MCM: Optimal Matrix Chain Multiplication

example of Dynamic Programming

Dynamic Programming vs. Recursion and Divide & Conquer

- In a recursive program, a problem of size *n* is solved by first solving a sub-problem of size *n-1*.
- In a divide & conquer program, you solve a problem of size *n* by first solving a sub-problem of size *k* and another of size *k-1*, where $1 < k < n$.
- In dynamic programming, you solve a problem of size *n* by first solving *all* sub-problems of *all* sizes *k*, where $k < n$.

Matrix Multiplication

Matrix A - *p* rows & *q* columns (dimension: p * q)

Matrix B - *q* rows & *r* columns (diamension: q * r)

Note: We can only multiply two matrices, A & B,
if number of columns in A = number of rows in B

Cost of AB = # of scalar multiplications

= *p q r*

Matrix-Chain Multiplication

Given a sequence of *n* matrices: $A_1A_2 \ldots A_n$

Compute $= A_1 * A_2 * \ldots * A_n$

The way we parenthesize a chain of matrices can have a dramatic impact on the cost of evaluating the product.

Matrix Chain Multiplication

- Given some matrices to multiply, determine the ***best*** order to multiply them so you minimize the number of single element multiplications.
  - i.e. Determine the way the matrices are parenthesized.

- First off, it should be noted that matrix multiplication is associative, but not commutative. But since it is associative, we always have:

- ((AB)(CD)) = (A(B(CD))), or any other grouping as long as the matrices are in the same consecutive order.

- BUT NOT: ((AB)(CD)) = ((BA)(DC))

Matrix-chain multiplication

- A product of matrices is **fully parenthesized** if it is either a single matrix, or a product of two fully parenthesized matrix product, surrounded by parentheses.

Example

Consider the chain: $A_1A_2A_3A_4$ of 4 matrices

There are 5 possible ways to compute the product $A_1A_2A_3A_4$

1. $(A_1(A_2(A_3A_4)))$
2. $(A_1((A_2A_3)A_4))$
3. $((A_1A_2)(A_3A_4))$
4. $((A_1(A_2A_3))A_4)$
5. $(((A_1A_2)A_3)A_4)$

Matrix-chain Multiplication

- To compute the number of scalar multiplications necessary, we must know:
  - Algorithm to multiply two matrices
  - Matrix dimensions

- Can you write the algorithm to multiply two matrices?

Algorithm to Multiply 2 Matrices

Input: Matrices $A_{p\times q}$ and $B_{q\times r}$ (with dimensions $p\times q$ and $q\times r$)
Result: Matrix $C_{p\times r}$ resulting from the product $A\cdot B$

MATRIX-MULTIPLY$(A_{p\times q}, B_{q\times r})$

```
1.  for i ← 1 to p
2.          for j ← 1 to r
3.                  C[i, j] ← 0
4.                  for k ← 1 to q
5.                          C[i, j] ← C[i, j] + A[i, k] · B[k, j]
6.  return C
```

Scalar multiplication in line 5 dominates time to compute CNumber of scalar multiplications = *pqr*

MATRIX MULTIPLY

```
MATRIX-MULTIPLY(A, B)
1  if A.columns ≠ B.rows
2       error "incompatible dimensions"
3  else let C be a new A.rows × B.columns matrix
4       for i = 1 to A.rows
5            for j = 1 to B.columns
6                 c_ij = 0
7                 for k = 1 to A.columns
8                      c_ij = c_ij + a_ik · b_kj
9       return C
```

Complexity:

- Let ***A*** be a $p \times q$ matrix, and ***B*** be a $q \times r$ matrix.
- Then the complexity is $p \times q \times r$

Matrix-Chain Multiplication Problem

Computing the minimum cost order of multiplying a list of n matrices:

$$A_1 A_2 \ldots A_n$$

For i = 1,2, ..,n; matrix A_i has dimension $p_{i-1} * p_i$

The number of alternative parenthesizations is exponential in *n (catalan number).*

Matrix-chain Multiplication

- Example: Consider three matrices $A_{10\times100}$, $B_{100\times5}$, and $C_{5\times50}$
- There are 2 ways to parenthesize
  - $((AB)C) = D_{10\times5} \cdot C_{5\times50}$
    - $AB \Rightarrow 10\cdot100\cdot5=5{,}000$ scalar multiplications
    - $DC \Rightarrow 10\cdot5\cdot50 = 2{,}500$ scalar multiplications

    } Total: 7,500
  - $(A(BC)) = A_{10\times100} \cdot E_{100\times50}$
    - $BC \Rightarrow 100\cdot5\cdot50=25{,}000$ scalar multiplications
    - $AE \Rightarrow 10\cdot100\cdot50 = 50{,}000$ scalar multiplications

Total: 75,000

Matrix Chain Multiplication
Optimal Parenthesization

- Example: A[30][35], B[35][15], C[15][5]
  minimum of A*B*C
    A*(B*C) = 30*35*5 + 35*15*5 = 7,585
    (A*B)*C = 30*35*15 + 30*15*5 = 18,000
- How to optimize:
  - Brute force – look at every possible way to parenthesize : $\Omega(4^n/n^{3/2})$
  - Dynamic programming – time complexity of $\Omega(n^3)$ and space complexity of $\Theta(n^2)$.

Matrix Chain Multiplication

- Let's get back to our example: We will show that the way we group matrices when multiplying A, B, C ***matters***:
  - Let A be a 2x10 matrix
  - Let B be a 10x50 matrix
  - Let C be a 50x20 matrix

- Consider computing **A(BC):**
  - # multiplications for (BC) = 10x50x20 = 10000, creating a 10x20 answer matrix
  - # multiplications for A(BC) = 2x10x20 = 400
  - Total multiplications = 10000 + 400 = 10400.

- Consider computing **(AB)C**:
  - # multiplications for (AB) = 2x10x50 = 1000, creating a 2x50 answer matrix
  - # multiplications for (AB)C = 2x50x20 = 2000,
  - Total multiplications = 1000 + 2000 = 3000

Matrix Chain Multiplication

- Our final multiplication will ALWAYS be of the form
  - $(A_0 \bullet A_1 \bullet \ldots A_k) \bullet (A_{k+1} \bullet A_{k+2} \bullet \ldots A_{n-1})$
- In essence, there is exactly one value of k for which we should "split" our work into two separate cases so that we get an optimal result.
  - Here is a list of the cases to choose from:
  - $(A_0) \bullet (A_1 \bullet A_{k+2} \bullet \ldots A_{n-1})$
  - $(A_0 \bullet A_1) \bullet (A_2 \bullet A_{k+2} \bullet \ldots A_{n-1})$
  - $(A_0 \bullet A_1 \bullet A_2) \bullet (A_3 \bullet A_{k+2} \bullet \ldots A_{n-1})$
  - ...
  - $(A_0 \bullet A_1 \bullet \ldots A_{n-3}) \bullet (A_{n-2} \bullet A_{n-1})$
  - $(A_0 \bullet A_1 \bullet \ldots A_{n-2}) \bullet (A_{n-1})$
- Basically, count the number of multiplications in each of these choices and **pick the minimum**.
  - One other point to notice is that you have to account for the minimum number of multiplications in each of the two products.

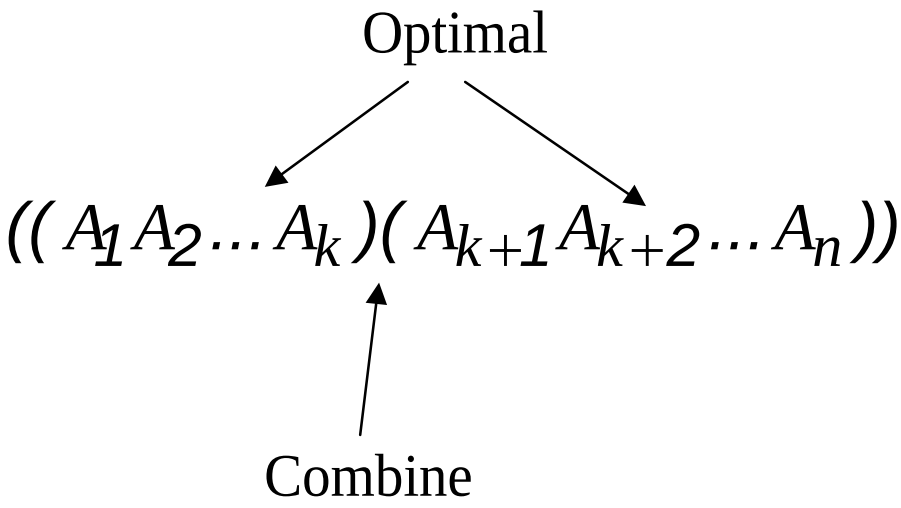
Catalan number: Number of ways to parenthesize

$$P(n) = \begin{cases} 1 & \text{if } n = 1 \\ \sum_{k=1}^{n-1} P(k)\,p(n-k) & \text{if } n \geq 2 \end{cases}$$

$$P(n) = C(n-1)$$

$$= \frac{1}{n+1}\binom{2n}{n} = \Omega\left(\frac{4^n}{n^{3/2}}\right)$$

Step 1: The structure of an optimal parenthesization



Step 2: A recursive solution

- Define *m*[*i, j*]= minimum number of scalar multiplications needed to compute the matrix $A_{i..j} = A_i A_{i+1} .. A_j$
- Goal find *m*[*1, n*]
-

$$m[i,j] = \begin{cases} 0 & i = j \\ \min_{i \le k < j} \{m[i,k] + m[k+1,j] + p_{i-1} p_k p_j\} & i \neq j \end{cases}$$

How to split the chain?

- Basically, we're checking different places to "split" our matrices by checking different values of k and seeing if they improve our current minimum value.

Step 3. Compute Optimal Cost

Input: Array $p[0\ldots n]$ containing matrix dimensions and n

Result: Minimum-cost table $m[1..n,1..n]$ and split table $s[1..n,1..n]$

MATRIX-CHAIN-ORDER($p[\,]$, n)

```
for i ← 1 to n
    m[i, i] ← 0
for l ← 2 to n
    for i ← 1 to n-l+1
            j ← i+l-1
            m[i, j] ← ∞
            for k ← i to j-1
                    q ← m[i, k] + m[k+1, j] + p[i-1] p[k] p[j]
                    if q < m[i, j]
                            m[i, j] ← q
                            s[i, j] ← k
return m and s
```

Takes $O(n^3)$ time

Requires $O(n^2)$ space

Constructing Optimal Solution

- Our algorithm computes the minimum-cost table m and the split table s
- The optimal solution can be constructed from the split table s
- Each entry $s[i, j] = k$ shows where to split (cut) the product chain $A_i A_{i+1} \ldots A_j$ for the minimum cost.

Step 4. Find the optimal cuts

```
PRINT-OPTIMAL-PARENS(s, i, j)
1  if i == j
2      print "A"_i
3  else print "("
4      PRINT-OPTIMAL-PARENS(s, i, s[i, j])
5      PRINT-OPTIMAL-PARENS(s, s[i, j] + 1, j)
6      print ")"
```

- example:

$$((A_1(A_2A_3))((A_4A_5)A_6))$$

Example: Given 6 matrices to multiply

$$
\begin{aligned}
A_1 \quad & 30 \times 35 \quad = p_0 \times p_1 \\
A_2 \quad & 35 \times 15 \quad = p_1 \times p_2 \\
A_3 \quad & 15 \times 5 \quad = p_2 \times p_3 \\
A_4 \quad & 5 \times 10 \quad = p_3 \times p_4 \\
A_5 \quad & 10 \times 20 \quad = p_4 \times p_5 \\
A_6 \quad & 20 \times 25 \quad = p_5 \times p_6
\end{aligned}
$$

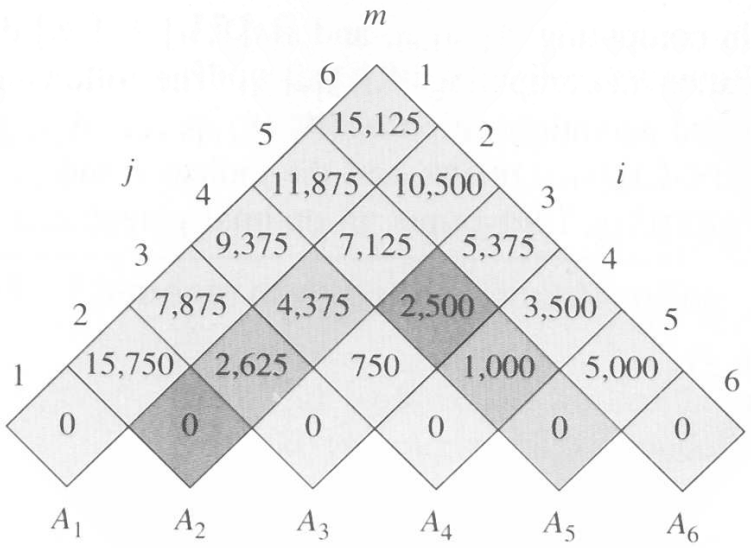
Find the cheapest way to multiply these 6 matrices

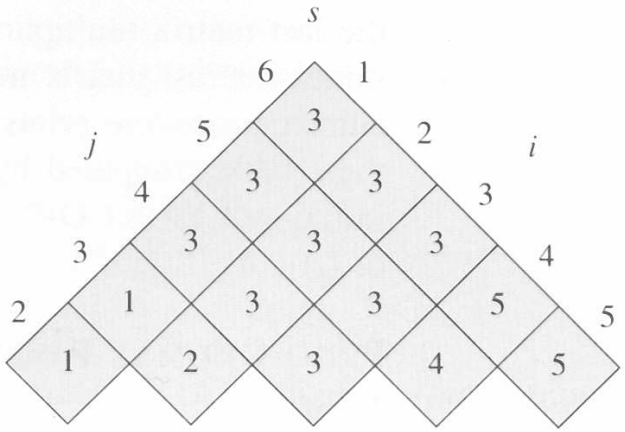
- Show how to multiply this matrix chain optimally

- Brute force Solution
  - Minimum cost 15 125
  - Optimal parenthesization $((A_1(A_2A_3))((A_4\,A_5)A_6))$

Matrix	Dimension
A_1	30×35
A_2	35×15
A_3	15×5
A_4	5×10
A_5	10×20
A_6	20×25

The m and s table computed by MATRIX-CHAIN-ORDER for n=6





((A1(A2A3))((A4 A5)A6))

Example, computation of m[2,5]

$m[2,5]= 7125 =$

Minimum of the 3 cuts {

$m[2,2]+m[3,5]+p_1p_2p_5 = 0+2500+35\times15\times20 = 13000,$

$m[2,3]+m[4,5]+p_1p_3p_5 = 2625+1000+35\times5\times20 = 7125,$

$m[2,4]+m[5,5]+p_1p_4p_5 = 4375+0+35\times10\times20 = 11374$

}

Example using the C program mcm.exe:

Input: A1[30x35] A2[35x15] A3[15x5] A4[5x10] A5[10x20] A6[20x25]

```
./mcm.exe 30 35 15 5 10 20 25
```

order of filling m is:

```
 |      1|      2|      3|      4|      5|      6
-+-------+-------+-------+-------+-------+-------+
1:      0       1       7      15      22      29
2:              0       2       9      17      24
3:                      0       3      10      18
4:                              0       4      12
5:                                      0       5
6:                                              0
```

Example: continued **A1[30x35] A2[35x15] A3[15x5] A4[5x10] A5[10x20] A6[20x25]**

m is:

	1	2	3	4	5	6
1:	0	15750	7875	9375	11875	15125
2:		0	2625	4375	7125	10500
3:			0	750	2500	5375
4:				0	1000	3500
5:					0	5000
6:						0

Example: continued **A1[30x35] A2[35x15] A3[15x5] A4[5x10] A5[10x20] A6[20x25]**

s is:

	1	2	3	4	5	6
1:	0	1	1	3	3	3
2:		0	2	3	3	3
3:			0	3	3	3
4:				0	4	5
5:					0	5
6:						0

Elements of dynamic programming

- Optimal substructure
- Overlapping subproblems
- How memoization might help

Exercise

```
mcm 1 2 3 4

A1[1x2]
 A2[2x3]
 A3[3x4]
```

Solution

```
mcm 1 2 3 4; A1[1x2] A2[2x3] A3[3x4]

m is:
 |      1|      2|      3
-+-------+-------+-------+
1:      0       6      18
2:              0      24
3:                      0
s is:
 |      1|      2|      3
-+-------+-------+-------+
1:      0       1       2
2:              0       2
3:                      0
Minimum mult is 18 by:(( A1 A2) A3)
```

Exercise

mcm 3 4 1 2

A1[3x4] A2[4x1]

A3[1x2]

Exercise Solution

```
A1[3x4] A2[4x1] A3[1x2]

m|      1|      2|      3
-+-------+-------+-------+
1:      0      12      18
2:              0       8
3:                      0

s|      1|      2|      3
-+-------+-------+-------+
1:      0       1       2
2:              0       2
3:                      0
Minimum mult is 18 by:(( A1 A2) A3)
```

Exercise

mcm.exe 1 5 3 2 1

A1[1x5]* A2[5x3]* A3[3x2]* A4[2x1]

```
m is:
 |      1|      2|      3|      4
-+-------+-------+-------+-------+
1:      0      15      21      23
2:              0      30      21
3:                      0       6
4:                              0

s is:
 |      1|      2|      3|      4
-+-------+-------+-------+-------+
1:      0       1       2       3
2:              0       2       2
3:                      0       3
Minimum mult is 23 by:((( A1 A2) A3) A4)
```

Exercise Solution

```
A1[1x5]* A2[5x3]* A3[3x2]* A4[2x1]
-----------------------------------
m|      1|      2|      3|      4
-+-------+-------+-------+-------+
1:      0      15      21      23
2:              0      30      21
3:                      0       6
4:                              0
-----------------------------------
s|      1|      2|      3|      4
-+-------+-------+-------+-------+
1:      0       1       2       3
2:              0       2       2
3:                      0       3
Minimum mult is 23 by:((( A1 A2) A3) A4)
```

Bipartite Matching

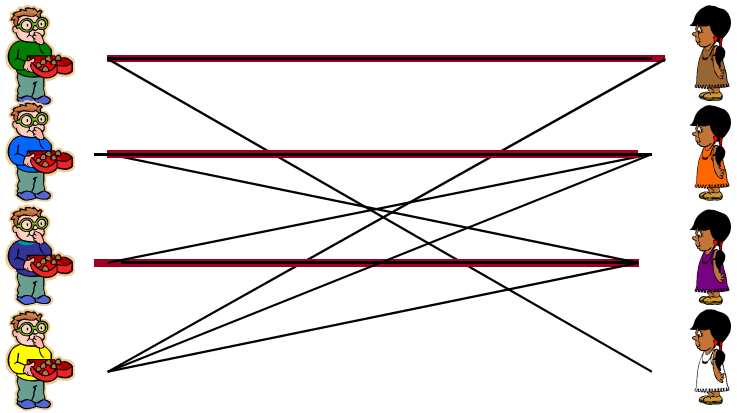
in Graphs

Bipartite Matching

The Bipartite Marriage Problem:

- There are n boys and n girls.
- For each pair, the pair is either compatible or not.

Goal: to find the maximum number of compatible pairs.

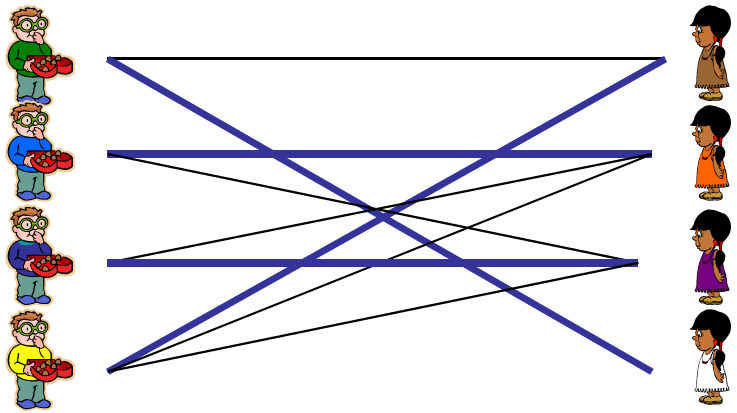


Bipartite Matching

The Bipartite Marriage Problem:

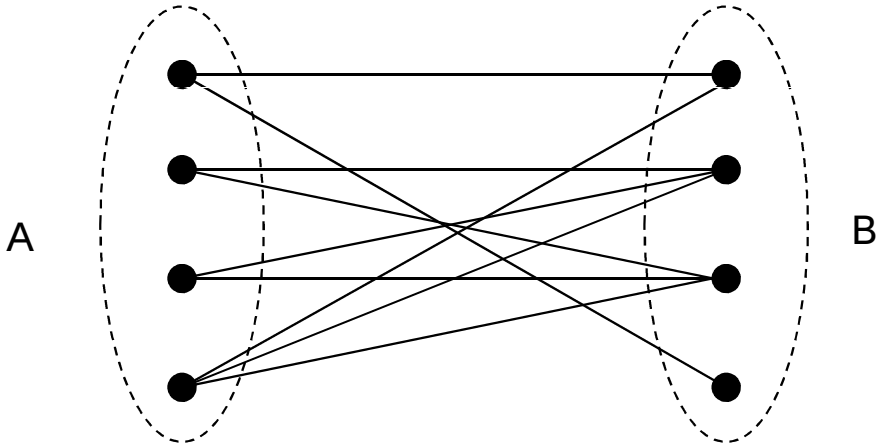
- There are n boys and n girls.
- For each pair, it is either compatible or not.

Goal: to find the maximum number of compatible pairs.



Graph Problem

A graph is bipartite if its vertex set can be partitioned into two subsets A and B so that each edge has one endpoint in A and the other endpoint in B.

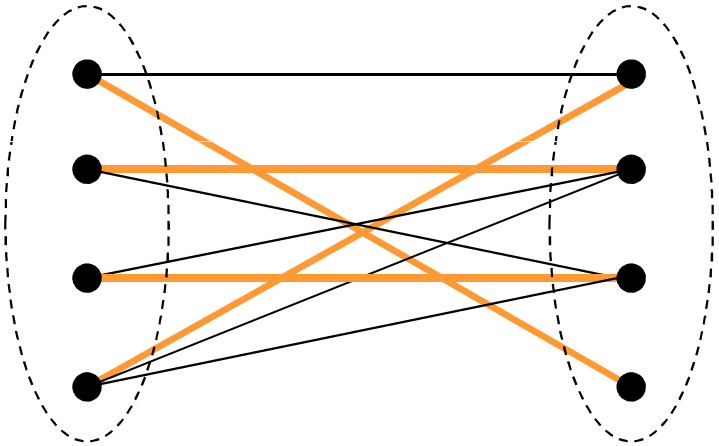


A matching is a subset of edges so that every vertex has degree at most one.

Maximum Matching

The bipartite matching problem:

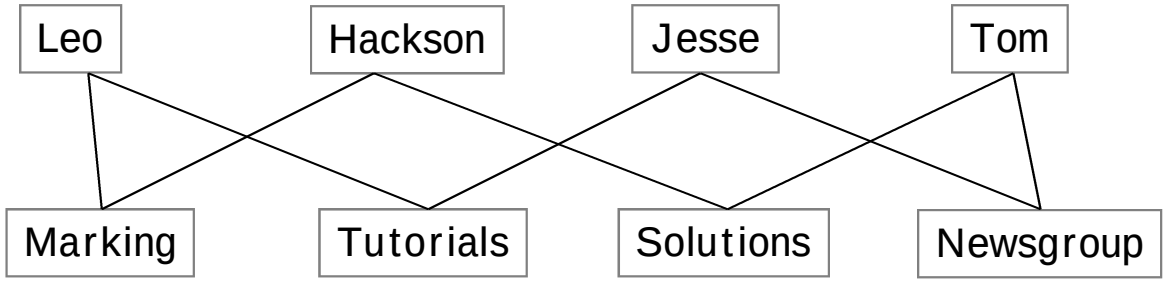
Find a matching with the maximum number of edges.



A perfect matching is a matching in which every vertex is matched.

The perfect matching problem: Is there a perfect matching?

Application of Bipartite Matching



Job Assignment Problem:

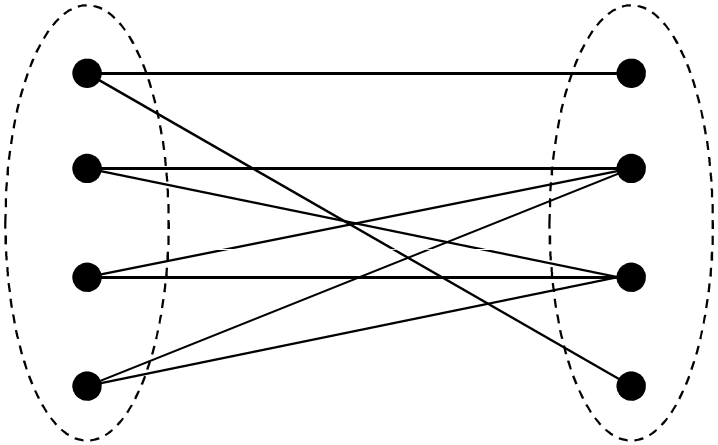
Each person is willing to do a subset of jobs.

Can you find an assignment so that all jobs are taken care of?

