost}_1 + f_4 + f_5}$$

→ 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

$$f_1, f_2, f_3, \dots, f_{n-2}, f_{n-1}+f_n$$



Algorithm:

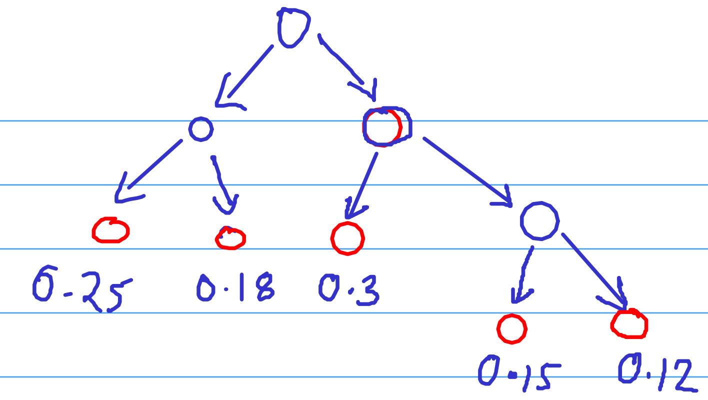
Input: $f_1 \geq f_2 \geq \dots \geq f_{n-1} \geq f_n$ (n char)

Recursively compute optimal binary tree for
→ $f_1, f_2, \dots f_{n-2}, f_{n-1}+f_n$ ($n-1$ char)

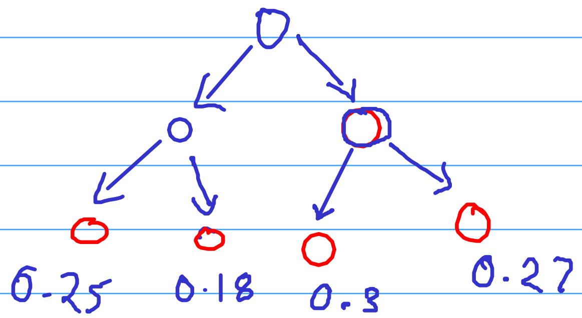
For the leaf labeled $f_{n-1}+f_n$, add two children with label f_{n-1}, f_n.

Example

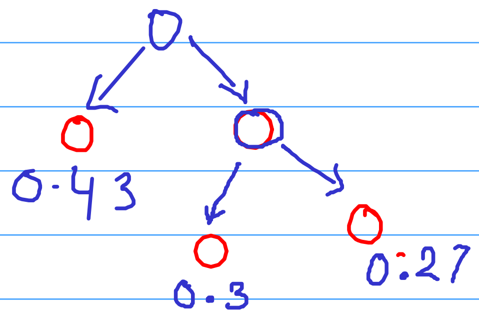
0.3, 0.25, 0.18, 0.15, 0.12,



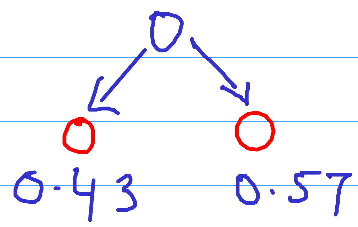
0.3, 0.25, 0.18, 0.27



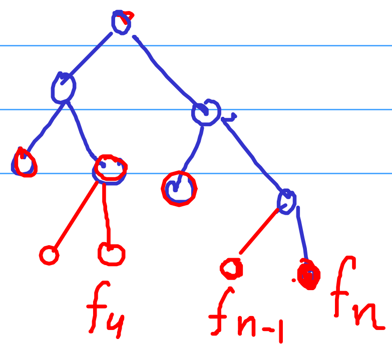
0.3, 0.27, 0.43

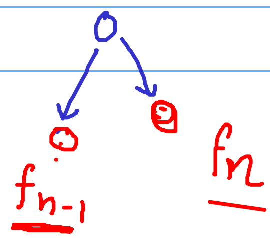


0.43, 0.57



$\sum f_i l_i$





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

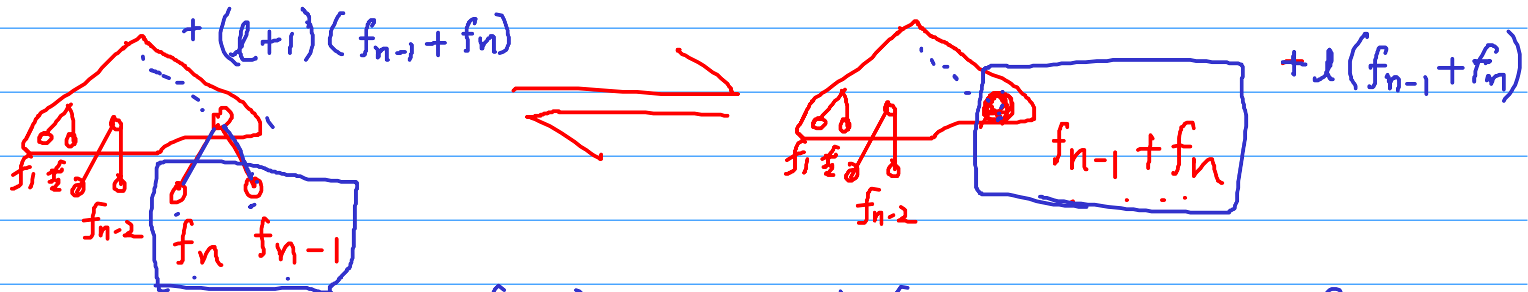
$A = \{$ full binary trees with n leaves labeled $f_1, f_2, \cdots, f_n$ where f_n and f_{n-1} are siblings $\}$

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

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

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

$$T_j \underset{\text{For the leaf labeled } f_{n-1}+f_n \text{, add two children labeled } f_{n-1}, f_n}{\overset{\text{Remove leaves labeled } f_{n-1}, f_n \text{. label their parent } f_{n-1}+f_n}{\rightleftarrows}} R_j$$



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

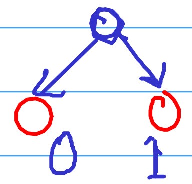
Thus, R_j is optimal in $B \iff T_j$ is optimal in A.

Huffman Codes:

Text over some large alphabet size $\longrightarrow$ 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.



b a g . . a a

001000110101000011100110000010

a	000	...2/8	$\longrightarrow$ 00
b	001	3/16	$\longrightarrow$ 100
c	⋮	⋮	⋮
⋮			
h	111	1/16	$\longrightarrow$ 1101

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