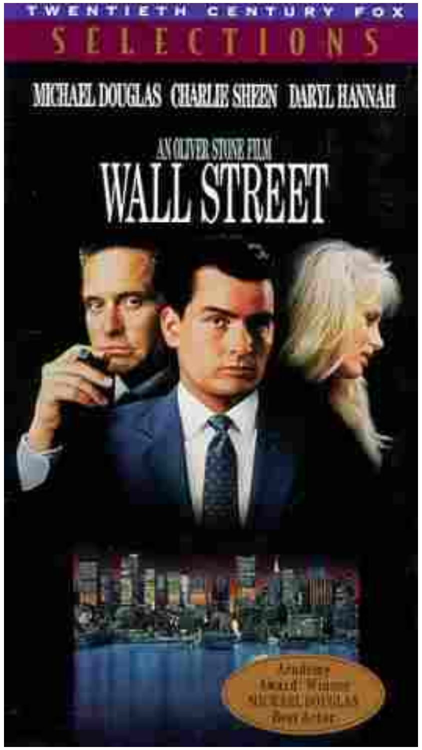
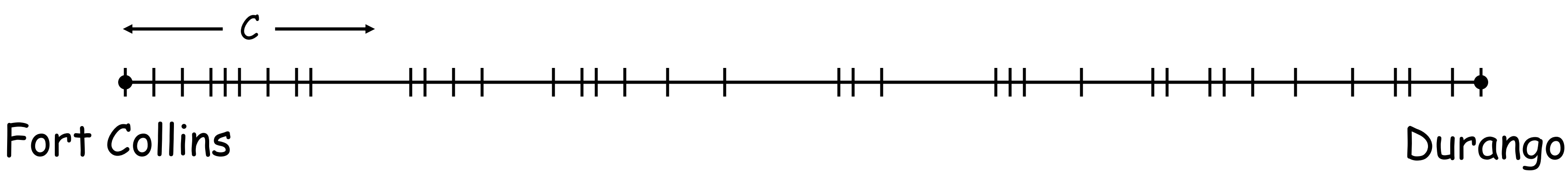
Greedy Algorithms

Kleinberg and Tardos, Chapter 4



Selecting gas stations

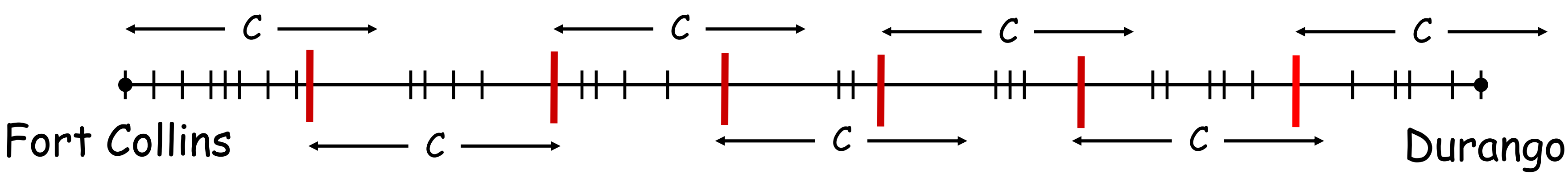
- Road trip from Fort Collins to Durango on a given route with length L, and fuel stations at positions b_i.
- Fuel capacity = C miles.
- Goal: make as few refueling stops as possible.



Selecting gas stations

- Road trip from Fort Collins to Durango on a given route with length L, and fuel stations at positions b_i.
- Fuel capacity = C.
- Goal: makes as few refueling stops as possible.

Greedy algorithm. Go as far as you can before refueling.
In general: **determine a global optimum via a number of locally optimal choices.**



Selecting gas stations: Greedy Algorithm

The road trip algorithm.