In fact, there is an efficient procedure to find such as assignment!

Perfect Matching

Does a perfect matching always exist?

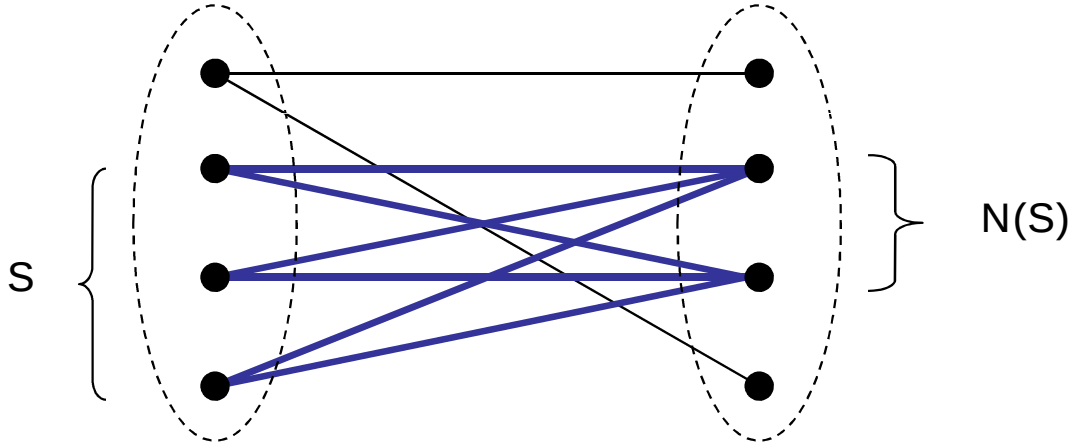


Suppose you work for shadi.com, and your job is to find a perfect matching between 200 men and 200 women. If there is a perfect matching, then you can show it to the shadi.com. But suppose there is no perfect matching, how can you convince your boss of this fact?

Perfect Matching

Does a perfect matching always exist?

Of course not, eg.
If there are more vertices on one side,
then of course it is impossible.



Let N(S) be the neighbours of vertices in S, for every subset S.

If |N(S)| < |S|, then it is impossible to have a perfect matching.

Hall's Theorem:

A Necessary and Sufficient Condition.

It tells you exactly when a bipartite graph does and does-not have a perfect matching.

Hall's Theorem: A bipartite graph G=(V,W;E) has a perfect matching

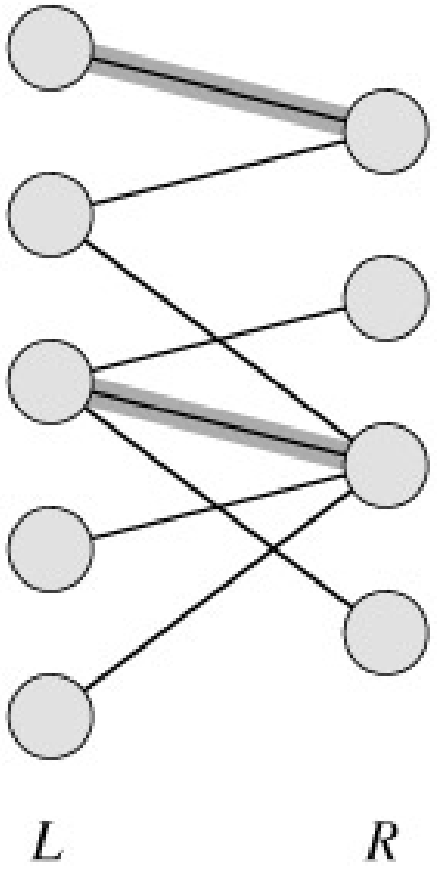
if and only if |N(S)| >= |S|

for every subset S of V and W.

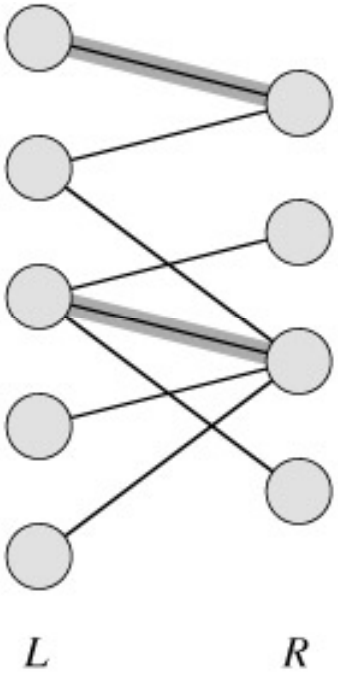
An Application of Max Flow: Maximum Bipartite Matching

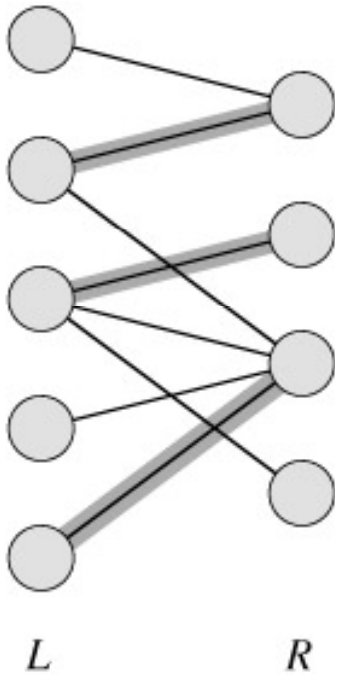
Maximum Bipartite Matching

- A bipartite graph is a graph *G=(V,E)* in which *V* can be divided into two parts *L* and *R* such that every edge in *E* is between a vertex in *L* and a vertex in *R*.
- e.g. vertices in *L* represent skilled workers and vertices in *R* represent jobs. An edge connects workers to jobs they can perform.



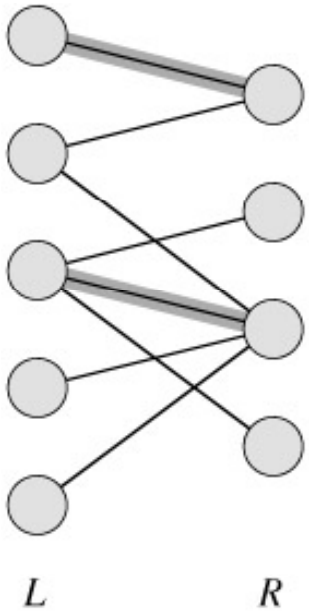
A matching in a graph is a subset *M* of *E*, such that for all vertices *v* in *V*, at most one edge of *M* is incident on *v*.



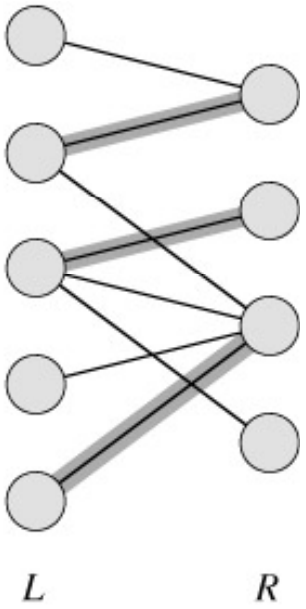


A **maximum matching** is a matching of maximum size (maximum number of edges).

not maximum

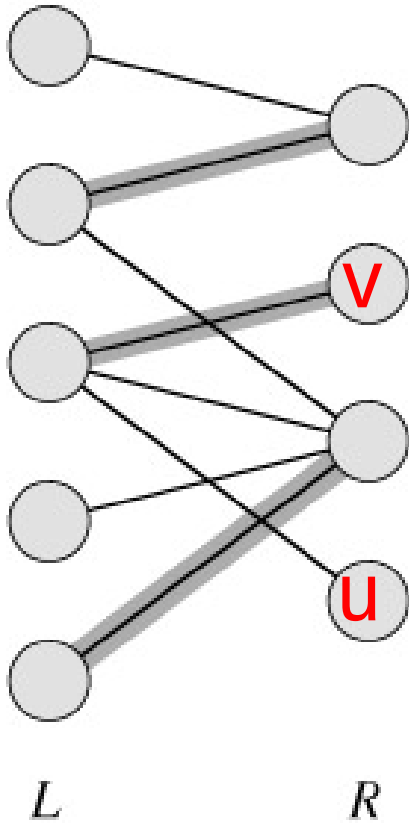


maximum



A Maximum Matching

- No matching of cardinality 4, because only one of v and u can be matched.
- In the workers-jobs example a maximal-matching provides work for as many people as possible.



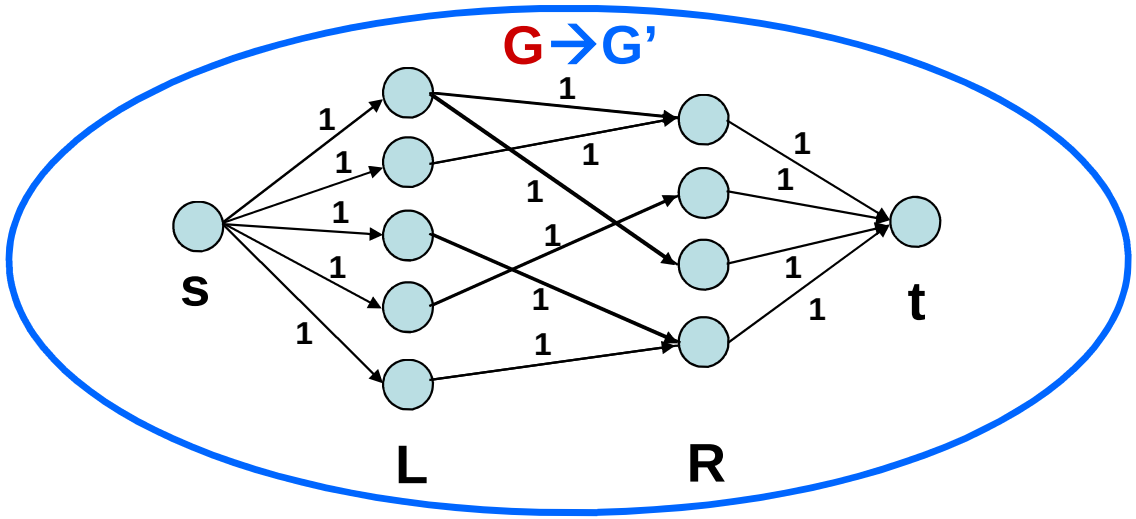
Solving the Maximum Bipartite Matching Problem

- Reduce the maximum bipartite matching problem on graph **G** to the max-flow problem on a corresponding flow network **G'**.

- Solve using Ford-Fulkerson method.

Corresponding Flow Network

- To form the corresponding flow network **G'** of the bipartite graph **G**:
  - Add a source vertex s and edges from s to L.
  - Direct the edges in E from L to R.
  - Add a sink vertex t and edges from R to t.
  - Assign a capacity of 1 to all edges.
- Claim: max-flow in **G'** corresponds to a max-bipartite-matching on **G**.



Solving Bipartite Matching as Max Flow

Let $G = (V, E)$ be a bipartite graph with vertex partition $V = L \cup R$.

Let $G' = (V', E')$ be its corresponding flow network.

If M is a matching in G,

then there is an integer-valued flow f in G' with value $|f| = |M|$.

Conversely if f is an integer-valued flow in G',

then there is a matching M in G with cardinality $|M| = |f|$.

Thus $\max |M| = \max(\text{integer } |f|)$

Does this mean that max |f| = max |M|?

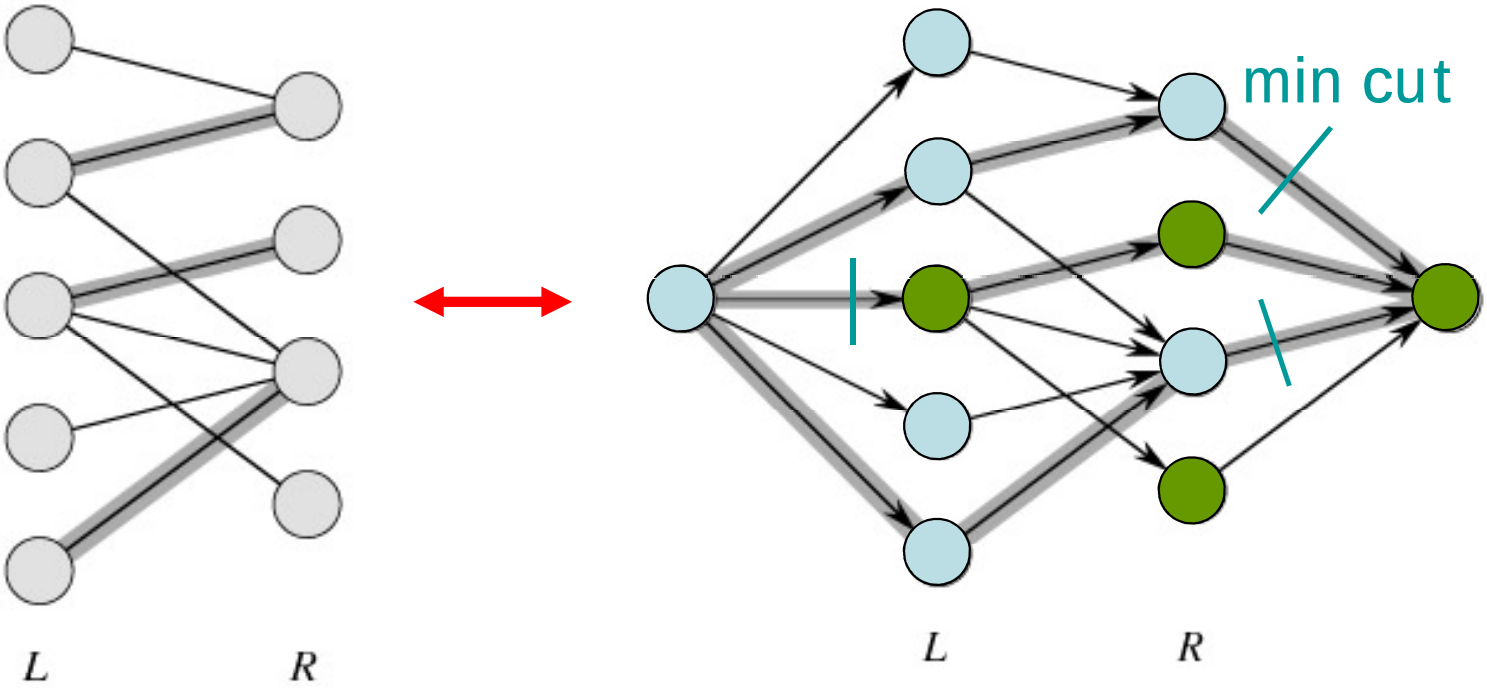
- Problem: we haven't shown that the max flow f(u,v) is necessarily integer-valued.

Integrality Theorem

- If the capacity function c takes on only integral values, then:
    1. The maximum flow f produced by the Ford-Fulkerson method has the property that |f| is integer-valued.
    2. For all vertices u and v the value f(u,v) of the flow is an integer.

- So max|M| = max |f|

Example



$|M| = 3 \quad \leftrightarrow \quad \max \text{ flow} = |f| = 3$

Conclusion

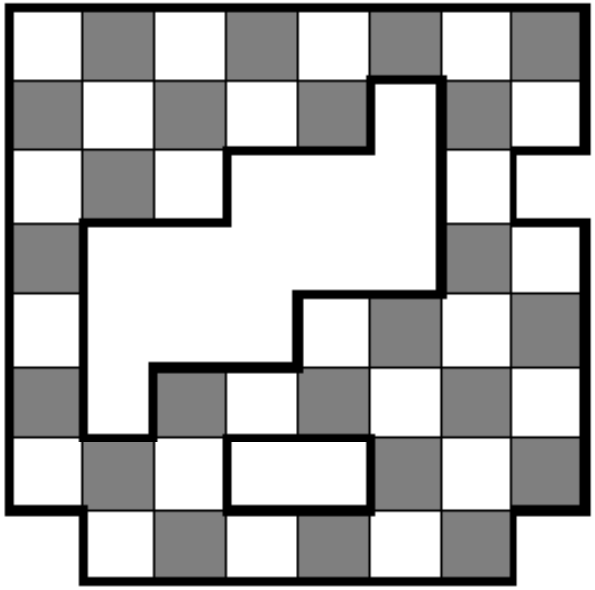
- Network flow algorithms allow us to find the maximum bipartite matching fairly easily.
- Similar techniques are applicable in other combinatorial design problems.

Example

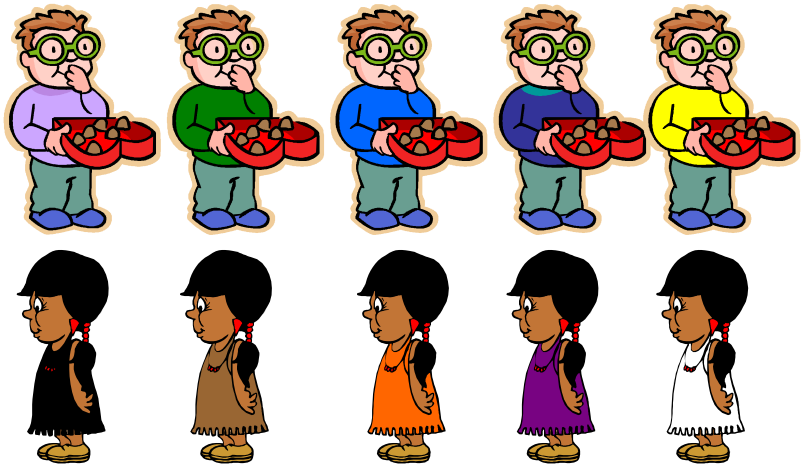
- Department has n courses and m instructors.
- Every instructor has a list of courses he or she can teach.
- Every instructor can teach at most 3 courses during a year.

- The goal: find an allocation of courses to the instructors subject to these constraints.

Applications of Graph Matching

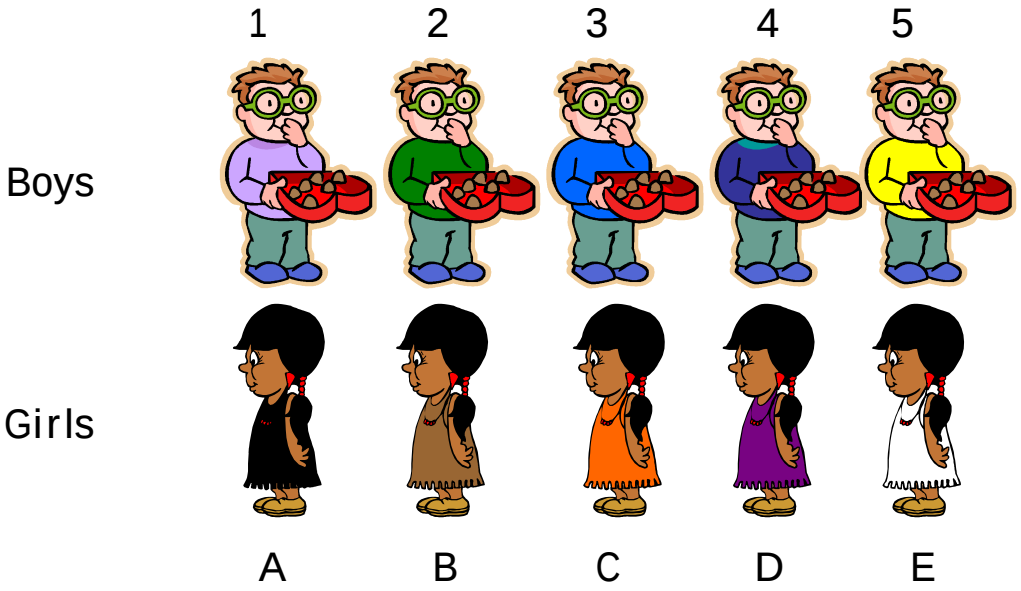


Stable Marriage Problem



What is a Matching?

- Each boy is matched to at most one girl.
- Each girl is matched to at most one boy.



What is a Good Matching?

- A stable matching: They have no incentive to break up.
  (unstable pairs - Willing to break up and switch).
- A maximum matching: To maximize the number of pairs matched.

Stable Matching Problem

- There are n boys and n girls.
- For each boy, there is a preference list of the girls.
- For each girl, there is a preference list of the boys.

Boys

1: CBEAD

2 : ABECD

3 : DCBAE

4 : ACDBE

5 : ABDEC

Girls

A : 35214

B : 52143

C : 43512

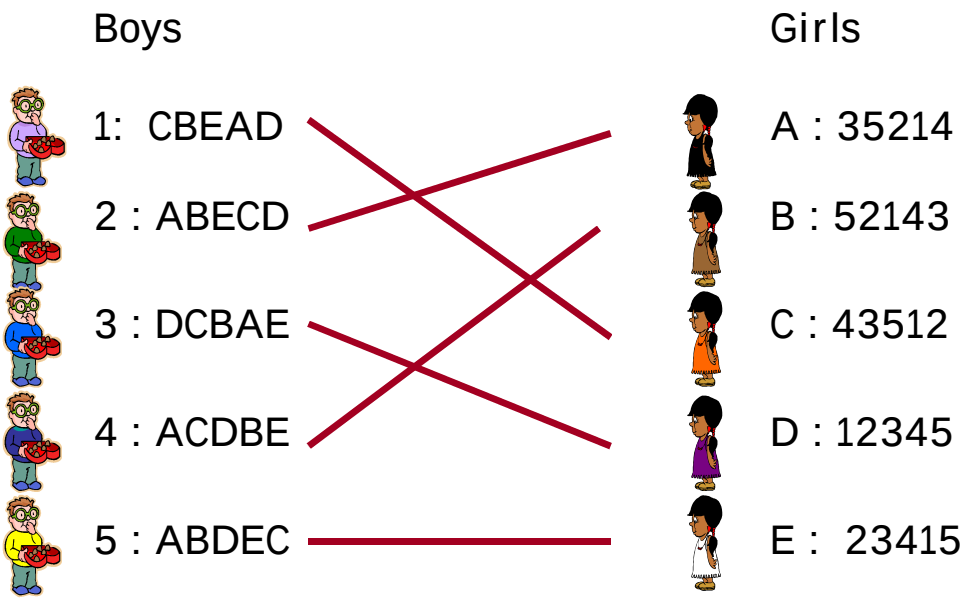
D : 12345

E : 23415

What is a ***stable*** matching?

Consider the following matching.

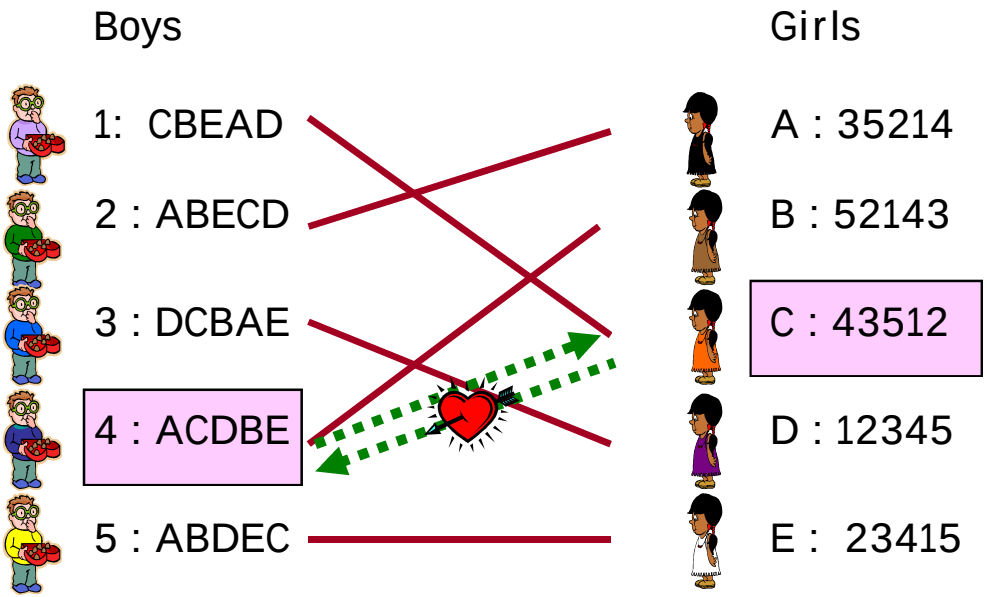
It is unstable, why?



Unstable pair

- Boy 4 prefers girl C more than girl B (his current partner).
- Girl C prefers boy 4 more than boy 1 (her current partner).

