Lec 5: Greedy Algorithms

UH COSC3320, Spring 2021

References:

- Algorithms, Jeff Erickson, Chapter 4
- Introduction to Algorithms, 3rd Ed, Chapter 16.

Storing Files on Tape

Suppose we have n files stored on a magnetic tape. To read a file, a user must fast-forward past all previous files, which takes significant time.

Let $L[1 \ldots n]$ be an array of the lengths of the files $1..n$. (file i has length $L[i]$. Then the cost of accessing file k is

$$\mathrm{cost}[k] = \sum_{i=1}^{k} L[i]$$

Different files have different accessing cost. Suppose that each file equally likely to be accessed, then the **expected** cost of accessing a random file is

$$\mathbb{E}[\text{cost}] = \frac{1}{n}\sum_{k=1}^{n}\sum_{i=1}^{k} L[i]$$

If we change the order of files on the tape, we change the cost of accessing; some files will be faster and some slower. Specifically, let $\pi[i]$ denote the index of file stored at position i on the tape. Then the expected cost of accessing random file is

$$\mathbb{E}[\text{cost}(\pi)] = \frac{1}{n}\sum_{k=1}^{n}\sum_{i=1}^{k} L[\pi[i]]$$

Different order (π) gives different $\mathbb{E}[\text{cost}(\pi)]$. What π minimizes the expected cost?

Intuitively, it seems placing shorter files first would be better, because

Files in front is more often getting in the way.

We can order the files on tape in increasing file sizes. Does this gives us the minimum expected cost of accesing random file?

It's easy to come up with greedy algorithms, but most of them won't work. For any greedy algorithm, we must prove it rigorously!

Let's try to prove it:

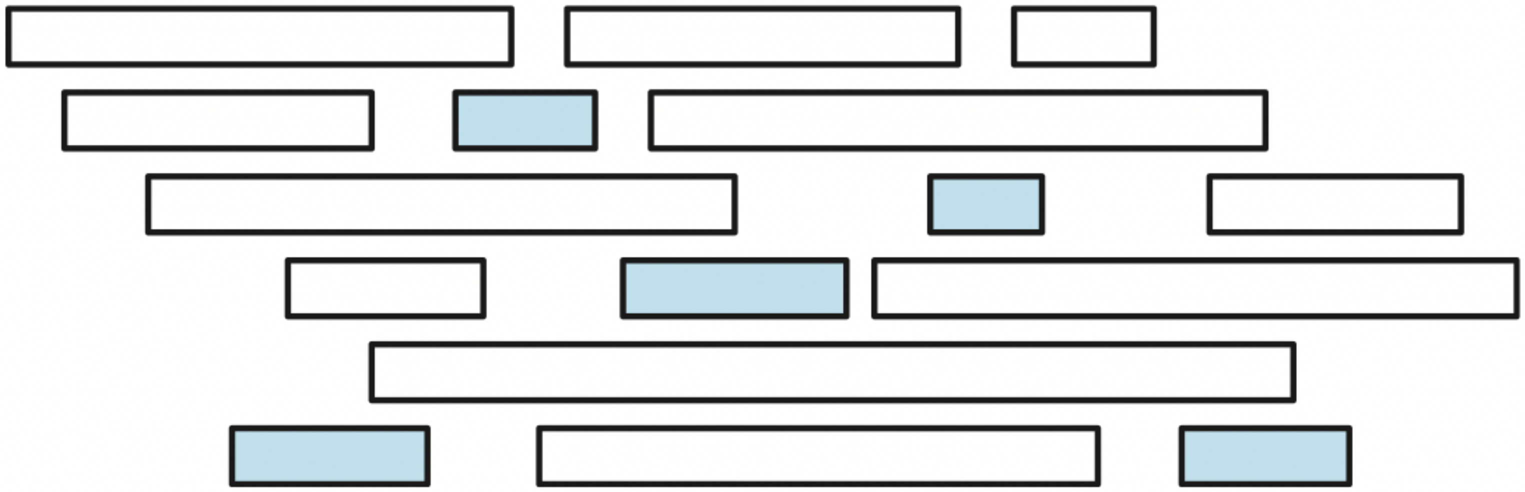
Lemma. $\mathbb{E}[\mathrm{cost}(\pi)]$ *is minimized when* $L[\pi[i]] \leqslant L[\pi[i+1]]$ *for all i.*

Sketch proof: if $\pi[i] > \pi[i+1]$, swap them and show that the expected cost decreases. □

This is a successful **greedy** algorithm. There is no systematically visiting all possibilities, no recursion or dynamic programming. It makes decision based on what seems to be the best next move (picking the shortest file next), and blindly plowing ahead.