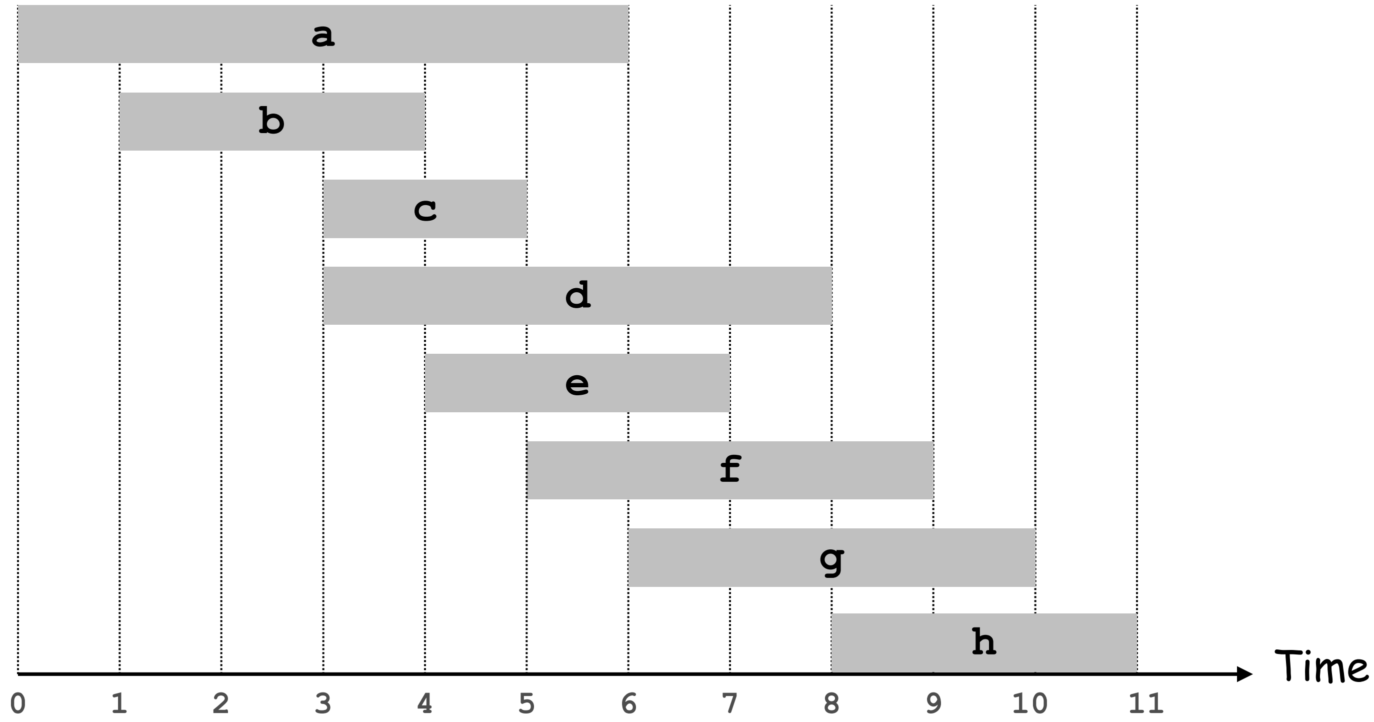
```
Sort stations so that: 0 = b₀ < b₁ < b₂ < ... < bₙ = L

S ← {0}        ← stations selected, we fuel up at home
x ← 0          ← current distance

while (x ≠ bₙ)
   let p be largest integer such that bₚ ≤ x + C
   if (bₚ = x)
      return "no solution"
   x ← bₚ
   S ← S ∪ {p}
return S
```

Interval Scheduling

- Also called activity selection, or job scheduling...
- Job j starts at s_j and finishes at f_j.
- Two jobs compatible if they don't overlap.
- Goal: find maximum size subset of compatible jobs.



Interval Scheduling: Greedy Algorithms

Greedy template. Consider jobs in some natural order. Take each job provided it's compatible with the ones already taken. Possible orders:

- [Earliest start time] Consider jobs in ascending order of s_j.
- [Earliest finish time] Consider jobs in ascending order of f_j.
- [Shortest interval] Consider jobs in ascending order of $f_j - s_j$.
- [Fewest conflicts] For each job j, count the number of conflicting jobs c_j. Schedule in ascending order of c_j.

Which of these surely don't work?
(hint: find a counter example)

Interval Scheduling: Greedy Algorithms

Greedy template. Consider jobs in some natural order. Take each job provided it's compatible with the ones already taken.

counterexample for earliest start time

counterexample for shortest interval

counterexample for fewest conflicts

Interval Scheduling: Greedy Algorithm

Greedy algorithm. Consider jobs in increasing order of finish time. Take each job provided it's compatible with the ones already taken.

```
Sort jobs by finish times so that f1 ≤ f2 ≤ ... ≤ fn.

    set of jobs selected
A ← φ
for j = 1 to n {
   if (job j compatible with A)
      A ← A ∪ {j}
}
return A
```

Implementation.