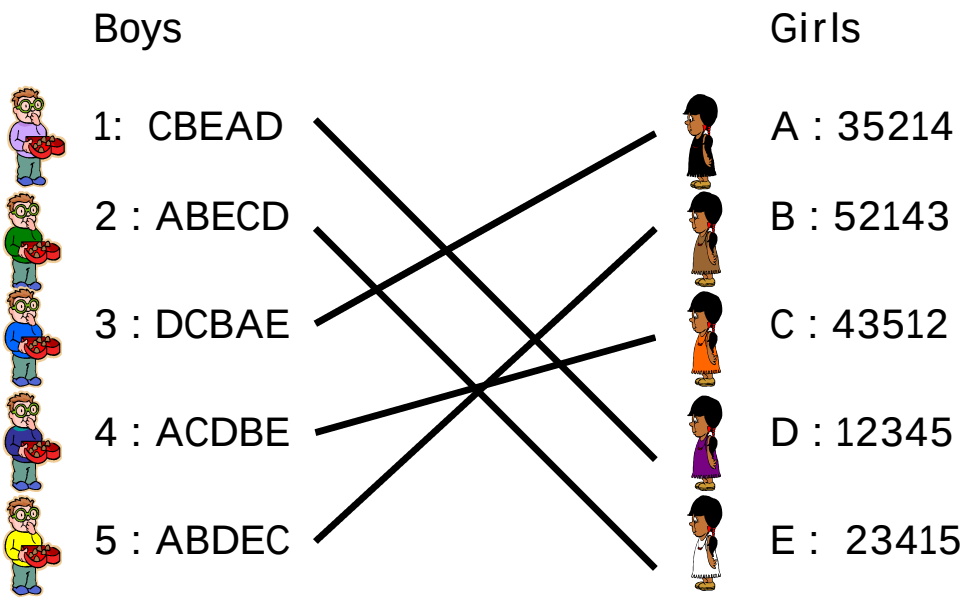
So they have the incentive to leave their current partners, and switch to each other, we call such a pair an unstable pair.



What is a ***stable*** matching?

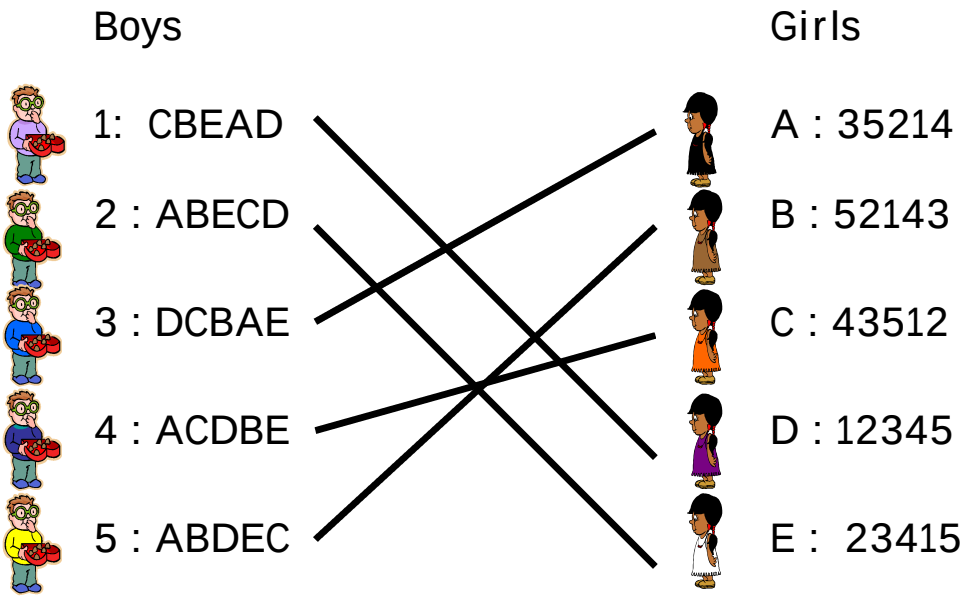
A stable matching is a matching with no unstable pair, and every one is married.

Can you find a stable matching in this case?



Does a stable matching always exists? - Not clear…

Can you find a stable matching in this case?
(continued after Stable Roommate problem)



Stable Marriage, Gale Shapley Algorithm

Stable Matching

Can you now construct an example where there is no stable matching?

not clear...

Gale,Shapley [1962]:

There is always a stable matching in the stable matching problem.

This is more than a solution to a puzzle:

- College Admissions (original Gale & Shapley paper, 1962)
- Matching Hospitals & Residents.
- Matching Dancing Partners.

Shapley received Nobel Prize 2012 in Economics for it!

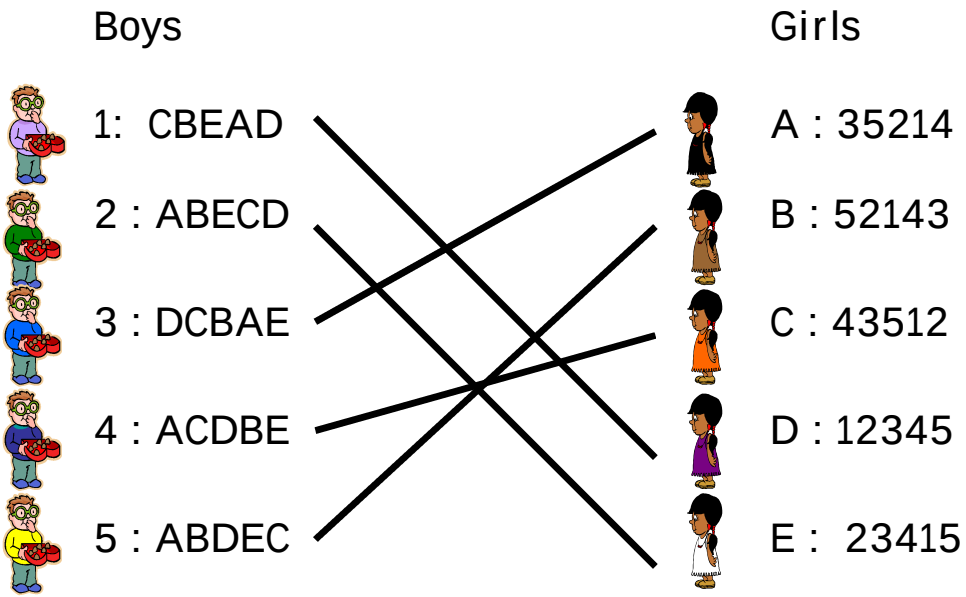
The proof is based on a marriage procedure...

Stable Matching

Why stable matching is easier than stable roommate?

Intuition: It is enough if we only satisfy one side!

This intuition leads us to a very natural approach.



The Marrying Procedure

Morning: boy propose to their favourite girl

Billy Bob

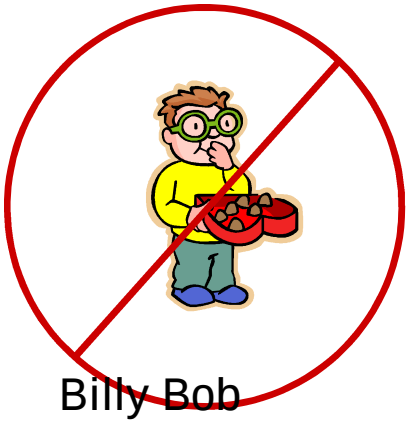
Brad

Angelina

The Marrying Procedure

Morning: boy propose to their favourite girl

Afternoon: girl rejects all but her favourite



Billy Bob

Brad

Angelina

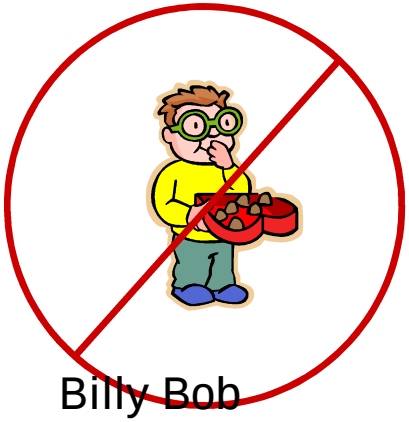
The Marrying Procedure

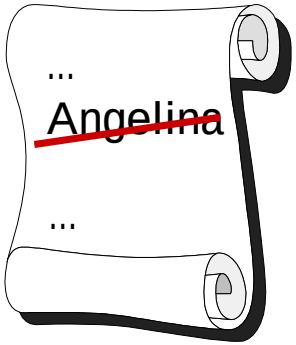
Morning: boy propose to their favourite girl

Afternoon: girl rejects all but her favourite

Evening: rejected boy writes off girl

This procedure is then repeated until all boys propose to a different girl



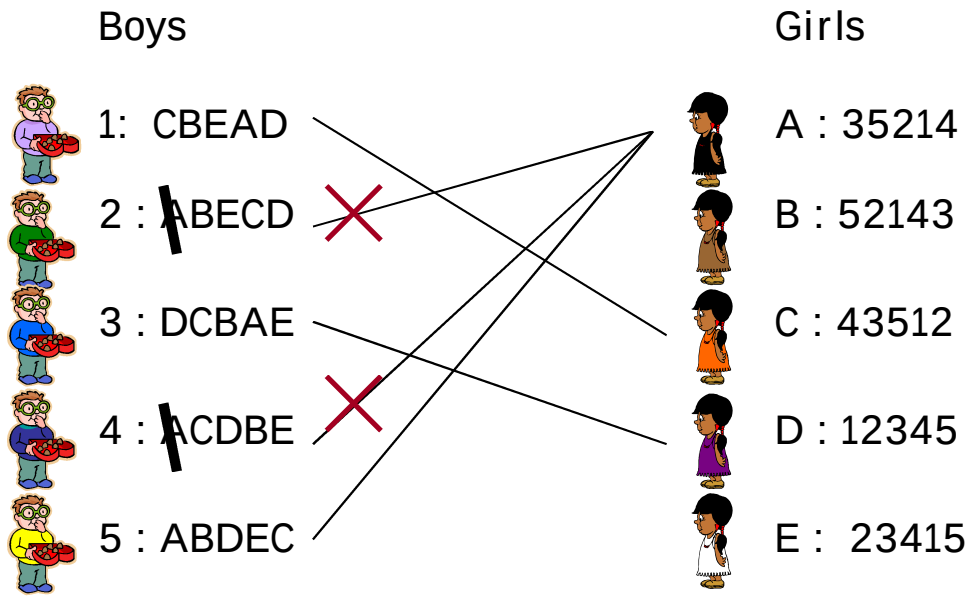


Day 1

Morning: boy propose to their favourite girl

Afternoon: girl rejects all but her favourite

Evening: rejected boy writes off girl

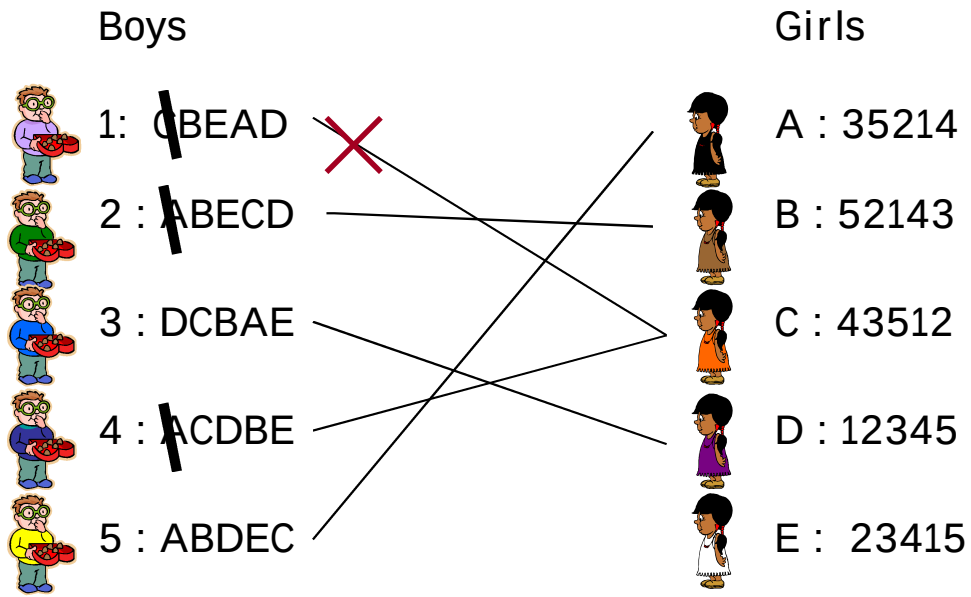


Day 2

Morning: boy propose to their favourite girl

Afternoon: girl rejects all but her favourite

Evening: rejected boy writes off girl

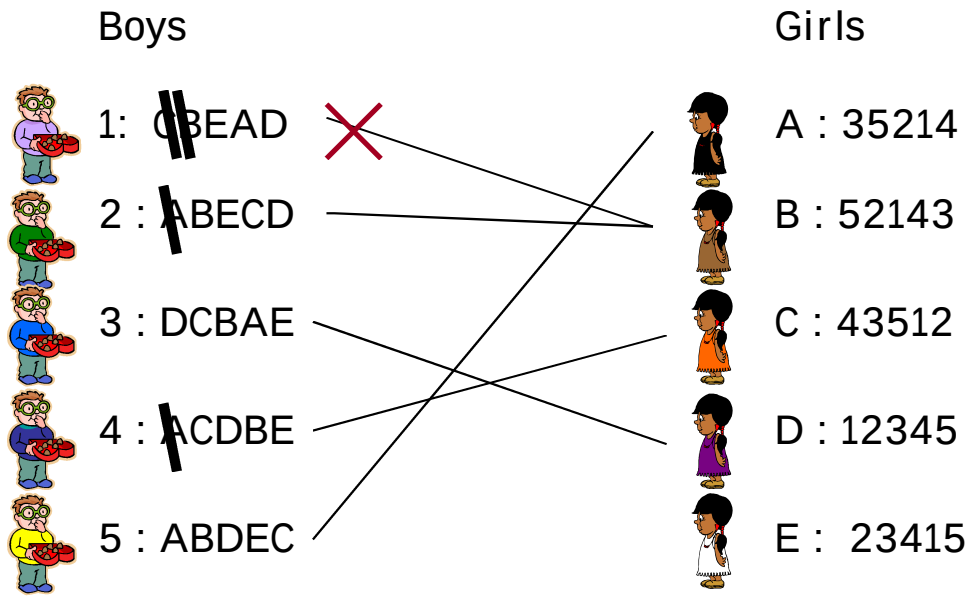


Day 3

Morning: boy propose to their favourite girl

Afternoon: girl rejects all but her favourite

Evening: rejected boy writes off girl



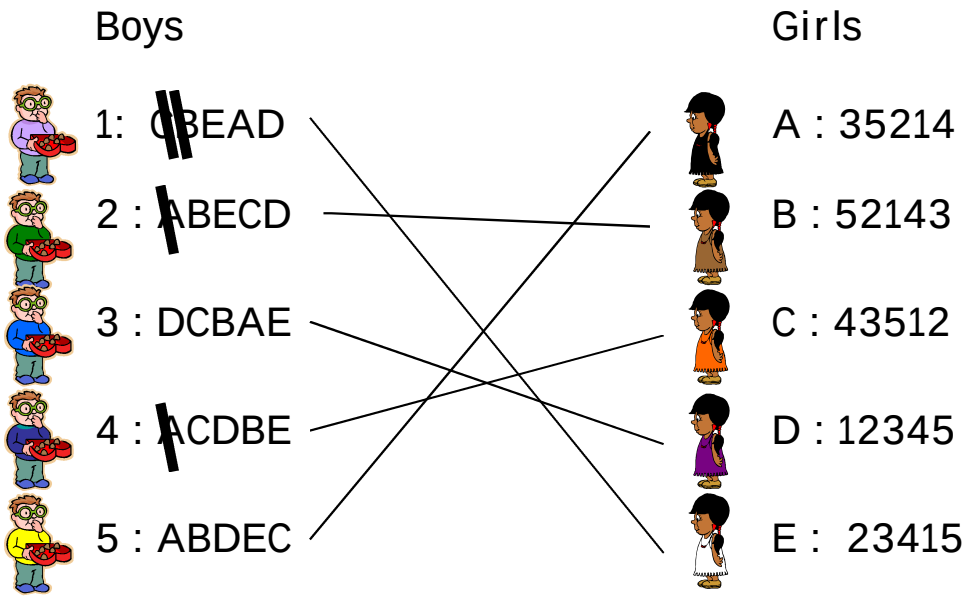
Day 4

Morning: boy propose to their favourite girl

Afternoon: girl rejects all but her favourite

Evening: rejected boy writes off girl

OKAY, marriage day!



Proof of Gale-Shapley Theorem

Gale,Shapley [1962]:
This procedure always find a stable matching in the stable marriage problem.

What do we need to check?

1. The procedure will terminate.
2. Everyone is married.
3. No unstable pairs.

Step 1 of the Proof

Claim 1. The procedure will terminate in at most n^2 days with (n boys and n girls) .

1. If every girl is matched to exactly one boy,
   then the procedure will terminate.

2. Otherwise, since there are n boys and n girls,
   there must be a girl receiving more than one proposal.

3. She will reject at least one boy in this case,
   and those boys will write off that girl from their lists,
   and propose to their next favourite girl.

4. Since there are n boys and each list has at most n girls,
   the procedure will last for at most n^2 days.

Step 2 of the Proof

Claim 2. Every one is married when the procedure stops.

Proof: by contradiction.

1. If *B* is not married, his list is empty.
2. That is, *B* was rejected by all girls.
3. A girl only rejects a boy if she already has a more preferable partner.
4. Once a girl has a partner, she will be married at the end.
5. That is, all girls are married (to one boy) at the end, but *B* is not married.
6. This implies there are more boys than girls, a contradiction.

Step 3 of the Proof

Claim 3. There is no unstable pair.

Fact. If a girl G rejects a boy B,
then G will be married to a boy (she likes) better than B.

Consider any pair (B,G).

Case 1. If G is on B's list, then B is married to be the best one on his list.
So B has no incentive to leave.

Case 2. If G is not on B's list, then G is married to a boy she likes better.
So G has no incentive to leave.

Proof of Gale-Shapley Theorem

Gale,Shapley [1962]:

There is always a stable matching in the stable marriage problem.

Claim 1. The procedure will terminate in at most n^2 days.

Claim 2. Every one is married when the procedure stops.

Claim 3. There is no unstable pair.

So the theorem follows.

Optional Questions

Intuition: It is enough if we only satisfy one side!

Is this marrying procedure better for boys or for girls??

- All boys get the best partner simultaneously!
- All girls get the worst partner simultaneously!

Why?

That is, among all possible stable matching,
boys get the best possible partners simultaneously.

Can a boy do better by lying? NO!

Can a girl do better by lying? YES!

Stable Roommate:

matching pairs in graphs

Stable Roommate Problem

- There are 2n people.
- There are n rooms, each can accommodate 2 people.
- Each person has a preference list of 2n-1 people.
- Find a stable matching (match everyone and no unstable pair).

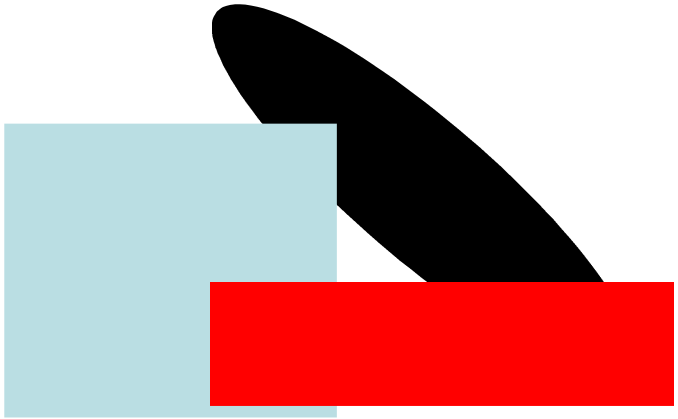
Does a stable matching always exist?
Not clear?

When is it difficult to find a stable matching?

Stable Roommate: No matching if there is a circular dependency

	1	2	3
a	b	c	d
b	c	a	d

- a prefers b more than c
- b prefers c more than a
- c prefers a more than b



Let's match (a & b) and (c & d), Then (b&c) is unstable, because b prefers c more than a, and c prefers b more than d.

Maxflow:

Maximum Network Flow

Network Flows

Important problem in the *optimal management* of *resources*.

Types of Networks

- Internet
- Telephone
- Cell
- Roads, Highways
- Railways
- Electrical Power
- Water
- Sewer
- Gas
- Any more?

Maximum Flow Problem

- How can we maximize the flow in a network from a source or set of sources to a destination or set of destinations?
- The problem reportedly rose to prominence in relation to the rail networks of the Soviet Union, during the 1950's.
- The US wanted to know how quickly the Soviet Union could get supplies through its rail network to its satellite states in Eastern Europe.

Source: Ibackstrom, The Importance of Algorithms, at www.topcoder.com

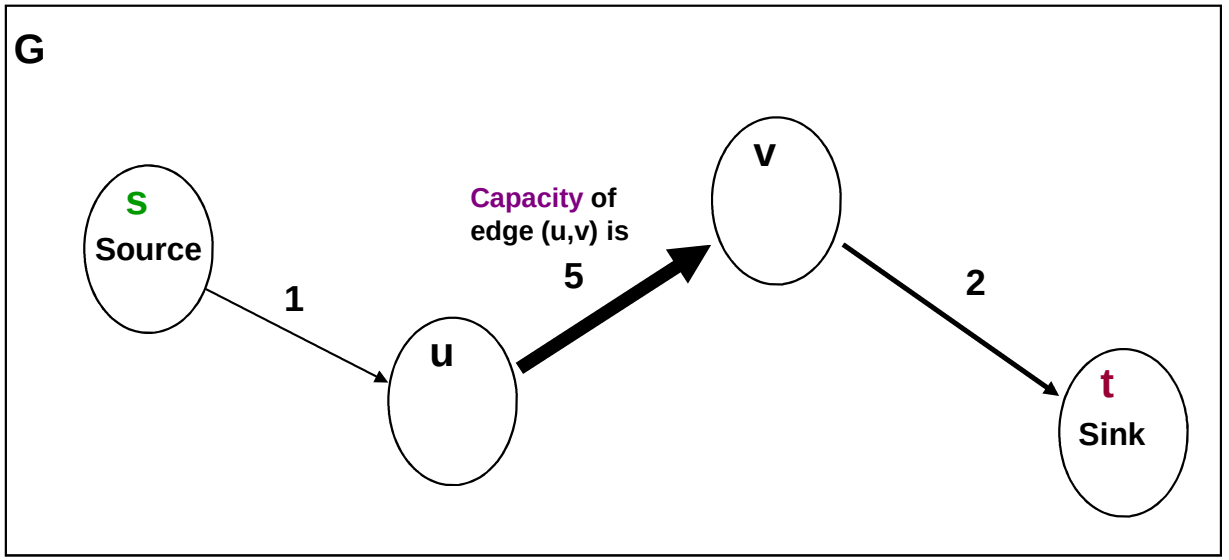
Dual of Max Flow is Min cut

- In addition, the US wanted to know which rails it could destroy most easily to cut off the satellite states from the rest of the Soviet Union.
- It turned out that these two problems were closely related, and that solving the **max flow problem** also solves the **min cut problem** of figuring out the cheapest way to cut off the Soviet Union from its neighbours.

Source: Ibackstrom, The Importance of Algorithms, at www.topcoder.com

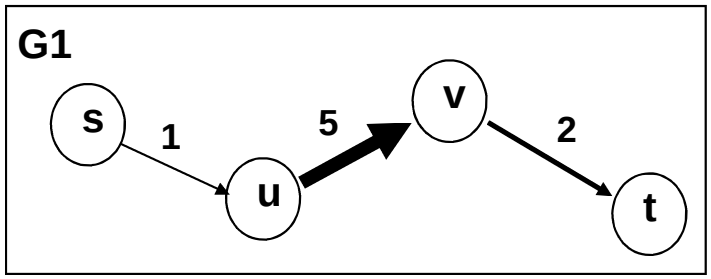
Network Flow definitions

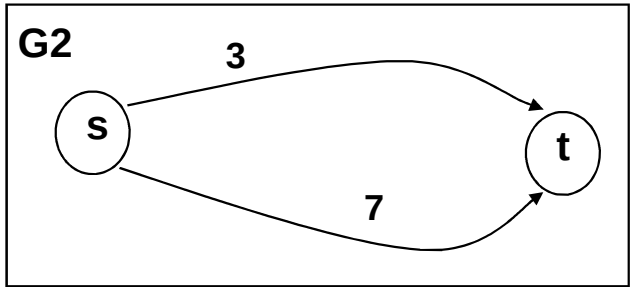
- Instance:
  - A Network is a *directed* graph G
  - Edges represent pipes that carry flow
  - Each edge (u,v) has a maximum capacity c(u,v)
  - A source node **s** from which flow arrives
  - A sink node **t** out of which flow leaves
- Example:

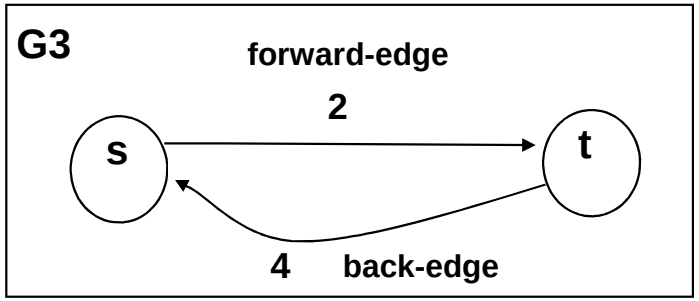


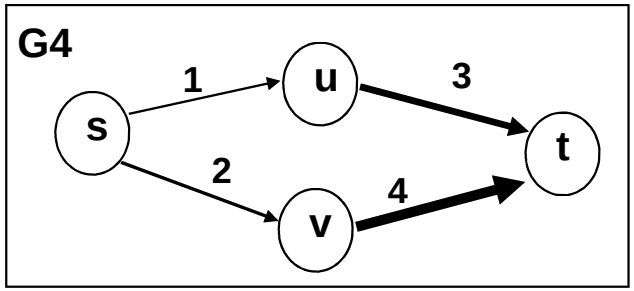
Network Flow Problems

What is the max flow in these Graphs?



