L~~11~~ 5800

12

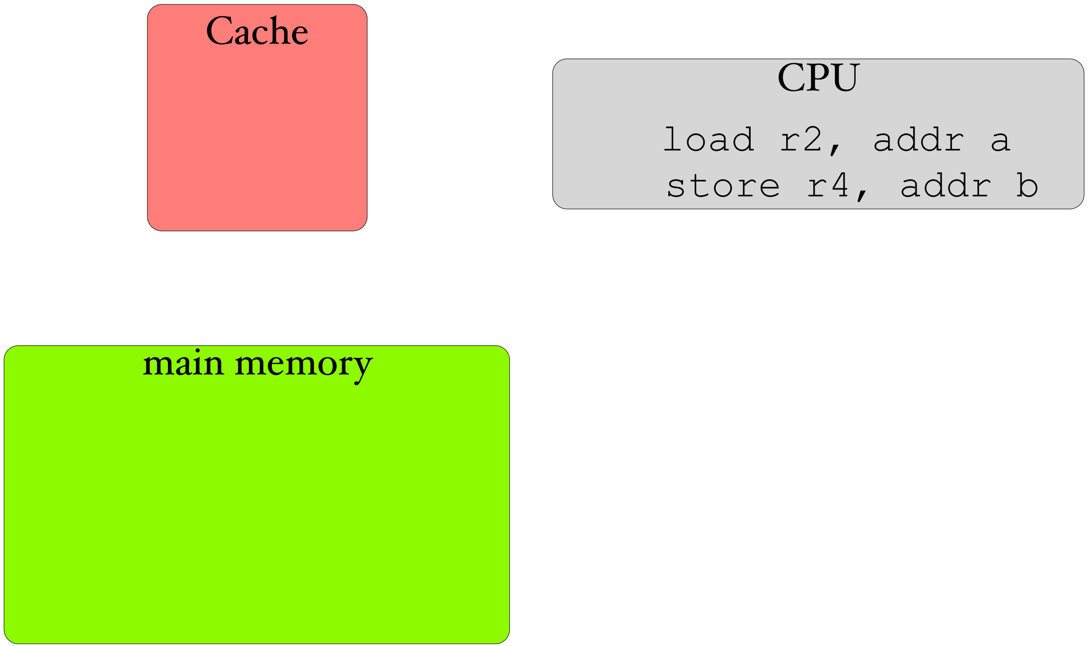
feb 27 2022

shelat

Greedy is only good for certain problems

caching

cache hit



question:

question:

How do we manage a fully-associate cache?

When it is full, which element do we replace?

problem statement

input:

output:

cache is

problem statement

input: K, the size of the cache

$d_1, d_2, \ldots, d_m$ memory accesses

output: schedule for that cache that minimizes # of cache misses while satisfying requests

cache is fully associative, line size is 1

contrast with reality

contrast with reality

In a real situation, we may not know the future memory access patterns.

Some caches have additional restrictions, like line-size, associativity, etc.

However, this algorithm can still be used to compare a real-world algorithm against the optimum cache miss rate possible.

Belady eviction rule

Belady eviction rule

Replace the element in the cache that is accessed "farthest into the future"

example

cache

a

b

c

a b c d a d e a d b a e c e a

example

Cache operations: nop nop nop Evict c for d.

cache

a a
b b
c d

Memory accesses: a b c d a d e a d b a e c e a

example

Cache operations: nop Evict (c,d) Evict (b,e)

cache			
	a	a	a
	b	b	e
	c	d	d

Memory accesses: a b c d a d e a d b a e c e a

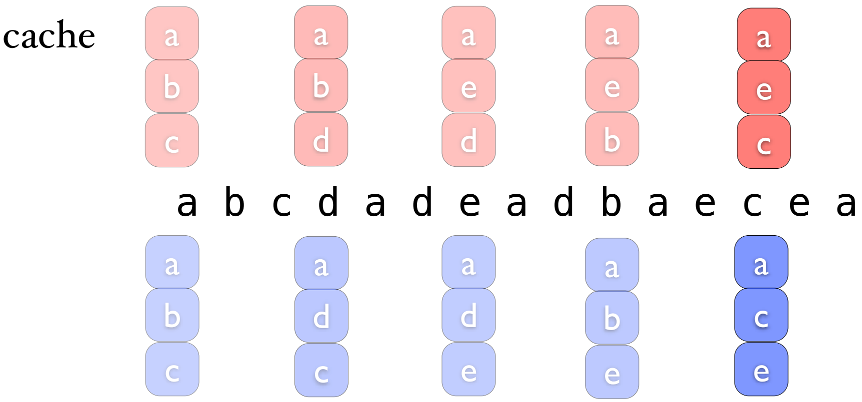
example

Cache operations: nop | Evict (c,d) | Evict (b,e) | Evict (d,b)

cache	nop	Evict (c,d)	Evict (b,e)	Evict (d,b)
	a	a	a	a
	b	b	e	e
	c	d	d	b

Memory accesses: a b c d a d e a d b a e c e a

example



Here is an alternate optimal set of cache operations.

Surprising theorem

Surprising theorem

The schedule S_{ff} produced by the Belady "farthest in the future" eviction rule is optimal.

schedule

Schedule for access pattern $d_1,d_2,\ldots,d_n$:

Reduced schedule:

schedule

Schedule for access pattern $d_1,d_2,\ldots,d_n$:

A list of instructions for each access that is either “NOP” or “evict x for y”

Reduced schedule:

schedule

Schedule for access pattern $d_1, d_2, \ldots, d_n$:

A list of instructions for each access that is either "NOP" or "evict x for y"

Reduced schedule:

A schedule in which"evict x for y" instruction only occurs when y is accessed.

schedule

Schedule for access pattern $d_1, d_2, \ldots, d_n$:

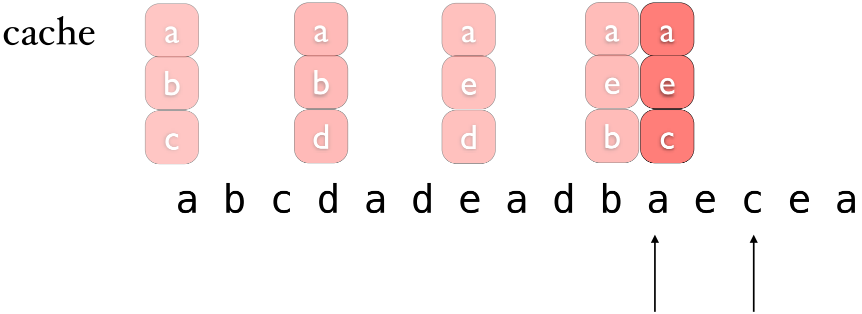
A list of instructions for each access that is either "NOP" or "evict x for y"

Reduced schedule:

A schedule in which"evict x for y" instruction only occurs when y is accessed.

Note: any schedule can be transformed into a reduced schedule with the same or fewer cache misses.
(Idea: starting at the end, defer "evict…t" until y is read)

Non-Reduced Schedule example



Example of a non-reduced schedule.
At this point, the cache evicts (b,c) when "a" is being accessed. It is possible to delay this eviction until "c" is accessed, thereby leading to a reduced schedule.

Exchange lemma

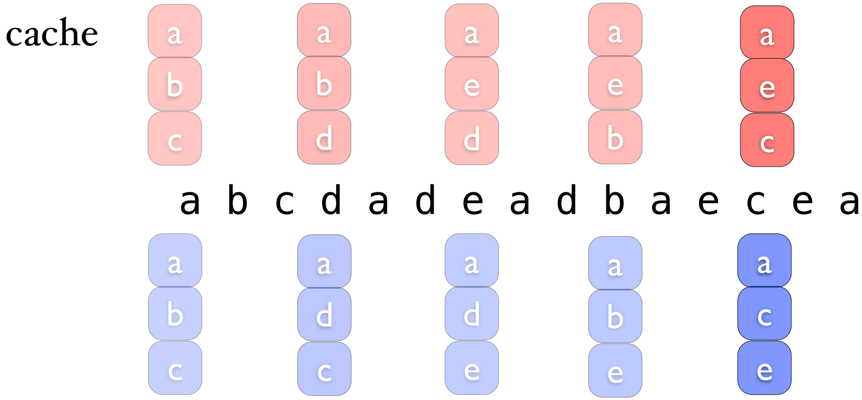
Exchange lemma

Let S be a reduced schedule that agrees with S_{ff} on the first j accesses.

Then there exists a schedule S' that agrees with S_{ff} on the first j+1 accesses and has the same or fewer misses.

What does it mean for 2 schedules to agree?

A schedule is a sequence of cache instructions:
NOP,NOP,NOP,evict(c,d),NOP,NOP,...



For example, these two schedules agree on the first three operations.

Some optimal schedule.

S^*

S_{ff}

Some optimal schedule.

S^*

S_1

Agrees with S_{ff} on the first access. Can be constructed by applying the Lemma to S^* which agrees on 0 accesses.

S_{ff}

Some optimal schedule.

S^* $\quad S_1 \quad S_2$ $\qquad\qquad S_{\text{ff}}$

Agrees with S_{ff} on the first access. Can be constructed by applying the Lemma to S^* which agrees on 0 accesses.

Agrees with S_{ff} on the first two accesses.

Some optimal schedule.

S^* S_1 S_2 S_3 S_{n-1} S_{ff}

Agrees with S_{ff} on the first access. Can be constructed by applying the Lemma to S^* which agrees on 0 accesses.

Agrees with S_{ff} on the first two accesses.

Agrees with S_{ff} on the first three accesses.

S_{ff} has the same number of cache misses as S^*.

Some optimal schedule.

S^* S_1 S_2 S_3 S_{n-1} S_{ff}

Agrees with S_{ff} on the first access. Can be constructed by applying the Lemma to S^* which agrees on 0 accesses.

Agrees with S_{ff} on the first two accesses.

Agrees with S_{ff} on the first three accesses.

S_{ff} has the same number of cache misses as S^*.

$$miss(S^*) \geq miss(S_1) \geq miss(S_2) \geq \cdots \geq miss(S_n)$$

Some optimal schedule.

S^*

S_{ff}

Since S^* is optimal, this means that all of these relations need to be equality.

This also means the S_{ff} is therefore optimal.

Some optimal schedule.

S^*

S_{ff}

$$miss(S^*) \geq miss(S_1) \geq miss(S_2) \geq \cdots \geq miss(S_n) = miss(S_{ff})$$

Since S^* is optimal, this means that all of these relations need to be equality.