- When is job j compatible with A?

Interval Scheduling: Greedy Algorithm

Greedy algorithm. Consider jobs in increasing order of finish time. Take each job provided it's compatible with the ones already taken.

```
Sort jobs by finish times so that f1 ≤ f2 ≤ ... ≤ fn.
A ←{1}
j=1
for i = 2 to n {
   if Si>=Fj
      A ← A ∪ {i}
      j ← i
}
return A
```

Implementation. O(n log n).

Eg

i	1	2	3	4	5	6	7	8	9	10	11
S_i	1	3	0	5	3	5	6	8	8	2	12
F_i	4	5	6	7	8	9	10	11	12	13	14

Eg

i	1	2	3	4	5	6	7	8	9	10	11
S_i	1	3	0	5	3	5	6	8	8	2	12
F_i	4	5	6	7	8	9	10	11	12	13	14

A = {1,4,8,11}

Greedy algorithms determine a **globally optimum solution** by a series of **locally optimal choices**.
Greedy solution is not the only optimal one:

A' = {2,4,9,11}

Greedy works for Activity Selection = Interval Scheduling

Proof by induction

BASE: Optimal solution contains activity 1 as first activity
Let A be an optimal solution with activity k != 1 as first activity
Then we can replace activity k (which has $F_k >= F_1$) by activity 1
So, picking the first element in a greedy fashion works

STEP: After the first choice is made, remove all activities that are incompatible with the first chosen activity and recursively define a new problem consisting of the remaining activities. The first activity for this reduced problem can be made in a greedy fashion by the base principle.

By induction, Greedy is optimal.

What did we do?

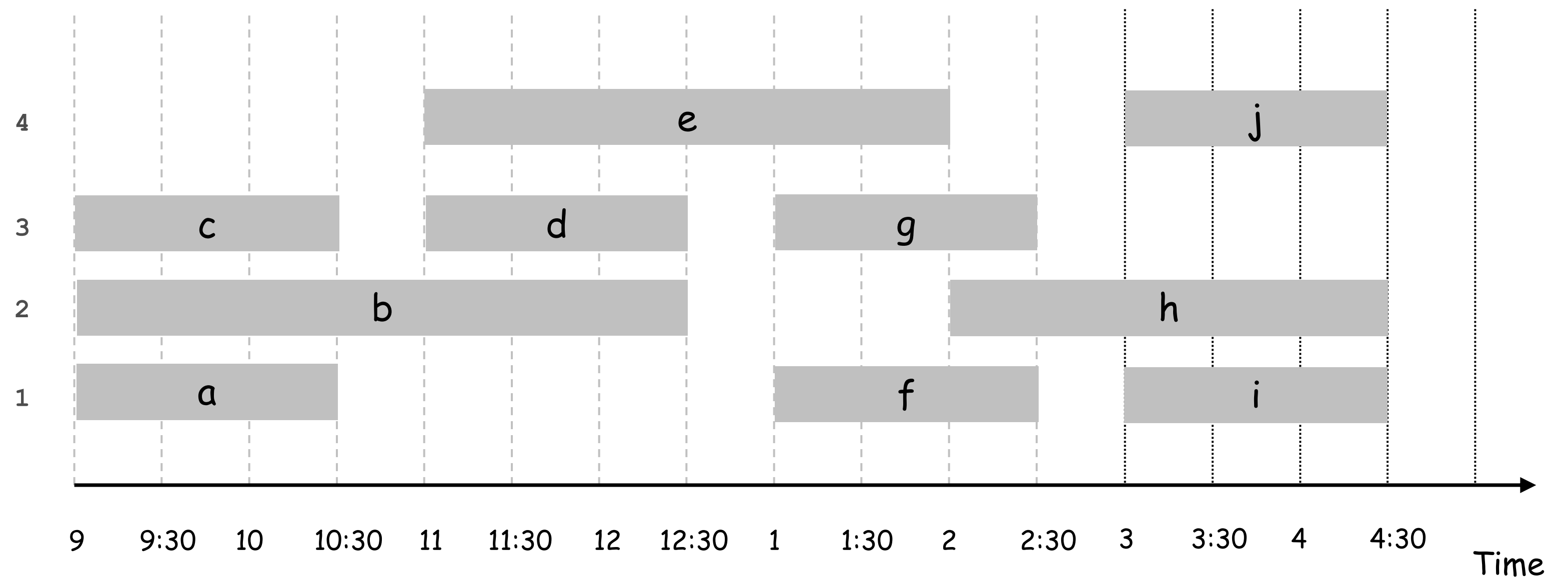
We assumed there was a non greedy optimal solution, then we stepwise **morphed** this solution into a greedy optimal solution, thereby showing that the greedy solution works in the first place.