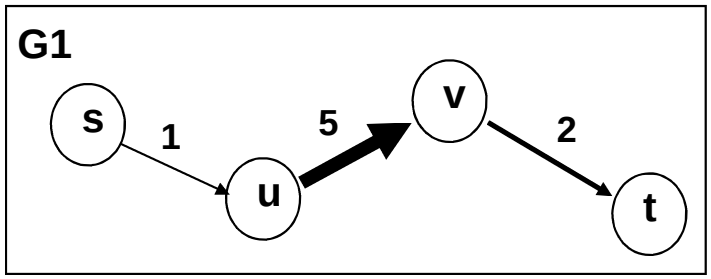




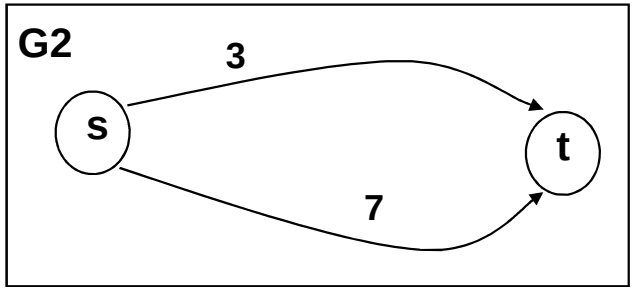


Network Flow Solutions

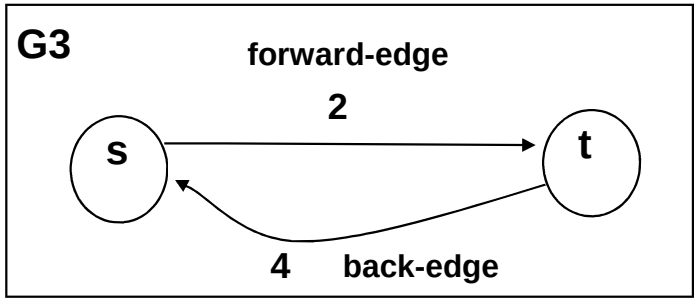
What is the max flow in these Graphs?



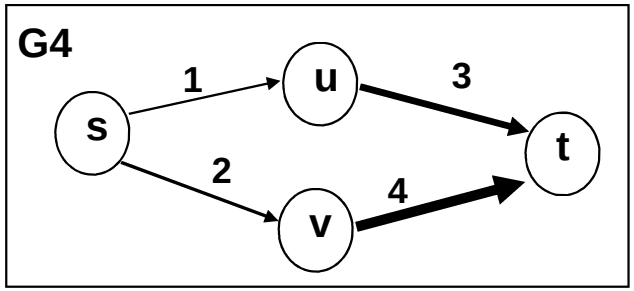
G1. maxflow =max(1,5,2)=1



G2. maxflow = 3 + 7 = 10



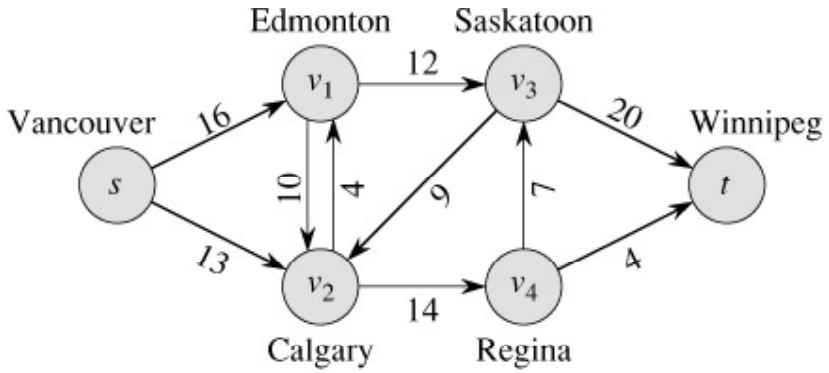
G3. maxflow = 2



G4. maxflow =
max(1,3)+max(2,4) = 1+2 = 3

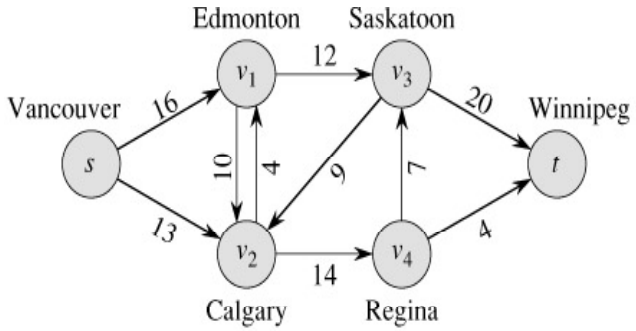
The General Maxflow Problem

- Use a graph to model material that flows through pipes.
- Each edge represents one pipe, and has a capacity, which is an upper bound on the flow rate, in some units / time.
- Think of edges as pipes of different sizes.
- Want to compute max rate that we can ship material from a given source to a given sink.
- Example:



What is a Flow Network?

- Each edge (u,v) has a nonnegative capacity c(u,v).
- If (u,v) is not in E, let c(u,v)=0.
- We have a source s, and a sink t.
- Assume that every vertex v in V is on some path from s to t.
- Here $c(s,v_1)=16$; $c(v_1,s)=0$; $c(v_2,v_3)=0$
- We can write it as a matrix



Constraints on Flow

- For each edge (u,v), the flow f(u,v) is a real-valued function that must satisfy 3 conditions:

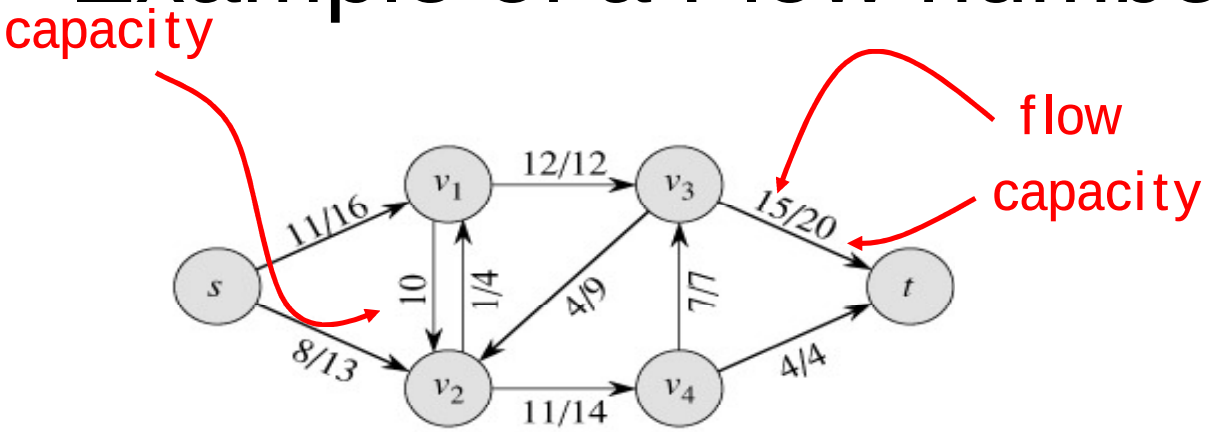
  Capacity constraint: $\forall u,v \in V,\ f(u,v) \le c(u,v)$

  Skew symmetry: $\forall u,v \in V,\ f(u,v) = -f(v,u)$

  Flow conservation: $\forall u \in V - \{s,t\},\ \sum_{v \in V} f(u,v) = 0$

  - Notes:
    - The skew symmetry condition implies that f(u,u)=0.
    - We show only the *positive* flows in the flow network.

Example of a Flow numbering:



- $f(v_2, v_1) = 1$ of available $c(v_2, v_1) = 4$.
- $f(v_1, v_2) = -1$ of available $c(v_1, v_2) = 10$.
- $f(v_3, s) + f(v_3, v_1) + f(v_3, v_2) + f(v_3, v_4) + f(v_3, t) = 0 + (-12) + 4 + (-7) + 15 = 0$.

The Value of a flow

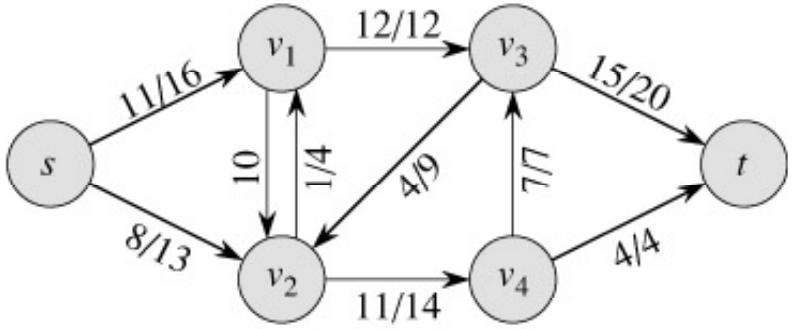
- The value of a flow is given by

$$| f | = \sum_{v \in V} f(s, v) = \sum_{v \in V} f(v, t)$$

This is the total flow leaving s

= the total flow arriving in t.

Example of flow at s and t



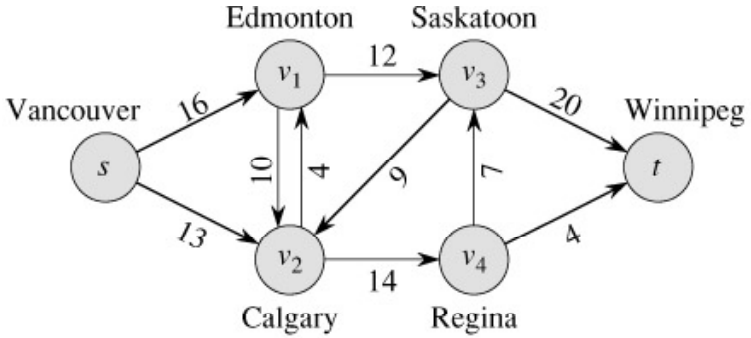
$|f| = f(s, v_1) + f(s, v_2) + f(s, v_3) + f(s, v_4) + f(s, t)$

$= 11 + 8 + 0 + 0 + 0 = 19$

$|f| = f(s, t) + f(v_1, t) + f(v_2, t) + f(v_3, t) + f(v_4, t)$

$= 0 + 0 + 0 + 15 + 4 = 19$

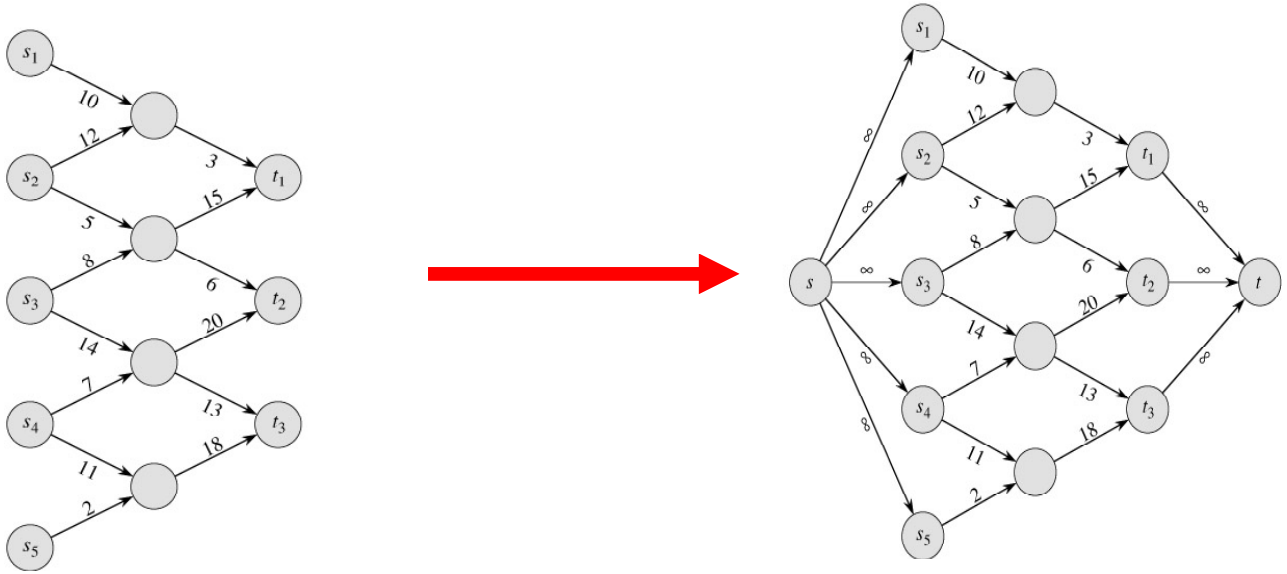
A flow in a network



- We assume that there is only flow in one direction at a time.
- Sending 7 trucks from Edmonton to Calgary and 3 trucks from Calgary to Edmonton has the same net effect as sending 4 trucks from Edmonton to Calgary.

Multiple Sources Network

- We have several sources and several targets.
- Reduce to max-flow by adding a supersource and a supersink:



Residual Networks (available capacity)

- The residual capacity of an edge *(u,v)* in a network with a flow *f* is given by:

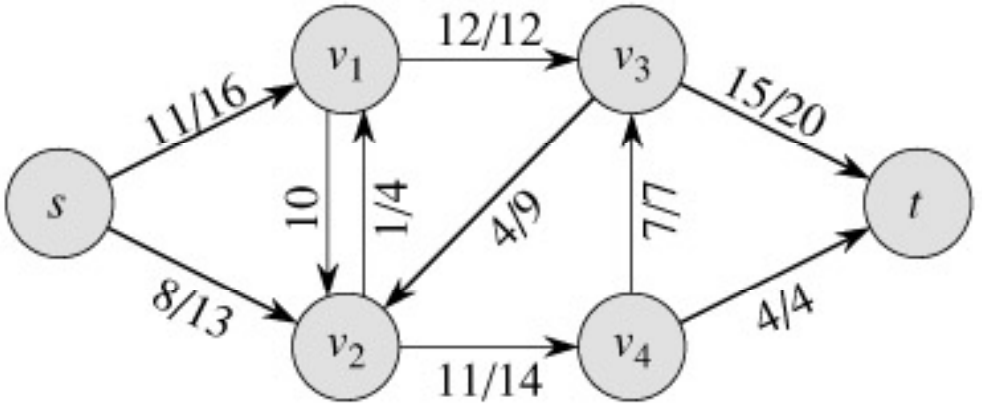
$$c_f(u,v) = c(u,v) - f(u,v)$$

- The residual network of a graph *G* induced by a flow *f* is the graph with only the edges having positive residual capacity, i.e.,

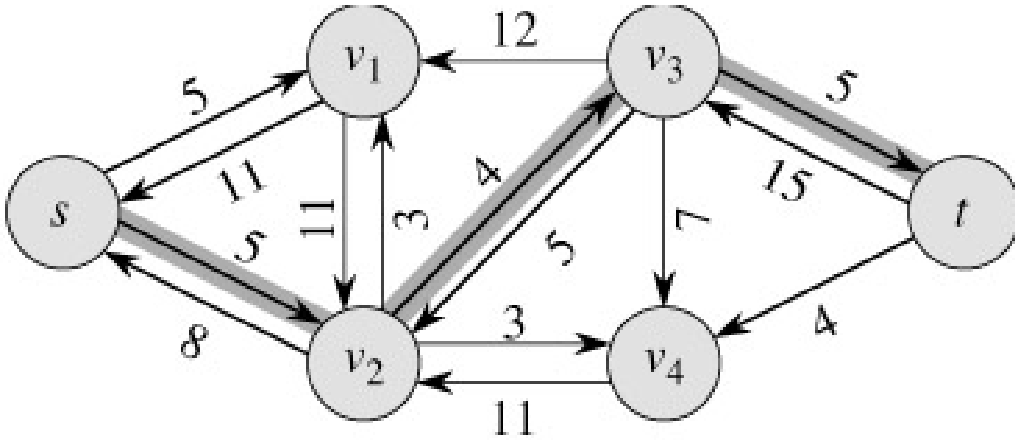
$G_f = (V, E_f)$, where $E_f = \{(u,v) \in V \times V : c_f(u,v) > 0\}$

Example of Residual Network

Flow Network:



Residual Network:



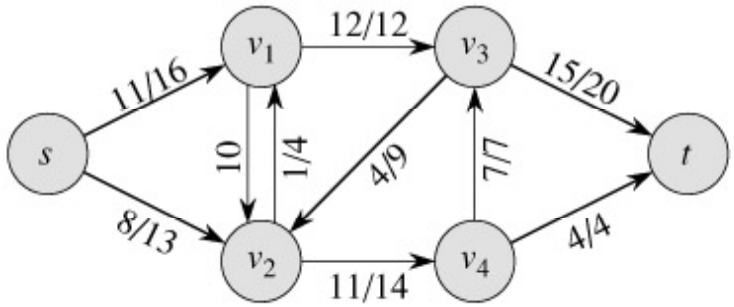
Augmenting Paths

- An augmenting path *p* is a simple path from s to t on the residual network.
- We can put more flow from *s* to *t* through *p*.
- We call the maximum capacity by which we can increase the flow on *p* the residual capacity of *p*.

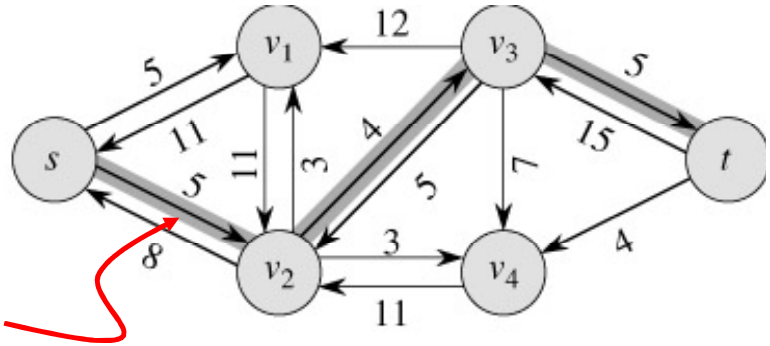
$$c_f(p) = \min\{c_f(u,v) : (u,v) \text{ is on } p\}$$

Augmenting Paths

Network:



Residual Network:



Augmenting path

The residual capacity of this augmenting path is 4.

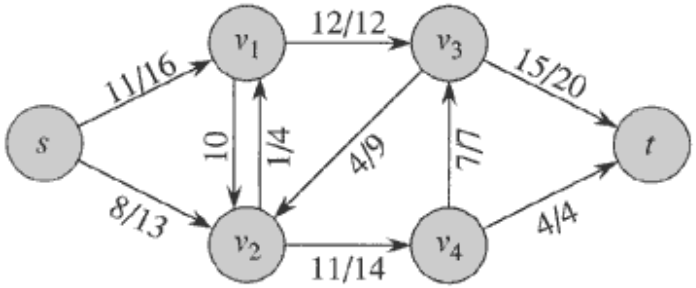
Computing Max Flow

- Classic Method:
  - Identify an augmenting path
  - Increase flow along that path
  - Repeat until no more paths

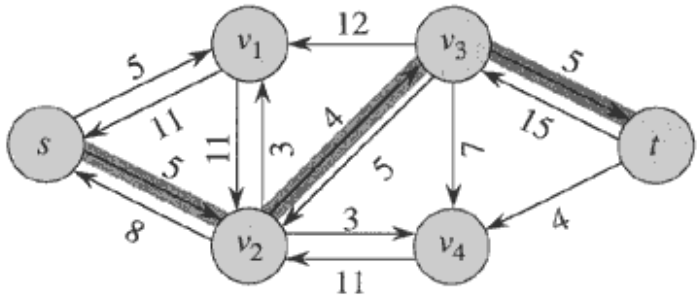
Ford-Fulkerson Method

```
Ford FulkersonMethod(G,s,t) {
  f = 0 // initial flow in G, from s to t.
  while ( there exists an augmenting path p
              with spare capacity h ) {
        f = f + h // augment flow along p
     }
  // No more paths, f is maxflow in G.
  return f
}
```

Example

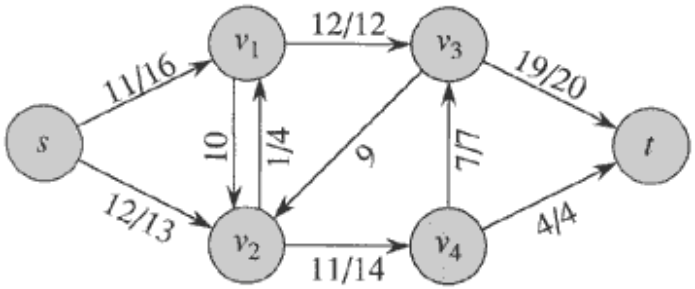


Flow(1)

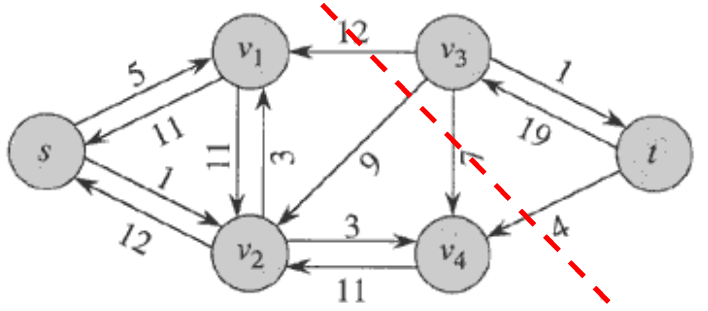


Residual(1)

No more augmenting paths → max flow attained.



Flow(2)

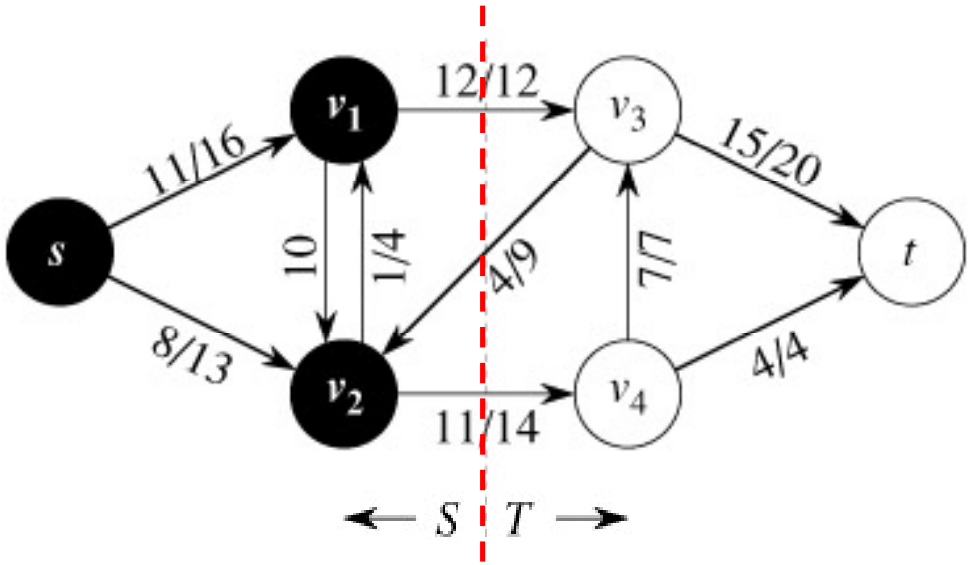


Residual(2)

Cut

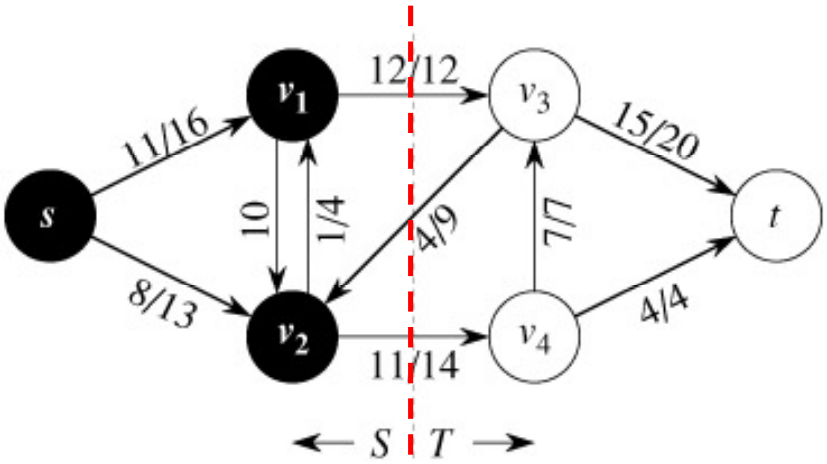
Cuts of Flow Networks

A cut (S,T) of a flow network is a partition of V into S and $T = V - S$ such that $s \in S$ and $t \in T$.