The cost is $O(n \log n)$ for the sorting step.

Scheduling Classes



Suppose we are given n classes with potentially overlapping lecture time. Class i starts at time $S[i]$ and finishes at $F[i]$. Find the maximum number of non-overlapping classes.

We can visualize the class as blocks on time axis. The goal is to find the largest subset of blocks with no vertical overlap.

To put the non-overlapping rule in other words, we are looking for subset of the indices $X \subset \{1 \ldots n\}$ such that if $i, j \in X$, then either $S[i] > F[j]$ or $S[j] > F[i]$.

Recursive solution:

To solve the class scheduling, we could try recursion (reduction). Let's look at class 1. It's either in optimal schedule, or not. If class 1 is in the optimal schedule, then we divide the rest of classes into two groups (ends before class 1 and starts after class 1 ends)

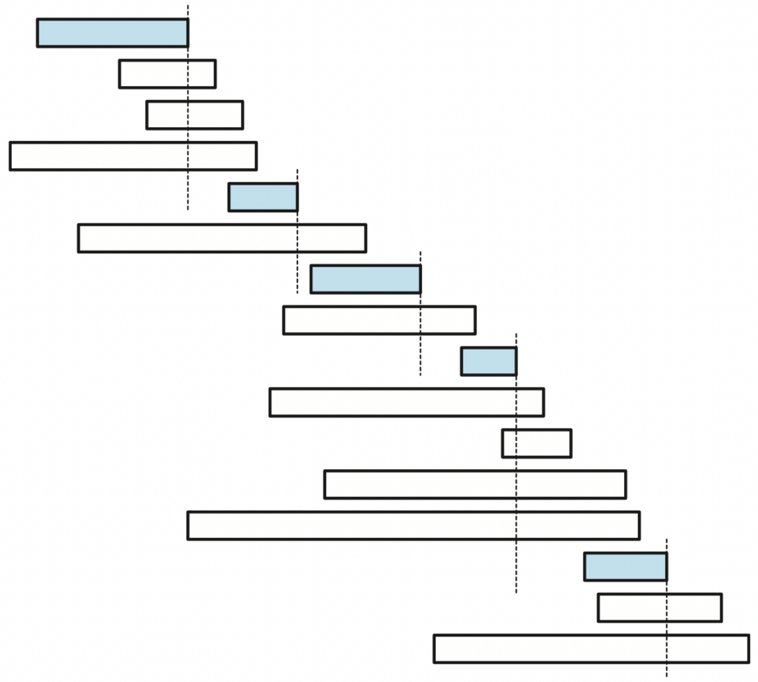
$$\begin{aligned} B &\coloneqq \{i : 2 \leqslant i \leqslant n \text{ and } F[i] < S[1]\} \\ A &\coloneqq \{i : 2 \leqslant i \leqslant n \text{ and } S[i] > F[1]\} \end{aligned}$$

The rest of classes must be scheduled optimally in their own group (recurse!). If class 1 is not in optimal schedule, then we recursively schedule class $2..n$.

What's the cost? Can we turn it into DP?

(Hint: Look at the recursion tree. Is there any duplicated nodes? You should be able to find an evaluation order that completes in $O(n^3)$)

Now let's *try* to come up with greedy algorithm. We pick class one by one. Which one do we pick? An intuition is that we'd pick the **class that finishes the earliest**, so it's more likely to schedule more other courses.



We sort the classes by the finishing time.

Now to prove that this greedy algorithm actually gives the optimal (globally) solution, we'll need to prove two properties:

- **Greedy choice property**: we can assemble a globally optimal solution by making locally optimal (greedy) choices. (This property is NOT required for Dynamic Programming!)
- **Optimal substructure**: an optimal solution to a problem contains within it a optimal solution to subproblems. (This property is also required for Dynamic Programming.)

The first property for our particular greedy algorithm translates into the following lemma:

Lemma. *At least one maximum conflict-free schedule include the class that finishes first.*

Sketch proof: Let f be the class that finishes first. Suppose a maximum conflict-free schedule X does not contain f. Replace the first finishing class in X with f. □

The second property, optimal substructure, is translated into the following lemma:

Lemma. *The best schedule that contains class f (the first finishing class) also contains an optimal schedule for the classes that does not conflict with f.*

Proof hint: by contradiction. □

With these two properties (lemmas), we can prove the greedy solution to be globally optimal by mathematical induction:

Assume that the subproblem is solved optimally by the greedy algorithm. Now argue that adding the current greedy choice is still optimal.