This also means the S_{ff} is therefore optimal.

Proof of Lemma

Let S be a reduced sched that agrees with S_{ff} on the first j items.
There exists a reduced sched **S'** that agrees with S_{ff} on the first j+1 items and has the same or fewer #misses as S.

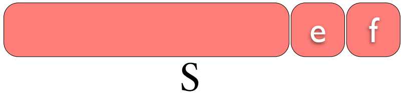
Proof of Lemma

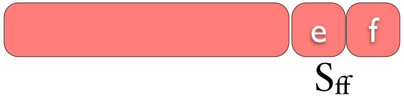
Let S be a reduced sched that agrees with S_{ff} on the first j items.
There exists a reduced sched **S'** that agrees with S_{ff} on the first j+1 items and has the same or fewer #misses as S.

At time j, both S and S_{ff} have the same state.
Let d be the element accessed at time j+1.

Proof of lemma

State of the cache after J operations under the two schedules.

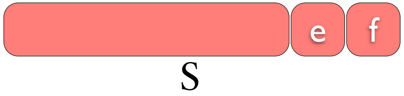


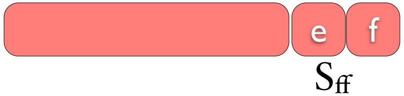


easy case 1

Proof of lemma

State of the cache after J operations under the two schedules.





easy case 1 d is in the cache.

Proof of lemma

State of the cache after J operations under the two schedules.

easy case 1 d is in the cache.

Both S and S_{ff} agree since both do NOPs at j+1.

Proof of lemma

State of the cache after J operations under the two schedules.

easy case 2

Proof of lemma

State of the cache after J operations under the two schedules.

easy case 2 d is not in the cache, but both schedules “evict e for d.”

Proof of lemma

State of the cache after J operations under the two schedules.

S S_{ff}

easy case 2 d is not in the cache, but both schedules “evict e for d.”

Both S and S_{ff} agree at j+1.

Proof of lemma

case 3

Proof of lemma

case 3 S does evict(d,e), and S_{ff} does evict(f,e)

Proof of lemma

case 3 S does evict(d,e), and S_{ff} does evict(f,e)

The state of the cache after this operation:

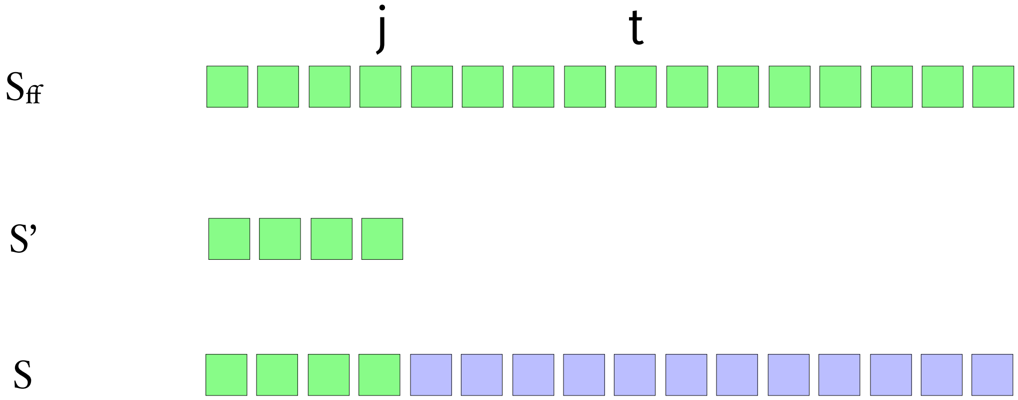
Proof of lemma

case 3 S does evict(d,e), and S_{ff} does evict(f,e)

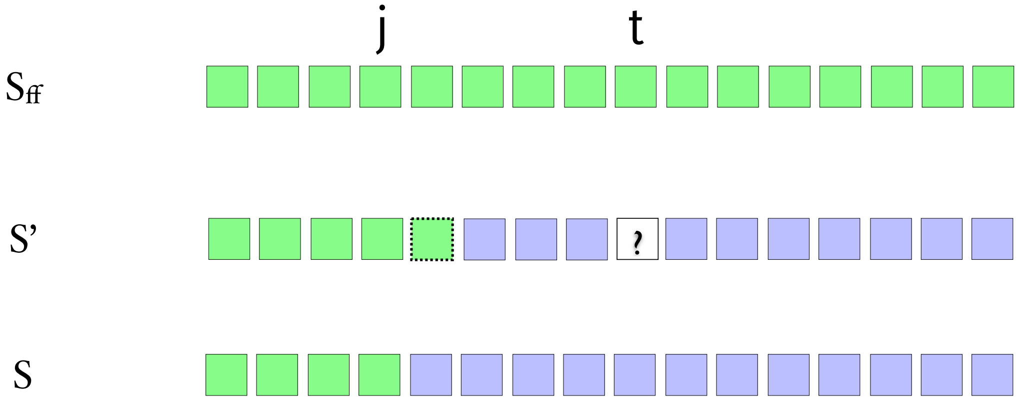
The state of the cache after this operation:

Challenge: the lemma requires us to find some schedule S' that agrees with S_{ff} and has the same or fewer misses as S.

Timeline

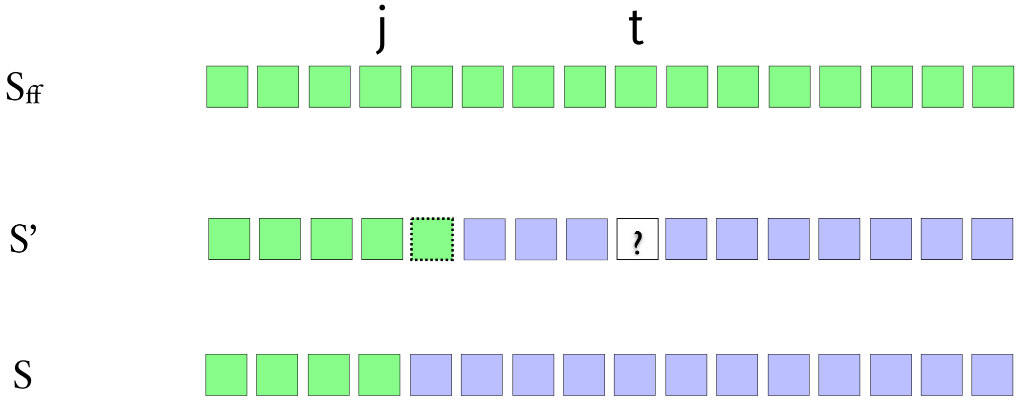


Timeline



Copy j+1 from S_{ff}. Then copy from S until t (the first time that either e or f are involved). Then copy from S until the end.

Timeline



Copy j+1 from S_{ff}. Then copy from S until t (the first time that either e or f are involved). Then copy from S until the end.

Challenge: Argue that S' has the same misses as S.

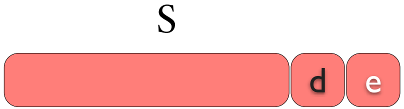
Proof of lemma

Let t be the first access that either e or f are involved.

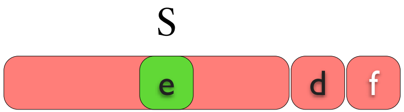
What if t is “access e”:

Proof of lemma

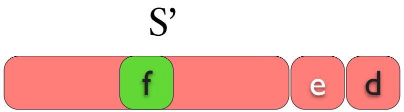
What if t = access e:



S needs to evict some element to load e.
If it evicts(f,e), then S’ can do a NOP.



If it evicts(h,e) $h \neq f$, S’ can evict(h,f)
and maintain equality of the cache.



Proof of lemma

what if t=access f ?

Proof of lemma

what if t=access f ?

This case is impossible because f is accessed “farthest in the future.”

Proof of lemma

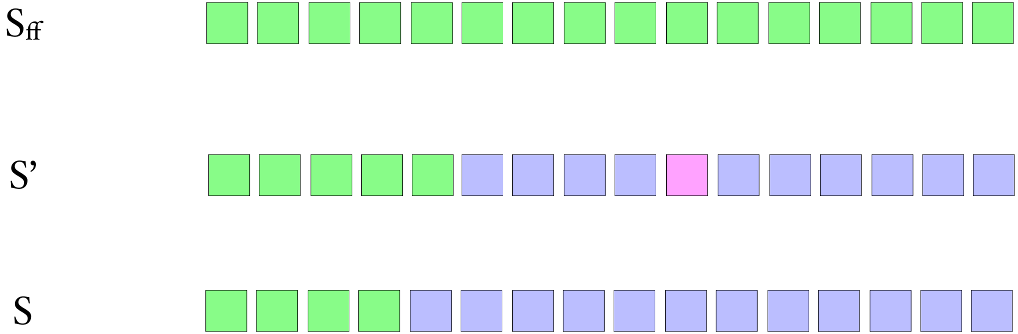
what if t is evict(f,x) ?

Proof of lemma

what if t is evict(f,x) ?

Then S’ can evict(e,x) and have the same cache state.

What have we shown



Let S be a reduced sched that agrees with S_{ff} on the first j items. There exists a reduced sched **S'** that agrees with S_{ff} on the first j+1 items and has the same or fewer #misses as S.