This is called the exchange argument.

Extension 1: Scheduling all intervals

- Lecture j starts at s_j and finishes at f_j.
- Goal: find minimum number of classrooms to schedule all lectures so that no two occur at the same time in the same room.

This schedule uses 4 classrooms to schedule 10 lectures:

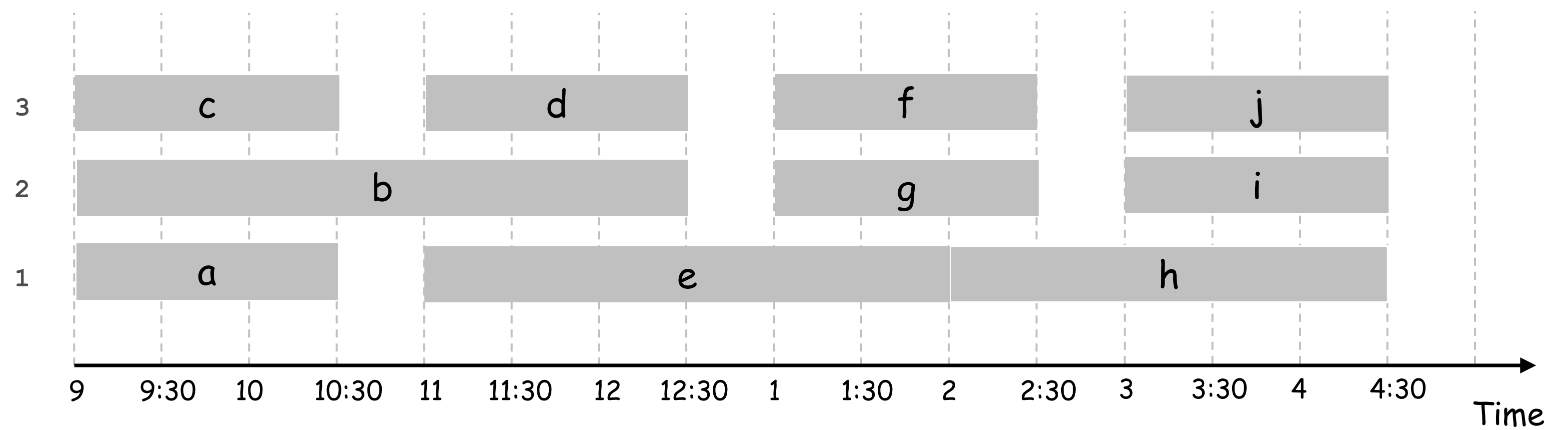


Can we do better?

Scheduling all intervals

- Eg, lecture j starts at s_j and finishes at f_j.
- Goal: find minimum number of classrooms to schedule all lectures so that no two occur at the same time in the same room.

This schedule uses 3:



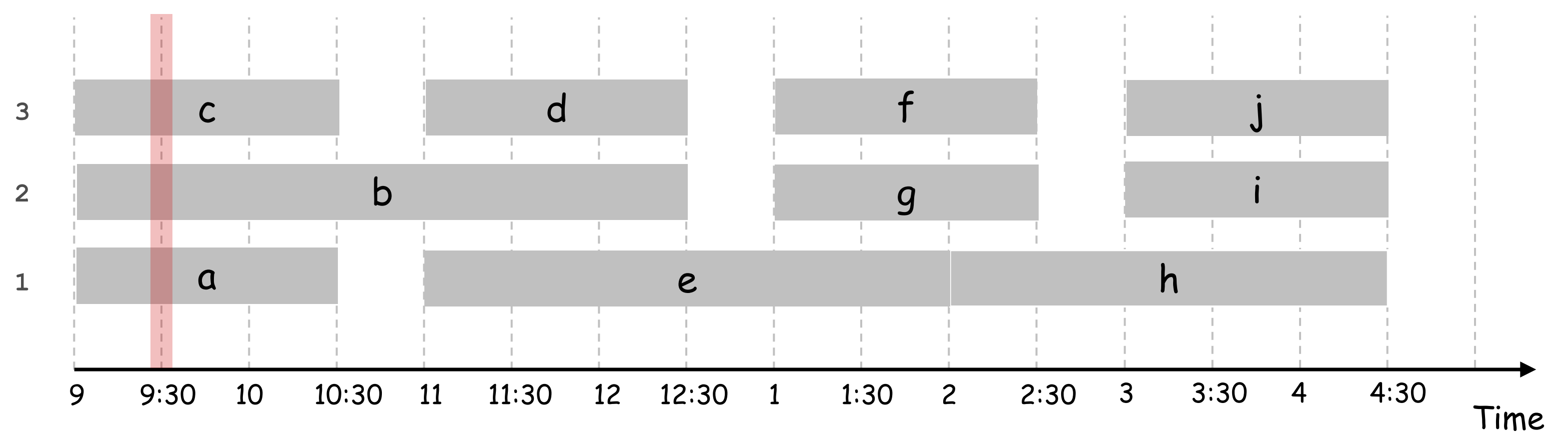
Interval Scheduling: Lower Bound

Key observation. Number of classrooms needed ≥ depth (maximum number of intervals at a time point)

Example: Depth of schedule below = 3 ⇒ schedule is optimal. We cannot do it with 2.

Q. Does there always exist a schedule equal to depth of intervals?

(hint: greedily label the intervals with their resource)



Interval Scheduling: Greedy Algorithm

Greedy algorithm.

```
allocate d labels(d = depth)
sort the intervals by starting time: I1,I2,..,In
    for j = 1 to n
        for each interval Ii that precedes and
            overlaps with Ij exclude its label for Ij
        pick a remaining label for Ij
```

Greedy works

```
allocate d labels (d = depth)
sort the intervals by starting time: I1,I2,..,In
    for j = 1 to n
       for each interval Ii that precedes and
          overlaps with Ij exclude its label for Ij
       pick a remaining label for Ij
```

Observations:

- There is always a label for I_j
  assume t intervals overlap with I_j ; these pass over a common point, so t+1 < d, so there is one of the d labels available for I_j
- No overlapping intervals get the same label
  by the nature of the algorithm

Huffman Code Compression

Huffman codes

Say I have a code consisting of the letters
a, b, c, d, e, f with frequencies (x1000)
45, 13, 12, 16, 9, 5
What would a fixed length binary encoding look like?

a	b	c	d	e	f
000	001	010	011	100	101

What would the total encoding length be?

100,000 * 3 = 300,000

Fixed vs. Variable encoding

	a	b	c	d	e	f
frequency(x1000)	45	13	12	16	9	5
fixed encoding	000	001	010	011	100	101
variable encoding	0	101	100	111	1101	1100

100,000 characters

Fixed: 300,000 bits

Variable?

(1*45 + 3*13 + 3*12 + 3*16 + 4*9 + 4*5)*1000 = 224,000 bits

25% saving

Variable **prefix** encoding

	a	b	c	d	e	f
frequency(x1000)	45	13	12	16	9	5
fixed encoding	000	001	010	011	100	101
variable encoding	0	101	100	111	1101	1100

what is special about our encoding?

no code is a prefix of another.
why does it matter?

We can concatenate the codes without ambiguities

001011101 = aabe

Two characters, frequencies, encodings

- Say we have two characters a and b,
  a with frequency f_a and b with frequency f_b
  e.g. a has frequency 70, b has frequency 30
- Say we have two encodings for these,
  one with length l_1 one with length l_2
  e.g. '101', l_1=3, '11100', l_2=5

Which encoding would we chose for a and which for b ?

if we assign a ='101' and b=11100'
what will the total number of bits be?

if we assign a ='11100' and b=101'
what will the total number of bits be?

Can you relate the difference to frequency and encoding length?

Frequency and encoding length

Two characters, a and b, with frequencies f1 and f2,
two encodings 1 and 2 with length l1 and l2

f1 > f2 and l1 > l2

I: a encoding 1, b encoding 2: f1*l1 + f2*l2
II: a encoding 2, b encoding 1: f1*l2 + f2*l1

Difference: (f1*l1 + f2*l2) - (f1*l2 + f2*l1) =
f1*(l1-l2) + f2*(l2-l1) = f1*(l1-l2) - f2*(l1-l2) =
(f1-f2)*(l1-l2)

So, for optimal encoding:
the higher the frequency, the shorter the encoding length

Cost of encoding a file: ABL

For each character c in C, f(c) is its frequency and d(c) is the number of bits it takes to encode c.

