Algorithms Week 7

Ljubomir Perković, DePaul University

Data compression

Given a large file (of characters), how do we represent it as a sequence of binary characters efficiently?

Data compression

Given a large file (of characters), how do we represent it as a sequence of binary characters efficiently?

For example: consider a file that contains 100,000 characters taken from the set $\{a, b, c, d, e, f\}$.

Data compression

Given a large file (of characters), how do we represent it as a sequence of binary characters efficiently?

For example: consider a file that contains 100,000 characters taken from the set $\{a, b, c, d, e, f\}$.

One approach is to store the file using the ASCII encoding.

Data compression

Given a large file (of characters), how do we represent it as a sequence of binary characters efficiently?

For example: consider a file that contains 100,000 characters taken from the set $\{a, b, c, d, e, f\}$.

One approach is to store the file using the ASCII encoding. That requires 8 bits per character.

Data compression

Given a large file (of characters), how do we represent it as a sequence of binary characters efficiently?

For example: consider a file that contains 100,000 characters taken from the set $\{a, b, c, d, e, f\}$.

One approach is to store the file using the ASCII encoding. That requires 8 bits per character. Thus the entire file would require 800,000 bits.

Can we do better?

For our second approach, we notice that 8 bits per character are not really needed.

Can we do better?

For our second approach, we notice that 8 bits per character are not really needed.

We only need 3 bits to encode those 6 characters.

Can we do better?

For our second approach, we notice that 8 bits per character are not really needed.

We only need 3 bits to encode those 6 characters. This suggests that we use a smaller fixed length code:

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

Can we do better?

For our second approach, we notice that 8 bits per character are not really needed.

We only need 3 bits to encode those 6 characters. This suggests that we use a smaller fixed length code:

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

The file would then require a total of 300,000 bits and a compression ration of 63% is achieved.

Decoding fixed length codes

Before we consider a third approach, let's consider the problem of decoding the file.

Decoding fixed length codes

Before we consider a third approach, let's consider the problem of decoding the file.

With fixed length codes, it's easy to decode the file:

Decoding fixed length codes

Before we consider a third approach, let's consider the problem of decoding the file.

With fixed length codes, it's easy to decode the file:

The file that starts with 000001010000101... would be decoded as:

000	001	010	000	101	...
↓	↓	↓	↓	↓	
a	*b*	*c*	*a*	*f*	...

Decoding fixed length codes

Before we consider a third approach, let's consider the problem of decoding the file.

With fixed length codes, it's easy to decode the file:

The file that starts with 000001010000101... would be decoded as:

000	001	010	000	101	...
↓	↓	↓	↓	↓	
a	*b*	*c*	*a*	*f*	...

We can't use a smaller fixed-length code for compressing our file...

Decoding fixed length codes

Before we consider a third approach, let's consider the problem of decoding the file.

With fixed length codes, it's easy to decode the file:

The file that starts with 000001010000101... would be decoded as:

000	001	010	000	101	...
↓	↓	↓	↓	↓	
a	*b*	*c*	*a*	*f*	...

We can't use a smaller fixed-length code for compressing our file...

... because using only two bits gives us four possible codes, but the file contains six different symbols to encode.

Compressing further

We can compress the file further if we know a bit more information about the distribution of the symbols in the file.

[1] in thousands

Compressing further

We can compress the file further if we know a bit more information about the distribution of the symbols in the file. Suppose that the symbols occur with the following frequencies in the file:

Symbol	Frequency[1]
a	45
b	13
c	12
d	16
e	9
f	5

[1] in thousands

Compressing further

We can compress the file further if we know a bit more information about the distribution of the symbols in the file. Suppose that the symbols occur with the following frequencies in the file:

Symbol	Frequency[1]
a	45
b	13
c	12
d	16
e	9
f	5

If we represent the more frequent symbols using fewer bits, then the number of bits needed to encode the entire file might decrease.

[1] in thousands

Suppose we use the following encoding:

Symbol	Code	Frequency[2] x Cost
a	0	45 x 1
b	101	13 x 3
c	100	12 x 3
d	111	16 x 3
e	1101	9 x 4
f	1100	5 x 4

[2] in thousands

Suppose we use the following encoding:

Symbol	Code	Frequency[2] x Cost
a	0	45 x 1
b	101	13 x 3
c	100	12 x 3
d	111	16 x 3
e	1101	9 x 4
f	1100	5 x 4

Then encoding the file would require 224,000 bits.

[2] in thousands

Suppose we use the following encoding:

Symbol	Code	Frequency[2] x Cost
a	0	45 x 1
b	101	13 x 3
c	100	12 x 3
d	111	16 x 3
e	1101	9 x 4
f	1100	5 x 4

Then encoding the file would require 224,000 bits. This is 72% fewer bits than the ASCII encoding and 25% fewer bits than the 3-bit fixed length code.

[2] in thousands

Variable-length codes

Not all variable-length codes are useful.

Variable-length codes

Not all variable-length codes are useful.

To see this, consider the problem of decoding.

Variable-length codes

Not all variable-length codes are useful.

To see this, consider the problem of decoding.

Suppose that we are encoding a file containing the characters $\{a, b, c\}$ and that c is the least frequent symbol in the file.

Variable-length codes

Not all variable-length codes are useful.

To see this, consider the problem of decoding.

Suppose that we are encoding a file containing the characters $\{a, b, c\}$ and that c is the least frequent symbol in the file.

Then we might try using the code:

Symbol	Code
a	0
b	1
c	01

Variable-length codes

Not all variable-length codes are useful.

To see this, consider the problem of decoding.

Suppose that we are encoding a file containing the characters $\{a, b, c\}$ and that c is the least frequent symbol in the file.

Then we might try using the code:

Symbol	Code
a	0
b	1
c	01

There is a problem!

Variable-length codes

Not all variable-length codes are useful.

To see this, consider the problem of decoding.

Suppose that we are encoding a file containing the characters $\{a, b, c\}$ and that c is the least frequent symbol in the file.

Then we might try using the code:

Symbol	Code
a	0
b	1
c	01

There is a problem! Decoding "01" is ambiguous because the code for the symbol "a" is a prefix of the code for the symbol "c".

Prefix-free codes

A prefix-free code is a code in which no code is a prefix of another.

Prefix-free codes

A prefix-free code is a code in which no code is a prefix of another.

The first variable-length code we saw happens to be a prefix-free code:

Symbol	Code
a	0
b	101
c	100
d	111
e	1101
f	1100

Prefix-free codes

A prefix-free code is a code in which no code is a prefix of another.

The first variable-length code we saw happens to be a prefix-free code:

Symbol	Code
a	0
b	101
c	100
d	111
e	1101
f	1100

Note that 0 is not a prefix of any other code, 100 is not a prefix of any other code etc.

Decoding revisited

Symbol	Code
a	0
b	101
c	100
d	111
e	1101
f	1100

Prefix-free codes allow us to easily decode. For example, decoding "010110001100..." gives:

$$\begin{array}{ccccccccccc} 0 & | & 101 & | & 100 & | & 0 & | & 1100 & | \\ \downarrow & & \downarrow & & \downarrow & & \downarrow & & \downarrow & & \end{array}$$

Decoding revisited

Symbol	Code
a	0
b	101
c	100
d	111
e	1101
f	1100

Prefix-free codes allow us to easily decode. For example, decoding "010110001100..." gives:

$$\begin{array}{ccccccccccc} 0 & | & 101 & | & 100 & | & 0 & | & 1100 & | \\ \downarrow & & \downarrow & & \downarrow & & \downarrow & & \downarrow & & \\ a & & & & & & & & & & \end{array}$$

Decoding revisited

Symbol	Code
a	0
b	101
c	100
d	111
e	1101
f	1100

Prefix-free codes allow us to easily decode. For example, decoding "010110001100..." gives:

$$\begin{array}{ccccccccccc} 0 & | & 101 & | & 100 & | & 0 & | & 1100 & | \\ \downarrow & & \downarrow & & \downarrow & & \downarrow & & \downarrow & & \\ a & & b & & & & & & & & \end{array}$$

Decoding revisited

Symbol	Code
a	0
b	101
c	100
d	111
e	1101
f	1100

Prefix-free codes allow us to easily decode. For example, decoding "010110001100..." gives:

$$\begin{array}{ccccccccccc} 0 & | & 101 & | & 100 & | & 0 & | & 1100 & | \\ \downarrow & & \downarrow & & \downarrow & & \downarrow & & \downarrow & & \\ a & & b & & c & & & & & & \end{array}$$

Decoding revisited

Symbol	Code
a	0
b	101
c	100
d	111
e	1101
f	1100

Prefix-free codes allow us to easily decode. For example, decoding "010110001100..." gives:

$$\begin{array}{ccccccccccc} 0 & | & 101 & | & 100 & | & 0 & | & 1100 & | \\ \downarrow & & \downarrow & & \downarrow & & \downarrow & & \downarrow & & \\ a & & b & & c & & a & & & & \end{array}$$

Decoding revisited

Symbol	Code
a	0
b	101
c	100
d	111
e	1101
f	1100

Prefix-free codes allow us to easily decode. For example, decoding "010110001100..." gives:

$$\begin{array}{ccccccccccc} 0 & | & 101 & | & 100 & | & 0 & | & 1100 & | & \\ \downarrow & & \downarrow & & \downarrow & & \downarrow & & \downarrow & & \\ a & & b & & c & & a & & f & & \ldots \end{array}$$

Decoding revisited

Symbol	Code
a	0
b	101
c	100
d	111
e	1101
f	1100

Prefix-free codes allow us to easily decode. For example, decoding "010110001100..." gives:

$$\begin{array}{ccccccccccc} 0 & | & 101 & | & 100 & | & 0 & | & 1100 & | & \\ \downarrow & & \downarrow & & \downarrow & & \downarrow & & \downarrow & & \\ a & & b & & c & & a & & f & & \ldots \end{array}$$

With a prefix-free code, as soon as we see a set of symbols that corresponds to a code, we can decode the code.