Let S be a reduced sched that agrees with S_{ff} on the first j items.
There exists a reduced sched **S'** that agrees with S_{ff} on the first j+1 items and has the same or fewer #misses as S.

$$S^* \qquad\qquad\qquad S_{\mathrm{ff}}$$

Recap

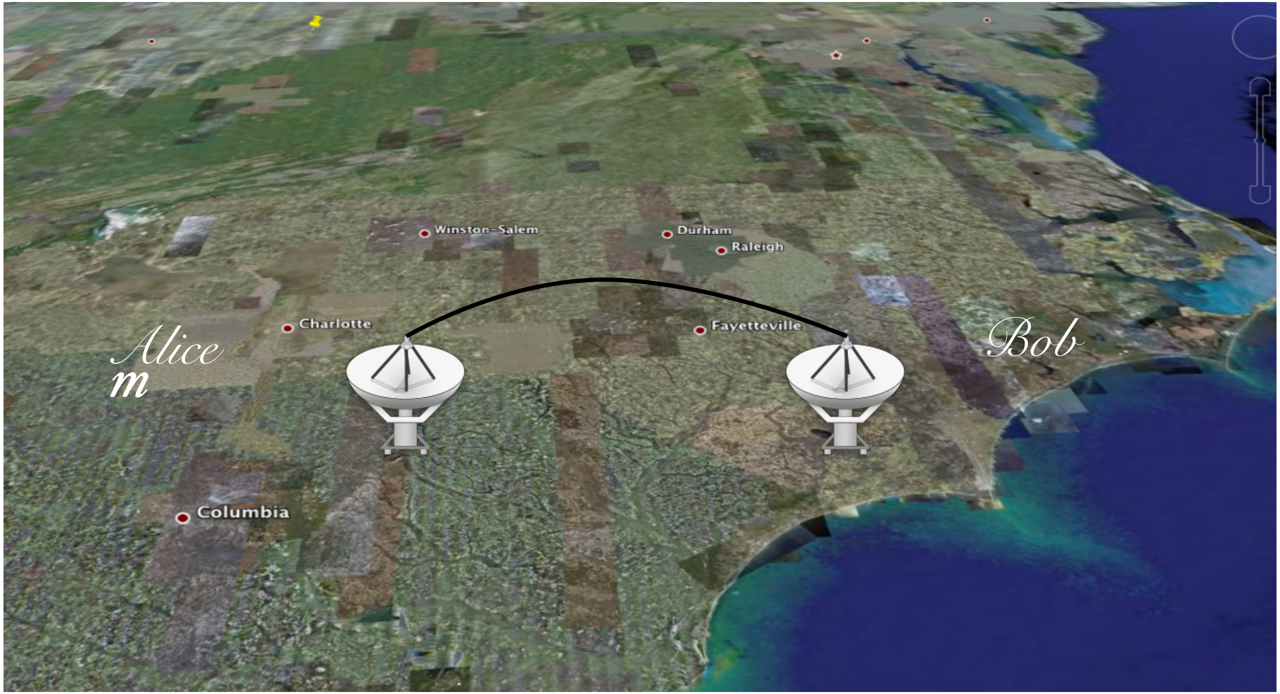
The greedy algorithm is quite simple.

But the analysis for why the solution works is more subtle and complicated.

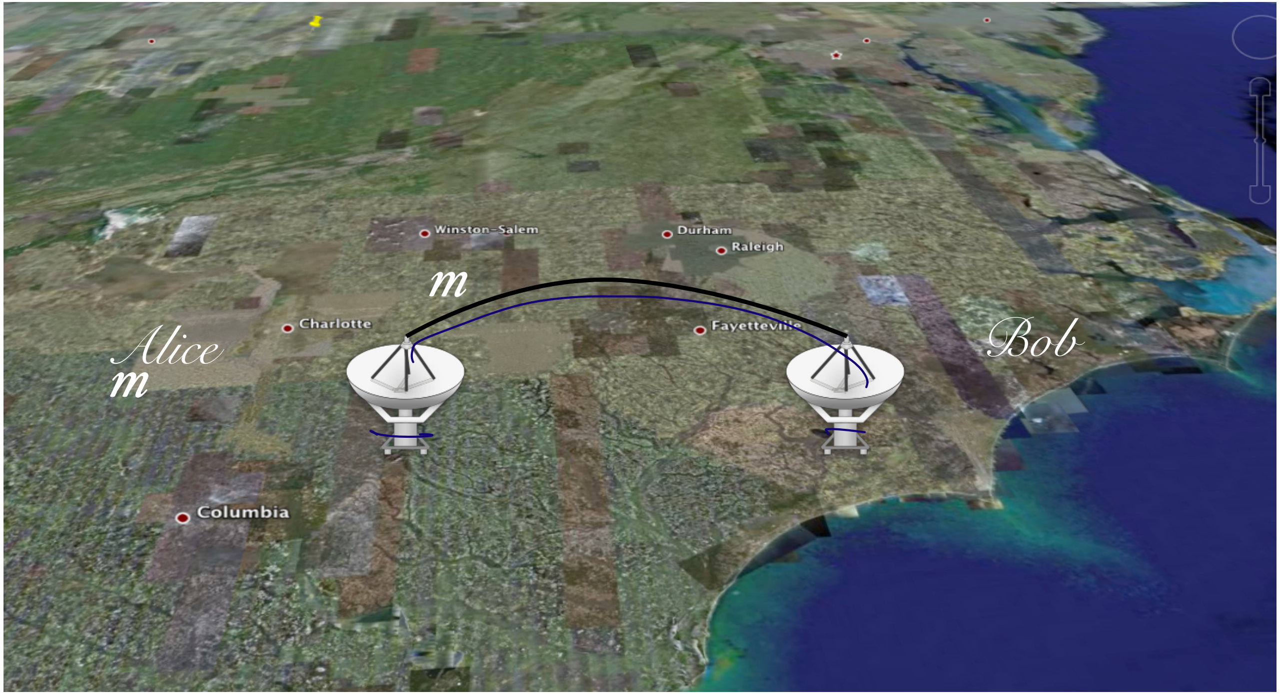
In this case, we had to apply the exchange lemma multiple times to prove optimality.

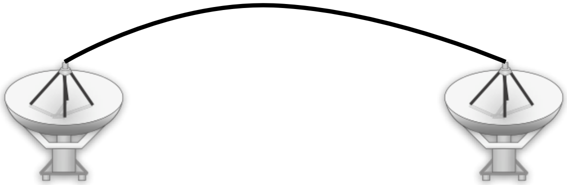
Huffman Coding

SAMUEL MORSE.



Winston-Salem
Durham
Raleigh
Charlotte
Fayetteville
Alice
m
Bob
Columbia

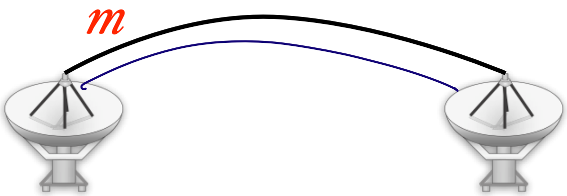

Winston-Salem
Durham
Raleigh
Charlotte
Fayetteville
Columbia
Alice
m
m
Bob



MOSCOW — President Vladimir V. Putin's typically theatrical order to withdraw the bulk of Russian forces from ~~Syria~~, a process that the Defense Ministry said it began on Tuesday, seemingly caught Washington, Damascus and everybody in between off guard — just the way the Russian leader likes it.

By all accounts, Mr. Putin delights at creating surprises, reinforcing Russia's newfound image as a sovereign, global heavyweight and keeping him at the center of world events.

cost of sending the message should be optimized



MOSCOW — President Vladimir V. Putin’s typically theatrical order to withdraw the bulk of Russian forces from Syria, a process that the Defense Ministry said it began on Tuesday, seemingly caught Washington, Damascus and everybody in between off guard — just the way the Russian leader likes it.

By all accounts, Mr. Putin delights at creating surprises, reinforcing Russia’s newfound image as a sovereign, global heavyweight and keeping him at the center of world events.

Characters in the msg

$c \in C$ $\quad f_c \qquad\qquad T$

```
e:  235
i:  200
o:  170
u:  87
p:  78
g:  47
b:  40
f:  24
```

881

code

$c \in C$	f_c	T	ℓ_c
e:	235	000	3
i:	200	001	3
o:	170	010	3
u:	87	011	3
p:	78	100	3
g:	47	101	3
b:	40	110	3
f:	24	111	3

881

def: cost of an encoding

times c appears in the m

$$B(T,\{f_c\}) = \sum_{c\in C} f_c \cdot \ell_c$$

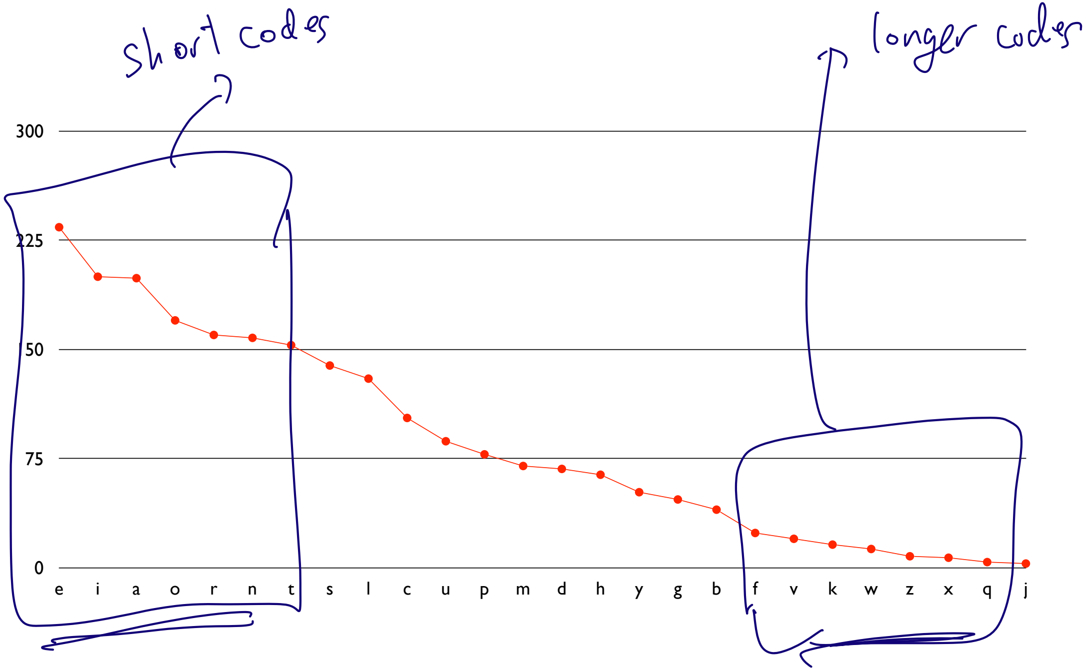
length of the encoding of c.