The Net Flow through a Cut (S,T)

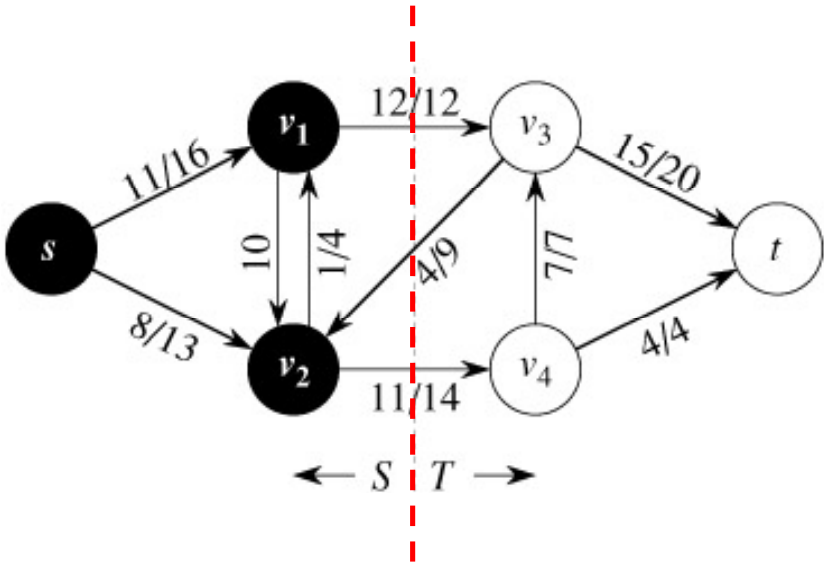
$$f(S,T) = \sum_{u \in S, v \in T} f(u,v)$$



- f(S,T) = 12 – 4 + 11 = 19

The Capacity of a Cut (S,T)

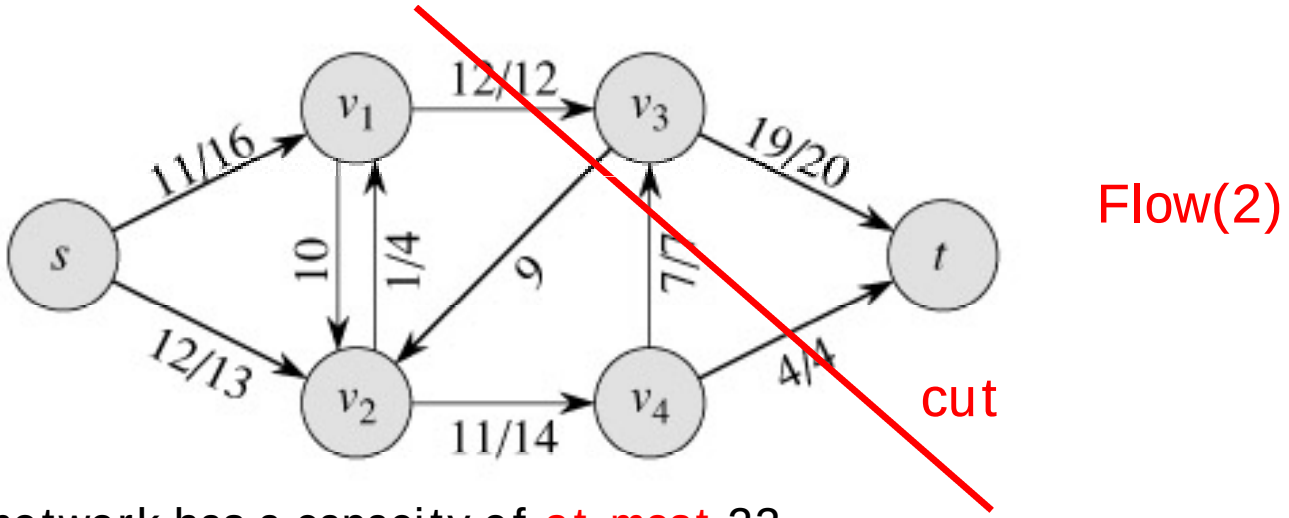
$$c(S,T) = \sum_{u \in S, v \in T} c(u,v)$$



- c(S,T)= 12+ 0 + 14 = 26

Augmenting Paths – example

Capacity of the cut
= maximum possible flow through the cut
= 12 + 7 + 4 = 23

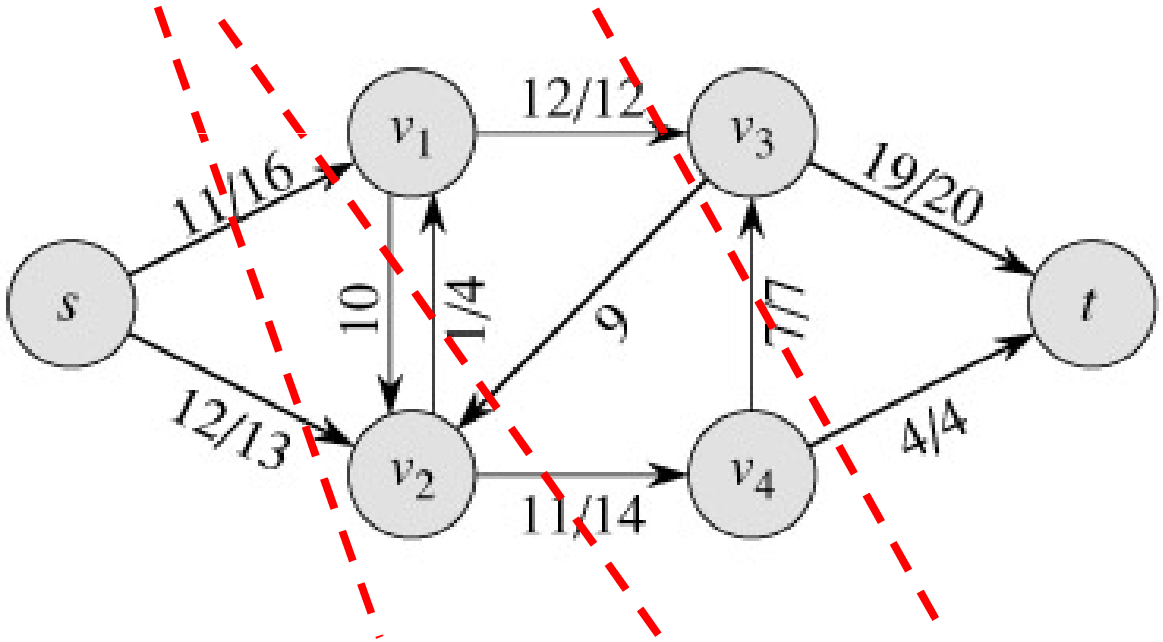


Flow(2)

- The network has a capacity of at most 23.
- In this case, the network does have a capacity of 23, because this is a minimum cut.

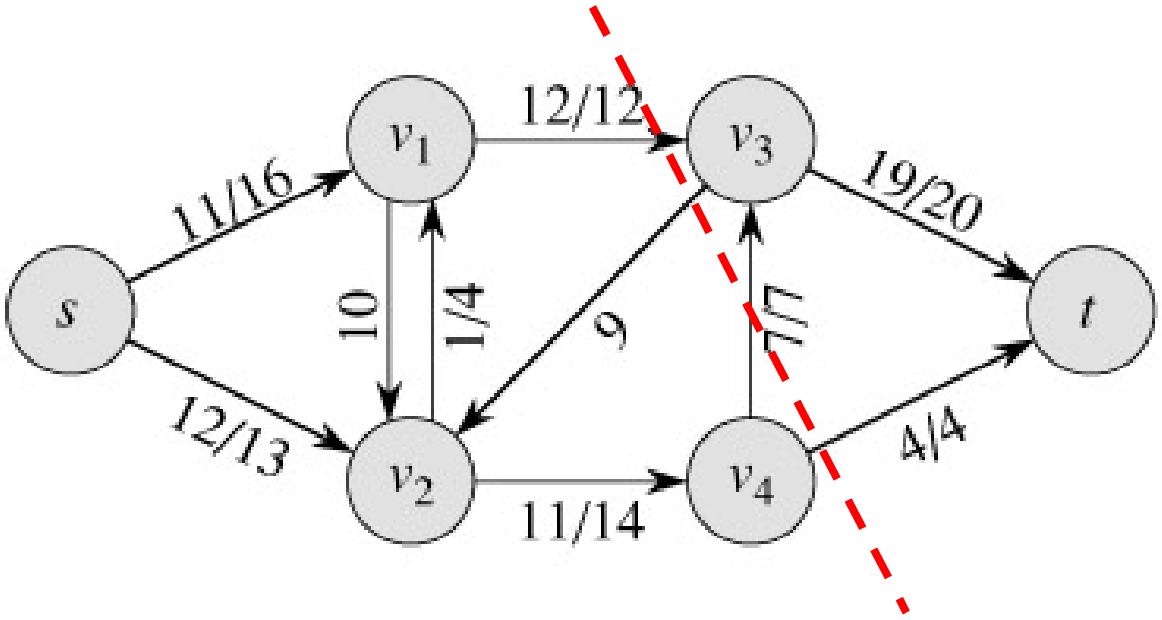
Net Flow of a Network

- The net flow across any cut is the same and equal to the flow of the network |f|.

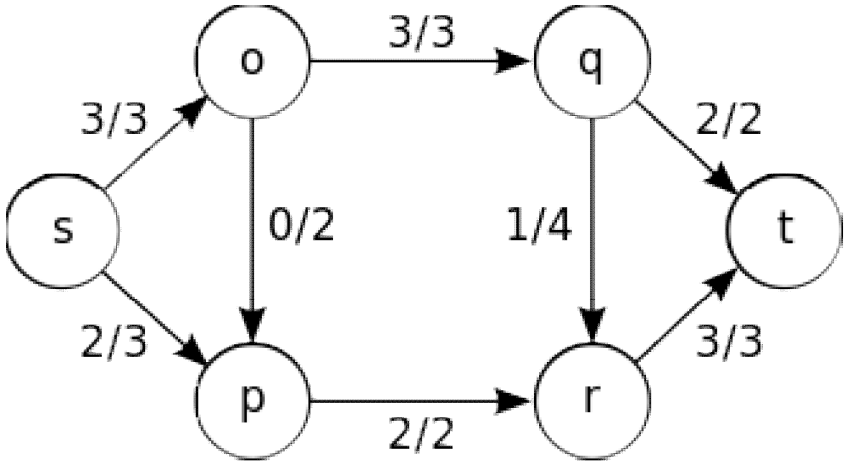


Bounding the Network Flow

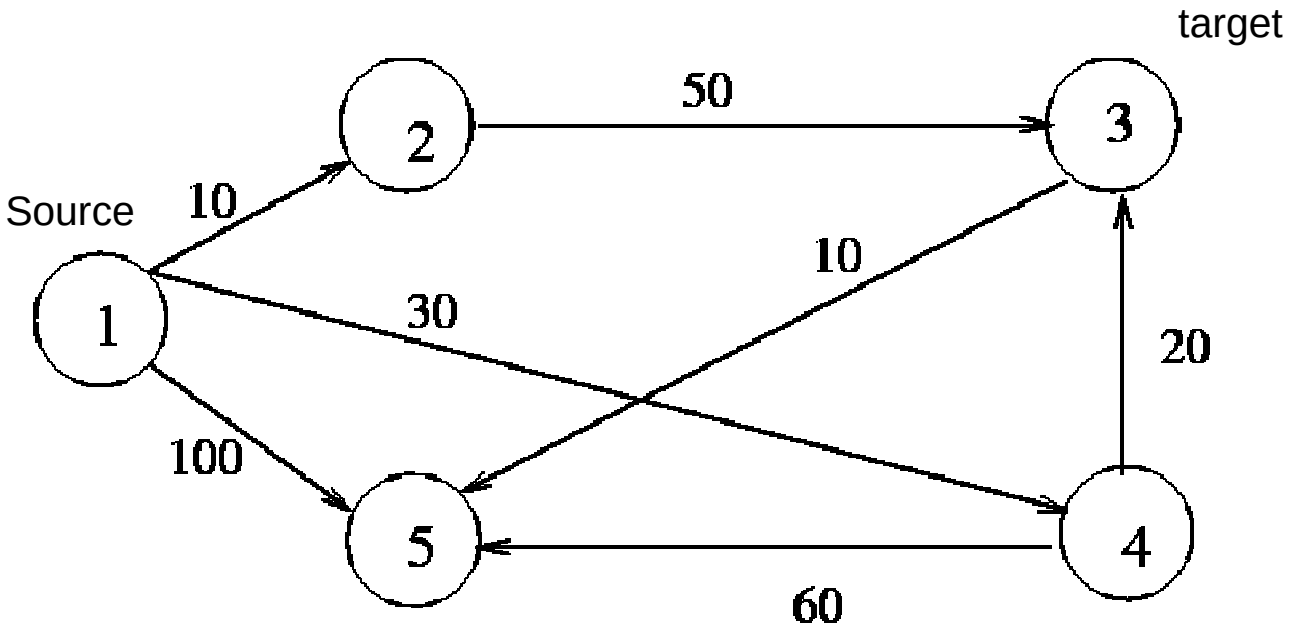
- The value of any flow f in a flow network G is bounded from above by the capacity of any cut of G.



Exercise: list all the cuts and find the min-cut



Exercise: Find maxflow from source to target, and show the mincut



Max-Flow Min-Cut Theorem

- If f is a flow in a flow network $G=(V,E)$, with source s and sink t, then the following conditions are equivalent:
  1. f is a maximum flow in G.
  2. The residual network G_f contains no augmented paths.
  3. $|f| = c(S,T)$ for some cut (S,T) (a min-cut).

Ford-Fulkerson for max flow

Ford-Fulkerson(*G*)

  $f = \mathbf{0}$

  while(∃ simple path p from s to t in G_f)

    $f := f + f_p$

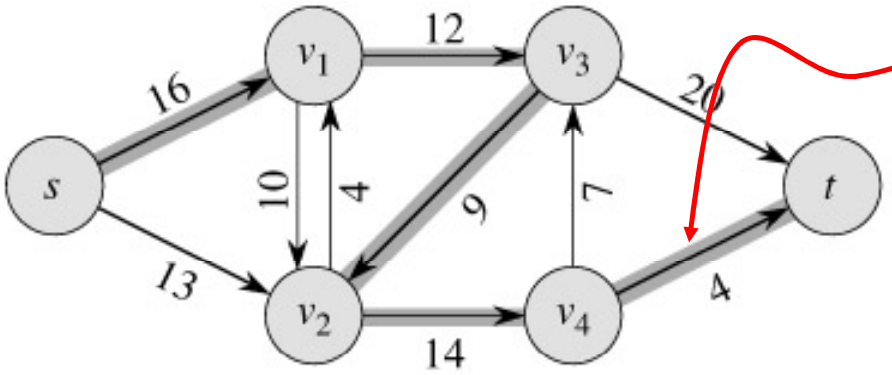
  output f

The Basic Ford-Fulkerson Algorithm

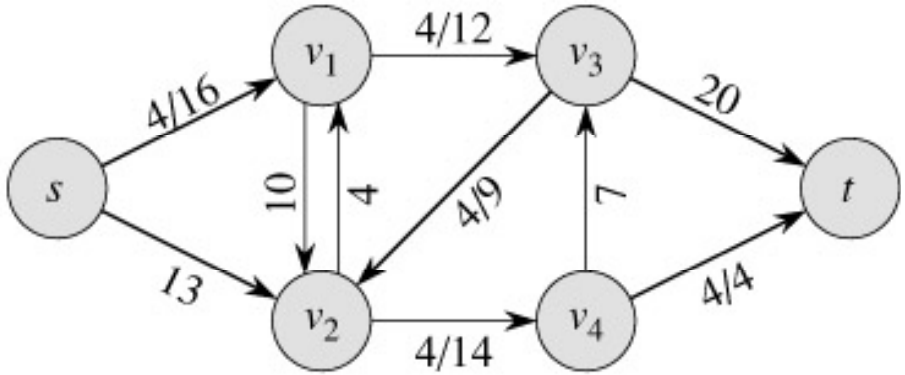
FORD-FULKERSON(G, s, t)

1 **for** each edge $(u, v) \in E[G]$
2 **do** $f[u, v] \leftarrow 0$
3 $f[v, u] \leftarrow 0$
4 **while** there exists a path p from s to t in the residual network G_f
5 **do** $c_f(p) \leftarrow \min\{c_f(u, v) : (u, v) \text{ is in } p\}$
6 **for** each edge (u, v) in p
7 **do** $f[u, v] \leftarrow f[u, v] + c_f(p)$
8 $f[v, u] \leftarrow -f[u, v]$

Example



Original Network

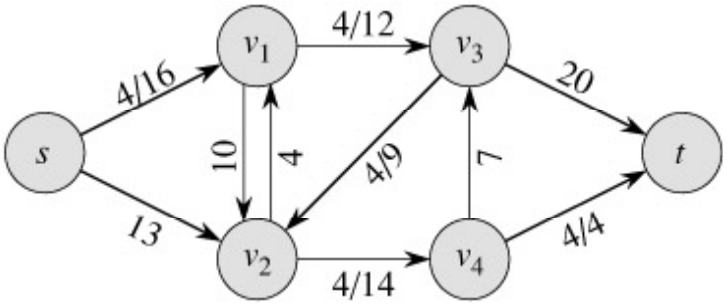


Flow Network

Resulting Flow = 4

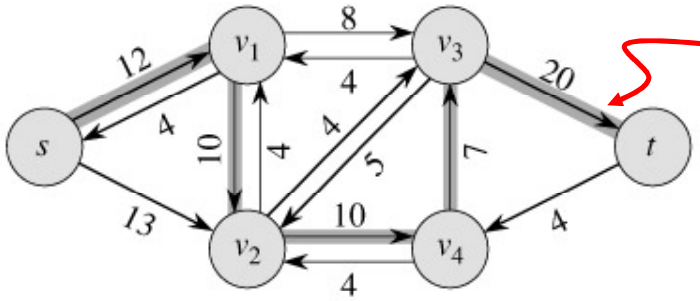
Example

Flow Network



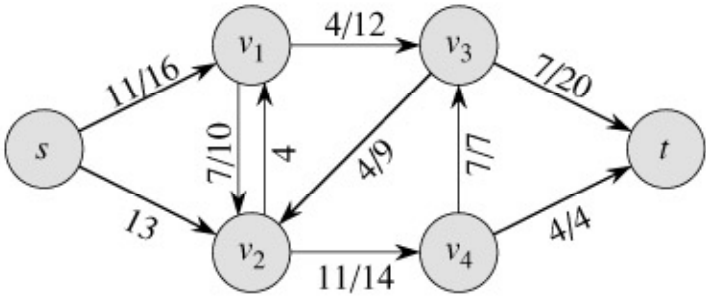
Resulting Flow = 4

Residual Network



augmenting path

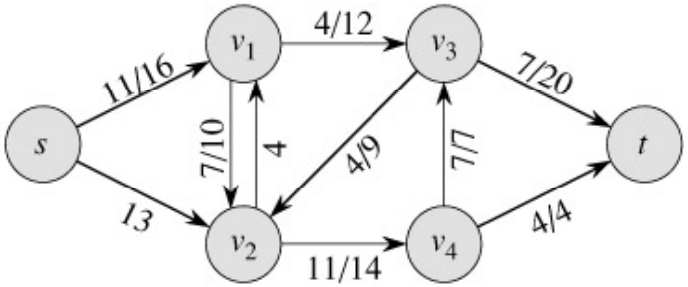
Flow Network



Resulting Flow = 11

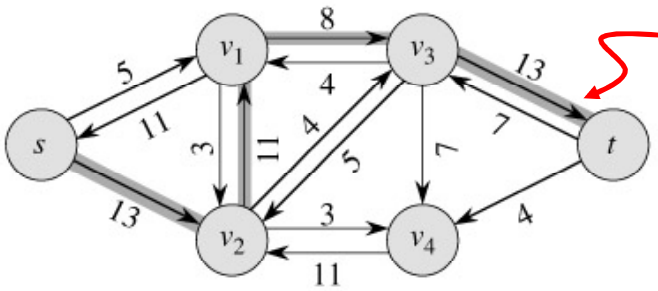
Example

Flow Network



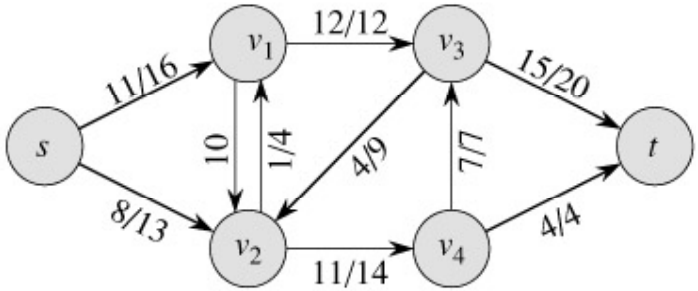
Resulting Flow = 11

Residual Network



augmenting path

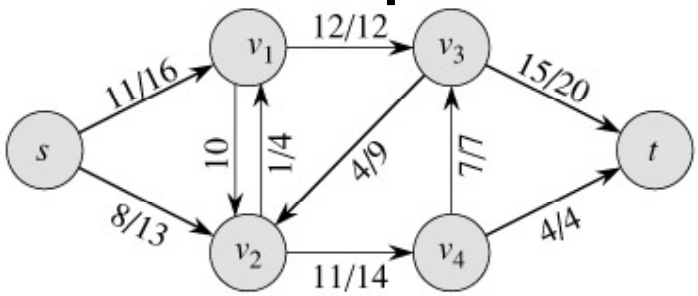
Flow Network



Resulting Flow = 19

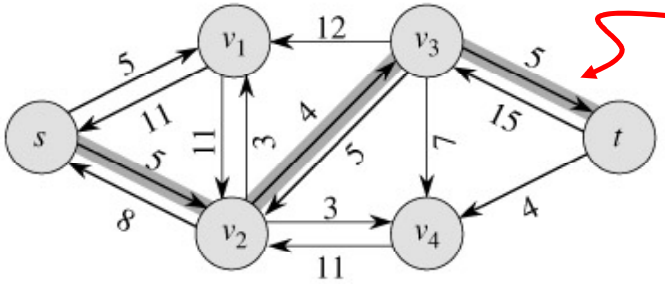
Example

Flow Network



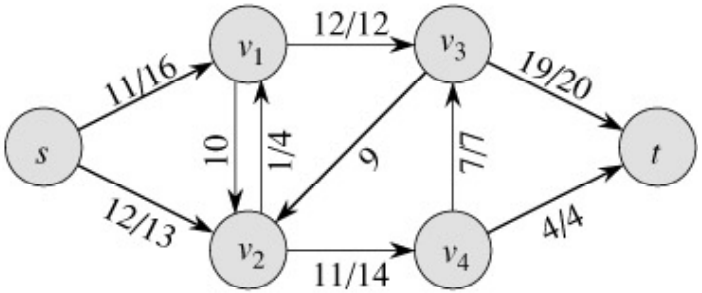
Resulting Flow = 19

Residual Network



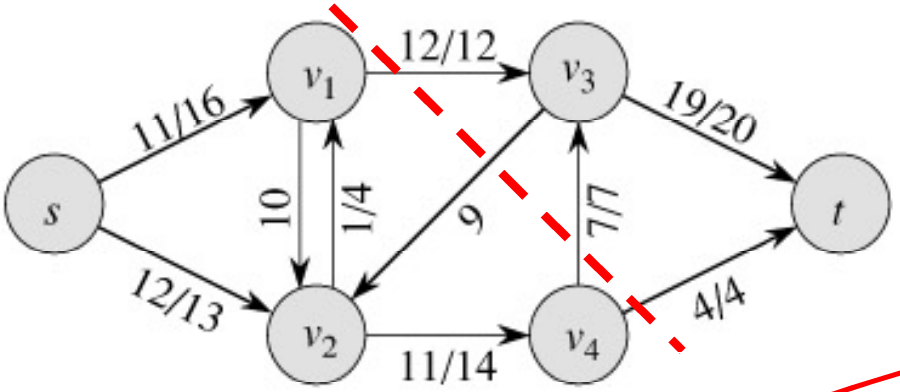
augmenting path

Flow Network



Resulting Flow = 23

Example

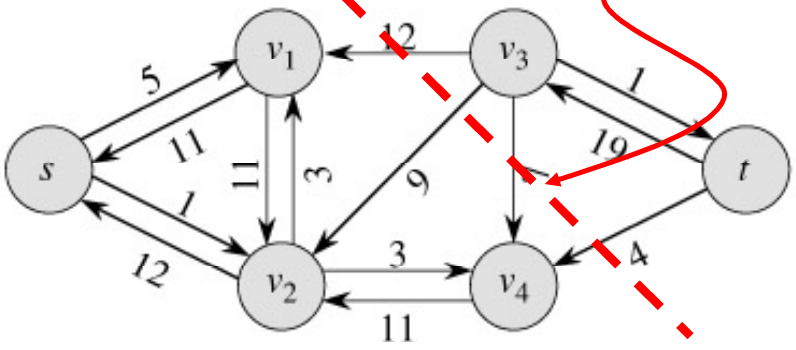


Resulting
Flow = 23

No augmenting path:

Maxflow=23

(e)



Residual Network

Analysis

```
FORD-FULKERSON(G, s, t)
1  for each edge (u, v) ∈ E[G]                                   } O(E)
2      do f[u, v] ← 0
3         f[v, u] ← 0                                                        ?
4  while there exists a path p from s to t in the residual network G_f
5      do c_f(p) ← min{c_f(u, v) : (u, v) is in p}               } O(E)
6         for each edge (u, v) in p
7             do f[u, v] ← f[u, v] + c_f(p)
8                f[v, u] ← −f[u, v]
```

Analysis

- If capacities are all integer, then each augmenting path raises $|f|$ by ≥ 1.
- If max flow is f^*, then need $\leq |f^*|$ iterations → time is $O(E|f^*|)$.
- Note that this running time is not polynomial in input size. It depends on $|f^*|$, which is not a function of $|V|$ or $|E|$.
- If capacities are rational, can scale them to integers.
- If irrational, FORD-FULKERSON might never terminate! But we can't use irrational capacities as inputs to digital computers anyway.