Representing a prefix-free code

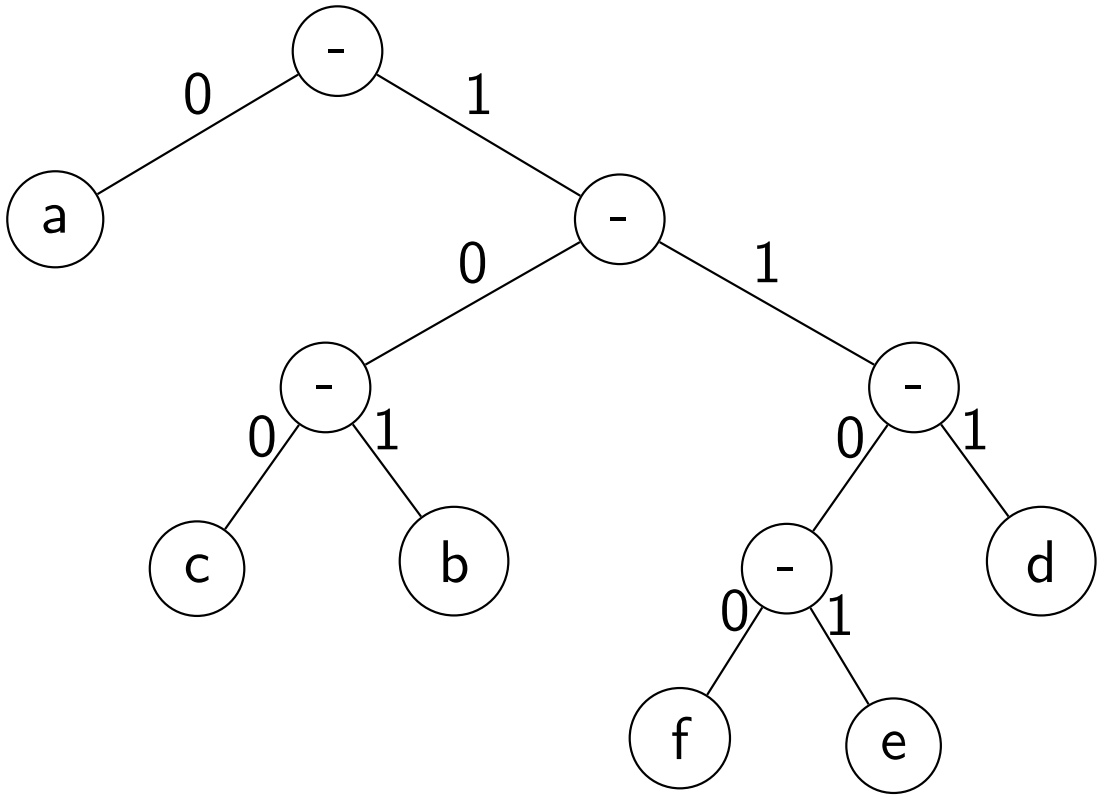
A prefix-free code can be represented using a full binary tree (a binary tree in which every node is either a leaf or has two children):

Symbol	Code
a	0
b	101
c	100
d	111
e	1101
f	1100

Representing a prefix-free code

A prefix-free code can be represented using a full binary tree (a binary tree in which every node is either a leaf or has two children):

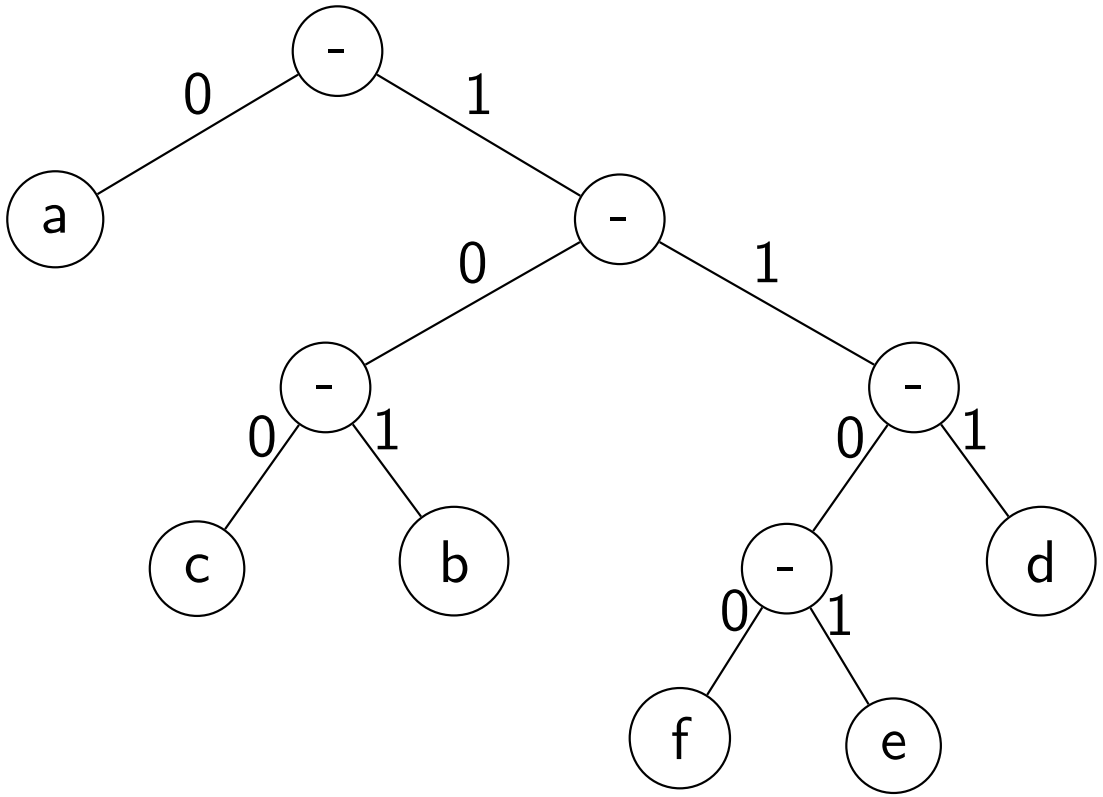
Symbol	Code
a	0
b	101
c	100
d	111
e	1101
f	1100



Representing a prefix-free code

A prefix-free code can be represented using a full binary tree (a binary tree in which every node is either a leaf or has two children):

- Each leaf of the tree corresponds to a symbol to be encoded,
- The left and right links from each nonleaf node correspond to 0 and 1, respectively.



Finding a good prefix-free code

We are now finally ready to state the problem:

Finding a good prefix-free code

We are now finally ready to state the problem:

Input: Characters $c[1], c[2], ..., c[n]$ and corresponding frequencies $f[1], f[2], ..., f[n]$.

Output: A tree representing the optimal prefix-free code.

Finding a good prefix-free code

We are now finally ready to state the problem:

Input: Characters $c[1], c[2], ..., c[n]$ and corresponding frequencies $f[1], f[2], ..., f[n]$.

Output: A tree representing the optimal prefix-free code.

If $d_T(i)$ is the depth of the node corresponding to character $c[i]$ in the tree T, our goal is to find a prefix-free code tree that optimizes the function:

$$B(T) = \sum_{i=1}^{n} f[i] \cdot d_T(i)$$

Huffman's greedy algorithm

The idea is to represent the more frequent characters using shorter codes.

Huffman's greedy algorithm

The idea is to represent the more frequent characters using shorter codes.

We will use the greedy approach to construct the prefix-free code tree:

Huffman's greedy algorithm

The idea is to represent the more frequent characters using shorter codes.

We will use the greedy approach to construct the prefix-free code tree:

1. Create a tree node for each character and put them in a container.
2. Repeat until there is only one tree in the container:
   a. Greedily merge the two least-frequent trees into a sub-tree whose frequency is equal to the sum of the two tree frequencies.
   b. Put the sub-tree back into the container.

Huffman's greedy algorithm

The idea is to represent the more frequent characters using shorter codes.

We will use the greedy approach to construct the prefix-free code tree:

1. Create a tree node for each character and put them in a container.
2. Repeat until there is only one tree in the container:
   a. Greedily merge the two least-frequent trees into a sub-tree whose frequency is equal to the sum of the two tree frequencies.
   b. Put the sub-tree back into the container.

The remaining tree represents the optimal prefix-free code.

Huffman's greedy algorithm

The idea is to represent the more frequent characters using shorter codes.

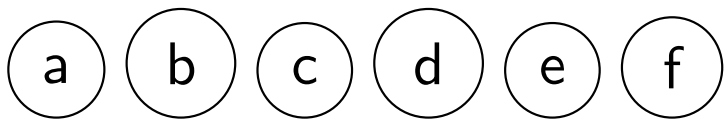
We will use the greedy approach to construct the prefix-free code tree:

1. Create a tree node for each character and put them in a container.
2. Repeat until there is only one tree in the container:
   a. Greedily merge the two least-frequent trees into a sub-tree whose frequency is equal to the sum of the two tree frequencies.
   b. Put the sub-tree back into the container.

The remaining tree represents the optimal prefix-free code. (We still have to prove that!)