$c \in C$	f_c	T	ℓ_c
e:	235	000	3
i:	200	001	3
o:	170	010	3
u:	87	011	3
p:	78	100	3
g:	47	101	3
b:	40	110	3
f:	24	111	3

881 · 3 = 2643

character frequency

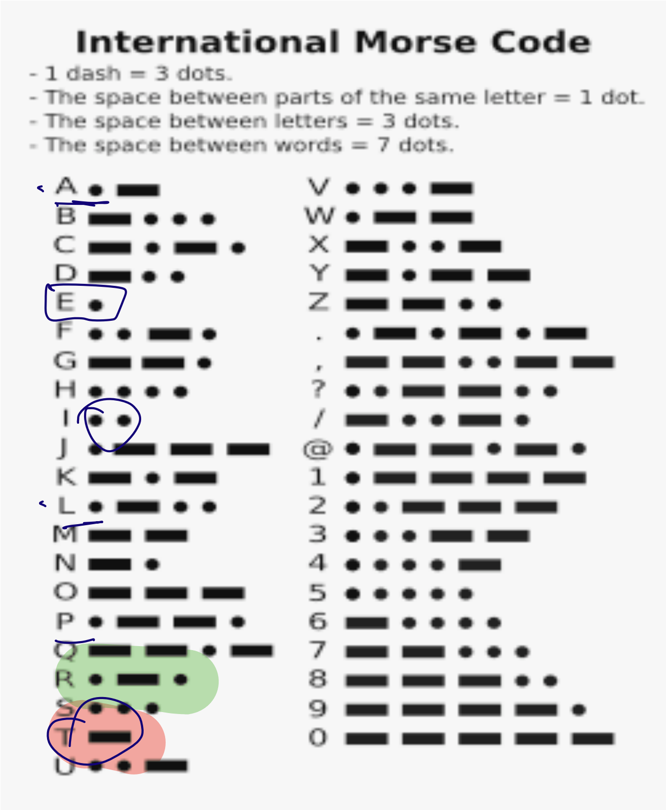
e: 234803
i: 200613
a: 198938
o: 170392
r: 160491
n: 158281
t: 152570
s: 139238
l: 130172
c: 103307
u: 87211
p: 78077
m: 70504
d: 68007
h: 64165
y: 51527
g: 47011
b: 40351
f: 24110
v: 20103
k: 16012
w: 13825
z: 8439
x: 6926
q: 3729
j: 3075



Morse code

image http://en.wikipedia.org/wiki/Morse_code

Morse code



• — • —

A A

ETET

RT

ENT

AET.

EK

def: prefix-free code

For any two different characters x, y in the alphabet C,

CODE(x) is NOT a prefix of CODE(y).

def: prefix-free code

$$\forall x, y \in C, x \neq y \implies \textsc{code}(x) \text{ not a prefix of } \textsc{code}(y)$$

not

equal

def: prefix code

$$\forall x, y \in C, x \neq y \implies \textsc{code}(x) \text{ not a prefix of } \textsc{code}(y)$$

e:	235	0
i:	200	10
o:	170	110
u:	87	1110
p:	78	11110
g:	47	111110
b:	40	1111110
f:	24	11111110

Example of a prefix free code

decoding a prefix code

e:	235	0
i:	200	10
o:	170	110
u:	87	1110
p:	78	11110
g:	47	111110
b:	40	1111110
f:	24	11111110

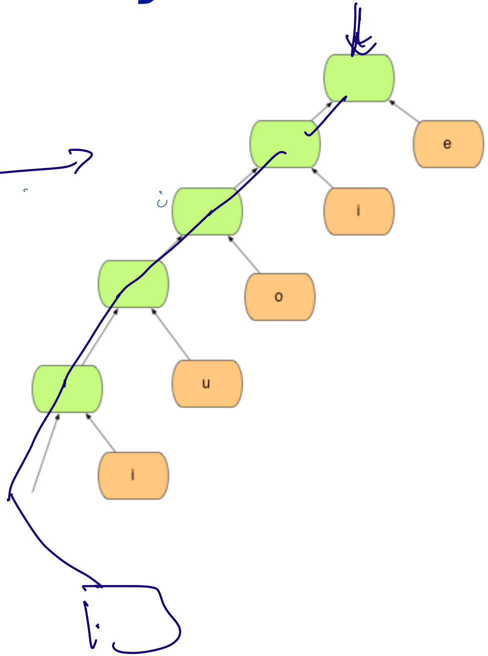
111111010111110

B I G

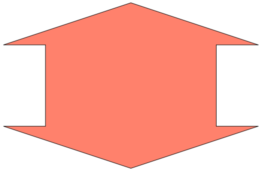
Prefix code to binary tree

e:	235	0
i:	200	10
o:	170	110
u:	87	1110
p:	78	11110
g:	47	111110
b:	40	1111110
f:	24	11111110

111111010111110



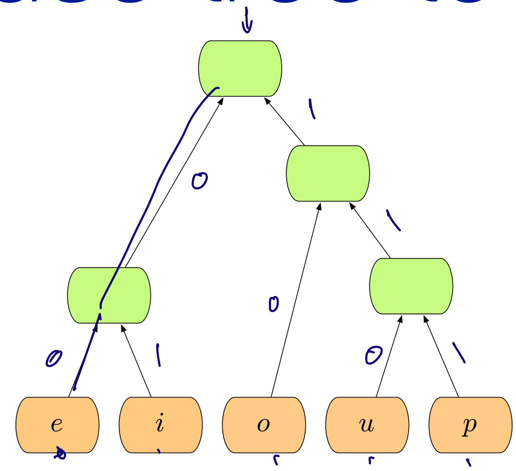
prefix code



binary tree

The prefix-free code and the binary tree are different representations of the same object.

use tree to encode



$c \in C$	f_c	T	ℓ_c
e:	235	00	2
i:	200	01	2
o:	170	10	2
u:	87	110	3
p:	78	111	3

goal

GIVEN THE character frequencies $\{f_c\}_{c \in C}$

produce a prefix-free code T for C with the smallest cost

$$\min_T B(T, \{f_c\})$$

goal

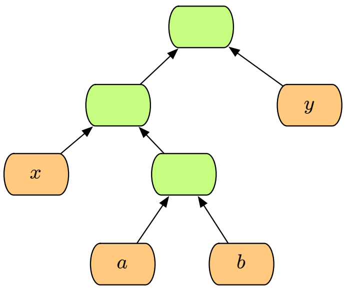
(all frequencies are > 0)

GIVEN THE CHARACTER FREQUENCIES $\{f_c\}_{c \in C}$

PRODUCE A PREFIX CODE T WITH SMALLEST COST

$$\min_T B(T, \{f_c\})$$

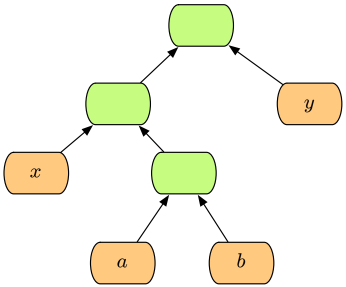
property



LEMMA: OPTIMAL TREE MUST BE FULL.

Full: means all nodes either have 0 children or 2 children.

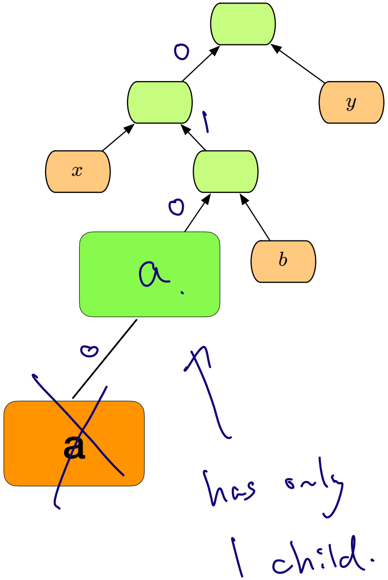
property



LEMMA:OPTIMAL TREE MUST BE FULL.

A full tree has nodes with either 0 or 2 children.

property



LEMMA:OPTIMAL TREE MUST BE FULL.

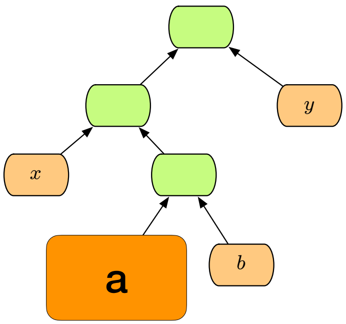
A full tree has nodes with either 0 or 2 children.

Consider a node with only 1 child.

a: 0100

a: 010

property



LEMMA:OPTIMAL TREE MUST BE FULL.

A full tree has nodes with either 0 or 2 children.

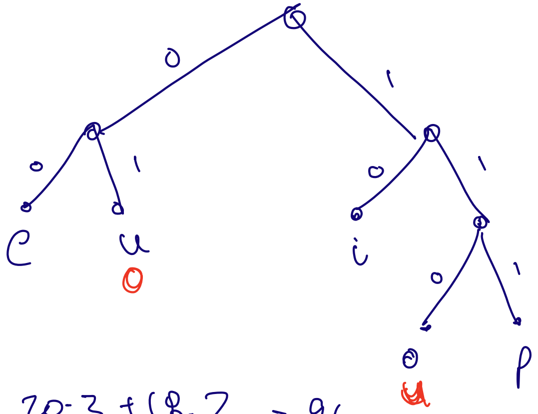
Consider a node with only 1 child.

The length of the code for this child can be reduced by replacing the parent with the child.

Thus, the cost of the code can be reduced or remain equal if the parent is replaced by the child

divide & conquer Tug of War?

Consider a "Tug of War" strategy in which we balance the weights of the teams and recurse.



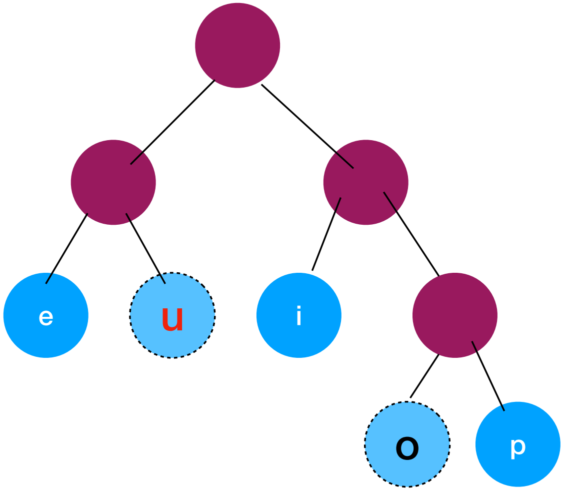
e:	32	2
i:	25	2
o:	20	3
u:	18	2
p:	5	3

Cost $20 \cdot 3 + 18 \cdot 2 = \underline{96}$

better code cost: $20 \cdot 2 + 18 \cdot 3 = \underline{94}$

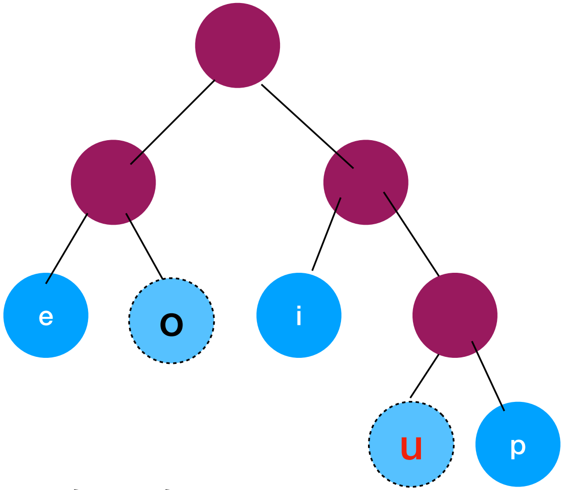
counter-example

```
e: 32  2: 64
i: 25  2: 50
o: 20  3: 60
u: 18  2: 36
p: 5   3: 15
          225
```