A failed greedy algorithm: what if we pick the earliest starting class as our greedy choice? Can you construct a simple example that such greedy algorithm does not lead to globally optimal schedule? Why does it not (which property does it violate)?

General Patterns

The basic structure of the correctness proof is based on inductive exchange argument.

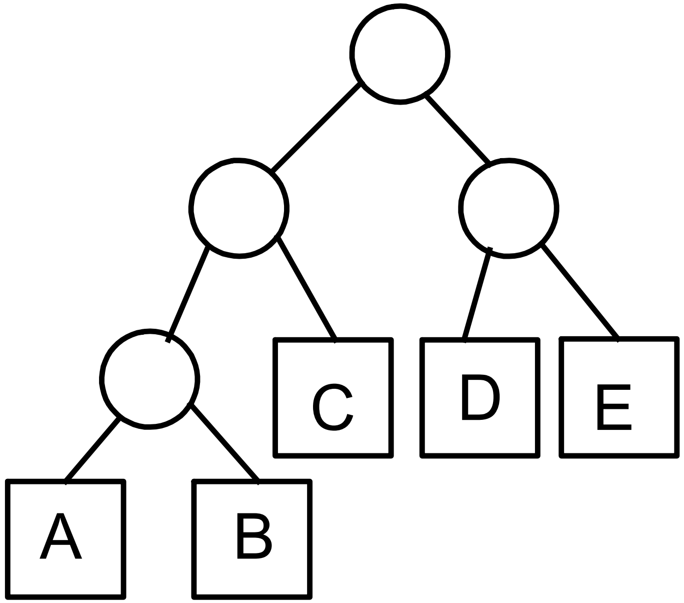
- Assume there is an optimal solution that's different from the greedy solution.
- Find the “first” difference between the two solutions.
- Argue that we can exchange the optimal choice for the greedy choice, *without making the solution worse*.

This argument implies that **some** optimal solution **contains** the entire greedy solution, and therefore **equal** the greedy solution.

Huffman Codes

Suppose we are encoding alphabet into binary strings (0s and 1s) of varying lengths. A binary code is *prefix-free*, if no code is a prefix of any other (otherwise it's hard to decode). E.g.: 7 bit ASCII and UTF-8 are both prefix-free binary coes. Morse code is NOT prefix-free, because the code for E(·) is prefix for I(··), S(···), and H(····).

Visualization. Any prefix-free binary code can be visualized as a binary tree with the encoded characters stored at the leaves. The code word for any symbol is the path from root to the corresponding leaf; 0 for left, 1 for right. The length of the code word for any symbol is the depth in the tree of the leaf.



In this example we have 5 symbols ABCDE, and their code words are 000,001,01,10,11, respectively. Note that it's prefix-free, with minimum code length 2 and maximum lenght 3.

Suppose we have a message written in n-character alphabet and we want to encode the message as short as possible. Specifically, given any frequency counts $f[1 \ldots n]$, we want to compute a prefix-free binary code that minimizes the total encoded length:

$$\sum_{i=1}^{n} f[i] \times \mathrm{depth}[i]$$

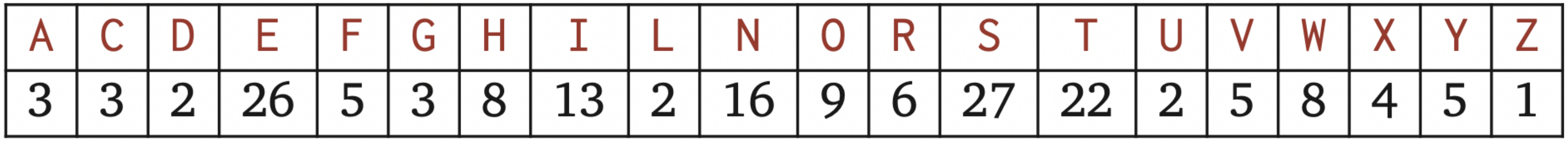
This problem has a surprisingly simple solution given by a PhD student at MIT David Huffman:

$$\boxed{\text{Merge the two least frequently used letters and recurse.}}$$

To illustrate, we look at an example. We have the following message:

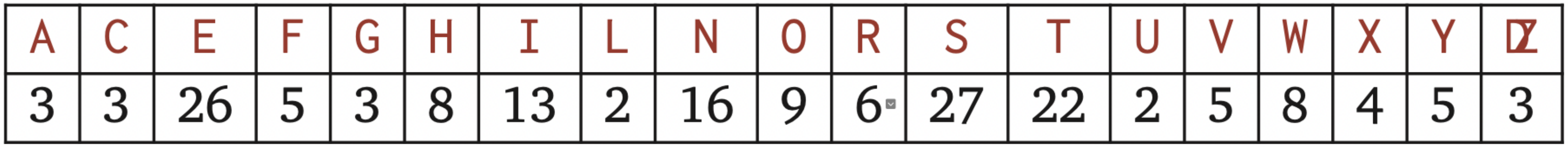
```
This sentence contains three a's, three c's,
two d's, twenty-six e's, five f's, three g's,
eight h's, thirteen i's, two l's, sixteen n's,
nine o's, six r's, twenty-seven s's, twenty-two
t's, two u's, five v's, eight w's, four x's,
five y's, and only one z.
```

Here's the frequency table for the letters (ignoring symbols other than English letters):



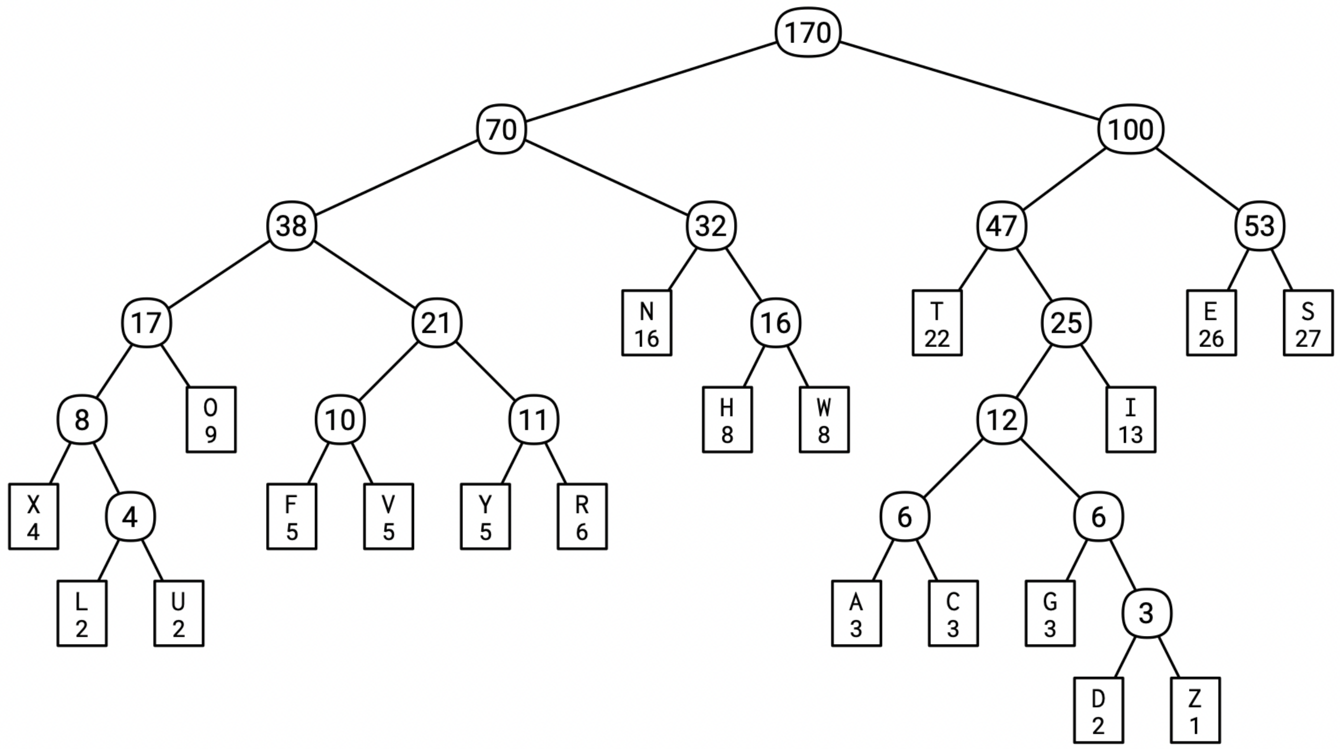
A	C	D	E	F	G	H	I	L	N	O	R	S	T	U	V	W	X	Y	Z
3	3	2	26	5	3	8	13	2	16	9	6	27	22	2	5	8	4	5	1