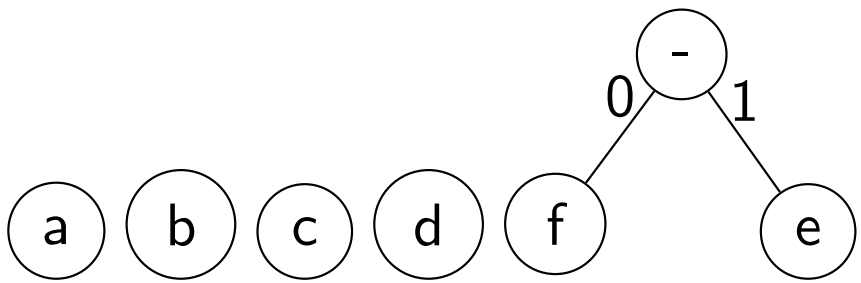
Huffman's greedy algorithm

Symbol	Frequency
a	45
b	13
c	12
d	16
e	9
f	5



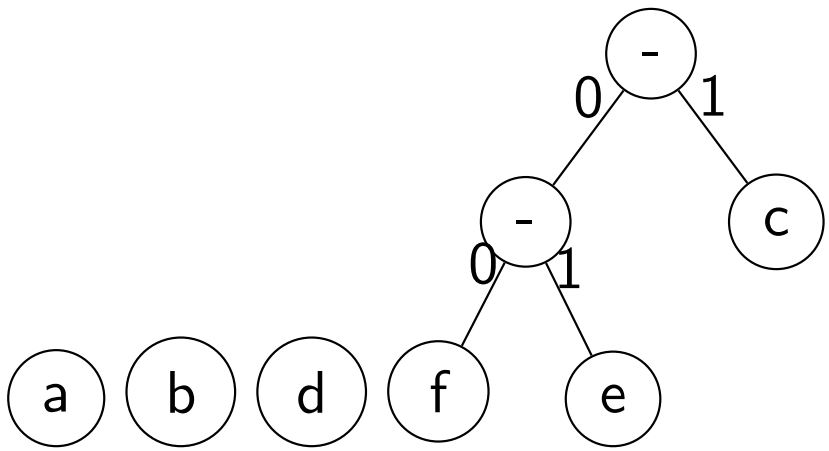
Huffman's greedy algorithm

Symbol	Frequency
a	45
b	13
c	12
d	16
e,f	14



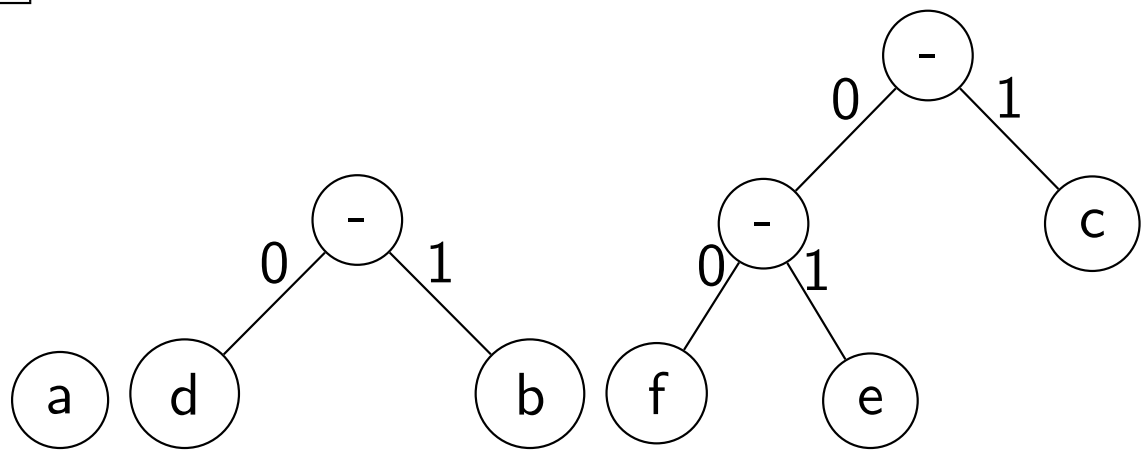
Huffman's greedy algorithm

Symbol	Frequency
a	45
b	13
c,e,f	26
d	16



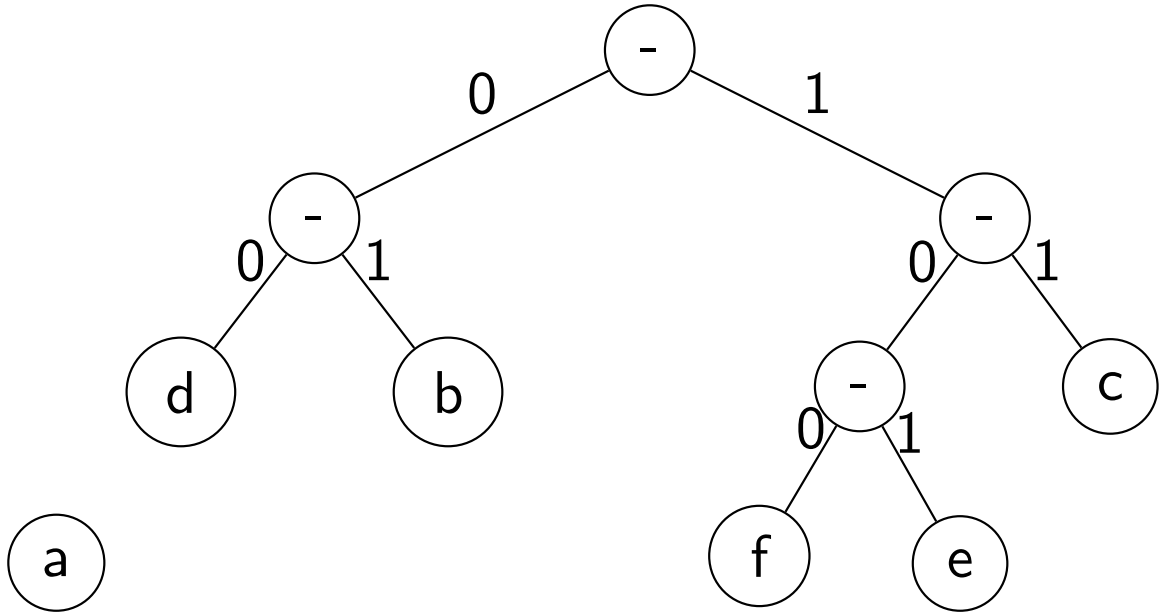
Huffman's greedy algorithm

Symbol	Frequency
a	45
c,e,f	26
b,d	29



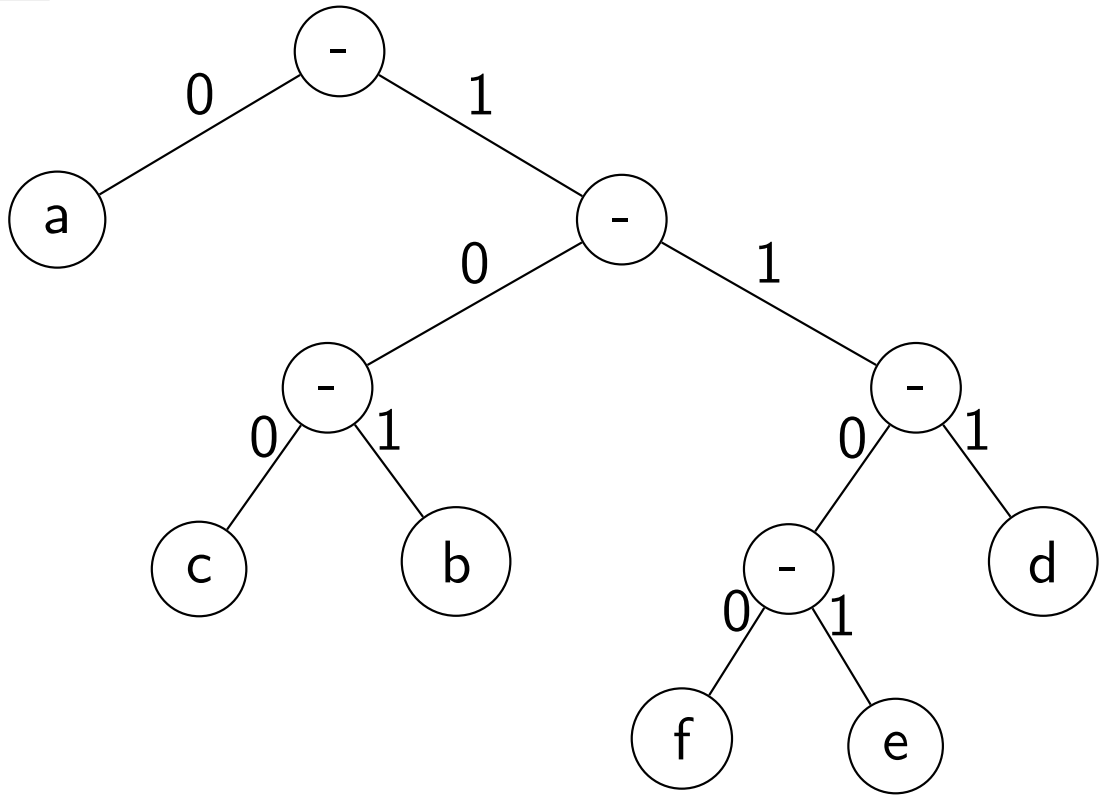
Huffman's greedy algorithm

Symbol	Frequency
a	45
b,c,d,e,f	55



Huffman's greedy algorithm

Symbol	Frequency
a,b,c,d,e,f	100



Huffman's greedy algorithm

What is the Abstract Data Type container that supports *insert* and *extractMin* methods?

Huffman's greedy algorithm

What is the Abstract Data Type container that supports *insert* and *extractMin* methods? What is the best data structure for implementing a priority queue?

Huffman's greedy algorithm

What is the Abstract Data Type container that supports *insert* and *extractMin* methods? What is the best data structure for implementing a priority queue?

```
Huffman(f[1..n], c[1..n])
  // PQ is a priority queue of (Node, freq) tuples
  for i ← 1 to n:
    t ← new Node(c[i],null,null)
    PQ.insert((t, f[i]))
  for i ← 1 to n-1
    (t1, f1) ← PQ.extractMin()
    (t2, f2) ← PQ.extractMin()
    t ← new Node('-',t1,t2))
    PQ.insert((t,f1+f2))
  return PQ.extractMin()
```

Huffman's greedy algorithm

What is the Abstract Data Type container that supports *insert* and *extractMin* methods? What is the best data structure for implementing a priority queue?

```
Huffman(f[1..n], c[1..n])
  // PQ is a priority queue of (Node, freq) tuples
  for i ← 1 to n:
    t ← new Node(c[i],null,null)
    PQ.insert((t, f[i]))
  for i ← 1 to n-1
    (t1, f1) ← PQ.extractMin()
    (t2, f2) ← PQ.extractMin()
    t ← new Node('-',t1,t2))
    PQ.insert((t,f1+f2))
  return PQ.extractMin()
```

Running time?