So the number of bits to encode the file is

$$\sum_{c \ in \ C} f(c)d(c)$$

The **A**verage **B**it **L**ength of an encoding **E**:

$$\mathbf{ABL(E)} = \frac{1}{n} \sum_{c \ in \ C} f(c)d(c)$$

where n is the number of characters in the file

Huffman code

An **optimal encoding** of a file has a minimal cost

- ie minimal ABL.

Huffman invented a greedy algorithm to construct an optimal prefix code called the **Huffman code**.

An encoding is represented by a **binary prefix** tree:

intermediate nodes contain **frequencies**

the sum frequencies of their children

leaves are the **characters + their frequencies**

paths to the leaves are the **codes**

the length of the encoding of a character c is the length of the path to $c{:}f_c$

Prefix tree for the variable encoding

```
a : 0              100
b : 101          0/    \1
c : 100          /       \
d : 111       a:45        55
e : 1101                 /    \
f : 1100             0/        \1
                     25        30
                  0/  \1    0/ \1
               c:12 b:13   14  d:16
                           /  \
                       0/      \1
                       f:5    e:9
```

Optimal prefix trees are full

- The frequencies of the internal nodes are the sums of the frequencies of their children.

- A binary tree is **full** if all its internal nodes have two children.

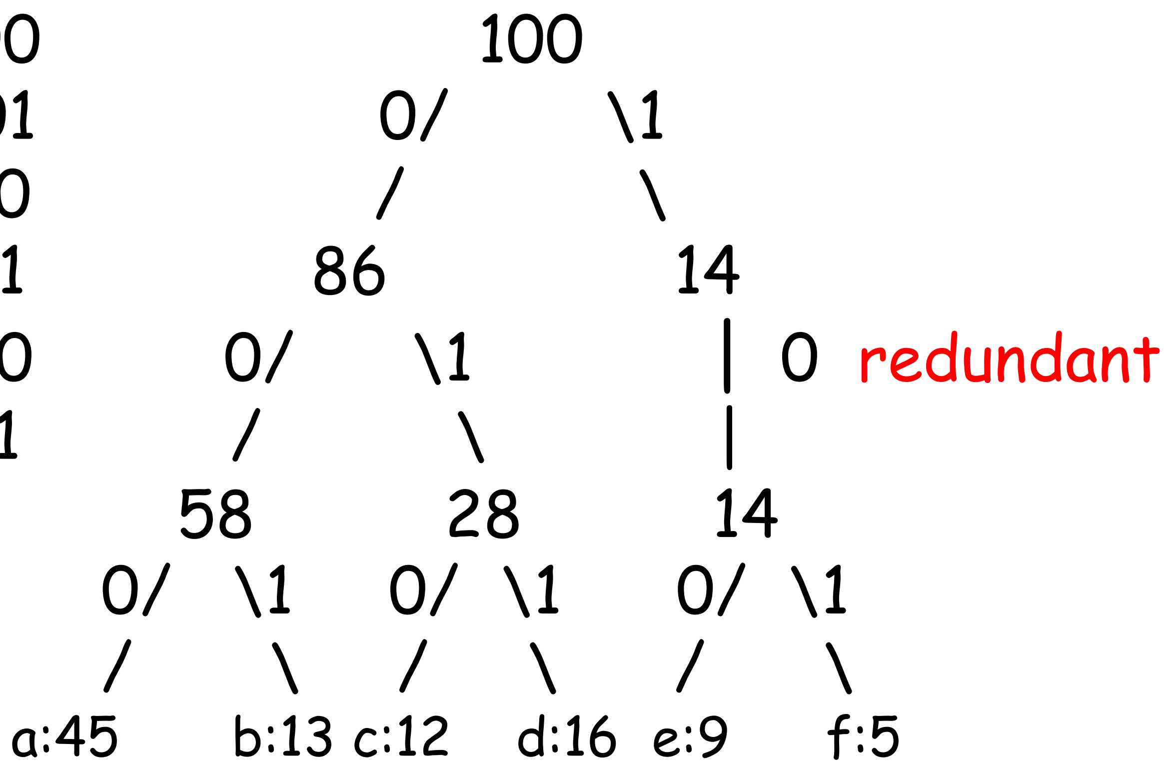
- If the prefix tree is not full, it is not optimal. **Why?**

If a tree is not full it has an internal node with one child labeled with a redundant bit.