Huffman's algorithm picks the two least frequently used letters: Z and D, and merge them into a single character called ZD with frequency 3 (the total frequency of Z and D). Now we have a table with 1 less letter:



A	C	E	F	G	H	I	L	N	O	R	S	T	U	V	W	X	Y	DZ
3	3	26	5	3	8	13	2	16	9	6	27	22	2	5	8	4	5	3

And then pick the two least frequent, which are U and L, and merge them. And so on. To illustrate the merging, we have the tree:



Encoding the message we have with the Huffman codes yield bit string like this:

```
100 0110 1011 111 111 110 010 100 110 010 101001 110 101001 0001 010 100 ...
 T    H    I   S   S   E   N   T   E   N     C     E     C     O    N   T
```

Here is a list of costs for encoding each character in the message, along with their contributions to the total length.

char	A	C	D	E	F	G	H	I	L	N	O	R	S	T	U	V	W	X	Y	Z
freq	3	3	2	26	5	3	8	13	2	16	9	6	27	22	2	5	8	4	5	1
depth	6	6	7	3	5	6	4	4	6	3	4	5	3	3	6	5	4	5	5	7
total	18	18	14	78	25	18	32	52	12	48	36	30	81	66	12	25	32	20	25	7

In total, it costs 649 bits to encode the message.

Given the simplicity of the Huffman code, it might be surprising to hear that the Huffman code is optimal—there is no prefix-free binary encoding for the message that is shorter than 649!

Why? Let's prove it. Again we need to prove the two properties.

Lemma. ***(Greedy choice property)*** *Let x and y be the two least frequent characters (breaking ties arbitrarily). There is a optimal binary tree in which x and y are siblings.*

Schetch proof: we can prove a stronger statement: there is an optimal binary code tree in which x and y are siblings, and have the longest length.

- Assume a, b are the longest two siblings (must have 2!) in an optimal code tree
- Swap x and a, and prove that the tree is no worse.
- Similarly, swap y and b.

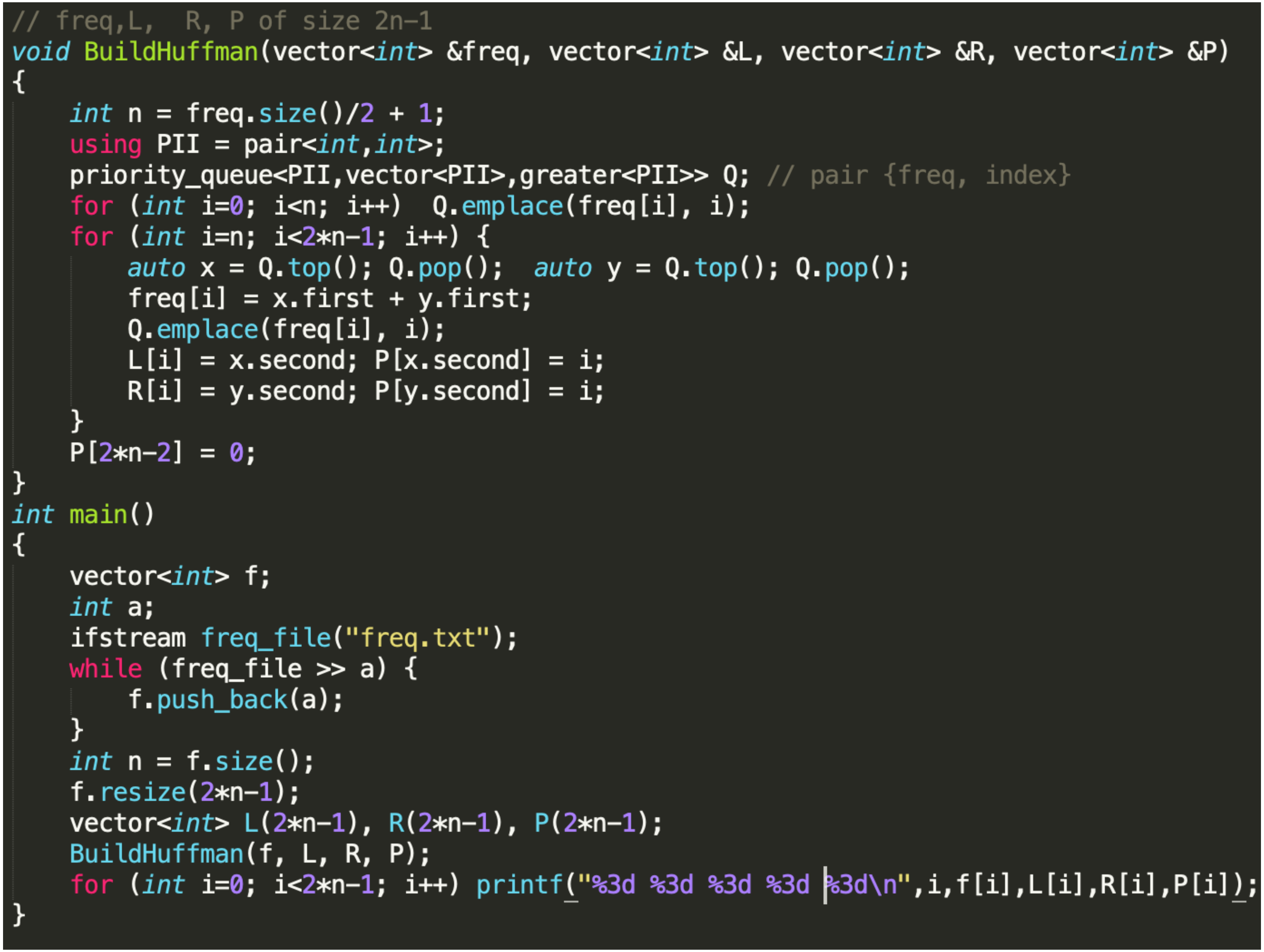
Lemma. ***(Optimal substructure)*** *For a optimal code tree with two least frequent characters x,y as deepest leaves. Combine the two leaves gives the optimal code tree for the new table with x,y combined.*