Running time analysis

Assuming that the priority queue PQ is implemented via a binary heap:

Running time analysis

Assuming that the priority queue PQ is implemented via a binary heap:

- Create a tree node (n times): $\Theta(1)$ each time

Running time analysis

Assuming that the priority queue PQ is implemented via a binary heap:

- Create a tree node (n times): $\Theta(1)$ each time
- Insert into a heap (n times): $\Theta(\log n)$ each time

Running time analysis

Assuming that the priority queue PQ is implemented via a binary heap:

- Create a tree node (n times): $\Theta(1)$ each time
- Insert into a heap (n times): $\Theta(\log n)$ each time
- Extract the minimum ($2(n-1)$ times): $\Theta(\log n)$ each time

Running time analysis

Assuming that the priority queue PQ is implemented via a binary heap:

- Create a tree node (n times): $\Theta(1)$ each time
- Insert into a heap (n times): $\Theta(\log n)$ each time
- Extract the minimum ($2(n-1)$ times): $\Theta(\log n)$ each time
- Merge two subtrees ($n-1$ times): $\Theta(1)$ each time

Running time analysis

Assuming that the priority queue PQ is implemented via a binary heap:

- Create a tree node (n times): $\Theta(1)$ each time
- Insert into a heap (n times): $\Theta(\log n)$ each time
- Extract the minimum ($2(n-1)$ times): $\Theta(\log n)$ each time
- Merge two subtrees ($n-1$ times): $\Theta(1)$ each time
- Insert into a heap ($n-1$ times): $\Theta(\log n)$ each time

Running time analysis

Assuming that the priority queue PQ is implemented via a binary heap:

- Create a tree node (n times): $\Theta(1)$ each time
- Insert into a heap (n times): $\Theta(\log n)$ each time
- Extract the minimum ($2(n-1)$ times): $\Theta(\log n)$ each time
- Merge two subtrees ($n-1$ times): $\Theta(1)$ each time
- Insert into a heap ($n-1$ times): $\Theta(\log n)$ each time
- Extract the remaining tree: $\Theta(1)$.

Running time analysis

Assuming that the priority queue PQ is implemented via a binary heap:

- Create a tree node (n times): $\Theta(1)$ each time
- Insert into a heap (n times): $\Theta(\log n)$ each time
- Extract the minimum ($2(n-1)$ times): $\Theta(\log n)$ each time
- Merge two subtrees ($n-1$ times): $\Theta(1)$ each time
- Insert into a heap ($n-1$ times): $\Theta(\log n)$ each time
- Extract the remaining tree: $\Theta(1)$.

Thus the total time required is $\Theta(n \log n)$.

Running time analysis

Assuming that the priority queue PQ is implemented via a binary heap:

- Create a tree node (n times): $\Theta(1)$ each time
- Insert into a heap (n times): $\Theta(\log n)$ each time
- Extract the minimum ($2(n-1)$ times): $\Theta(\log n)$ each time
- Merge two subtrees ($n-1$ times): $\Theta(1)$ each time
- Insert into a heap ($n-1$ times): $\Theta(\log n)$ each time
- Extract the remaining tree: $\Theta(1)$.

Thus the total time required is $\Theta(n \log n)$.

We still have to prove correctness!

Verifying the greedy properties

We must now show that Huffman's algorithm constructs a binary tree that represents an optimal prefix-free code.

Verifying the greedy properties

We must now show that Huffman's algorithm constructs a binary tree that represents an optimal prefix-free code.

To do so, we must show that the optimal prefix-free code problem exhibits the greedy choice and the optimal substructure properties.

Verifying the greedy properties

We must now show that Huffman's algorithm constructs a binary tree that represents an optimal prefix-free code.

To do so, we must show that the optimal prefix-free code problem exhibits the greedy choice and the optimal substructure properties.

Before we do that we first show that:

Theorem

The optimal prefix-free code is always represented by a full binary tree.

Theorem

The optimal prefix-free code is always represented by a full binary tree.

Theorem

The optimal prefix-free code is always represented by a full binary tree.

- Suppose not.

Theorem

The optimal prefix-free code is always represented by a full binary tree.

- Suppose not.
- Then there must exist a nonleaf node x with only one child y.

Theorem

The optimal prefix-free code is always represented by a full binary tree.

- Suppose not.
- Then there must exist a nonleaf node x with only one child y.
- WLOG, assume that y is the left child of x, corresponding to binary 0.

Theorem

The optimal prefix-free code is always represented by a full binary tree.

- Suppose not.
- Then there must exist a nonleaf node x with only one child y.
- WLOG, assume that y is the left child of x, corresponding to binary 0.
- We can obtain another tree by deleting the link (x, y) and replacing node x by the subtree rooted at y.

Theorem

The optimal prefix-free code is always represented by a full binary tree.

- Suppose not.
- Then there must exist a nonleaf node x with only one child y.
- WLOG, assume that y is the left child of x, corresponding to binary 0.
- We can obtain another tree by deleting the link (x, y) and replacing node x by the subtree rooted at y.
- That turns out to be a better tree than the original, which is a contradiction.

The greedy choice property

Lemma

If x and y are the two characters with the lowest frequency, then there exists an optimal prefix-free code where x and y differ only in their last bit.

The greedy choice property

Lemma

If x and y are the two characters with the lowest frequency, then there exists an optimal prefix-free code where x and y differ only in their last bit.

This lemma tells us that it is a "safe" first choice to merge the symbols with the lowest frequencies into a sub-tree.

The greedy choice property

Lemma

If x and y are the two characters with the lowest frequency, then there exists an optimal prefix-free code where x and y differ only in their last bit.

This lemma tells us that it is a "safe" first choice to merge the symbols with the lowest frequencies into a sub-tree.

So our "greedy choice" of choosing the two smallest frequencies to merge was valid.

The greedy choice property

Lemma

If x and y are the two characters with the lowest frequency, then there exists an optimal prefix-free code where x and y differ only in their last bit.

The proof idea is to take the tree T representing an arbitrary optimal prefix-free code and modify it to make a tree representing another optimal prefix-free code such that the characters x and y appear as sibling leaves of maximum depth in the new tree.

The greedy choice property

Lemma

If x and y are the two characters with the lowest frequency, then there exists an optimal prefix-free code where x and y differ only in their last bit.

The proof idea is to take the tree T representing an arbitrary optimal prefix-free code and modify it to make a tree representing another optimal prefix-free code such that the characters x and y appear as sibling leaves of maximum depth in the new tree.

If we can do this, then their codes will have the same length and differ only in the last bit.

The greedy choice property

- Let b and c be two characters that are sibling leaves at maximum depth in T.

The greedy choice property

- Let b and c be two characters that are sibling leaves at maximum depth in T. Then $d_T(b), d_T(c) \geq d_T(x), d_T(y)$.

The greedy choice property

- Let b and c be two characters that are sibling leaves at maximum depth in T. Then $d_T(b), d_T(c) \geq d_T(x), d_T(y)$.
- Let $f(x)$ be the frequency of character x.
- Without loss of generality, assume that $f(b) \leq f(c)$ and $f(x) \leq f(y)$.

The greedy choice property

- Let b and c be two characters that are sibling leaves at maximum depth in T. Then $d_T(b), d_T(c) \geq d_T(x), d_T(y)$.
- Let $f(x)$ be the frequency of character x.
- Without loss of generality, assume that $f(b) \leq f(c)$ and $f(x) \leq f(y)$. Since $f(x)$ and $f(y)$ are the two lowest frequencies, and $f(b)$ and $f(c)$ are arbitrary frequencies, we know that $f(x) \leq f(b)$ and $f(y) \leq f(c)$.

The greedy choice property

- Let b and c be two characters that are sibling leaves at maximum depth in T. Then $d_T(b), d_T(c) \geq d_T(x), d_T(y)$.
- Let $f(x)$ be the frequency of character x.
- Without loss of generality, assume that $f(b) \leq f(c)$ and $f(x) \leq f(y)$. Since $f(x)$ and $f(y)$ are the two lowest frequencies, and $f(b)$ and $f(c)$ are arbitrary frequencies, we know that $f(x) \leq f(b)$ and $f(y) \leq f(c)$.
- We swap the positions of x and b in tree T to get tree T' and swap the positions of y and c in T' to get the tree T''.

The greedy choice property

- Let b and c be two characters that are sibling leaves at maximum depth in T. Then $d_T(b), d_T(c) \geq d_T(x), d_T(y)$.
- Let $f(x)$ be the frequency of character x.
- Without loss of generality, assume that $f(b) \leq f(c)$ and $f(x) \leq f(y)$. Since $f(x)$ and $f(y)$ are the two lowest frequencies, and $f(b)$ and $f(c)$ are arbitrary frequencies, we know that $f(x) \leq f(b)$ and $f(y) \leq f(c)$.
- We swap the positions of x and b in tree T to get tree T' and swap the positions of y and c in T' to get the tree T''.
- The difference in cost between the trees T and T' is:

$$B(T) - B(T') = \sum_{c \in C} f(c) d_T(c) - \sum_{c \in C} f(c) d_{T'}(c)$$

$$
\begin{aligned}
B(T) - B(T') &= \sum_{c \in C} f(c) d_T(c) - \sum_{c \in C} f(c) d_{T'}(c) \\
&= f(x) d_T(x) + f(b) d_T(b) - f(x) d_{T'}(x) - f(b) d_{T'}(b)
\end{aligned}
$$

$$
\begin{aligned}
B(T) - B(T') &= \sum_{c \in C} f(c) d_T(c) - \sum_{c \in C} f(c) d_{T'}(c) \\
&= f(x) d_T(x) + f(b) d_T(b) - f(x) d_{T'}(x) - f(b) d_{T'}(b) \\
&= f(x) d_T(x) + f(b) d_T(b) - f(x) d_T(b) - f(b) d_T(x)
\end{aligned}
$$

$$
\begin{aligned}
B(T) - B(T') &= \sum_{c \in C} f(c) d_T(c) - \sum_{c \in C} f(c) d_{T'}(c) \\
&= f(x) d_T(x) + f(b) d_T(b) - f(x) d_{T'}(x) - f(b) d_{T'}(b) \\
&= f(x) d_T(x) + f(b) d_T(b) - f(x) d_T(b) - f(b) d_T(x) \\
&= (f(b) - f(x))(d_T(b) - d_T(x))
\end{aligned}
$$

The greedy choice property

$$
\begin{aligned}
B(T) - B(T') &= \sum_{c \in C} f(c) d_T(c) - \sum_{c \in C} f(c) d_{T'}(c) \\
&= f(x) d_T(x) + f(b) d_T(b) - f(x) d_{T'}(x) - f(b) d_{T'}(b) \\
&= f(x) d_T(x) + f(b) d_T(b) - f(x) d_T(b) - f(b) d_T(x) \\
&= (f(b) - f(x))(d_T(b) - d_T(x)) \\
&\geq 0
\end{aligned}
$$

A similar argument shows that $B(T') - B(T'') \geq 0$.

$$
\begin{aligned}
B(T) - B(T') &= \sum_{c \in C} f(c) d_T(c) - \sum_{c \in C} f(c) d_{T'}(c) \\
&= f(x) d_T(x) + f(b) d_T(b) - f(x) d_{T'}(x) - f(b) d_{T'}(b) \\
&= f(x) d_T(x) + f(b) d_T(b) - f(x) d_T(b) - f(b) d_T(x) \\
&= (f(b) - f(x))(d_T(b) - d_T(x)) \\
&\geq 0
\end{aligned}
$$

A similar argument shows that $B(T') - B(T'') \geq 0$.