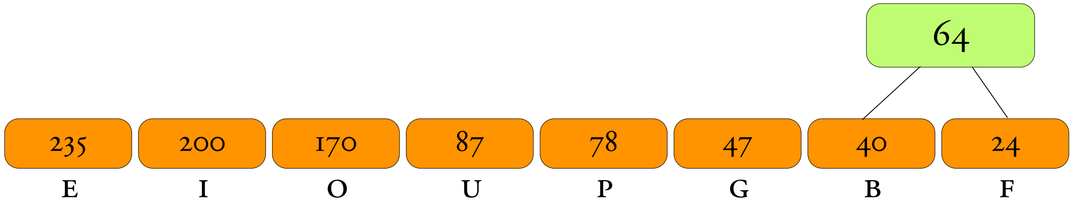
counter-example

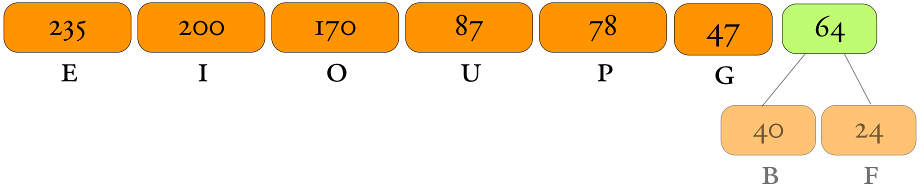
e:	32	2:	64
i:	25	2:	50
o:	20	2:	40
u:	18	3:	54
p:	5	3:	15
			223

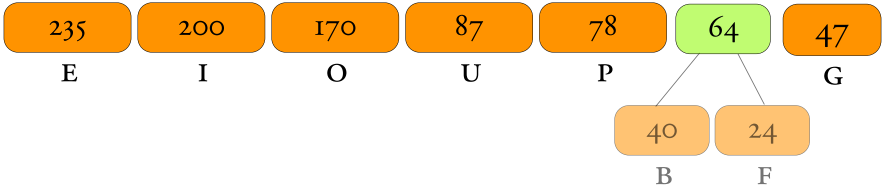


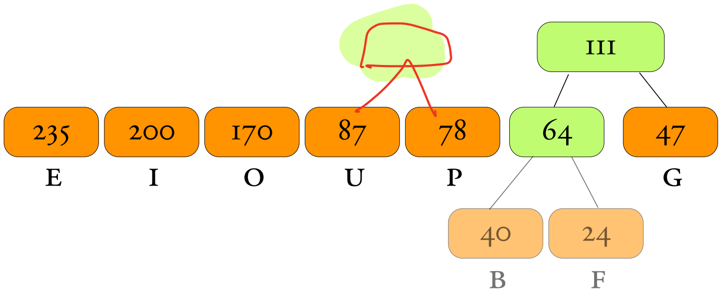
By switching {u,o}, the cost of the code can be reduced.
It can be reduced further with an optimal code.