Polynomial time algorithms

- **Defintion:** A ***polynomial time algorithm*** is an algorithm than runs in time polynomial in *n*, where *n* is the ***number of bits*** of the input.

- How we intend to encode the input influences if we have a polynomial algorithm or not. Usually, some “standard encoding” is implied.

- In general, we want Polynomial ¼ Fast
  because Exponential ¼ Slow

Complexity of Ford-Fulkerson

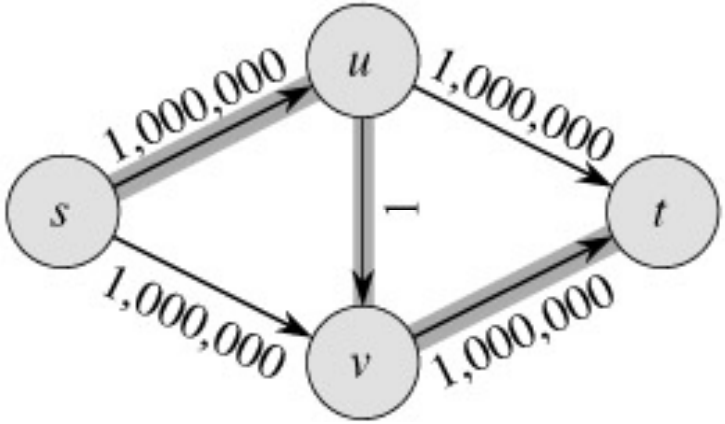
- With standard (decimal or binary) representation of integers, Ford-Fulkerson is an ***exponential*** time algorithm (in size of capacities).
- Example: a capacity 999 can be written in as 3 digits, but it may take 999 iterations.

Complexity of Ford-Fulkerson

- With *unary* (4 ~ 1111) representation of integers, Ford-Fulkerson is a *polynomial* time algorithm.
- Intuition: When the input is longer it is easier to be polynomial time as a function of the input length.
- An algorithm which is polynomial if integer inputs are represented in unary is called a *pseudo-polynomial* algorithm.
- Intuitively, a pseudo-polynomial algorithm is an algorithm which is fast if all numbers in the input are small.

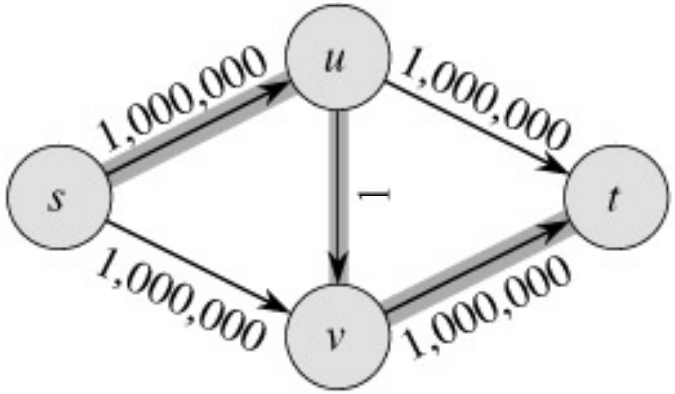
The Basic Ford-Fulkerson Algorithm

- With time O (E |f*|), the algorithm is **not** polynomial.
- This problem is real: Ford-Fulkerson may perform badly if we are unlucky:

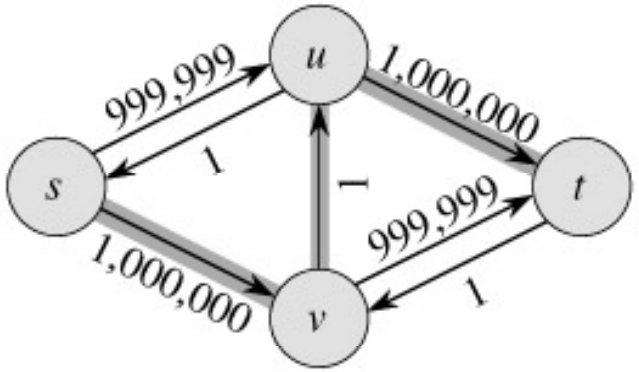


|f*|=2,000,000

Run Ford-Fulkerson on this example

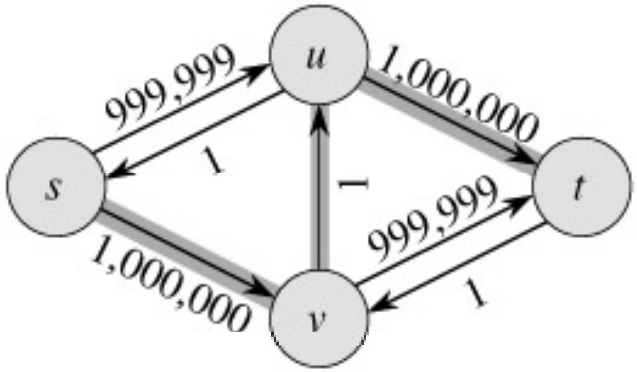


Augmenting Path

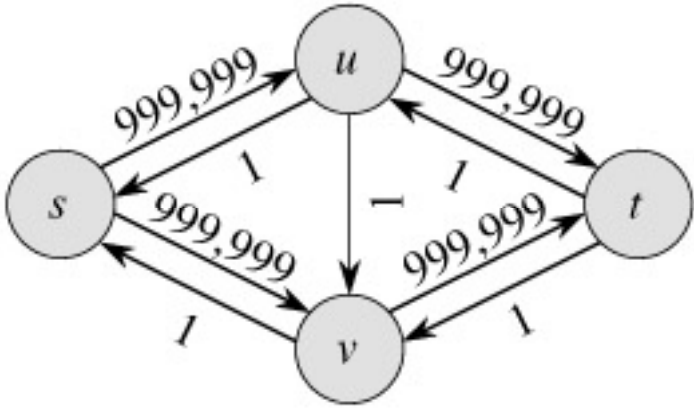


Residual Network

Run Ford-Fulkerson on this example

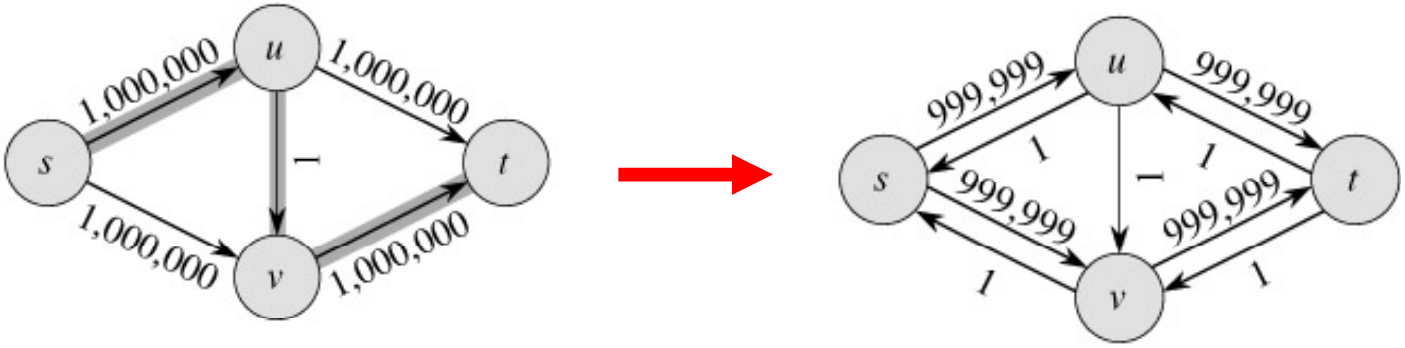


Augmenting Path



Residual Network

Run Ford-Fulkerson on this example



- Repeat 999,999 more times…
- Can we do better than this?

Edmonds-Karp max flow algoritm

Implement Ford-Fulkerson by always choosing the ***shortest possible*** augmenting path, i.e., the one with fewest possible edges.

The Edmonds-Karp Algorithm

- A small fix to the Ford-Fulkerson algorithm makes it work in polynomial time.
- Select the augmenting path using **breadth-first search** on residual network.
- The augmenting path *p* is the shortest path from *s* to *t* in the residual network (treating all edge weights as 1).
- Runs in time $O(V E^2)$.

Complexity of Edmonds-Karp

- Each iteration of the while loop can still be done in time $O(|E|)$.
- The number of iterations are now at most $O(|V||E|)$ regardless of capacities – to be seen next.
- Thus, the total running time is $O(|V| |E|^2)$ and Edmonds-Karp is a polynomial time algorithm for Max Flow.

Why at most O(|V| |E|) iterations?

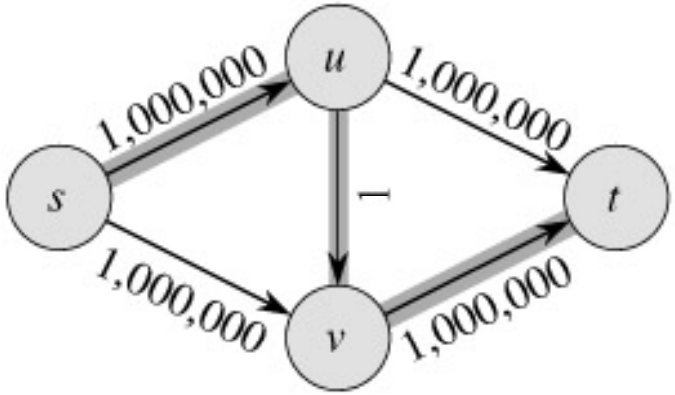
When executing Edmonds-Karp, the residual network G_f gradually changes (as f changes). This sequence of different residual networks G_f satisfies:

Theorem

1) The distance between s and t in G_f ***never decreases*****:** After each iteration of the while-loop, it either increases or stays the same.
2) The distance between s and t in G_f can stay the same for at most $|E|$ iterations of the while-loop before increasing.

As the distance between s and t can never be more than $|V|$-1 and it starts out as at least 1, it follows from the theorem that we have at most $(|V|-2)|E|$ iterations.

The Edmonds-Karp Algorithm - example



- The Edmonds-Karp algorithm halts in only 2 iterations on this graph.

Max flow algorithms and their complexity

- Fulkerson Ford $O (E |f^*|)$.. psuedo-poly
- Edmonds Karp $O(V E^2)$.
- Push-relabel $O(V^2 E)$ (CLRS, 26.4)
- The relabel-to-front $O(V^3)$ (CLRS, 26.5)

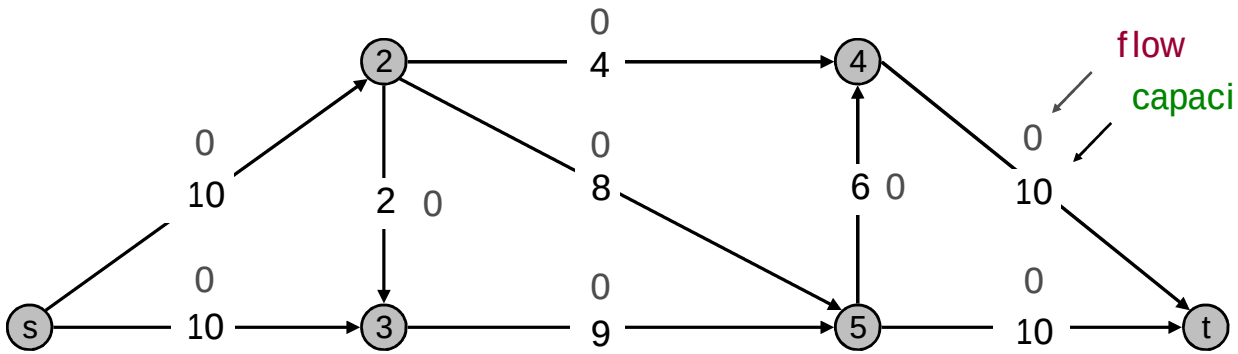
Maxflow Animation

of

Ford-Fulkerson Algorithm

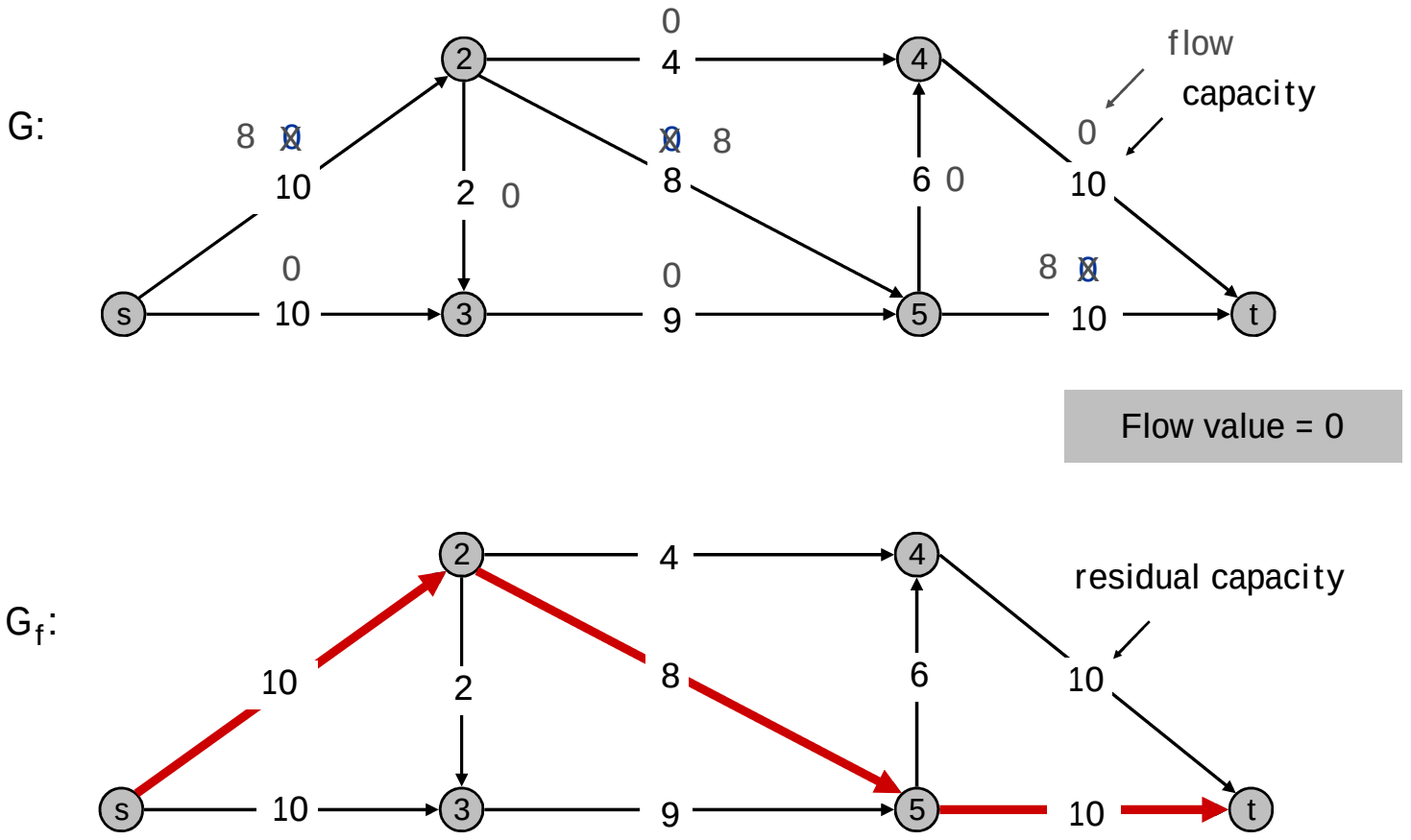
Ford-Fulkerson Algorithm

G:



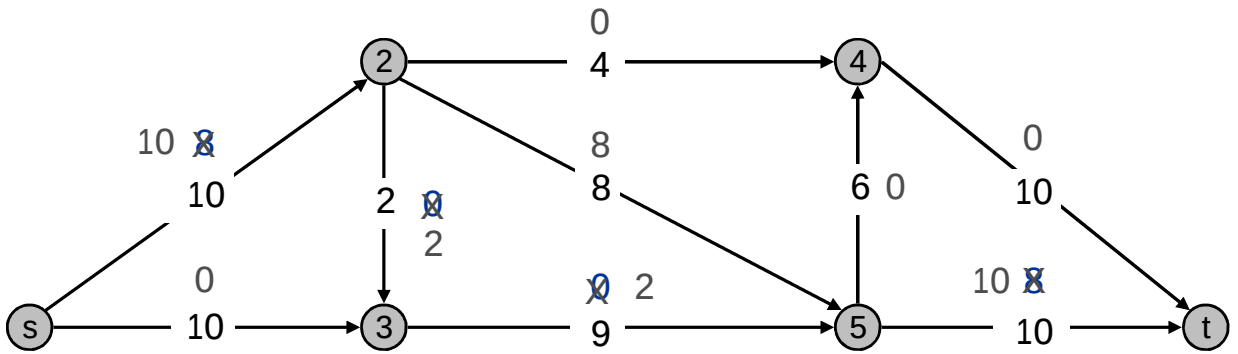
Flow value = 0

Ford-Fulkerson Algorithm



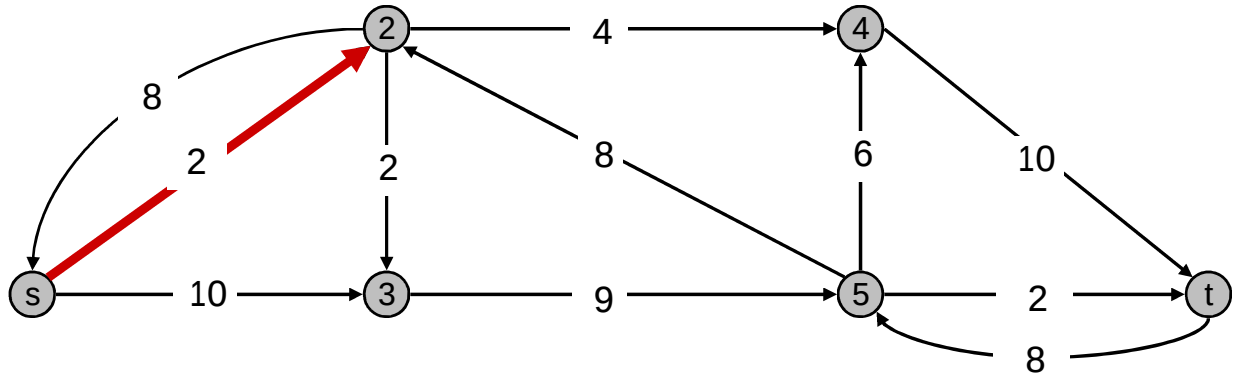
Ford-Fulkerson Algorithm

G:



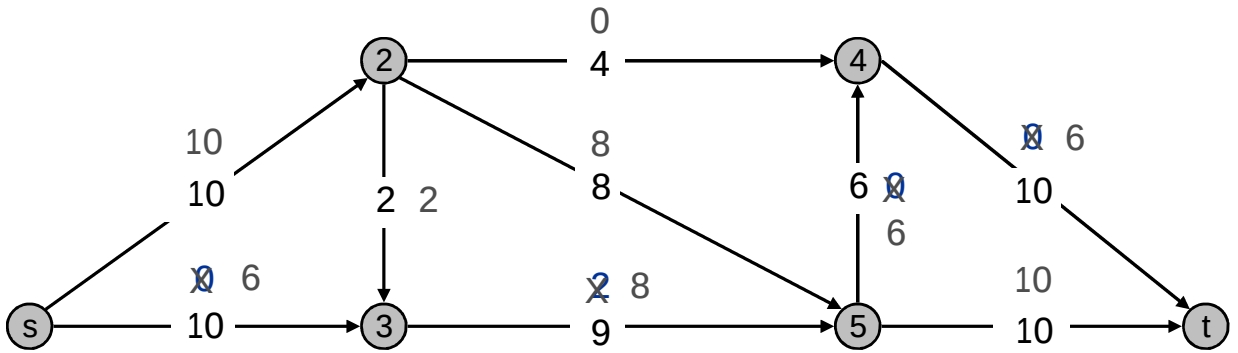
Flow value = 8

G_f:



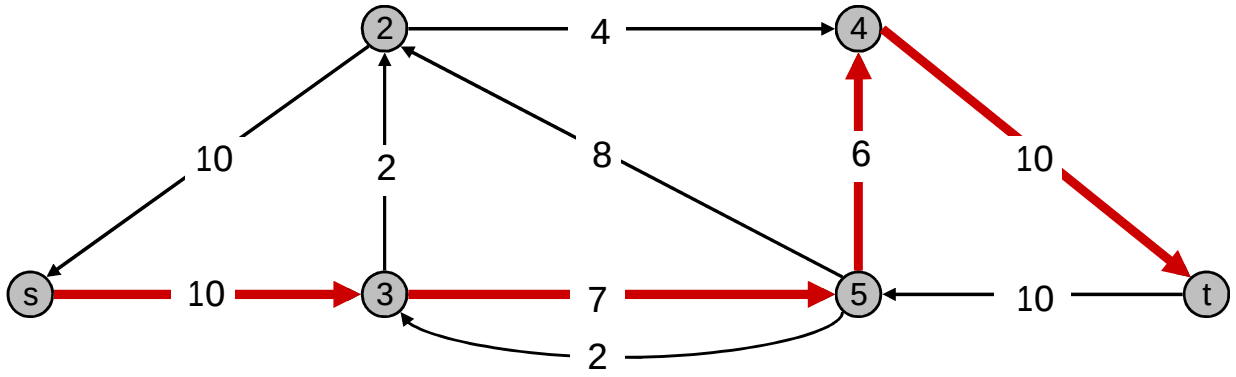
Ford-Fulkerson Algorithm

G:



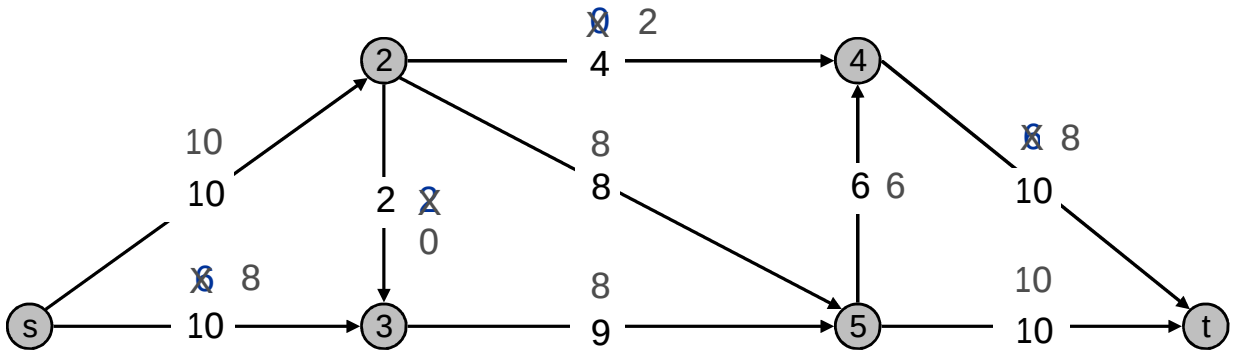
Flow value = 10

G_f:



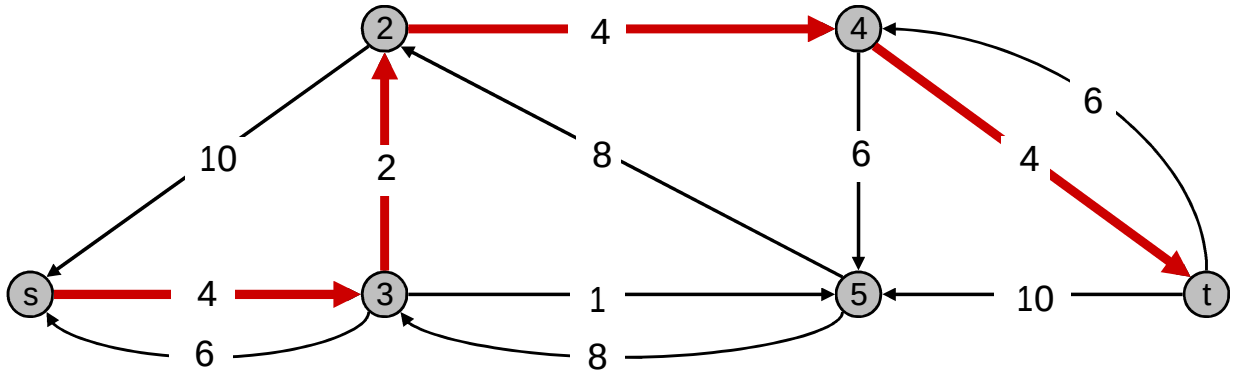
Ford-Fulkerson Algorithm

G:



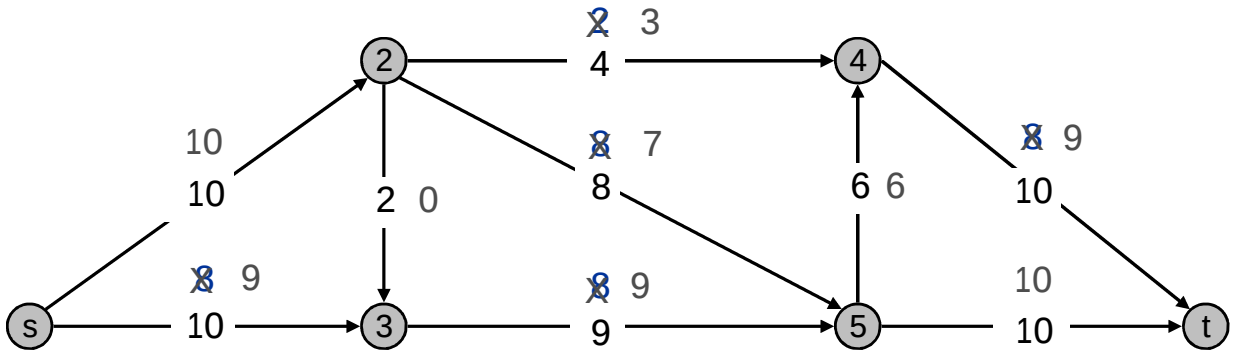
Flow value = 16

G_f:



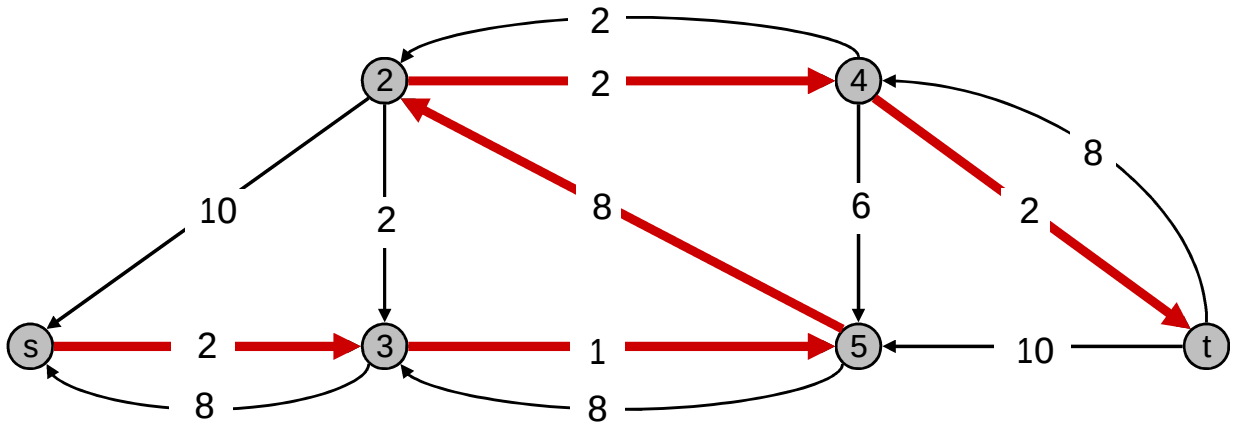
Ford-Fulkerson Algorithm

G:



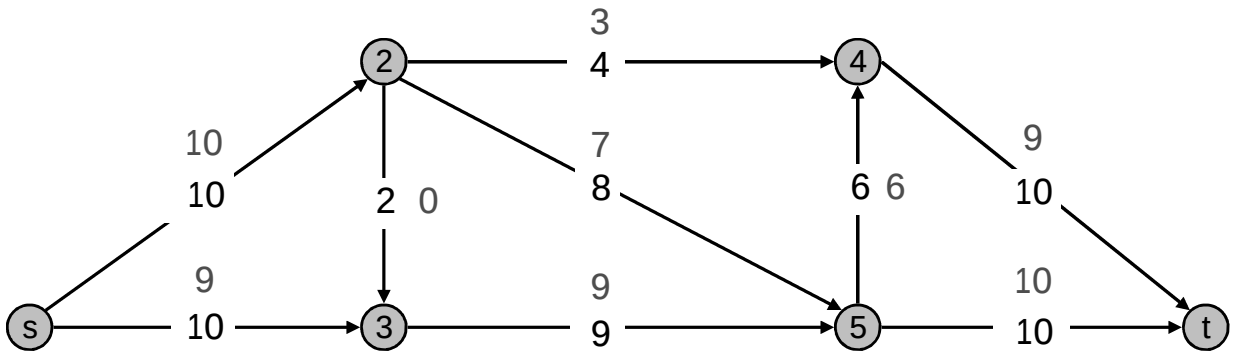
Flow value = 18

G_f:



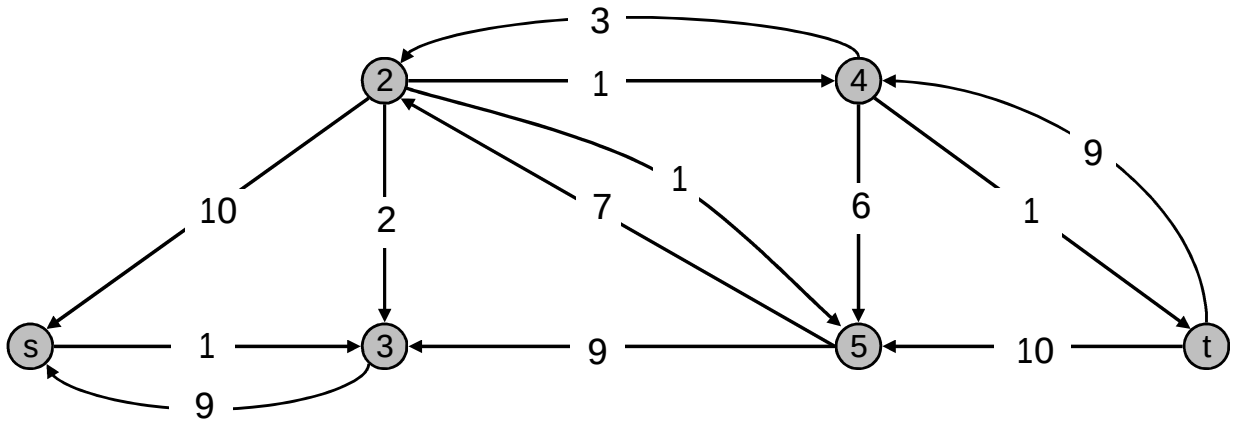
Ford-Fulkerson Algorithm

G:

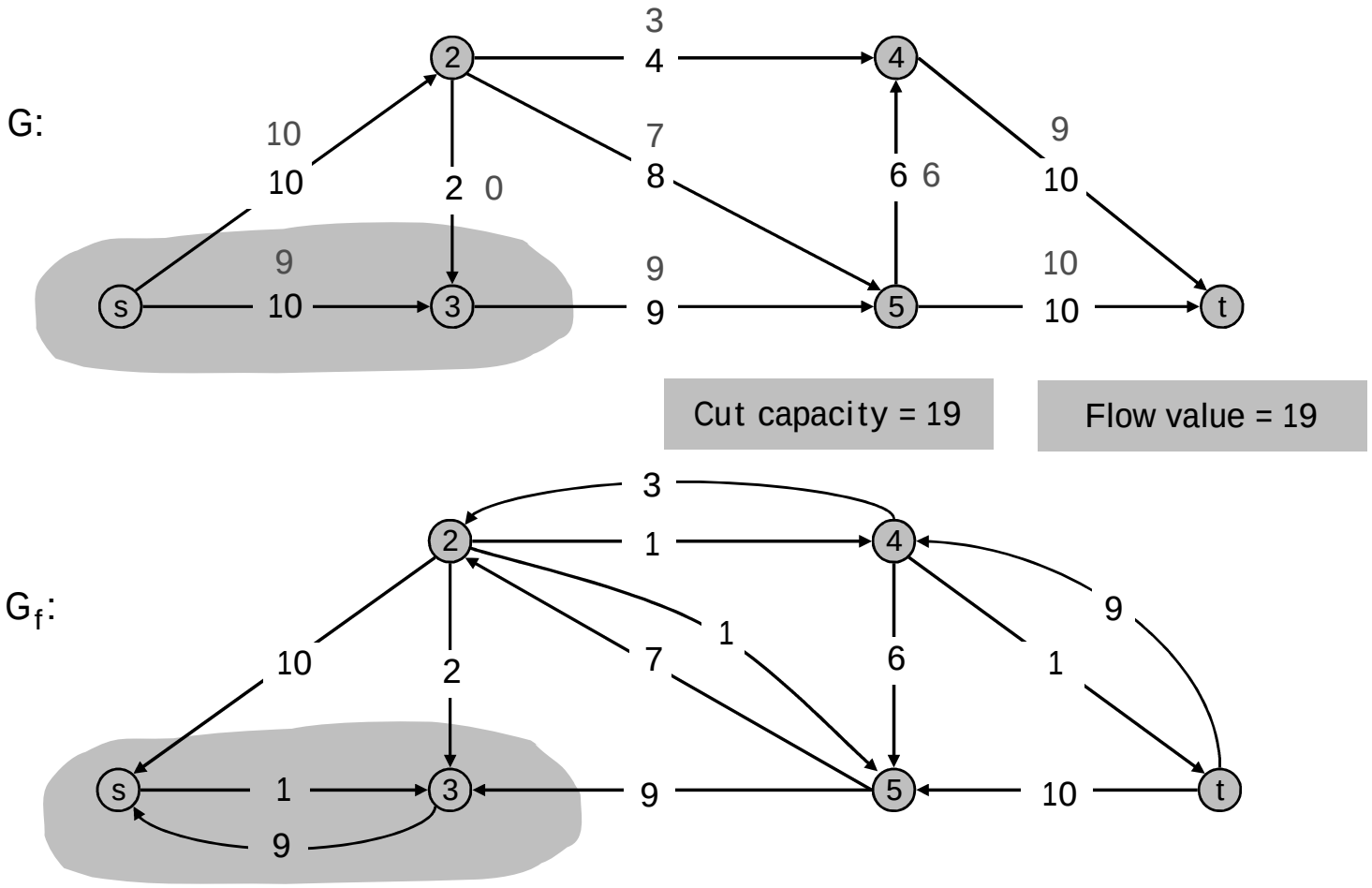


Flow value = 19

G_f:

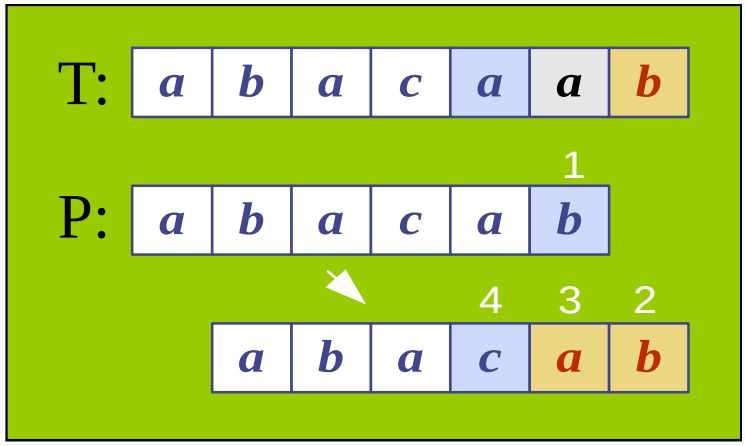


Ford-Fulkerson Algorithm



String Searh and Pattern Matching

(naïve algorithm)



Uses

- Text search using text-editor/word-processor
- Web search
- Packet filtering
- DNA sequence matching
- C program: strstr(P, T)

String Searching

- String searching problem is to find the location of a specific text pattern within a larger body of text (e.g., a sentence, a paragraph, a book, etc.).
- Problem: given a string T and a pattern P, find if and where P occurs in T.
- Example: Find P="n th" in T="the rain in spain stays mainly on the plain"
- We want fast, save space, efficiency.

Substring, prefix, suffix

- Given a string S of size m = S[0..m-1].
- A *substring* S[i .. j] of S is the string fragment between indexes i and j inclusive.
- A *prefix* of S is a substring S[0 .. i]

  w pre x, w ● x, w is a prefix of x,

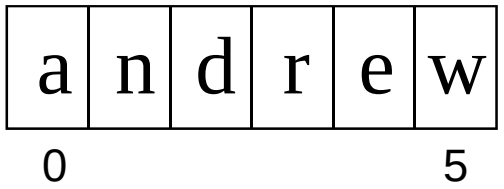
  e.g. aba pre abaz.
- A *suffix* of S is a substring S[i .. m-1]

  w suf x, w ✿ x, w is a suffix of x,

  e.g. abc ✿ zzabc.
- i and j lie between 0 and m-1, inclusive.

Examples of prefixes and suffixes

S =



- Substring S[1..3] is "ndr"
- Prefixes of S are:
  - "andrew", "andre", "andr", "and", "an", "a"
- Suffixes of S are:
  - "andrew", "ndrew", "drew", "rew", "ew", "w"

String matching algorithms

- Naive Algorithm
- RK: Rabin-Karp Algorithm
- String Matching using Finite Automata
- KMP: Knuth-Morris-Pratt Algorithm
- BM: Boyer Moore

Algorithm	Preprocessing Time	Matching Time
Naive	0	$O((n-m+1)m)$
Rabin-Karp	$\Theta(m)$	$O((n-m+1)m)$
Finite Automaton	$O(m\|\Sigma\|)$	$\Theta(n)$
Knuth-Morris-Pratt	$\Theta(m)$	$\Theta(n)$
Boyer-Moore	$\Theta(m)$	$\Theta(n)$

KMP: Knuth-Morris-Pratt Algorithm

- KMP Algorithm has two parts: (1) PREFIX function (2) Matcher.
- Information stored in prefix function to speed up the matcher.
- Running time:

  Creating PREFIX function takes $O(m)$

  MATCHER takes $O(m+n)$ (instead of O(m n)).

Boyer-Moore Algorithm

- Published in 1977
- The longer the pattern is, the faster it works
- Starts from the end of pattern, while KMP starts from the beginning
- Works best for character string, while KMP works best for binary string
- KMP and Boyer-Moore
  - Preprocessing existing patterns
  - Searching patterns in input strings

Automata and Patterns: Regular Expressions

- notation for describing a set of strings, possibly of infinite size
- ε denotes the empty string.
- a + c denotes the set {a, c} (a or c).
- a* denotes the set {ε, a, aa, aaa, ...}, 0 or more a.
- Examples

  (a+b)* all the strings from the alphabet {a,b}

  b*(ab*a)*b* strings with an even number of a's

  (a+b)*sun(a+b)* strings containing the pattern "sun"

  (a+b)(a+b)(a+b)a 4-letter strings ending in a

Alphabet

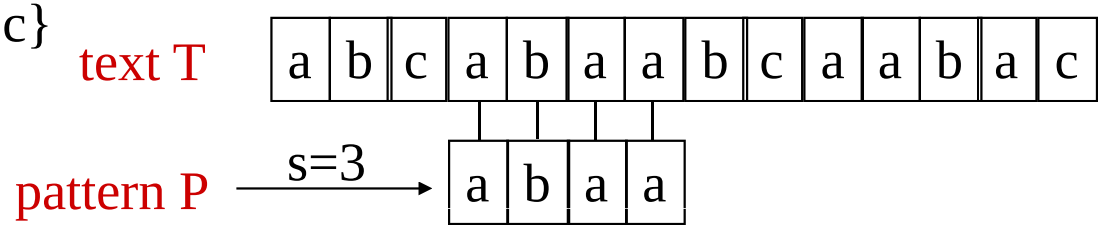
- Elements of *T* and *P* are characters from a *finite alphabet* Σ
- The *pattern* is in an array *P* [1..*m*]
- The *text* is in an array *T* [1..*n*]
- E.g., $\Sigma = \{0,1\}$ or $\Sigma = \{a, b, \ldots, z\}$
- Usually *T* and *P* are called *strings* of characters
- ε (epsilon or lambda is the empty string)

String Matching

Given: Two strings P[1..m] and T[1..n] over alphabet Σ.

Find all occurrences of P[1..m] in T[1..n]

Example: $\Sigma = \{a, b, c\}$



Terminology:

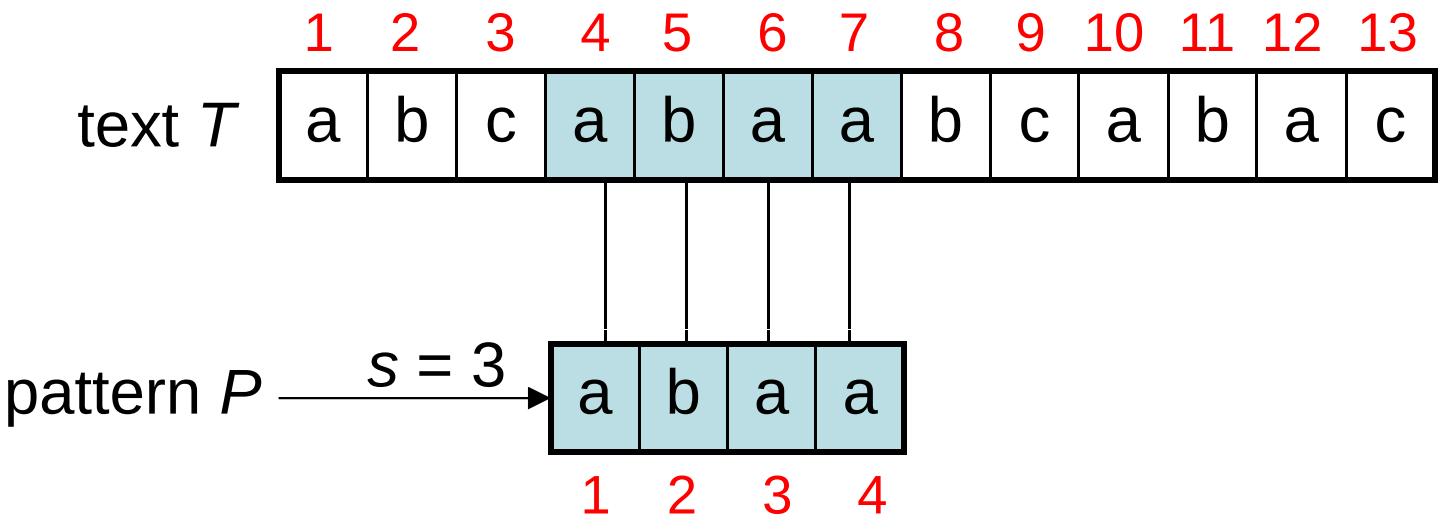
P occurs with shift s in T

P occurs beginning at position s+1.

s is a valid shift.

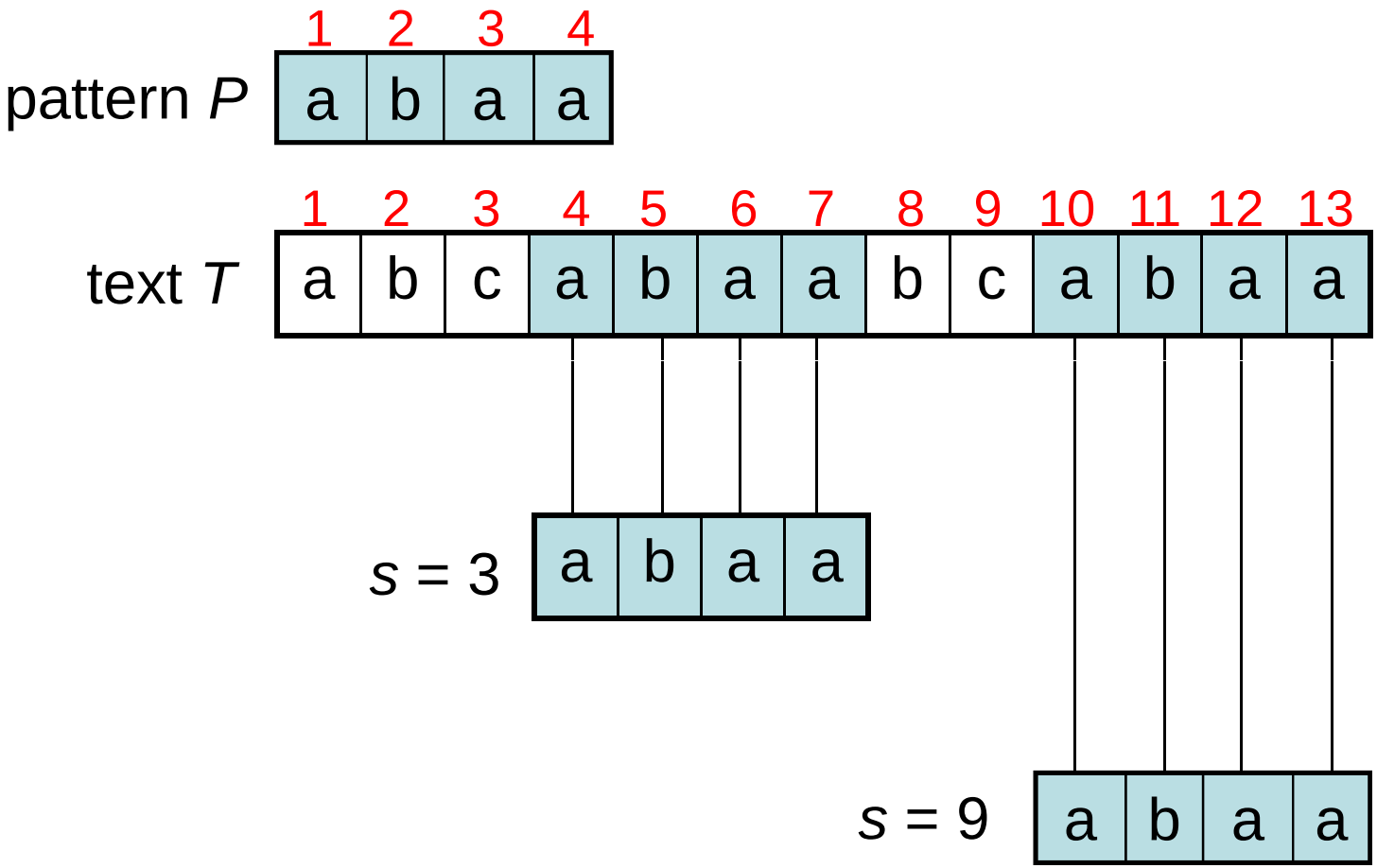
Goal: Find all valid shifts

Example 1, valid shift is a match



shift $s = 3$, is a valid shift
(n=13, m=4 and $0 \leq s \leq n\text{-}m$ holds)

Example 2, another match



Naive Brute-Force String matching

```
Naïve(T, P)
    n := length[T];
    m := length[P];
    for s := 0 to n – m do
        if P[1..m] = T[s+1..s+m] then
            print "pattern occurs with shift s"
    done
```

Running time is $\Theta((n - m + 1)m)$.
Bound is tight. Consider: $T = a^n$, $P = a^m$.

Exercise

Q1. Given p="abab", t="abababa".

How many places does p match with t?

Q2. Given p="xx", t="xxxx".

How many places does this match?

Exercise

- Write the C function to find pattern in text:

`char *sfind(char *pattern, char *text)`

1. to return NULL if p does not occur in t, or
2. to return pointer into t, where p occurs.

- Also read **`strstr`** C-function documentation, and compare strstr wit sfind.