Thus $B(T) \geq B(T'')$.

$$
\begin{aligned}
B(T) - B(T') &= \sum_{c \in C} f(c) d_T(c) - \sum_{c \in C} f(c) d_{T'}(c) \\
&= f(x) d_T(x) + f(b) d_T(b) - f(x) d_{T'}(x) - f(b) d_{T'}(b) \\
&= f(x) d_T(x) + f(b) d_T(b) - f(x) d_T(b) - f(b) d_T(x) \\
&= (f(b) - f(x))(d_T(b) - d_T(x)) \\
&\geq 0
\end{aligned}
$$

A similar argument shows that $B(T') - B(T'') \geq 0$.

Thus $B(T) \geq B(T'')$. But since T is optimal, it must be the case that $B(T) = B(T'')$ and so T'' is optimal too.

Optimal substructure

Lemma

Let x and y be sibling leaves in an optimal tree T for C. Replace x and y by node z, and assign to z the frequency $f(z) = f(x) + f(y)$. Then $T' = T - \{x, y\}$ is an optimal tree for $C' = C - \{x, y\} \cup \{z\}$.

Optimal substructure

Lemma

Let x and y be sibling leaves in an optimal tree T for C. Replace x and y by node z, and assign to z the frequency $f(z) = f(x) + f(y)$. Then $T' = T - \{x, y\}$ is an optimal tree for $C' = C - \{x, y\} \cup \{z\}$.

In other words: the optimal solution T to the prefix-free code tree problem can be obtained by combining the greedy choice with the solution T' to the problem in which x and y have been replaced by a node z whose frequency is $f(z) = f(x) + f(y)$.

Optimal substructure

- We start by showing that the cost of the tree T can be expressed in terms of the cost of the tree T'.
- For each $c \in C - \{x, y\}$, we know that $d_T(c) = d_{T'}(c)$. This is because any leaf other than x or y is located at the same level in tree T' as in tree T.
- Thus $f(c)d_T(c) = f(c)d_{T'}(c)$ for every $c \in C - \{x, y\}$.
- Since $d_T(x) = d_T(y) = d_{T'}(z) + 1$, we know that

$$\begin{aligned} f(x)d_T(x) + f(y)d_T(y) &= (f(x) + f(y))(d_{T'}(z) + 1) \\ &= f(z)d_{T'}(z) + f(x) + f(y) \end{aligned}$$

- This means that $B(T) = B(T') + f(x) + f(y)$.

Optimal substructure

- Suppose that T' is not optimal for $C - \{x, y\} \cup \{z\}$.
- That means that there is a tree T'' whose leaves are characters in $C - \{x, y\} \cup \{z\}$ such that $B(T'') < B(T')$. Since z is a character in $C - \{x, y\} \cup \{z\}$ it must appear as a leaf in the tree T''.
- So by adding x and y as children of z, we obtain a prefix-free codes for C with cost $B(T'') + f(x) + f(y) < B(T') + f(x) + f(y) = B(T)$.
- This contradicts the optimality of T.
- So it must be the case that T' is optimal for $C - \{x, y\} \in \{z\}$.

Resident Match Problem

Each year, doctors submit a ranked list of all hospitals where they would accept an internship ...

Resident Match Problem

Each year, doctors submit a ranked list of all hospitals where they would accept an internship ... and each hospital submits a ranked list of doctors they would accept as interns.

Resident Match Problem

Each year, doctors submit a ranked list of all hospitals where they would accept an internship ... and each hospital submits a ranked list of doctors they would accept as interns.

The NRMP computes a stable matching between doctors and hospitals.

Resident Match Problem

Each year, doctors submit a ranked list of all hospitals where they would accept an internship ... and each hospital submits a ranked list of doctors they would accept as interns.

The NRMP computes a stable matching between doctors and hospitals.

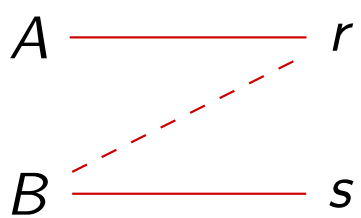
A matching is unstable if there is a doctor r and hospital B that would be happier with each other than with their current match:

Resident Match Problem

Each year, doctors submit a ranked list of all hospitals where they would accept an internship ... and each hospital submits a ranked list of doctors they would accept as interns.

The NRMP computes a stable matching between doctors and hospitals.

A matching is unstable if there is a doctor r and hospital B that would be happier with each other than with their current match:

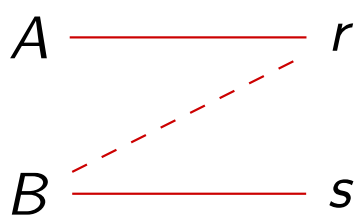


Resident Match Problem

Each year, doctors submit a ranked list of all hospitals where they would accept an internship ... and each hospital submits a ranked list of doctors they would accept as interns.

The NRMP computes a stable matching between doctors and hospitals.

A matching is unstable if there is a doctor r and hospital B that would be happier with each other than with their current match:



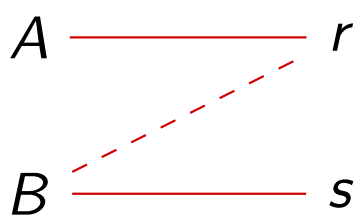
We call (r, B) an unstable pair for the matching.

Resident Match Problem

Each year, doctors submit a ranked list of all hospitals where they would accept an internship ... and each hospital submits a ranked list of doctors they would accept as interns.

The NRMP computes a stable matching between doctors and hospitals.

A matching is unstable if there is a doctor r and hospital B that would be happier with each other than with their current match:



We call (r, B) an unstable pair for the matching.

The goal of the Resident Match Problem is a matching with no unstable pairs, i.e. a stable matching.

Resident Match Problem

3 doctors q, r, s and 3 hospitals A, B, C rank each other:

	A	B	C
1.	r	s	q
2.	q	q	r
3.	s	r	s

	q	r	s
1.	A	C	A
2.	C	A	B
3.	B	B	C

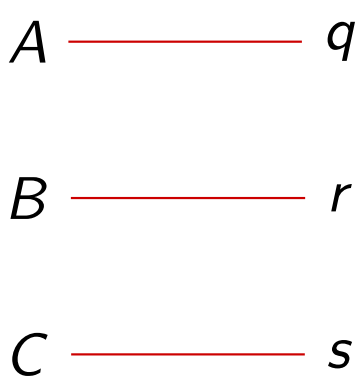
Resident Match Problem

3 doctors q, r, s and 3 hospitals A, B, C rank each other:

	A	B	C
1.	r	s	q
2.	q	q	r
3.	s	r	s

	q	r	s
1.	A	C	A
2.	C	A	B
3.	B	B	C

The matching $\{A - q, B - r, C - s\}$



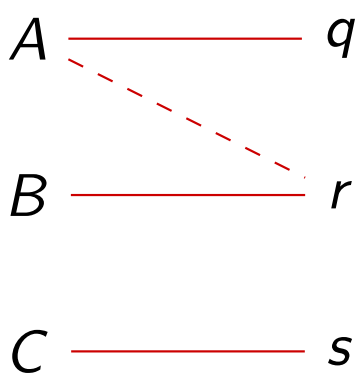
Resident Match Problem

3 doctors q, r, s and 3 hospitals A, B, C rank each other:

	A	B	C
1.	r	s	q
2.	q	q	r
3.	s	r	s

	q	r	s
1.	A	C	A
2.	C	A	B
3.	B	B	C

The matching $\{A-q, B-r, C-s\}$ is unstable, because A would rather hire r than q, and r would rather work at A than at B.



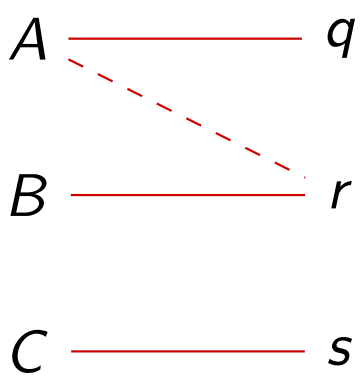
Resident Match Problem

3 doctors q, r, s and 3 hospitals A, B, C rank each other:

	A	B	C
1.	r	s	q
2.	q	q	r
3.	s	r	s

	q	r	s
1.	A	C	A
2.	C	A	B
3.	B	B	C

The matching $\{A-q, B-r, C-s\}$ is unstable, because A would rather hire r than q, and r would rather work at A than at B.



(A, r) is an unstable pair for this matching.

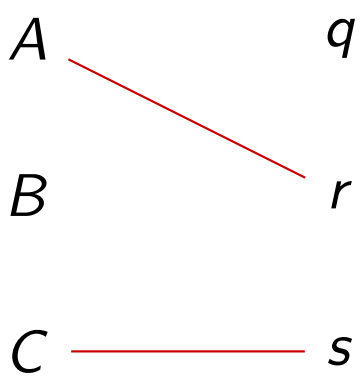
Resident Match Problem

3 doctors q, r, s and 3 hospitals A, B, C rank each other:

	A	B	C
1.	r	s	q
2.	q	q	r
3.	s	r	s

	q	r	s
1.	A	C	A
2.	C	A	B
3.	B	B	C

The matching $\{A - q, B - r, C - s\}$ is unstable, because A would rather hire r than q, and r would rather work at A than at B.



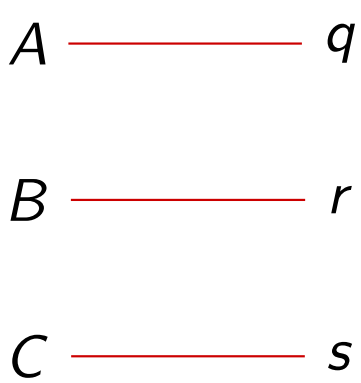
(A, r) is an unstable pair for this matching.

A possible greedy approach?