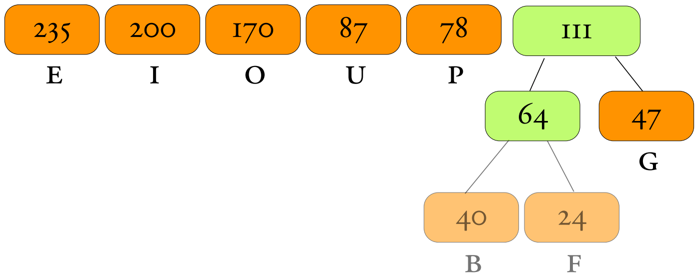
Huffman construction

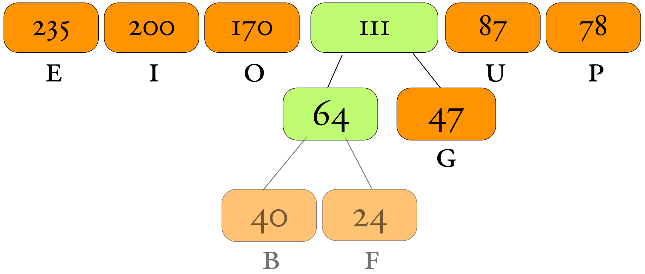


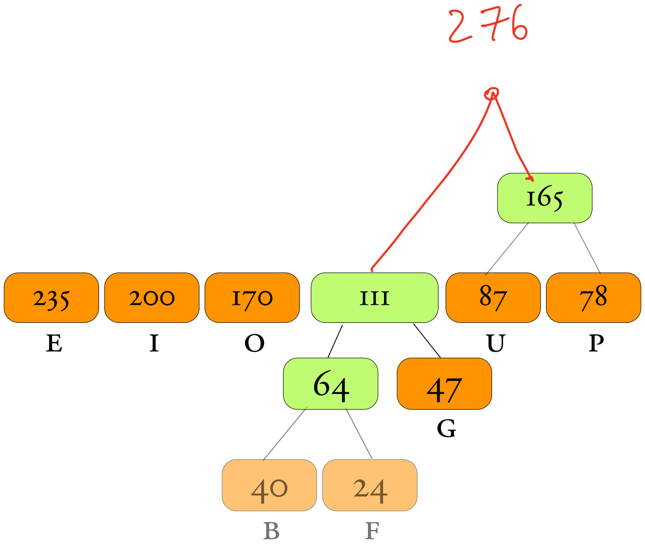

235
E
200
I
170
O
87
U
78
P
47
G
64
40
B
24
F

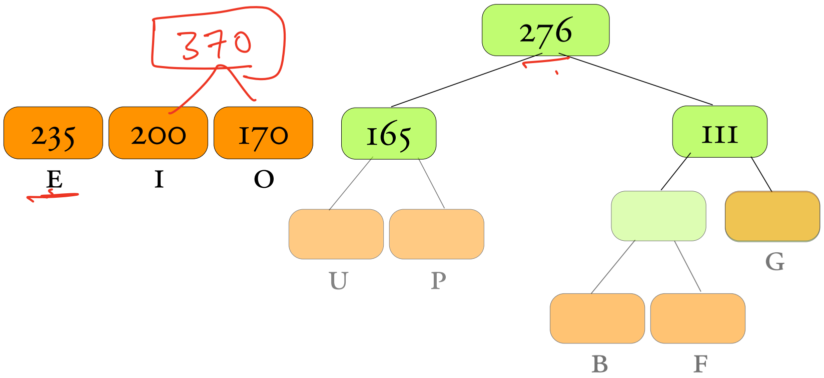

235
E
200
I
170
O
87
U
78
P
64
47
G
40
B
24
F

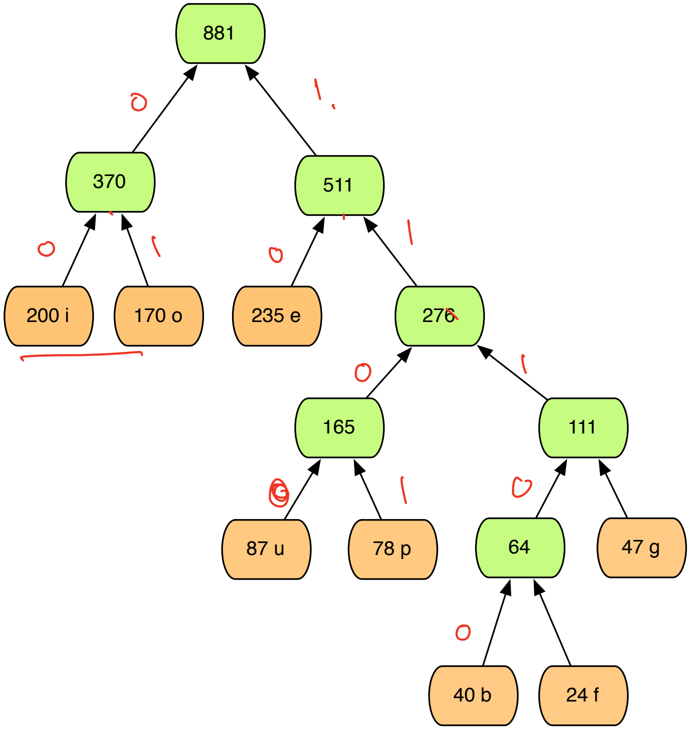

111
235
200
170
87
78
64
47
E
I
O
U
P
G
40
24
B
F

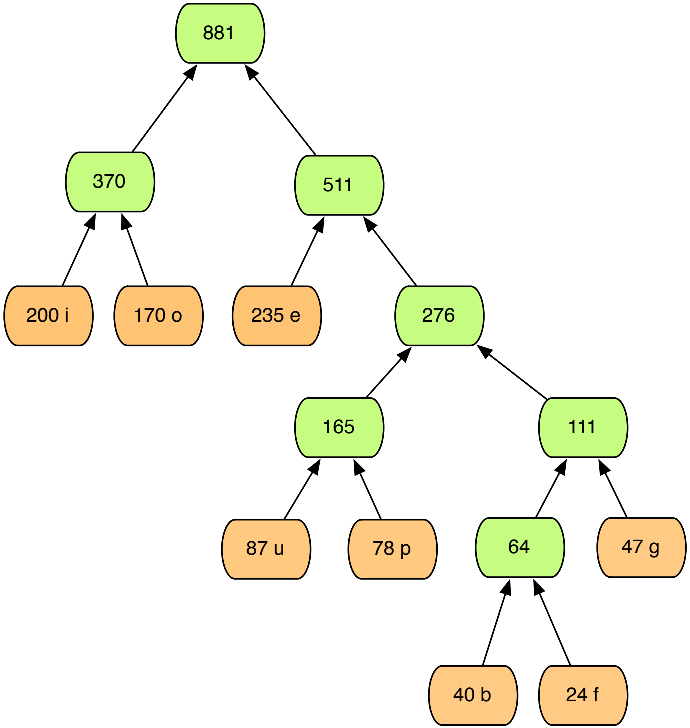

235
E
200
I
170
O
87
U
78
P
111
64
47
G
40
B
24
F


235
E
200
I
170
O
111
87
U
78
P
64
47
G
40
B
24
F


276
165
235
200
170
111
87
78
E
I
O
U
P
64
47
G
40
24
B
F

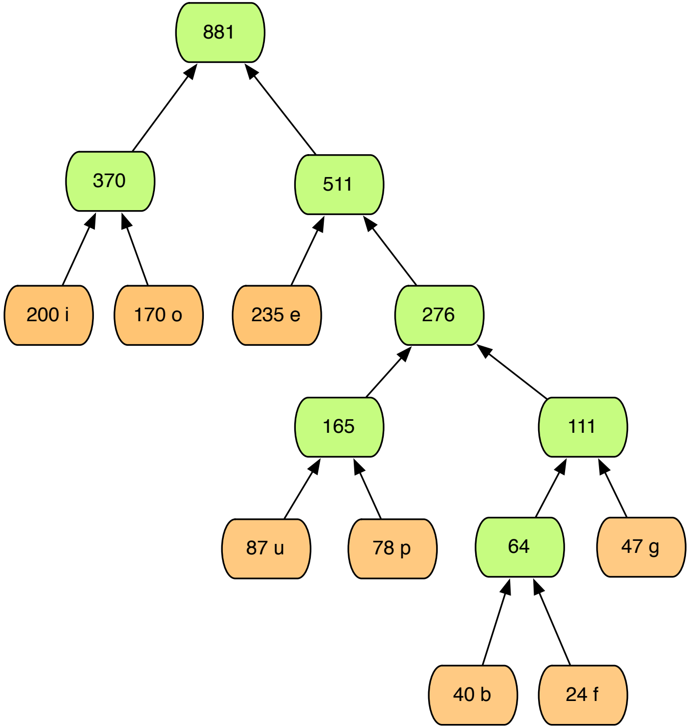

370
235
200
170
E
I
O
276
165
111
U
P
G
B
F


881
370
511
200 i
170 o
235 e
276
165
111
87 u
78 p
64
47 g
40 b
24 f
0
1
0
1
0
1
0
1
0
1
0
0



Resulting code

```
e: 235 01
i: 200 11
o: 170 10
u: 87  0011
p: 78  0010
g: 47  0000
b: 40  00011
f: 24  00010
```



e:	235	01	470
i:	200	11	400
o:	170	10	340
u:	87	0011	348
p:	78	0010	312
g:	47	0000	188
b:	40	00011	200
f:	24	00010	120
			2378

objective

objective

The goal is to prove that the procedure outlined produced an optimal code. Taking a greedy step to make the problem one size smaller is optimal.

exchange argument

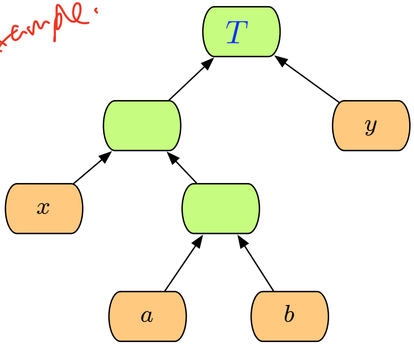
LEMMA: Let x, y be the characters in C with the smallest frequencies.

There exists an optimal prefix-free code T in which x and y are siblings.

exchange argument

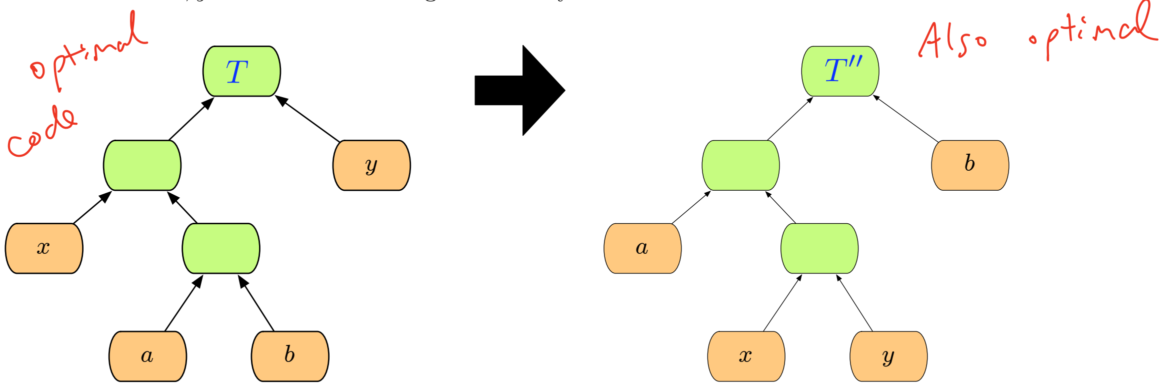
LEMMA: Let $x, y \in C$ be characters with smallest frequencies f_x, f_y. There exists an optimal prefix code T'' for C in which x, y are siblings. That is, the codes for x, y have the same length and only differ in the last bit.

Example:



exchange argument

LEMMA: Let $x, y \in C$ be characters with smallest frequencies f_x, f_y. There exists an optimal prefix code T'' for C in which x, y are siblings. That is, the codes for x, y have the same length and only differ in the last bit.



Idea: take an arbitrary optimal tree T for a prefix code and modify it into another optimal tree in which x,y are sibling children at the lowest level of the tree.

exchange argument

LEMMA: Let $x, y \in C$ be characters with smallest frequencies f_x, f_y. There exists an optimal prefix code T'' for C in which x, y are siblings. That is, the codes for x, y have the same length and only differ in the last bit.

PROOF: Let T be an optimal code.

If x, y are siblings in T, then the lemma holds.

If not, let a, b be the sibling nodes with the largest depth. These 2 must exist because T is full.

exchange argument

LEMMA: Let $x, y \in C$ be characters with smallest frequencies f_x, f_y. There exists an optimal prefix code T'' for C in which x, y are siblings. That is, the codes for x, y have the same length and only differ in the last bit.

PROOF:

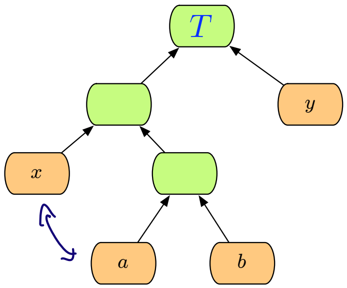
Let T be an optimal code. If x, y are siblings in T, then the lemma holds.

Otherwise, since T is full, let a, b be the sibling nodes with the largest depth. (Q: Why do a, b exist?)

exchange argument

LEMMA: Let $x, y \in C$ be characters with smallest frequencies f_x, f_y. There exists an optimal prefix code T'' for C in which x, y are siblings. That is, the codes for x, y have the same length and only differ in the last bit.

EXAMPLE OF SUCH A TREE



without loss of generality, we can say that

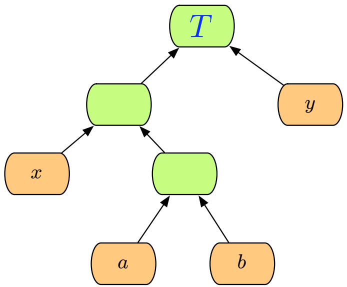
$f_x \leq f_a \qquad f_y \leq f_b.$ because x, y have the smallest frequencies.

In our first step, we will swap x and a and show that the tree must still be optimal.

exchange argument

LEMMA: Let $x, y \in C$ be characters with smallest frequencies f_x, f_y. There exists an optimal prefix code T'' for C in which x, y are siblings. That is, the codes for x, y have the same length and only differ in the last bit.

EXAMPLE OF SUCH A TREE

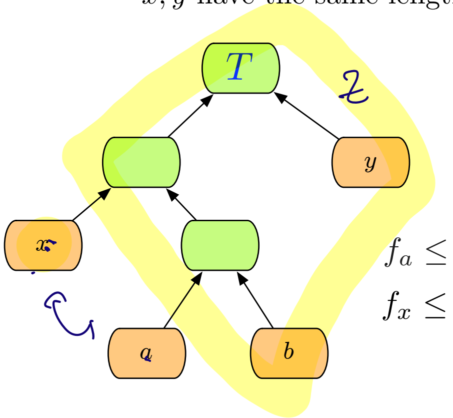


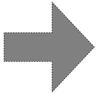
Suppose wlog that $f_x \le f_a, f_y \le f_b$

The first step is to exchange x with a to construct a new tree T'.

exchange argument

LEMMA: Let $x, y \in C$ be characters with smallest frequencies f_x, f_y. There exists an optimal prefix code T'' for C in which x, y are siblings. That is, the codes for x, y have the same length and only differ in the last bit.

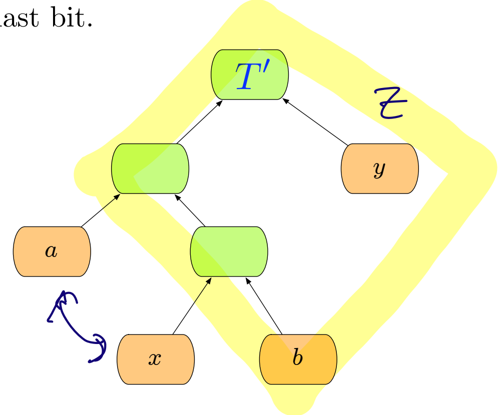




FIRST STEP

$f_a \leq f_b \qquad f_x \leq f_a$

$f_x \leq f_y \qquad f_y \leq f_b$

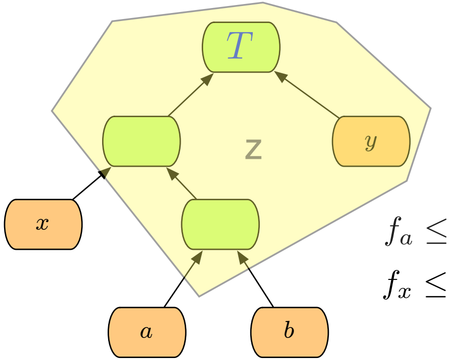


$B(T) = Z + f_x \cdot l_x + f_a \cdot l_a$

$B(T') = Z + f_x \cdot l_a + f_a \cdot l_x$

exchange argument

LEMMA: Let $x, y \in C$ be characters with smallest frequencies f_x, f_y. There exists an optimal prefix code T'' for C in which x, y are siblings. That is, the codes for x, y have the same length and only differ in the last bit.