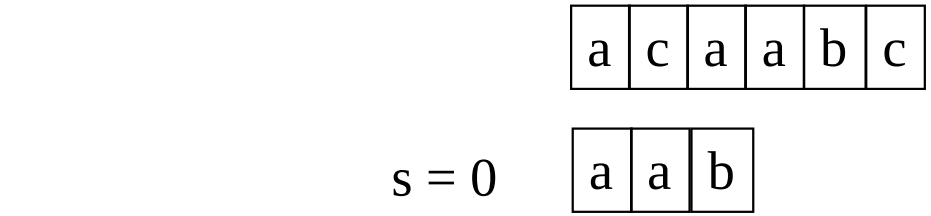
Solution: using sfind

```
1.#include <stdio.h>
2.char *sfind(char *pat, char *text);
3.int main( ) {
4.  char *const pattern = "ab", *text =  "abababa";
5.  char *t = text, *found;   int i=0;
6.  printf("Searching p='%s' in t='%s'\n", pattern,
  t);
7.  while( found = sfind(pattern, t)) {
8.     printf("%2d Found in text[%2d..]=%s\n",
9.        ++i, found-text, found);
10.     t=found+1;
11.  }
12.  return 0;
13.}
```

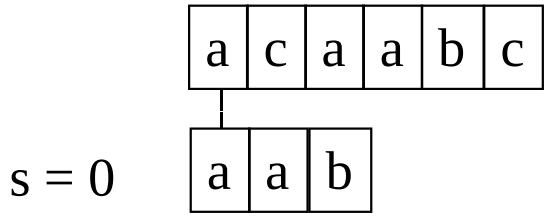
Solution: sfind

```
1.char *sfind(char *pat, char *text) {
2.   char *t = text;
3.   while(*t) {
4.     char *q=pat, *s=t;
5.     while(*q && *s && *q == *s)
6.       q++, s++;
7.     if(!*q)  // reached end of q?
8.        return t;
9.     t++;
10.   }
11.   return 0;
12.}
```

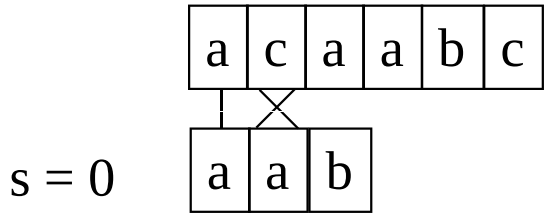
Example



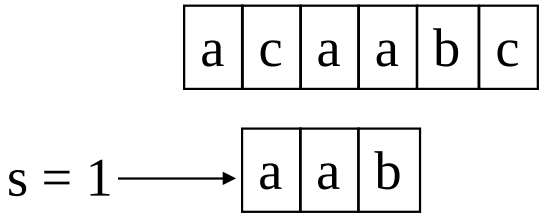
Example



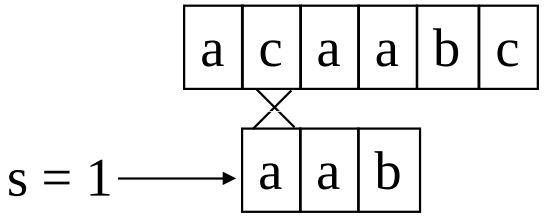
Example



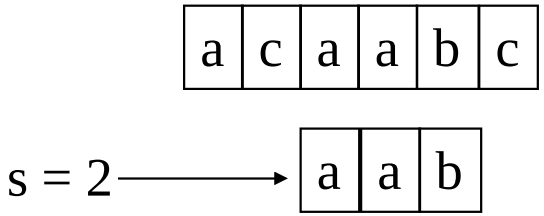
Example



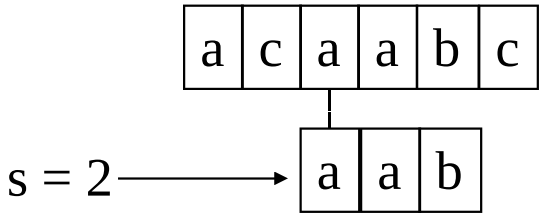
Example



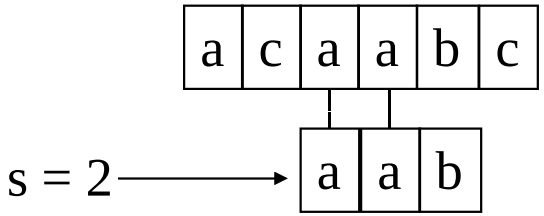
Example



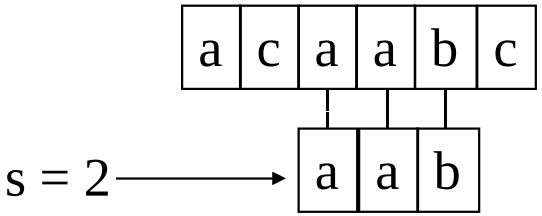
Example



Example

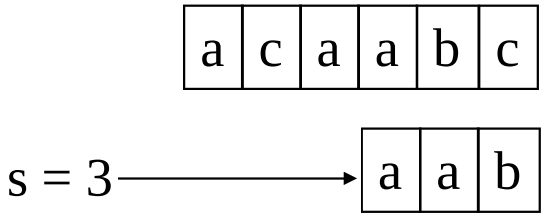


Example

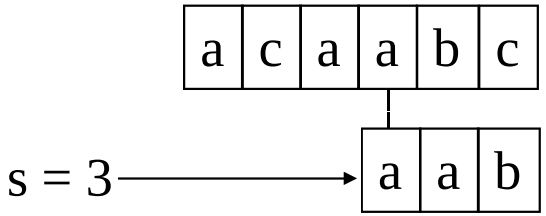


match!

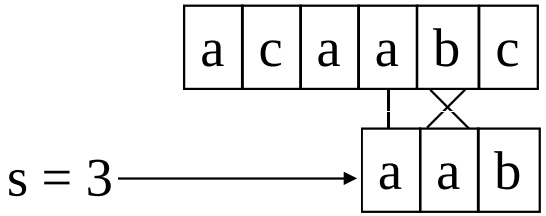
Example



Example



Example



Worst-case Analysis

- Brute force pattern matching can take O(m n) in the worst case.
- But most searches of ordinary text is fast: O(m+n).
- There are *m* comparisons for each shift into *n*-*m*+1 shifts: worst-case time is $\Theta(m(n-m+1))$

Why is naive matching slow?

- Naive method is inefficient because information from a mismatch is not used to jump upto m-steps forward.
- The brute force algorithm is fast when the alphabet of the text is large
  - e.g. {A..Z, a..z, 0..9, ..}
- It is slower when the alphabet is small
  - e.g. binary, image, DNA {G,C,A,T}.

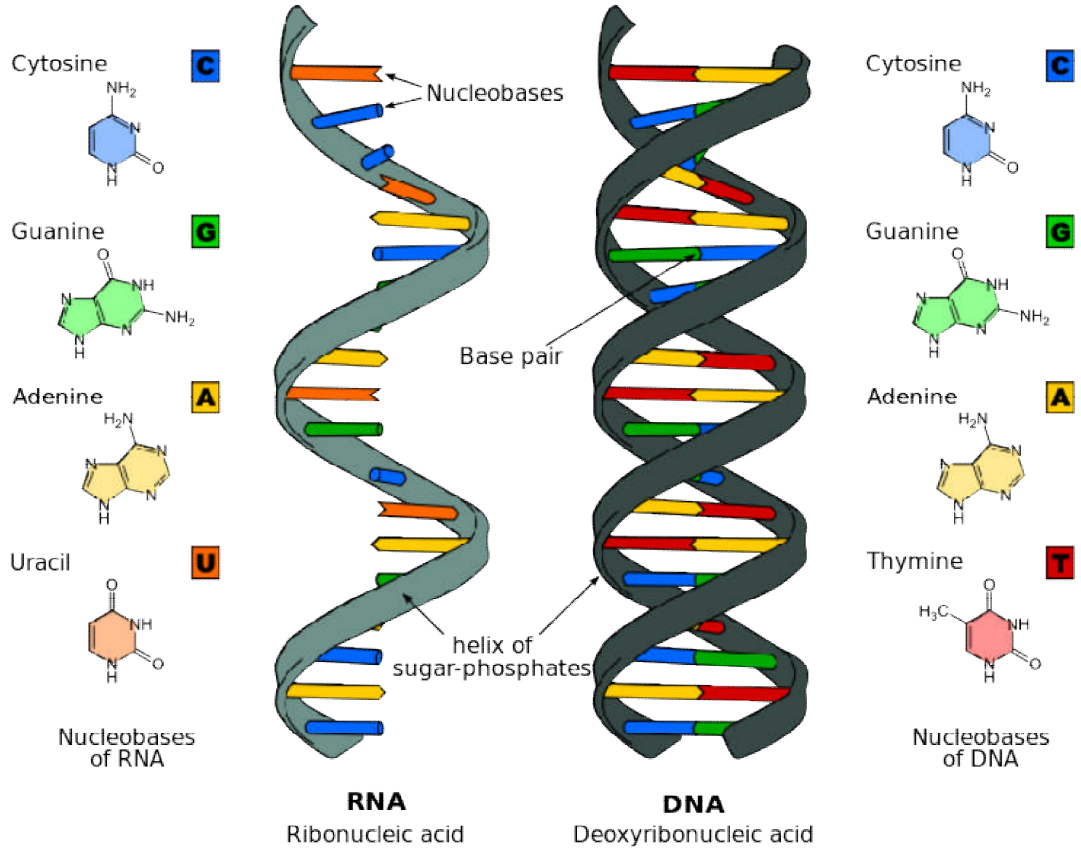
continued

Worst and Average case examples

- Example of a worst case:
  - P: "aaah"
  - T: "aaaaaaaaaaaaaaaaaaaaaaaaaaah"

- Example of a more average case:
  - P: "store"
  - T: "a string searching example is standard"

DNA are strings over {C,G,A,T}
RNA are strings over {C,G,A,U}



DNA Matching

- The main objective of DNA analysis is to get a visual representation of DNA left at the scene of a crime.
- A DNA "picture" features columns of dark-colored parallel bands and is equivalent to a fingerprint lifted from a smooth surface.
- To identify the owner of a DNA sample, the DNA "fingerprint," or profile, must be matched, either to DNA from a suspect or to a DNA profile stored in a database.

Homework: **`lastfind(s,t)`**

Write a C function, lastfind(char *s, char*t) to return the ptr to last occurrence of s in t.

In the main program call lastfind( ) with various inputs, and assert the expected output. E.g.

```
assert(lastfind(s="ab",t="abcabc"), t+3);
assert(lastfind("ab","ba")==NULL);
```

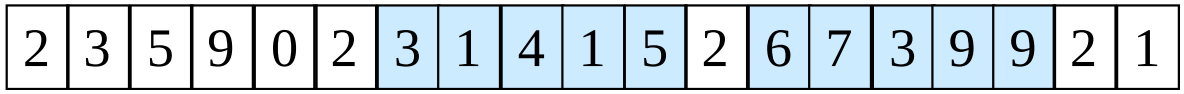
Homework: Write brute force string search function

- Write find2(char * source, char *target) to find the 2nd occurrence of source in target.
- E.g. find2("xy", t="axybxyc") returns t+4
- E.g. find2("x", t="x") return NULL
- E.g. find2("x", t="xx") returns t+1.

Rabin Karp String Matching Algorithm

Rabin-Karp Algorithm

We can treat the alphabet as numbers
Σ = {0, 1, 2, …, 9} = digits instead of characters.
So we can view P as a decimal number.
We can also treat substrings of text as decimal numbers.

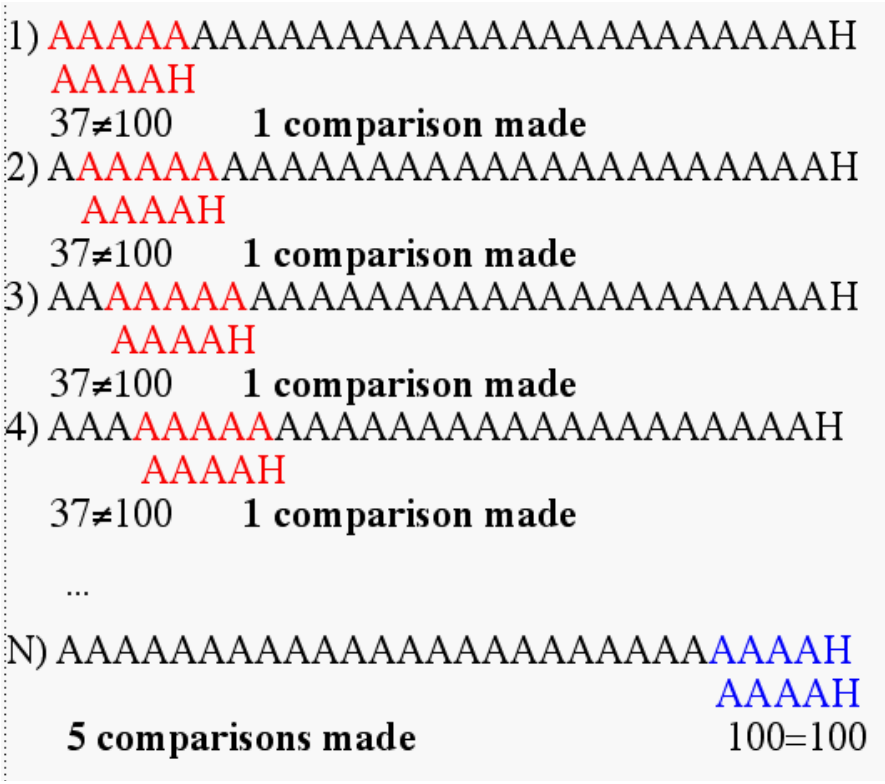


Rabin-Karp

- The Rabin-Karp string searching algorithm calculates a hash value for the pattern, and a hash for each M-character subsequence of text to be compared.
- If the hash values are unequal, the algorithm will calculate the hash value for next M-character sequence.
- Else the hash values are equal, the algorithm will do a Brute Force comparison between the pattern and the M-character sequence.
- In this way, there is only one comparison per text subsequence, and Brute Force is only needed when hash values match.

Rabin-Karp Example

- Hash value of “AAAAA” is 37
- Hash value of “AAAAH” is 100



Modular Equivalence

if *(a mod q) = (b mod q),* we say:

"a is equivalent to b, modulo q" , we write:

$a \equiv b \pmod{q}$, i.e. *a* and *b* have the same remainder when divided by q.

E.g. $23 \equiv 37 \equiv -19 \pmod{7}$.

The mod function (%) has nice properties:

1. $[(x \% q) + (y \% q)] \% q \equiv (x + y) \% q$
2. $[(x \% q) * (y \% q)] \% q \equiv (x * y) \% q$
3. $(x \% q) \% q \equiv x \% q$

Hashing a string

- The alphabet consists of 10 letters = { a, b, c, d, e, f, g, h, i, j }
- Let “a” correspond to 1, “b” to 2 ..
- hash(“cah”) would be:

c*100 + a*10 + h*1

3*100 + 1*10 + 8*1 = 318

Rabin-Karp, incremental mod

We don't compute a new value.

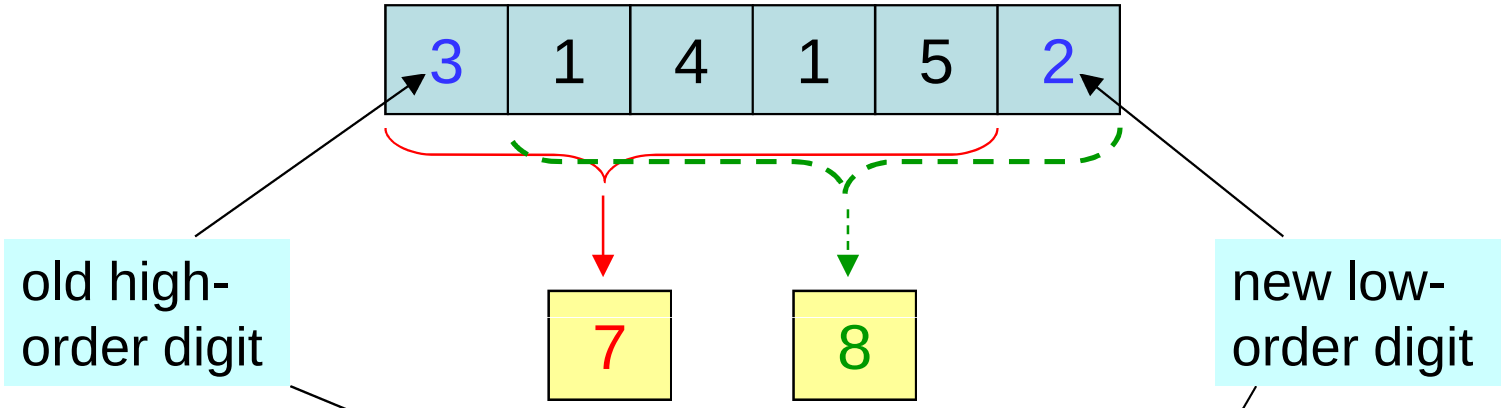
We simply adjust the existing value as move right.

Given x(i), we can compute x(i+1) for the next subsequence t[i+1 .. i+M] in constant time, as follows:

```
h(i)=( (t[i]  * b^M-1    % q)
       + (t[i+1] * b^M-2 % q) + ...
         +(t[i+M-1]       % q))  % q

h(i+1) =( h(i) * b   % q          # Shift left one digit
            - t[i] * b^M % q      # Subtract leftmost digit
             +t[i+M]    % q )     # Add new rightmost digit
                        % q
```

How incremental hash mod 13 is computed

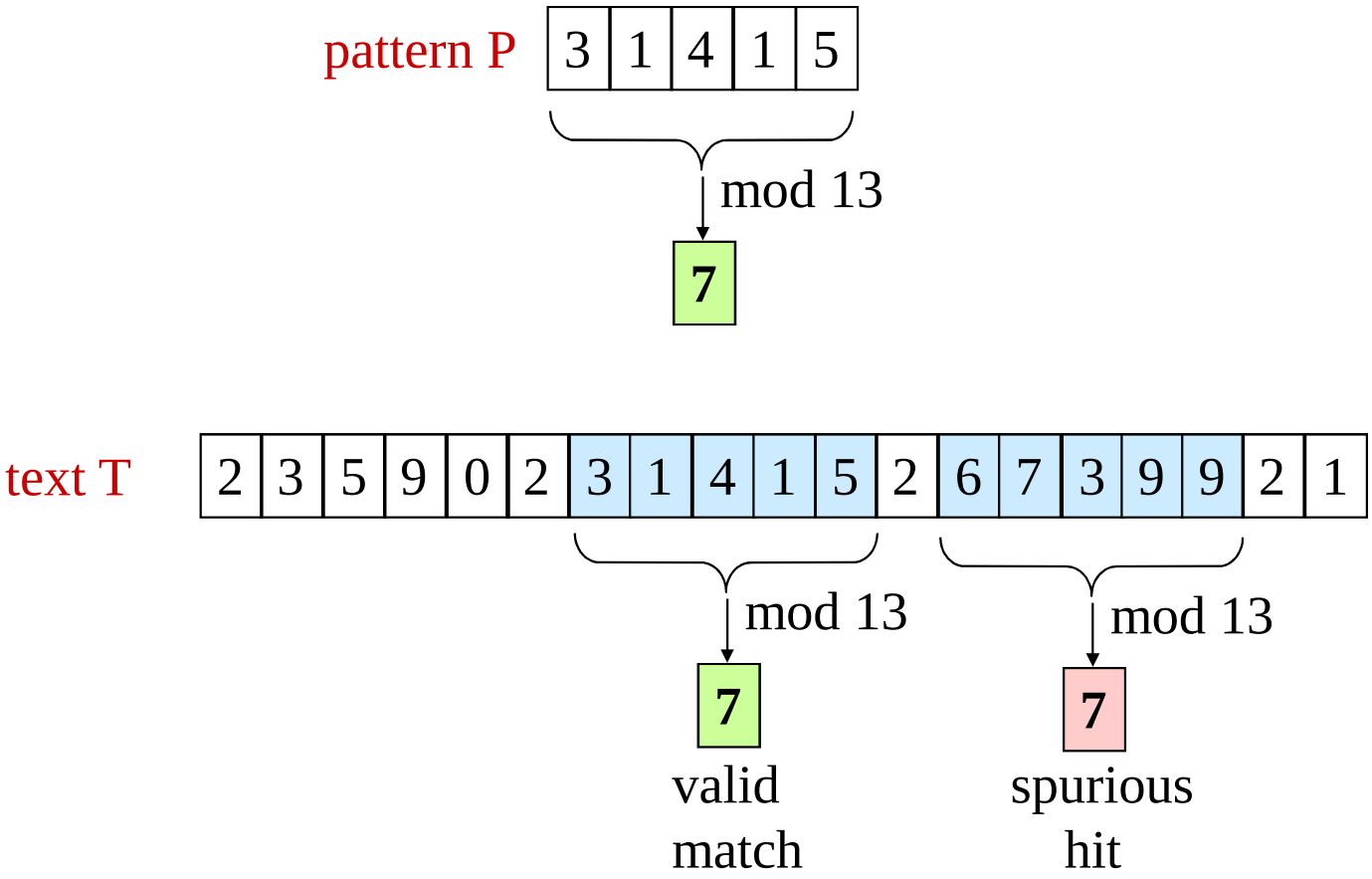


$$14152 \equiv (31415 - 3 \cdot 10000) \cdot 10 + 2 \pmod{13}$$

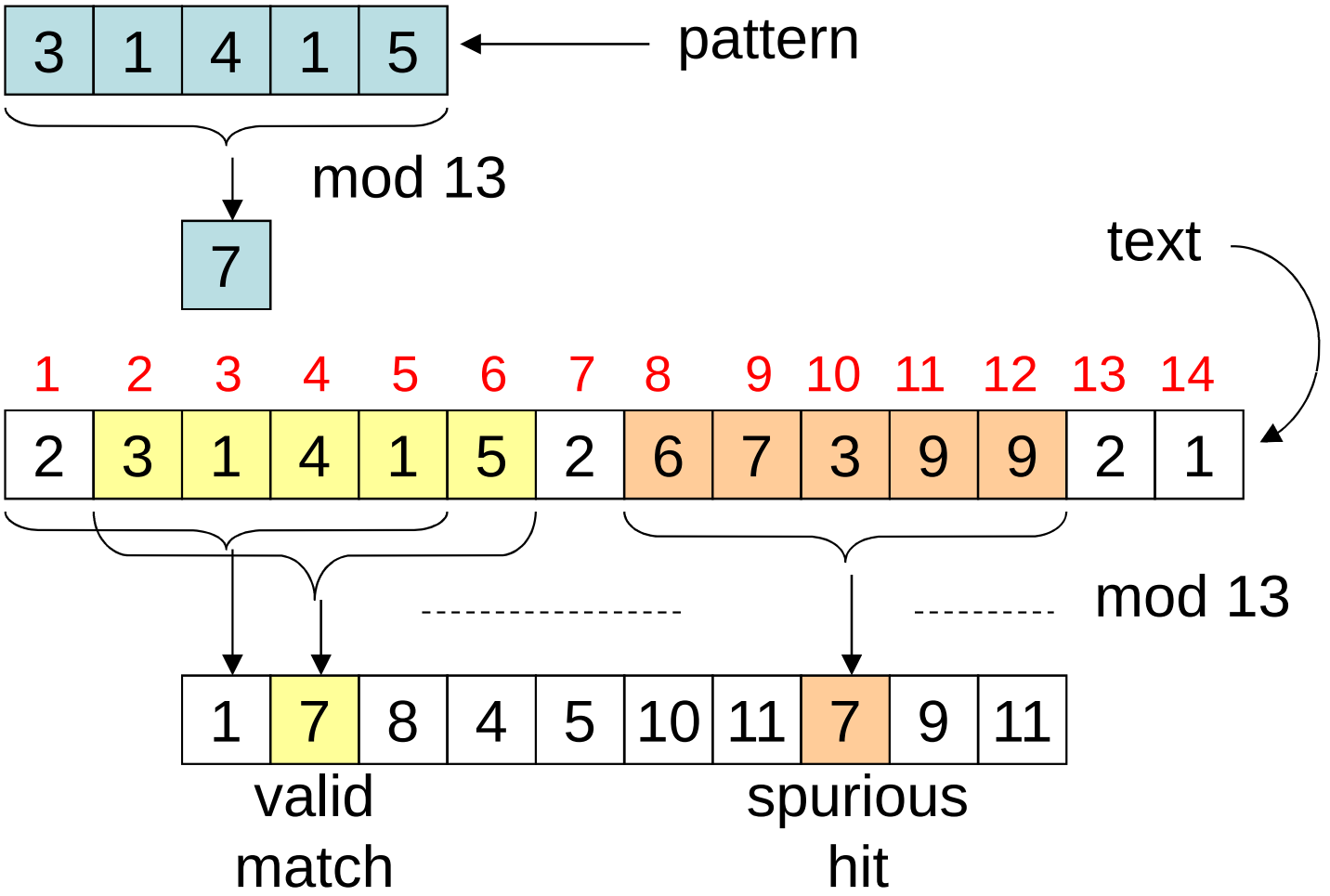
$$\equiv (7 - 3 \cdot 3) \cdot 10 + 2 \pmod{13}$$

$$\equiv 8 \pmod{13}$$

Spurious match



Example


3 1 4 1 5
pattern
mod 13
7
text
1 2 3 4 5 6 7 8 9 10 11 12 13 14
2 3 1 4 1 5 2 6 7 3 9 9 2 1
mod 13
1 7 8 4 5 10 11 7 9 11
valid match
spurious hit

Rabin Karp Algorithm

•RK: Rabin Karp
T: Text
P: Pattern
d: power, e.g. 10
h: 10^m-1, e.g. 10000
q: modulus, e.g. 13

•Compute initial hashes

•Avoid spurious hits by explicit check whenever there is a potential match.

•Compute incremental hash as we go right

```
RK(T, P, d, q)
      n := length[T];
      m := length[P];
      h := d^(m-1) mod q;
      p := 0;
      t0 := 0;
      for i := 1 to m do
            p := (dp + P[i]) mod q;
            t0 := (dt0 + P[i]) mod q
      od;
      for s := 0 to n – m do
            if p = ts then
                  if P[1..m] = T[s+1..s+m] then
                        print "pattern occurs with shift s"
                  fi
            fi;
            if s < n-m then
                  ts+1 := (d(ts – T[s+1]h) + T[s+m+1]) mod q
            fi
      od
```

Rabin-Karp Complexity

- O (*n* * *m*) in Worst case, if every shift has to be verified.
- O (*n* + *m*) in Average case, using a large prime number in *hash function*, less collisions.

where

- n = size(Text),
- m = size(Pattern).

Homework: Write **Rabin Karp String Search**

Write a C function **`rkfind(s,t)`** to search string s in t.

Let hash(str) be the sum of chars in str mod 256.

In the main function, check these:

```
assert(rkfind(s="ab", t="xabc") == t+1);
assert(rkfind("ab", "baa")==NULL);
```

Homework: Rabin Karp String search

- Implement Rabin karp string search in C.
- Use xor(chars of the string) as the hash function.