3 doctors q, r, s and 3 hospitals A, B, C rank each other:

	A	B	C
1.	r	s	q
2.	q	q	r
3.	s	r	s

	q	r	s
1.	A	C	A
2.	C	A	B
3.	B	B	C



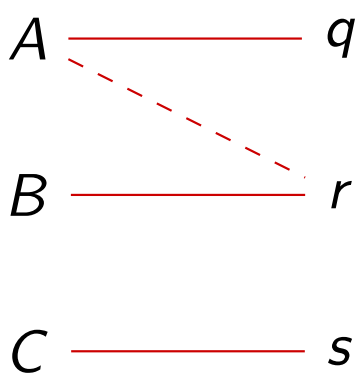
How about we greedily fix the unstable pairs?

A possible greedy approach?

3 doctors q, r, s and 3 hospitals A, B, C rank each other:

	A	B	C
1.	r	s	q
2.	q	q	r
3.	s	r	s

	q	r	s
1.	A	C	A
2.	C	A	B
3.	B	B	C



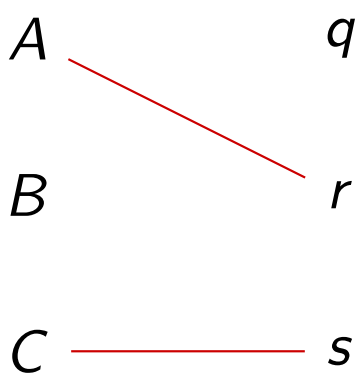
How about we greedily fix the unstable pairs?

A possible greedy approach?

3 doctors q, r, s and 3 hospitals A, B, C rank each other:

	A	B	C
1.	r	s	q
2.	q	q	r
3.	s	r	s

	q	r	s
1.	A	C	A
2.	C	A	B
3.	B	B	C



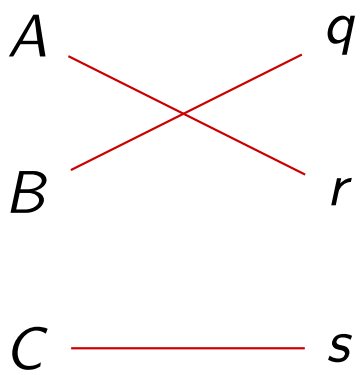
How about we greedily fix the unstable pairs?

A possible greedy approach?

3 doctors q, r, s and 3 hospitals A, B, C rank each other:

	A	B	C
1.	r	s	q
2.	q	q	r
3.	s	r	s

	q	r	s
1.	A	C	A
2.	C	A	B
3.	B	B	C



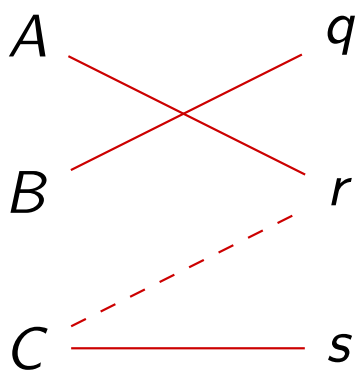
How about we greedily fix the unstable pairs?

A possible greedy approach?

3 doctors q, r, s and 3 hospitals A, B, C rank each other:

	A	B	C
1.	r	s	q
2.	q	q	r
3.	s	r	s

	q	r	s
1.	A	C	A
2.	C	A	B
3.	B	B	C



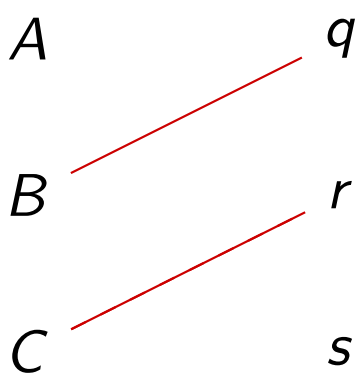
How about we greedily fix the unstable pairs?

A possible greedy approach?

3 doctors q, r, s and 3 hospitals A, B, C rank each other:

	A	B	C
1.	r	s	q
2.	q	q	r
3.	s	r	s

	q	r	s
1.	A	C	A
2.	C	A	B
3.	B	B	C



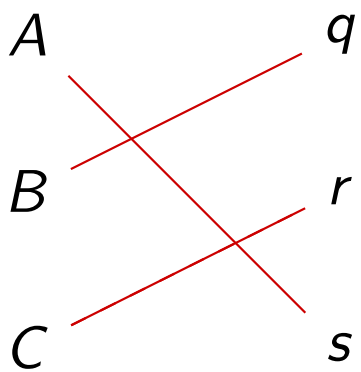
How about we greedily fix the unstable pairs?

A possible greedy approach?

3 doctors q, r, s and 3 hospitals A, B, C rank each other:

	A	B	C
1.	r	s	q
2.	q	q	r
3.	s	r	s

	q	r	s
1.	A	C	A
2.	C	A	B
3.	B	B	C



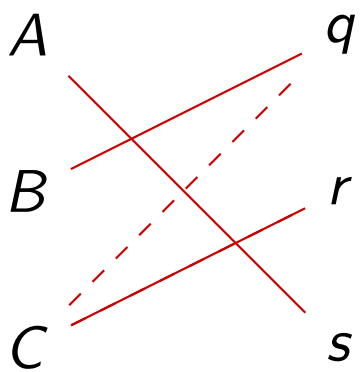
How about we greedily fix the unstable pairs?

A possible greedy approach?

3 doctors q, r, s and 3 hospitals A, B, C rank each other:

	A	B	C
1.	r	s	q
2.	q	q	r
3.	s	r	s

	q	r	s
1.	A	C	A
2.	C	A	B
3.	B	B	C



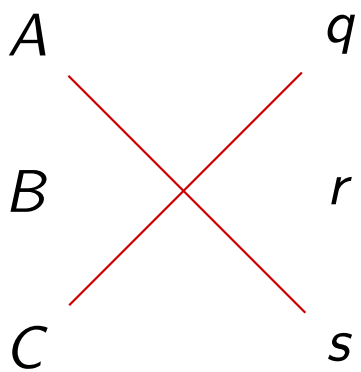
How about we greedily fix the unstable pairs?

A possible greedy approach?

3 doctors q, r, s and 3 hospitals A, B, C rank each other:

	A	B	C
1.	r	s	q
2.	q	q	r
3.	s	r	s

	q	r	s
1.	A	C	A
2.	C	A	B
3.	B	B	C



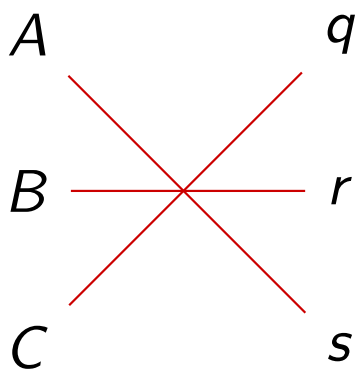
How about we greedily fix the unstable pairs?

A possible greedy approach?

3 doctors q, r, s and 3 hospitals A, B, C rank each other:

	A	B	C
1.	r	s	q
2.	q	q	r
3.	s	r	s

	q	r	s
1.	A	C	A
2.	C	A	B
3.	B	B	C



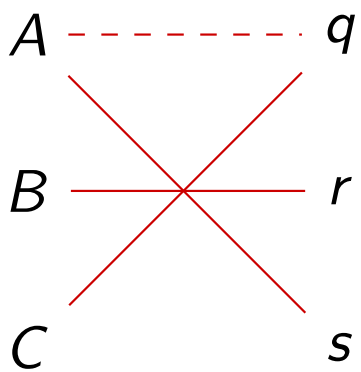
How about we greedily fix the unstable pairs?

A possible greedy approach?

3 doctors q, r, s and 3 hospitals A, B, C rank each other:

	A	B	C
1.	r	s	q
2.	q	q	r
3.	s	r	s

	q	r	s
1.	A	C	A
2.	C	A	B
3.	B	B	C



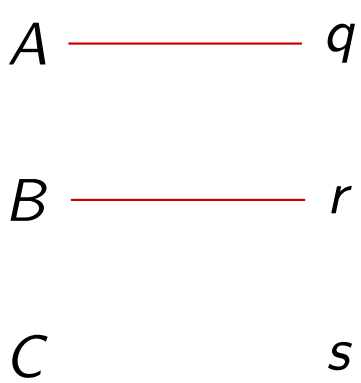
How about we greedily fix the unstable pairs?

A possible greedy approach?

3 doctors q, r, s and 3 hospitals A, B, C rank each other:

	A	B	C
1.	r	s	q
2.	q	q	r
3.	s	r	s

	q	r	s
1.	A	C	A
2.	C	A	B
3.	B	B	C



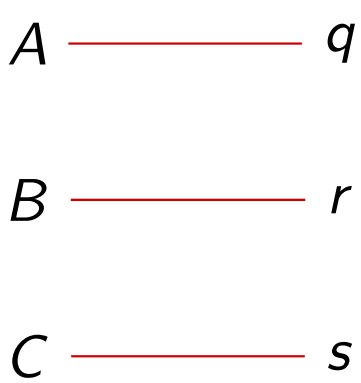
How about we greedily fix the unstable pairs?

A possible greedy approach?

3 doctors q, r, s and 3 hospitals A, B, C rank each other:

	A	B	C
1.	r	s	q
2.	q	q	r
3.	s	r	s

	q	r	s
1.	A	C	A
2.	C	A	B
3.	B	B	C



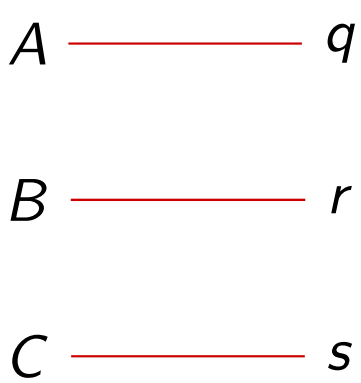
How about we greedily fix the unstable pairs?

A wrong greedy approach?

3 doctors q, r, s and 3 hospitals A, B, C rank each other:

	A	B	C
1.	r	s	q
2.	q	q	r
3.	s	r	s

	q	r	s
1.	A	C	A
2.	C	A	B
3.	B	B	C



How about we greedily fix the unstable pairs? Nope... Back to where we started.

Gale-Shapley Stable Matching algorithm

The Gale-Shapley algorithm proceeds in rounds until every position has been accepted.

Gale-Shapley Stable Matching algorithm

The Gale-Shapley algorithm proceeds in rounds until every position has been accepted.

Each round has two stages:

1. An arbitrary unmatched hospital A offers its position to the best doctor a (according to A's preference list) who has not already rejected it.
2. If a is unmatched, she (tentatively) accepts A's offer.