Check the fixed encoding:

a:000 b:001 c:010 d:011 e:100 f:101

a: 000
b: 001
c: 010
d: 011
e: 100
f: 101



Huffman algorithm

- Create |C| leaves, one for each character

- Perform |C|-1 **merge** operations, each creating a new node, with **children** the nodes with **least two frequencies** and with frequency the sum of these two frequencies.

- By using a **heap** for the collection of intermediate trees this algorithm takes O(nlgn) time.

```
buildheap
do |C|-1 times
   t1 = extract-min
   t2 = extract-min
   t3 = merge(t1,t2)
   insert(t3)
```

```
1)  f:5  e:9  c:12  b:13  d:16 a:45
```

```
2)  c:12 b:13   14    d:16 a:45
               /  \
              f    e
```

```
3)  14      d:16      25   a:45
   /  \              /  \
  f    e            c    b
```

```
4)  25        30      a:45
   /  \      /  \
  c    b   14    d
          /  \
         f    e
```

```
5)  a:45    55
           /   \
         25     30
        /  \   /  \
       c    b 14   d
             /  \
            f    e
```

```
6)       100
        /  \
       a   55
          /   \
        25     30
       /  \   /  \
      c    b 14   d
            /  \
           f    e
```

Huffman is optimal

Base step of inductive approach:

Let x and y be the two characters with the minimal frequencies, then there is a minimal cost encoding tree with x and y of equal and highest depth (see e and f in our example above).
How?

The proof technique is the same exchange argument have we have used before:

If the greedy choice is not taken then we show that by taking the greedy choice we get a solution that is as good or better.

Base of the inductive proof

Let leaves x,y have the lowest frequencies.
Assume that two other characters **a and b with higher frequencies** are siblings at the lowest level of the tree T

```
     T
    / \
   O   x
  / \
 y   O
    / \
   a   b
```

Since the frequencies of x and y are lowest, the cost can only improve if we swap y and a, and x and b:
why?

```
     T
    / \
   O   b
  / \
 a   O
    / \
   y   x
```

Proof of base

```
          T                    T
         / \                  / \
        O   x                O   b
       / \                  / \
      y   O                a   O
         / \                  / \
        a   b                y   x
```

Since the frequencies of x and y are lowest, the cost can only improve if we swap y and a, and x and b. We need to prove:

cost left tree > cost right tree

(a,y part of) cost of left tree: $d_1f_y+d_2f_a$, of right tree: $d_1f_a+d_2f_y$

$d_1f_y+d_2f_a - d_1f_a-d_2f_y = d_1(f_y-f_a) +d_2(f_a-f_y) = (d_2-d_1)(f_a-f_y) > 0$

same for x and b

Greedy works: base and step

- **Base:** we have shown that putting the lowest two frequency characters lowest in the tree is a good **greedy** starting point for our algorithm.

- **Step:** We create an alphabet ***C' = C*** with x and y replaced by a new character z with frequency f(z)=f(x)+f(y) with the induction hypothesis that encoding tree **T' for *C'* is optimal**, then we must show that the larger encoding tree T for *C* is optimal.

(eg, T' is the tree created from steps 2 to 6 in the example)

Proof of step: by contradiction

1. $cost(T) = cost(T') + f(x) + f(y)$:

   $d(x)=d(y)=d(z)+1$ so
   $f(x)d(x)+f(y)d(y) = (f(x)+f(y))(d(z)+1) = f(z)d(z)+f(x)+f(y)$
   (because $f(z)=f(x)+f(y)$)

2. Now suppose T is not an optimal encoding tree, then there is another optimal tree T''. We have shown (base) that x and y are siblings at the lowest level of T''.
   Let T''' be T'' with x and y replaced by z, then
   $cost(T''') = cost(T'')-f(x)-f(y) < cost(T)-f(x)-f(y) = cost(T')$.

   But that yields a **contradiction** with the induction hypothesis that T' is optimal for C'.

Hence greedy Huffman produces an optimal prefix encoding tree.

Conclusion: Greedy Algorithms

At every step, Greedy makes the locally optimal choice, "without worrying about the future".

Greedy stays ahead. Show that after each step of the greedy algorithm, its solution is at least as good as any other.

Show Greedy works by exchange / morphing argument. Incrementally transform any optimal solution to the greedy one without worsening its quality.

Not all problems have a greedy solution. None of the NP problems (eg TSP) allow a greedy solution.