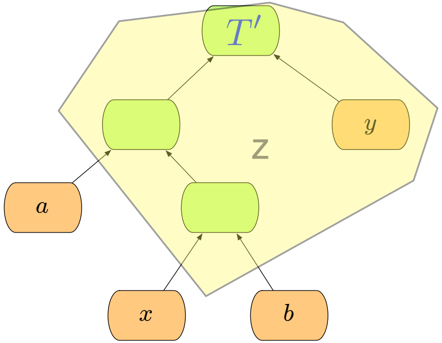
FIRST STEP

$f_a \leq f_b$

$f_x \leq f_y$

$f_x \leq f_a$

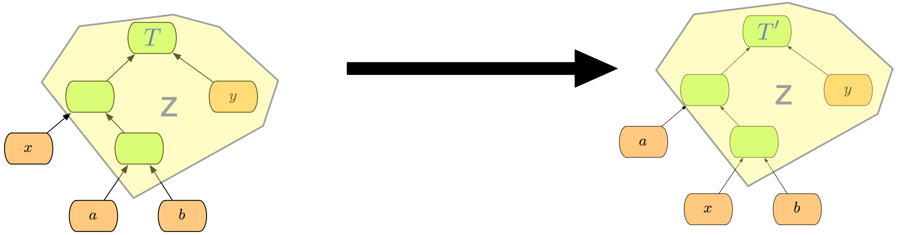
$f_y \leq f_b$



$$B(T) = Z + f_x \cdot \ell_x + f_a \cdot \ell_a$$

$$B(T') = Z + f_x \cdot \ell_a + f_a \cdot \ell_x$$

This tree is optimal.



$$B(T) = \cancel{Z} + f_x \cdot \ell_x + f_a \cdot \ell_a \qquad B(T') = Z + f_x \cdot \ell_a + f_a \cdot \ell_x$$

$$-\cancel{Z} - f_x \ell_a - f_a \cdot \ell_x$$

$$B(T) - B(T') = f_x(\ell_x - \ell_a) + f_a(\ell_a - \ell_x)$$

$$= f_x(\ell_x - \ell_a) - f_a(\ell_x - \ell_a)$$

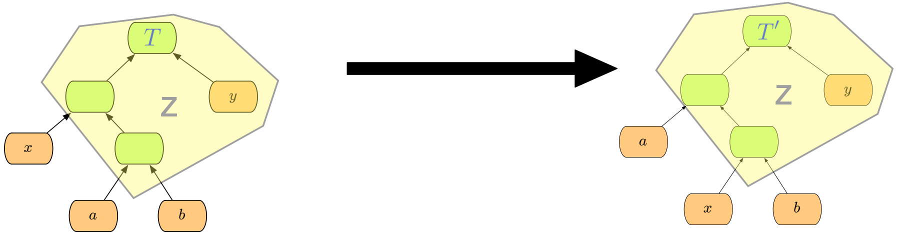
$$= \underbrace{(f_x - f_a)}_{\le 0}\underbrace{(\ell_x - \ell_a)}_{\le 0}$$

$B(T) - B(T') = 0$

because $B(T)$ is optimal

$\Rightarrow B(T')$ is optimal

This tree is optimal.

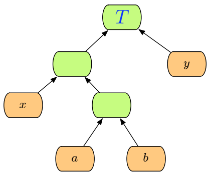


$$B(T) = Z + f_x \cdot \ell_x + f_a \cdot \ell_a \qquad B(T') = Z + f_x \cdot \ell_a + f_a \cdot \ell_x$$

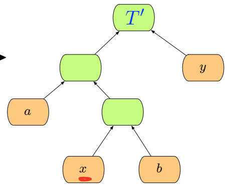
$$\begin{aligned} B(T) - B(T') &= f_x\ell_x + f_a\ell_a - f_a\ell_x - f_x\ell_a \\ &= f_x(\ell_x - \ell_a) - f_a(\ell_x - \ell_a) \\ &= (f_x - f_a)(\ell_x - \ell_a) \end{aligned}$$

Both terms must be ≤ 0 because
$f_x \leq f_a, \ell_x \leq \ell_a$

But since $B(T)$ is optimal, the product must be 0.



$f_x \le f_a$

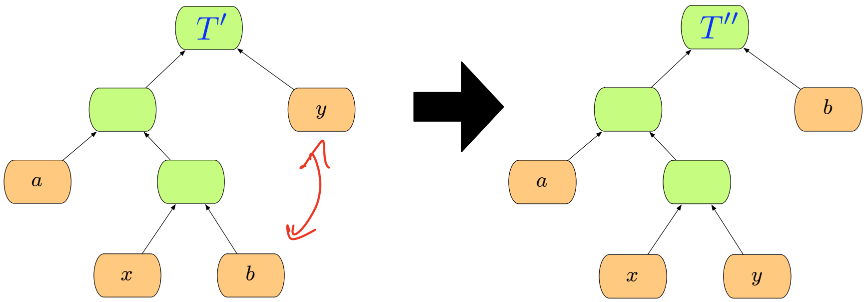


$$B(T) = \sum_c f_c \ell_c + f_x \ell_x + f_a \ell_a \qquad B(T') = \sum_c f_c \ell'_c + f_x \ell'_x + f_a \ell'_a$$

$$B(T) - B(T') = 0$$

This means that T' is also an optimal code.

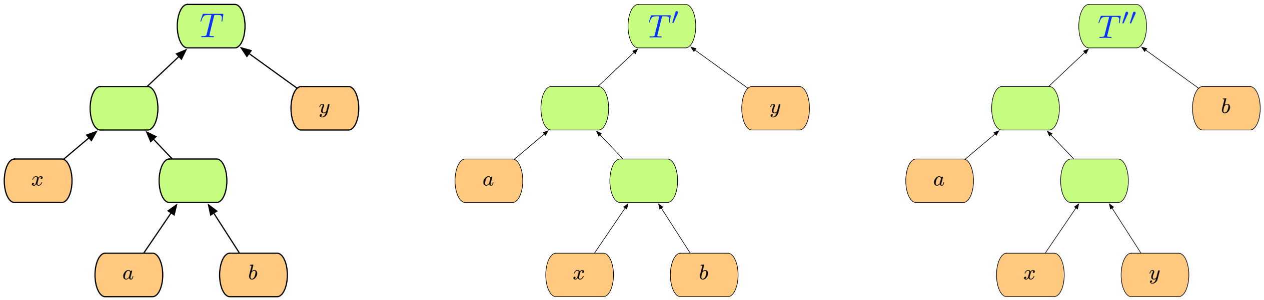
exchange argument

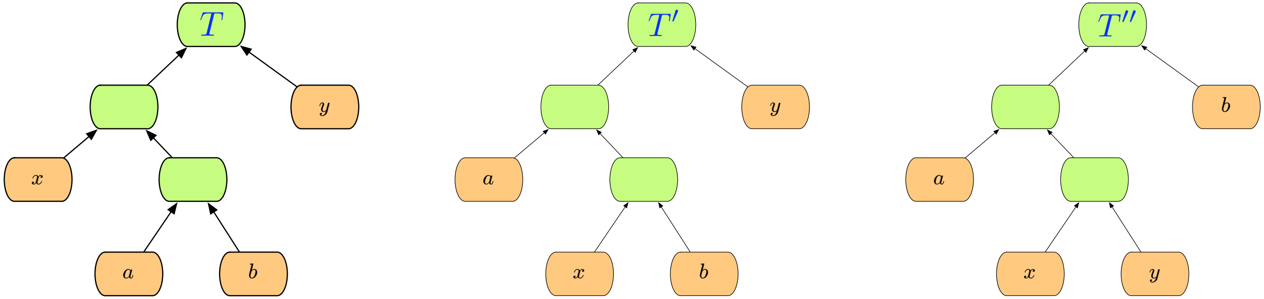


We can apply the same argument to y, b.

$$B(T') - B(T'') = 0$$

$\Rightarrow T''$ is also optimal.


T
y
x
a
b
T'
y
a
x
b
T''
b
a
x
y

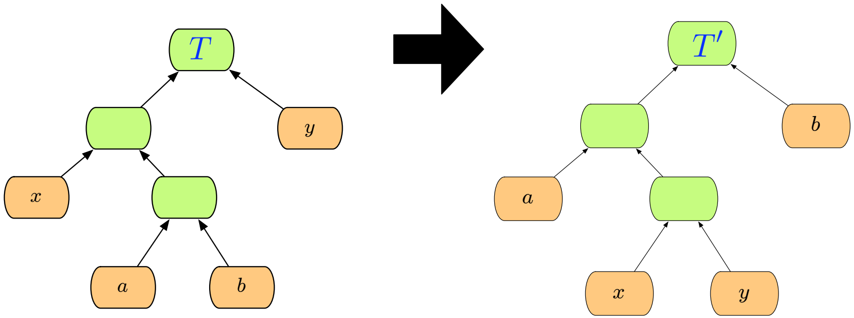


$$B(T) - B(T') \geq 0 \qquad B(T') - B(T'') \geq 0$$

T'' IS ALSO OPTIMAL

exchange argument

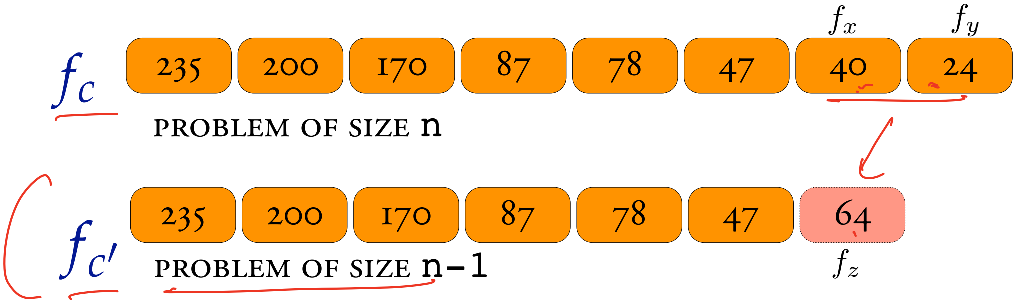
LEMMA: Let $x, y \in C$ be characters with smallest frequencies f_x, f_y. There exists an optimal prefix code T'' for C in which x, y are siblings. That is, the codes for x, y have the same length and only differ in the last bit.