Gale-Shapley Stable Matching algorithm

The Gale-Shapley algorithm proceeds in rounds until every position has been accepted.

Each round has two stages:

1. An arbitrary unmatched hospital A offers its position to the best doctor a (according to A's preference list) who has not already rejected it.
2. If a is unmatched, she (tentatively) accepts A's offer. If a already has a match but prefers A, she rejects her current match and (tentatively) accepts the new offer from A.

Gale-Shapley Stable Matching algorithm

The Gale-Shapley algorithm proceeds in rounds until every position has been accepted.

Each round has two stages:

1. An arbitrary unmatched hospital A offers its position to the best doctor a (according to A's preference list) who has not already rejected it.
2. If a is unmatched, she (tentatively) accepts A's offer. If a already has a match but prefers A, she rejects her current match and (tentatively) accepts the new offer from A. Otherwise, a rejects the new offer.

Each doctor ultimately accepts the best offer that she receives, according to her preference list.

Gale-Shapley Stable Matching algorithm

The Gale-Shapley algorithm proceeds in rounds until every position has been accepted.

Each round has two stages:

1. An arbitrary unmatched hospital A offers its position to the best doctor a (according to A's preference list) who has not already rejected it.
2. If a is unmatched, she (tentatively) accepts A's offer. If a already has a match but prefers A, she rejects her current match and (tentatively) accepts the new offer from A. Otherwise, a rejects the new offer.

Each doctor ultimately accepts the best offer that she receives, according to her preference list.

Hospitals make offers greedily, doctors accept offers greedily.

3 doctors q, r, s and 3 hospitals A, B, C rank each other:

	A	B	C	D
1.	t	r	t	s
2.	s	t	r	r
3.	r	q	s	q
4.	q	s	q	t

	q	r	s	t
1.	A	A	B	D
2.	B	D	A	B
3.	C	C	C	C
4.	D	B	D	A

A q

B r

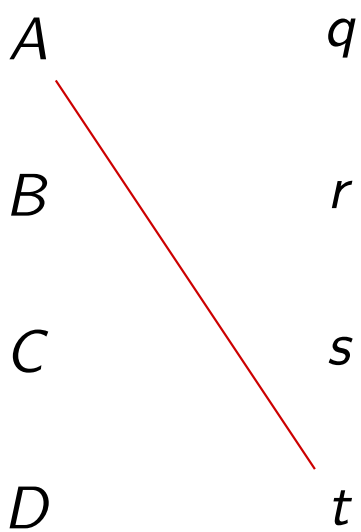
C s

D t

3 doctors q, r, s and 3 hospitals A, B, C rank each other:

	A	B	C	D
1.	t	r	t	s
2.	s	t	r	r
3.	r	q	s	q
4.	q	s	q	t

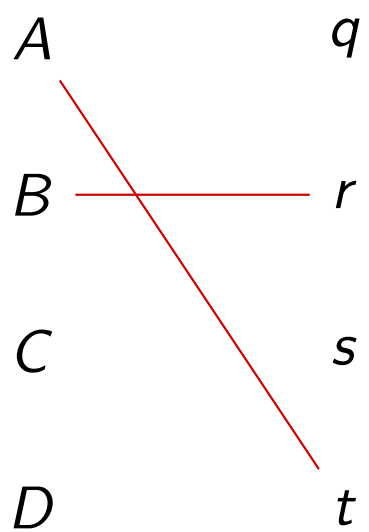
	q	r	s	t
1.	A	A	B	D
2.	B	D	A	B
3.	C	C	C	C
4.	D	B	D	A



3 doctors q, r, s and 3 hospitals A, B, C rank each other:

	A	B	C	D
1.	t	r	t	s
2.	s	t	r	r
3.	r	q	s	q
4.	q	s	q	t

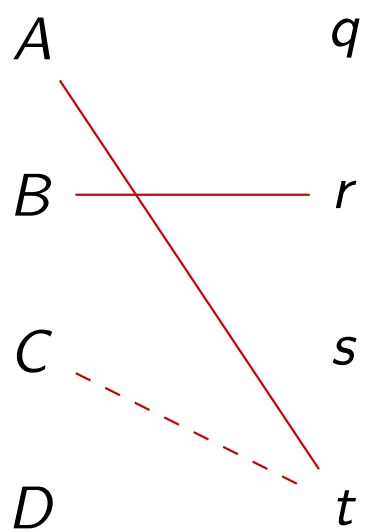
	q	r	s	t
1.	A	A	B	D
2.	B	D	A	B
3.	C	C	C	C
4.	D	B	D	A



3 doctors q, r, s and 3 hospitals A, B, C rank each other:

	A	B	C	D
1.	t	r	t	s
2.	s	t	r	r
3.	r	q	s	q
4.	q	s	q	t

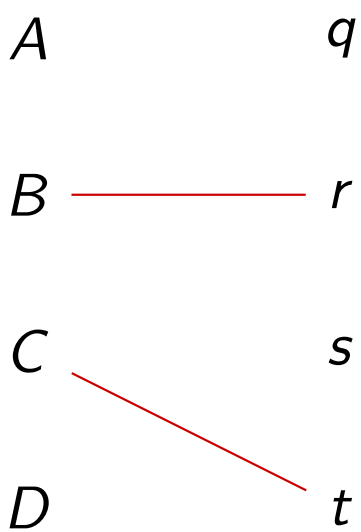
	q	r	s	t
1.	A	A	B	D
2.	B	D	A	B
3.	C	C	C	C
4.	D	B	D	A



3 doctors q, r, s and 3 hospitals A, B, C rank each other:

	A	B	C	D
1.	t	r	t	s
2.	s	t	r	r
3.	r	q	s	q
4.	q	s	q	t

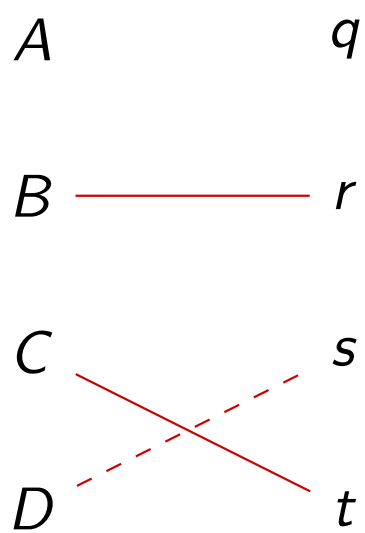
	q	r	s	t
1.	A	A	B	D
2.	B	D	A	B
3.	C	C	C	C
4.	D	B	D	A



3 doctors q, r, s and 3 hospitals A, B, C rank each other:

	A	B	C	D
1.	t	r	t	s
2.	s	t	r	r
3.	r	q	s	q
4.	q	s	q	t

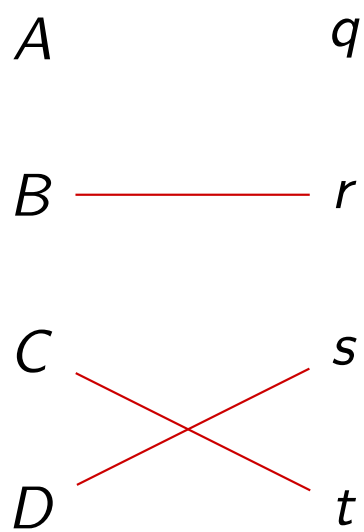
	q	r	s	t
1.	A	A	B	D
2.	B	D	A	B
3.	C	C	C	C
4.	D	B	D	A



3 doctors q, r, s and 3 hospitals A, B, C rank each other:

	A	B	C	D
1.	t	r	t	s
2.	s	t	r	r
3.	r	q	s	q
4.	q	s	q	t

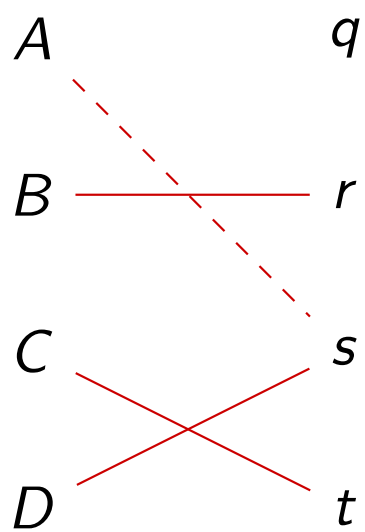
	q	r	s	t
1.	A	A	B	D
2.	B	D	A	B
3.	C	C	C	C
4.	D	B	D	A



3 doctors q, r, s and 3 hospitals A, B, C rank each other:

	A	B	C	D
1.	t	r	t	s
2.	s	t	r	r
3.	r	q	s	q
4.	q	s	q	t

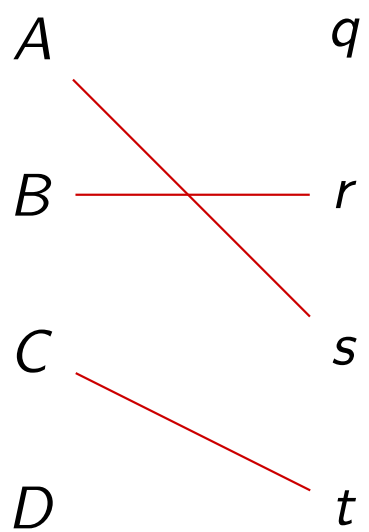
	q	r	s	t
1.	A	A	B	D
2.	B	D	A	B
3.	C	C	C	C
4.	D	B	D	A



3 doctors q, r, s and 3 hospitals A, B, C rank each other:

	A	B	C	D
1.	t	r	t	s
2.	s	t	r	r
3.	r	q	s	q
4.	q	s	q	t

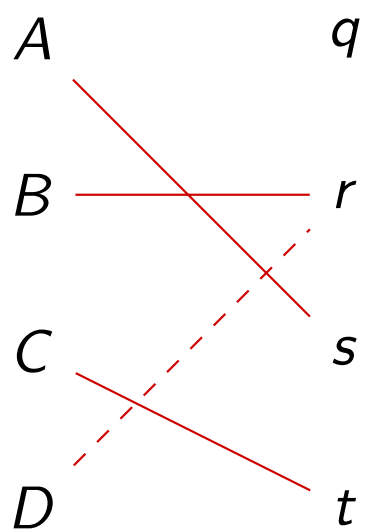
	q	r	s	t
1.	A	A	B	D
2.	B	D	A	B
3.	C	C	C	C
4.	D	B	D	A