Hint:

To efficiently contruct Huffman code: we represent the binary code tree with three arrays: $L[i]$, $R[i]$, $P[i]$ are the left/right child and parent of node i; the leaves are nodes at indices $1..n$, root is with index $2n-1$

```
BuildHuffman($f[1\ldots n]$):
  for $i \leftarrow 1$ to $n$
    $L[i] \leftarrow 0$; $R[i] \leftarrow 0$
    $Q$.insert($i$, $f[i]$)
  for $i \leftarrow n+1$ to $2n-1$
    $x \leftarrow Q$.ExtractMin()
    $y \leftarrow Q$.ExtractMin()
    $f[i] \leftarrow f[x] + f[y]$
    $Q$.insert($i$, $f[i]$)
    $L[i] \leftarrow x$; $P[x] \leftarrow i$;
    $R[i] \leftarrow y$; $P[y] \leftarrow i$;
  $P[2n-1] \leftarrow 0$
```

BuildHuffman() performs $2n - 1$ Q.insert and $2n - 2$ Q.ExtractMin. What is a good data structure to use for the container Q? *Priority queue with binary heap underneath*, for which both Insert and ExtractMin costs $O(\log n)$. In total, the building of Huffman tree costs $O(n \log n)$. Here's an example C++ implementation with Standart Template Library:



```cpp
// freq,L,  R, P of size 2n-1
void BuildHuffman(vector<int> &freq, vector<int> &L, vector<int> &R, vector<int> &P)
{
    int n = freq.size()/2 + 1;
    using PII = pair<int,int>;
    priority_queue<PII,vector<PII>,greater<PII>> Q; // pair {freq, index}
    for (int i=0; i<n; i++)  Q.emplace(freq[i], i);
    for (int i=n; i<2*n-1; i++) {
        auto x = Q.top(); Q.pop();  auto y = Q.top(); Q.pop();
        freq[i] = x.first + y.first;
        Q.emplace(freq[i], i);
        L[i] = x.second; P[x.second] = i;
        R[i] = y.second; P[y.second] = i;
    }
    P[2*n-2] = 0;
}
int main()
{
    vector<int> f;
    int a;
    ifstream freq_file("freq.txt");
    while (freq_file >> a) {
        f.push_back(a);
    }
    int n = f.size();
    f.resize(2*n-1);
    vector<int> L(2*n-1), R(2*n-1), P(2*n-1);
    BuildHuffman(f, L, R, P);
    for (int i=0; i<2*n-1; i++) printf("%3d %3d %3d %3d %3d\n",i,f[i],L[i],R[i],P[i]);
}
```

Stable Matching