optimal sub-structure

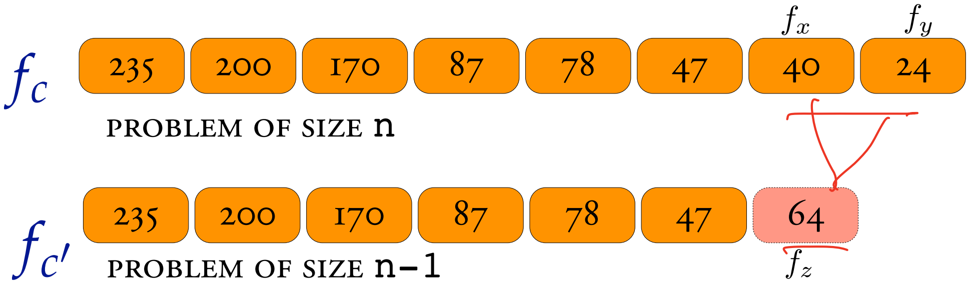
							f_x	f_y
f_c	235	200	170	87	78	47	40	24

optimal sub-structure



LEMMA: The optimal prefix free code T for $\{f_c\}$ consists of computing the optimal code for $\{f_{c'}\}$ and then replacing z with $\{x, y\}$.

optimal sub-structure

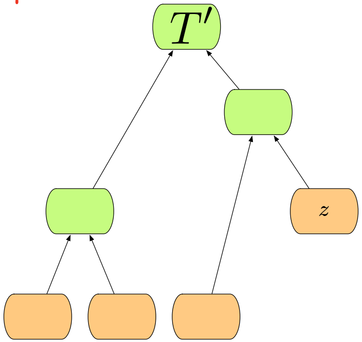


LEMMA: The optimal solution T for f_c consists of computing an optimal solution T' for $f_{c'}$ and replacing the node for z with an internal node having children x, y.

Let T' be an optimal solution for $f_{c'}$ of size n-1.

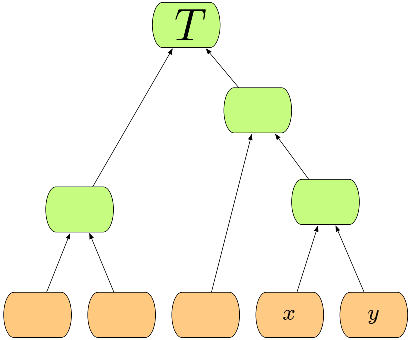
Our lemma suggests constructing T by replacing z with {x,y} leaves.

optimal for $\{f_{c'}\}$



$B(T')$

Lets analyze $B(T)$

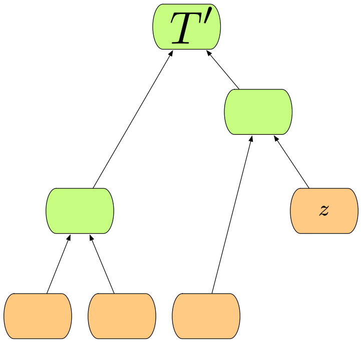


$B(T)$

$$B(T) = B(T') - f_z \cdot l_z + f_x(l_z+1) + f_y(l_z+1)$$

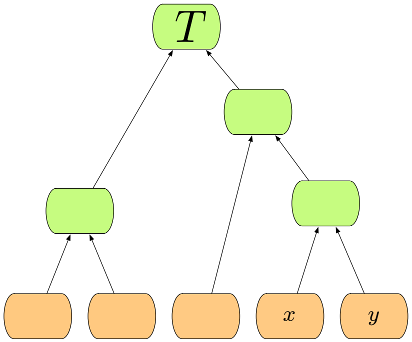
$$= B(T') \cancel{- f_z \cdot l_z} + \underbrace{\cancel{(f_x+f_y)}}_{=f_z} \cdot l_z + f_x + f_y$$

$$= B(T') + f_x + f_y$$



$B(T')$

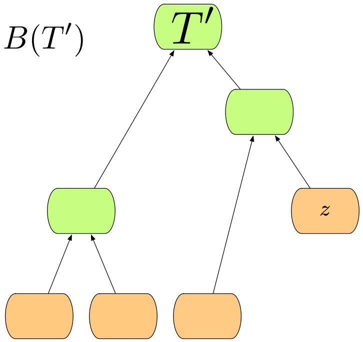
Lets analyze $B(T)$

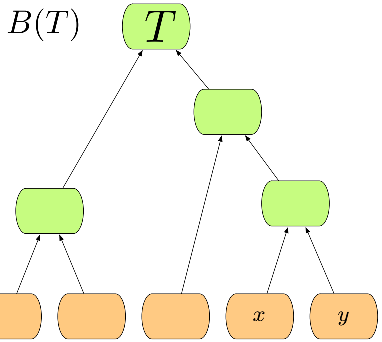


$B(T)$

$$B(T) = B(T') - f_z \ell_z + (\ell_z + 1)(f_x + f_y)$$

$$= B(T') + f_x + f_y$$





Rearranging, we get

$$B(T') = B(T) - f_x - f_y$$

Suppose T is not optimal

What does that mean?

There exists some other code $\mathcal{U}$ s.t.

$B(\mathcal{U}) < B(T)$ and in $\mathcal{U}$, x and y are siblings

by the exchange lemma.

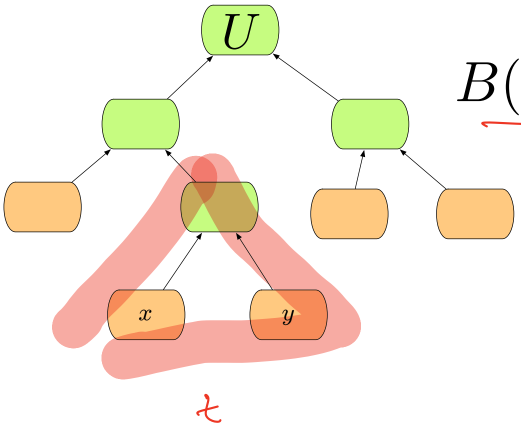
Suppose T is not optimal

What does that mean?

There exists another tree U such that $B(U) < B(T)$.

Moreover, by the exchange lemma, there exists a U' such that x,y are siblings.

Suppose T is not optimal



$$B(U) < B(T) = B(T') + f_x + f_y$$

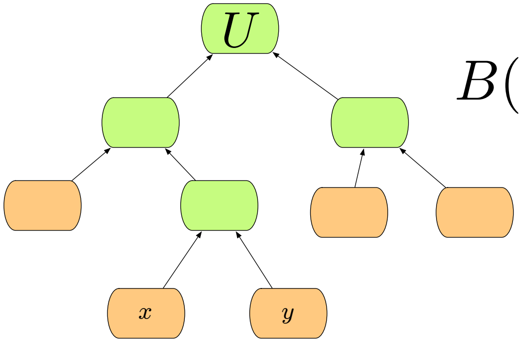
$$B(U) - f_x - f_y < B(T')$$

But then we could construct a new tree U' such that

$$B(U') < B(T').$$

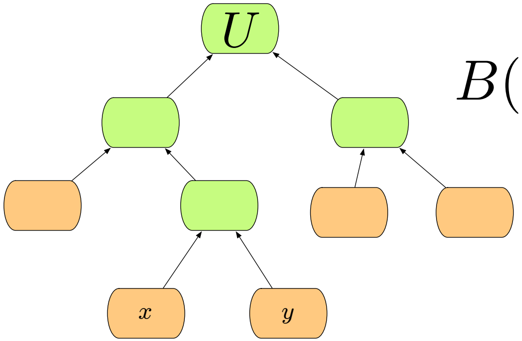
This contradicts the assumption that T' was optimal. Thus, the supposition that T is not optimal must be false.

Suppose T is not optimal



$$B(U) < B(T) = B(T') + f_x + f_y$$

Suppose T is not optimal

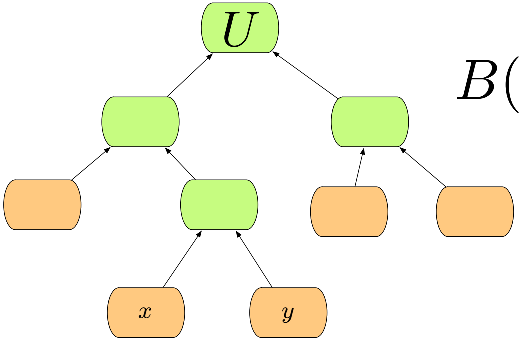


$$B(U) < B(T) = B(T') + f_x + f_y$$

This implies

$$B(U) - f_x - f_y < B(T')$$

Suppose T is not optimal

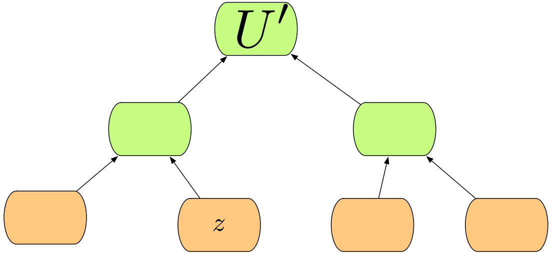


$$B(U) < B(T) = B(T') + f_x + f_y$$

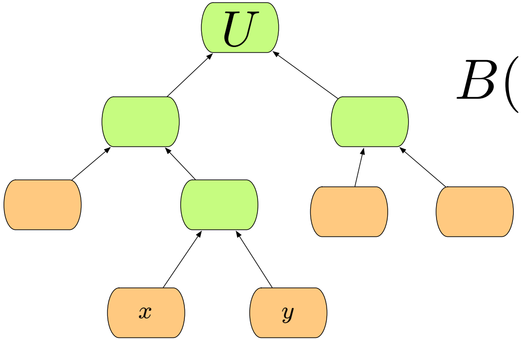
This implies

$$B(U') = B(U) - f_x - f_y < B(T')$$

$$B(U') < B(T')$$



Suppose T is not optimal

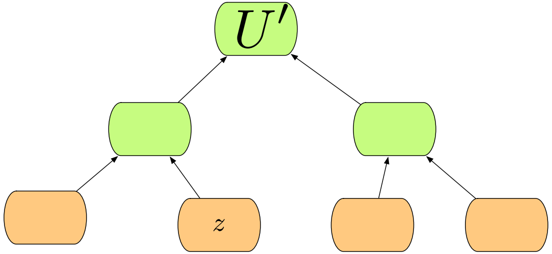


$$B(U) < B(T) = B(T') + f_x + f_y$$

This implies

$$B(U) - f_x - f_y < B(T')$$

$$B(U') < B(T')$$



Which means that T' was not optimal!
This is a contradiction, which means that our supposition (T not optimal) must be wrong.