3 doctors q, r, s and 3 hospitals A, B, C rank each other:

	A	B	C	D
1.	t	r	t	s
2.	s	t	r	r
3.	r	q	s	q
4.	q	s	q	t

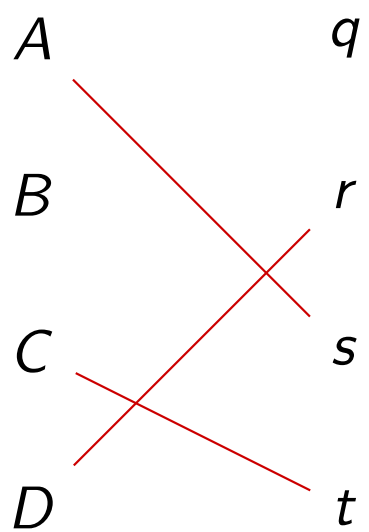
	q	r	s	t
1.	A	A	B	D
2.	B	D	A	B
3.	C	C	C	C
4.	D	B	D	A



3 doctors q, r, s and 3 hospitals A, B, C rank each other:

	A	B	C	D
1.	t	r	t	s
2.	s	t	r	r
3.	r	q	s	q
4.	q	s	q	t

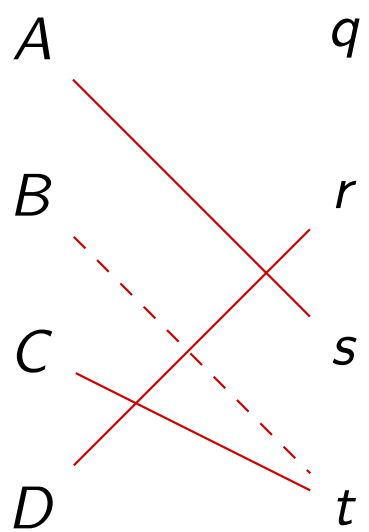
	q	r	s	t
1.	A	A	B	D
2.	B	D	A	B
3.	C	C	C	C
4.	D	B	D	A



3 doctors q, r, s and 3 hospitals A, B, C rank each other:

	A	B	C	D
1.	t	r	t	s
2.	s	t	r	r
3.	r	q	s	q
4.	q	s	q	t

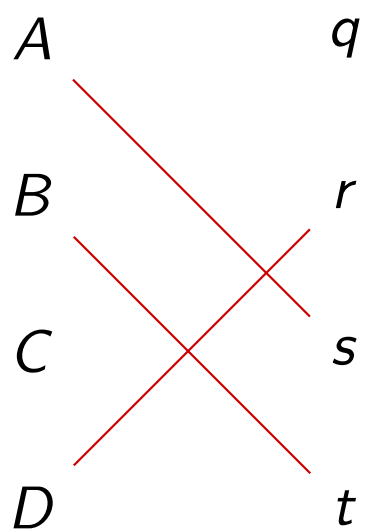
	q	r	s	t
1.	A	A	B	D
2.	B	D	A	B
3.	C	C	C	C
4.	D	B	D	A



3 doctors q, r, s and 3 hospitals A, B, C rank each other:

	A	B	C	D
1.	t	r	t	s
2.	s	t	r	r
3.	r	q	s	q
4.	q	s	q	t

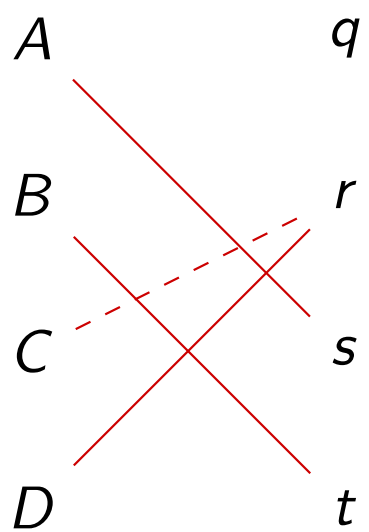
	q	r	s	t
1.	A	A	B	D
2.	B	D	A	B
3.	C	C	C	C
4.	D	B	D	A



3 doctors q, r, s and 3 hospitals A, B, C rank each other:

	A	B	C	D
1.	t	r	t	s
2.	s	t	r	r
3.	r	q	s	q
4.	q	s	q	t

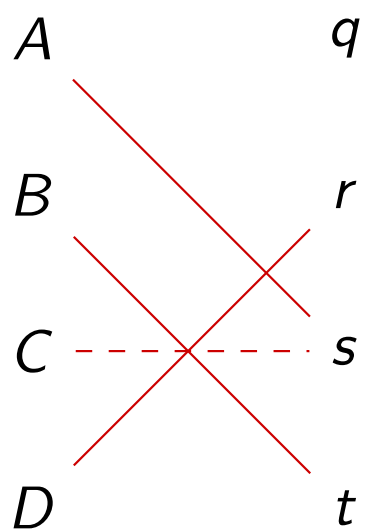
	q	r	s	t
1.	A	A	B	D
2.	B	D	A	B
3.	C	C	C	C
4.	D	B	D	A



3 doctors q, r, s and 3 hospitals A, B, C rank each other:

	A	B	C	D
1.	t	r	t	s
2.	s	t	r	r
3.	r	q	s	q
4.	q	s	q	t

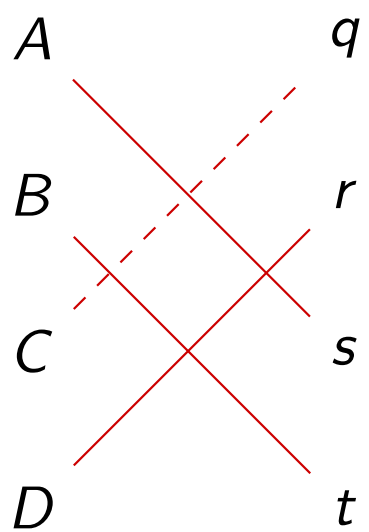
	q	r	s	t
1.	A	A	B	D
2.	B	D	A	B
3.	C	C	C	C
4.	D	B	D	A



3 doctors q, r, s and 3 hospitals A, B, C rank each other:

	A	B	C	D
1.	t	r	t	s
2.	s	t	r	r
3.	r	q	s	q
4.	q	s	q	t

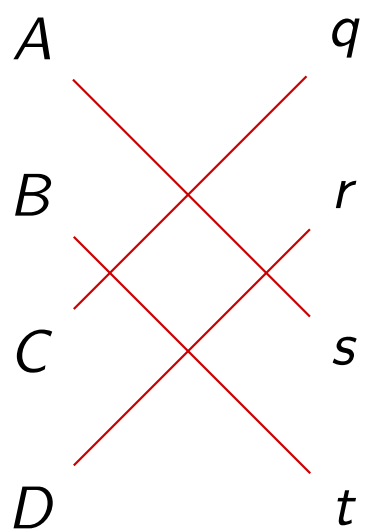
	q	r	s	t
1.	A	A	B	D
2.	B	D	A	B
3.	C	C	C	C
4.	D	B	D	A



3 doctors q, r, s and 3 hospitals A, B, C rank each other:

	A	B	C	D
1.	t	r	t	s
2.	s	t	r	r
3.	r	q	s	q
4.	q	s	q	t

	q	r	s	t
1.	A	A	B	D
2.	B	D	A	B
3.	C	C	C	C
4.	D	B	D	A



Algorithm correctness

We first argue that the algorithm produces a matching of doctors to hospitals.

Algorithm correctness

We first argue that the algorithm produces a matching of doctors to hospitals.

If any doctor is unmatched, then no hospital has offered that doctor a job,

Algorithm correctness

We first argue that the algorithm produces a matching of doctors to hospitals.

If any doctor is unmatched, then no hospital has offered that doctor a job, which implies that the hospitals have not exhausted their preference lists.

Algorithm correctness

We first argue that the algorithm produces a matching of doctors to hospitals.

If any doctor is unmatched, then no hospital has offered that doctor a job, which implies that the hospitals have not exhausted their preference lists.

Also, after received their first offer doctors will always be matched to exactly one hospital.

Algorithm correctness

We first argue that the algorithm produces a matching of doctors to hospitals.

If any doctor is unmatched, then no hospital has offered that doctor a job, which implies that the hospitals have not exhausted their preference lists.

Also, after received their first offer doctors will always be matched to exactly one hospital.

It follows that when the algorithm terminates (after at most n^2 rounds), every doctor is matched, and therefore every position is filled.

Algorithm correctness

We first argue that the algorithm produces a matching of doctors to hospitals.

If any doctor is unmatched, then no hospital has offered that doctor a job, which implies that the hospitals have not exhausted their preference lists.

Also, after received their first offer doctors will always be matched to exactly one hospital.

It follows that when the algorithm terminates (after at most n^2 rounds), every doctor is matched, and therefore every position is filled.

In other words, the algorithm always computes a perfect matching between doctors and hospitals.

Algorithm correctness

We now show that the matching is stable.

Algorithm correctness

We now show that the matching is stable.

Suppose the algorithm matches some doctor a to some hospital A, even though she prefers another hospital B.

Algorithm correctness

We now show that the matching is stable.

Suppose the algorithm matches some doctor a to some hospital A, even though she prefers another hospital B.

Because every doctor accepts the best offer she receives, a received no offer she liked more than A;

Algorithm correctness

We now show that the matching is stable.

Suppose the algorithm matches some doctor a to some hospital A, even though she prefers another hospital B.

Because every doctor accepts the best offer she receives, a received no offer she liked more than A; in particular, B never made an offer to a.

Algorithm correctness

We now show that the matching is stable.

Suppose the algorithm matches some doctor a to some hospital A, even though she prefers another hospital B.

Because every doctor accepts the best offer she receives, a received no offer she liked more than A; in particular, B never made an offer to a.

On the other hand, B made offers to every doctor they prefer over their final match b. It follows that B prefers b over a, which means (a, B) is not an unstable pair.

Algorithm correctness

We now show that the matching is stable.

Suppose the algorithm matches some doctor a to some hospital A, even though she prefers another hospital B.

Because every doctor accepts the best offer she receives, a received no offer she liked more than A; in particular, B never made an offer to a.

On the other hand, B made offers to every doctor they prefer over their final match b. It follows that B prefers b over a, which means (a, B) is not an unstable pair.

We conclude that there are no unstable pairs; the matching is stable!